Expat: Expectation-based Policy Analysis and Enforcement for Appified Smart-Home Platforms

Moosa Yahyazadeh
The University of Iowa
moosa-yahyazadeh@uiowa.edu

Proyash Podder
Florida International University
ppodder@fiu.edu

Endadul Hoque
Florida International University
ehoque@fiu.edu

Omar Chowdhury
The University of Iowa
omar-chowdhury@uiowa.edu

ABSTRACT

This paper focuses on developing a security mechanism geared towards appified smart-home platforms. Such platforms often expose programming interfaces for developing automation apps that mechanize different tasks among smart sensors and actuators (*e.g.*, automatically turning on the AC when the room temperature is above 80°F). Due to the lack of effective access control mechanisms, these automation apps can not only have unrestricted access to the user's sensitive information (*e.g.*, the user is not at home) but also violate user expectations by performing undesired actions. As users often obtain these apps from *unvetted* sources, a malicious app can wreak havoc on a smart-home system by either violating the user's security and privacy, or creating safety hazards (*e.g.*, turning on the oven when no one is at home). To mitigate such threats, we propose Expat which ensures that user expectations are never violated by the installed automation apps at runtime. To achieve this goal, Expat provides a *platform-agnostic*, *formal* specification language Uei for capturing user expectations of the installed automation apps' behavior. For effective authoring of these expectations (as policies) in Uei, Expat also allows a user to check the desired properties (*e.g.*, consistency, entailment) of them; which due to their formal semantics can be easily discharged by an SMT solver. Expat then enforces Uei policies *in situ* with an inline reference monitor which can be realized using the same app programming interface exposed by the underlying platform. We instantiate Expat for one of the representative platforms, OpenHAB, and demonstrate it can effectively mitigate a wide array of threats by enforcing user expectations while incurring only modest performance overhead.

CCS CONCEPTS

• **Security and privacy → Formal security models**; **Access control**; **Malware and its mitigation**.

ACM ISBN 978-1-4503-6753-0/19/06…$15.00
https://doi.org/10.1145/3322431.3325107

KEYWORDS

Appified smart-home platforms; IoT security; policy enforcement; inline reference monitoring

ACM Reference Format:
Moosa Yahyazadeh, Proyash Podder, Endadul Hoque, and Omar Chowdhury. 2019. Expat: Expectation-based Policy Analysis and Enforcement for Appified Smart-Home Platforms. In *The 24th ACM Symposium on Access Control Models and Technologies (SACMAT '19), June 3–6, 2019, Toronto, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3322431.3325107

1 INTRODUCTION

In recent years, the Internet-of-Things (IoT) has undergone drastic innovations, from merely a network of sensors and actuators to a complex ecosystem consisting of a wide range of smart devices (*e.g.*, lights, cameras) additionally equipped with automation applications (also known as *apps*/*rules*),[1] and cloud-based backend services. *Smart-home platforms* (*e.g.*, SmartThings [34], OpenHAB [29], Apple HomeKit [20]) are nothing but instances of such complex *appified* IoT ecosystems. While such platforms use cloud-based backend services to enable remote monitoring and control, they often expose programming interfaces for developing flexible and customized automation apps as deemed desirable by the users. However, these platforms lack effective access control mechanisms, and as a result, automation apps can exercise *unrestricted* access to user's sensitive information (*e.g.*, the user is not at home) and misuse smart devices (*e.g.*, smartlock) [35]. Therefore, automation apps, often obtained from *unvetted* sources (*e.g.*, community-driven and/or third-party marketplaces), are lucrative targets for an adversary. Installing malicious apps can not only compromise the *security* and *privacy* but also sabotage the *safety* of a smart-home. For instance, a malicious app controlling the smartlock at the front door can result in break-ins, theft, or even physical injury.

Automation apps predominantly follow the *trigger-action* paradigm, where an app reacts to a *trigger* (*e.g.*, when the user leaves home) by commanding an *action* (*e.g.*, lock the front door). Sometimes an app can send multiple actions to the same device and/or different devices. While a trigger can activate multiple apps simultaneously, an action can also trigger other apps directly or indirectly. Since a typical smart-home is equipped with many such apps, the design principle of this paradigm can potentially lead to inconceivable consequences due to subtle interplay among apps. For instance, two apps sending *conflicting* actions to a smart device, or two or

[1] The terms *app* and *rule* are interchangeably used in this paper.

more apps *negating* each other's actions. An adversary can take advantage of this loophole. To perform an undesired action (*e.g.*, unlock the front door when the user is not at home), an adversary can directly embed a sneaky command into its app or develop an app to exploit such subtle interplay. Apart from malicious apps, such misbehavior can also manifest due to programming mistakes in *benign* apps. Existing smart-home platforms provide no built-in defense mechanism against these threats. Therefore, *this paper focuses on developing a security mechanism that mediates the behavior of such automation apps through the enforcement of user expectations.*

Existing efforts. The majority of the existing efforts [7, 10, 13] focus on developing static analysis-based approaches that try to identify violation of user expectations prior to app installation. Although such pre-deployment analysis approaches do not incur any overhead at runtime, they all suffer from the following two limitations: (1) These approaches are inherently prone to imprecision due to the underlying static (over- or under-approximation) analysis; (2) When a violation is observed during analysis, these approaches do not prescribe any solution to the inexperienced user to mitigate the issue, rendering the combination of apps completely unusable. To the best of our knowledge, the only system that aims to provide runtime protection is IoTGuard [8]. IoTGuard, however, outsources all the relevant internal information of the system to an *off-site* for conducting runtime checking, raising a major privacy concern.

Our approach. In this paper, we propose Expat which ensures that user expectations are never violated by the installed automation apps at runtime. Expat provides a specification language called Uei in which the user can express one or more policies, each consisting of the invariants that they desire apps to comply with at runtime. An example of the expected invariant could be: "*I expect the front door gets unlocked only if the vacation mode is turned off*". Any automation app that contemplates on opening the front door of the house will be ultimately blocked by Expat at runtime when the vacation mode is turned on. Expat hinges on runtime analysis because of its precision and its capability of concentrating on a specific execution of an app, which is essential in this context. In addition, Expat adopts this policy-driven approach to decouple user-defined expectations from the automation apps.

The Uei language used in Expat is designed in a general and extensible fashion so that they can be adopted for a wide-variety of platforms. To achieve generality and extensibility, we leave several aspects of the language abstract and open which we expect will be appropriately instantiated in the context of a target platform. Currently, the native Uei policy vocabularies (e.g., current time, states of smart devices) are chosen after investigating the concepts of existing appified smart-home platforms [29, 34].

The Uei policy semantics, on the other hand, are devised in a way so that they can be directly translated to quantifier-free first order logic formulae with appropriate theories (e.g., linear real arithmetic, strings). This enables us to support different policy analysis tasks (*e.g.*, consistency) by leveraging an SMT solver [36]. To analyze if the user-defined UEI policies accurately capture user expectations, we have designed a meta-level policy analysis language, called Pal, in which one can express more general policy analysis tasks as logical formulae for policies expressed in Uei. Expat provides a compiler that automatically compiles down the appropriate Uei policies and its analysis task expressed in Pal to SMT-LIB language [5] which can then be discharged by an SMT solver.

Although Uei policies are platform-agnostic, their enforcement mechanisms are platform-specific. To demonstrate the feasibility and generality of Expat, we instantiate its enforcement engine for the OpenHAB platform – a representative open-source smart-home platform – by developing an *in situ*, inline reference monitor. For enforcing Uei policies, we use the app execution engine of the platform. It may appear that one can utilize the programming interface to implement the policy checking and enforcement mechanism as a separate standalone automation app. However, this straightforward design is not effective because (i) apps execute in isolation, that is, one app cannot access information/functionality of the other app; (ii) each app requires to access the policy checking functionality; and (iii) concurrent app executions can result in inconsistencies during policy checking.

Instead, in our design for OpenHAB, we have developed the policy checker as a utility script which can be accessed by all apps. This script, however, does not directly solve the concurrency issues. Therefore, we additionally employ built-in synchronization primitives supported by the domain-specific language (DSL) designed for OpenHAB apps. Once the policy checking script is synthesized *automatically* from a given Uei policy, we then make apps amenable to policy checking by instrumenting them. We have developed an automated app-instrumentation approach that guards each (sensitive) action performed by an app with a call to the policy checker. The action contemplated by the app is allowed only if the policy checker script suggests compliance with the given Uei policy.

Empirical evaluation. We evaluated Expat using our own testbed akin to a smart-home equipped with 18 different smart IoT devices. We installed 15 automation apps/rules and 8 user-defined polices. We created 8 scenarios capturing different types of undesirable situations that could occur due to subtle interplay between apps and malicious apps. Our experiments demonstrated how Expat was able to block undesirable actions violating the user's expectations while incurring a very low overhead (i.e., ~63 ms).

Contributions. To summarize, the paper makes the following technical contributions:

1. **In-situ deployment:** For policy enforcement, Expat does not require access to platform's backend. It only leverages platform's capability of executing an app and its programming interface. Everything including Policy Decision Point (PDP) and Policy Enforcement Point (PEP) remain inside the platform. Control or data *never* leave the platform, enabling Expat to avoid any privacy or performance concerns. Expat is agnostic to whether the platform operates in a local server (*e.g.* OpenHAB) or a remote-cloud (*e.g.* SmartThings). As long as the platform provides a programming environment that can run apps, Expat is general enough to be deployable.
2. **Policy language:** We present a platform-agnostic, general specification language Uei with its precise semantics, which can precisely and in a fine-grained fashion capture the user expectations from the behavior of a set of installed automation apps. Uei can also be easily adopted for expressing fine-grained, contextual access control policies for smart-home platforms in which support for such policies are inadequate.

(3) **Policy analysis:** For effective Uei policy authoring, we designed a language Pal in which users can express policy analysis tasks to be carried out on Uei policies. These tasks can then be discharged with the help of an SMT solver.
(4) **Instantiation of Expat:** We demonstrate the generality and feasibility of Expat by instantiating it for OpenHAB. We also demonstrate Expat's effectiveness through several case studies on OpenHAB. In our evaluation, we observed that Expat can effectively thwart malicious behavior from apps while incurring a small latency overhead.

2 PRELIMINARIES

As IoT devices establish more embedded connectivities and become more prevalent, vendors are striving to make them easier to use and compatible with different home automation systems. These automation systems range from home assistants (*e.g.*, Amazon Alexa [2], Google Home [17], Apple HomePod [4]) mainly used to voice-control the smart devices to appified smart-home platforms (*e.g.*, Samsung SmartThings [34], OpenHAB [29], Apple HomeKit [20]) facilitating the automation and interoperability between them.

The appified smart-home platforms are specifically devised to bring a seamless automation precesses among the smart devices at home using which a user is envisioned to have a minimum intervention in the operation of the system. They are designed to be as simple as possible in order to be adopted by a wide range of home-users, while being powerful enough to handle complex automation scenarios expressed by superusers/app developers. These platforms might have different architectures, residing on a propitiatory cloud or living on a local server on the user side; however, in the end, they all provide a programming interface for users to develop the automation tasks. For instance, Samsung SmartThings provides a cloud based architecture in which smart devices can be managed directly by SmartApps (*i.e.* Groovy-based automation apps) on the cloud or through a connected Hub [34]. User can also participate in controlling the devices directly though a companion SmartThings mobile app. OpenHAB on the other hand, provides both cloud-based and local-server architecture in which users can write some automation rules in a domain-specific language to establish some interactions among the smart devices.

The programming interface provided by such platforms generally follows the *trigger-action* paradigm. In the trigger-action programming, the user requires to specify *(i)* a trigger, the condition or event under which the system should do something, and *(ii)* action(s), which is a command sent to another device or a particular function accomplishing a task.

3 OVERVIEW OF EXPAT

We start this section by describing our threat model. We then present the problem Expat aims at mitigating. Finally, we present the high-level functionality of Expat and how it can be leveraged by a user to protect her appified smart-home system against relevant threats from malicious or misbehaving automation apps.

3.1 Threat Model

In our threat model, we assume automation apps for smart-home platforms, obtained possibly from unvetted sources, can be malicious. In this threat model, a single malicious app can carry out some undesired behavior; possibly under very specific conditions (e.g., logic bombs). In a more complex scenario, the adversary can hide its true malicious intent in a series of apps which may coordinate among themselves to exhibit an undesired behavior. We want to emphasize that our threat model also allows an undesirable behavior occurring benignly due to design or implementation flaws in apps. Finally, we consider adversaries' ability to compromise the smart-home platform itself, due to underlying platform's network, system, or software vulnerabilities, to be out of scope of this paper.

3.2 Problem Definition

Expat aims at preventing installed automation apps in a smart-home platform to carry out some (malicious) actions that violate the user's intended expectation.

An unintended action could be triggered purely due to the user's lack of understanding of some apps' behavior, or because of the app developers' malicious intent. A possible way a malicious app can sneak in a user smart-home is when the mischievous developer advertises an app with a lucrative mechanization functionality which also sneakily performs some other malicious action(s). For instance, let us consider an app that advertises the functionality of switching off all the lights in the house whenever the user turns on the night mode. The malicious developer, however, also sneaked in other unadvertised actions including one that unlocks the front door of the house at that time. As many platforms allow apps to access web services, it is plausible that the app could communicate back to the developer with an approximate geo-location of the house before opening the door.

Another possible way an unintended action could arise when the user installs apps that interact with each other in inconceivable ways leaving the system in a hazardous state. Suppose there are two automation apps app_1 and app_2 providing some home safety features. app_1 aims at protecting the user from fire and thus turns on the sprinkler whenever the house temperature is over 135°F and it senses smoke. app_2, on the other hand, protects the user from damages due to water leak so whenever it detects a water leak with one of its sensors then it turns off the main water valve. In case of a fire, app_1 will switch on the sprinklers. After the sprinklers switch on, app_2 can get triggered—due to sensing of water—closing the main water valve and cutting off water from the sprinklers. As a result, the house can get damaged, possibly, jeopardizing the life of pets. In general, such unconceivable actions triggered by apps can result in unauthorized access, physical harms, financial loss, or any other undesirable situations. Expat aims at mitigating such threats induced by the installed (malicious) automation apps.

3.3 Expat Workflow

In this section, we briefly discuss the high-level architecture of Expat (EXpectation-based Policy Analysis and enforcemenT) and its intended usage. Figure 1 depicts a typical workflow of Expat.

According to the figure, a user of the appified smart-home platform browses through the app store or developer community forum

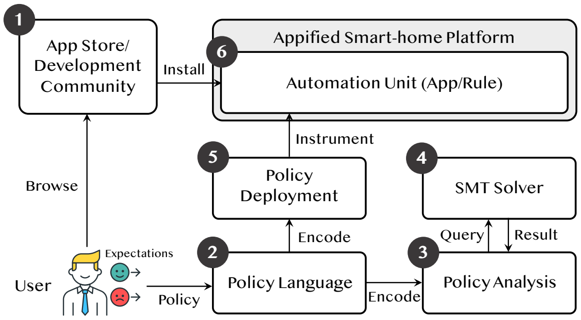

Figure 1: Expat workflow

and select some apps to download and finally installs them on the platform (step ❶). Given that those apps usually are from *unvetted* sources, malicious motives are likely to be encoded in the apps which could be baffling for the regular users to pinpoint. Hence, the user specifies their expectations of the smart-home system in the presence of those apps using a high-level policy language Uei (step ❷). Having contemplated that the user policies have been fed to the system, Expat can analyze the provided policies (*e.g.*, for consistency check) to make sure they are sound and well established. Policy analysis is performed by encoding policies as an SMT problem (step ❸) and then consulting with an SMT-solver (step ❹). After verifying that policies are aligned with the user expectations and there is no conflict in it, Expat deploys the policies on a target smart-home platform by instrumenting those apps (step ❺) such that the policies can be automatically enforced *in situ* before any action is executed (step ❻), as we discuss them in Section 4. Hence, given the well-defined policies, Expat ensures no action is taken that violates user's expectations by enforcing them at runtime. Installation of new apps would restart the workflow.

4 EXPECTATION-BASED POLICY LANGUAGE, ANALYSIS, AND DEPLOYMENT

We start off this section with an abstract model of an appified smart-home platform. We then present the syntax and formal semantics of Expat's expectation-based policy language, Uei (short for, *user-expectation invariants*), and then discuss the aspects of Expat's policy analysis and deployment in a smart-home platform.

4.1 Abstract Smart-home Model with Expat

We present an abstract model of an appified smart-home platform and use it to explain the enforcement of Uei policies at a high-level. In our context, a smart-home $\mathcal{H}$ can be viewed as an labeled transition system (LTS) of the following form: $\langle \mathcal{S}, \mathcal{A}, \mathcal{V}, \mathcal{R} \rangle$.

The $\mathcal{S}$ component of $\mathcal{H}$ represents a non-empty, possibly infinite, set of states. Note that, in the definition of $\mathcal{H}$, we do not explicitly include a designated set of initial states intentionally to allow the system to start at any state. $\mathcal{A}$ in $\mathcal{H}$, on the other hand, represents the non-empty set of possible actions (e.g., turning on the light) recognized by $\mathcal{H}$. For generality, we intentionally leave the structure of an action to be abstract. One can envision the action to be a mapping of variables to values (of appropriate type). For instance, $a \in \mathcal{A}$ can be a tuple of the following form: $\langle$requesting_app $\mapsto$ app_1, action_device $\mapsto$ $smartLock_1$, action_command $\mapsto$ $unlock\rangle$.

$\mathcal{V}$ represents an arbitrary but finite set of *typed* variables. $\mathcal{V}$ can be decomposed into two mutually exclusive set of variables $\mathcal{V}_e$ and $\mathcal{V}_s$, that is, $\mathcal{V} = \mathcal{V}_e \cup \mathcal{V}_s$ and $\mathcal{V}_e \cap \mathcal{V}_s = \emptyset$. The variables in $\mathcal{V}_e$ and $\mathcal{V}_s$ denote environment-controlled variables (e.g., temperature) and state variables (e.g., the lock status of the front door), respectively. Each state $s \in \mathcal{S}$ can be viewed as a **labeling function** that maps each variable $v \in \mathcal{V}$ to a value in the domain with appropriate type. For instance, given a variable $v \in \mathcal{V}$ representing the lock status of a front door, the state s will map v to the one of the elements in the domain $\{\texttt{locked}, \texttt{unlocked}\}$, that is, $s(v) \in \{\texttt{locked}, \texttt{unlocked}\}$.

The transition relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ dictates how the system $\mathcal{H}$ changes states after observing an action. More precisely, for any $s_1, s_2 \in \mathcal{S}$ and $a \in \mathcal{A}$, if $\langle s_1, a, s_2 \rangle \in \mathcal{R}$ (or, in short $s_1 \xrightarrow{a} s_2$), then it signifies that after observing action a at state s_1 the system $\mathcal{H}$ moves to a state s_2. For instance, in a state in which the front door is locked when the system observes an action to unlock the front door then it will move to a state where the front door is now unlocked. We consider $\mathcal{R}$ to be *left-total* and $\mathcal{H}$ to be a deterministic LTS.

For a given smart-home $\mathcal{H}$, Expat's objective is to regulate $\mathcal{H}$'s behavior so that every state, $\mathcal{H}$ transitions to because of an app action, must satisfy the user expectations. Suppose user expectations are represented as quantifier-free first order logic (QF-FOL) formulae. The actual syntax and semantics of Expat's policy language Uei is presented just below. Given a user expectation Ψ as a QF-FOL formula, Expat modifies the original transition relation $\mathcal{R}$ of a given $\mathcal{H}$— provided that the initial state of $\mathcal{H}$ satisfies Ψ—to a new transition relation $\mathcal{R}_\Psi$ which is defined in the following way: $\mathcal{R}_\Psi = \{\langle s_1, a, s_2 \rangle \mid \langle s_1, a, s_2 \rangle \in \mathcal{R} \text{ and } s_2 \models \Psi\}$. Informally, for a given user expectation Ψ, this new transition relation $\mathcal{R}_\Psi$ essentially allows those transitions $s_1 \xrightarrow{a} s_2$ in $\mathcal{R}$ that take the system to a state s_2 that satisfies the user expectations Ψ (i.e., $s_2 \models \Psi$). For any state s and user expectations Ψ, we say $s \models \Psi$ *iff* the ground formula, obtained by replacing each variable $v \in \Psi$ with the concrete value $s(v)$, evaluates to true. For instance, given $s = \{x \mapsto 10, y \mapsto 1\}$ and $\Psi = x \geq y$, we can write $s \models \Psi$ as $s(x) \geq s(y)$ (or, simply $10 \geq 1$). The same state $s = \{x \mapsto 10, y \mapsto 1\}$, however, does not satisfy $\Psi_1 = x \geq y + 100$; written $s \not\models \Psi_1$.

4.2 Syntax of Uei

We now describe the concrete syntax of Uei (User Expectation Invariant) in which users can specify the invariants on a smart-home platform that they intend Expat to maintain. The syntax of Uei is shown as a BNF grammar in Figure 2. Uei was designed with generality in mind and thus some aspects of it are intentionally left as abstract (*e.g.*, predicates). We use "..." inside the production rules of Figure 2 to denote such abstract but extensible portions. The built-in constructs of Uei are designed after consulting different smart-home platforms [29, 34] and relevant literature [1, 7, 10, 13, 23, 31].

An Uei policy consists of one or more *policy statements*. Uei does not explicitly impose any ordering among the policy statements. Each policy statement is labeled with an identifier and comprises of an unordered sequence of *invariants*. The policy statement construct is purely syntactic in Uei, introduced particularly for grouping invariants based on some criteria (*e.g.*, regulating behavior of similar devices). The policy identifiers particularly comes in handy for referring to a group of invariants in the policy analysis tasks.

```
⟨Policy_Language⟩ ::= ⟨Policy_Statement⟩+
⟨Policy_Statement⟩ ::= 'Policy' ⟨String⟩ ':' ⟨Invariant⟩+
⟨Invariant⟩ ::= 'Invariant' ⟨String⟩ ':' ⟨Invariant_Body⟩
⟨Invariant_Body⟩ ::= ⟨Situation_Block⟩⟨Desire_Block⟩⟨Expectation_Block⟩
⟨Situation_Block⟩ ::= 'Situation' ':' ⟨Situation_Condition⟩
⟨Situation_Condition⟩ ::= 'any' | ⟨Condition⟩
⟨Desire_Block⟩ ::= 'Desire' ':' ⟨Desire_Value⟩
⟨Desire_Value⟩ ::= 'Expect' | 'Not' 'Expect'
⟨Expectation_Block⟩ ::= 'Expectation' ':' ⟨Condition⟩
⟨Condition⟩ ::= '('⟨Condition⟩')'| ⟨Key_Name⟩ ⟨Operator⟩ ⟨Value⟩
   | 'not' ⟨Condition⟩
   | ⟨Condition⟩ ⟨Boolean_Operator⟩ ⟨Condition⟩ | ...
⟨Boolean_Operator⟩ ::= 'and' | 'or'
⟨Key_Name⟩ ::= ⟨Trigger_Source_Related_Key⟩
   | ⟨Triggered_Event_Key⟩ | ⟨Device_Related_Key⟩
   | ⟨Action_Related_Key⟩ | ⟨Date_Time_Related_Key⟩
⟨Trigger_Source_Related_Key⟩ ::= 'rule_name' | ...
⟨Triggered_Event_Key⟩ ::= 'triggered_event_device'
   | 'triggered_event' | 'trigger_type' | ...
⟨Device_Related_Key⟩ ::= 'state' '(' ⟨String⟩ ')' | ...
⟨Action_Related_Key⟩ ::= 'action_device' | 'action_command' | ...
⟨Date_Time_Related_Key⟩ ::= 'current_time' | 'current_date' | ...
⟨Operator⟩ ::= '=' | '!=' | '>' | '>=' | '<' | '<=' | ...
⟨Value⟩ ::= ⟨String⟩ | ⟨Number⟩ | ⟨Time⟩ | ⟨Date⟩ | ⟨Boolean⟩ | ...
```

Figure 2: Concrete syntax of UEI as a BNF

```
Policy P1:
  Invariant I1:
    Situation: any
    Desire: Not Expect
    Expectation: state(FrontDoorLock) = OFF and
    (current_time >= 22:00:00 and current_time <= 6:00:00)
```

Figure 3: Policy example 1

An *invariant* in UEI, labeled by an identifier, captures the user expectations on the system state $\mathcal{S}$ at a particular situation. Conceptually, each invariant in UEI can be viewed to be of the following form: "when **situation** holds then **system property** must hold" in which **situation** refers to the *condition* under which the invariant is applicable, whereas **system property** expresses the condition the system state $\mathcal{S}$ must satisfy in that case. For instance, the user can define an invariant expressing "*in any situation, I do not expect the front door to be unlocked between 10 pm and 6 am*" (see Figure 3).

A user can define such an invariant with three internal blocks: *situation*, *desire*, and *expectation*. The situation block, starting with the label "`Situation:`", contains the *condition* under which the system invariant specified in the expectation block must be respected with accordance to the desire block. When the user wants to express any situation in an invariant, they can use the built-in "`any`" keyword. In the desire block, identified with the label "`Desire:`", the user specifies whether they expect the condition in the following expectation block to hold or not by using `Expect` or `Not Expect` keywords, respectively (see Figures 3 and 4). Finally, in the expectation block, the user specifies the condition that must be respected based on whether `expect` or `not expect` is used.

Conditions are boolean expressions with the logical operators (*i.e.*, `and`, `or`, `not`) connecting atomic conditional constructs. Atomic conditional constructs are an extensible set of *predicates*. Native UEI predicates are expressed with the infix notation and have the following form: `Key Operator Value` (*e.g.*, `current_time > 10:00:00`). `Key` is a either a system or an environmental variable, that is, `Key` $\in \mathcal{V}$. *Operator*, on the other hand, represents the built-in relational operators (e.g., $\neq, \geq$) while *Value* represents a constant whose types can be one of the following: String; Number; Time; Date; Boolean. Note that, types are also extensible in UEI.

```
Policy P2:
  Invariant I2:
    Situation: state(OutsideTemperatureSensor) < 50
    Desire: Expect
    Expectation: state(LivingRoomWindowLock) = ON
```

Figure 4: Policy example 2

The built-in keys (or, variables) `Key`$\in \mathcal{V}$ in UEI can be categorized into the following classes. For our current discussion, supposed that an event ev was triggered (e.g., door bell rang) which caused the system $\mathcal{H}$ to execute an automation rule/app r which in turn contemplated to take an action a (e.g., switch on the porch light).

(1) Trigger source-related key: This extensible set of keys is regarding different aspects of r which triggered the action a. Examples of such keys include `rule_name` (for, OpenHAB), `app_name` (for, SmartThings), or `app_id`.

(2) Triggered event-related key: This extensible set of keys is regarding properties of ev. The `triggered_event_device` (e.g., door bell) and `triggered_event` (e.g., ringing of door bell) are two examples of such keys.

(3) Device-related key: This set of keys allows the user to refer to the current state (*e.g.*, ON or OFF, 36.5°F) and types (*e.g.*, Switch, Contact Sensor) of devices in $\mathcal{H}$. We use the syntactic sugar `state(`d`)` for referring to the state of device d.

(4) Action-related key: This extensible set of keys is regarding the action a. For instance, `action_device` (e.g., porch light) and `action_command` (e.g., turn on) are two action-related keys.

(5) Date/Time-related key: Finally, this set of built-in keys $v \in \mathcal{V}_e$ allows users to express conditions regarding the current time and date of the system. Using the overloaded relational operators (*e.g.*, `<`, `!=`), date/time-related keys can be compared against the user provided literals (*e.g.*, `current_date >= 2019-01-14`).

Well-formed UEI policies. As UEI is a typed language, we expect a given UEI policy to respect the usual typing rules. For instance, one cannot write `state(FrontDoorLock)=36.5` because `state(FrontDoorLock)` is of enum type with the domain {ON, OFF} whereas the latter (*i.e.*, `36.5`) has the type real.

Also, we only allow action-related keys to appear in the situation block, not in the expectation block. Allowing action-related keys in the expectation block would allow the UEI policies to represent *obligatory actions* [22] which cannot be enforced right away as it may contradict with other policies. To explain this subtlety, let us take the policy in Figure 4 which states "*in the situation that outside temperature is below 50°F, I expect the living room window to be locked*". It may seem that *situation/expectation* concept is identical to the *trigger/action* paradigm used for writing automation apps. In the trigger/action paradigm, when a condition is satisfied (or, an event occurs), the specified action takes place right after. However, setting `Desire` to `Expect`, w.l.o.g. in the situation/expectation case, whenever the condition in the *situation* block holds, the *expectation*'s condition must already be satisfied. Given that, the invariant `I2`, in Figure 4, states that when outside temperature is below 50°F, the window must be already closed (and, as a result of this invariant the window must remain closed). Assuming that the system started in a good state (*i.e.*, window locked), this invariant will be maintained throughout the execution.

4.3 Formal Semantics of UEI

In this section, we present the formal semantics of UEI. UEI is intentionally designed to be declarative, that is, there is no ordering constraints on the invariants (or, policy statements) in a policy. Also, invariants (or, policy statements) of a policy are combined with a "*deny overrides allow*" approach, that is, an action is allowed only when all the invariants (or, policy statements) are satisfied. We want to, however, note that UEI is expressive enough to both encode priorities among the rules and support other combination approaches (e.g., first allow) through meta-variable introduction.

We provide the semantics of UEI policies by showing how to translate a UEI policy to a quantifier free first order logic (QF-FOL) formula with appropriate theories (*e.g.*, linear integer arithmetic, theory of finite strings, real arithmetic). Theories in QF-FOL provide interpretation to the different predicate symbols used in a formula (e.g., $\geq, \neq$). As a QF-FOL formula has formal semantics, the translation to QF-FOL allows UEI to also have a formal semantics. Such an approach of defining semantics has the particular benefit of enabling the use of SMT solvers to carry out different policy analysis tasks, as shown in the next section. Additionally, any extension of UEI which introduces predicates that SMT solvers can reason about will enjoy the same benefit regarding policy analysis. In what follows, we use $[\![Y]\!]^X$ to denote a function that takes as input an UEI **policy construct** Y (e.g., policy statement, invariant) with its **type** $X \in \{\mathsf{P}, \mathsf{PS}, I, C\}$ where $\mathsf{P}, \mathsf{PS}, I, C$ correspond to the type of UEI policy, policy statement, invariant, condition, respectively. $[\![Y]\!]^X$ outputs a corresponding QF-FOL formula that is equivalent to Y. We define $[\![Y]\!]^X$ inductively as follows.

Given an UEI policy $\mathcal{P} = [P_1, P_2, \ldots, P_n]$ where each P_i ($1 \leq i \leq n$) represents a policy statement, $\mathcal{P}$ is interpreted as a conjunctive QF-FOL logic formula of the form $\bigwedge_{i=1}^{n} [\![P_i]\!]^{\mathsf{PS}}$, written $[\![\mathcal{P}]\!]^{\mathsf{P}} = \bigwedge_{i=1}^{n} [\![P_i]\!]^{\mathsf{PS}}$. Recall that, each policy statement P_i has the following form $[I_1, I_2, \ldots, I_m]$ where each I_j ($1 \leq j \leq m$) represents an invariant. Each policy statement is also interpreted as a conjunctive formula of the following form: $[\![P_i]\!]^{\mathsf{PS}} = \bigwedge_{j=1}^{m} [\![I_j]\!]^I$. The definition of $[\![I_j]\!]^I$ thus will complete the presentation of UEI semantics.

Recall that, each invariant I_j has the form $\langle S, D, E \rangle$ where S corresponds to the situation condition, D refers to the desire block, and E signifies the expectation condition. Depending on D, $[\![I_j]\!]^I$ is defined in one of the following mutually exclusive ways in which $\Rightarrow$ signifies logical implication whereas $\neg$ refers to logical negation.

(1) $[\![I_j]\!]^I = [\![S]\!]^C \Rightarrow [\![E]\!]^C$ (when D contains `Expect`).
(2) $[\![I_j]\!]^I = [\![S]\!]^C \Rightarrow \neg[\![E]\!]^C$ (when D contains `Not Expect`).

The definition of $[\![S]\!]^C$ and $[\![E]\!]^C$ are similar with one exception, that is, $[\![\texttt{any}]\!]^C = \top$ where $\top$ signifies logical true. We show the case when the conditional expression inside S or E has the following form: `<Key_Name> <Operator> <Value>`. In this case, $[\![S]\!]^C = x \otimes c$ where x is the logical variable corresponding to the key, $\otimes$ denotes the predicate symbol specified by `<Operator>` (*e.g.*, $\leq$), and c indicates the typed constant value that corresponds to `<Value>` (*e.g.*, 36.5). The rest of the cases can be derived inductively following the UEI syntax directly (e.g., "`and`" becomes $\wedge$, "`not`" becomes $\neg$).

Example. Consider an UEI policy $\mathcal{P}_{\mathrm{ex}}$ with two policy statements `P1` and `P2` (see Figures 3 and 4). We can then write $[\![\mathcal{P}_{\mathrm{ex}}]\!]^{\mathsf{P}} =$ $(\top \Rightarrow \neg(\texttt{FrontDoorLock\_state} = \texttt{OFF} \wedge \texttt{current\_time} \geq 22{:}00{:}00 \wedge \texttt{current\_time} \leq 6{:}00{:}00)) \bigwedge (\texttt{OutsideTemperatureSensor\_state} < 50 \Rightarrow \texttt{LivingRoomWindowLock\_state} = \texttt{ON})$

4.4 Policy analysis

As the guarantees EXPAT's enforcement can provide are as strong as the UEI policy it is enforcing, for effective policy authoring, EXPAT provides support for policy analysis. The overarching goal of EXPAT's policy analyzer is to allow users to check whether a UEI policy captures the requirements the user intended.

PAL. For fine-grained policy analysis tasks, EXPAT provides a language called PAL (short for, Policy Analysis Language). The concrete syntax of PAL is presented in Figure 5. A PAL script has one or more *analysis commands*. Each analysis command describes the *analysis type* (e.g., consistency, entailment, equivalence) and up to two arguments of *policy formula* type. A policy formula could be a *policy identifier*, an *invariant identifier*, or the logical combinations of them. An example PAL script is shown in Figure 6.

For performing the analysis, a PAL specification is first converted into a QF-FOL satisfiability problem using EXPAT's PAL compiler. The compiler inductively constructs a QF-FOL formula while heavily using the semantic function $[\![Y]\!]^X$ (cf. § 4.3). An SMT-solver is then consulted to check for satisfiability/validity, depending on the analysis instructions. The policy analysis along with some additional feedback (*i.e.*, consistent model or counter-example) are presented to the user.

We now present the individual analyses supported by EXPAT.

4.4.1 *Consistency.*

The consistency analysis takes zero or one argument. In case, it is not provided an argument it considers the whole policy. However, when it is provided with a policy formula as an argument it focuses its analysis on that portion of the policy. It checks to see whether there is an action that will be allowed by the policy. Suppose the argument to consistency analysis is the policy formula f which after PAL compiler processes yield the QF-FOL formula Ψ. The consistency analysis tries to find concrete $s_1, s_2 \in \mathcal{S}$, and $a \in \mathcal{A}$ such that $s_1 \xrightarrow{a} s_2$ and $s_2 \models \Psi$. This can be carried out easily by consulting an SMT solver to check the satisfiability of Ψ. In case, SMT determines Ψ to be unsatisfiable, we return the UNSAT-core (*i.e.*, a smaller sub-formula of Ψ which is unsatisfiable) to the user which can help her to diagnose the problem in the policy.

As UEI policy invariants are of the form $\alpha \rightarrow \beta$, there is a possibility each invariant is vacuously true (*i.e.*, α is false). More precisely, the policy accepts all actions as all the situation conditions (*i.e.*, αs) are unsatisfiable. To this end, during policy consistency checking, we also check to see whether all αs are satisfiable. If all of them are unsatisfiable, we conclude that the policy is vacuous and we notify the user.

⟨*Policy_Analysis*⟩ ::= ⟨*Analysis_Commands*⟩$^+$
⟨*Analysis_Commands*⟩ ::= '`Check`' '`consistency`'(⟨*Policy_Formula*⟩|⟨*empty*⟩)
  | '`Check`' '`entailment`' ⟨*Policy_Formula*⟩'=>'⟨*Policy_Formula*⟩
  | '`Check`' '`equivalence`' ⟨*Policy_Formula*⟩'=='⟨*Policy_Formula*⟩
⟨*Policy_Formula*⟩ ::= ⟨*Policy_Identifier*⟩ | ⟨*Invariant_Identifier*⟩
  | '('⟨*Policy_Formula*⟩')' | '`not`' ⟨*Policy_Formula*⟩
  | ⟨*Policy_Formula*⟩ '`and`' ⟨*Policy_Formula*⟩
  | ⟨*Policy_Formula*⟩ '`or`' ⟨*Policy_Formula*⟩

Figure 5: Concrete Syntax of Pal

```
check consistency
check consistency P1 and not P2
check equivalence I1 == I2
check entailment I1 => P2
```

Figure 6: An example Pal script.

4.4.2 Entailment. The equivalence analysis takes two arguments of policy formula type. It then checks to see whether the first policy induced by the first policy formulae is less-or- equally permissive than the second policy, that is, there is no action a such that the first policy accepts it whereas the second one rejects it.

Suppose arguments to entailment analysis are the policy formulae f_1 and f_2 which after Pal compiler processes yield the QF-FOL formulae Ψ_1 and Ψ_2, respectively. It checks to see whether for all concrete $s_1, s_2 \in \mathcal{S}$, and $a \in \mathcal{A}$ such that $s_1 \xrightarrow{a} s_2$ the following holds: $(s_2 \models \Psi_1) \Rightarrow (s_2 \models \Psi_2)$. For checking this, we consult the SMT solver and check whether the formula $\neg(\Psi_1 \Rightarrow \Psi_2)$ is satisfiable. If the SMT solver concludes the formula to be unsatisfiable, then we notify the user that the first policy entails the second. Otherwise, we notify the user about the failure and provide the model returned by the SMT solver as the counterexample.

4.4.3 Equivalence. The equivalence analysis takes two arguments of policy formula type. It then checks to see whether policies induced by those policy formulae are equivalent, that is, for all possible actions a both policies return the same decision. Such analysis is particularly relevant when the user refactors a current policy to obtain a new policy and wants to check whether both policies are functionally equivalent.

Suppose arguments to equivalence analysis are the policy formulae f_1 and f_2 which after Pal compiler processes yield the QF-FOL formulae Ψ_1 and Ψ_2, respectively. It checks to see whether for all concrete $s_1, s_2 \in \mathcal{S}$, and $a \in \mathcal{A}$ such that $s_1 \xrightarrow{a} s_2$, the following is true: $(s_2 \models \Psi_1) \Leftrightarrow (s_2 \models \Psi_2)$ where $\Leftrightarrow$ denotes logical equivalence. For checking this, we consult the SMT solver and check whether the formula $\neg(\Psi_1 \Leftrightarrow \Psi_2)$ is satisfiable. If the SMT solver concludes the formula to be unsatisfiable, then we notify the user that the policies are equivalent. Otherwise, we notify the user that the policies are not equivalent and provide the model returned by the SMT solver as the counterexample.

4.5 Policy deployment

In this section, we will describe how Expat ensures that installed apps in a smart-home platform do not violate user expectations specified in the Uei language.

Towards this goal, Expat provides a runtime monitoring mechanism which takes as input an Uei policy $\mathcal{P}$ and then decides whether each requested action by apps in a smart-home system (*e.g.* unlocking the door) is aligned with $\mathcal{P}$. If the action a is aligned with $\mathcal{P}$, then a is permitted to be taken; otherwise, it is simply withdrawn without any interruption to the system operation. To check whether an action a complies with a policy $\mathcal{P}$, the runtime monitor relies on a policy decision function δ which we define below.

Policy decision function: The heart of Expat's runtime monitoring mechanism is the policy decision function δ which takes a Uei policy $\mathcal{P}$, a contemplated action a by an app, and the current system state s_c, and decides whether a is compliant with $\mathcal{P}$, that is, $\delta : \mathbb{P} \times \mathcal{A} \times \mathcal{S} \rightarrow \{\text{permit}, \text{deny}\}$ where $\mathbb{P}$ is the set of all possible policies specified in Uei. Given $s_c \xrightarrow{a} s_n$, the decision function δ just checks to see whether $s_n \models^? [\![\mathcal{P}]\!]^{\mathsf{P}}$. If $s_n \models [\![\mathcal{P}]\!]^{\mathsf{P}}$, then δ returns permit; otherwise, it returns deny. Recall that, $[\![\mathcal{P}]\!]^{\mathsf{P}}$ is a QF-FOL formula. Thus, δ just needs to evaluate the formula $[\![\mathcal{P}]\!]^{\mathsf{P}}$ with respect to s_n.

As the readers may have realized, the deployment of Expat's runtime monitoring mechanism relies entirely on the target smart-home platform. With that in mind, we need to investigate feasible deployment alternatives based on the existing platforms' architectural designs. For effectively deploying Expat's runtime monitoring mechanism in a smart-home platform, one has to answer the following two questions:

(1) How should one intercept each app's contemplated action?
(2) Where should the policy decision function be deployed so that it has a global and consistent view of the system state?

Appified smart-home platforms come generally in two flavors, either shipped with a proprietary hub backed by cloud-based services (*e.g.* Samsung SmartThings) or shipped with open-source implementations (*e.g.* OpenHAB). Although one can gain a full control of an open-source platform and flexibly deploy Expat wherever it fits best, both categories of systems provide users with a programming interface to build the automation apps, serving as an entry point to the platforms. To keep our approach as general as possible so that it can be applied to a wide variety of smart-home platforms, we leverage a platform's programming interface for deploying Expat.

4.5.1 Intercepting actions of each app. The programming interface provided by a platform enables users to write automation apps through a web-based IDE or just a text editor and then stores/installs them on the platform to mechanize automation processes amongst the smart devices. Recall that the automation apps are written based on the trigger/action paradigm. That is, an app requires to specify which triggers of interest it needs to be subscribed for and determine which actions should be taken after occurring those triggers. In order to authorize those requested actions, we need to place our reference monitor in appropriate location to first monitor the request context (*e.g.*, action, target device) and then check it against user's expectations using the decision function δ. Given that the programming interface (*i.e.*, app source code) is our entry point to the platforms, the only location we can monitor an action request is where it is being called in the app. This can be done through guarding each action request by an inline reference monitoring.

Having contemplated the set of all possible actions in the target platform, Expat can spot the appropriate places in the app source code to instrument with inline reference monitoring. This inline reference monitoring is achieved by putting the requested actions into an *if* statement block whose condition is a call to the decision function by passing the request context as its arguments. Hence, the result of the decision function determines as to whether the requested action should be taken or not.

4.5.2 Deploying policy decision function. Since the appified smart-home platforms generally deliver decent programming capabilities for the app developments, there are two alternatives for deploying/implementing the decision function: (1) off-site, in which all its functionalities are implemented in an external server and then being used by an app inside the platform for each requested action [8]; (2) *in situ*, in which the decision function is implemented locally within the platform to be used in inline reference monitoring. We choose the *in situ* deployment of δ because along with its benefit to privacy, this approach does not require going out of the platform to make a decision and hence reducing the policy checking overhead.

There are, however, the following two main challenges for *in situ* deployment of the decision function in smart home platform.

- The decision function needs to have a centralized view of the entire system to be able to make an informed decision according to user expectations, while each app in the appified platforms are designed to have partial isolated view from the rest of system.
- Each call to the decision function by inline reference monitoring must be synchronized to avoid any possible inconsistency caused by the concurrency which is very common in these platforms.

Addressing the first challenge requires direct support from the platform. One needs a built-in mechanism that enables the decision function, written in the target platform language, to have access to the system state and thus achieving a centralized view of the entire system. In Section 5, one candidate solution has been proposed for OpenHAB. There are other mechanisms providing this capability in other platforms as well.

The second challenge arises due to the high degree of concurrency and the asynchronous nature of app execution. If two apps have the same trigger condition and the condition is satisfied, those get executed concurrently. However, this concurrency can lead to a inconsistent situation as follows: suppose Expat checks the policy invariants and allows an action a based on the current system state where a device is in state s, but right before executing a the device's state changes to s' (from outside of this function), and operating a in s' will lead to an undesired state. This is a well-known race condition example in software security, called *the time of check to time of use bug* (TOCTOU). To prevent this issue, we use a global *mutex* (*i.e.* lock) to make any call to the decision function, leading to any state change in the system, synchronous. Although using a *mutex* resolves the issue, some performance overhead will be incurred which we, however, argue is negligible (see Section 6). To make it explicit, using locks leads to sequential execution of rules. Unlike cyber-physical systems (e.g., power-plants) where deadlines are crucial, in smart-home platforms, we believe the incurred overhead is tolerable. This sequential execution also does not limit Expat usage since platforms execute each triggered rule in a separate thread but their side- effects are sequential (*i.e.* action commands are sequentialized at hub). So, true concurrency is not possible.

Having considered all these aspects in the design of inline reference monitoring and the decision function, Expat parses the source code of the automation apps in the platform and automatically instruments all the actions inside them with in line reference monitoring which in turn calls the deployed decision function to enforce the policies inside the platform itself at the runtime.

```
rule "R1"
when
Item InteriorMotionSensor received command ON
then
FrontDoorLock.sendCommand(OFF)
end

rule "R2"
when
Item LivingroomTemperature received update
then
if (LivingroomTemperature >= 80){
LivingRoomWindowRemoteControl.sendCommand(OFF)
}
end
```

Figure 7: Rule DSL example in OpenHAB

5 IMPLEMENTATION

A prototype of Expat has been implemented to concertize its conceptual design as well as demonstrating the feasibility of our proposed approach. The Expat prototype is implemented for OpenHAB smart-home platform [29] which is an open-source, technology-agnostic system used for automating processes between smart devices. The automation units in OpenHAB are called *rules* and a user can write them in a DSL (Domain Specific Language). Figure 7 shows an example with two rules, *R1* and *R2*, written in OpenHAB DSL.

Following the trigger/action paradigm, each rule has a trigger section in `when` block, and a script section in `then` block where action(s) can be performed. *R1* unlocks the front door when the interior motion detector detects a motion whereas R2 opens the living room window when the temperature is higher than 80°F. Having understood the OpenHAB DSL basics, we now describe the implementation details of Expat using a simple scenario.

Uei. In our Expat implementation, we use the concrete portion of Uei's syntax described in Figure 2. Given the installed rules (Figure 7), in our scenario, the user wants to ensure that the front door will not get unlocked for any reason whenever they are away from home. This expectation can be violated if something/someone (e.g., a pet) trips the motion sensor. Hence the user specifies the Uei policy shown in Figure 8 $\Psi_{\text{frontDoorLock}}$ that states "if the front door gets unlocked, then it must be the case that I am home."

Policy analysis. For policy analysis, Expat hinges on the z3 SMT-Solver [36]. Expat first parses the Uei policy and based on the analysis task converts it to an SMT problem. Expat uses ANTLR [3] for generating the Uei parser whereas uses z3py [41] (z3's Python binding) to communicate with the solver programmatically. In our SMT encoding of Uei and the associated policy analysis task, we use *linear integer arithmetic* (LIA) theory to encode date/time related constructs of Uei, and *scalars* sort (*i.e.*, enumeration types) to define the domain of smart devices (*i.e.*, "items" in OpenHAB terminology) and the domain of possible states (*e.g.*, commands). Figure 9 illustrates the SMT-LIB [5] encoding of $\Psi_{\text{frontDoorLock}}$ along with the rest of the consistency checking done by Expat.

Policy deployment. Policy deployment takes a Uei policy policy invariants in Uei language and the installed rules source code as inputs and replaces the installed rules file with the instrumented

```
Policy P1:
  Invariant I3:
    Situation: state(FrontDoorLock) = OFF
    Desire: Expect
    Expectation: state(HomeMode) = ON
```

Figure 8: A policy dubbed $\Psi_{\text{frontDoorLock}}$ for ensuring that the front door remains locked when the user is away.

```
Encoding of input policy in SMT-LIB format for consistency analysis:
---------------------------------------------------------------------
(declare-datatypes ((Command 0)) ((Command (ON) (OFF) (OPEN) (CLOSED)
        (UP) (DOWN) (STOP) (MOVE))))
(declare-fun Presence_state () Command)
(declare-fun FrontDoorLock_state () Command)
(declare-fun I3 () Bool)
(declare-fun P1 () Bool)
(assert (= I3 (=> (= FrontDoorLock_state OFF) (= Presence_state ON))))
(assert (= P1 I3))

Result:
---------------------------------------------------------------------
Invariants are consistent!

Model:
---------------------------------------------------------------------
I3: True
FrontDoorLock_state: ON
Presence_state: OFF
P1: True
```

Figure 9: Expat's analysis output of the policy $\Psi_{\text{frontDoorLock}}$.

one in OpenHAB. The policy deployment begins with a consistency check on the input Uei policy to ensure policy consistency. It then parses both the policy and rules where the policy is used to synthesize the policy decision function δ in OpenHAB DSL whereas the rules are instrumented to guard each action with a call to the decision function. OpenHAB has several categories of actions [30] with which an app can send a command to a device, perform audio/voice-related actions, transfer an information via HTTP, *etc.* Expat uses this as reference to spot any action in the rules file. For instance, `sendCommand` and `postUpdate` are two methods used for sending a command to an item and updating an item's status, respectively. Figure 10 shows the instrumented rules DSL given the policy $\Psi_{\text{frontDoorLock}}$, and rules *R1* and *R2*.

In the instrumented DSL, each action is guarded by an *if* statement which in turn calls the decision function, `policy_check`, with the appropriate request context as arguments. Notice that, each `policy_check` call is synchronized by *mutex* provided by OpenHAB. The `policy_check` function is also defined in the same rules file to orchestrate the permission to take each action requested by those rules. The function contains the encoding of the policy invariant, `I3`, in line 6 and its result is passed to the `permission` variable in line 7 as the final decision. However, if we had multiple policy invariants, the permission would become the conjunction of them all. This function also declares a variable for the state of each device used in the policy (lines 4 and 5) and it is assigned with either the current state of the device or the requested action command parameter, if this state variable pertains to the device on which an action is requested. This is done as `policy_check` wants to test whether the requested state change of the device (after performing the requested action) complies with the policy. Once the instrumented rules' file has been generated by Expat, it updates the old rules' file and thus OpenHAB picks up the instrumented rules' file and OpenHAB enforces the policy *in situ* at runtime.

```
1              ... //necessary package imports
2 val ReentrantLock lock=new ReentrantLock()//global mutex
3 val policy_check = [String my_rule_name, GenericItem
      my_triggered_event_device, State my_triggered_event,
      String my_trigger_type, GenericItem my_action_device,
       State my_action_command |
4   val FrontDoorLock_state = if (FrontDoorLock ==
        my_action_device) my_action_command else
        FrontDoorLock.state
5   val Presence_state = if (Presence == my_action_device)
        my_action_command else Presence.state
6   val I3 = ! ( FrontDoorLock_state == OFF ) || (
        Presence_state == ON )
7   val permission = I3
8   ...  // log the request and policy decision
9   return permission] //end of policy decision function
10 rule "R1" //start of instrumented rule 1
11 when  Item InteriorMotionSensor received command ON //
      Trigger
12 then  val rule_name = 'r1'
13   val triggered_event_device = InteriorMotionSensor
14   val trigger_type = 'command'
15   val triggered_event = NULL
16   lock.lock() //acquiring lock
17   try {
18     //Inline call to the reference monitor
19     if (policy_check.apply(rule_name,
          triggered_event_device, triggered_event,
          trigger_type, FrontDoorLock, OFF)) {
20       FrontDoorLock.sendCommand(OFF)
21     }
22   } finally{lock.unlock()}//releasing lock
23 end //end of instrumented rule 1
24 rule "R2" //start of instrumented rule 2
25 when  Item LivingroomTemperature received update //Trigger
26 then  val rule_name = 'r2'
27   val triggered_event_device = LivingroomTemperature
28   val trigger_type = 'update'
29   val triggered_event = NULL
30   if (LivingroomTemperature >= 80) {
31     lock.lock() //acquiring lock
32     try {
33       //Inline call to the reference monitor
34       if (policy_check.apply(rule_name,
            triggered_event_device, triggered_event,
            trigger_type, LivingRoomWindowRemoteControl,
            OFF)) {
35         LivingRoomWindowRemoteControl.sendCommand(OFF)
36       }
37     } finally{lock.unlock()} //releasing lock
38   }
39 end //end of instrumented rule 2
```

Figure 10: OpenHAB rule DSL after Expat instrumentation.

6 EVALUATION

We evaluate Expat by applying it in a smart-home equipped with a multitude of apps/rules and and policies. Specifically, we seek to answer the following research questions: (a) Is Expat effective in ensuring user expectations? (b) How much overhead does Expat incur for enforcing Uei policies at runtime? The configuration of our setup and the datasets are publicly available [12].

6.1 Setup

Testbed information. For our experiment, we created our own testbed, where we deployed OpenHAB 2.4 [29] on a Raspberry Pi 3

1. Front Door Lock
2. Garage Door Lock
3. Car
4. Indoor Motion Sensor
5. Living Room Window
6. TV
7. Smoke Detector
8. Surveillance Camera
9. Bedroom Window
10. Water Valve
11. AC
12. Living Room Light
13. Fire Sprinkler
14. Bed Room Light
15. Leak Detector
16. Illuminance Sensor
17. Ceiling Fan
18. Temperature Sensor

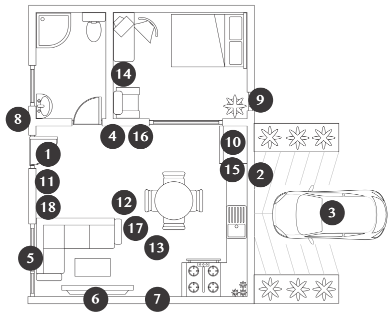

Figure 11: The virtual smart-home used as our testbed

Model B+. As the rules, policies, and Expat all run in the backend (*i.e.*, OpenHAB), we do not necessarily need actual physical smart devices to evaluate Expat. Thus, we leveraged the OpenHAB's web portal and created a virtual smart-home consisting of 1 bedroom, 1 living room, 1 garage, and other safety and security devices as shown in Figure 11. To create the necessary triggering events (*e.g.*, smoke `Detected`, TV is `On`), we used the web portal to directly interact with the devices. In reality, there are some indirect cause and effect relationships between devices. For instance, turning `on` the fire sprinkler starts flowing water, which the water leak detector senses and considers as a water leak. To enable such indirect activations in the virtual testbed, we manually simulated these triggers using the web portal as required during our experiments.

Datasets. To evaluate Expat, we crafted our own datasets instead of obtaining third-party rules from the marketplaces. The rationale behind this choice is two-fold. (i) While a large of rules exists in the wild, there is no existing oracle to establish the ground-truth and understand the semantic/intention of a rule without tedious manual effort. (ii) Unlike pre-deployment analysis approaches, Expat aims to enable runtime protection, and hence third-party rules from the marketplaces may not include the nuanced context to demonstrate Expat's efficacy. Therefore, we resorted to well studied problematic rules available in the literature [7, 8, 10, 13]. All these rules were created for the SmartThings platform [34]: some were from the official GitHub repository and some were entirely synthetic. Since the rule language for OpenHAB is different from the one for SmartThings, we ported all these rules for OpenHAB.

- *Rules.* We installed 15 rules in our smart-home, which are described in Table 1 using simple words. For instance, whenever the smoke detector detects smoke and the temperature increases above 135°F, rule R1 will turn `on` the fire sprinkler to contain the fire. Similarly, whenever the water leak sensor detects water, it triggers R2 which turns `off` the water valve to prevent financial loss due to damaged property and water bills. Rules can be malicious as well. For example, R9 is a malicious rule which embeds a sneaky command "[`unlock` front door]" such that whenever the user sets sleep mode *on*, the rule turns `off` the light and sends a stealthy command to `unlock` the front door.
- *Policies.* For our experiment, we used 8 policies written in Uei. Table 2 shows a simplified description of each policy. For example, policy PI1 states that whenever the water leak detector wants to shutdown the water valve, the user does *not* expect that the smoke detector to sense any smoke. Similarly, PI5 *inversely* states that the front door must not be unlocked by a rule in any condition. To demonstrate Expat's flexibility, we used different types of policies: conservative (*e.g.*, PI5) and contextual (*e.g.*, PI7).

Table 1: Rules used in our setup

ID	Rule Description
R1	smoke *detected* and temperature > *135°F* ↠ turn `on` fire sprinkler
R2	water leak *detected* ↠ turn `off` water valve
R3	*every Sunday at 8 PM* ↠ turn `on` TV
R4	TV turns *on* ↠ `open` living room window
R5	every working day at 6 AM ↠ `open` bed room window
R6	every working day at 8 AM ↠ `close` bed room window
R7	every day at sunset ↠ turn `on` light
R8	temperature > 80°F ↠ turn `on` ceiling fan
R9	sleep mode on ↠ turn `off` light [`unlock` front door]
R10	temperature > 75°F ↠ turn `on` AC
R11	temperature < 65°F ↠ turn `on` heater
R12	sleep mode on ↠ turn `off` all appliances
R13	indoor motion sensor *detected* ↠ `unlock` front door
R14	car distance from home < *150 YDS* ↠ `unlock` garage door
R15	garage door lock *opened* ↠ `unlock` front door

Table 2: Policy invariants used in our setup

ID	Policy Invariant Description
PI1	Water leak detector can shutdown water valve only if the smoke detector is not sensing smoke.
PI2	Any lights/windows can be turned on/opened only if the system is not on sleep mode.
PI3	In any situation, surveillance camera must remain on.
PI4	Bedroom window/light can be opened/switched on only if the vacation mode is turned off.
PI5	In any situation, front door must remain locked.
PI6	AC can be switched on only if the heating is off.
PI7	Lights/windows can be switched on/opened only if I am at home.
PI8	Living room window can be opened only if both heater and AC are off.

Table 3: Experiment scenarios and outcomes demonstrating Expat's effectiveness in blocking undesirable actions

Category	Group ID	Rules Involved	Policy Enforced	Denied Action
Implicit interplay	G1	(R1, R2)	PI1	water valve won't shut down
	G2	(R10, R11)	PI6	either AC or heater will turn on
Explicit interplay	G3	(R3, R4)	PI7	window won't open
Sneaky command	G4	R9	PI5	front door won't be unlocked
Benign but contextually undesired	G5	R12	PI3	surveillance camera will not turn off
	G6	R5	PI4	window will not be open
	G7	R8	PI7	fan will not turn on
	G8	R7	PI2	light will not turn on

6.2 Experimental Results

Effectiveness of Expat. To evaluate Expat's effectiveness, we adopted a careful guided approach instead of sampling a rule at random and activating its triggering events. Since Expat aims to ensure user expectations at runtime, we created 8 hand-crafted scenarios (see Table 3) to capture some unexpected situations by triggering one or more rules from Table 1. For each scenario, we wanted to check if Expat was able to enforce the policies in Table 2 and block any undesirable action according to the policies. Like our rules, some of the scenarios are based on the literature [7, 8, 10, 13].

Category-1: *Blocking undesired implicit interplay.* Scenarios G1 and G2 involve rules that interplay with each other *implicitly* (see Table 3). For instance, in case of G1, we first triggered R1 which turned `on` the fire sprinkler; then after sometime the water leak detector (virtually) sensed a water leak and triggered R2 which attempted to turn `off` the water valve. However, Expat denied R2's action as it violates PI1. Without Expat, R2's action would have shutdown the water valve causing severe damage due to the fire.

Category-2: *Blocking undesired explicit interplay.* The scenario G3 demonstrates how two rules (R3, R4) can explicitly interplay and lead to an unexpected situation. For G3, we first set the user was *away* and then virtually triggered R3 which turned `on` the TV. Once the TV was *on*, R4 was also triggered, but it failed to `open` the living room window, because Expat blocked R4's action due to PI7.

Category-3: *Blocking sneaky commands.* The scenario G4 demonstrates how Expat can block malicious rules containing sneaky commands. For G4, we first set the sleep mode `on`, which triggered R9 and as a result, the rule turned `off` the bedroom light. So far the execution went smoothly. However, being a malicious rule, R9 had a sneaky command to `unlock` the front door, which Expat denied since it violated PI5.

Category-4: *Blocking contextually undesired benign commands.* We demonstrated four scenarios (G5–G8) where the actions were benign but undesired in the specific context. For instance, in case of G6, we first enabled the `vacation` mode and virtually triggered R5, which was supposed to `open` the bedroom window but failed. Expat denied R5's action as it violated PI4.

Performance Overhead. To measure Expat run-time overhead, we ran each rule *without* and *with* the policies enabled, measured the difference in the elapsed time. In the former experiment, we ran each rule as it was written in the OpenHAB DSL, whereas in the latter experiment we used the instrumented version of each rule. In both experiments, we collected 10 data points for each rule. We observed that Expat incurs an average overhead of 63.11 ms (milliseconds) with standard deviation 5.91 for checking policies for each app which we argue is modest in our context.

7 DISCUSSION

Usability of Uei. As Uei is intended to be a user-facing language, its usability is of paramount importance. We do not have quantitative measurements backing the effectiveness of Uei. As the current paper focuses particularly on laying down the formal foundation of policy enforcement in appified smart-home platforms, we leave the evaluation of Uei through a user study as a future work.

Integration with IFTTT. IFTTT is a web-service that provides platform-agnostic automation mechanism [21]. Although OpenHAB can be integrated with IFTTT, the actions contemplated by an IFTTT rule is not visible to the app execution engine where Expat is deployed. To effectively regulate IFTTT, it is necessary to deploy Expat enforcement mechanism in a lower-level with full visibility of all actions. Such an approach, however, is not general enough to be deployed in other platforms (*e.g.*, Samsung SmartThings). In this paper, we aimed for generality and hence the current deployment mechanism of Expat cannot mediate IFTTT triggered actions.

Bootstrapping Expat. Expat's enforcement mechanism at the time of deployment requires the system to begin with a state where none of the invariants is violated. If the system were to be in a state where all invariants are violated, Expat would block all actions. To mitigate this, one can design a bootstrapping rule to initialize the system to be in a good state (*e.g.*, all lights are off, doors are locked).

8 RELATED WORK

While much work has gone into discovering or analyzing the vulnerabilities in smart IoT devices [19, 28, 32, 38] and in communication protocols [16, 25, 33, 40], an avenue of recent research [1, 6–11, 13, 18, 23, 24, 27, 31, 37, 39] focuses on the security of appified smart-home platforms, specifically, their backend and programming interfaces. We can conceptually categorize the prior work closely related to Expat based on their underlying mechanisms.

Static analysis. Prior efforts [6, 7, 10, 13] aim to develop static-analysis based mechanisms for detecting apps that violate the user's expectations prior to installation in her smart-home. These apps can violate expectations by commanding undesirable actions [13], leaking information [6], interfering with existing apps [10], or not satisfying high-level functional properties [7]. While early detection of vulnerable apps prior to installation can be effective during the initial setup, these approaches are prone to imprecision because of the incurred over- or under-approximation during static analysis. In addition, these approaches often rely on an abstract environment model to scale their analysis, and the lack of details in the model can amplify the false positive rates. We, on the other hand, relies on dynamic analysis to enable runtime checking of user expectations.

Dynamic analysis. Dynamic analysis based approaches [8, 23, 39] do not suffer from the limitations of static analysis. For providing runtime defense, some techniques (*e.g.*, ContexIoT [23], ProvThings [39]) rely on domain knowledge to enumerate vulnerable actions while IoTGuard [8] hinges on policies that express the user's expectations. However, they fall short in mitigating the threats. IoTGuard considers a mutli-app environment while ContexIoT analyzes each app in isolation, but they both ship all necessary state/context information from SmartThings platform to an outside entity (dubbed local server) using http(s) because their policy decision maker resides outside the platform. To make their local web server (with private IP) accessible to SmartThings, IoTGuard, for instance, needs to trust a third-party service (*i.e.* ngrok [26]). We argue that such outsourcing not only raises privacy issues but also incurs significant overhead because each individual action (*i.e.* turning on a light) requires communicating with the external server for policy decision. Contrarily, Expat only leverages platform's capability of executing an app and its programming interface by an *in situ* deployment, which does not mean deploying it in a "local server" (as discussed in IoTGuard). ProvThings focuses on forensic analysis and hence collects execution traces to draw causality inference, whereas Expat enables runtime protection to ensure user expectations.

Others. Toward IoT security, some existing research takes a different approach by proposing new system design [15], access control

[18, 24], programming framework immune to information leakage [14], risk-based analysis [11, 31], fuzzing smartphone apps controlling IoT devices [9]. Unlike them, Expat takes an orthogonal direction to enable runtime enforcement of user expectations.

9 CONCLUSION AND FUTURE WORK

To protect users of smart-home automation platforms from undesired actions of automation apps, this paper presents Expat. Expat is envisioned to be deployed in existing appified smart-home platforms as an *in situ* runtime monitor. To capture the user expectations of apps' behavior, Expat provides a specification language dubbed Uei which has a formal syntax and semantics. For effective policy authoring, Expat enables users to check desired properties of Uei policies through the use of an SMT solver. Finally, we demonstrated that Expat can be effortlessly instantiated for OpenHAB through instrumentation of installed automation apps which guards each contemplated action of apps with an inline call to the Expat's reference monitor. Our instantiation of Expat incurs only a modest overhead (*i.e.*, ~63 ms) while maintaining the users' expectations as invariants.

Future work. In future, to decrease runtime overhead, we would like to focus on a hybrid enforcement approach which given an Uei policy will first perform some static analysis to check whether some inline calls to the reference monitor in an app can be removed by statically proving compliance. For the rest of the actions, we would follow the same approach as in Expat of inline runtime monitoring.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by NSF grant CNS-1657124 and DARPA CASE program award N66001-18-C-4006. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government or the NSF.

REFERENCES

[1] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. SoK: Security Evaluation of Home-Based IoT Deployments. In *2019 IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[2] Amazon Alexa 2014. https://developer.amazon.com/alexa

[3] ANTLR 2019. ANother Tool for Language Recognition. https://www.antlr.org

[4] Apple HomePod 2019. https://www.apple.com/homepod/

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

[6] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. 2018. Sensitive Information Tracking in Commodity IoT. In *27th USENIX Security Symposium (USENIX Security 18)*.

[7] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 147–158.

[8] Z. Berkay Celik, Gang Tan, and Patrick McDaniel. 2019. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *Network and Distributed System Security Symposium (NDSS)*.

[9] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Network and Distributed System Security Symposium (NDSS)*.

[10] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. 2018. Cross-App Interference Threats in Smart Homes: Categorization, Detection and Handling. *CoRR* abs/1808.02125 (2018). arXiv:1808.02125

[11] Wenbo Ding and Hongxin Hu. 2018. On the Safety of IoT Device Physical Interaction Control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 832–846.

[12] EXPAT Github Repository 2019. https://github.com/expat-paper/expat.git.

[13] E. Fernandes, J. Jung, and A. Prakash. 2016. Security Analysis of Emerging Smart Home Applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, Vol. 00. 636–654. https://doi.org/10.1109/SP.2016.44

[14] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks.. In *USENIX Security Symposium*. 531–548.

[15] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. 2018. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *Proceedings 2018 Network and Distributed System Security Symposium*.

[16] Behrang Fouladi and Sahand Ghanoun. 2013. Honey, I'm Home!!, Hacking ZWave Home Automation Systems. *Black Hat USA* (2013).

[17] Google Home 2019. http://home.google.com/

[18] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. 2018. Rethinking access control and authentication for the home internet of things (iot). In *27th USENIX Security Symposium (USENIX Security 18). USENIX Association, Baltimore, MD*. 255–272.

[19] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wagner. 2016. Smart locks: Lessons for securing commodity internet of things devices. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*. ACM, 461–472.

[20] HomeKit 2019. https://www.apple.com/ios/home/

[21] IFTTT 2019. A world that works for you. https://ifttt.com/

[22] Keith Irwin, Ting Yu, and William H. Winsborough. 2006. On the Modeling and Analysis of Obligations. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. ACM, New York, NY, USA, 134–143. https://doi.org/10.1145/1180405.1180423

[23] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z. Morley Mao, and Atul Prakash. 2017. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *21st Network and Distributed Security Symposium (NDSS)*.

[24] Sanghak Lee, Jiwon Choi, Jihun Kim, Beumjin Cho, Sangho Lee, Hanjun Kim, and Jong Kim. 2017. FACT: Functionality-centric access control system for IoT programming frameworks. In *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*. ACM, 43–54.

[25] Natasha Lomas. 2015. Critical Flaw IDed In ZigBee Smart Home Devices. https://techcrunch.com/2015/08/07/critical-flaw-ided-in-zigbee-smart-home-devices/.

[26] ngrok - secure introspectable tunnels to localhost 2019. https://ngrok.com/.

[27] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V Krishnamurthy, Edward JM Colbert, and Patrick McDaniel. 2018. IotSan: fortifying the safety of IoT systems. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. ACM, 191–203.

[28] Sukhvir Notra, Muhammad Siddiqi, Hassan Habibi Gharakheili, Vijay Sivaraman, and Roksana Boreli. 2014. An experimental study of security and privacy risks with emerging household appliances. In *Communications and Network Security (CNS), 2014 IEEE Conference on*. IEEE, 79–84.

[29] openHAB 2019. https://www.openhab.org

[30] openHAB Action 2019. https://www.openhab.org/docs/configuration/actions.html#actions

[31] Amir Rahmati, Earlence Fernandes, Kevin Eykholt, and Atul Prakash. 2018. Tyche: A Risk-Based Permission Model for Smart Homes. In *2018 IEEE Cybersecurity Development (SecDev)*. IEEE, 29–36.

[32] Eyal Ronen and Adi Shamir. 2016. Extended functionality attacks on IoT devices: The case of smart lights. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 3–12.

[33] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin OâĂŹFlynn. 2017. IoT goes nuclear: Creating a ZigBee chain reaction. In *2017 IEEE Symposium on Security and Privacy (S&P)*.

[34] SmartThings 2019. https://www.smartthings.com/

[35] Bruce Sussman. 2018. 2-Second Hack: The Smart Lock, the YouTuber, and the Pen Tester. https://www.secureworldexpo.com/industry-news/iot-devices-hacked-example

[36] The Z3 Theorem Prover 2019. https://github.com/Z3Prover/z3

[37] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. 2017. SmartAuth: User-Centered Authorization for the Internet of Things. In *26th USENIX Security Symposium (USENIX Security 17)*.

[38] Blase Ur, Jaeyeon Jung, and Stuart Schechter. 2013. The current state of access control for smart devices in homes. In *Workshop on Home Usable Privacy and Security (HUPS)*. HUPS 2014.

[39] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. 2018. Fear and Logging in the Internet of Things. In *ISOC NDSS*.

[40] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. 2015. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *ACM Workshop on Hot Topics in Networks*.

[41] Z3Py Guide 2019. https://ericpony.github.io/z3py-tutorial/guide-examples.htm