Cross-App Interference Threats in Smart Homes: Categorization, Detection and Handling

Haotian Chi, **Qiang Zeng**, Xiaojiang Du, Jiaping Yu

TEMPLE UNIVERSITY

UNIVERSITY OF South Carolina

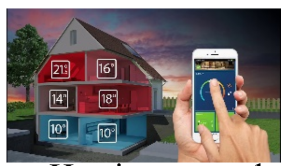
Home Automation

- Home-automation **apps**
  - *Lock the door when all leave*
  - *When arriving home, if the room is too hot, turn on A/C*

Single-device automation

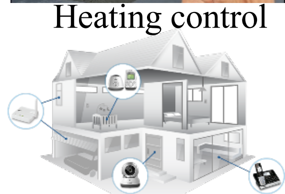
Smoke detection

Subsystem-level automation



Heating control

Lighting control



Camera surveillance

Smart kitchen

Home-wide automation

Rule Abstraction

A home-automation app contains one or more rules, each in the form of *T-C-A*

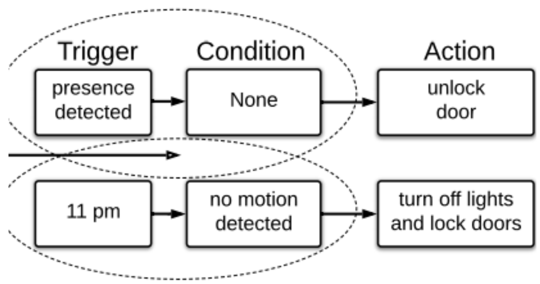
- **Trigger**: "when resident arrives home"
- **Condition:** "if room temperature < 18c"
- **Action**: "turn on heater"

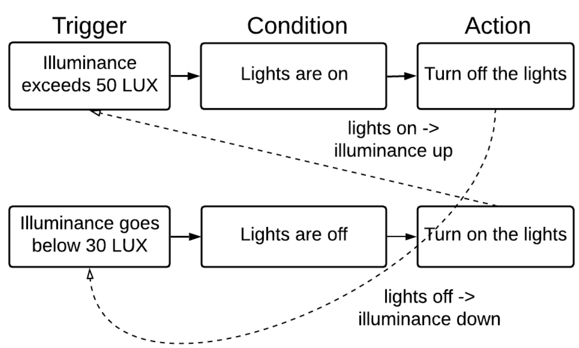
Rule = TCA template + Configuration

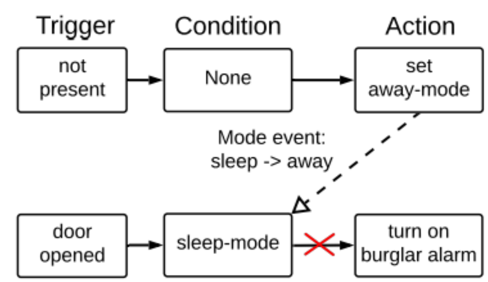
- ❑ **Configuration**: app-device binding relations, threshold, etc.

Cross-App Interference (CAI) Threats

When multiple rules interplay in a home, they may interfere with each other







Research Questions

- ❑ How to systematically categorize CAI threats?
- ❑ How to precisely detect them?
- ❑ How to assist users to handle them?

Categorization of CAI Threats

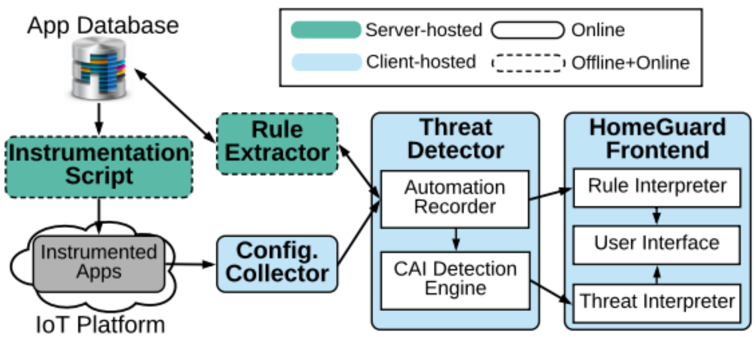
Rule 1: T1-C1-A1; Rule 2: T2-C2-A2

Category	Basic Pattern	Auxiliary Pattern[1]	ID	Validation	Description
Action-Interference Threats	$A_1 = \neg A_2$	$T_1 = T_2$, $C_1 \wedge C_2$	A.1	✓	R_1 and R_2 are executed simultaneously to perform conflict actions.
		$T_1 = T_2$, $\sim(C_1 \wedge C_2)$	–	✗	R_1 and R_2 cannot be both executed although they are both triggered.
		$T_1 \neq T_2$, $C_1 \wedge C_2$	A.2	✓	R_1 and R_2 may be executed within a short period to perform conflict actions.
		$T_1 \neq T_2$, $\sim(C_1 \wedge C_2)$	–	✗	R_1 and R_2 are unrelated and have no interaction.
Trigger-Interference Threats	$A_1 \mapsto T_2$	$C_1 \wedge C_2$, $\sim(A_2 \mapsto T_1)$, $A_1 \neq \neg A_2$	T.1	✓	R_1 triggers R_2, which does not interfere with R_1 in turn.
		$C_1 \wedge C_2$, $\sim(A_2 \mapsto T_1)$, $A_1 = \neg A_2$	T.2	✓	R_1 triggers R_2, which performs a conflict action and thus invalidate R_1.
		$C_1 \wedge C_2$, $A_2 \mapsto T_1$, $A_1 \neq \neg A_2$	T.3	✓	R_1 and R_2 trigger each other alternately.
		$C_1 \wedge C_2$, $A_2 \mapsto T_1$, $A_1 = \neg A_2$	T.4	✓	R_1 and R_2 trigger each other and perform conflict actions alternately.
		$\sim(C_1 \wedge C_2)$		✗	R_2 fails its condition checking and cannot be executed.
Condition-Interference Threats	$A_1 \Rightarrow C_2$	$T_1 = T_2$	C.1	✓*	R_1 turns a constraint in R_2's condition to true, which increases the probability of R_2 being executed.
		$T_1 \neq T_2$	C.2	✓	
	$A_1 \nRightarrow C_2$	$T_1 = T_2$	C.3	✓*	R_1 turns a constraint in R_2's condition to false, which decreases the probability of R_2 being executed.
		$T_1 \neq T_2$	C.4	✓	

Challenges

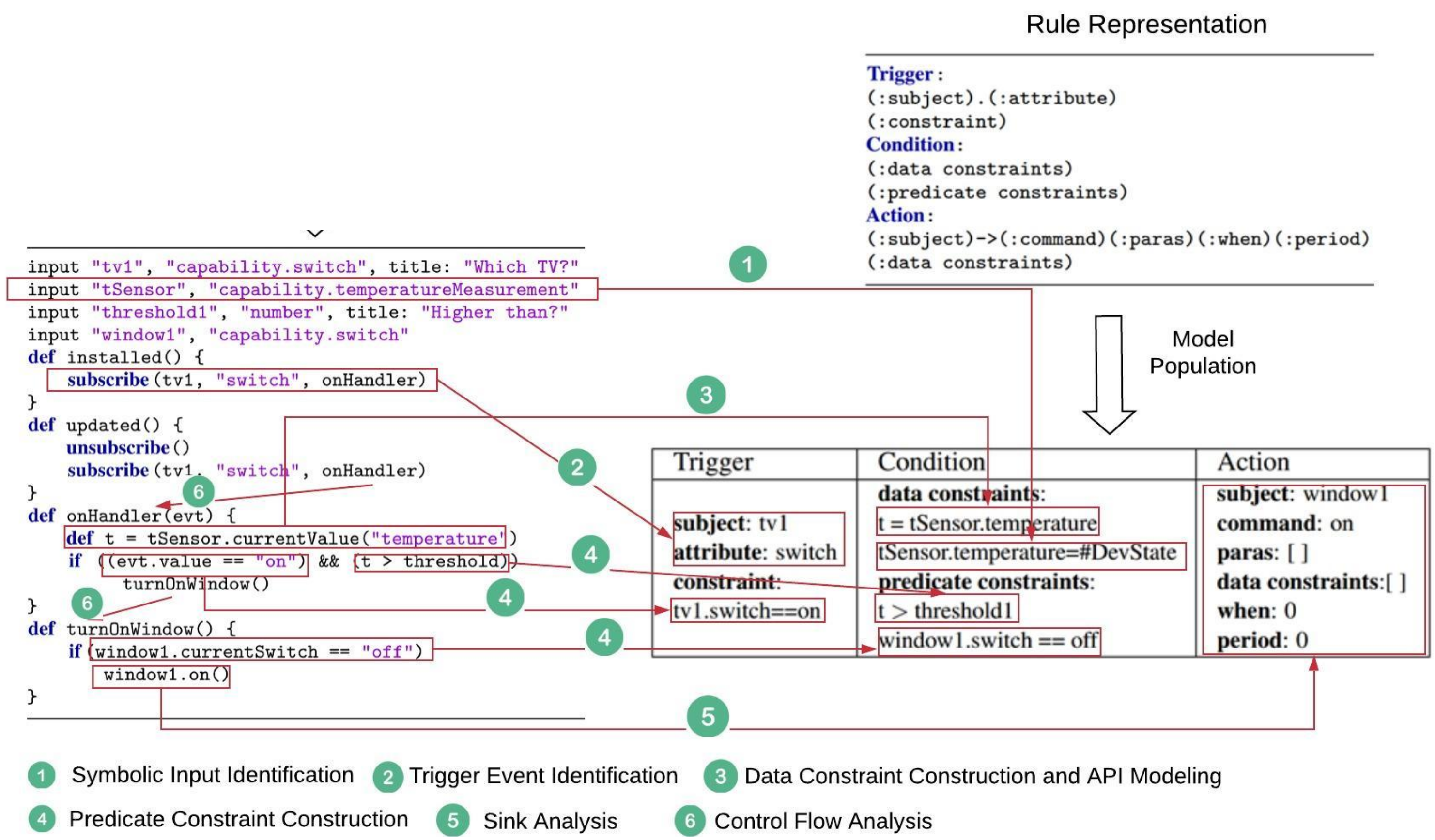
- Extract rules from app code precisely
- Obtain user configuration without co-operation of IoT platforms
- Automatic CAI threat detection

System Overview



- ❑ Rule Extractor
  - extracts rule semantics from app source code
- ❑ Configuration Collector
  - collects configuration upon a new app is installed
- ❑ Threat Detector
  - analyzes whether any pair of rules causes CAI threats
- ❑ Frontend (A Mobile App)
  - presents detection result

Rule Extraction – Symbolic Execution



Configuration Collection - Code Instrumentation

```
//...
//Specify the phone that runs HomeGuard
input "patchedphone", "phone", required: true, title: "Phone
      number?"
//...
def updated() {
    //...
    // inserted code
    def appname = "ComfortTV"
    def devices = [[devRefStr:"tv1", devRef:tv1],
          [devRefStr:"tSensor", devRef:tSensor],
          [devRefStr:"window1", devRef:window1]]
    def values = [[varStr:"threshold1", var:threshold1]]
    collectConfigInfo(appname, devices, values)
}
//...
def collectConfigInfo(appname, devices, values) {
    def uri = "http://my.com/appname:${appname}/"
    devices.each { dev ->
        uri = uri + dev.devRefStr + ":" + dev.devRef.getId()
              + "/"
    }
    values.each { val->
        uri = uri + val.varStr + ":" +val.var + "/"
    }
    sendSmsMessage(patchedphone,uri)
}
```

Collect app-device binding info.

Collect user-input values

Pack all configuration info.

Send the configuration info. out through the supported SMS or HTTP APIs.

CAI Threat Detection

When a new app is installed, our system detects CAI threats between every pair of rules

- Each CAI type is encoded as a set of **symbolic constraints**
- Thus, CAI detection is transformed into solving **SMT** (satisfiability modulo theory) problems

Category	Basic Pattern	Auxiliary Pattern[1]	ID	Validation	Description
Action-Interference Threats	$A_1 = \neg A_2$	$T_1 = T_2$, $C_1 \wedge C_2$	A.1	✓	R_1 and R_2 are executed simultaneously to perform conflict actions.
		$T_1 = T_2$, $\sim (C_1 \wedge C_2)$	–	✗	R_1 and R_2 cannot be both executed although they are both triggered.
		$T_1 \neq T_2$, $C_1 \wedge C_2$	A.2	✓	R_1 and R_2 may be executed within a short period to perform conflict actions.
		$T_1 \neq T_2$, $\sim (C_1 \wedge C_2)$	–	✗	R_1 and R_2 are unrelated and have no interaction.
Trigger-Interference Threats	$A_1 \mapsto T_2$	$C_1 \wedge C_2$, $\sim (A_2 \mapsto T_1)$, $A_1 \neq \neg A_2$	T.1	✓	R_1 triggers R_2, which does not interfere with R_1 in turn.
		$C_1 \wedge C_2$, $\sim (A_2 \mapsto T_1)$, $A_1 = \neg A_2$	T.2	✓	R_1 triggers R_2, which performs a conflict action and thus invalidate R_1.
		$C_1 \wedge C_2$, $A_2 \mapsto T_1$, $A_1 \neq \neg A_2$	T.3	✓	R_1 and R_2 trigger each other alternately.
		$C_1 \wedge C_2$, $A_2 \mapsto T_1$, $A_1 = \neg A_2$	T.4	✓	R_1 and R_2 trigger each other and perform conflict actions alternately.
		$\sim (C_1 \wedge C_2)$		✗	R_2 fails its condition checking and cannot be executed.
Condition-Interference Threats	$A_1 \Rightarrow C_2$	$T_1 = T_2$	C.1	✓*	R_1 turns a constraint in R_2's condition to true, which increases the probability of R_2 being executed.
		$T_1 \neq T_2$	C.2	✓	
	$A_1 \not\Rightarrow C_2$	$T_1 = T_2$	C.3	✓*	R_1 turns a constraint in R_2's condition to false, which decreases the probability of R_2 being executed.
		$T_1 \neq T_2$	C.4	✓	

[1] The auxiliary pattern of each CAI type does not conform to the basic pattern in other categories if not explicitly specified. We elide the negation constraints for conciseness.

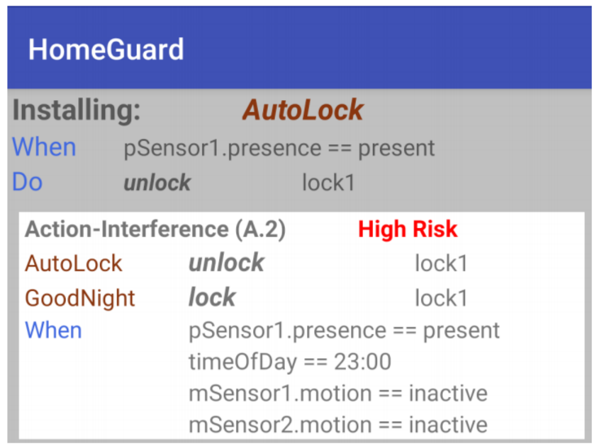
Frontend App Presents Detection Result

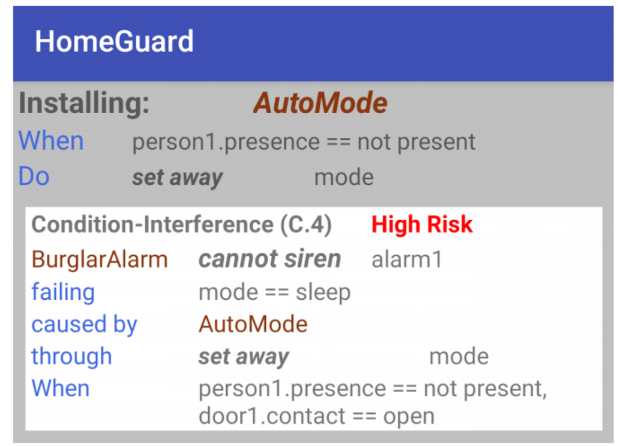
AutoLock – **Rule 1**

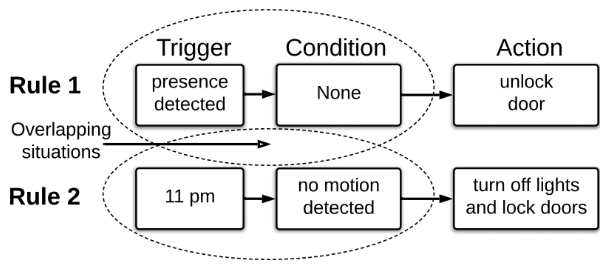
GoodNight – **Rule 2**

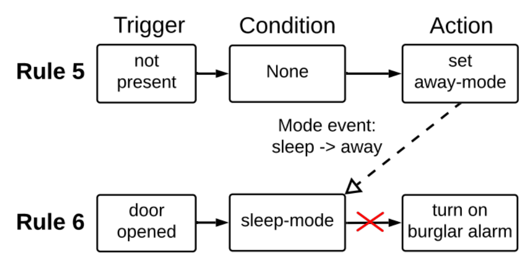
AutoMode – **Rule 5**

BurglarAlarm – **Rule 6**





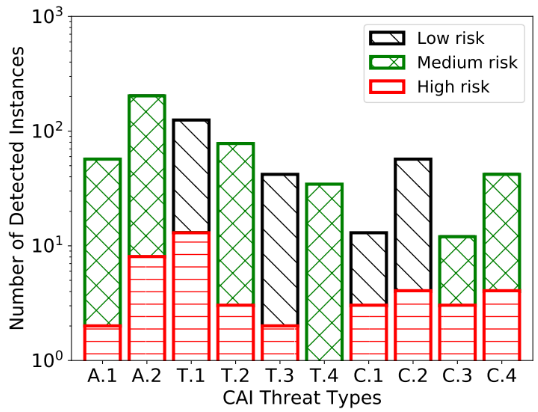


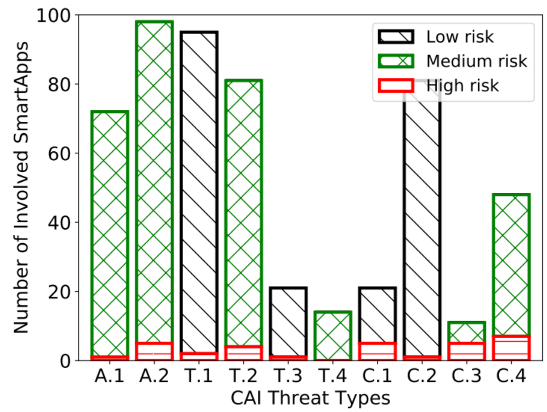


CAI threats in Market Apps

Run our CAI detection on 146 SmartApps in SmartThings official app repository

- 663 CAI instances in total
- 101 out of 146 apps are involved into at least one type of CAI threat

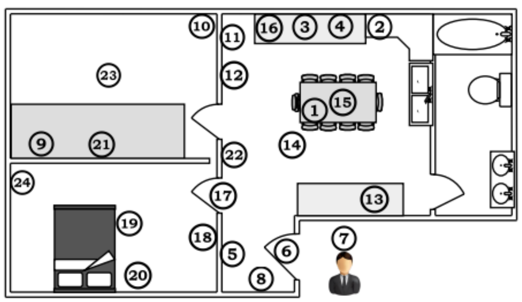




Real-world Testbed

- 18 official apps
- 24 devices

Devices and their layout



1. Motion sensor
2. Oven
3. Fan
4. Temp. sensor
5. Switch
6. Door Lock
7. Presence sensor
8. Thermostat
9. Temp. sensor
10. Fan
11. Power meter
12. Humidity sensor
13. Humidifier
14. Ventilator
15. Light
16. Luminance sensor
17. Contact sensor
18. Luminance sensor
19. Light
20. Floor Lamp
21. Motion sensor
22. Alarm
23. Light
24. Fan

Detected (and verified) CAI threats

App Name	Rule and Configuration	Set #	CAI Type
CurlingIron Virtual Thermostat	When motion ① detected, turn on oven ② and fan ③ for 30 minutes. When motion ① detected, if temperature ④ is lower than 72°F, turn off fan ③.	1	**A.1**
NFCTagToggle LockItWhenILeave	When the user touches on mobile app, toggle switch ⑤ and door lock ⑥. When presence sensor ⑦ becomes “not present”, lock door ⑥.	2	**A.2**
CurlingIron SwitchChangesMode MakeItSo	When motion ① detected, turn on oven ② and fan ③. When oven ② is turned on, set home to “party” mode. When change to “Party” mode, unlock door ⑥ and turn on thermostat ⑧.	3	**T.1**
It’sTooHot EnergySaver	When temperature ⑨ exceeds 80°F, turn on fan ⑩. When power usage ⑪ exceeds 3000 W, turn off fan ⑩.	4	**T.2**
SmartHumidifier HumidityAlert!	When humidity ⑫ is below 30%, turn on humidifier ⑬; when humidity ⑫ exceeds 50%, turn off humidifier ⑬. When humidity ⑫ exceeds 50%, turn on ventilator ⑭; when humidity ⑫ is below 30%, turn off ventilator ⑭.	5	**T.3**
LightUptheNight	When illuminance ⑯ exceeds 50 lx, turn off light ⑮; when illuminance ⑯ gets below 30 lx, turn on light ⑮.	6	**T.4**
Brighten Dark Places LetThereBeDark	When door ⑰ is opened, if illuminance ⑱ is below 10 lx, turn on light ⑲. When door ⑰ is opened, turn off lights ⑳; when door ⑰ is closed, restore the state of lights ⑳.	7	**C.1**
Forgiving Security Scheduled Mode Change	When motion sensor ① or ㉑ becomes “active”, if the home is in “Away” mode, siren alarm ㉒. Set home to “Away” mode at 10 am and set home to “Night” mode at 6pm.	8	**C.2, C.4**
Forgiving Security Rise and Shine	When motion sensor ㉑ becomes “active”, if home is in “Work” mode, turn on light ㉓ after 1 second. When motion sensor ㉑ becomes “active”, set home to “At-Home” mode (“Work” mode).	9	**C.3 (C.1)**
GoodNight Once a Day MakeItSo	When motion ①㉑ detected, if switches ②③⑤⑩⑬⑭⑮⑲⑳㉓㉔ are all off, set home to “sleep” mode. Turn on fan ㉔ at 11 pm and turn off fan ㉔ at 12 am. When changed to “sleep” mode, lock door ⑥.	10	**C.4**

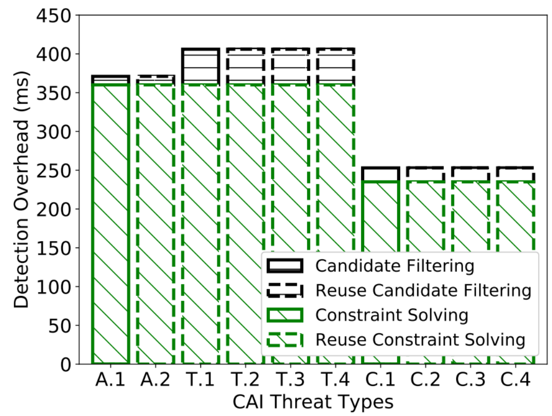
Evaluation of Performance

Rule extraction speed

- 1,341 millisecond per app on average

CAI detection speed

- Averaged 671 millisecond



Related Work

	Publication Date[1]	# of CAI Threat Types	Systematic Categorization?	Symbolic Threat Modeling?	CAI Threat Detection			Risk Ranking?
					Precise Semantics Extraction?	Leverage App Configuration?	No Need For Specification?	
SIFT [21]	Apr 2015	1	✗	✗	✗[2]	✓	✓	✗
Surbatovish et al. [20]	Apr 2017	1	✗	✗	✗	✗	✓	✗
IoTA [22]	Oct 2017	3	✗	✗	✗	✓	✓	✗
Soteria [17]	May 2018	3	✗	✗	✓	✗	✗	✗
IoTSan [18]	Oct 2018	2	✗	✗	✓	✓	✗	✗
IoTMon [16]	Oct 2018	1	✗	✗	✓[3]	✗	✓	✓
IoTGuard [19]	Feb 2019	3	✗	✗	✓	✓	✗	✗
SafeChain [23]	Oct 2019	1	✗	✗	✗	✗	✗	✓
iRuler [24]	Nov 2019	8	✓	✓	✗[4]	✓	✓	✗
HomeGuard	**Aug 2018**	**10**	✓	✓	✓	✓	✓	✓

Conclusion

- ❑ **First** comprehensive categorization of CAI threats (first posted on arXiv in **Aug 2018**)
- ❑ **First** symbolic representation of CAI threats
- ❑ **First** work that leverages SMT for CAI threat detection
- ❑ An end-to-end implementation without co-operation of IoT platforms

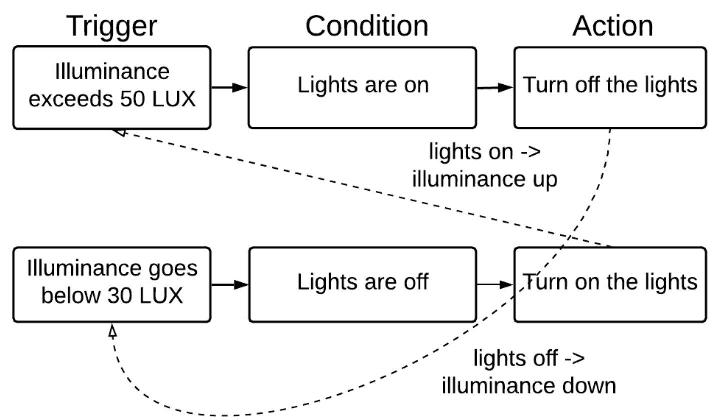
Thank You!

Q & A

Another Example

A SmartApp in SmartThings public repository:

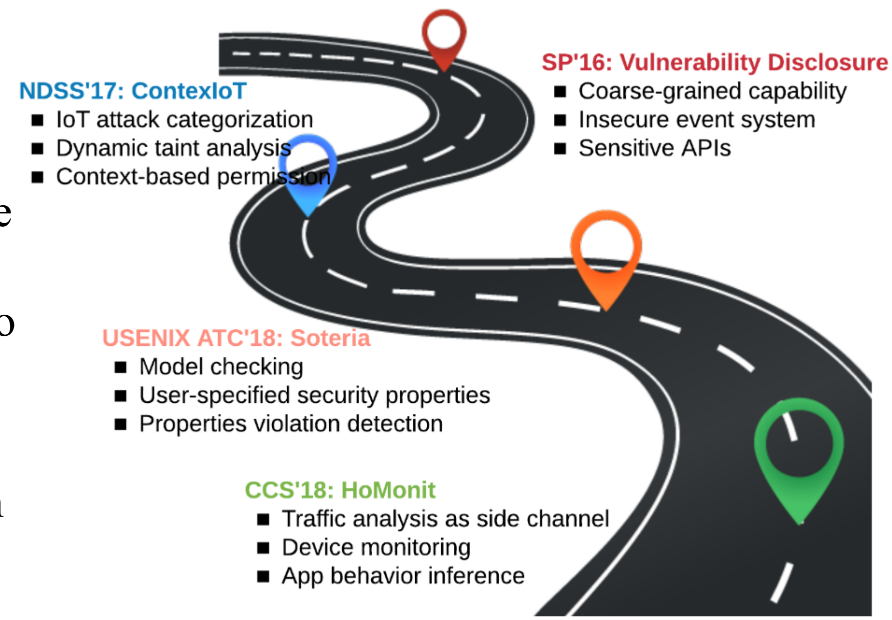
```
definition(
    name: "Light Up the Night",
    namespace: "smartthings",
    author: "SmartThings",
    description: "Turn your lights on when it gets dark and off when it becomes light again.",
    category: "Convenience",
    iconUrl: "https://s3.amazonaws.com/smartapp-icons/Meta/light_outlet-luminance.png",
    iconX2Url: "https://s3.amazonaws.com/smartapp-icons/Meta/light_outlet-luminance@2x.png"
)

def installed() {
        subscribe(lightSensor, "illuminance", illuminanceHandler)
}

def illuminanceHandler(evt) {
        def lastStatus = state.lastStatus
        if (lastStatus != "on" && evt.integerValue < 30) {
                lights.on()
                state.lastStatus = "on"
        }
        else if (lastStatus != "off" && evt.integerValue > 50) {
                lights.off()
                state.lastStatus = "off"
        }
}
```



Intra-App Attacks

Intra-app attacks (a.k.a., malicious apps, or malware) is a well-known threat

- Gain unauthorized access to smart home devices
  - **Send malicious commands** to home IoT devices
    - ✓ Unlock a door when the owner is absent
    - ✓ Turn on the oven over a long time to cause a fire accident
    - ✓ DoS the devices to demand a ransom payment to restore them
  - **Exfiltrate sensitive data** from the sensor devices
    - ✓ Medical device data expose the health condition of the homeowner
    - ✓ A presence sensor reveals the occupation of a house



CAI Threat Detection (Cont'd)

$T_1 = T_2$ and $A_1 = \neg A_2$ are relatively straightforward to determine by respectively looking at whether:

- Two rule triggers subscribe to the same event (same device, attribute and value)
- Two rule actions control the same device with contradictory commands

$A_1 \mapsto T_2$ has two cases:

- A_1 and T_2 are handling the **same device** and **attribute**
  - For example, turning on a ON/OFF switch triggers a rule that subscribes to the switch's "ON" event
- A_1 controls an actuator that changes the reading of a sensor subscribed to by T_2 via a **physical channel**
  - For instance, turning on a heater triggers a rule that subscribes to the reading of a temperature sensor

$C_1 \wedge C_2$: **condition overlapping detection**

- We often need to detect if two rules' conditions have overlaps. For example,
  - tSensor1.temperature > 70, tv1.switch == "on", 13.00 < time < 19.50
  - tSensor1.temperature < 75, tv1.switch == "off"
- The overlapping detection is transformed into a **constraint satisfaction** problem
  - We use the Java Constraint Programming (JaCoP) library as the solver

Risk Ranking - Help Users to Handle CAI Threats

A detected CAI instance is labeled with a risk level $L \in \{low, medium, high\}$.

- **Two observations**
  - Each CAI threat type has specific effects on the involved rules:
    - ✓ Positive (+): unexpectedly triggers the execution of a rule
    - ✓ Negative (-): unexpectedly invalidates the execution of a rule
    - ✓ Loop (o): makes two rules trigger each other to form a loop execution

CAI Type	A.1	A.2	T.1	T.2	T.3	T.4	C.1	C.2	C.3	C.4
Action A_1	−	−		−	+	o				
Action A_2	−	−	+	+	+	o	+	+	−	−

  - The risk of implication of three effects (+, -, o) depends on two factors:
    - ✓ The functionality of the involved rule
    - ✓ The sensitivity of the rule action

Consider an example. R1 ("unlock the door and open the window when smoke is detected") and R2 ("open the window when air quality is low")

- If R3 imposes negative effects on "open the window", it imposes a higher risk to R1 than R2 since R1 is for safety purpose and R2 is for comfort purpose.
- If R3 imposes positive effects on "open the window", it does not break the functionality of both rules but is still considered dangerous due to the sensitivity of the action "open the window" itself.

Risk Ranking (Cont'd)

CAI threat instance $\leftarrow$ $I(type, R_1, R_2)$, where $R_1 = (T_1, C_1, A_1)$ and $R_2 = (T_2, C_2, A_2)$;

A risk level $\leftarrow$ $L \in \{-1, 0, 1\}$ $(low, medium, high)$;

The effect of I on each rule R_i $\leftarrow$ e_i, determined by $type$

The category of a rule $\leftarrow c_i$

CAI Type	A.1	A.2	T.1	T.2	T.3	T.4	C.1	C.2	C.3	C.4
Action A_1	$-$	$-$		$-$	$+$	o				
Action A_2	$-$	$-$	$+$	$+$	$+$	o	$+$	$+$	$-$	$-$

The command type of R_i's action $\leftarrow cmd_i$

The risk imposed by I on each rule R_i is defined as:
$L_i = \max\big(M_1(e_i, c_i), M_2(e_i, cmd_i)\big)$,
where M_1 and M_2 are two mappings designed based on the two observations.

The mapping table of M_1:

Category	Effect($+$)	Effect($-$)	Effect(o)
safety rules	low	high	high
non-safety rules	low	medium	medium

Risk Ranking (Cont'd)

To establish the mapping table of M_1, i.e., the risk level regarding CAI effect e_i and command type cmd_i, an algorithm is designed to extract such knowledge from 146 official SmartApps.

The simple insight in the algorithm is that security-sensitive commands are used more frequently in SmartApps of the "Safety & Security" category than those of other categories; with non-sensitive commands, the opposite is true.

Algorithm 1: The algorithm for extracting device control sensitivity under different CAI effects from SmartApps

```
Input  : Apps ← the source code of all SmartApps
Output: Risk knowledge model of capability-supported commands KM
foreach app ∈ Apps do
    Rules ← ExtractRuleSemantics(app)
    catetory ← ExtractCategory (app)
    foreach rule ∈ Rules do
        capability ← FindCapability(rule.action.device)
        cmd ← rule.action.command
        capCmd ← Concatenate(capability, cmd)
        oppCapCmd ← FindOppositeCmd(capCmd)
        if category is "Safety & Security" then
            count[capCmd]['+']['low']++
            count[capCmd]['−']['high']++
            count[capCmd]['o']['high']++
            count[oppCapCmd]['+']['high']++
            count[oppCapCmd]['−']['low']++
            count[oppCapCmd]['o']['high']++
        else
            count[capCmd]['+']['low']++
            count[capCmd]['−']['medium']++
            count[capCmd]['o']['medium']++
foreach capCmd ∈ count.keys() do
    foreach e ∈ {'high', 'medium', 'low'} do
        KM[capCmd][e]= max(count[capCmd][e]['high'],
        count[capCmd][e]['medium'],
        count[capCmd][e]['low'])
```

Part of the results (out of 46 commands)

Capability.command	Effect(+)	Effect(−)	Effect(o)
alarm.off	H	L	H
alarm.siren	L	H	H
light.off	L	M	M
location.setLocationMode	L	M	M
lock.lock	L	H	H
lock.unlock	H	M	H
switch.on	L	M	M
valve.close	L	H	H
valve.open	H	L	H