2019 IEEE Security and Privacy Workshops (SPW)

Analysis of the Susceptibility of Smart Home Programming Interfaces to End User Error

Mitali Palekar, Earlence Fernandes, Franziska Roesner
Paul G. Allen School of Computer Science & Engineering
University of Washington

***Abstract*—Trigger-action platforms enable end-users to program their smart homes using simple conditional rules of the form: if *condition* then *action*. Although these rules are easy to program, subtleties in their interpretation can cause users to make errors that have consequences ranging from incorrect and undesired functionality to security and privacy violations. Based on prior work, we enumerate a set of nine classes of errors that users can make, and we empirically study the relationship between these classes and the interface design of eight commercially available trigger-action platforms. Particularly, we examine whether each interface prevents (e.g., via good design) or allows each class of error. Based on this analysis, we develop a framework to classify errors and extract insights that lay a foundation for the design of future trigger-action programming interfaces where certain classes of errors can be mitigated by technical means or by alerting the user of the possibility of an error. For instance, we identify that an analysis of a dataset of functionally-similar trigger-action rules could be used to predict whether certain types of error patterns are about to occur.**

I. Introduction

Smart home platforms and devices, such as Samsung SmartThings, Amazon Echo, Google Home, Phillips Hue lights, Nest thermostats and cameras, and many others, are being increasingly deployed in the homes of end users. One value proposition of smart homes is their ability to be programmed and automated by their users—for example, users can create rules to turn their lights on when someone arrives at home, or to send themselves a notification when a door is opened. This end-user programming is typically done via a *trigger-action framework* that enables users to program *if-then* rules, e.g., "*if* the door opens, *then* send me a notification". This programming paradigm appears both in native smart home apps (e.g., Samsung's SmartThings app) as well as third-party services that can be integrated with a user's smart devices (e.g., If-This-Then-That, or SmartRules).

Unfortunately, prior work (e.g., [5, 2, 6, 8]) has found that end users may make errors when they program trigger-action rules. These errors may stem from simply forgetting to program part of the intended behavior (e.g., programming a rule to turn on the lights but no rule to turn them back off) or from more fundamental misunderstandings of the trigger-action framework (attempting to program rules whose behavior is not logically well-defined). The impact of these errors can range from mere annoyance to serious security or privacy concerns—for example, incorrectly unlocked doors, fire hazards, or privacy risks due to smart cameras.

Moreover, debugging these kinds of errors after the fact can be challenging for users. A user may not realize anything is amiss until some time after a rule has been programmed and the user observes something strange happening in their physical home (e.g., the lights turning off at the wrong time). Depending on the number and complexity of the user's setup and automations, it may be non-trivial to track down exactly what went wrong. Ideally, instead, a user's trigger-action programming errors should be prevented at programming time, by the trigger-action programming platform itself.

In this work, we thus ask: *do* current trigger-action platforms prevent users from making these types of errors? We empirically study eight commercially available trigger-action platforms that support smart home integrations. For each platform, we investigate whether its programming interface allows users to make the types of errors we collected from prior work, and how it prevents errors (if any).

What we find suggests room for improvement. For most of the platforms we analyze, we find that they either disallow errors by lacking the features that would make the error possible in the first place (e.g., the ability to have trigger conjunctions, i.e., "if x *and* y, then") or they disallow errors that do not make logical sense. (For example, Stringify and SmartRules both prevent users from creating rules such as *if the light is on and the door is open, do something*. It is unclear when this rule should run, as both triggers refer to continuous states, the light being on and the door being open.) By contrast, interfaces currently do nothing to prevent errors that fall into a grey area: they are logically sensible but may not have been intended by the user.

We then classify these errors that are currently allowed by programming interfaces based on whether they can be automatically detected and/or prevented by trigger-action platforms. We observe that while some errors can be automatically detected and prevented based on logic alone—such as those prevented by Stringify and SmartRules—others rely fundamentally on the user's intention. For example, a rule that poses a potential privacy violation by posting photos to Facebook [6] is only problematic if it does not match the

DOI 10.1109/SPW.2019.00034

user's intention; likewise for a rule where the user forgot to include one of two intended trigger conditions (e.g., "if the light turns on and it after 6pm" versus simply "if the light turns on"). We propose strategies that trigger-action programming interfaces and researchers should explore in future work to help users avoid these types of errors. For example, by leveraging a corpus of common rules from many users, a platform may be able to detect and suggest *possible* omissions in a user's ruleset.

In summary, our main contributions are as follows:

1) We consolidate (based on prior work) a set of errors that end users of smart home trigger-action platforms may make.
2) We empirically evaluate the end-user programming interfaces of eight commercially available smart home trigger-action platforms with respect to whether they allow users to make these errors.
3) We classify the errors according to whether they could be automatically detected and/or prevented by the platform.
4) We make concrete recommendations for both the design of end user trigger-action programming interfaces as well as for future research to help reduce all types of end user errors.

II. Related Work

Various aspects of trigger-action programming have been studied. Early work by Ur *et al.* characterized 224,590 user-created trigger-action rules from the IFTTT platform, discussing statistics like the number of times the rules have been shared and the types of online services being used [7]. A large body of work has built upon this to understand the errors *end-users* may make while programming trigger-action rules [2, 6, 5, 8]. Huang *et al.* discuss how trigger conjunctions can lead to subtle logic errors [2]. Surbatovich *et al.* discuss rules that create privacy or integrity violations as well as unexpected interactions between user-created rules [6]; Yarosh *et al.* discuss interactions between rules [8]. Nandi *et al.* discuss errors resulting from users programming too few trigger conditions [5].

Our work systematizes these prior studies and presents a characterization of the types of errors. Furthermore, these prior works analyze the errors either in the context of hypothetical trigger-action platforms or systems intended for technically skilled programmers (e.g., OpenHab). Thus, it is unclear whether such errors are likely to happen in the context of commercially available platforms for non-technical end users. Our work fills this gap, and reports on an empirical study of eight commercial programming interfaces about whether such programming errors are prevented, or can occur. In concurrent work, Brackenbury *et al.* also identify similar classes of trigger-action programming bugs, and investigate how users reason about them (but do not evaluate commercially available platforms) [1].

Prior work has proposed automated platform-level mechanisms to detect and prevent certain classes of errors [4, 6, 5]. For example, Surbatovich *et al.* propose information flow analyses to detect when a privacy violation might occur [6], and Nandi *et al.* propose a static analysis tool to vet rules for missing triggers under certain conditions [5]. Expanding on these possible prevention strategies, our work takes a wide view of the types of errors that can occur, observing that identifying and correcting some errors requires understanding the user's intent in creating a rule. We propose additional mitigation strategies for end-user errors in Section VI-C.

III. Background

Trigger-action programming involves simple rules of the form: if *condition* then *action*. For example, "if the door is opened, then send me a notification." The simplicity and utility of trigger-action programming makes it common in home automation. Trigger-action programming platforms often include:

Event triggers are instantaneous signals [2] such as *the door is opened*. The event occurs at the moment in time when the door transitions from being closed to being opened.

State triggers are boolean conditions that evaluate to a specific value for a period of time [2]. For example, *the door is open* can be true for any amount of time. In this case, the state trigger is true for the period of time the door is open.

Actions are operations that are initiated by the trigger-action platform when a rule is executed. For example, *unlock the door* or *send a text message*.

Trigger conjunctions are the boolean "and" of two or more triggers in an unspecified order. For example, *the light is on AND a person is in the room*.

Action conjunctions are the boolean "and" of two or more actions in an unspecified order. For example, *turn the light on AND open the door*.

Conditions on triggers allow filtering whether an action is invoked based on the value of a trigger attribute. For example, *"if I receive an email AND the subject contains URGENT"* contains a condition on the trigger.

Do While is a feature where actions are executed until a certain condition is not met. For example, *"If no one is home and it is past 9pm, turn the sprinklers on every 30 minutes until someone comes home or it is past 6am."*

Integration with smart devices such as smart lights and locks via SmartThings and/or **non-smart home devices** such as cloud services (e.g., GMail) might be enabled in current trigger-action platforms. In this work, we primarily consider interfaces that are intended for use by end users, especially in smart home contexts (rather than business logic trigger-action use cases).

IV. Error Systematization

We consolidate a list of errors (i.e., situations where the user's intended behavior and platform's actual rule execution are different) based on prior work. We focus only on errors that can me made by end users rather than platform-level bugs or vulnerabilities.

Lack of action reversal occurs when a user programs only one half of an intended rule and forgets the other half [2]. For example, a user might program a rule to *turn the lights on at 6pm*, but forget to program a rule to *turn the lights off at 12am*. This type of error can only occur when the action causes a change that is true for a sustained period of time and needs to be reversed at the end of that period of time (a light once turned on will remain on). By contrast, an action such as *add an entry to the database* has no associated "off" state.

Feature interaction occurs when multiple rules interact with each other, creating a logical conflict in determining the resulting state [8]. For example, suppose a home system includes a rule to lock all doors at night and another rule to unlock the door if someone comes home [8]. In this case, after someone has arrived home at night, there may be uncertainty about whether the door would be locked or unlocked afterwards due to a logical conflict in the rules.

Feature chaining occurs when the action of one rule sets off the trigger for another rule. There are two types of feature chaining: direct linking and physical connections. Rules A and B are directly linked if A's action channel and B's trigger channel are the same and A's action fires B's trigger [6]. For example, *send an email to my email account* will directly fire the trigger *I received a new email*. Rules A and B are linked via physical connections if A's action channel affects a physical-world state, such as temperature or illumination, that causes Rule B's trigger to be fired.

Event+event rules conjoin two or more instantaneous signals (i.e., events) as the trigger [2]. In practice, two events are highly unlikely to occur at the same time; as such, this type of rule is too narrow for what users are trying to achieve. For example (from prior work [2]), suppose a user wants to send themselves an email when they are on time to work. If they program the rule *"If I arrive at work and it is 9am, then send myself an email"*, this rule would only send them an email if they arrive at work *exactly* when it turns 9am (relatively unlikely), but not if they arrive before 9am (as probably intended).

State+state rules conjoin two or more states as the trigger [2]. For example, the rule *if the door is open and the light is on* should be avoided as it is unclear whether the action should fire when this dual state *becomes* true (an event), or continuously while both states are true (which is technically what the rule states). Prior work [2] has shown that users have unclear mental models about how the platform does or should handle such state triggers. Due to this ambiguity, such rules lead to errors and should be avoided.

Missing trigger errors occur when the set of triggers used is insufficient to attain the desired functionality. For example, a user may program *"If it is 11pm, turn off the porch light"* and be unhappy to find the light off when they return from a late night out, having forgotten the additional trigger condition *and I am home*.

Missing action errors occur when the set of actions is insufficient to attain the desired functionality. For example, a home owner might want to ensure that all doors are closed at night, and might program the rule: *"If it is 9pm, then lock the front and back door"*, forgetting to lock the basement door.

Secrecy violations occur when private information leaks to a public audience [6]. For example, the rule *"If I receive a private message, post a public Facebook status with its contents"* contains a secrecy violation.

Integrity violations occur when information from a less trusted information source influences a more trusted sink, potentially corrupting it [6]. For example, the rule *"If there is a new Instagram photo by anyone in my area, turn on the smart light"* creates an integrity violation as information from an untrusted source (Instagram, in this example) affects the state of a possibly safety- or security-sensitive physical device.

V. Analysis Methodology

Prior studies have surfaced the possibility of user-created errors, but left open the question of whether these errors are likely to occur in commercially available platforms. In this section, we discuss our methodology to investigate this possibility.

We evaluate eight current end-user programming interfaces based on the above set of errors and features: Amazon Alexa[1], automate.io[2], Google Home[3], IFTTT[4], Microsoft Flow[5], SmartRules[6], Stringify[7], and Zapier[8]. We choose these platforms based on whether they were appropriate for smart home programming, and whether they supported a feature set that is compatible with our consolidated set of errors. Our results are based on the publicly available versions of these platforms as of January 21, 2019.

Two researchers independently evaluated whether errors can occur in a given programming interface. All researchers used a set of canonical example rules to determine whether a particular type of error can occur. If there were discrepancies in the analysis, the researchers came to a consensus based on discussion and additional investigation of the interface. To evaluate each end-user programming interface, we used the following scale:

1) $Possible$: Error is possible and the interface does nothing to prevent it.
2) NA ($Not\ Applicable$): Error is impossible due to a feature limitation by the interface.
3) $Prevented$: Error is impossible as it is prevented by the interface.

Some errors, such as feature chaining, depends on the specific triggers, actions, and devices supported by

[1]https://alexa.amazon.com
[2]https://automate.io
[3]https://store.google.com/product/google_home
[4]https://ifttt.com
[5]https://flow.microsoft.com
[6]http://smartrulesapp.com
[7]https://www.stringify.com
[8]https://zapier.com

TABLE I. End User Error Susceptibility of Smart Home Programming Interfaces

Type of Error	**Amazon Alexa**	**automate.io**	**Google Home**	**IFTTT**	**Microsoft Flow**	**SmartRules**	**Stringify**	**Zapier**
Lack of Action Reversal	*Possible*	*Possible*	*Possible*	*Possible*	*Possible*	*Possible*	*Possible*	*Possible*
Feature Interaction	*Possible*	*Possible*	*Possible*	*Possible*	*Possible*	*Possible*	*Possible*	*Possible*
Feature Chaining	NA*	*Possible*	NA	*Possible*	*Possible*	*Possible*	*Possible*	*Possible*
Event+Event Rule	NA	NA	NA	NA	NA	*Prevented*	*Prevented*	NA
State+State Rule	NA	NA	NA	NA	NA	*Prevented*	*Prevented*	NA
Missing Triggers	NA	NA	*Possible*	NA	NA	*Possible*	*Possible*	NA
Missing Actions	NA	*Possible*	*Possible*	NA	*Possible*	*Possible*	*Possible*	*Possible*
Secrecy Violation	*Possible*	*Possible*	*Possible*	*Possible*	*Possible*	*Possible*	*Possible*	*Possible*
Integrity Violation	*Possible*	*Possible*	*Possible*	*Possible*	*Possible*	NA	*Possible*	*Possible*

* As discussed in Section V, the marked result depends on the specific triggers/actions/devices we analyzed, not the full possible set of all supported devices.

TABLE II. Feature Support of Smart Home Programming Interfaces

Type of Feature	**Amazon Alexa**	**automate.io**	**Google Home**	**IFTTT**	**Microsoft Flow**	**SmartRules**	**Stringify**	**Zapier**
Event Triggers	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*
State Triggers	*N*	*Y*	*N*	*N*	*N*	*Y*	*Y*	*Y*
Event Actions	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*
Trigger Conjunctions	*N*	*N*	*Y*	*N*	*N*	*Y*	*Y*	*N*
Action Conjunctions	*Y*	*Y*	*Y*	*N*	*Y*	*Y*	*Y*	*Y*
Conditions on Triggers	*N*	*Y*	*N*	*N*	*Y*	*N*	*N*	*Y*
Do While	*N*	*N*	*N*	*N*	*Y*	*N*	*N*	*N*
Integration with Smart Home Devices	*Y*	*N*	*Y*	*Y*	*N*	*Y*	*Y*	*N*
Other Integrations	*N*	*Y*	*Y*	*Y*	*Y*	*N*	*Y*	*Y*

the platform. For the result marked with an asterisk in Table I (Amazon Alexa), our conclusion is based on the triggers/actions we evaluated, not all possible triggers/actions/devices. (In other NA cases without an asterisk, the sets of possible triggers or actions was fixed and small enough to fully enumerate.)

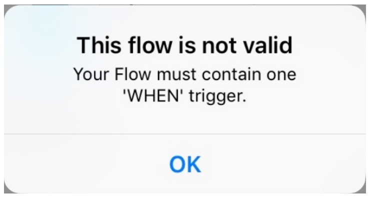
VI. Results and Analysis

In this section, we first discuss our analysis of what kinds of errors programming interfaces are susceptible to. Based on this initial analysis, we group error types into categories that define how a specific error class might be detected and prevented. Finally, we propose points in the design space to address the errors at the end-user programming interface level.

A. *Programming Interface Analysis*

Table I contains a summary of our evaluation results, and we discuss our observations in more detail here.

No interface prevents all errors. At a high level, we observe that *no* end-user programming interface from our set prevents all errors, establishing a clear need for design-level improvements. Furthermore, most interfaces either do not take special measures to prevent the error classes, or prevent them trivially due to feature limitations. For example, IFTTT does not support trigger conjunctions, thus trivially preventing state+state or event+event errors. We note that while feature limitations can help reduce errors (though may not be intentionally used for that purpose by platform designers), they also, of course, limit the potential utility of the platform and are thus not necessarily a desirable way to reduce end user errors.


This flow is not valid

Your Flow must contain one 'WHEN' trigger.

OK

Fig. 1. Stringify disallows state+state and event+event rules.

The only errors directly prevented, by Stringify and SmartRules, are state+state and event+event errors. Stringify directly detects when the user attempts to create a rule involving a state+state or event+event conjunction, and generates an error message when the user tries to save the rule (Figure 1). SmartRules prevents users from creating state+state and event+event rules by construction: it only allows rules of the form "IF *any* of these event triggers are true" **or** "WHILE *all* of these state triggers are true". This construction prevents a user from creating rules such as *"If the door unlocks (event) and the light turns on (event)* or *if the door is unlocked (state) and the light is on (state)*.

The fact that *only* these classes of errors are prevented across all the interfaces we studied suggests that there is opportunity for interfaces to prevent and reduce the likelihood of other classes of end user programming errors, as we discuss further below.

TABLE III. Categorization of Errors Based on what Interfaces can do to detect and prevent them

Category	Error Types
Automatically detectable **and** automatically preventable	Event+Event, State+State
Automatically detectable, **not** automatically preventable	Lack of Action Reversal, Feature Interaction, Feature Chaining, Secrecy Violation, Integrity Violation
Neither automatically detectable **nor** automatically preventable	Missing Triggers, Missing Actions

Most interfaces have incomplete feature sets. Table II summarizes a feature-based evaluation of trigger-action programming interfaces. We note that no current interface has a complete feature set. For example, IFTTT does not allow trigger or action conjunctions.

B. Addressing Errors at Programming Time

One of the goals of our work is to introduce design principles for programming interfaces that will help detect and prevent the above errors. Building upon our analysis, we categorize errors based on what interfaces can do to detect and prevent them, shown in Table III.

Some errors can be both automatically detected and prevented. As we have seen, some errors—like event+event and state+state rules—can be both automatically detected and prevented. Our analysis has shown that Stringify and SmartRules do detect and prevent these errors (Table I).

Some errors can be automatically detected, but preventing them depends on user intent. For example, Surbatovich *et al.* propose an information flow control approach to detecting secrecy and integrity errors [6]. However, while one can logically detect a *possible* secrecy violation, one cannot determine automatically whether the information flow—e.g., sharing a photo to Facebook—was in fact desired by the user.

Some errors require user intent to be detected *and* prevented (neither auto-detectable nor auto-preventable). Finally, some errors depend on user intent even for detection. For example, a user might want to program a rule to ensure that all doors are locked at night and if not, to send a notification to lock that door. In programming the rule, the user may miss including a particular door as one of the intended triggers. Based on the logical rule alone, the programming interface cannot tell that something is awry; below, we propose possible approaches to bridge this gap.

C. Design Recommendations

We make recommendations for trigger-action programming interfaces to handle these error categories.

1) Automatically detect and prevent errors when possible: For the errors that can be automatically detected and prevented, we recommend that future trigger-action platforms implement functionality that prevents such kinds of errors directly in the end-user interface. For example, to prevent state+state errors, an interface might maintain a list of the state triggers, and when a user tries to program a state+state rule, the interface might instantly throw an error. Alternatively, preventing such errors by construction (as in SmartRules) directs users to write correct rules without throwing exceptions.

2) Detect possible errors: For errors that cannot be automatically detected, or where the user's intent is required to verify that a possible error is an actual error, we propose approaches for platforms to detect and verify *possible* errors. This is the first step towards designing interfaces that eventually prevent such errors. These directions present significant opportunities for trigger-action programming platforms to improve—recall that our analysis shows that interfaces currently do nothing to detect or prevent such ambiguous errors.

Large-scale pattern-based predictions. A trigger-action programming interface could use large datasets of user-created rules to predict when a user programs a possibly undesired rule. For example, if many users program rules that turn on lights at a specific time, and then turn off lights at another specific time, it is likely that this is a common two-rule pattern. When a user programs only half of the pattern, e.g., forgetting to turn off the lights, the platform could detect this divergence from common rule patterns and prompt the user to suggest the other half of the rule. We believe that this principle of using data on rule patterns to predict errors is applicable to other types of errors as well, such as missing triggers and missing actions.

Detecting feature chaining via the physical world. In principle, it is possible to detect feature interaction errors, but it is not possible to automatically prevent them because it requires understanding user intent; perhaps the interaction is intended in some cases. To detect such errors, a simple strategy is to create a list of pairs (X, Y), where X is an action and Y is a trigger, and the actuation of X can eventually cause Y to trigger. However, a challenge arises when the pair is related through the physical world. For example, consider the pair (Thermostat, Temperature). Although there is no direct digital connection between the two, there is a physical connection: actuating a thermostat will eventually cause the temperature in a room to rise, which may trigger other rules. Detecting such interactions will likely require modeling physical phenomena.

3) Prompt users for their intent when possible errors are detected: Once a possible error is detected, either automatically or using the above-suggested heuristics, the next question is what should be done in response. We propose directions for programming interfaces to alert users of the possible errors and allow them to react in a way that matches their intention.

Prompt users for next steps. The simplest design is to prompt a user whenever the interface detects or predicts a possible error. This prompt could include helpful information on the type of error, why the interface thinks it is an error, and an option to go back and correct the rule. A challenge here is that end users are susceptible to prompt fatigue, and thus, interface designers should use them judiciously.

Suggest alternate or additional rules. Beyond prompting users to alert them of a possible error, a programming interface could make concrete suggestions for additional rules, leveraging the large-scale pattern-based approach proposed above. E.g., if a user were to create a rule *"If someone is in the room, turn the lights on"*, an interface might automatically suggest an additional rule (to correct lack of action reversal) *"If no one is in the room, turn lights off"*. Or it might suggest an alternate rule (to correct a missing trigger): *"If someone's in the room and it's evening, turn the lights on"*.

Testing by demonstration. Interfaces could also surface errors to end users by allowing them to virtually test their rules. For example, if a user were to program a rule to turn the camera on when there is no one is the house and it is night, the interface could allow the user to test what would occur if the user was at home at night and then left home. Seeing such demonstrations, the user may be able to detect incorrect functionality (e.g., lack of action reversal) in advance of encountering it in real life. A challenge to this approach is how to design the demonstration interface to best help the user explore the possible space of real-world states that may affect the rule in unexpected ways.

Interactive unit testing. Finally, we propose that programming interfaces could enable users to interactively unit test their rules. For example, if the user were to program a rule to lock the doors after 8pm, the interface could ask the user questions such as *If it is 9pm, should the door be locked or unlocked?* [3]. By doing so, the interface could test whether the expectations of the user match the reality of the programmed rule, thereby detecting possible inconsistencies. A challenge in implementing this approach is automatically determining which "unit test" questions to ask the user to keep the number of questions manageable while exploring a useful set of possible questions (i.e., which questions are most likely to surface errors).

VII. Additional Discussion

Mental model inaccuracies. For more complex control structures like Do...While, we observe that current trigger-action programming interfaces do not instill accurate mental models. For instance, a user with programming knowledge might expect that the Do...While construct is a loop that executes at a specific frequency. However, the platforms we studied that support this feature do not communicate how often the loop executes. Non-technical users might not have any mental models about how such structures are supposed to work, and might end up creating non-sensical rules. A challenge is determining how to communicate the right mental models to users about less intuitive control structures.

Platform heterogeneity. It is common for a single home to include rules that are programmed in multiple different trigger-action platforms. For example, a Wemo switch can be programmed using an interface that is included in its control app, but it can also be actuated by voice assistants like Google Home. Detecting and preventing error classes like feature chaining and interactions becomes complex in such settings because no single platform has a global view of all the rules that exist in the home. An open challenge is to tackle platform heterogeneity at the programming interface level, where a user might program the home using a single interface that automatically translates rules into formats that are understandable by specific platforms.

VIII. Conclusion

We empirically studied whether eight commercially available trigger-action platforms prevent or permit nine classes of end-user errors, systematized from prior work. We found that most platforms either do not prevent users errors or prevent them incidentally due to feature limitations. Based on this analysis, we categorized errors on what programming interfaces can do to detect and prevent them and then extracted insights that laid the foundation for the design of future trigger-action programming platforms (for example, using of datasets of functionally-similar rules to predict user errors). We believe this work is the first step towards trigger-action interface designs that significantly mitigate user error.

Acknowledgements

We thank Maya Cakmak for helpful early discussions. This work was supported in part by the National Science Foundation under Award CNS-1513584, the MacArthur Foundation, and the University of Washington Tech Policy Lab.

References

[1] W. Brackenbury, A. Deora, J. Ritchey, J. Vallee, W. He, G. Wang, M. L. Littman, and B. Ur. How users interpret bugs in trigger-action programming. In *Conference on Human Factors in Computing Systems (CHI)*, 2019.

[2] J. Huang and M. Cakmak. Supporting mental model accuracy in trigger-action programming. In *ACM International Joint Conf. on Pervasive and Ubiquitous Computing (Ubicomp)*, 2015.

[3] A. J. Ko and B. A. Myers. Designing the Whyline: A debugging interface for asking questions about program behavior. In *Conf. on Human Factors in Computing Systems (CHI)*, 2004.

[4] C.-J. M. Liang, L. Bu, Z. Li, J. Zhang, S. Han, B. F. Karlsson, D. Zhang, and F. Zhao. Systematically debugging iot control system correctness for building automation. In *3rd ACM International Conference on Systems for Energy-Efficient Built Environments (BuildSys)*, 2016.

[5] C. Nandi and M. D. Ernst. Automatic trigger generation for rule-based smart homes. In *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, 2016.

[6] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of ifttt recipes. In *26th International Conference on World Wide Web (WWW)*, 2017.

[7] B. Ur, E. McManus, M. Pak Yong Ho, and M. Littman. Practical trigger-action programming in the smart home. In *Conf. on Human Factors in Computing Systems (CHI)*, 2014.

[8] S. Yarosh and P. Zave. Locked or not?: Mental models of iot feature interaction. In *Conference on Human Factors in Computing Systems (CHI)*, 2017.