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

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

Endadul Hoque  
Florida International University  
ehoque@fiu.edu

Proyash Podder  
Florida International University  
ppodder@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.

---

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.

SACMAT ’19, June 3–6, 2019, Toronto, ON, Canada  
© 2019 Association for Computing Machinery.  
ACM ISBN 978-1-4503-6753-0/19/06 ... $15.00  
https://doi.org/10.1145/3322431.3325107

---

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 mishbehavior 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 policies. 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 programing 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 process 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 proprietary 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 through 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 app₁ and app₂ providing some home safety features. app₁ 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. app₂, 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, app₂ will switch off the sprinklers. After the sprinklers switch on, app₂ 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 enforcementT) 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.
and select some apps to download and finally installs them on the platform (step ❸). 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 pin-point. Hence, the user specifies their expectations of the smart-home system in the presence of those apps using a high-level policy language $\text{Uei}$ (step ❹). 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 ❹) 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 ❹) such that the policies can be automatically enforced in situ before any action is executed (step ❹), 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, $\text{Uei}$ (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 $\text{Uei}$ policies at a high-level. In our context, a smart-home $H$ can be viewed as a labeled transition system (LTS) of the following form: $\langle S, A, V, \gamma, R \rangle$.

The $S$ component of $H$ represents a non-empty, possibly infinite, set of states. Note that, in the definition of $H$, we do not explicitly include a designated set of initial states intentionally to allow the system to start at any state. $A$ in $H$, on the other hand, represents the non-empty set of possible actions (e.g., turning on the light) recognized by $H$. 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, $a \in A$ can be a tuple of the following form: (requesting_app $\rightarrow$ app$_1$, action_device $\rightarrow$ smartLock$_1$, action_command $\rightarrow$ unlock).

$V$ represents an arbitrary but finite set of typed variables. $V$ can be decomposed into two mutually exclusive set of variables $V_s$ and $V_e$, that is, $V = V_s \cup V_e$ and $V_s \cap V_e = \emptyset$. The variables in $V_s$ and $V_e$ denote environment-controlled variables (e.g., temperature) and state variables (e.g., the lock status of the front door, respectively).

Each state $s \in S$ can be viewed as a labeling function that maps each variable $v \in V$ to a value in the domain with appropriate type. For instance, given a variable $v \in V$ representing the lock status of a front door, the state $s$ will map $v$ to one of the elements in the domain $\{\text{locked}, \text{unlocked}\}$, that is, $s(v) \in \{\text{locked}, \text{unlocked}\}$.

The transition relation $R \subseteq S \times A \times S$ dictates how the system $H$ changes states after observing an action. More precisely, for any $s_1, s_2 \in S$ and $a \in A$, if $\langle s_1, a, s_2 \rangle \in R$ (or, in short $s_1 \xrightarrow{a} s_2$), then it signifies that after observing action $a$ at state $s_1$ the system $H$ 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 $R$ to be left-total and $H$ to be a deterministic LTS.

For a given smart-home $H$, Expat’s objective is to regulate $H$’s behavior so that every state, $H$ 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 $\text{Uei}$ is presented just below. Given a user expectation $\Psi$ as a QF-FOL formula, Expat modifies the original transition relation $R$ of a given $H$—provided that the initial state of $H$ satisfies $\Psi$—to a new transition relation $R_\Psi$ which is defined in the following way: $R_\Psi = \{ \langle s_1, a, s_2 \rangle \mid \langle s_1, a, s_2 \rangle \in R \text{ and } s_2 \models \Psi \}$. Informally, for a given user expectation $\Psi$, this new transition relation $R_\Psi$ essentially allows those transitions $s_1 \xrightarrow{a} s_2$ in $R$ that take the system to a state $s_2$ that satisfies the user expectations $\Psi$ (i.e., $s_2 \models \Psi$). For any state $s$ and user expectations $\Psi$, we say $s \models \Psi$ iff the ground formula, obtained by replacing each variable $v \in V$ 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 $\text{Uei}$

We now describe the concrete syntax of $\text{Uei}$ (User Expectation Invariant) in which users can specify the invariants on a smart-home platform that they intend Expat to maintain. The syntax of $\text{Uei}$ is shown as a BNF grammar in Figure 2. $\text{Uei}$ 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 $\text{Uei}$ are designed after consulting different smart-home platforms [29, 34] and relevant literature [1, 7, 10, 13, 23, 31].

An $\text{Uei}$ policy consists of one or more policy statements. $\text{Uei}$ 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 $\text{Uei}$, 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.
An invariant in Uei, labeled by an identifier, captures the user expectations on the system state 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 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).

Figure 4: Policy example 2

> 10:00:00. Key is either a system or an environmental variable, that is, Key ∈ V. Operator, on the other hand, represents the built-in relational operators (e.g., ≠, ≥) 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.

The built-in keys (or, variables) Key ∈ 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 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 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 (e.g., 22:08:00) and current_time <= 6:00:00) are two action-related keys.

Well-formed Uei policies. As Uei is a typed language, we expect a given Uei policy to represent 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.r.t. 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., ≥, ≤). 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 \{P, PS, I, C\}$ where $P$, $PS$, $I$, $C$ 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 $\bigwedge_{i=1}^{n} [P_i]^{PS}$, written $\mathcal{P}^\mathcal{P} = \bigwedge_{i=1}^{n} [P_i]^{PS}$.

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: $[P_i]^{PS} = \bigwedge_{j=1}^{m} [I_j]^{I}$. The definition of $[I_j]^I$ thus will complete the presentation of Uei semantics.

Recall that, each invariant $I_j$ has the form $⟨S, D, E⟩$ 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 $\Leftrightarrow$ refers to logical negation.


The definition of $[S]^C$ and $[E]^C$ are similar with one exception, that is, $[\neg Y]^C = \top$ where $\top$ signifies logical true. We show the case when the conditional expression inside $S$ or $E$ has the following form: $⟨\text{Key Name}⟩ \langle \text{Operator}⟩ \langle \text{Value}⟩$. 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 $\langle \text{Operator}⟩$ (e.g., $\leq$), and $c$ indicates the typed constant value that corresponds to $\langle \text{Value}⟩$ (e.g., $36.5$). The rest of the cases can be derived inductively following the Uei syntax directly (e.g., “and” becomes $\land$, “not” becomes $\neg$).

**Example.** Consider an Uei policy $\mathcal{P}_{ex}$ with two policy statements $P_1$ and $P_2$ (see Figures 3 and 4). We can then write $\mathcal{P}_{ex}^\mathcal{P} = (\top \Rightarrow (\text{FrontDoorLock} = \text{OFF} \land \text{current time} \geq 22:00:00 \land \text{current time} \leq 6:00:00)) \land (\text{OutsideTemperatureSensor state < 50}) \Rightarrow \text{LivingRoomWindowLock state = ON})$

### 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 $\Psi$. The consistency analysis tries to find concrete $s_1, s_2 \in S$ and $a \in A$ such that $s_1 \overset{a}{\rightarrow} s_2$ and $s_2 \models \Psi$. This can be carried out easily by consulting an SMT solver to check the satisfiability of $\Psi$. In case, SMT determines $\Psi$ to be unsatisfiable, we return the UNSAT-core (i.e., a smaller sub-formula of $\Psi$ 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., $\alpha$ is false). More precisely, the policy accepts all actions as all the situation conditions (i.e., $\alpha$s) are unsatisfiable. To this end, during policy consistency checking, we also check to see whether all $\alpha$s are satisfiable. If all of them are unsatisfiable, we conclude that the policy is vacuous and we notify the user.
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 formula is less-or-equal 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 actions to entailment analysis are the policy formulae \( f_1 \) and \( f_2 \) which after \( \text{P}_{\text{AL}} \) compiler processes yield the QF-FOL formulae \( \Psi_1 \) and \( \Psi_2 \), respectively. It checks to see whether for all concrete \( s_1, s_2 \in S \), and \( a \in A \) 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 actions to equivalence analysis are the policy formulae \( f_1 \) and \( f_2 \) which after \( \text{P}_{\text{AL}} \) compiler processes yield the QF-FOL formulae \( \Psi_1 \) and \( \Psi_2 \), respectively. It checks to see whether for all concrete \( s_1, s_2 \in S \), and \( a \in A \) such that \( s_1 \xrightarrow{a} s_2 \), the following is true: \( (s_2 \models \Psi_1) \Leftrightarrow (s_2 \models \Psi_2) \). 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 \textit{Expat} ensures that installed apps in a smart-home platform do not violate user expectations specified in the Utiu language.

Towards this goal, \textit{Expat} provides a runtime monitoring mechanism which takes as input an Utiu policy \( P \) and then decides whether each requested action by apps in a smart-home system (e.g., unlocking the door) is aligned with \( P \). If the action \( a \) is aligned with \( P \), 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 \( P \), the runtime monitor relies on a policy decision function \( \delta \) which we define below.

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

As the readers may have realized, the deployment of \textit{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 \textit{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?

Appied 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 \textit{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 \textit{Expat}.

4.5.1 Interception of 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/install 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 \( \delta \). 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, \textit{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 inconsistent situations 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 the concurrency which in turn calls the deployed decision function to enforce the policies inside the platform itself at the runtime.

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 for 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 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 #)) ((Command (ON) (OFF) (OPEN) (CLOSED)) (UP) (DOWN) (STOP) (MOVE)))
(declare-fun Presence_state () Command)
(declare-fun FrontDoorLock_state () Command)
(declare-fun P1 () Bool)
(declare-fun I3 () Bool)
(assert (= P1 (+ (= FrontDoorLock_state OFF) (= Presence_state ON))))
(assert (= I3 (+ (= FrontDoorLock_state OFF) (= Presence_state ON))))
```

Result:

```
Model:
```

```
for each device used in the policy (lines 7)
    rules file to orchestrate the permission to take each action requested
    with either the current state of the device or the requested action
    of them all. This function also declares a variable for the state of
    the old rules’ file and thus OpenHAB picks up the instrumented
    rules’ file and OpenHAB enforces the policy in situ at runtime.
```

```
Desire: Expect
Expectation: state(HomeMode) = ON
```

Invariants are consistent!

```
I3: True
FrontDoorLock_state: OFF
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
test 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 $\delta$ 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
```
when Item InteriorMotionSensor received command ON // Trigger
then val_rule_name = ‘r1’
val triggered_event_device = InteriorMotionSensor
val trigger_type = ‘command’
val triggered_event = NULL
lock.lock() //acquiring lock
try {
  //Inline call to the reference monitor
  if (policy_check.apply(rule_name, triggered_event_device, triggered_event, trigger_type, FrontDoorLock, OFF)) {
    FrontDoorLock.sendCommand(OFF)
  }
  finally(lock.unlock()) //releasing lock
end //end of instrumented rule 1
```

```
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
Rule Description
PI7
Bedroom window/light can be opened/switched on only if the vacation sleep mode on every working day at 8 AM light will not turn on fan will not turn on surveillance camera will
In any situation, front door must remain locked.

Policy Invariant Description
PI7
PI3
water valve won't shut
R8
In any situation, surveillance camera must remain on.
temperature > 75 every day at sunset window will not be open car distance from home (R10, R11) either AC or heater will
refrigerator window will not be open

Denied Action
Water leak detector can shutdown water valve only if the smoke detector detects smoke and the temperature increases above 135°F, rule R11 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 UEL. Table 2 shows a simplified description of each policy. For example, policy P11 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, P15 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., P15) and contextual (e.g., P17).

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

<table>
<thead>
<tr>
<th>ID</th>
<th>Rule Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>smoke detected and temperature &gt; 135°F ↞ turn on fire sprinkler</td>
</tr>
<tr>
<td>R2</td>
<td>water leak detected ↞ turn off water valve</td>
</tr>
<tr>
<td>R3</td>
<td>every Sunday at 8 PM ↞ turn on TV</td>
</tr>
<tr>
<td>R4</td>
<td>every working day at 8 AM ↞ open living room window</td>
</tr>
<tr>
<td>R5</td>
<td>every working day at 8 AM ↞ open bed room window</td>
</tr>
<tr>
<td>R6</td>
<td>every working day at 8 AM ↞ close bed room window</td>
</tr>
<tr>
<td>R7</td>
<td>every day at sunset ↞ turn on light</td>
</tr>
<tr>
<td>R8</td>
<td>temperature &gt; 80°F ↞ turn on ceiling fan</td>
</tr>
<tr>
<td>R9</td>
<td>sleep mode on ↞ turn off light [unlock front door]</td>
</tr>
<tr>
<td>R10</td>
<td>temperature &lt; 60°F ↞ turn on AC</td>
</tr>
<tr>
<td>R11</td>
<td>temperature &lt; 60°F ↞ turn on heater</td>
</tr>
<tr>
<td>R12</td>
<td>sleep mode on ↞ turn off all appliances</td>
</tr>
<tr>
<td>R13</td>
<td>indoor motion sensor detected ↞ unlock front door</td>
</tr>
<tr>
<td>R14</td>
<td>car distance from home &gt; 150 YDS ↞ unlock garage door</td>
</tr>
<tr>
<td>R15</td>
<td>garage door lock opened ↞ unlock front door</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Policy Invariant Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P11</td>
<td>Water leak detector can shutdown water valve only if the smoke detector is not sensing smoke.</td>
</tr>
<tr>
<td>P12</td>
<td>Any lights/windows can be turned on/opened only if the system is not on sleep mode.</td>
</tr>
<tr>
<td>P13</td>
<td>In any situation, surveillance camera must remain on.</td>
</tr>
<tr>
<td>P14</td>
<td>Bedroom window/light can be opened/switched on only if the vacation mode is turned off.</td>
</tr>
<tr>
<td>P15</td>
<td>In any situation, front door must remain locked.</td>
</tr>
<tr>
<td>P16</td>
<td>AC can be switched on/off only if the heating is on/off.</td>
</tr>
<tr>
<td>P17</td>
<td>Lights/windows can be switched on/opened only if I am at home.</td>
</tr>
<tr>
<td>P18</td>
<td>Living room window can be opened only if both heater and AC are off.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
<th>Group ID</th>
<th>Rules Involved</th>
<th>Policy Enforced</th>
<th>Denied Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit interplay</td>
<td>G1</td>
<td>(R1, R2)</td>
<td>P11</td>
<td>water valve won't shut down</td>
</tr>
<tr>
<td></td>
<td>G2</td>
<td>(R10, R11)</td>
<td>P16</td>
<td>either AC or heater won't turn on</td>
</tr>
<tr>
<td>Explicit interplay</td>
<td>G3</td>
<td>(R3, R4)</td>
<td>P17</td>
<td>window won't open</td>
</tr>
<tr>
<td>Sneaky command</td>
<td>G4</td>
<td>R9</td>
<td>P15</td>
<td>front door won't be unlocked</td>
</tr>
<tr>
<td>Benign but contextually undesired</td>
<td>G5</td>
<td>R12</td>
<td>P13</td>
<td>surveillance camera will not turn off</td>
</tr>
<tr>
<td></td>
<td>G6</td>
<td>R5</td>
<td>P14</td>
<td>window will not be open</td>
</tr>
<tr>
<td></td>
<td>G7</td>
<td>R8</td>
<td>P17</td>
<td>fan will not turn on</td>
</tr>
<tr>
<td></td>
<td>G8</td>
<td>R7</td>
<td>P12</td>
<td>light will not turn on</td>
</tr>
</tbody>
</table>
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., ContextIoT [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 multi-app environment while ContextIoT 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

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