33RD USENIX SECURITY SYMPOSIUM

TAPFixer: Automatic Detection and Repair of Home Automation Vulnerabilities based on Negated-property Reasoning

Yinbo Yu[1,2], Yuanqi Xu[1], Kepu Huang[1], Jiajia Liu[1]

[1]National Engineering Laboratory for Integrated Aero-Space-Ground-Ocean Big Data Application Technology, School of Cybersecurity, Northwestern Polytechnical University, China

[2]Research & Development Institute of Northwestern Polytechnical University in Shenzhen, China

AUGUST 14–16, 2024
PHILADELPHIA, PA, USA

Home Automation (HA)

HA in daily life

Lighting Control

Security Monitoring

Smart Watering

Smart Cleaning

HA platforms

Samsung SmartThings

HomeKit
Apple Homekit

IFTTT
IFTTT

MI HOME

Home Assistant

openHAB

Trigger-Action Programming (TAP)

Paradigm of a TAP rule

IF a [Trigger] occurs **WHILE** a [Condition] satisfies , **THEN** perform an [Action].

Example

Keep the room at the proper temperature:

IF [motion sensor detects user returning home] , THEN [turn on heater].

[Trigger] [Action]

Power off before bedtime:

IF [11 p.m.] WHILE [motion sensor detects no one moving] , THEN [turn off all power].

[Trigger] [Condition] [Action]

Interaction Vulnerabilities

TAP rule1 : **IF** **[motion sensor detects user returning home]** , **THEN** **[turn on heater]**.
[Trigger] **[Action]**

TAP rule2 : **IF** **[11 p.m.]** **WHILE** **[motion sensor detects no one moving]** , **THEN** **[turn off all power]**.
[Trigger] **[Condition]** **[Action]**

	Secure Case	**Defective Case**
Interaction Pattern	Rules **run independently**	Rules **interact with each other**
Execution Sequence	Rule2 triggers **later than** rule1 since the user is usually home **before 11 p.m.**	Rule2 triggers **earlier than** rule1 since the user is home **after 11 p.m.**
Vulnerability	Nonexistent	The heater may still be running while sleeping, which may cause a **fire hazard**.

Motivation: Previous Approaches

Vulnerability Detection:

Model Checking-based / Symbolic Execution-based

Logical-physical Space:

- Determine correctness of rule interactions

- Latency l_1, l_2, l_3:

l_1: delay defined in rules for specifying the device execution time
l_2: delay on a tardy attribute change to a certain value
l_3: platform delay

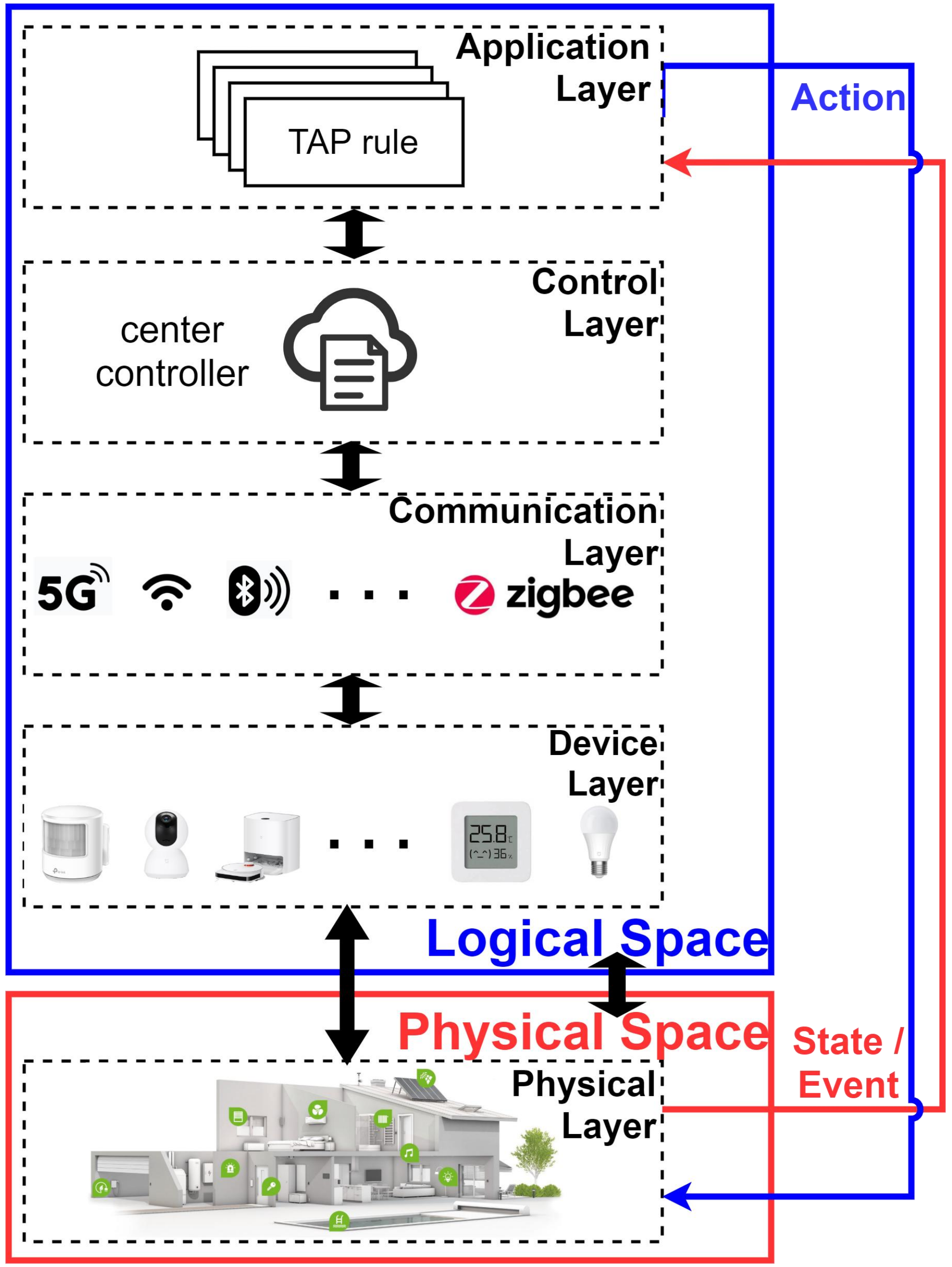
- Physical interaction phy_1,phy_2,phy_3:

phy_1: implicit physical effect phy_2: joint physical effect
phy_3: nondeterminacy

Limitation:

- Existing works neglect above key logical-physical features

- Fail to detect vulnerabilities with such features



Motivation: Previous Approaches

Vulnerability Repair:

Dynamic Access Control-based / Static Semantic Modification-based

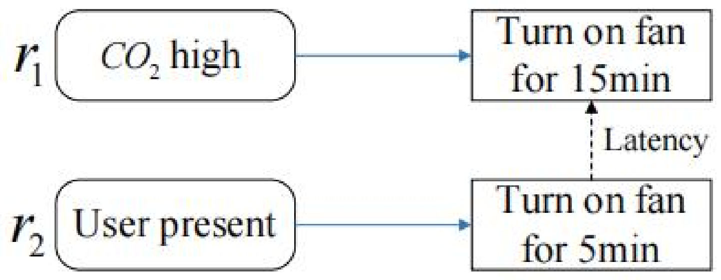
Limitation of Dynamic Approaches:

- Unable to fix flaws in rule semantics (root cause of vulnerabilities)
- Introduce additional runtime overhead

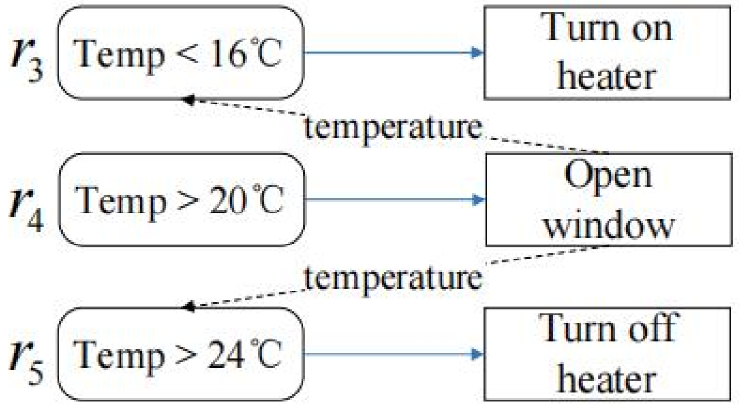
Limitation of Static Approaches:

- Not considering dynamic factors in physical space and fail to repair related expanded vulnerabilities.

Scenarios with Dynamic Factors



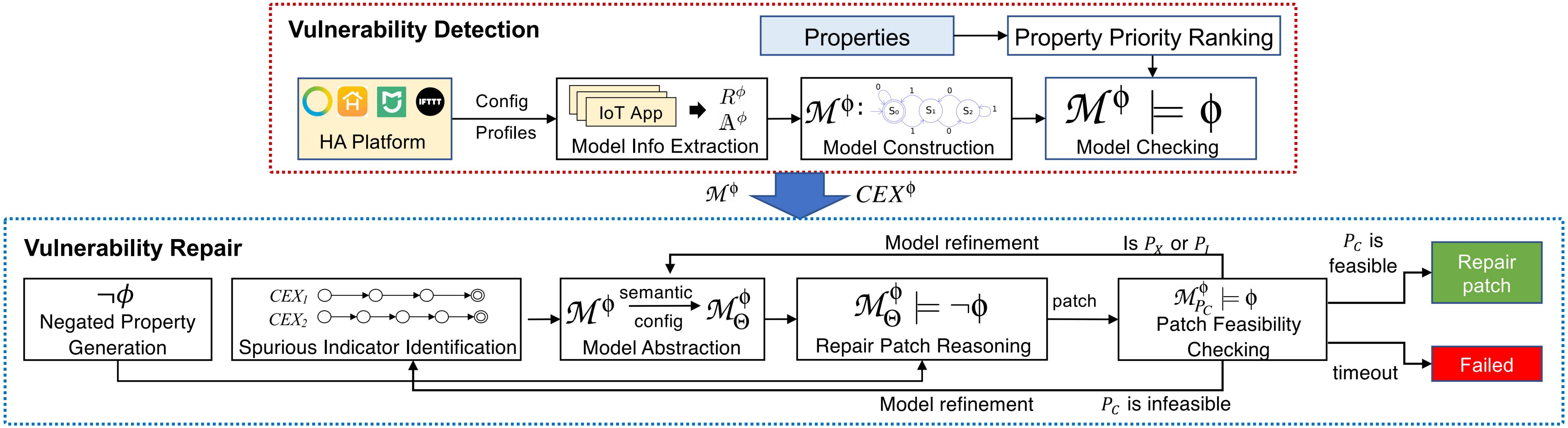
vulnerability with latency



vulnerability with implicit interactions

TAPFixer

To our best knowledge, TAPFixer is the first work that
can essentially detect and fix rule interaction vulnerabilities both in the logical and physical space.



Addressing Challenge 1: Comprehensive modeling logical-physical features

Physical model-based HA system modeling

1. Latency Modeling

$\boldsymbol{l_1}$: modeled as a timer configured with a timeout value
$\boldsymbol{l_2}$: set the physical changing threshold
$\boldsymbol{l_3}$: set the updating interval threshold

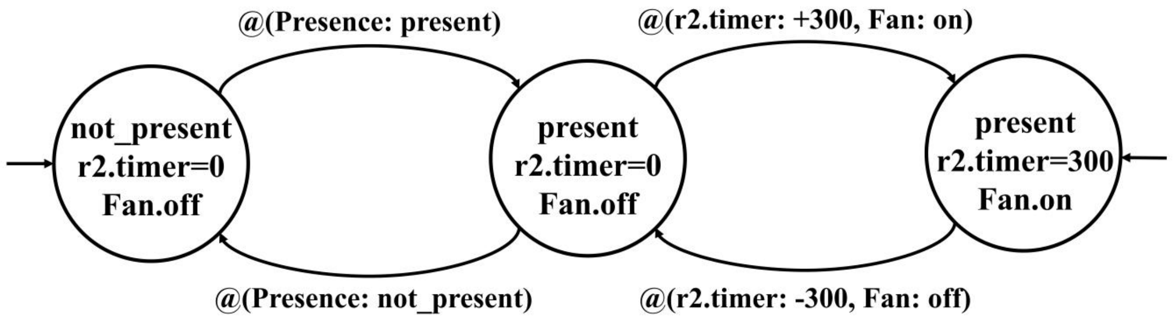
2. Physical Interaction Modeling

$\boldsymbol{phy_1}$: create a mapping of device actions to implicitly affected physical channels
$\boldsymbol{phy_2}$: modeled as the sum of the physical effects of the device's individual operation
$\boldsymbol{phy_3}$: modeled as a series of arbitrary values

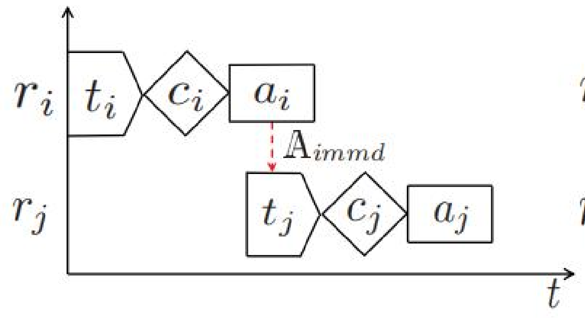
3. Finite Automata Construction

$$\mathcal{M}_{RI} := \{S, I, \sum\}$$

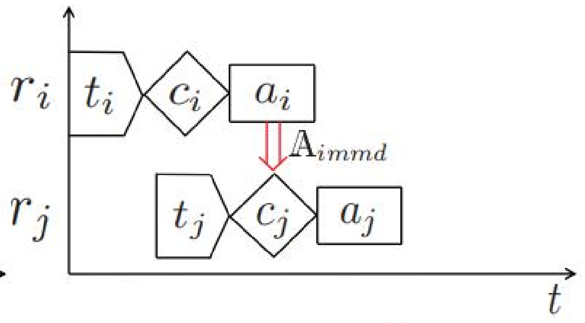
Automata state universal set *S*, initial state set *I*:
modeled as device and environment attributes
State transfer function φ ∈ ∑:
transfer conditions: modeled as trigger and condition
transfer label: modeled as action


@(Presence: present)
@(r2.timer: +300, Fan: on)
not_present r2.timer=0 Fan.off
present r2.timer=0 Fan.off
present r2.timer=300 Fan.on
@(Presence: not_present)
@(r2.timer: -300, Fan: off)

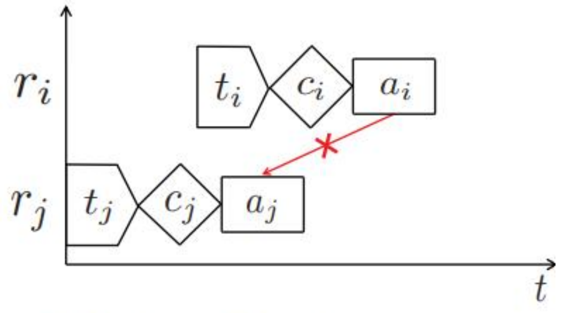
Addressing Challenge 2: Detecting expanded vulnerabilities



(a) **V1**: Trigger-Interference Basic Pattern.



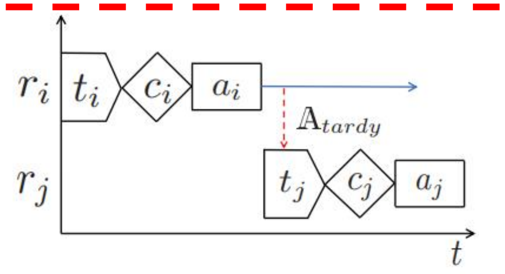
(b) **V2**: Condition-Interference Basic Pattern.



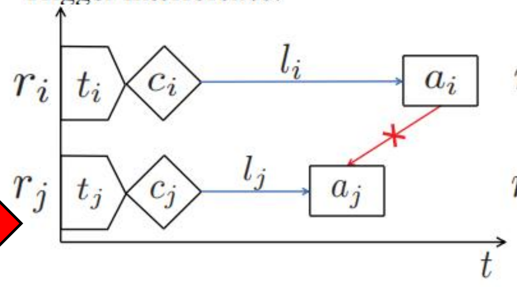
(c) **V3**: Action-Interference Basic Pattern.

Basic Vulnerability Pattern (V1-V3)

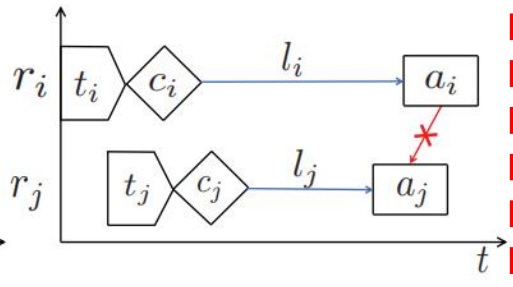
Latency, physical features



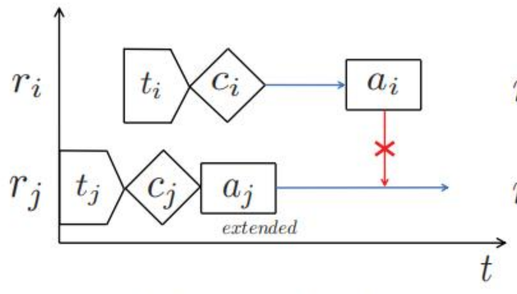
(d) **V4**: Tardy-channel-based Trigger Interference.



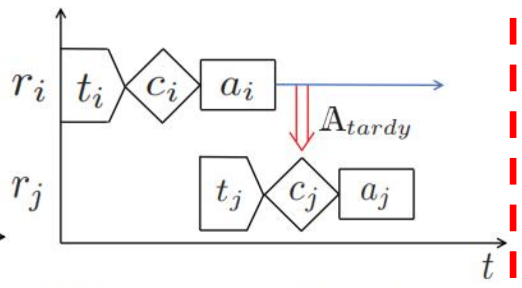
(e) **V5**: Disordered Action Scheduling.



(f) **V6**: Action Overriding.



(g) **V7**: Action Breaking.



(h) **V8**: Tardy-channel-based Condition Interference.

Expanded Vulnerability Pattern (V4-V8)

Addressing Challenge 2: Detecting expanded vulnerabilities

Correctness property categorizing and ranking-based vulnerability detection

- **Correctness Property:** a criterion to describe what automation behavior is safe or not.
- Categorize 9 language templates of properties into 2 logical templates for property specification

Table 12: Logically equivalent correctness property types.

Summarised property types	Property types	Natural language templates
Event-based	One-Event Unconditional	[*event*] should [*never*] happen
	Event-State Conditional (always)	[*event*] should [*always*] happen when [$state_1$,..., $state_n$]
	Event-State Conditional (never)	[*event*] should [*never*] happen when [$state_1$,..., $state_n$]
State-based	One-State Unconditional (always)	[*state*] should [*always*] be active
	One-State Unconditional (never)	[*state*] should [*never*] be active

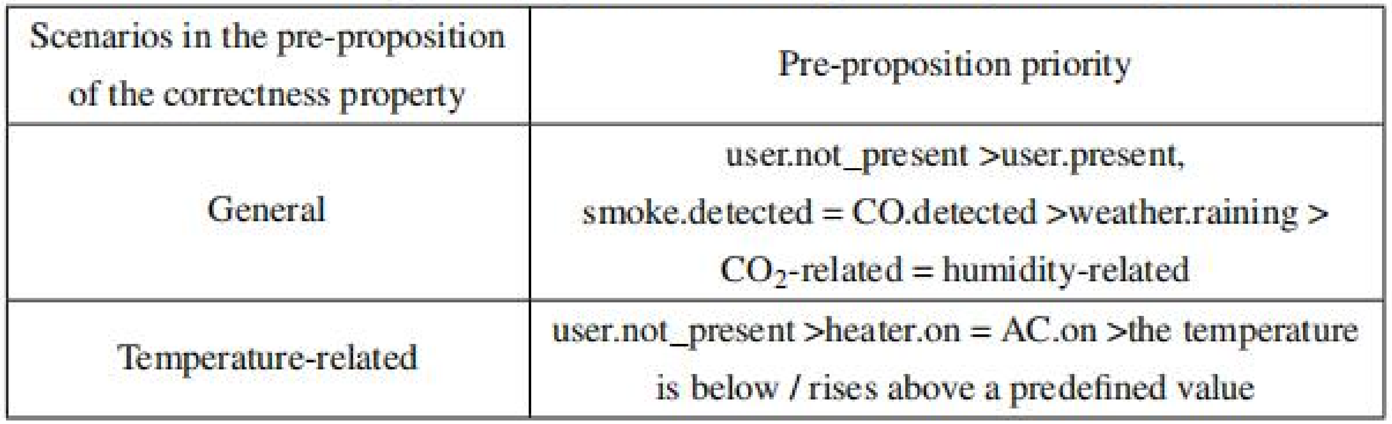
Addressing Challenge 2: Detecting expanded vulnerabilities

● Define **pre- and post-proposition priority ranking** to resolve **property violations**.

P.1: close the window if it rains
P.2: open the window if CO is detected

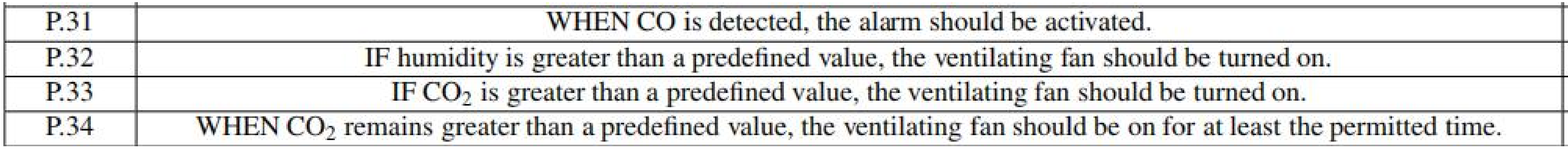
If it rains and CO is detected
Close or open window ??

Table 13: Sorting descriptions of the pre-proposition priority.



Scenarios in the pre-proposition of the correctness property	Pre-proposition priority
General	user.not_present >user.present, smoke.detected = CO.detected >weather.raining > CO_2-related = humidity-related
Temperature-related	user.not_present >heater.on = AC.on >the temperature is below / rises above a predefined value

● Supply more properties for different scenarios (e.g., **properties with permitted latencies P.34**), finally conduct 53 properties for vulnerability detection.



P.31	WHEN CO is detected, the alarm should be activated.
P.32	IF humidity is greater than a predefined value, the ventilating fan should be turned on.
P.33	IF CO_2 is greater than a predefined value, the ventilating fan should be turned on.
P.34	WHEN CO_2 remains greater than a predefined value, the ventilating fan should be on for at least the permitted time.

● If the vulnerability exists, a system execution path that violates the correctness property (i.e., **counterexample**) will be returned by the model checker

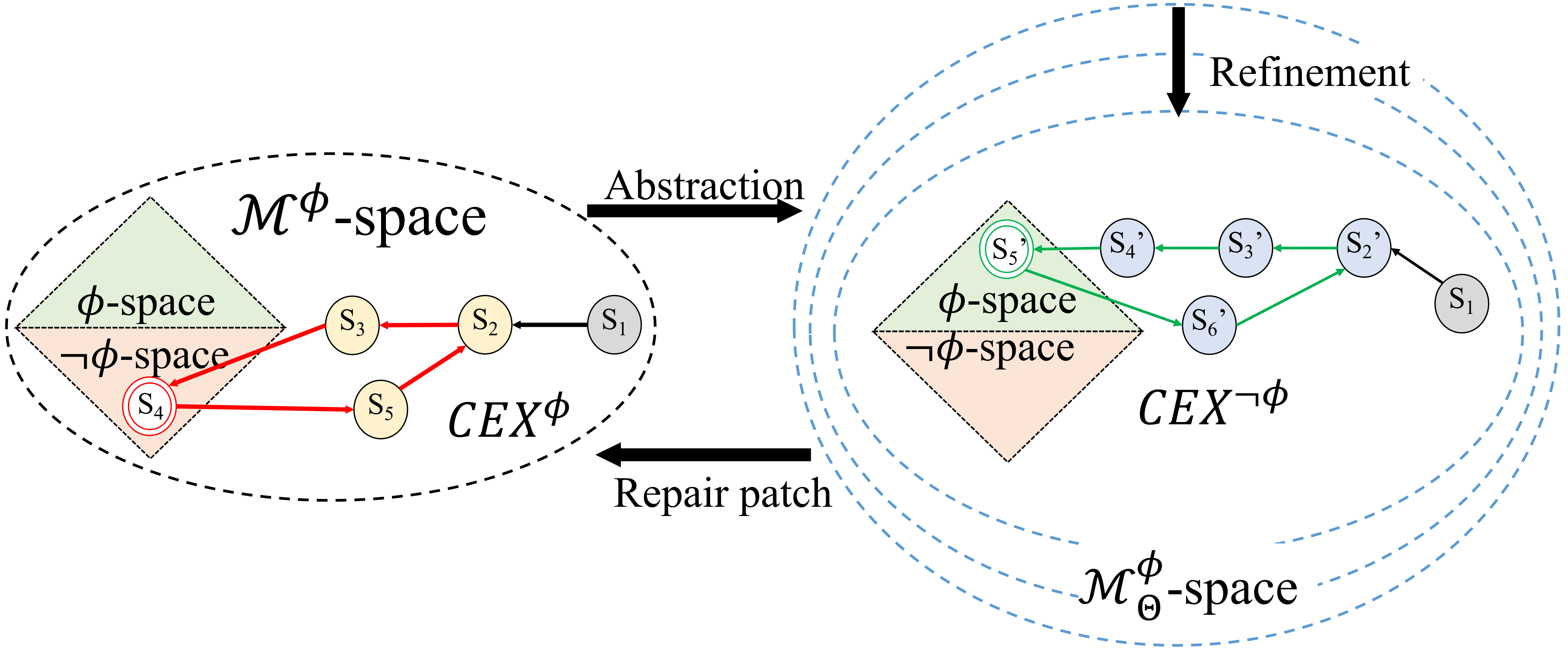
Addressing Challenge 3: Repairing expanded vulnerabilities

	Vulnerability Detection	**How to repair vulnerability?** **Negated-property Reasoning (NPR) Algorithm**
Applied Property	**Correctness Property $\boldsymbol{\phi}$:** IF the user is not at home, the heater should be turned off.	**Negated-property $\neg\boldsymbol{\phi}$ logically opposite to $\boldsymbol{\phi}$:** IF the user is not at home, the heater should be turned on **(negated)**.
Verification Process	model checking with $\boldsymbol{\phi}$ on model $\mathcal{M}$ ($\mathcal{M} \vDash \boldsymbol{\phi}$)	model checking with $\neg\boldsymbol{\phi}$ on model $\mathcal{M}$ ($\mathcal{M} \vDash \neg\boldsymbol{\phi}$)
Verification Result	Return a violation of $\boldsymbol{\phi}$ in $\mathcal{M}$: **scenarios where no one is home but the heater is running (fire hazard)**	Return a violation of $\neg\boldsymbol{\phi}$ in $\mathcal{M}$: **scenarios where no one is home and the heater is off (potential fix information)**

- **Secure scenario reasoned by $\neg\boldsymbol{\phi}$ can fix vulnerability violated $\boldsymbol{\phi}$**
- Reasoned result of $\neg\boldsymbol{\phi}$ does not violate $\boldsymbol{\phi}$, i.e., the reasoned result of $\neg\boldsymbol{\phi}$ constitutes the repair space for the vulnerability violated $\boldsymbol{\phi}$

Addressing Challenge 3: Repairing expanded vulnerabilities

Negated-property reasoning (NPR) algorithm



● **The core idea of our NPR algorithm for vulnerability repair.**

● **Model abstraction via interpolation is used to involve a larger state space for patch searching, so the negated counterexample CEX$\neg\phi$ can contain repair patches for the vulnerability CEXϕ.**

Addressing Challenge 3: Repairing expanded vulnerabilities

Negated-property reasoning (NPR) algorithm

Step1: Negated Property Generation & Spurious Indicator Identification

- Negate the latter part of the LTL template divided by "⇒" to generate the negated property
- **Spurious Indicator:** assess whether a patch can fix the vulnerability
- Spurious indicator is the violating state in the counterexample

Step2: Patch Reasoning

- Limited repair information in the state space of model $\mathcal{M}$
- Abstract model $\mathcal{M}$ to $\mathcal{M}_{\Theta}^{\phi}$
- Reason patch P from abstract model $\mathcal{M}_{\Theta}^{\phi}$

Step3: Patch Feasibility Checking

- **Patch Category**
- **Feasibility Checking**
- **Reasoning-guided Abstraction Refinement**

Evaluation

Case Study of Vulnerability Detection and Repair Accuracy

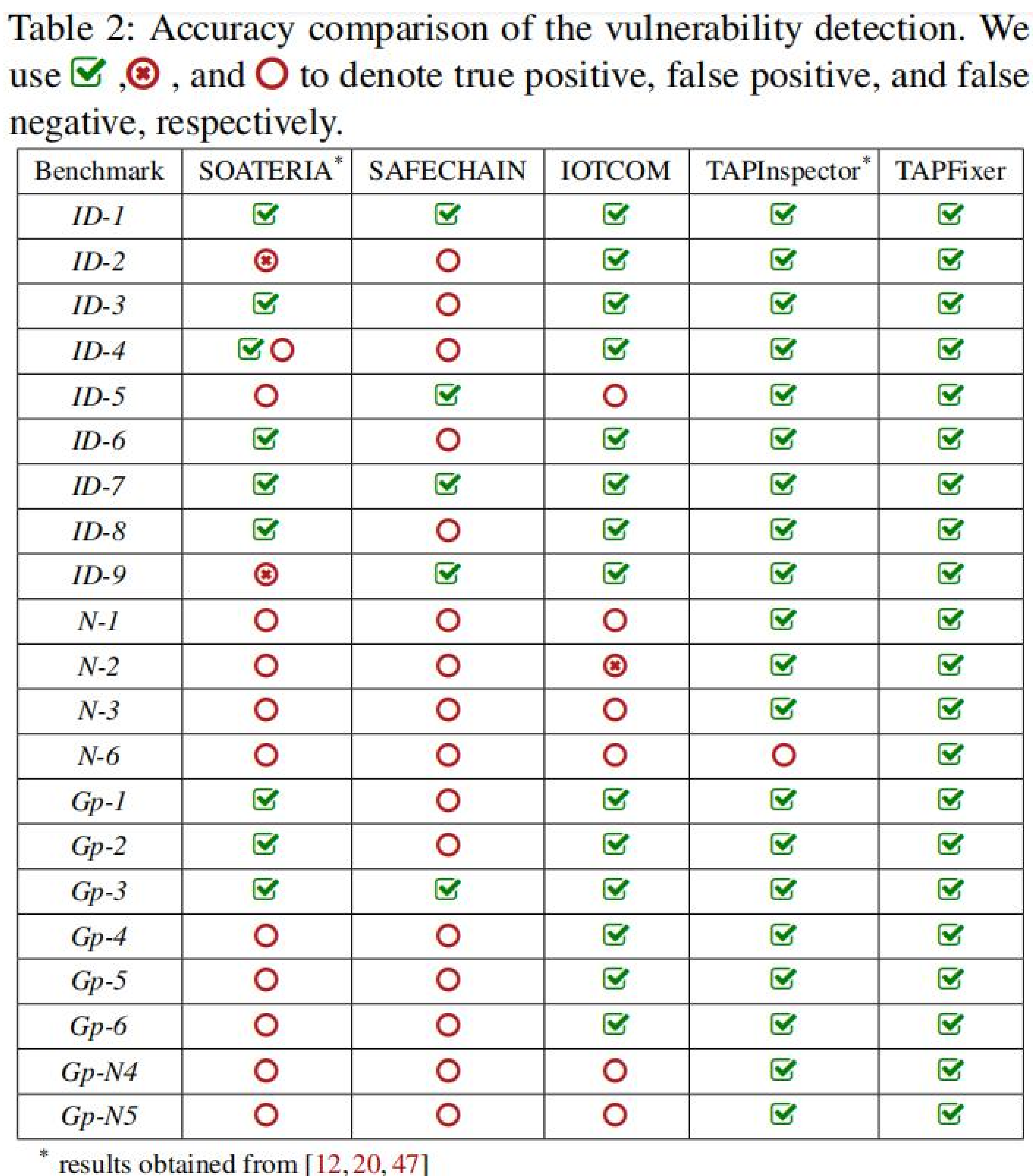


Table 2: Accuracy comparison of the vulnerability detection. We use ☑ , ⊗ , and ○ to denote true positive, false positive, and false negative, respectively.

Benchmark	SOATERIA[*]	SAFECHAIN	IOTCOM	TAPInspector[*]	TAPFixer
ID-1	☑	☑	☑	☑	☑
ID-2	⊗	○	☑	☑	☑
ID-3	☑	○	☑	☑	☑
ID-4	☑○	○	☑	☑	☑
ID-5	○	☑	○	☑	☑
ID-6	☑	○	☑	☑	☑
ID-7	☑	☑	☑	☑	☑
ID-8	☑	○	☑	☑	☑
ID-9	⊗	☑	☑	☑	☑
N-1	○	○	○	☑	☑
N-2	○	○	⊗	☑	☑
N-3	○	○	○	☑	☑
N-6	○	○	○	○	☑
Gp-1	☑	○	☑	☑	☑
Gp-2	☑	○	☑	☑	☑
Gp-3	☑	☑	☑	☑	☑
Gp-4	○	○	☑	☑	☑
Gp-5	○	○	☑	☑	☑
Gp-6	○	○	☑	☑	☑
Gp-N4	○	○	○	☑	☑
Gp-N5	○	○	○	☑	☑

[*] results obtained from [12, 20, 47]

- Benchmark contains expanded vulnerabilities V4-V8
- TAPFixer is more effective at identifying and repairing expanded vulnerabilities

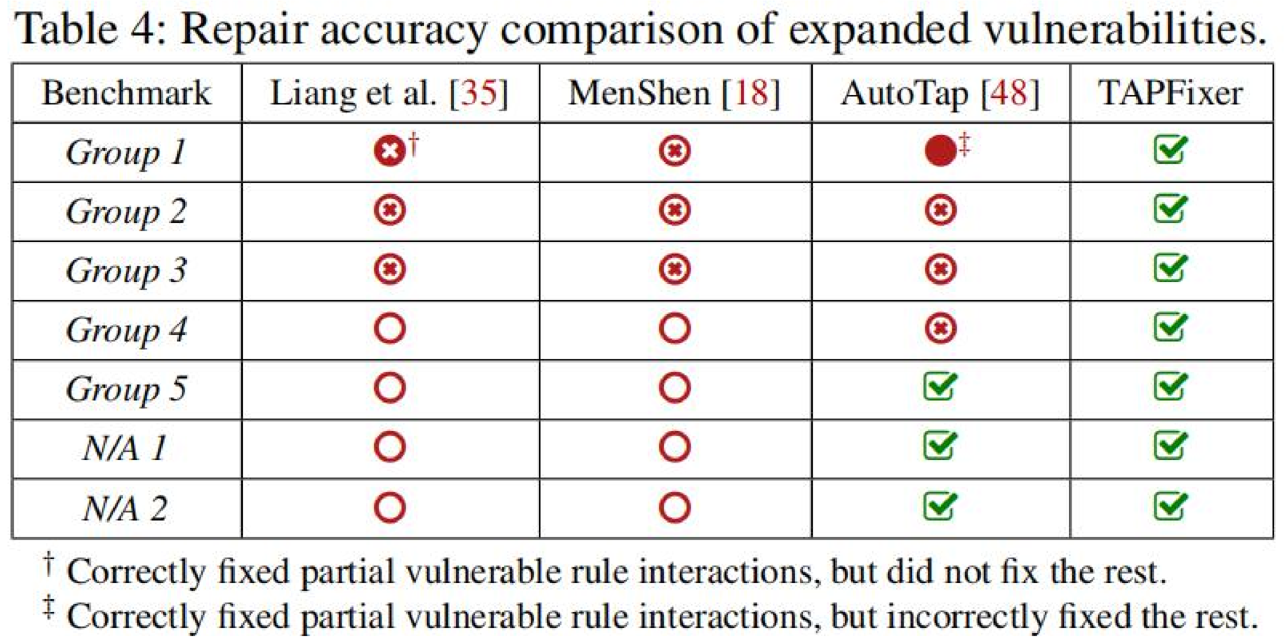


Table 4: Repair accuracy comparison of expanded vulnerabilities.

Benchmark	Liang et al. [35]	MenShen [18]	AutoTap [48]	TAPFixer
Group 1	⊗[†]	⊗	●[‡]	☑
Group 2	⊗	⊗	⊗	☑
Group 3	⊗	⊗	⊗	☑
Group 4	○	○	⊗	☑
Group 5	○	○	☑	☑
N/A 1	○	○	☑	☑
N/A 2	○	○	☑	☑

[†] Correctly fixed partial vulnerable rule interactions, but did not fix the rest.
[‡] Correctly fixed partial vulnerable rule interactions, but incorrectly fixed the rest.

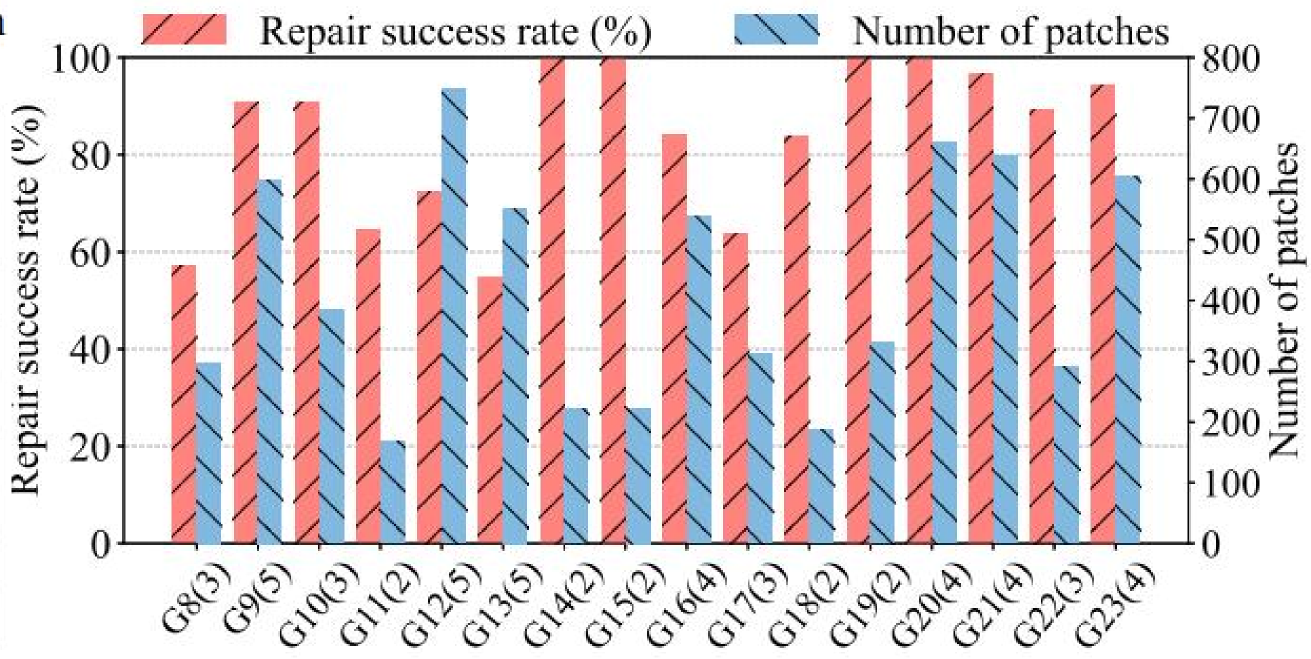
Evaluation

Market App Study of Vulnerability Repair

- **1177 TAP rules** from SmartThings SmartApp and IFTTT applets and **110 test groups**
- Scenario-based Vulnerability Repair：
  repair 4544 of 5244 found vulnerabilities **Repair Success Rate (RSR): 86.65%**
- Priority-based Violation Repair：
  repair 4460 of 5335 found vulnerabilities **Repair Success Rate (RSR): 83.60%**

Table 5: Summary of detection and repair results for G1-G7 with 110 rule groups, each of which contains 15-30 TAP rules.

Application scenarios	Fixed violations	Unfixable violations	Safe cases	Generated patches	RSR↑
G1 (2 properties)	179	35	6	364	83.64%
G2 (21 properties)	1675	277	358	2228	85.81%
G3 (6 properties)	459	201	0	902	69.55%
G4 (8 properties)	687	59	134	1145	92.09%
G5 (9 properties)	870	68	52	1288	92.75%
G6 (3 properties)	272	58	0	491	82.42%
G7 (4 properties)	402	2	36	419	99.50%
Total	4544	700	586	6837	86.65%



Evaluation

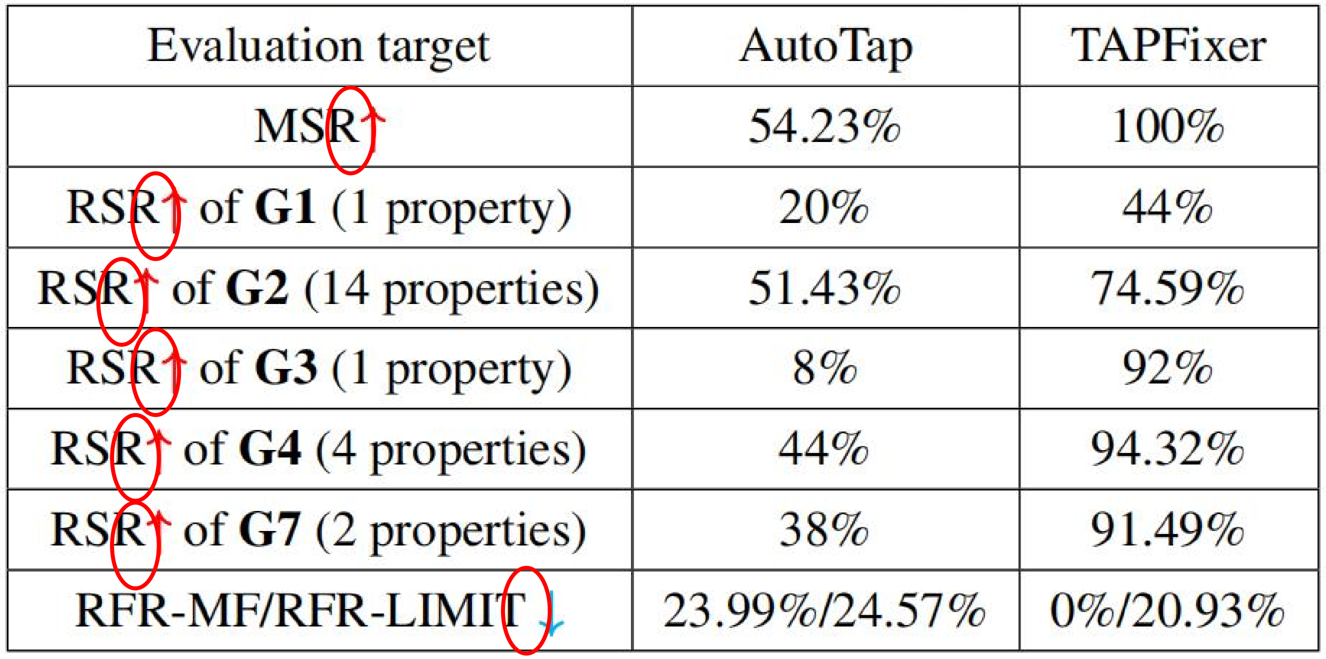
Market App Study of Vulnerability Repair

- Comparison with the SOTA approach
- **Modeling Success Rate (MSR)**: assess the integrity of rule modeling
- **Repair Failure Rate (RFR)**

  RFR-MF: RFR caused by modeling failures

  RFR-LIMIT: RFR caused by modeling failures repair algorithm limitations

Table 6: Comparison between AutoTap and TAPFixer.

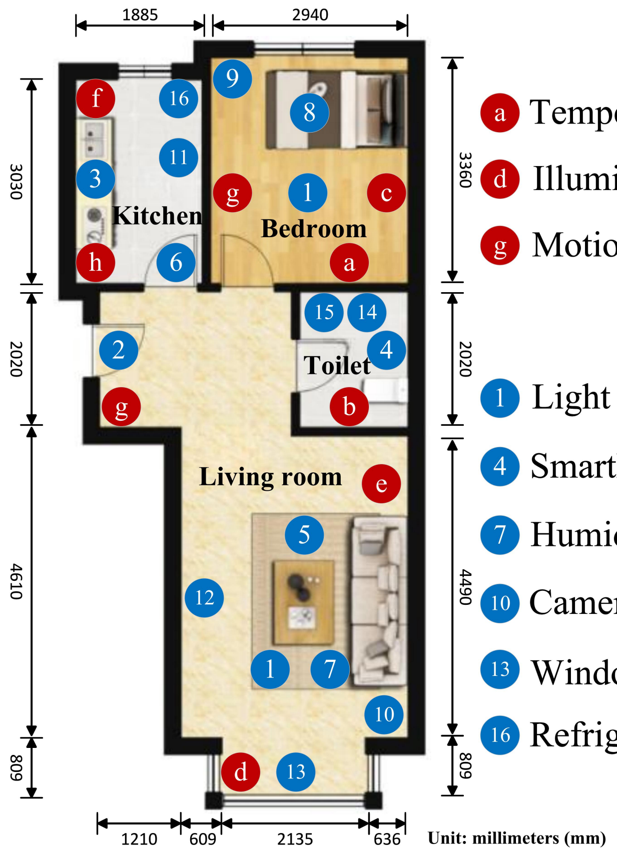


Evaluation target	AutoTap	TAPFixer
MSR↑	54.23%	100%
RSR↑ of **G1** (1 property)	20%	44%
RSR↑ of **G2** (14 properties)	51.43%	74.59%
RSR↑ of **G3** (1 property)	8%	92%
RSR↑ of **G4** (4 properties)	44%	94.32%
RSR↑ of **G7** (2 properties)	38%	91.49%
RFR-MF/RFR-LIMIT↓	23.99%/24.57%	0%/20.93%

Evaluation

User Survey on the Quality of Vulnerability Detection and Repair

- RSR: **94.5%**
- Satisfaction Rate of the Detection and Repair Quality： **99%**



Sensors

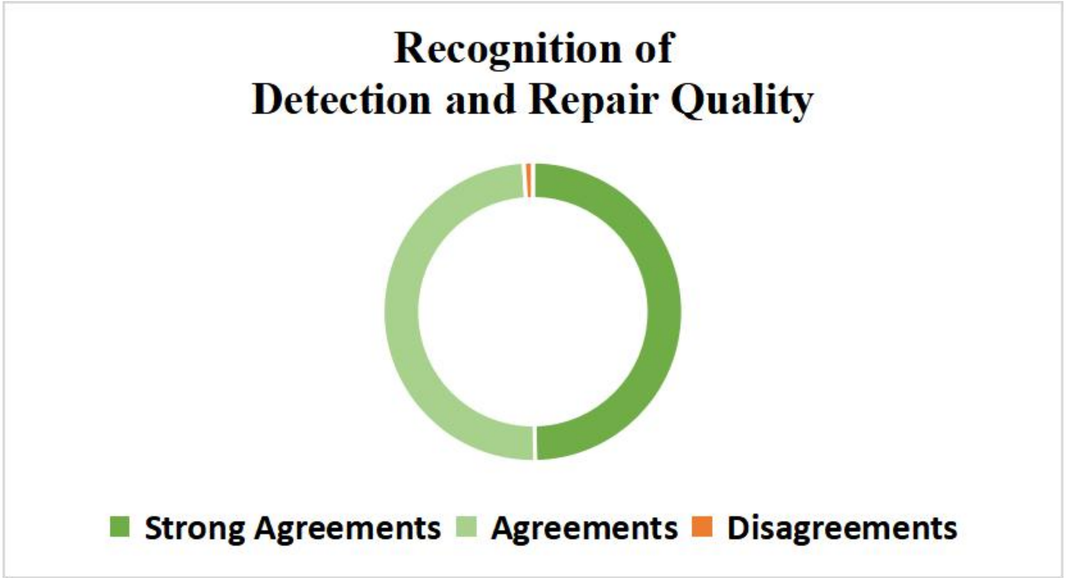
a Temperature	b Humidity	c CO_2
d Illuminance	e Smart Speaker	f CO
g Motion	h Smoke	

Actuators

1 Light Bulb	2 Door Lock	3 Cooker
4 SmartFan	5 Heater	6 Alarm
7 Humidifier	8 Electric Blanket	9 AC
10 Camera	11 Sprinkler	12 TV
13 Window	14 Handwash	15 Dryer
16 Refrigerator		

Table 7: Number of identified and fixed vulnerabilities in 129 rules.

	V1	V2	V3	V4	V5	V6	V7	V8
# found violations	5	6	9	23	0	4	5	3
# fixed violations	3	6	8	23	0	4	5	3
RSR	60%	100%	89.9%	100%	N/A	100%	100%	100%



Evaluation

Performance

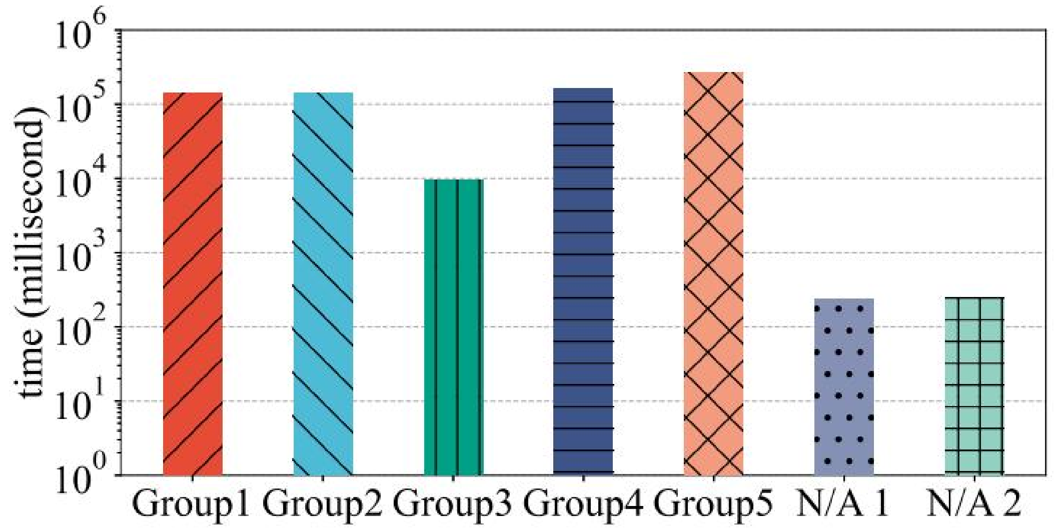


Figure 7: Verification and repair time of each 21-rule benchmark dataset and initialization scenarios.

For case study:

- 21-rule benchmarks (Group 1-5)
  Average time **2.69 min**

- Initialization Scenarios (N/A 1-2)
  Average time **228 ms**

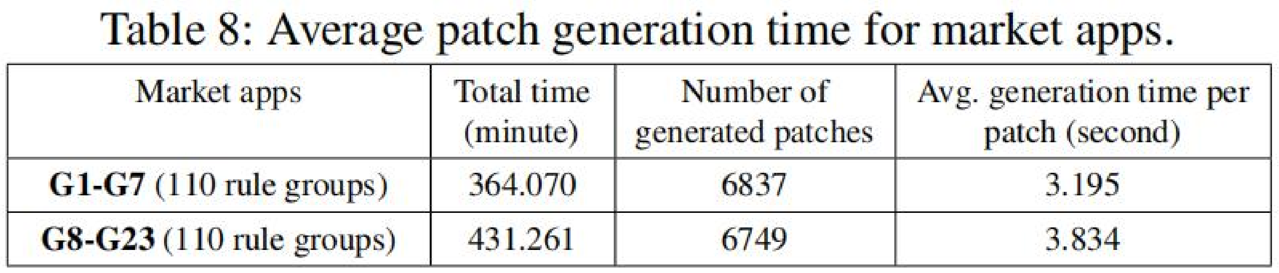


Table 8: Average patch generation time for market apps.

Market apps	Total time (minute)	Number of generated patches	Avg. generation time per patch (second)
G1-G7 (110 rule groups)	364.070	6837	3.195
G8-G23 (110 rule groups)	431.261	6749	3.834

For 110 test groups in market apps：

- Average time **6.62 h**

- Average time to reason a patch **3.51s**

Conclusion

- We propse the physical model-based HA system modeling that can model rules with **more practical latency and physical features**.
- We propose the correctness property categorizing and ranking-based vulnerability detection that can **identify more expanded interaction vulnerabilities**.
- We propose **a novel negated-property reasoning algorithm (NPR)** that can accurately **generate valid patches for eliminating vulnerabilities both in the logical and physical space.**
- We implement **TAPFixer**, **the first framework** that can essentially detect and repair rule interaction vulnerabilities **both in the logical and physical space**.
- TAPFixer **achieves very good results** from aspects of accuracy analysis, repair capabilities of market apps, real user study, and execution performance.

Thanks

Q&A