

IOT SECURITY WITH PARAMETRIC SIGNAL TEMPORAL LOGIC

by

Yifei Yang

A Thesis

Presented to the
Computer Science Department
within the Carnegie Mellon University
School of Computer Science
in partial fulfillment of
the requirements for the degree of
Masters
in
Computer Science

December, 2021

© COPYRIGHT 2021 BY YIFEI YANG
ALL RIGHTS RESERVED

TABLE OF CONTENTS

List of Figures	v
List of Tables	vi
List of Abbreviations	vii
1 Introduction	viii
1.1 Motivating Example	viii
1.2 Defining Safety	viii
1.3 Our Contribution	ix
2 Preliminaries	x
2.1 Samsung Smartthings IoT Environment	x
2.1.1 Smartapps	x
2.1.2 Device Handlers	x
2.1.3 Samsung Smartthings Hub	x
2.2 PSTL Rules	x
2.2.1 Syntax and Semantics of STL	xi
2.2.2 Parametric Signal Temporal Logic	xi
2.2.3 PSTL Primitives in IoT Environments	xi
2.3 Boolean expressions and SMT Formulas	xii
2.4 Powerset and Cartesian Products	xii
3 Learning Rules with Decision Tree	xiii
3.1 Multi-Class Classification Problem (MCCP)	xiii
3.2 STLTree	xiii
3.2.1 Model Overview	xiii
3.2.2 Stop Condition	xiv
3.2.3 Finding the Best Partition	xiv
3.2.4 Checking Rule Satisfaction	xv
3.2.5 Inference with Tree Model	xv
3.2.6 Visualizing Device Interaction Rules	xv
3.2.7 Modifications to the framework	xvi
3.3 TreeNoSTL	xviii
3.3.1 Model Overview	xix
3.3.2 Preprocessing Data	xix
3.3.3 Evaluation with Tree Model	xx
3.3.4 Visualizing Device Interaction Rules	xx
3.4 Evaluations	xx
3.4.1 Environments Using State Machine	xx

3.4.2	Environments Using Smartthings Simulation	xxi
3.4.3	Results	xxi
4	STLMon: Runtime Monitoring of Smartthings Environments	xxiv
4.1	Parsing Device Interaction Rules	xxiv
4.1.1	Parsing STLTree Rules	xxiv
4.1.2	Parsing TreeNoSTL Rules	xxv
4.1.3	Parsing User Rules	xxvi
4.2	Monitor Structure	xxviii
4.2.1	Monitor Backend	xxviii
4.2.2	Communicating with Smartthings Hub	xxix
4.2.3	Issuing Command to Devices	xxix
4.3	Checking Rule Violations	xxx
4.3.1	Checking Temporal DONT Rules	xxx
4.3.2	Checking Temporal DO Rules	xxxii
4.3.3	Checking Immediate Rules	xxxvi
4.4	Evaluation	xxxvii
4.4.1	Generating Rules	xxxvii
4.4.2	Device Change Log Generation	xxxviii
4.4.3	Evaluation Results	xl
5	STLCheck: Checking Conflicts within Interaction Rules	xli
5.1	Checking Rule Satisfaction	xli
5.2	Checking Conflicts within Environment	xlii
5.3	Samsung Smartthings Conflict Simulation	xliii
5.4	Evaluation	xliv
6	Related Works	xlvi
7	Conclusion and Future Work	xlvii
8	References	xlviii
9	Appendix	xl ix
9.1	Information Gain (IG) with robustness	xl ix
9.2	Simulated Annealing (SA) in STLTree	xl ix
9.3	Treepruning with Effective Alpha	xl ix
9.4	OneHotEncoder	1

ABSTRACT

Being one of the most rapidly growing technologies today with its predicted count of 22 billion in 2025, IoT devices and the systems containing these devices are getting increasingly more diverse and complicated. Keeping these IoT systems secure is an important yet challenging problem due to the hidden interactions between devices that can potentially result in security rule conflicts and violations of user intentions. To detect these conflicts and violations due to hidden interactions, we first introduce a novel approach to characterizing time-based interaction rules using Parameterized Signal Temporal Logic (PSTL). Then, we then propose STLTree, an adaptation and expansion of a decision tree learning algorithm for PSTL, to learn these interaction rules from a device operational log. Next, we define a notion of security in an IoT environment as an environment free of conflicts between interactions and user-desired security rules. Building on this work, we then propose two complementary approaches to detecting rule conflicts: (1) STLMon, which dynamically monitors IoT device behavior for rule violations, and (2) STLCheck, statically checks for the conflicting behaviors between and within device state changes and the user-defined rules. Finally, we show that STLTree is able to automatically generate rules that capture relevant interactions in our testing environment. Also, we demonstrate the efficiency of STLMon in detecting rule violations at runtime, and the utility of STLCheck in static generation of possible rule violations.

Dedication

Thank you Professor Eunsuk Kang to being a great thesis mentor, thank you for your continuous encouragement, support and steering to me towards the right direction for moving the project forward.

Thank you Professor Yuvraj Agarwal for reviewing my thesis and your valuable feedbacks.

Thank you to all the friends and family that continued to support me throughout my masters at Carnegie Mellon University.

List of Figures

1	Example hidden interaction, turning on the oven sets of smoke that unlocks the door	viii
2	Capability Specification for Lock (partial), 'lock' change state to locked while 'unlock' change state to unlocked, the other state values can only be achieved by errors in executing the state change	x
3	Example translation, the above tree translates to a dictionary mapping 1 to $(p1 \wedge p2) \vee (\neg p1)$ and 2 to $(p1 \wedge \neg p2)$	xvi
4	Example data handling, Top: Before, Bottom: After	xvii
5	Example dictionary mapping states to class number for Figure 4	xvii
6	Example rule output after text translation for STLTree	xix
7	Example data handling, Top: Before, Bottom: After	xx
8	Example rule output after text translation for TreeNoSTL	xx
9	Example rule. A high temperature of greater than 85 degrees triggers the smoke alarm siren . .	xxi
10	Samsung Smartthings simulation	xxi
11	Successful time related rule: Switch 1 on leads to Switch 2 on after 3 seconds	xxii
12	Successful regular rule: Switch 2 off leads to Smoke Alarm off	xxii
13	Failure in capturing interaction in entirety	xxii
14	Example DONT rule dictionary, there are two rules for the "Door_lock" to be locked	xxv
15	Conversion of DO dict from figure 14	xxv
16	Example DONT rule, there are two rules for the "Door_lock" basing on thermostat temperature	xxvi
17	Conversion of DO dict from figure 16	xxvi
18	Conversion of user defined rule. Above: rule, Below: converted list used to construct the DONT dict after Cartesian product method	xxviii
19	Runtime Monitor Structure	xxviii
20	Input dictionary for fuzz testing	xxxvii
21	Generated rule output	xxxviii
22	Generated log output	xl
23	Generated log output for conflict between the two rules. Each entry in log is in 4-tuple of (times-tamp from current time, device, state, value)	xlii

List of Tables

1	Average runtime of STLCheck for environments with different parameter combinations	xlv
---	--	-----

List of Abbreviations

G

GG Gini Gain. xix

I

IG Information Gain. ii, xiv, xvii, xlix

IoT Internet of Things. iii, viii

M

MCCP Multi-Class Classification Problem. i, xiii

MCR Misclassification Rate. xiii

P

PSTL Parametric Signal Temporal Logic. iii, viii, ix, xi, xlvi, xlvii

S

SA Simulated Annealing. ii, xviii, xxii, xlix

SMT Satisfiability Modulo Theory. xii

STL Signal Temporal Logic. xi, xlvi

1 Introduction

Internet of Things (IoT) is a blanket term for various gadgets that most people do not think as computers but still have the preprocessing power and an internet connection, which makes them able to communicate with each other in their corresponding IoT system environment. IoT have a tremendous impact on people's daily lives in the modern era of automation; for example, even common household kitchenware are able to communicate with each other through IoT. As a result, it is crucial to ensure the security of these IoT environments due to the functionality of the system's devices in the user's private household settings. However, it is often difficult to keep these smart systems secure as they can have complicated structures resulting from many hidden interactions between individual devices. This thesis project is motivated by the complications in achieving IoT security from the hidden interactions, and aims to solve the difficulties of these interactions by characterizing, learning, monitoring, and detecting conflicts them through Parametric Signal Temporal Logic (PSTL).

In subsection 1, we would like to elaborate on an motivating example of hidden interactions in IoT environments. Then, from the inspiration of our example, we define our notion of IoT environment safety in subsection 2. Finally, in subsection 3, we would expand on the contributions of our thesis to the security complications of IoT hidden interactions.

1.1 Motivating Example

To demonstrate the effect of hidden interactions in IoT environments, consider an apartment where the owner has installed an oven, a smoke alarm, and a door as IoT devices. In an apartment setting, these IoT devices are often implemented and installed by third parties with interaction rules that our owner is not sure about. In our example case, it is very likely that our apartment owner only know the safety property claimed by the third parties such as the door will be unlocked to let the owner out to safety when the smoke alarm sirens without actually knowing the implementations of the devices.

Now, our apartment owner is anxious since while unlocking the apartment door is a necessary measure to protect the owner from fires, unlocking the door in situations such as when the owner is not at home during a false alarm could also lead to security risks. Although for some situations it is safe for the owner to assume that the IoT devices are not installed maliciously, due to the increasing complexity in IoT household environments, it is fairly easy for designers to overlook hidden interactions within devices during implementation that could result in unsafe behaviors.

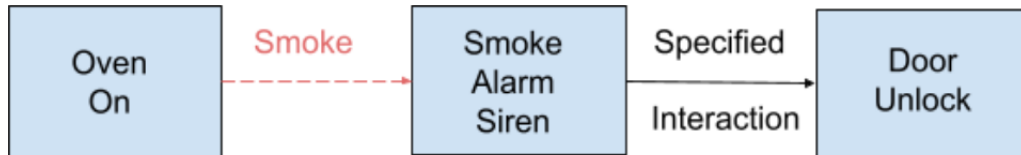


Figure 1: Example hidden interaction, turning on the oven sets of smoke that unlocks the door

For example, in an device implementation shown in Figure 1, while there is no direct device interaction between the oven and the door, the oven has the hidden behavior that it is able to set of smoke when turning on at high temperature for a long time. While the smoke alarm may be highly secure against attacks due to its key role in the security of the apartment with its ability to unlock the door, the oven may be implemented in a less secure way since it has no observable or physically implemented interaction with other devices. As a result, it would be much easier for an attacker to compromise the security of the apartment when the owner is not at home by turning the oven on and wait for the door to open as the smoke alarm sirens.

1.2 Defining Safety

As shown in our motivating example above, to ensure the safety of the apartment, it would be necessary for the owner to know the actual rules that the device interactions follow, including each individual device's inherent hidden behaviors such as the case above with oven and smoke. Furthermore, the owner would need a way to check for whether any of the interaction rules would violate the owner's desired safety behavior (For example, the behavior in Figure 1 would violate the safety desire for the door to stay locked if the owner is not home). Finally, both the device interaction rules and the user designed rules should not be conflicting within themselves, since it is impossible to specify a device, say a light switch, to be on and off at the same time. Formally, with device interactions and user desired behaviors to be described as abstract rules, we define an IoT environment to be safe in Definition 1 and 2 below.

Definition 1: Safety Given a set of device interactions $R = \{r_1, r_2, \dots, r_n\}$ in environment E and user desired behaviors $U = \{u_1, \dots, u_n\}$, E is safe if and only if there are no conflicts in $R \cup U$.

Definition 2: Conflict: Given a set of rules S , a conflict occurs if there are multiple rules s_1, \dots, s_n that specifies the same devices to be in different state values k_1, \dots, k_n that have rule preconditions that can be satisfied at the same time.

1.3 Our Contribution

In our work, we first characterized the device interaction and user desired behavior through immediate rules specifying immediate device changes and PSTL rules defined in [ADMN12] specifying time related behavior. We then constructed STLTree, a decision learning tree algorithm adapted from the tree model in [BVP⁺16] and expanded into learn PSTL rules from device change logs of each IoT environment and TreeNoSTL, the decision tree learning algorithm for immediate rules. Then, to detect for potential violations and verify learned environment rules, we implemented STLMon, a runtime monitor that is able to dynamically check for device changes and detect rule violations within each environment. Finally, to verify whether each IoT environment is safe by our Definition 1 and 2, we then created STLCheck, a conflict checker to check for the potential unsafe properties of the learned interaction rules with respect to user desired safety behavior. Currently, our tools are implemented specifically for the Samsung Smartthings environments due to the simplicity in evaluation with the platform’s virtual simulation feature; however, the implementations can be trivially converted for environments on other platforms with minor adjustments in acquiring and preprocessing device state change data.

For the remaining of the paper, Section 2 introduces the background for Samsung Smartthings Environment, PSTL rules, and mathematical foundations needed for the implementation of the tools. Then, Section 3, 4, and 5 elaborates on the implementation and evaluation of STLTree, STLMon, and STLCheck respectively. Finally, we discuss the related work, conclusion, and future work in section 7 and 8.

2 Preliminaries

2.1 Samsung Smartthings IoT Environment

The IoT environments our models will be targeting and using for testing are constructed through the Samsung Smartthings platform [Smab]. In a Samsung Smartthings environment, each IoT device is called a Smart device that can be either controlled physically or through Smartapps. Through internet connection, the Smartapps are able to automatically change a device's behavior when conditions for the app to run are met. Thus, the device interactions in a Samsung Smartthings environment that we would like to analyze are solely controlled by the Smartapps installed. However, due to the ability to control a device physically, there could be noise in the device interactions, as devices are able to change states by purely random choice of user's physical control. Due to this noise, our models would only operate on device interactions that they have a high confidence on, so that this noise would have a negligible impact.

2.1.1 Smartapps

Each Samsung Smartapp first specifies a list of devices it has access to in a preference block. When a device grants access to the Smartapp, it gives the Smartapp's ability to subscribe to its state changes, directly change its state, and access all of its past state changes. Each Smartapp also has the ability to communicate outside of Samsung Smartthings using OAuth, and is able to handle requests from local servers with the correct key and token for accessing data.

2.1.2 Device Handlers

Each Samsung Smart device is associated with a device handler, which is used to handle state change commands sent by the Smartapp. Each device handler has specific functions for each possible state change, which is specified by the capabilities of the device. In order to change the device state, the Smartapp can simply call the function on the device.

For most capabilities, the corresponding state change function has the identical name as the state value the user wanted to change. For example, calling *switch.on()* turns a switch on. For others, it is fairly easy to infer the state change function for the corresponding value change as shown in Figure 2 below. A complete list of Samsung Smartthings device capabilities and their corresponding state change functions can be found in [Smaa].

```
{
  "name": "Lock",
  "attributes": {
    "lock": {
      "schema": {
        "properties": {
          "value": {"enum": ["locked", "unknown", "unlocked", "unlocked with timeout"]}
        },
      },
    },
  },
  "commands": {
    "lock": {"arguments": []}, "unlock": {"arguments": []}
  }
}
```

Figure 2: Capability Specification for Lock (partial), 'lock' change state to locked while 'unlock' change state to unlocked, the other state values can only be achieved by errors in executing the state change

2.1.3 Samsung Smartthings Hub

The Samsung Smartthings Hub is used to coordinate the Smart devices and the Smartapp interactions as specified by the environment. When a device state change occurred, the hub sends the event to all the Smartapps that subscribed to the event to handle. And when a Smartapp responds with a device change, the Smartthings sends the device change to the device handler to execute. All the events and device state changes in a Smartthings environment are also stored in Samsung Smartthings Hub and can be retrieved through Smartapp communication. However, it is only possible to store 200 state and event changes for each device in the hub, and past changes will be replaced by newer ones.

2.2 PSTL Rules

In a typical IoT environment, most device interactions does not happen immediately. Instead, a device interaction would trigger often after a period of time when the conditions for the interaction is satisfied, or stays satisfied for a

period of time. As a result, we introduced Signal Temporal Logic (STL) for learning the rules to the device interactions in the environment to encapsulate this time-sensitive characteristic. To learn the device rules with a decision tree, we extended STL with Parametric Signal Temporal Logic (PSTL), where the parameters are the learning objectives.

2.2.1 Syntax and Semantics of STL

A signal is defined to be a continuous-time, continuous valued function s mapping from \mathbb{R}^+ to \mathbb{R}^n , and we use $s(t)$ to denote the value of signal s at t , and $s[t]$ to denote the value of s shifted by t time units, that is, $s[t](\tau) = s(\tau + t)$, $\forall \tau \in \mathbb{R}^+$. The boolean clauses of a STL rule is corresponding to each individual component s_i of s , $i \in \{1, 2, \dots, n\}$, which each component can be obtained through functions in set $G = \{g_i : \mathbb{R}^n \rightarrow \mathbb{R}, g_i(s) = s_i\}$. Now, the syntax of STL is defined as follows: [MN04]

$$\phi ::= \top \mid p_{g(x) \leq \mu} \mid \neg \phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 U_{[a,b]} \phi_2$$

where \top is True, and $p_{g(x) \leq \mu}$ is a predicate clause over \mathbb{R}^n defined by the function $g \in G, \mu \in \mathbb{R}$, and $p_{g(x) \leq \mu} \iff g(x) \leq \mu$. \neg , negates the predicate, \vee, \wedge joins the predicates by boolean or and and relations, and $U_{[a,b]}$ is the until operator. With this, the semantics of STL is defined as: [MN04]

$$\begin{aligned} s[t] \models \top &\iff \top \\ s[t] \models p_{g(x) \leq \mu} &\iff g(s(t)) \leq \mu \\ s[t] \models \neg \phi &\iff \neg(s[t] \models \phi) \\ s[t] \models (\phi_1 \wedge \phi_2) &\iff (s[t] \models \phi_1) \wedge (s[t] \models \phi_2) \\ s[t] \models (\phi_1 \vee \phi_2) &\iff (s[t] \models \phi_1) \vee (s[t] \models \phi_2) \\ s[t] \models (\phi_1 U_{[a,b]} \phi_2) &\iff \exists t' \in [t+a, t+b] \text{ s.t. } (s[t'] \models \phi_2) \\ &\quad \wedge (\forall t_1 \in [t, t'], s[t_1] \models \phi_1) \end{aligned}$$

A signal $s \in S$ is said to satisfy STL formula $\phi \iff s[0] \models \phi$. Due to their common appearance, $\top U_{[a,b]} \phi$ is defined to be the *eventually* operator, denoted $\mathbf{F}_{[a,b]} \phi$, and $\neg \mathbf{F}_{[a,b]} \neg \phi$ is defined to be the *globally* operator, denoted $\mathbf{G}_{[a,b]} \phi$. The name *eventually* and *globally* is due to the operator's semantics according to our semantics above, since we can see that:

$$\mathbf{F}_{[a,b]}(\phi) \iff \exists t' \in [t+a, t+b] \text{ s.t. } (s[t'] \models \phi)$$

and

$$\mathbf{G}_{[a,b]}(\phi) \iff \forall t' \in [t+a, t+b], \neg(s[t'] \models \neg \phi) \iff \forall t' \in [t+a, t+b], s[t'] \models \phi$$

Finally, $p_{g(x) > \mu}$ can be trivially extended to our syntax, defined as $\neg p_{g(x) \leq \mu}$.

2.2.2 Parametric Signal Temporal Logic

A PSTL formula is an extension to a STL formula where all the time bounds $[a, b]$ in temporal operators and associated constants μ are replaced by free parameters [ADMN12]. The parameters for $[a, b]$ and μ are called time and space parameters respectively. Note that if ψ is a PSTL formula, then every parameter assignment θ induces a corresponding STL formula ϕ_θ , where all the space and time parameters of ψ is fixed according to θ . We refer this process as a valuation of ψ . Finally, we note that the parameter space for a PSTL formula ψ with no additional restrictions is infinite, as there are infinite possible tuples $[a, b]$ for time parameters where a, b is in domain \mathbb{R}^+ .

2.2.3 PSTL Primitives in IoT Environments

To characterize the device interaction rules in an IoT environment, the PSTL formulas can be expanded the include the following two types of primitives below[BVP⁺16].

First-Level Primitives Let $S \subset \mathbb{R}^n$ be a set of signals with $n \geq 1$, the first level-primitives is defined as

$$P_1 = \{\mathbf{F}_{[a,b]}(x_i \sim \mu) \text{ or } \mathbf{G}_{[a,b]}(x_i \sim \mu) \mid i \in \{1, \dots, n\}, \sim \in \{\leq, >\}\}$$

where a, b, μ being the parameters. The space of parameters is defined to be

$\Theta_1 = \mathbb{R} \times \{(\tau_1, \tau_2) \mid \tau_1 < \tau_2, \tau_1, \tau_2 \in \mathbb{R}^+\}$. The meaning of first level primitives can be directly translated from the definition of their respective temporal operators. With $\mathbf{F}_{[a,b]}(x_i \sim \mu)$ meaning $x_i \sim \mu$ happens at least once in $[a, b]$, and $\mathbf{G}_{[a,b]}(x_i \sim \mu)$ meaning $x_i \sim \mu$ happens for all times in $[a, b]$ from semantics in 2.2.1.

Second-Level Primitives Let $S \subset \mathbb{R}^n$ be a set of signals with $n \geq 1$, the second level-primitives is defined as

$$P_2 = \{\mathbf{F}_{[a,b]}\mathbf{G}_{[0,c]}(x_i \sim \mu) | i \in \{1, \dots, n\}, \sim \in \{\leq, >\}\}$$

where a, b, c, μ being the parameters. The space of parameters is defined to be

$\Theta_1 = \mathbb{R} \times \{(\tau_1, \tau_2) | \tau_1 < \tau_2, \tau_1, \tau_2 \in \mathbb{R}^+\} \times \mathbb{R}^+$. The meaning of this primitive can be translated as $x_i \sim \mu$ of duration c must be performed with its start time in $[a, b]$ deriving from semantics in 2.2.1.

Note that the proposed PTSL primitives are not the only possible formations, it is possible to have other combinations of \mathbf{F}, \mathbf{G} for second-level primitives, and it is possible to create higher level primitives. For example, one possible second-level primitive we can construct primitives from \mathbf{FG} primitives is \mathbf{GF} . Similarly to conversion with in First-Level primitives through manipulation of the semantics in 2.2.1, where we have $\mathbf{FG}(\phi) = \neg\mathbf{GF}(\neg\phi)$. And intuitively from the semantics in 2.2.1, $\mathbf{G}_{[a,b]}\mathbf{F}_{[0,c]}(\phi)$ would mean for every interval of size c with start time in $[a, b]$, ϕ is true for at least once.

However, the cases for such primitives may specify to more complicated relationships that are not as helpful in an IoT environment. We do note that while \mathbf{GF} primitives are not directly used in describing our IoT relationships, it is used to check the validity for the negation of \mathbf{FG} rules in our runtime monitor in section 4.

2.3 Boolean expressions and SMT Formulas

A boolean expression is a combination of boolean variables, operators AND (\wedge), OR (\vee), NOT (\neg) and parenthesis. Furthermore, for boolean clauses ϕ_1, ϕ_2 , the operators and parenthesis joins the boolean variables with the following semantics:

- $\phi_1 \wedge \phi_2$ is True if both ϕ_1, ϕ_2 are True, False otherwise
- $\phi_1 \vee \phi_2$ is False if both ϕ_1, ϕ_2 are False, True otherwise
- $\neg\phi_1$ is True if ϕ_1 is False, True otherwise.

Then, a Satisfiability Modulo Theory (SMT) formula can be seen as a expansion of boolean expression with the addition of predicates, which are binary-valued function of non-binary variables such as linear inequalities (ex. $3x > 4$). As a result, each SMT is an combinations of boolean variables and predicates joined by the operators AND, OR, NOT, and parenthesis. In the scope of our research project, we will transform learned device interaction rules in an IoT environment into SMT formulas with boolean expressions and predicates with integer variables. The transformed SMT formulas are then used to detect existence of conflicts within interaction rules through a SMT solver in section 5.

2.4 Powerset and Cartesian Products

Given a set S , the powerset $P(S)$ of S is the subset of all subsets of S , including the empty set and S itself. For example, let $s = \{x, y, z\}$, we have $P(S) = \{\{\}, \{x\}, \{y\}, \{z\}, \{x, y\}, \{y, z\}, \{x, z\}, \{x, y, z\}\}$.

Given sets S_1, S_2, \dots, S_n , a cartesian product $S_1 \otimes S_2 \dots \otimes S_n$ is a set of n -tuples defined as follows:

$$\{(x_1, \dots, x_n), x_1 \in S_1, x_2 \in S_2, \dots, x_n \in S_n\}$$

Both set theory ideas are used in deriving all possible rule combinations that can cause conflicts in our checking tool described in section 5.

3 Learning Rules with Decision Tree

3.1 Multi-Class Classification Problem (MCCP)

In order to learn rules specifying device interaction and behaviors with a decision tree, we would need to find a way to separate data traces produced by a system with desired property for the rule from other traces. In an IoT environment context, the behavior can be categorized into multiple classes that each representing the state value the device for the rule should be. Thus, specifying the behavior of a device when rule ψ is satisfied can be formulated into a MCCP, formalized below.

MCCP Let $C = \{C_1, \dots, C_n\}$ be the set of classes, let s^i be an n dimensional signal and let $l^i \in C$ be its label. Find a formula ϕ for class $C_i, i \in \{1, \dots, n\}$ such that the Misclassification Rate (MCR) is minimized, where $|\cdot|$ represents set cardinality, N the total number of signal label pairs, and

$$MCR(\phi) := \frac{|\{s^i | (s^i \models \phi \wedge l^i \neq C_i) \vee (s^i \not\models \phi \wedge l^i = C_i)\}|}{N} \quad (1)$$

3.2 STLTree

As described in 2.2, a significant portion of device interaction rules in an IoT environment is time sensitive, which we aim to characterize using the PSTL rules defined in the previous section. Thus, we constructed STLTree, a decision tree model that learns device interactions with PSTL rules to solve MCCP using environment's device state change logs. The main structure of STLTree is based on the approach for data classification described in [BVP⁺16], and a detail description to all the modifications we made to the framework in section 3.2.7.

3.2.1 Model Overview

The decision tree in our implementation is binary to accommodate for the binary nature of $\leq, >$ operators we use in defining the PSTL primitives. Without loss of generality, we have set the left subtree to be the true branch and right subtree false branch. The implementation of STLTree is illustrated in the algorithm below.

```

1    //ϕ: The path representing PSTL formula upto current split
2    //S: Label set to split on, h: Current depth
3    //Returns a subtree that splits S
4    buildTree(ϕ, S, h):
5        if stopCondition(ϕ, S, h):
6            return leaf(argmaxc∈C prob(S, c, ϕ), 1-maxc∈Cprob(S, c, ϕ))
7        ϕ* = findBestPartition(S)
8        t = Node(ϕ*)
9        newpath = ϕ* if ϕ != null else ϕ ∧ ϕ*
10       ST*, SF* = partition(S, newpath)
11       t.left = buildTree(newpath, ST*, h+1)
12       t.right = buildTree(newpath, SF*, h+1)
13       return t

```

Since learning the optimal decision tree is a NP complete problem [HR76], our model restricts the total depth and least number of data traces to split on with **stopCondition** (details in 3.2.2), and performs a greedy growing algorithm with **findBestPartition** (details in 3.2.3). If the stop condition is satisfied, we return a leaf node with the corresponding predicted class that has the highest probability to satisfy each individual traces in S . Intuitively, it is set to be the class with the most labels in the set, defined below:

$$\text{argmax}_{c \in C} \text{prob}(S, c, \phi) = \text{argmax}_{c \in C} \frac{|\{(s^i, l^i) | l^i = c\}|}{|S|} \quad (2)$$

Otherwise, we find the best PSTL primitive ϕ^* to split our dataset S with line 7, construct a node t representing ϕ^* , and continues to recursively grow the tree from the split on ϕ^* and the grown path $\phi \wedge \phi^*$ (details for splitting S through

checking rule satisfaction is in 3.2.4). We add in the special case where ϕ is null for when we construct the tree from the root, since at the beginning there is no split path yet. The splitted S_T^*, S_F^* represents data satisfying the true and false branch of ϕ^* correspondingly, specifically:

$$S_T^* = \{(s^i, l^i) \in S | S^i \models \phi^*\}, \quad S_F^* = \{(s^i, l^i) \in S | S^i \not\models \phi^*\} \quad (3)$$

From the algorithm, a decision tree can be learned by executing *buildTree*(*null*, *S*, 0), where *S* is the starting set of labeled signal traces. Note that each device has different classes corresponding to their state values; therefore, we would need to build a separate tree for classifying each individual device in the environment to learn all the device interaction rules. Thus, STLTree builds a tree for classifying each device in the environment with the exception of devices with real valued states such as a thermostat, since there will be too many states to classify, all with not many interesting interactions associated.

3.2.2 Stop Condition

Several stop criteria can be set for *buildTree* in 3.2.1. While it is straightforward to split until each node only contains signals from a signal class, such stopping conditions results in large trees that takes long to train and are very prone to overfitting. In STLTree, the stop condition is fairly restrictive. We chose to stop training once a large majority of signals belong to the same class or the number of signals for each node is less than a certain threshold. Furthermore, while IoT environments can be complex with vast amount of device interactions, it is reasonable to assume each individual interaction to be fairly simple. That is, we should not have a very long PSTL rule that associate with tens of devices with each other. Thus, for our training of STLTree, we also limited the height of the tree to be at most 5 to avoid overfitting and overcomplication.

3.2.3 Finding the Best Partition

In order to find the PSTL rule that gives the "best" partition, STLTree first defines best according to the Information Gain (IG) measure, defined below: [Qui14]

$$IG(S, \{S_T, S_F\}) = H(S) - \left(\frac{|S_T|}{|S|} H(S_T) + \frac{|S_F|}{|S|} H(S_F) \right) \quad (4)$$

where H to be the entropy measure with $prob(S, c, \phi)$ defined as in 3.2.1:

$$H(S) = - \sum_{c \in C} prob(S, c, \phi) \log(prob(S, c, \phi)) \quad (5)$$

As a result, each time STLTree finds the PSTL rule split that gives the greatest information gain or, in other words, minimizes the split entropy H .

Now, we have the **findBestPartition** as follows. In the algorithm, line 7 bounds the search space for time and space parameters as described above and Line 8 constructs the PSTLprimitive by specifying the type of the primitive, what the relation is, and the time and space parameter restrictions. Note by our definition of PSTL, there are a total of 6 combinations to construct a primitive, and for each primitive, we compute the parameters for PSTL rule ϕ that gives the least entropy defined in equation (5). We compute the parameters through either iterating through all possible assignments or through a random walk process called simulated annealing when the problem space for parameter assignments gets too large(details in 3.2.7). Finally, as remarked in the beginning of this section, the PSTL rule ϕ with the least entropy upon split will give the greatest IG as defined in equation (4), which we will use to split our set S through checking rule satisfaction methods for *buildTree* in 3.2.1 (details in 3.2.4).

```

1    //S: Label set to split on
2    //Returns the PSTL primitive to split S on
3    findBestPartition(S):
4        minEntropy = math.Inf
5         $\phi^* = \top$ 
6        for ptype in [F, G, FG]:
7            for sim in [≤, >]:
8                timebound, spacebound = boundParams(S)
9                p = PSTLprimitive(ptype, sim, timebound, spacebound)
```



```

10         entropy,  $\phi$  = FindParameters(S, p)
11         if entropy < minEntropy:
12             minEntropy = entropy
13              $\phi^* = \phi$ 
14         return  $\phi^*$ 

```

3.2.4 Checking Rule Satisfaction

In order to split our data when growing the decision tree and to evaluate data using our learned model, we would also need to check whether a data trace satisfies the respective PSTL rule efficiently. We check whether a data trace is valid for each type of PSTL primitive as follows:

1. $F_{[a,b]}(x \sim \mu)$. To check the **F** rule, we would just need $x \sim \mu$ to be true for at least 1 occurrence in $[a, b]$. Thus, if \sim is \leq , we can simply compare the minimum value v for the signal dimension corresponding to x in the interval $[a, b]$ with μ . If $v \leq \mu$, we know $x \sim \mu$ is true. Similarly, if \sim is $>$, we can compare maximum value m for x in $[a, b]$ to see if $m > \mu$.
2. $G_{[a,b]}(x \sim \mu)$. To check the **G** rule, note that by 2.2.1, we can convert $G_{[a,b]}(x \leq \mu)$ into $\neg F_{[a,b]}(x > \mu)$ and $G_{[a,b]}(x > \mu)$ into $\neg F_{[a,b]}(x \leq \mu)$ and we can check whether the corresponding **F** rule is not satisfied instead.
3. $F_{[a,b]}G_{[0,c]}(x \sim \mu)$ To check the **FG** rule, we would need $x \sim \mu$ to be true for c time stamps for at least 1 occurrence in $[a, b]$. We check the rule by computing min max or max min filters with window size c . If \sim is \leq , for each window i of length c starting within $[a, b]$, we compute the maximum value m_i for signal dimension corresponding with μ , and we return $m = \min_i m_i$ for the minimum window. If $x \leq \mu$, by our construction, the window m represents must have all value satisfying $x \leq \mu$ and thus satisfying our rule. Similarly if \sim is $>$, we use a max min filter by finding minimum value v_i over all windows and computing $v = \max_i v_i$ to compare with μ .

3.2.5 Inference with Tree Model

Inferencing data using our learned model is straightforward. Given a data trace for a device, we recursively evaluate the data trace on the decision tree learned for the device. Starting from root, if the data trace satisfies the PSTL rule for the node, we traverse to the true branch of the tree corresponding to S_T^* in 3.2.1, and otherwise to false branch corresponding to S_F^* . We stop until a leaf node is reached and we output the class the leaf node predicts. A prediction conflict occurs when the output prediction and the actual state for the device in data trace does not match.

3.2.6 Visualizing Device Interaction Rules

Converting the decision tree into their corresponding learned PSTL rules can be done by simply traversing down the tree from root, and joins each PSTL primitive each node along the path represents. STLTree adapts to the algorithm to convert such rules by [BVP⁺16], with details shown below.

```

1    //root - root node
2    //Returns a dictionary mapping device class to associated rules
3    Tree2STL(root):
4        ruledict = {}
5        //t - subtree's root node
6        //phi - PSTL rule respective to current path
7        Recurse(t, phi):
8            if t is a leaf with class  $C_i$ :
9                if t.error > THRESHOLD:
10                    return
11                if i in ruledict's keys:
12                    ruledict[i] = ruledict[i]  $\vee$   $\phi$ 
13                else:

```

```

14         ruledict[i] =  $\phi$ 
15     return
16      $\phi^* = t.PSTLprimitive$ 
17     Recurse(t.left,  $\phi \wedge \phi^*$ )
18     Recurse(t.right,  $\phi \wedge \neg\phi^*$ )
19     return
20 Recurse(root,  $\top$ )
21 return ruledict

```

Note the check on line 9 makes sure that only the device interaction rules with high confidence will be reported. We enter the inner recursive helper with start primitive \top for the case where our entire tree is a leaf node and all of our data traces will be predicted to the class corresponding to the leaf. In this case, we will fall into the case in line 11, which will result in the rule for predicting that class to be \top as desired (Since \top simply means True). An example for the algorithm’s translation is shown in Figure 3 below, with $C1, C2$ as classes and $p1, p2$ as PSTL primitives.

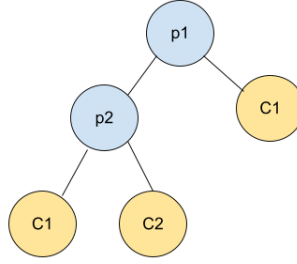


Figure 3: Example translation, the above tree translates to a dictionary mapping 1 to $(p1 \wedge p2) \vee (\neg p1)$ and 2 to $(p1 \wedge \neg p2)$

3.2.7 Modifications to the framework

To accommodate for learning rules in an IoT environment setting, we made the following modifications to the decision tree framework described in [BVP⁺16],

listed below.

Handling rules learned due to random events As described in section 2.1, it is possible for our tree model to learn unwanted rules in the environment due to random event noises such as the homeowner manually changing device state. To address such issue, we include an error term for each leaf when training our tree, which is defined to be $1 - \max_{c \in C} \text{prob}(S, c, \phi)$, with $\text{prob}(S, c, \phi)$ being the proportion of traces that class c satisfy in the dataset corresponding to the leaf as described in equation (2) of section 3.2.1. Then, we added an error threshold constant so that only rules with error less than the constant will be visualized as described in 3.2.6 and only prediction conflicts for rules with error less than the threshold will be reported when evaluating as described in 3.2.5. These rules represents the environment interaction our tree model is most confident about, and prediction conflicts with regards to these rules are the most likely the potentially dangerous behaviors that resulted from anomalies in the environment violating the device interactions.

Preprocessing data In section 2.2.1, signals are defined to be continuous-time, continuous valued functions, which [BVP⁺16]’s framework is also targeting towards. However, in data traces for an IoT environment device states, device change data is formed by logs recording discrete time stamps and device state values corresponding to the change (described in 2.1.3 for Samsung Smarthings). The values for device state change themselves in some cases are discrete as well (Ex. A light switch can only be on or off).

To handle the discrete nature for timestamps in state change logs, given a log of device states in an IoT environment, we first set the trace with smallest time stamp to timestamp 1, and increment other traces with time stamp relative to the smallest time stamp. For STLTree, we chose to be the difference in seconds since an IoT hub would typically need ~ 1 second to handle device events, making smaller time units unnecessary.

Each trace is defined to be a small fixed interval basing on timestamps and we generate such interval based on a gap g . That is, if data trace for interval i starts at timestamp t , the data trace for interval $i + 1$ would start at timestamp $t + g$. For the timestamps on the device change log, we would mark the device corresponding to the state with the value of the state change. Other devices that have not changed state at that time stamp would be set to Null. Now, to make sure we have enough timestamp to generate an interval for each logged device change, we would also fill in dummy traces (all device state value set to Null) for g seconds before each device change timestamp, if it is not already there. Finally, we would backfill the traces by putting device states with value None to be the last known state it has before this timestamp, since we know there are no state changes from the last timestamp for the device.

Now, for the devices with state values that are not real numbers (for example, a light switch can only be "on" or "off"), we assume that there are only a finite number of states the device can be in. Then, we would iterate through all data traces in the raw data to give each device state a class number, and create an dictionary mapping the device state to its corresponding class number. We change these device states to the class number it corresponds to instead of its actual state name. An example of the data processing process is portrayed in Figure 4, 5 below.

Finally, for some devices, it is significantly more likely for them to be in one state over others (For example, Smoke Alarm should be off most of the time). To avoid the complication of our decision tree model overlooking certain device interactions due to lack of significance in data for a certain state value of the device, the data for each device with different state value labels that has appeared in the dataset is kept below a fixed ratio r . For our STLTree, r is set to be 10, so that there is at least a 1 : 10 ratio among data for different values of the same device state.

Timestamp,	Virtual Switch 2_switch,	Virtual Switch1_switch,	Smoke Alarm_alarm,	Door_lock
1	on	off	off	unlocked
2	Null	on	Null	locked
Trace,	Virtual Switch 2_switch,	Virtual Switch1_switch,	Smoke Alarm_alarm,	Door_lock
1	1	0	1	1
2	1	1	1	0

Figure 4: Example data handling, Top: Before, Bottom: After

```
{'Virtual Switch 2_switch': {0: 'off', 1: 'on'},
'Virtual Switch1_switch': {0: 'off', 1: 'on'},
'Smoke Alarm_alarm': {0: 'both', 1: 'off', 2: 'siren'},
'Door_lock': {0: 'locked', 1: 'unlocked'}}
```

Figure 5: Example dictionary mapping states to class number for Figure 4

Impurity Measures In the original model proposed by [BVP⁺16], the measure that determines the best partition rule is based on an Information Gain (IG) on robustness measurement, which not only measures the degree of misclassification of the split, but also the extent of misclassification (Details in section 9.1). However, such measurement would not make sense in the context of discrete valued device states. As a result, we would use the robustness measure only on the finding split rules for continuous valued states. For discrete valued state splits, we used normal IG as measurement instead, shown in equation (4). After obtaining the best partition rule for each device state, we then calculate the normal IG of the best split rules for continuous valued states from robustness measurement to compare each rule, and using the rule with the best IG to split our tree.

Restricting search space for PSTL parameters As shown in 2.2.2, the search space for first level and second level primitive parameters is infinite, as there are infinite possible time parameter tuples $[a, b]$. Thus, we restrict our parameter search as follows:

1. Bounding the time parameter. We first iterated through all the data traces and computed the shortest time interval $[\tau_0, \tau_1]$ that contains all the data traces. As a result, τ_0 would be the smallest starting time stamp of all data traces, and τ_1 would be the largest ending time stamp. Note any time interval outside of $[\tau_0, \tau_1]$ is not useful since none of our data traces would contain information about that time. As a result, we restrict picking the time parameter $[a, b]$ to be within the range $[\tau_0, \tau_1]$. For second level primitives, we pick c within the range $(0, \tau_1 - b]$

2. Bounding the space parameter. Similarly to bounding the time parameter, we iterated through all the data traces. For each dimension of the signal that does not have real value states, we count the number of different class labels it has, and we would restrict our space parameter μ to not exceed the number. By a straightforward translation, $x_i \leq \mu$ would mean x_i is in one of the first μ class labels, similarly translation applies for $x_i > \mu$. Trivially, such μ can be restricted to only integers. In our case for IoT environments, the dimension of the signal are the devices in the environment and the class labels would be what each device's current state values map to as described in 3.2.2. For real valued dimensions, μ is simply restricted to integers bounded by the lowest and uppermost value the signal achieves in the data trace. We round real values to integers under the assumption that in a household environment, rounding rules to the nearest whole number is precise enough for the device interactions.
3. Simulated Annealing (SA). After bounding the time and space parameters, the search space for time and space parameters could still be fairly large for the decision tree to explore all possible PSTL rules. For example, there are already $O(|\tau_1 - \tau_0|^2)$ ways to pick the space parameter $[a, b]$. To speed up our training process, STLTree utilizes a simulated annealing algorithm in picking the PSTL rule for splitting. The algorithm performs a random walk on selecting the PSTL parameters that will converge at a local maximum in IG for splitting [Ing96] (details in section 9.2). In STLTree, the model first computes an upper bound on the total possible parameter assignments for each PSTL formula, and utilizes SA if the total assignments exceeds an user defined threshold. Otherwise, STLTree would iterate through all possible parameter assignments to find the best split.

Tree pruning As described in 3.2.2, since almost all device interaction rules in an IoT environment should be fairly simple, our learned decision tree is very susceptible to overfit. To reduce the overfitting behavior, we conduct an optional pruning on the learned tree model as follows:

1. Set a improve threshold, so that if for a split we have neither the left subtree error or the right subtree error gets reduced by more than the threshold, the split is considered futile. For futile splits, we can prune them by making the parent node a leaf and chopping out its children. We start from the parents of leaves and work upwards toward the root.
2. After pruning above, we run a cross validation on the new tree model. Then, we continue to chop off leaves of the tree until we only have the root node. The chopping order of leaves is determined by the effective alpha of parent nodes, which we have the leaf parent with the least effective alpha to be chopped first (details in section 9.3). Each time we chop off a leaf, we store the current tree model in a dictionary and run validation on the same validation set. Finally, the tree model with the least validation error is returned.

Translating PSTL rules to a readable format After translating the tree into a dictionary mapping device states to PSTL rules in 3.2.6, we translated each PSTL rule into a more readable text format for user friendliness, with instruction shown below:

1. $F[a, b](x \sim \mu)$: Between seconds a to b , x becomes $\sim \mu$
2. $G[a, b](x \sim \mu)$: From seconds a to b , x is always $\sim \mu$
3. $F[a, b]G[0, c](x \sim \mu)$: From seconds a to b , x becomes $\sim \mu$ for at least c consecutive seconds
4. $\neg F[a, b]G[0, c](x \sim \mu)$: In anytime from a to b , x becomes $(\neg \sim)\mu$ within c seconds

where we define $(\neg \sim)$ to be the following: $(\neg \leq)$ being $>$, $(\neg >)$ being \leq . For the $\neg \mathbf{F}$ and $\neg \mathbf{G}$ rules, note that $G[a, b](x \sim \mu) = \neg F[a, b](x(\neg \sim)\mu)$, we can simply translate $\neg F[a, b](x \sim \mu)$ to $G[a, b](x(\neg \sim)\mu)$ and $\neg G[a, b](x \sim \mu)$ to $F[a, b](x(\neg \sim)\mu)$. Finally, \wedge and \vee are translated directly to or and and, respectively. An example translated rule output is shown in Figure 6 below.

3.3 TreeNoSTL

While a significant portion of device interactions in an IoT environment is time sensitive, there are also interactions that happens immediately when the conditions are satisfied. To detect such interaction rules that are not time sensitive at all, we constructed TreeNoSTL, a decision tree that learns immediate device interaction rules to solve MCCP with the environment's device state change logs.

```

Device: LightA
State: OFF
Under Condition:
  1. From seconds 3 to 8, Thermostat is always > 60 and
     From 0 to 5, AC becomes > 1 for at least 3 seconds and
     Between seconds 8 to 9, LightB becomes >= 1
or 2. In anytime from 0 to 5, AC becomes <= 1 within 1 seconds and
     From seconds 8 to 9, LightB is always < 1

```

Figure 6: Example rule output after text translation for STLTree

3.3.1 Model Overview

TreeNoSTL model trains a decision tree using Python’s sklearn tree module [PVG⁺11]. Similarly to STLTree, TreeNoSTL learns a binary tree and restricts the complexity of the rules learned with a stop condition on the maximum depth of the tree to avoid overfitting. Instead of finding the best rule to split our data and grow out the tree with SA, TreeNoSTL finds the best rule to split on directly by preprocessing data log into discrete valued traces (details in 3.3.2). Furthermore, TreeNoSTL utilizes Gini Gain (GG) instead of IG as measure for evaluating the quality of a rule split, defined below: [Qui14].

$$GG(S, \{S_T, S_F\}) = Gini(S) - \left(\frac{|S_T|}{|S|} Gini(S_T) + \frac{|S_F|}{|S|} Gini(S_F) \right) \quad (6)$$

where S is our dataset, S_T the data traces that satisfies our rule, and S_F the data traces that does not. Also, we have

$$Gini(S) = \sum_{c \in C} prob(S, c, \phi) (1 - prob(S, c, \phi)) \quad (7)$$

where C is the set of possible classes for a device that is obtained from the device’s possible states (details in 3.3.2). ϕ is the rule we split on, and $prob(S, c, \phi)$ is defined the same as in STLTree described in 3.2.1 equation (2). Similar to how we have trained for STLTree, we learn a decision tree for each device that has non real value states.

3.3.2 Preprocessing Data

Similarly to data processing for STLTree in 3.2.7, we first generate our dataset from device state change logs in environment through a relative timestamp and backfilling unknown states (unknown states are marked Null). Since we would like to learn the immediate device changes within the environment, for any time stamp i , we are only interested in the interaction between timestamp i and $i + 1$. As a result, we form a data trace by combining adjacent rows on a data, if their corresponding timestamps are also adjacent and we generate dummy rows for singleton rows and backfill the device states to what they are last known. For a device with non-real value in the data, we simply combine the rows by modifying each entry of the data into the format *statebefore_stateafter*. Similarly to the preprocessing done in STLTree, the states between different *statebefore_stateafter* tuples is kept under the threshold $r = 10$.

For a real numbered device, due to the difficulty for decision trees in handling continuous valued data, we separate the device state into fixed value intervals and classify the state values based on the lowest possible value their interval can achieve. The fixed interval for each continued valued data is provided from the user as a dictionary.

The lowest starting point of the fixed value interval is determined by the largest integer lower than the lowest value the device achieves, and we stop adding intervals once the interval corresponding to the largest values includes the largest value the device achieves in the log. For example, for a thermostat with lowest value 0 and fixed intervals of 5, a temperature of 84 will be classified in interval 80, and a temperature of 85 will be classified in interval 85.

While such processing loses information regarding the real valued devices, the model’s accuracy can be improved by reducing the interval size. The interval sizes for each continuous device state tuple will be stored in a dictionary for future use. We also note that the number of intervals would never grow too large, as there is often limitations in the values a device can achieve (For example, a thermostat could probably only reach -20 to 120 degrees F.). An example of our data after processing is shown in Figure 7 below.

As a result of this data processing, the set of possible classes C is obtained through all possible combinations of *statebefore_stateafter* in the dataset, and we will be splitting on rules based on whether a device has a specific *statebefore_stateafter* value. Finally, sklearn’s tree model only supports states that are integer valued; therefore, we used sklearn’s OneHotEncoder to convert all the state and classes to integers states, with which we learn our decision tree and convert the state back to how we have formulated above [PVG⁺11] (details in 9.4).

Timestamp,	Thermostat_temperature,	Virtual Switch1_switch,	Smoke Alarm_alarm,	Door_lock
1	84	off	off	unlocked
2	85	on	Null	Null
Trace,	Thermostat_temperature,	Virtual Switch1_switch,	Smoke Alarm_alarm,	Door_lock
1	80_85	off_on	off_off	unlocked_unlocked

Figure 7: Example data handling, Top: Before, Bottom: After

3.3.3 Evaluation with Tree Model

Evaluating our learned tree model can simply be done using sklearn tree model's *predict_proba* function, which returns the probabilities for each class the device is in. We output the class with the highest probability; however, similar to what is done in STLTree, only prediction conflicts within rules that our decision tree is confident about will be considered, which is controlled by a threshold constant.

3.3.4 Visualizing Device Interaction Rules

Visualizing the device interaction rules for TreeNoSTL can be done in a similar fashion as described in 3.2.7, using the rule splits learned for our model instead of PSTL primitives. We traverse down the tree from root and joins each rule along the path. We would also return a dictionary that maps each class to the device interaction rules our tree has learned and has a confidence level passing our threshold as described in 3.3.3.

```
Rules:
Virtual Switch1's switch state stays off, under:
  0. Thermostat's temperature state is not 60 or changes from 60 and
    Thermostat's temperature state stays 65 and
    Door's lock state is not unlocked or changes from unlocked
or
  1. Thermostat's temperature state stays 60 and
    Door's lock state is not locked or changes from locked
```

Figure 8: Example rule output after text translation for TreeNoSTL

Finally, to convert the *statebefore_stateafter* into a more readable text format, we used the following translation below:

1. $valA_valA$: Device's state stays $valA$
2. $\neg valA_valA$: Device's state is not $valA$ or changes from $valA$
3. $valA_valB$: Device's state changes from $valA$ to $valB$
4. $\neg valA_valB$: Device's state does not change from $valA$ to $valB$

Finally, we join rule splits along the same path with *and*, and join different paths with *or* for the same class. An example for rule translation result is shown in Figure 8 above.

3.4 Evaluations

To evaluate the performance of our decision tree models, we generated 5 household IoT environments with fixed rules that we would like our tree models to learn. For the first 2 household IoT environments, we generated the training data through a state machine that follows our set fixed rules. For the last 3 environments, we created the environment and interactions through Samsung Smartthings devices and Smartapps, and we generated the training data through automated testing on Samsung Smartthings' simulation platform using Selenium [Smac] [Aut].

3.4.1 Environments Using State Machine

To generate an IoT environment using state machines, we first set a dictionary of transition rules with a 5-tuple of $(InEvent, InState, OutState, OutEvent, TimeDelay)$, which represents an occurrence of *InEvent* at *InState* will

results in *OutState* and trigger the *OutEvent* after *TimeDelay* timestamps. An example transition rule is shown in figure 9 below. In the figure, an *InState* of None means the event can be triggered from any state.

```
(Temp_Chg_H,      None,    T_High,    ThermoGT85,    0),
(ThermoGT85,      T_High,   T_High,    SmokeSiren,     5),
```

Figure 9: Example rule. A high temperature of greater than 85 degrees triggers the smoke alarm siren

An event can also occur randomly at each timestamp depending on the device that the event is associated with to accommodate for users' random behavior in physical control of devices as described in 2.1. We generate the environment device state data through a timer object, a counter of the timestamps that performs the random triggering of events as well as stores the events that need to be triggered after a time delay. At each timestamp, device objects in the state machine perform the state changes as specified by the events generated from the timer, and their final states for the timestamp are recorded as data trace.

3.4.2 Environments Using Smartthings Simulation

To generate an IoT environment on the Samsung Smartthings hub, we defined our device interaction rules through Samsung Smartapps. After installing the app on the Smart devices, we are able to virtually simulate the interactions on the Smartthings hub through a sidebar of buttons as shown in figure 10 below. We can change a device state through clicking the tiles representing the devices, and we can physically trigger an app run through the *trigger now* button.

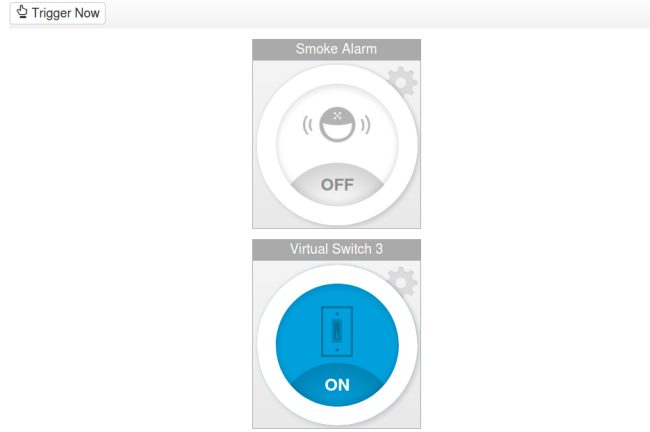


Figure 10: Samsung Smartthings simulation

We generate the data traces Selenium automation on this simulation. At each timestamp, the automation can trigger events randomly by clicking on the device tiles similarly to how we did with state machines. To obtain the event informations on Samsung Smartthings, we created a monitor Smartapp to request for the device change logs stored on the hub as described in section 2.1.1. The data log is then processed as described in 3.2.7 and 3.3.2 for the two models.

In a real life Samsung household environment, the training data would be derived in a similar manner by directly requesting state change information from the hub and only looking at timestamps at a fixed time interval before. We finally note, from section 2.1.3, due to Samsung hub can only store a maximum of 200 state changes for each device, the training data in real life would need to be obtained in a quite frequent manner.

3.4.3 Results

For each state machine environment, we trained our model on 10000 data traces, and for each Samsung Smartthings environment, we trained our model on 3000 data traces (less due to limitations on Samsung Hub in storing states changes, as described in 3.2.3). For STLTree, we processed our data into 10 second intervals to look for device interactions. For both models, we restricted the maximum depth to be 4 and we stop splitting once we reached an accuracy of 0.90. As a result, we would also only record the learned rules with confidence more than 0.90. For STLTree, the training time for state machine environment is about 8 hours and for Smartthings environment is about 4 hours. For TreeNoSTL, the model is able to terminate within minutes.

Out of 7 time related rules and 5 regular rules for the environments generated by state machines, our STLTree is able to learn 5 out of 7 of the rules and TreeNoSTL is able to learn 5 out of 5 rules. Out of 10 time related rules and 3 regular

rules for Samsung Smartthings environment, our STLTree is able to learn 8 out of the 10 rules and TreeNoSTL is able to learn 3 out of 3 rules.

Successfully Learned Rules In terms of accuracy for our models, we note that we have defined *successfully* learning a rule to be that the model have learned a rule that closely resembles of the rule’s desired behavior. It is normal to have minor discrepancies between learned rules to our designed rules in the environment due to both limitations in training data size and the difference in perceptions between the model and human brain. For all of the successfully learned rules, our model is able to learn rules with a fairly close resemblance, as shown by the example in figure 11 and figure 12 below.

```
or 3. From seconds 7 to 8, Smoke Alarm_alarm is always < 2 and
    In anytime from 3 to 5, Virtual Switch1_switch becomes >= 1 within 4 seconds and
    Between seconds 8 to 9, Smoke Alarm_alarm becomes <= 1 and
    Between seconds 8 to 9, Smoke Alarm_alarm becomes >= 2(Error rate: 0.0)
```

Figure 11: Successful time related rule: Switch 1 on leads to Switch 2 on after 3 seconds

```
Smoke Alarm's alarm state stays off, under:
0. Virtual Switch 2's switch state stays off and
   Door's lock state is not unlocked or changes from unlocked and
   Door's lock state did not change from locked to unlocked(error: 0.00588235294117645)
or
1. Virtual Switch 2's switch state stays off and
   Door's lock state is not unlocked or changes from unlocked and
   Door's lock state change from locked to unlocked(error: 0.06666666666666665)
or
2. Virtual Switch 2's switch state stays off and
   Door's lock state stays unlocked and
   Virtual Switch1's switch state stays on(error: 0.018018018018018056)
```

Figure 12: Successful regular rule: Switch 2 off leads to Smoke Alarm off

As shown in line 2 for the rule learned in figure 11 (1 means on and 0 means off for switches), while we did not directly learn that switch 2 turns on after 3 seconds when switch 1 turns on, if switch 1 is on at 3 seconds before at second 7, the rule on the line will be satisfied. In both figures, there are additional rule restrictions for device interactions that we did not design our environment to have; however, this is also normal due to the limitations in training data size and side effects from other interactions. For example, in figure 11, the additional restrictions regarding the smoke alarm resulted from the interaction rule that switch 1 on leads to smoke alarm siren.

Unsuccessfully Learned Rules In our evaluation, we have only encountered failure in learning time related rules. This is as expected as time related rules are significantly more difficult to learn, especially with the vast search space in PSTL parameters and the limitation in our approximation algorithm with Simulated Annealing (SA) as described in 3.2.3. For 3 out of 4 failures, STLTree failed to learn the rule completely, which we believe they are mainly resulted by the noise in random device state changes as described in 2.1. Such failures can be reduced by training our model on a greater dataset size.

```
or 3. Between seconds 1 to 6, Door_lock becomes <= 0 and
    In anytime from 0 to 5, Virtual Switch 2_switch becomes < 1 within 3 seconds and
    Between seconds 6 to 9, Virtual Switch 2_switch becomes >= 1(Error rate: 0.03296703296703296)
```

Figure 13: Failure in capturing interaction in entirety

The remaining failure showed our model’s limitation in capturing interactions in its entirety. In our environment, we have set the rules to be

1. Switch 2 off for 1 second —> Switch 1 on
2. Switch 2 on —> Switch 2 off after 1 second

which we have Switch 2 turns on leads to Switch 2 turns off directly. As shown in figure 13 below, our model only learned Switch 2 on leads to Switch 1 on after 2 seconds while ignoring the abnormal circular interaction defined by rule 2. This failure could be a result of our splitting algorithm when learning the tree, since our learned rule is correct and can be directly inferred by the two rules above. We will leave resolving such complications in rule learning for our future work.

Learning Unintended Rules Despite our effort in recording rules with a high confidence only, we have learned a total of 31 rules out of the 25 rules defined in our data environments. The learning of these unintended rules, as described in 2.1, is also significantly due to the noise in random device state changes and can be reduced in training our model under a greater dataset size.

4 STLMon: Runtime Monitoring of Smartthings Environments

After learning the rules and device interactions of Samsung Smartthings environments with STLTree and TreeNoSTL introduced in the previous section, we created STLMon, a runtime monitor on Smartthings environments to check for any violations of its interaction and user defined security rules in real time. To build our monitor, we first created a parser that converts the learned rules from our decisiontree models and additional rules defined by the user for the monitor to store. Then, we built our monitor structure with an http server that communicates with the Samsung Smartthings Hub to both access device change informations in the environment and issue commands to the environment if violations occur. Finally, we designed algorithms for our monitor to check for these potential rule violations and evaluated our monitor using Smartthings Environment simulations. The details for each part of the implementation is described in the sections below.

4.1 Parsing Device Interaction Rules

Since both handling of data and learned rules are different in STLTree and TreeNoSTL, the rules learned from the two models are parsed differently into two types of dictionaries. Similar to what we did in rule visualization for the decision tree models, only learned rules with accuracy exceeding a set threshold will be considered for our monitor. For user defined rules, we have limited each rule to be specified under a fixed language of keywords for ease of parsing. The user is able to specify additional time related and immediate rules, with time related rules being parsed in the STLTree dictionaries and immediate rules in TreeNoSTL.

4.1.1 Parsing STLTree Rules

We parse our temporal rules in STLTree in a similar way as how we printed our learned rules defined in 3.2.6 by traversing and joining PSTL primitives. Instead of simply printing out the PSTL primitives, we instead process each primitive into a 6 tuple in the order as follows:

1. The *deviceName_deviceState* tuple associated with the primitive.
2. The primitive type ($F/G/FG/GF^*$).
3. The inequality associated with primitive.
4. The time interval associated with primitive. A triple of (From last x timestamps from now, To last y timestamps from now, duration of event). The value of x, y is obtained through the size of interval i we used for learning the rule, which, intuitively, for a time parameter of $[a, b]$, $x = i - a, y = i - b$. The duration of event is only useful for secondary primitive FG , and it is assigned to the duration parameter learned.
5. The state values that satisfies the primitive. This is obtained using the classdict generated in data processing described in 3.2.7. For the space parameter μ learned in the primitive for discrete variables, we simply include all the possible state values for the device in the classdict that satisfies the primitive's inequality with respect to μ . (or does not satisfy if we are currently on the False branch). If the primitive is corresponding to a continuous variable, the item would be a singleton for μ and we would check the validity through the inequality specified.
6. The timestamp unit associated with learning the primitive (Ex. Seconds/Minutes). Since every primitive and rule is learned with the same timestamp unit by our data processing, the timestamp unit for each rule would always be the same for each primitive.

For the primitives on the False branch of the decision tree, we utilize the STL rule property described in 2.2.1 that $\neg F(\neg\phi) = G(\phi)$, we can check $\neg F(\phi)$ by creating a new primitive specifying $G(\neg\phi)$. Similarly we check $\neg G(\phi)$ with $F(\neg\phi)$. Finally, we can check $FG(\phi)$ by $GF(\neg\phi)$. Thus, while GF rules are not learned in STLTree, we would still need an algorithm to check conditions for such primitives in our monitors.

Finally, we note that by the process defined in 3.2.6, each primitive is a node of the tree, and rules are formed by joining primitives. We thus represent each rule as a list of the tuples, and joining is simply done by elongating the list. After processing the rules, we generate two dictionaries corresponding to the "DONT" and "DO" rules of the environment, which we adapted the names from work done by [IoT]. The details for the two dictionaries are listed below.

DONT Rule Dictionary We would first form a dictionary for the "DONT" rules, which are defined to be that if the conditions for the rule is satisfied, the device state corresponding to the rule should not change to a state value that is different from what is specified. This dictionary is simply generated through mapping the device state tuple for each

learned tree to a dictionary mapping each state value to a list of the processed primitive rules classifying the device at that state value. An example for the DONT rule dictionary is shown in figure 14. Also, we note STLTree learns rules for discrete valued devices only, there would only be DONT rules for discrete valued devices.

```
ruledict = {'Door_lock': {'locked': [[('Virtual Switch 2_switch', 'F', '<=', (7, 4, -1), ['off'], 'seconds'),
                                     ('Thermostat_temperature', 'F', '<=', (9, 2, -1), ['72'], 'seconds')],
                                [('Virtual Switch 2_switch', 'G', '>', (6, 4, -1), ['on'], 'seconds')],
                                ],
            'unlocked': []},
```

Figure 14: Example DONT rule dictionary, there are two rules for the "Door_lock" to be locked

DO Rule Dictionary From the DONT rule dictionary, we are able to build the dictionary for DO rules, which are defined to be that if the conditions for the rule is satisfied, the device state corresponding to the rule should change to the state value specified in the near future. Since satisfying each primitive can lead to an entire rule the primitive is a part of to be satisfied in the case that the primitive is the only missing condition, for each device change we encounter, we would want to check if any rule containing primitives satisfied by the device change is satisfied.

Thus, we form our DO rule dictionary by iterating through each primitive in all rules contained in the DONT dictionary, for each primitive, we update our entries in our deeply layered DO rule dictionary in the following manner. For primitive p in rule ϕ that specifies $deviceA_stateA$ to be v , say p is specified for $deviceB_stateB$ with satisfying list l , we would map $deviceB_stateB$ to a dictionary mapping each μ in l to rule dictionary $Dict_r$. In $Dict_r$, we map $deviceA_stateA$ to a dictionary mapping v to a list containing ϕ (Append to the list if there are already rules being mapped, create a singleton list if first time).

For continuous variables, we construct μ by appending the inequality after the singleton value specified for the primitive. As an example, the converted DO dictionary for DONT dictionary in figure 14 is shown below.

```
{'Virtual Switch 2_switch':
  {'on':
    {'Door_lock': {'locked': [[('Virtual Switch 2_switch', 'G', '>', (6, 4, -1), ['on'], 'seconds')]]}},
  'off':
    {'Door_lock': {'locked': [[('Virtual Switch 2_switch', 'F', '<=', (7, 4, -1), ['off'], 'seconds'),
                                ('Thermostat_temperature', 'F', '<=', (9, 2, -1), ['72'], 'seconds')]]}}},
  'Thermostat_temperature':
    {'72 <=':
      {'Door_lock': {'locked': [[('Virtual Switch 2_switch', 'F', '<=', (7, 4, -1), ['off'], 'seconds'),
                                ('Thermostat_temperature', 'F', '<=', (9, 2, -1), ['72'], 'seconds')]]}}}
```

Figure 15: Conversion of DO dict from figure 14

We finally note that for temporal rules it is possible to have multiple primitives within each rule specifying the same $deviceA_stateA$ with value v in the satisfying list with different STL operators and time intervals. To avoid duplication of rules in our dictionary, only the first occurrence of primitive specifying each $(deviceA_stateA, v)$ in each rule is added according to the method above.

4.1.2 Parsing TreeNoSTL Rules

Similarly to parsing STLTree rules, the rules for TreeNoSTL is also parsed into "DO" and "DONT" dictionaries. First, we processed the learned immediate rules through the tree traversal method in 3.3.4 into a list of 6-tuples (converting each node in the tree to a 6-tuple in the process). However, since the immediate rules have an emphasis on transition of state values rather than the classification of state value at the exact timestamp as in temporal rules, the 6-tuple is processed differently in the order as follows.

1. The $deviceName_deviceState$ tuple associated with the node.
2. The start state corresponding to the node.
3. The end state corresponding to the node.
4. Boolean value on whether the node is corresponding to a state change, found through checking whether start and end state are the same.

5. Boolean value on whether we are on the false branch.
6. The timestamp unit for the rule. Since every primitive and rule is learned with the same timestamp unit by our data processing, the timestamp unit for each rule would always be the same for each primitive.

We note that the class labels for immediate rules, as described in 3.3.2, is in the format of *statebefore_stateafter*. As a result, the second and third item in the tuple can be obtained by simply pattern matching. For continuous variables, we note that TreeNoSTL processed them into fixed value discrete intervals, which can be processed in the exact same way with our saved dictionary for each continuous variable interval in 3.3.2. Now, we generate our DO and DONT rule in a similar way as in STLTree, described below.

DONT Rule Dictionary In the context of immediate rules, a DONT rule can be defined as the device state should not perform a transition that does not match the transition specified (or at all if the rule does not specify a state change) when the conditions for the rule are satisfied. The DONT dictionary is generated in a similar way as STLTree with a simpler structure, where we map each device_state tuple to a dictionary mapping a state change key tuple (*statebefore, stateafter*) to a list of rules associated with the state change. An example DONT rule dictionary is shown in figure 16 below. Similarly to STLTree, TreeNoSTL learns rules for discrete valued devices only, and thus there would only be DONT rules for discrete valued devices.

```
{'Door_lock': {
  ('unlocked', 'locked'):
    [[('Virtual Switch 3_switch', 'off', 'on', True, False, 'seconds'),
      ('Virtual Switch 3_switch', 'on', 'off', True, False, 'seconds')]],
  ('locked', 'unlocked'): [[('Virtual Switch 3_switch', 'off', 'on', True, True, 'seconds')]]}
}
```

Figure 16: Example DONT rule, there are two rules for the "Door_lock" basing on thermostat temperature

DO Rule Dictionary Similarly, the immediate DO rules specifies the state change specified by the rule should happen in the near future if all the conditions are satisfied. We first note that intuitively, DO rules should only be triggered through state changes that makes the rule conditions to be satisfied by our definition, thus we can only consider nodes that correspond to a state change, or on the False branch for a non state change node.

From the DONT rule dictionary, we construct our deeply layered DO rule dictionary in a similar way. For each node p in rule ϕ that satisfies our precondition and specifies that *deviceA_stateA* to have a transition from v to v' and p specifies *deviceB_stateB* to have a transition from μ to μ' , we map *deviceB_stateB* to a dictionary mapping key triple (*falsebranch?*, μ , μ') to rule dictionary *Dict_r*. In *Dict_r*, we map *deviceA_stateA* to a dictionary mapping (v, v') to a list of rules containing ϕ (Append or create a singleton list if needed). As an example, the converted DO dictionary for DONT dictionary in figure 16 is shown below.

```
{'Virtual Switch 3_switch': {
  (False, 'off', 'on'):
    {'Door_lock': {'unlocked', 'locked':
      [[('Virtual Switch 3_switch', 'off', 'on', True, False, 'seconds'),
        ('Virtual Switch 3_switch', 'on', 'off', True, False, 'seconds')]]}},
  (False, 'on', 'off'):
    {'Door_lock': {'unlocked', 'locked':
      [[('Virtual Switch 3_switch', 'off', 'on', True, False, 'seconds'),
        ('Virtual Switch 3_switch', 'on', 'off', True, False, 'seconds')]]}},
  (True, 'off', 'on'):
    {'Door_lock': {'locked', 'unlocked': [[('Virtual Switch 3_switch', 'off', 'on', True, True, 'seconds')]]}}
}
```

Figure 17: Conversion of DO dict from figure 16

4.1.3 Parsing User Rules

Aside from the monitoring the environment under rules learned from STLTree and TreeNoSTL, we have also provided a language for users to additionally specify their desired security rules within the environment, both temporal and immediate. For temporal rules, the syntax for the rule language is specified under a combination of keywords, with grammar listed below.

- {*Specification*} WHEN {*Condition*} AND/OR {*Condition*} ...

In *Specification*, we specify the device and state value corresponding to the rule in the format of

- THE $\{device_state\}$ tuple IS $\{value\}$ (AFTER $\{time\}$ SECONDS/MINUTES)

where the parenthesis part specifying time is optional. If time is specified, the state change is assumed to happen within the next second/minute. Since both STLTree and TreeNoSTL learn rules for discrete valued device states only, we also only allow discrete valued rules in the *Specification*.

Each rule can have multiple conditions separated by AND/OR relation. When parsing our language, AND has a higher precedence than OR. For each *Condition*, the condition for the rule to be satisfied is in the format of

- $\{state\}$ OF $\{device\}$ $\{Restriction\}$

where in *Restriction* we specify the rule corresponding to their temporal logic, listed below:

- IS $\{value\}$ FOR $\{time\}$ SECONDS/MINUTES – G rule
- BECOME $\{value\}$ (IN LAST $\{time\}$ SECONDS/MINUTES) – F rule
- BECOME $\{value\}$ IN LAST $\{time\}$ SECONDS/MINUTES FOR $\{time\}$ SECONDS/MINUTES – FG rule

For discrete valued states, $\{value\}$ is simply be the value v , while for continuous states, $\{value\}$ can be specified through

- GREATER/LESS/GREATER EQUAL/LESS EQUAL THAN $\{v\}$

which corresponds to $<, >, \geq, \leq$ in temporal logic inequalities. For discrete variables, the *value* can only be of one single value rather than a list of possible values. If the rule works under specifications of multiple values of the state, we would currently need to specify an additional condition in the rule joined by OR.

The user rules will then be parsed in the same way as the temporal rules for STLTree, with each condition being parsed into the 6-tuple specified in 4.1.1 in the order as follows:

- The *deviceName_deviceState* tuple: obtained through combining $\{state\}$ and $\{device\}$ in the $\{Conditions\}$ section of rule.
- Primitive type: obtained through pattern matching the $\{Restriction\}$ section.
- Inequality with primitive: For discrete valued states, the field is irrelevant. For continuous valued states, obtained through pattern matching the $\{value\}$ field for restriction as specified above.
- Time interval associated: We note that this is a triple of (From last x timestamps from now, To last y timestamps from now, duration of event). The duration of event is simply obtained by the time value after "FOR" in the FG rule restriction (irrelevant for other cases). We then determine y , which is default to be 0 unless specified in the $\{Specification\}$ behind the keyword AFTER. Then, x is defined to be $y + t$, where t is the time specified in the restriction after FOR in G rule and after 'IN LAST' for F rule. For user rules, we default timestamp unit to be seconds, so that rule containing other time unit specification (Ex. Minutes) will be converted to seconds for this term.
- State values satisfying the primitive: Specified by the $\{value\}$ in the $\{Restriction\}$
- Timestamp unit: By our method of processing time intervals, this is always set to seconds.

Since conditions are joined by AND/OR keywords for each rule and AND has a higher parse precedence, each rule is then parsed into a list l of the converted 6-tuple lists. Each list of converted 6-tuple lists appear in AND relation and each individual 6-tuples within the same list appears in OR relation. In other words, for a rule condition to be satisfied, at least one 6-tuple of each list in l needs to be satisfied. Thus, we can form a list of rules for the described *Specification* with our provided conditions from each individual user provided rule by taking a Cartesian product of each 6-tuple list within the list l , since satisfying any element of the Cartesian product implies our rule is satisfied by this OR relation.

With this process, we form a DONT rule dict that has the same structure as described in 4.1.1 for user defined rules, which maps *device_state* tuples to a dictionary mapping *values* to a list of rules converted from the Cartesian product method. Since the dictionary has the same structure as in 4.1.1, the DO dictionary is converted in the same manner. An example of such rule conversion is shown in figure 18 below.

```

'''THE Door_lock IS unlocked AFTER 3 SECONDS WHEN
    alarm OF Smoke Alarm IS siren FOR 5 SECONDS OR switch OF Virtual Switch 2 BECOME on IN LAST 3 SECONDS
    AND temperature OF thermostat BECOME GREATER THAN 75 IN LAST 5 SECONDS FOR 2 SECONDS'''

[('Door_lock', 'unlocked',
  [('Smoke Alarm_alarm', 'G', '=', (8, 3, -1), ['siren'], 'seconds'),
   ('thermostat_temperature', 'FG', '>', (8, 3, 2), ['75'], 'seconds'])],
 ('Door_lock', 'unlocked',
  [('Virtual Switch 2_switch', 'F', '=', (6, 3, -1), ['on'], 'seconds'),
   ('thermostat_temperature', 'FG', '>', (8, 3, 2), ['75'], 'seconds'])])

```

Figure 18: Conversion of user defined rule. Above: rule, Below: converted list used to construct the DONT dict after Cartesian product method

We note only the time restraint for F rule can be optional, and it will default to becoming in last second. Allowing such syntax would make us able to specify immediate rules without setting up a separate rule for parsing, since immediate transitions essentially means it will transition within the last second by how we trained our TreeNoSTL. We specify a transition of device2's state2 to value2 implies device1's state1 transition to value1 through the following:

- THE $\{device1_state1\ tuple\}$ IS $\{value1\}$ WHEN $\{state2\}$ OF $\{device2\}$ BECOME $\{value2\}$

Rules specified in this way will be translated to the 6-tuple in the exact manner as above, with rule to be F and the time interval tuple being $(1, 0, -1)$ (From last 1 second).

4.2 Monitor Structure

As described in 2.1.3, in order to access the events from devices and issue state change commands once violation is detected, our monitor would need to directly communicate with the Samsung Smartthings Hub. Thus, aside from a backend that stores the device changes and rule dictionaries, our monitor is constructed through a public http server receiving and sending requests from Samsung Smartthings, and through a Smartapp on the Smartthings Hub side with Oauth communication as described in 2.1.1. The monitor structure is illustrated in figure 19 below.

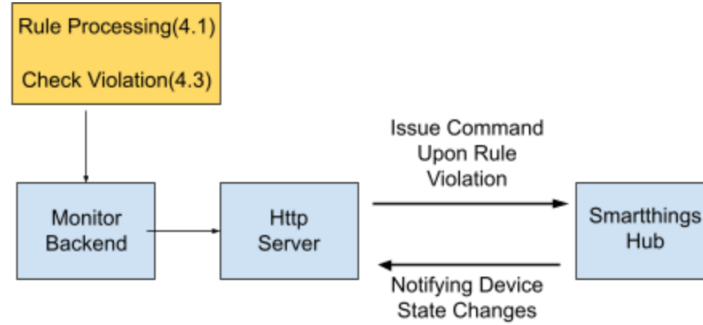


Figure 19: Runtime Monitor Structure

4.2.1 Monitor Backend

In order for our monitor to check for rule violations, it not only needed to store the DO and DONT dictionaries for both temporal and immediate rules that we have created in section 4.1, it would also need to keep an internal record of device state changes of the environment. To check temporal rules, we would need to store not only the last known state value for each device, but a list of device changes that has happened since it is possible for a device to experience multiple state changes in a time interval. Our monitor uses a fixed size list to store the state changes, and when the maximum number for the list is reached, we would pop out the earliest state change in the record. Each state changes is stored as a tuple of a tuple (*time of change, value of change*) in the list.

The state changes for each device is stored in a dictionary, which maps device to a dictionary mapping each of its state to its state change lists. We would need to use the state change lists in order to check for rule violations, which is detailed in section 4.3. Finally, to handle continuous variables for immediate rules, our monitor would also need to access the continuous variable dictionary we created in 3.3.2.

4.2.2 Communicating with Smartthings Hub

Similar to how we obtained device change logs when generating data on Smartthings platform, our monitor would need to communicate with the Hub through a Smartapp. As described in 2.1.1, Smartapps has the ability to issue state changes and access past logs for all the devices it subscribe to. Thus, monitoring the devices in the environment can simply be done through subscribing all the devices to the monitor Smartapp. When a device change occurred in the environment, our Smartapp then sends a request to our server with the monitor regarding the state change with parameters in the order as follows:

1. Time of the state change.
2. Device name for the change.
3. Device state for the change.
4. State value for the change.

To receive these requests from Samsung Smartthings Hub, our monitor would need a public http server. Also, it would need to know the aforementioned Smartapp APIKey and APIEndpoint to issue commands to the Smartapp through OAuth. Aside from the necessary structure for handling requests, the server also have a scheduler object which helps our monitor in handling commands for DO rules, described in the section below.

4.2.3 Issuing Command to Devices

When our monitor receive the device state change information from our Smartapp, the monitor would first check for DONT rules for both immediate and temporal rules in the environments to see if any conflicts is detected. If a conflict is detected, the http server would directly send to the Smartapp the correct state for the device should be according to the DONT rule, which the Smartapp executes the state change through devicehandler specified function calls to change the state to the correct value as described in 2.1.2. We note DONT rules would only apply to discrete valued devices.

If no DONT rule is violated by the state change, our monitor would register the state change to its corresponding state change list in the backend record and then check for DO rule violations. For each DO rule r with condition being satisfied, we would know by our method in 4.3 the time t for the amount of time we need to wait for the device to change state according to the DO rule (Since temporal DO rules will mostly not happen immediately and for immediate rules, $t = 0$). We would then send the device, corresponding state value, and t to the http server's scheduler to schedule sending the event regarding the state change in after $t + 1$ timestamp unit. We add an additional buffer on the time t since the DO rule is only violated if the desired state change did not happen after t , and we allow 1 timestamp unit duration for the change to happen. The timestamp unit is determined by the 6-tuple for the rule primitives as described in 4.1.

After time $t + 1$ timestamp unit has passed, the scheduler conduct a final check of the state change regarding rule r before it is sent to Samsung Smartapp to check for whether there is additional state changes that invalidates the conditions that rule r needs, and whether the desired state change for rule r is already issued. If none of the checks are true, the DO rule is violated and the scheduler would issue the state change command to the Smartapp similar to DONT rule violations. In our monitor, the scheduler for DO rule checks is implemented through python's `apscheduler` package [Gr1].

It is possible to have race conditions in the scheduler due to direct conflicts within DO rules, which we could have commands that change the same device to different states at the same time due to the wait time difference for rules. To avoid such race conditions, all the device change commands for the same device at the same timestamp unit is handled altogether sequentially with the final check method above, with only the first scheduled device change being sent to the Smartthings Hub.

Finally, we note that the devicehandler specified state change functions may not have identical names as the values we would like the device to change as described in 2.1.1. As a result, the monitor included a dictionary of mapping common household device capabilities such as Switch and Lock to their corresponding value change functions to send commands. The dictionary is obtained manually through Samsung Smartthing's capability reference site [Smaa], and additional capabilities that is not included can be found on the same reference site and manually added to the monitor. We would like to include the complete list of capabilities on the reference site to our monitor for future work.

4.3 Checking Rule Violations

From our construction, for each relevant state change event happening in the environment, we would need to check whether the change violate any DONT rules and whether the change would trigger any DO rule condition to be satisfied for both temporal and immediate rules. We first note that each device change event is passed from the Hub to our monitor backend under the 4-tuple $(currtime, device, state, value)$ as described in 4.2.2. While the algorithm checking the four types of rules are fundamentally similar, we elaborate on each case separately in the sections below.

Since it is not possible to specify a device to be in different state values at the same time, in the case of conflicts within the rules as defined by our notion in Definition 2 in section 1.2, we prioritize the immediate rule over temporal rule. Also, if the two rules are of the same type, the monitor would pick behavior according to the the last rule violated with all conditions satisfied. We would also only check for DO rules for state changes that does not violate any DONT rules. Finally, by our rule handling described in 4.1 and how we have trained STLTree and TreeNoSTL only on classification of discrete state devices, there would not be any DONT rules for continuous variables.

4.3.1 Checking Temporal DONT Rules

Consider a device state change where we have $deviceA_stateA$ change to value v . To check whether any DONT rules is violated, we would iterate through our DONT dictionary $Dict$ to check for rules in set R is satisfied, defined by

$$R = \{rules \in Dict[deviceA_stateA][key], \text{ key} \neq v\}$$

Then, if any rule $r \in R$ is satisfied, we know that r is violated in the state change, since the device should have state value specified by r 's key in the DONT dictionary.

To check for whether a rule is satisfied, we iterate through all the conditions specified by the 6-tuple in 4.1.1 to check whether they are all satisfied. For the four condition types for F , G , FG and GF rules, without loss of generality for device D with state S in interval $[hi, lo, dur]$ with available states l (Or for continuous variables, l representing values satisfying the inequality), we check if the condition is satisfied through the following:

F rule For F rules, we iterate through the record for all the past state changes for D with state S to find all the state change values that is in l . Then, out of all the satisfying device changes, we note by 4.2.1 they are stored in tuple $(t_{chg}, value)$, which we can use to compute the relative time of the stored change to the current time for our current device state change by computing $diff = currtime - t_{chg}$ and convert it to the timestamp unit associated with the rule. Finally, by how we designed the time interval when we converted the 6-tuple in 4.1.1, we need to check whether $lo \leq diff \leq hi$, and the existence of any state change satisfying this inequality would mean the condition is satisfied.

If the condition is not satisfied after the previous check and the first state change within the interval is not exactly at hi seconds away from current, we would also need to consider the case that the device is already in a valid state upon entering the interval. To do so, we check whether the last state before the first state change within the interval exists in our record, and whether it is in a valid state if it exists. The rule would be satisfied if the check returns True, and not satisfied otherwise.

G rule Checking G rule is very similar to F rules, and the monitor would instead need to make sure the device stays in a valid state value for the entirety of the interval. To find if the rule is satisfied, the monitor verifies whether the device change passes the following checks:

1. Find all state changes that has relative time difference $diff$ to the current state change being in interval $lo \leq diff \leq hi$, if any of the state change has an invalid value not in l , the condition is not satisfied since the valid state does not last the entirety of interval.
2. Find the last state change for D with state S that happened before or at exactly hi timestamps from $currtime$, if it is not a valid state in l , the condition is not satisfied since the device would be at an invalid state immediately after hi in the interval. If we can not find such state change, which may be a result of insufficient capacity of our record in holding device changes or lack of information, we also assume the condition is not satisfied.

FG rule To check FG rules, the monitor would additionally need to take account of the duration dur . The monitor first obtains all the state changes from its record for D with state S that has a relative time difference $diff$ within $lo \leq diff \leq hi$. Then, for each device state change chg , the monitor would first check if chg is a satisfying state change in l , and if it is, the monitor then checks whether there is any invalid changes (state changes with value outside of l) within the next dur seconds relative to the state change time of chg , and return satisfied if none of the invalid

changes occur. The monitor also stores *first_invalid_date* that represents the first state change within the relative time difference interval that is not in a valid state, if any, for our next step check.

If the above check failed, the monitor checks the last state change before the first state change within the interval to see if it exists in the record. If the state change exists, the monitor would then check whether *first_invalid_date* happens at least *dur* time stamps away from *hi* or does not exist at all. Then, the monitor would know that the device is in a valid state for at least *dur* at the beginning and return satisfied. The monitor returns not satisfied otherwise.

GF rule Checking *GF* rules is more complicated as we would need to make sure the device is in a valid state for all of the intervals of size *dur* within the range $[lo, hi]$. We check whether the condition is satisfied through simultaneously iterating the interval lists and device change state list in our record in the algorithm below.

```

1 intervals: The intervals of size dur to check, stored in the order
2           where first interval is farthest away from current time.
3 changes: The state changes in our record
4           (relative time diff, value) pairs
5 CheckGF:
6     last_valid, last_not =  $\infty, \infty$ 
7     intval_idx, chg_idx = 0, 0
8     while intval_idx < len(intervals) and chg_idx < len(changes):
9         diff, value = changes[chg_idx]
10        curlo, curhi = intervals[intval_idx]
11        if diff < curlo:
12            if last_not  $\geq$  curhi and last_not < last_valid:
13                return False
14            else:
15                intval_idx += 1
16        else:
17            if value is valid and last_valid  $\geq$  last_not:
18                last_satisfied = diff
19            elif value is not valid and last_not  $\geq$  last_valid:
20                last_not = diff
21                chg_idx += 1
22        while intval_idx < len(intervals):
23            curlo, curhi = intervals[intval_idx]
24            if not(last_not  $\geq$  curlo and last_not  $\leq$  curhi):
25                return last_valid < last_not
26            intval_idx += 1
27        else:
28            return True

```

The *last_valid* and *last_not* defined on line 6 represents last occurrence of a state change from a not valid value to valid or valid to not valid correspondingly, in terms of relative timestamp difference from current time (being less would mean happened more recently). We iterate the two provided lists simultaneously based on different conditions. If condition on line 11 is True, we know that the current analyzed state change occurs after our current interval since it is less time stamps away. If that is the case, line 12 checks whether the entire interval is in a not valid value, and return not satisfied if it is True. Otherwise, we know this interval is satisfied, we go check the next interval.

If condition on line 12 is False, we continue to update state changes, since the state changes are still within the interval. Line 18 and Line 20 updates the *last_valid* and *last_not* accordingly. Finally, line 22 checks whether all the intervals have been checked, and if not, we continue to check the intervals to see if they still contain the last invalid state change. If check on line 24 passes, we know the last invalid state change is no longer within the interval before all intervals have

been checked. Then, we finally check if our device is currently in a valid state on line 25 and return True if the check passes. Otherwise, we know all the intervals have been checked and satisfied, and we return satisfied accordingly.

4.3.2 Checking Temporal DO Rules

Consider a device state change where we have $deviceA_stateA$ change to value v . To check for any DO rules has their conditions being satisfied by the device change, we would iterate through our DO dictionary Do to find the rule dictionary $Rule$ containing rules having condition associated with our current device state change. By how we have constructed the DO dictionary in 4.1.1, we have

$$Rule = Do[deviceA_stateA][v]$$

We note by our construction, $Rule$ has the exact same structure as the monitor's DONT dictionary, mapping devices to a dictionary mapping value to its corresponding rules. However, it is possible to have multiple rule dict $Rule$ corresponding to the change in the case of continuous variables, since, for example, a value specified to be > 80 would imply values specified by > 70 to also be satisfied by the current change. Then, for each rule dict $Rule$ corresponding to the current change, the set of rules R for the monitor to check for condition satisfaction can be defined by

$$R = \{rules \in Rule[devB_stateB][key], devB_stateB \in Rule.keys, key \in Rule[devB_stateB].keys\}$$

For DO rules, one additional factor into consideration is that they are most likely not specifying immediate behaviors, and when a device change that corresponding to a condition happens, the condition may not be immediately satisfied. For example, for a F rule condition of $switch$ being on with time interval $[hi, lo, dur]$, we would need to wait at least lo timestamp unit for a state change of turning on a switch to have our rule condition be in the valid time interval to satisfy our condition.

As a result, for each candidate DO rule in R corresponding to our state change, the monitor instead searches for a time t such that after waiting t timestamp unit, all the conditions will be satisfied for the rule. This t is used to schedule sending commands upon DO rule violation through the monitor server's scheduler described in 4.2.3 above. If it is not possible to find such t , we know that the conditions will not be satisfied, and no possible violations will exist for the rule.

To find the wait time t or verify such t does not exist for a rule in R , the monitor checks in two steps. First, similarly to how the DONT rules are checked, the monitor would iterate through each condition i for the rule to find a list of wait time intervals l_i for each condition, so that $\forall t \in [t_{lo}^i, t_{hi}^i], [t_{lo}^i, t_{hi}^i] \in l_i$, the condition is satisfied in the interval.

Before iterating each condition, we first check whether the wait time interval for the condition corresponding to the current change is satisfied. The monitor first assumes the rule is not satisfied before our current change, since otherwise the rule would already be processed by the previous valid change. Now, If the current change condition is corresponding to a F rule with specified time interval $[currhi, currlo, currdur]$, the wait time must be in the $range = [currlo, \infty]$, since waiting less than $currlo$ timestamps would put our current state change outside of the condition specified by interval. FG rule similarly restricts the wait time to be in $range = [currlo + currdur, \infty]$ to account for the duration requirement. For G rules, however, we would have to wait at least $currhi$ times since we would need the condition to be satisfied for the entire interval, and we can wait for as long as possible since the device would always stay in this valid state with our current information (we ignore possible change of states in the future for now), giving us $range = [currhi, \infty]$. Finally, GF rules behaves similar to G rules, but we just need the first satisfying state change to be within dur timestamps from hi , giving $range = [currhi - currdur, \infty]$

As described in 4.1.1, there could be multiple primitives within the same rule that corresponds to the same $deviceA_stateA$ with value v in its satisfying value list, but for different operators and respective time intervals. We note that since we assume there is no state change after the current change, waiting longer would not invalidated any primitives that is already satisfied. Thus, we determine $range$ based on the primitive that would require us wait the longest (having the largest lower bound in range) if multiple primitives for the same device state value occur.

With this information, the wait time intervals $[t_{lo}^i, t_{hi}^i] \in l_i$ for other conditions can be restricted within the interval specified by $range$. For the rest of conditions, the wait time interval for each condition F , G and FG is found in the method described below:

G rule For a G rule with time interval $[hi, lo, dur]$ and $range = [rangelo, rangehi]$, the relevant time frame would be $[max(0, lo - rangehi), max(0, hi - rangelo)]$. Similar in checking for DONT rules, the monitor determines the list of valid time intervals in the following steps:

1. Finding the last state change s_{last} before or at exactly $max(0, hi - rangelo)$ relative timestamps away from current time, check if it is of a valid state. If it is a valid state, add in interval $[rangelo, hi - t_1 - 1]$ to the list. t_1 is determined by the relative timestamps from the next invalid state change after s_{last} within the relevant

time frame. If such t_1 can not be found, the monitor can wait for as long as possible and we add in interval $[rangelo, rangehi]$ instead. If we can not find s_{last} , we skip this step since we have no information.

2. For all the state changes starting from the next state change from s_{last} that is in the relevant time frame, we start an interval upon encountering a valid state change and there is no existing interval being started, and we end the interval upon the next invalid state change encountered. If there is an open interval in the end of the process, we set its endpoint to be $rangehi$ and append it to the satisfying interval list. The algorithm for this process is shown in *findIntervalsG* below.
3. For all the intervals in the satisfying interval list constructed from above, the monitor only keeps intervals with range longer than $hi - lo$, since we would need the device to be in the state value for the entirety of $[lo, hi]$ interval after waiting. For the intervals with range longer than $hi - lo$, we keep the portion of $(startpt, endpt - (hi - lo))$ to satisfy this requirement. The process is done in the call to function *removeInvalid* in *findIntervalsG*

We note by our method, the intervals in the list will not have any overlaps as overlapping valid state changes will be contained in the same interval by step 2.

```

1 relevantStates: state change in record within the relevant time frame
2   (described in step 2 above)
3 findIntervalsG:
4   intervals = [] or  $[(rangelo, hi - t_1)]$  from step 1
5   startpt = -1
6   for (date, value) in relevantStates:
7       if startpt < 0 and value is valid:
8           startpt = hi - relative(date)
9       elif startpt ≥ 0 and value is not valid:
10          endpt = hi - relative(date) - 1 #inclusiveness
11          intervals.append((startpt, endpt))
12          startpt = -1
13   if startpt ≥ 0:
14       intervals.append((startpt, rangehi))
15   return removeInvalid(intervals)

```

F rule For a F rule with time interval $[hi, lo, dur]$ and $range = [rangelo, rangehi]$, the monitor checks validity of the condition, similarly to the method in checking G rule above with relevant time frame $[max(0, lo - rangehi), max(0, hi - rangelo)]$ in the following 2 steps. We would not need to execute step 3 for F rule as there would be no restriction on interval size, since the device state would not need to be valid for the entire $[lo, hi]$ interval after waiting.

1. The monitor would again first find the last state change s_{last} before or exactly at $max(0, hi - rangelo)$ from current time and add in $[rangelo, hi - t_1]$ if it is a valid state. This time, t_1 is determined by the next state change after s_{last} , not necessarily need to be invalid. This may result in overlaps in intervals, which we will handle later. If such t_1 can not be found, the monitor can wait for as long as possible and we add in interval $[rangelo, rangehi]$ instead. If we can not find s_{last} , we skip this step since we have no information.
2. Step 2 behaves identical to the G rule step 2, but this time the interval $startpt$ on line 8 is calculated through $lo - relative(date)$, since the F rule becomes valid upon the state change enters the $[lo, hi]$ interval after waiting rather than need to last an entirety. The endpoint stays the same as $hi - relative(date) - 1$ for a similar reason since we can have the state change to be at any point of the interval.

We note that this method will likely result in overlapping intervals since it is possible to have multiple valid state changes in the interval $[lo, hi]$ after waiting a found valid waittime, and we use a *removeOverlap* method to combine overlapping intervals in the satisfying waittime list to return. To remove the overlaps, we take advantage of the fact that device change tuples in the monitor's record are stored in the order of time for their occurrence. Since in our algorithm we processed the device changes in the same order, the found interval list would already be sorted in terms of their

startpt. Then, *removeOverlap* simply iterates through each stored interval in the list, and combine them if possible, in the process shown below:

```

1 removeOverlap(intervals):
2     result = []
3     currstartpt, currendpt = -1, -1
4     for startpt, endpt in intervals:
5         if startpt < currendpt:
6             if endpt > currendpt:
7                 currendpt = endpt
8         else:
9             result.append((currstartpt, currendpt))
10            currstartpt, currendpt = startpt, endpt
11    result.append((currstartpt, currendpt))
12    return result[1:]

```

In the algorithm, the $[currstartpt, currendpt]$ represents the previous combined interval that our current interval represented by $[startpt, endpt]$ can overlap with, and if *endpt* is farther than *currendpt*, we extend the interval to be $[currstartpt, endpt]$. Otherwise we know the intervals are disjoint, and for the other unchecked intervals, they would also be disjoint due to the sorted nature. As a result, we append $[currstartpt, currendpt]$ to the result list and start a new interval to be checked for overlaps with the rest. Line 11 accounts for the last interval we formed, and Line 12 removes the dummy value $[-1, -1]$ we set in the beginning.

FG rule Checking *FG* rule is essentially checking a *F* rule with taking account of the duration. With time interval $[hi, lo, dur]$, $range = [rangelo, rangehi]$, and relevant time frame $[max(0, lo - rangehi), max(0, hi - rangelo)]$, finding valid intervals for *FG* rule is identical to the algorithm defined for *F*, with a different definition for a valid state change. For *FG* rule, in order for a state change to be valid, it not only need to be of a valid value within the relevant time frame, it would also need to stay in a valid state from the time of the change for *dur*. This check is identical to what the monitor used to check a *FG* rule condition is satisfied in DONT rules, and *startpt* would be $lo - relative(date) + gap$ instead to take account of *dur*. Similarly, the *endpt* calculated would be $hi - relative(date) - gap - 1$ upon encountering an invalid value.

GF rule Checking *GF* rule is more different comparing to the other 3 rules above. For a *GF* rule with time interval $[hi, lo, dur]$ and $range = [rangelo, rangehi]$, we check the validity of the condition with the following 2 steps:

1. The monitor first iterate through the record of device state changes, and create a list of intervals representing the range of relative timestamps away from current time in which the device is in a valid state. This is done simply in an identical way as in the method in *findIntervalsG*, with starting *interval* to be $[], startpt$ and *endpt* defined on line 8 and 10 to be *relative(date)*. We note again that the valid intervals found in this list would appear in the order where the first interval is farthest away from current time.
2. Then, the monitor find the valid waittime intervals by iterating through all possible waittimes *t* within $range = [rangelo, rangehi]$ to see if it satisfies our condition. Similarly to checking DONT rules, the monitor checks whether each waittime satisfies the condition through simultaneously iterating the list of valid intervals found in step 1 and the list of size *dur* intervals we need to check the condition is satisfied. The algorithm is shown below:

```

1 validlist = list of valid intervals found in step 1
2 findIntervalsGF:
3     reslist = [], startpt = -1
4     for t in [rangelo, rangehi]:
5         intervals = Intervals of size dur, within range [hi - t, lo - t]
6             order where first interval farthest from curTime.
7         intervalidx, valididx = 0, 0

```

```

8         while intervalidx < len(intervals) and valididx < len(validlist) - 1:
9             satisfied = True
10            intlo, inthi = intervals[intervalidx]
11            vallo, valhi = validlist[valididx]
12            if inthi < validlo:
13                valididx += 1
14            else if validhi < intlo:
15                if startpt >= 0:
16                    reslist.append((startpt, t))
17                    startpt = -1
18                satisfied = False
19                break
20            else:
21                intervalidx += 1
22        if satisfied:
23            if intervalidx < len(intervals):
24                checkLastValidInterval()
25            else:
26                startpt = t if startpt < 0 else startpt
27    return reslist

```

The algorithm checks for whether there is an intersection between each interval in the list of size *dur* intervals with a valid interval found in Step1 and if there is, we know there is a valid state in the interval for at least 1 time stamp and we can continue to check the next one specified on line 20. Line 12 and Line 14 checks for the two other cases, with 12 checking for our size *dur* interval appears after our current checked valid interval, which we increment the index to check the next valid interval. Line 14 checks whether current valid interval appears after the size *dur* interval, since both *validlist* and *intervals* are sorted based on farthest from current time, we know that this size *dur* interval is not satisfied since otherwise previous valid intervals would already check this interval with line 20. We terminate the current satisfying interval, if exists, by setting this waittime as an endpoint.

We note by line 8 that we set the last valid interval item as a special case, which we handle by *checkLastValidInterval()*. We check the case similarly to before, with the addition that if there is no overlap between the last valid interval and current checking intervals, the condition is not satisfied for the current waittime. Furthermore, if the last valid interval appears before the current checking interval, since there are no change of states after, any future waittimes will not satisfy our condition either, and we can terminate our entire loop. Similarly, if there is an overlap, we check whether we are still at the first interval and whether the last valid interval has *validlo* = 0 (In other words, the device is currently in a valid state). If that is the case, there is no need to check future waittimes, since they will always be satisfied. We return with our result interval list with the addition of (*startpt*, *offsethi*).

Otherwise, if all the checks have been passed, we start a new satisfying interval if there is not one already on line 26. Also, due to our special case in *checkLastValidInterval()*, the loop will guarantee to terminate before waittime *t* exceeds *hi*, since there would be not any new information in [*lo*, *hi*] interval. This makes sure our loop termination while *rangehi* may be initially set to ∞ .

Finding Waittime for Entire Rule After finding the list of satisfying waittime intervals for each condition of the rule, the monitor would know that the rule is not satisfied if any of the wait time interval list is empty. Otherwise, the monitor would then need to determine whether there exist a waittime that satisfies all conditions. While there may be multiple waittimes satisfying all conditions, the rule should be satisfied immediately when all conditions are met, and thus the monitor would simply need to check for the earliest possible waittime to return for issuing commands described in 4.2.3. This is equivalent to finding the earliest value *t* that is in range of some interval $[t_{lo}^i, t_{hi}^i]$ for each one of the condition interval list *li*. The algorithm for *findWaittime* is shown below:

```

1    n: Number of conditions
2    li: interval list for condition i

```

```

3     findWaittime:
4         endptlist = []
5         for i in range(n):
6             for  $t_{lo}^i, t_{hi}^i$  in  $l^i$ :
7                 endptlist.append(( $t_{lo}^i$ , L))
8                 endptlist.append(( $t_{hi}^i$ , R))
9         sortedlist = sorted(endptlist, compare_fn)
10        count = 0
11        for time, id in sortedlist:
12            if id == L:
13                count += 1
14            else if id == R:
15                count -= 1
16            if count == n:
17                return time
18        return -1
19
20    compare_fn((t0, id1), (t0, id2)): if id1 == L then > else <
21    compare_fn((t0, id1), (t1, id2)): if t0 < t1 then < else >

```

The algorithm takes advantage of the fact that each interval list l^i has no overlaps, so that if we have found an overlap of n after combining all intervals for all the condition interval lists, all the wait times t within the interval can satisfy all of the n conditions. The number of overlap can be found through first sorting the list that we formed through combining all intervals' start and end points for all rules. We sort our list based on time in *compare_fn* and prioritizing start points over end points with the same time due to our rule's inclusive nature. We can then obtain the number of intervals that is overlapped up to the time specified by *time* on line 11 by counting the number of start points we have encountered in our sorted list subtracting the number of endpoints we have encountered.

Since we are interested in the earliest time this is achieved, we return the start point t for the last interval that makes the overlap happen the first time since we can not go earlier than that. Our monitor would then send the rule and t to our monitor's scheduler to send commands in 4.2.3. If we can not find such wait time t , the rule is not satisfied by our current device change.

4.3.3 Checking Immediate Rules

For immediate rules, checking DO and DONT rules are done in identical ways since finding waittime would no longer be an issue. Similar to checking temporal rules, the immediate rules are checked by breaking each individual rule into a list of conditions and check if all the conditions have been satisfied. For a device state change where we have *deviceA_stateA* to change to value v , the set of rules R to be checked for DO and DONT immediate rules are defined identically to their temporal counterparts in 4.3.1 and 4.3.2. However, the conditions specifying for each rule is in a different 6-tuple, defined in 4.1.2 to be

(*device_state*, *startState*, *endState*, *statechange*, *falsebranch*, *timestamp_unit*)

Checking if each condition is met is done under the following algorithm, described below:

```

1     record: The monitor's record of the state changes for condition's device
2     checkCondition:
3         lastChangeTime, lastChangeValue = record[-1]
4         satisfied = False
5         if lastChangeValue == endState:
6             if not statechange:
7                 satisfied = relative(lastChangeTime, currTime) > 1

```

```

8         else:
9             beforeChangeTime, beforeChangeValue = record[-2]
10            satisfied = relative(lastChangeTime, currTime) ≤ 1 and
11                        beforeChangeValue == startState
12        if falsebranch:
13            return !satisfied
14        return satisfied

```

In the algorithm, the *relative* function computes the relative timestamp unit difference between the two parameters, which we would need to be within 1 unit by our definition of immediate. If a condition is corresponding to a state change, then it must have been transitioned to *endState* from *startState* within the last time stamp unit, which is checked in line 10 and 11. Otherwise, the device must stay in the *endState/startState*, which can be checked in line 7 to show it stayed in the state value for more than a timestamp unit. Finally, we return the negation of satisfied if we are on the false branch.

One scenario the algorithm above ignores is when we are unable to find *lastChange* or *beforeChange* on line 3, 9 due to the lack of information in our monitor's record. As a result, we can not infer anything about the condition, and the monitor would return False for either true or false branch the condition corresponds to. If all conditions for a rule are satisfied, the rule is then passed to our monitor server to check for violation and issue commands.

4.4 Evaluation

To evaluate the performance of our monitor, we incorporated a fuzztesting framework to randomly generate device change log input for the monitor to detect rule violations. To begin our log generation, we would first need to manually input a dictionary *input* mapping each *device_state* tuple in the environment to a list of possible values the state may take. For continuous variables such as temperature, the possible values would instead be a list of 2 items, the first being the value's lower bound while the second being upper bound. An example input is shown in Figure 20 below.

```

device_dict = {
    'Door_lock': ['locked', 'unlocked'],
    'Virtual Switch 2_switch': ['on', 'off'],
    'Virtual Switch1_switch': ['on', 'off'],
    'Thermostat_temperature': ['75', '95'],
}

```

Figure 20: Input dictionary for fuzz testing

Then, the generation process takes 2 steps. First, we would generate a random set of rules for the environment; then, we would use the rules to generate a device change log that is likely to reflect generated rule behavior. The details for each process is described in the sections below, and the evaluation result would be discussed in the final section.

4.4.1 Generating Rules

From the rule parsing technique described in 4.1, it is sufficient for our testing framework to only generate the DONT rules for both temporal and immediate behavior, since the DO rules can be directly converted from them.

Our rule generation first takes in a *maxdepth* and *timebound* parameter for the environment, where the *maxdepth* specifies the maximum number of condition per rule in our environment (recall in 4.1 that each rule is a list of conditions either specifying temporal primitives or immediate behavior) and *timebound* specifies the maximum value for *hi* can be in each temporal rule condition time intervals.

Then, for temporal rules, we simply generate each member of the 6-tuple for each condition randomly by picking random associated primitive type, a random satisfying time parameter [*hi*, *lo*, *dur*], a random inequality, and a random integer we use to find satisfying state values with the inequality we picked. For discrete values, the random integer is picked between 0 and the length of the possible value list in our input dictionary *input*, and all the values of indices satisfying the inequality to the integer we picked is kept as valid state. For continuous valued variables, the valid value is picked between the middle 50% of the value range described by the corresponding lower and upper bounds in *input*.

For immediate rules, the start and end state values are picked in a similar way for discrete variables through a random index in possible value list. For continuous variables, we categorize the possible values into 5 equally gapped discrete states and we store the gap for the state in a *gapdict* for generating state change logs.

For each rule we want to generate for the environment, we first randomly select whether the rule would be a temporal rule or an immediate rule. Since we expect that a household environment to have more temporal rules than immediate rules in real life, the number of temporal rules to immediate rules is kept at approximately a 7 : 3 ratio. An example generated rule dictionary for input in Figure 20 is shown below.

```
#Temporal rule:
{'Door_lock': {'locked':
  [[('Virtual Switch1_switch', 'G', '>=', (2, 0, -1), ['off'], 'seconds'), ('Thermostat_temperature', 'F', '<=', (1, 0, -1), ['85'], 'seconds')]],
  'Virtual Switch1_switch': {'off':
    [[('Virtual Switch2_switch', 'F', '>', (4, 1, -1), ['off'], 'seconds'), ('Door_lock', 'FG', '<', (7, 1, 3), ['locked'], 'seconds'),
      ('Thermostat_temperature', 'GF', '>', (2, 0, 1), ['86'], 'seconds'), ('Thermostat_temperature', 'FG', '>=', (4, 3, 1), ['84'], 'seconds')]],
    'Virtual Switch2_switch': {'on': [[('Thermostat_temperature', 'FG', '<=', (9, 2, 2), ['87'], 'seconds')]]}}

#Immediate rule:
{'Virtual Switch2_switch': {'on', 'on'}:
  [[('Thermostat_temperature', '79', '79', False, False, 'seconds'), ('Door_lock', 'locked', 'unlocked', True, False, 'seconds')]],
  'Virtual Switch1_switch': {'off', 'on'}: [[('Thermostat_temperature', '75', '79', True, False, 'seconds')]]}

#Gap dict:
{'Thermostat temperature': 4}
```

Figure 21: Generated rule output

We finally note that it is possible for us to generate rules specifying impossible behavior, such as a device being on and off at the same time interval, due to randomness. In practice, however, such occurrence of rules occurs negligibly and will not have a impact on the accuracy of our evaluation.

4.4.2 Device Change Log Generation

In order for our monitor to check for rule violations for each device change, we would need both information about the current change to be checked and a log of past device state changes to store in the monitor's record. To generate the log of past state changes, we first note that it is difficult to generate device changes that satisfies the conditions of our generate rules from pure randomness. Thus, in order to have a high chance to satisfy conditions for our device interaction rules generated above, we form our device state change logs basing on each individual rule in two steps. First, we pick a random rule and generate a event log based on the conditions of the rule, so that if no other state changes is added to the log, all the conditions of the rule would be satisfied. Then, we add in random device state changes in the log for the randomness in input. As a result, a significant portion of the generated logs would remain not impacted by the added random changes to make sure we have both satisfying and not satisfying test cases for the monitor.

For the current change, we first need to generate the current time *curTime* to be a random time within 24 hours of the time when testing is ran. We recall from Figure 21 and section 4.1 that both DO and DONT temporal dictionaries have a structure mapping *deviceA_stateA* to a dictionary mapping each *valueA* of the state to either the list of rules (for DONT case) or dictionary of rules (for the DO case) corresponding to when *deviceA_stateA* changed value to *valueA*. For immediate rule dictionaries, the structure is similar, mapping *deviceA_stateA* to (*ValueBefore*, *ValueAfter*) value pairs specifying a state or nonstate change. As a result, for temporal rules, the *deviceA_stateA* and *valueA* for the current change is determined by what the rule we have picked to generate change logs corresponds to. For immediate rules, the current change behaves the same with *valueA* determined by *ValueAfter*. The device change generation is done somewhat differently for immediate rules, temporal DONT rules, and temporal DORules, but in a similar manner. The device change generation would need our input device dictionary *input* and the *timebound* and *gapdict* in addition to the generated rule dictionaries from our rule generation above, and our details for generating past device change log and current change for each rule is described in the sections below.

Generate Device Change Log for temporal DONT Rules To generate the current device change from a temporal DONT rule, we would first randomly determine whether we have a violating change or satisfying change for the rule, if all conditions are satisfied. If we determined to have a satisfying change, we would simply return a change at *curTime* of *deviceA_stateA* to *valueA*. Otherwise, we return a change for our device state to change to *valueB* \neq *valueA*, a randomly selected value that *deviceA_stateA* can be.

Now, for generating an environment satisfying the conditions of the rule, we iterate through all the conditions of the rule to generate an device change event to satisfy each condition. For temporal DONT rules, each condition is a

primitive specifying a *device_state*, for each of the primitive, we construct a state change event satisfying the condition as follows (Recall each primitive is in the 6-tuple structure defined in 4.1.1):

1. Determining the time of the state change. For a primitive defined with time interval $[hi, lo, dur]$, we first determine the *lowerbound* for the state change. Note that if we have *F* or *FG* rules, we know the state change must happen at least *lo* timestamps before the current time ($lo + dur$ for *FG* rules). Similarly, for *G* and *GF* rules, the state change must happen at least *hi* timestamps before the current time ($hi - dur$ for *GF* rules). We set *lowerbound* accordingly.

The upper bound for state change can be determined by *timebound*, since we have no rules regarding state changes more than *timebound* away. Thus, the time of the state change will be randomly picked within the interval $[lowerbound, timebound]$ away before *curTime*.

2. Determining the value of the state change. This is simply randomly picked by index from the possible states specified in the primitive for discrete states. For continuous variables, the value is randomly picked within the lowerbound and upperbound specified by our manual inputted dictionary *input*, with a value satisfying the inequality in the primitive.

Finally, we add in random state changes to the log generated by the step above. For each timestamp from *timebound* away to *curTime*, we determine the number of state changes happening through a random generator. For each state change, we simply randomly pick a *device_state* in the environment and change its value to a possible value specified in *input* at the corresponding timestamp.

Generate Device Change Log for temporal DO Rules For generating device change for temporal DO rules, we need to additionally consider the waittime needed for the condition to be satisfied. Thus, we first randomly generate the wait time *t* with *timebound* to be the upperbound for the rule to be satisfied. For the lowerbound of *t*, we note that as described in checking DO rule method in 4.3.2, the $(deviceA_stateA, valueA)$ pair we picked in current change should corresponds to a primitive condition *p* with time interval $[hi, lo, dur]$ in our picked rule. And similarly to determining the range in 4.3.2, we would need to wait at least *lo* for *F* rules, $lo + dur$ for *FG* rules, *hi* for *G* rules, and $hi - dur$ for *GF* rules for *p* to be satisfied. We set this to be the lowerbound in picking *t*.

After picking *t*, we simply adjust the time interval $[hi', lo', dur']$ for each condition in the rule to be $[hi' - t, lo' - t, dur']$ that does not correspond to $(deviceA_stateA, valueA)$ pair, so that after waiting *t* time stamps, the condition will be satisfied with respect to time interval $[hi', lo', dur']$. The rest of the data log and current change would follow identically as the temporal DONT rule case on the newly modified condition intervals.

Generate Device Change Log for immediate Rules Generating immediate rule device change is different than temporal rules in that we also need to consider both the before state and after state, and whether the condition is on the negate branch for each condition to be satisfied. As a result, for each condition in the rule, we generate an environment satisfying the rule in our first step of generating device state change log for each of the 4 cases as follows: (Recall each primitive is in the 6-tuple structure defined in 4.1.2)

1. StateChange: True, Negate Branch: False. In this case, we need to generate two events for the condition, one to make sure the device is changed to the before state before the last timestamp, and one to make sure the device is changed to the after state within the last timestamp. For the before change event, the device can change to this state at any timestamp before the last timestamp, which we pick the time to be a random value within the interval $[1, timebound]$ away before the *curTime* (we set *timebound* as the upperbound for similar reason as temporal rule case). For the after state change, the event would need to happen within 0 or 1 timestamp before *curTime*.
2. StateChange: False, Negate Branch: False. In this case, we simply need one event to make sure the device is changed to the before state before the last timestamp, and we pick the time to be within interval $[1, timebound]$ away before *curTime*.
3. StateChange: True, Negate Branch: True. In this case, we first randomly determine whether the device changes state or not. If the device changes state, we first randomly pick the before state and after state from the available states provided by our *Input* that does not match the states specified by the condition. Then, the event changes corresponding to the before state and after state we picked can be generated identically as in case 1. If the device does not change state, we randomly pick an available state provided by *Input* and generate event change respective to the picked state identically as in case 2.

4. StateChange: False, Negate Branch: True. Similar to the case above, we randomly determine whether device changes state or not. If the device changes state, we randomly pick a before state and after state pair and generate event changes as in case 1. If the device does not change state, we randomly pick a state that is different from the specified state in the condition and generate event change as in case 2.

After the environment satisfying the conditions have been generated, we add in the random state changes at each time stamps identically to what we did when generating temporal DONT and DO rules. However, before the random state changes are added, we also need to make sure that our the *deviceA_stateA* that our picked rule corresponds to has value *valueBefore* before the current change for *valueAfter* happens. As a result, we add in an event changing state value to *valueBefore* to our generated log from step above, with event time to be randomly picked within interval $[1, timebound]$ away before *curTime*. The log is then added with random state change events and paired with the current change to *valueAfter* to send to our monitor to be checked in an identical fashion as for temporal DONT rules and DO rules.

We finally note that for the steps above, we also need to consider the difference between the immediate rules and temporal rules is in the handling of continuous variables. For a continuous value state x specified by a condition in immediate rules, we can obtain the gap g in generating the state intervals for the continuous state in our generated *gapdict* in 4.4.1. From this, we can randomly pick a valid value within the state x in the range $[x, x + g - 1]$.

Example Output An example for generated (Device Change Log, current Change) pair is shown in Figure 22 below. Each device change in the log is specified by a 5-tuple of (timestamps away from *curTime*, change Time, device, state, value), and current change is specified by (*curTime*, device, state, value). We note due to the insertion of random state changes, it is possible for a device to change to the same state value as its current value as shown by the Door's lock state in the figure. Such behavior would not impact our monitor in any aspect.

```
Events Before
(10, '2021-08-19T17:22:54', 'Door', 'lock', 'unlocked'),
(9, '2021-08-19T17:22:55', 'Thermostat', 'temperature', '95'),
(8, '2021-08-19T17:22:56', 'Virtual Switch1', 'switch', 'on'),
(8, '2021-08-19T17:22:56', 'Thermostat', 'temperature', '80'),
(7, '2021-08-19T17:22:57', 'Thermostat', 'temperature', '94'),
(6, '2021-08-19T17:22:58', 'Door', 'lock', 'locked'),
(4, '2021-08-19T17:23:00', 'Door', 'lock', 'locked'),
(2, '2021-08-19T17:23:02', 'Virtual Switch1', 'switch', 'on'),
(1, '2021-08-19T17:23:03', 'Thermostat', 'temperature', '73'),

Currchg: ('2021-08-19T17:23:04', 'Thermostat', 'temperature', 72)
```

Figure 22: Generated log output

4.4.3 Evaluation Results

We ran our fuzz testing on a simple environment with 4 devices, 3 being discrete stated and 1 being continuous shown in Figure 20. From the environment, we randomly generated 10 device interaction DONT rules with *maxdepth* set to 4 and *timebound* set to 10(a combination of temporal and immediate rules as described in 4.4.1). From the DONT rules, we then constructed the DO rules in the environment. For each rule we generated, we then generate 2 (device change log, current change) pairs as described in 4.4.2 corresponding to the rule for the monitor to be checked.

The process above is done 5 times to ensure the robustness of the monitor, and each evaluation of the monitor is timed to ensure the efficiency. In testing, each evaluation on a (Device change log, current change) pair takes less than *0.4ms*, showing our monitor is highly efficient. To test for the accuracy of the monitor, each generated log pairs is manually analyzed with the randomly generated rules to compare with monitor's output. Out of the 268 generated test cases for both temporal and immediate DO and DONT rules, our monitor has accurately detected all the rule violations, showing robustness.

5 STLCheck: Checking Conflicts within Interaction Rules

During our runtime monitoring of Smartthings environments, one particular factor we chose to overlook is the potential for conflicts in our learned and user defined device interaction rules. Recall in definition 1 and 2 of section 1.2, these conflicts are the source of undermining individual environment's security properties. As a result, we have created STLCheck, a tool to check for these conflicts in rule interaction by generating an example device change log exhibiting each conflict if it exists. Then, to track for the actual behaviors of the generated conflicts, we also created a simulator that is able to simulate the generated conflict logs on Samsung Smartthings platform. The details for each part of the implementation is described in the sections below. As a final note, through the parsing algorithm in section 4.1.3, each user defined security rule can be conveniently converted to an PSTL rule, ridding us the need of creating separate conflict checkers for learned and user defined rules to check for security.

5.1 Checking Rule Satisfaction

By our definition above, if a conflict exists, it is necessary and sufficient for our generated device conflict log to satisfy all the rules associated with the conflict at the time of current change. In order to generate this log, we utilized the *z3* SMT solver to derive an environment satisfying all the conditions for each of the rules in near linear time [dMB08] (Details about SMT formulas can be found in section 2.3).

Before generating our log, we would first need information on the devices and the rules that we would like to find conflicts on. Specifically, we would need an input dictionary *input* mapping each *device_state* tuple in the environment to a list of the possible values the state may take if the tuple represents a discrete variable. If the tuple is representing a continuous variable, it will instead be mapped to a 2-item list, with the first item being the variable's lower bound, second being its upper bound. Similarly to the case for fuzz testing described in 4.4, we would also need the *timebound* parameter representing the upper bound for *hi* in temporal rules for each device to restrict the number of timestamps we would need to consider in our log generation, since the device state information before *timebound* will not impact our rule conditions.

In order to derive a state change log that satisfies all the conditions for all of the rules, we would need to know the exact values each *device_state* tuple in the environment needs to be in all the relevant timestamps. As a result, we created a timestamp integer variable d_i to represent the value of *device_state* tuple *d* at timestamp *i* away from current time, *i* ranges from 0 to *timebound* from our restriction above. For discrete valued *d*, each d_i is mapped to an index in the list of available values specified in *input*, and for continuous valued *d*, each d_i is mapped to a value within the range of $[lowerbound_d, upperbound_d]$ from *input*. This also gives us the initial SMT constraints below:

1. $\forall d \in Discrete, 0 \leq d_i \leq len(input[d])$
2. $\forall d \in Continuous, lowerbound_d \leq d_i \leq upperbound_d$

Now, we treat each conditions of the rules as constraints that needs to be satisfied by our d_i 's for each *device_state* *d* and derive a SMT formula for each condition in the methods below.

Temporal Rules For temporal rules, we recall from 4.1 that each condition is represented by a 6-tuple (*device_name*, *type*, *inequality*, *timeInterval*, *satisfyingStates*, *timestamp unit*) corresponding to the primitive. For each type of primitive with *device_state* *d*, *timeInterval* $[hi, lo, dur]$, and satisfyingStates *l*, its respective formula is derived as follows:

1. For *F* rules, we need the device state to be in one of the values in *l* for at least 1 timestamp in the range. This can be expressed through the *or* operator. That is, $Or_{i \in [lo, hi]} (d_i \in l)$, where we define $Or_{\phi_1, \phi_2, \dots, \phi_n} = \phi_1 \vee \phi_2 \dots \vee \phi_n$ for SMT formulas ϕ_i 's. Finally, we note that $d_i \in l$ can be expressed through the SMT formula $Or_{v_i \in l} (d_i = v_i)$ for any specific *i* for discrete variables, and can simply be converted into $d_i (inequality) (l[0])$ for continuous variables by our 6-tuple construction.
2. For *G* rules, we need the device to be in one of the values in *l* for all of the respective timestamps in the range, which can be expressed through the *and* operator with $And_{i \in [lo, hi]} (d_i \in l)$. ($And_{\phi_1, \phi_2, \dots, \phi_n} = \phi_1 \wedge \phi_2 \dots \wedge \phi_n$ similarly to the *Or* case).
3. For *FG* rules, we need the device to be in one of the values in *l* for at least *dur* timestamps in the range, which can be expressed through $Or_{i \in [lo, hi]} And_{j \in [i, i+dur]} (d_j \in l)$.
4. For *GF* rules, we need the device to be in one for the values in *l* for at least 1 timestamp in all of the size *dur* intervals within the range. This can be expressed through $And_{i \in [lo, hi]} Or_{j \in [i, i+dur]} (d_j \in l)$.

Immediate Rules As described in 4.1, each immediate rule condition is also represented by a 6-tuple (*device_state*, *startState*, *endState*, *stateChange*, *Negate*, *timestamp unit*). We handle the SMT formula generation for the immediate rule conditions by separate into two cases based on the *Negate* flag as follows:

1. If the *Negate* flag is set to *False*, we know that the *device_state* *d* immediately changed from *startState* to *endState* before current time. This can be expressed as $d_1 = startState \wedge d_0 = endState$. We note the *stateChange* flag does not affect this, since if *startState* = *endState*, we need the device to be in the *startState* for at least 1 second, which is exactly expressed by our formula above.
2. If the *Negate* flag is set to *True*, we know that transitions at the last second that correspond to all (*start*, *end*) pairs different from (*startState*, *endState*) satisfies our condition. Thus, the condition can be expressed as $Or_{(start,end) \in l'} (d_1 = start \wedge d_0 = end)$ where *l'* is defined as the list of (*start*, *end*) pairs defined above.

Since it is necessary and sufficient for all of the condition of all of the rules to be satisfied at current time for a conflict to occur, our derived environment would need to satisfy both the initial constraints and all of the SMT formulas corresponding to each of the conditions generated from our method above. Formally, define ϕ_{init} to be the set of our initial SMT constraints from our *input* dictionary, and ϕ_{cond} be the set of SMT constraints from the conditions of our rules, a satisfying environment that will result in rule conflict can be represented by the solution to the SMT formula

$$\Phi = (And_{\phi \in \phi_{init}} \phi) \wedge (And_{\phi' \in \phi_{cond}} \phi')$$

If *z3* determined our Φ as unsat, we know that there are no conflicts within the set of rules, since they can never be satisfied at the same time. Otherwise, by our construction, *z3* would return a solution to a satisfying assignment for Φ , which would be a dictionary mapping *d_i*'s for each corresponding *device_state* tuple *d* to there corresponding assignment value (or index of the value list for discrete variables). We note that from how we have defined the *d_i*'s, each *d_i* can be interpreted as the state value *d* at *i* seconds away from current time. As a result, we can generate a log from our solution set as follows:

1. For each device *d*, starting from *i* = *timebound* to *i* = 0, every time the device is not of the same state as its previous timestamp ($d_i \neq d_{i+1}$), we know the device has changed state and generate a state change log at the corresponding timestamp away from current time.
2. For other variable *d_i*'s, there is no state changes needed, and no entries would need to be added to the log.

We note that the *z3* solver would only guarantee the existence of a satisfying formula; that is, the log representing the conflict within rules, if it exists, is not unique. Finally, for clarity, the event log generated from step above is sorted in the order of timestamps away from current time. An example log output for when a rule conflict is detected is shown in Figure 23 below.

```
Rule conflicts for device: Virtual Switch1_switch
1
Value:off
Rule:[('Thermostat_temperature', 'G', '>', (4, 3, -1), ['89'], 'seconds'), ('Virtual Switch 2_switch', 'F', '>', (1, 0, -1), ['off'], 'seconds'),
('Thermostat_temperature', 'FG', '<=', (5, 2, 1), ['90'], 'seconds'), ('Thermostat_temperature', 'G', '>=', (10, 1, -1), ['86'], 'seconds')]

BeforeValue:off, AfterValue:on
Rule:[('Door_lock', 'unlocked', 'unlocked', True, True, 'seconds'), ('Virtual Switch 2_switch', 'off', 'off', True, False, 'seconds'),
('Virtual Switch 2_switch', 'off', 'on', False, True, 'seconds')]

With example violation log:
(10, 'Thermostat', 'temperature', '86'),
(4, 'Thermostat', 'temperature', '90'),
(2, 'Thermostat', 'temperature', '86'),
(1, 'Door', 'lock', 'locked'),
(1, 'Virtual Switch 2', 'switch', 'off'),
(0, 'Door', 'lock', 'unlocked'),
```

Figure 23: Generated log output for conflict between the two rules. Each entry in log is in 4-tuple of (timestamp from current time, device, state, value)

5.2 Checking Conflicts within Environment

With our method in checking rule satisfaction above, we are able to analyze conflicts in an IoT environment through both of its learned and user defined rules. Such check is straightforward and can be done through trying to generate a conflict log specified in 5.1 for all possible rule combinations. We note that only rules specifying for the same

device_state with different values can be in conflict, which means we can only select at most 1 rule for each value in each combination for a fixed *device_state*. We also note that only discrete variables are associated with the rules, that is, continuous variables are only allowed in the conditions, but are not specified by rules themselves by our construction. Now, for a *device_state* d , we find all the possible combinations of temporal and immediate rules in the environment that can be in conflict in the following steps:

1. First, we find the list of all the possible values l_d that d can take through the same input dictionary *input* as described in 5.1. For each value $v_i \in l_d$, we then consider all the possible combinations for temporal rules by creating a set of rules S_i containing all the temporal DONT rules regarding changing d to v_i . We then add in an empty rule for each set, so that for all of the values $v_i \in l$, there is a corresponding set S_i that has at least 1 element *Empty*.
2. Now, since only rules corresponding to different values can be in conflict, a list of all possible combinations of the temporal rules can be obtained through taking a Cartesian product of all the S_i 's (Details of Cartesian product is described in 2.4). Define S_{\otimes} to be the Cartesian product. Then, by our construction, each element $e \in S_{\otimes}$ is of a $len(l_d)$ -tuple, with the i^{th} entry corresponds to the i^{th} possible value of l_d . We treat *Empty* entries in each element e as no rules being selected for the corresponding value and any element.
3. Then, we add in the possible combinations with immediate rules. For each element e in S_{\otimes} , we check for any entries assigned with *Empty* and form a list l_{empty} containing the values in l_d corresponding to these entries. To form a new combination e' , we would then add at least 1 and at most $len(l_{empty})$ (1 for each *Empty* entry) immediate rules to e . Then, for an element $e \in S_{\otimes}$, we can then form set S_e containing all possible e' we can form from e in step 4 below.
4. To form S_e , we first select all the possible entries that we can add an immediate rule to by finding the powerset $P(l_{empty})$ of l_{empty} (Details of powerset is described in 2.4). Then, for each element $p \in P(l_{empty})$, we find a set S'_v for each value v in p containing the set of immediate rules specifying a change of state to value v and we can form a Cartesian product of all of S'_v , denote $S'_{\otimes,p}$. If any of the S'_v is empty, we simply set $S'_{\otimes,p}$ to be empty and skip the element p since there is no immediate rules we can add for the value. Otherwise, for each element $s' \in S'_{\otimes,p}$, we can form a valid new combination through $e' = s' \oplus e$ (adding rules in s' to the corresponding *Empty* entries in e). S_e , then, can be found through $\{p \in P(l_{empty}), s' \in S'_{\otimes,p}, s' \oplus e\}$.
5. Finally, our set of all possible rule combination S can be obtained through $S = S_{\otimes} \cup (\bigcup_{e \in S_{\otimes}} S_e)$. Since we need at least 2 rules to check for conflict, any element in S with at least 2 non-Empty entries is then a valid combination to run our SMT solver described in 5.1.

5.3 Samsung Smartthings Conflict Simulation

After generating all the potential rule conflicts within the IoT system by our conflict checking tool, we are interested in finding out the actual behavior of the conflict in a real life situation. As a result, we constructed a simulation tool that simulates the conflict log generated from checking using Samsung Smartthings Hub's virtual devices. Our simulation is achieved through 2 steps: processing the generated conflict log example locally, and sending the processed log to Samsung Smartthings Hub's to perform the simulation. The details of each step is described below.

Conflict Log Processing Given our generated conflict log by checking tool, we first recall each entry of the rule is a 4-tuple of (*timestamp from current time*, *device*, *state*, *value*). This is not as useful in our simulation, since we can not start from current time and change device states from the past, and we are more interested in the actual commands to change the *device_state* to the specified *value*. As a result, we process our conflict log in the following two steps.

1. Shifting the timeline: We first note that time in our generated conflict log is all relative, since we just need all the state changes in the log to satisfy all the temporal and immediate conditions at some timestamp in the near future to witness the conflict behavior. As a result, we can shift the timeline generated from the conflict log so that the first described event that is t_0 away from current time in the log is happening at current time + 1 timestamp. Then, for each future event t_i away from current time, they can then be shifted to $t_i - t_0 + 1$ timestamp away from current time. Then, from our shifting schema, the conflict would then happen at $t_n + (t_n - t_0 + 1)$ seconds away from current time, where t_n corresponds to the time for last event in the log before our timeline shift.
2. Retrieving state change commands: Obtaining the state change command for each state is identical to what is done in the monitor in section 4.2.3. Our tool reuses the dictionary mapping capabilities to value change

functions as described there to retrieve the device change. One additional factor to consider, however, is that we would need to change continuous variables states in our device log, instead of only the discrete variables in our monitor, that does not have such value change functions. Instead, these continuous variable changes are done through the function `set{StateName}(value)` instead [Smaa].

As a result, our final processed conflict log would be in a list of 3-tuple, of $(shifted\ timestamp, device, command)$.

Simulating Conflict through Samsung Hub After processing our conflict log, we then send the processed log to Samsung Smartthings Hub through adding a simulation feature to our monitor Smartapp used in 4.2.2 through Oauth (details 2.1.1). Upon receiving the processed log, the monitor Smartapp then is able to schedule the events corresponding to the log with the following two steps.

1. Finding devices: Before the monitor schedules to trigger the events, it would first need to find the corresponding device in the environment as specified from *device* in each 3 – tuple of the processed log. This is simply done through iterating all devices the monitor subscribed to in the environment to find a match.
2. Scheduling Events: We utilized the *runIn* feature on Samsung SmartHub Apps to schedule the events with their corresponding shifted timestamps in the future from current time. As described in 2.1.2, the device change events can simply be done through calling *device.command* provided from our processed conflict log. There are limitations in the SmartHub scheduling feature on large quantities of events as it is only able to provide limited memory, however, this should not be an issue as we expect the environment rules to be simple and thus the generated log to be small.

Unfortunately, it is not possible for us to know locally when the scheduled events on Samsung Smartthings Hub gets finished, as a result, we have set a safety time buffer to wait between each simulation. A natural number that we determined is the *timebound* parameter we used to generate the conflict log in our conflict checking tool, since all the events should be happening within the *timebound* interval by our method described in 5.1, guaranteeing the simulation’s termination.

5.4 Evaluation

To evaluate the correctness and the efficiency of our conflict checking tool, we utilized our fuzz testing framework described in 4.4. We used the same devices as described in Figure 20, and generated random environments with 10 device interaction rule specifications. Then, we used our conflict checker to detect whether there are any conflicts in the generated temporal and immediate rules. The process is done 5 times to ensure the robustness of our tool, and we manually checked for each detected conflict that the output log is valid.

One concern we had in our conflict checking tool before evaluation was the runtime of our SMT solver due to SMT problems are inherently exponential. To access the scalability of our STLCheck, we first note that the runtime of our z3 SMT checker is impacted by the number of rule sets to check, number of SMT formulas, and number of SMT variables translated from each IoT environment. These factors, from our translation schema of the rules described in section 5.1, are directly influenced by the number of devices, number of total rules, and the maximum timebound *timebound* defined in the environment. Consequently, we ran our STLCheck on increasingly complex environments through incrementing these parameters. For each of the parameter combinations, STLCheck is ran on 10 randomly generated environment under these parameters with our fuzz testing framework, and the average runtime for each combination is recorded in Table 1.

The *numDevices* and *Rules/Device* parameters picked for the environment are done through the reasoning that while IoT systems are becoming increasingly prevalent, it is still fairly unlikely for a typical household to have a very large set of devices and rules per device. We did not select greater *timebound* parameters since a greater timebound in learning rules can be interchanged with a greater granularity in timestamp unit, which would not impact our runtime efficiency much.

For small *numDevices* and *Rules/Device* combinations, STLCheck’s runtime grows fairly linearly as expected with z3’s near linear algorithm. The impact for increasing the *timebound* parameter does not impact the runtime as much in these small combinations as we expected for linear growth. One possible explanation is that while increasing *timebound* increases the number of parameters for SMT formulas, not all parameters are relevant since we do not have enough rules to have PSTL intervals that cover the entire time period specified by *timebound*.

Overall, for each of the generated environment, STLCheck is able to detect for conflicts within all possible combinations of the rules, even with the drastic overestimate of rules and devices for a typical household IoT

numDevices	Rules/Device	Timebound	runtime(s)
4	4	10	0.336
4	8	10	0.614
4	8	30	0.971
8	8	10	1.33
8	8	30	1.94
16	8	30	4.58
16	16	30	17.8

Table 1: Average runtime of STLCheck for environments with different parameter combinations

environment in our last combination, in under half a minute. This evaluation results shows STLCheck significantly exceeds our efficiency expectations.

6 Related Works

Current research on IoT security mostly rely on static checking for developing security rules and analyzing device interactions. Our work is initially inspired by the work done by [IoT], which defined seven classes of multi-app interaction threats that motivated us to look into the hidden interactions between IoT devices and implemented formal models for IoT systems to automatically detect such threats. As our work’s static analysis counterpart, [TAC⁺20] created IoTCheck, an interaction learning and conflict detection tool through statically analyzing pairs of Smarthings apps and their execution traces. Similarly, [DH18] implemented IoTMon, a static checking framework that discovers and assess the security risks of hidden interaction chains among Samsung Smartapps by statically checking each Smartapp’s source code. Interestingly, [DH18] has also utilized machine learning models of natural language processing in discovering the device interactions; however, it is still restricted to using the model statically in analyzing the description of Smartapps. Currently, no known research as been found in learning IoT device interactions through environment’s runtime logs and in learning time-related rules, making our work novel. Furthermore, all of the related work described above are limited in that only learning interactions and detect conflicts between pairs of Smartapps, which is able to capture significantly less security properties than our work does with our ability to explore the hidden interaction with multiple Smartapps that is dependent on the depth set for STLTree and TreeNoSTL and detect conflicts within any subset of the interaction and user defined rules using STLCheck.

There is currently no known work for incorporating STL or PSTL rules to IoT environments. However, PSTL rules is a powerful tool with prevalent uses in other fields of cyberphysical systems. In the original decision tree framework that STLTree is adapted from, [BVP⁺16] utilizes PSTL in distinguishing abnormal naval trajectories’ and vehicle fuel sensors’ signal data to issue warnings. Similar work on learning rules about vehicles brakes through STL is explored in [JKB14]. From this project, we are also fascinated by Signal Temporal Logic’s expressive power, and are also excited to expand and experiment with STL and PSTL into other cyberphysical systems in the future.

7 Conclusion and Future Work

In our work, we developed a novel way in representing device interactions and user desired environment security properties through Parametric Signal Temporal Logic (PSTL). Then, in order to learn these interaction rules and capture hidden behaviors, we adapted and expanded the decision tree learning algorithm on PSTL rules proposed in [BVP⁺16] with our model STLTree. To capture all the interactions in an environment holistically, we also implemented a decision learning algorithm TreeNoSTL to learn all the immediate rules of the environment.

From our learned rules with STLTree and TreeNoSTL, we implemented a runtime monitor STLMon and a static conflict checker STLCheck to evaluate the safety properties of an IoT environment described in Definition 1 and Definition 2 of 1.2. To express individual user's desired safety behaviors, we implemented a language for user defined rules that can be parsed into PSTL formulas. After receiving user's desired safety rules of the environment, STLMon is able to dynamically check for both interaction rule and user defined safety rules of the environment, and physically revoke a state change action if the action results in a rule violation. Finally, STLCheck is able to statically check for conflicts between and within both learned device interaction rules and user defined safety rules to ensure the safety property of an IoT environment defined in Definition 1.

Since introducing PSTL logic is new to IoT environment security, there are plenty of opportunities to expand on our current work in the future. First, we would like to consider adding static analysis to our learned rules to see if we are able to construct longer correlation chains than our decision tree algorithms are able to learn. For example, in our motivation example in section 1.1, it is likely that our decision tree algorithm to learn the separate pieces that oven is able to trigger smoke alarm with smoke, and smoke alarm is able to unlock the door than simply the entirety of behavior of oven unlocking the door, since the interaction with oven and the door is significantly less likely to happen than smoke alarm and the door. Being able to form a control flow graph between learned fragments of the rules to forge longer rules would make us able to capture more interactions in the environment.

Also, for our evaluation of STLTree, we have set the interval for data processing to be 10 timestamps, which is simply a number for convenience of testing. This interval size may not be the best measure for reflecting the interaction rules between devices, and additional analysis on determining the best time intervals for preprocessing the device state changes for various situations and environments could lead to our decision tree model learning more accurate rules.

Furthermore, our implemented STLMon and STLCheck are only able to detect potential security conflicts and remedy the security violations by revoking state changes. This is often not enough in preventing actual attacks, and we would like to implement other mechanisms to IoT environments such as fail safe measures at time of attacks on the vulnerabilities of conflicts (For example, making the door to always stay locked unless there is sufficient evidence for not doing so) to further strengthen the security of IoT systems.

Finally, one key measure of our security analysis in STLMon and STLCheck is the user desired security properties that is parsed through our defined rule language into STL rules. In the future, we would like to expand upon this language to capture more behaviors, and implementing a more user friendly interface than the current requirement of physically writing down the rules.

8 References

- [ADMN12] Eugene Asarin, Alexandre Donze, Oded Maler, and Dejan Nickovic. Parametric identification of temporal properties. In *In: Runtime Verification*, pages 147–160. Springer, 2012.
- [Aut] SeleniumHQ Browser Automation. <https://www.selenium.dev/>.
- [BVP⁺16] Giuseppe Bombara, Cristian-Ioan Vasile, Francisco Penedo, Hirotoshi Yasuoka, and Calin Belta. A decision tree approach to data classification using signal temporal logic. pages 1–10, 04 2016.
- [DH18] Wenbo Ding and Hongxin Hu. On the safety of iot device physical interaction control. New York, NY, USA, 2018. Association for Computing Machinery.
- [dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [Gr1] Alex Grönholm. Advanced python scheduler. <https://github.com/agronholm/apscheduler>, 2021.
- [HR76] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15–17, 1976.
- [Ing96] Lester Ingber. Adaptive simulated annealing (asa): Lessons learned. In *Control Cybern*, pages 33–54, 1996.
- [IoT] Iotcom: Scalable analysis of interaction threats in iot systems. <https://sites.google.com/view/iotcom/home>.
- [JKB14] Austin Jones, Zhaodan Kong, and Calin Belta. Anomaly detection in cyber-physical systems: A formal methods approach. volume 2015, 12 2014.
- [MN04] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *In: Proceedings of FORMATS-FTRTFT. Volume 3253 of LNCS*, pages 152–166. Springer, 2004.
- [Per19] Matthew Perry. simanneal. <https://github.com/perrygeo/simanneal>, 2019.
- [PVG⁺11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [Qui14] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Ebrary online. Elsevier Science, 2014.
- [sci] scikit. 1.10.8 minimal cost complexity pruning. <https://scikit-learn.org/stable/modules/tree.html#minimal-cost-complexity-pruning>.
- [Smaa] Samsung Smartthings. Smartthings device capability reference. <https://smartthings.developer.samsung.com/docs/api-ref/capabilities.html>.
- [Smab] Samsung Smartthings. Smartthings documentation. <https://smartthings.developer.samsung.com/docs/index.html>.
- [Smac] Samsung Smartthings. Smartthings groovy ide. <https://graph.api.smartthings.com/>.
- [TAC⁺20] Rahmadi Trimananda, Seyed Amir Hossein Aqajari, Jason Chuang, Brian Demsky, Guoqing Harry Xu, and Shan Lu. Understanding and automatically detecting conflicting interactions between smart home iot applications. New York, NY, USA, 2020. Association for Computing Machinery.

9 Appendix

9.1 Information Gain (IG) with robustness

For continuous valued device states, there could be multiple valid splits with the same information gain. For example, in a temperature measurement of a household, learning a rule that the temperature should always be less than 110 degrees would be more useful than learning a rule of temperature should always be less than 1000 degrees, while both may give the same one sided splits that has the same information gain with equation (4).

In order to distinguish such rules and pick the better one for the context, [BVP⁺16] proposed a robustness measurement that not only calculates information gain based on misclassification, but also on the degree of misclassification. In this measurement, robustness $r(s_i)$ for a signal s_i under rule with space parameter μ is defined to be the same way as how we checked for validity of PSTL rules in 3.2.4, as follows

1. $F_{[a,b]}(x \leq \mu) = \mu - \min_{[a,b]} v$, where v is the value for the dimension at a timestamp in the interval $[a, b]$ for signal s_i . Similarly, $F_{[a,b]}(x > \mu) = \max_{[a,b]} v - \mu$
2. $G_{[a,b]}(x \leq \mu) = -F_{[a,b]}(x > \mu)$ and $G_{[a,b]}(x > \mu) = -F_{[a,b]}(x \leq \mu)$ by 2.2.1.
3. $F_{[a,b]}G_{[0,c]}(x \leq \mu) = \mu - \max_{i \in [a,b-c]} \min_{[i,i+c]} v$ with the max min window filter of length c method described in 3.2.4. In this case, v is the value for the dimension at a timestamp in the interval window $[i, i + c]$ for signal s_i . Similarly, $F_{[a,b]}G_{[0,c]}(x > \mu) = \min_{i \in [a,b-c]} \max_{[i,i+c]} v - \mu$

With the robustness measure, the information gain is calculated by adjusting equation (4) through changing $prob(S, c, \phi)$ when calculating entropy as follows:

$$prob(S, c, \phi) = \frac{\sum_{(s^i, l^i) \in S, l^i = c} r(s_i)}{\sum_{(s, l) \in S} r(s)} \quad (8)$$

Intuitively, instead of only factoring the count of misclassification, the extent of misclassification is included through the robustness measure.

9.2 Simulated Annealing (SA) in STLTree

In order to efficiently find the PSTL rule to split our data and growing the tree over a potentially large search space of parameters, we utilizes the Simulated Annealing algorithm when training STLTree. In simulated annealing, the algorithm performs a random walk over the parameter search space in a function of temperature and energy [Ing96] [Per19]. At each iteration of the random walk, the algorithm computes the energy corresponding to the walk's state. If the algorithm has walked to a state with lower energy, the algorithm decides whether to change the current state to the lower energy state randomly depending on the walks temperature value. The higher the temperature value is, the more likely it is for our algorithm to change to the lower energy state. To speed up convergence for the algorithm, temperature would set initially to be a high number and continues to decrease as iteration increases.

In STLTree, we utilized Python's simanneal package by [Per19]. When running the algorithm, we set the energy to be our objective function, which is the entropy for rule split as described in equation (5) in 3.2.4. We set the state to be the parameters for the PSTL primitive, and a random walk to be a random selection of the time and space parameters within the restricted bounds as described in 3.2.4. After running the algorithm for enough iterations, SA would converge at a local minimum, which means we have found the parameters that gives the least entropy split (which also means the greatest Information Gain (IG)) for the current PSTL primitive type [Ing96].

9.3 Treepruning with Effective Alpha

As described in 3.2.2, the decision tree models learned from our IoT environment device logs are very prone to overfit. As a result, we adapted Python sklearn library's Cost Complexity Pruning to post prune our learned STLTrees [sci]. In the process, a tree T 's cost complexity measure is given by

$$R_\alpha(T) = R(T) + \alpha|T| \quad (9)$$

where $R(T)$ is the total training error of leaf nodes, $|T|$ is the number of leaf nodes, and α being a complexity parameter. The Cost Complexity Pruning process aims to find the subtree T' of T that minimizes $R_\alpha(T')$.

We note for a single node t , $R_\alpha(t) = R(t) + \alpha$, and in general, for a tree T_t rooted at t , $R(T_t) < R(t)$ since splitting a tree should reduce the total prediction error, and in our case described in 3.2.7, it should reduce by at least the improve

threshold on one branch. However, by how we have defined α , we can have $R_\alpha(T_t) = R_\alpha(t)$ since $|T_t| \geq 1$. Thus, we define the effective alpha of a node t to be the value that equality happens, that is

$$\alpha_{eff}(t) = \frac{R(t) - R(T_t)}{|T| - 1} \quad (10)$$

We note a small α_{eff} would mean $R(t)$ and $R(T_t)$ are fairly close, which means the split is less helpful in improving tree model's classification. In our tree pruning method described in 3.2.7, leaves will be pruned based on the order of their parents' effective order, with the smallest effective alpha to be pruned first. The process continues until we reached a minimum in validation error.

9.4 OneHotEncoder

Sklearn's tree module only accepts data traces that are solely formed by integers. In order for us to learn a tree with non-real value device states (Ex. A switch's state can be *on_off* based on our preprocessing in 3.3.2), we utilizes Sklearn's OneHotEncoder module [PVG⁺11]. In the encoder, the transformer encodes our non-real values into a numeric array using a one-hot encoding scheme for each unique value in our data trace. After learning our tree, we can simply get back our original class value for each device by calling the *encoder.decode()* function.