

# VETIoT: On Vetting IoT Defenses Enforcing Policies at Runtime

Akib Jawad Nafis\*, Omar Chowdhury†, and Endadul Hoque\*

\*Syracuse University, NY USA

†Stony Brook University, NY USA

**Abstract**—Smart homes are powered by numerous programmable IoT platforms. Despite tremendous innovations, these platforms often suffer from safety and security issues. One class of defense solutions dynamically enforces safety and security policies, which essentially capture the expected behavior of the IoT system. While many proposed works were built on this runtime approach, they all are under-vetted. The primary reason lies in their evaluation approach. They are mostly self-evaluated in isolation using a virtual testbed combined with manually orchestrated test scenarios that rely on user interactions with the platform’s UI. Such hand-crafted and non-uniform evaluation setups are limiting not only the reproducibility but also a comparative analysis of their efficacy results. Closing this gap in the traditional way requires a huge upfront manual effort, which causes the researchers turn away from any large-scale comparative empirical evaluation. Therefore, in this paper, we propose a highly-automated uniform evaluation platform, dubbed VetIoT, to vet the defense solutions that hinge on runtime policy enforcement. Given a defense solution, VetIoT easily instantiates a virtual testbed inside which the solution is empirically evaluated. VetIoT replaces manual UI-based interactions with an automated event simulator and manual inspection of test outcomes with an automated comparator. We developed a fully-functional prototype of VetIoT and applied it on three runtime policy enforcement solutions: Expat, Patriot, and IoTGuard. VetIoT reproduced their individual prior results and assessed their efficacy results via stress testing and differential testing. We believe VetIoT can foster future research/evaluation.

**Keywords**—IoT Security, Testbed, Evaluation

## I. INTRODUCTION

Smart homes are powered by numerous programmable IoT platforms (e.g., SmartThings, OpenHAB, IFTTT) to facilitate automation and over-the-air control and monitoring. Despite tremendous innovations, smart home systems suffer from safety and security issues [1], [2]. As a result, this problem has garnered much attention from the security research community, which has led to several proposals for defense solutions [3]–[11] focused on curbing unexpected (i.e., unsafe/insecure) behavior on these platforms. One category of these defenses [6]–[8], [10], [11] hinges on the enforcement of safety and security policies at runtime, where each policy essentially captures the expected behavior of the IoT system regarding one or more installed IoT devices. Unfortunately, they are all under-vetted because they were evaluated against a small number of testcases. In addition, identifying a better defense solution has been difficult as they all lack comparative empirical evaluation.

The reason behind using a small number of testcases lies in their current evaluation methodology. First, a testbed (predominantly, virtual) along with the desired automation apps and policies is manually set up to evaluate the defense solution

in isolation. Secondly, an evaluator comes up with testcases, which have to be manually orchestrated. A testcase defines a sequence of events that can occur naturally (e.g., a certain time of the day) or due to an interaction with a IoT device (e.g., opening the front door). The evaluator feeds a testcase into the testbed by manually interacting with the platform’s UI (e.g., a web-based UI or a phone app). The testcases that can demonstrate the defense’s efficacy are primarily selected for evaluation. Thirdly, after feeding a testcase, the log information from the platform is collected and manually inspected for reporting the achieved efficacy. Finally, for the next testcase, a manual cleanup of the testbed is required to bring it back to the initial (clean) state. This huge upfront manual effort causes the researchers to resort to only hand-crafted testcases for evaluation, leaving the defense solutions mostly under-vetted.

Such non-uniform evaluation setups explain the absence of comparative empirical assessments of the effectiveness of these defense solutions. For comparing a new defense (say,  $\mathcal{A}$ ) with an existing one ( $\mathcal{B}$ ), an evaluator must be able to test both  $\mathcal{A}$  and  $\mathcal{B}$  on the  $\mathcal{B}$ ’s original test-suite and a new test-suite. The former calls for reproducing  $\mathcal{B}$ ’s empirical results, and the latter calls for a uniform evaluation mechanism irrespective of the differences between  $\mathcal{A}$  and  $\mathcal{B}$ . Unfortunately, the traditional way of evaluating these defense solutions limit both reproducibility and comparative analysis.

As an example, consider a smart home testbed where the initial states of the devices are: {IndoorMotionSensor = OFF, FrontDoor = CLOSED, HomeMode = ON, SleepMode = OFF}. In addition, an automation app “R1: When IndoorMotionSensor senses motion, execute FrontDoor.Open()” and a policy “P1: Deny opening FrontDoor, if user is not at Home” are installed in the testbed. Suppose we have a testcase (aka, a sequence of events): (HomeMode = OFF, IndoorMotionSensor = ON). The first event simply changes the state of HomeMode to OFF, and the second event triggers R1, which attempts to open FrontDoor. While defenses like ExPAT [7] and PatrIoT [8] block FrontDoor.Open() as opening FrontDoor when the user is away will violate P1, the other solutions such as IoTGuard [6] and IoTSafe [10] fail to block FrontDoor.Open() even though the user is away. Without testing against a large number of testcases and evaluating for a comparative analysis, the shortcomings of a defense solution will remain undetected.

Prior efforts on some standardization in IoT evaluations proposed testbeds with physical devices [12], [13] or emulated devices [14], and benchmarks consisting of IoT applications [15], [16]. However, none of these efforts automate the evaluation of policy enforcing IoT defense solutions.

In this paper, we present VETIOT, a highly-automated uniform evaluation platform for vetting smart home defenses that hinge on runtime policy enforcement. VETIOT is designed to address the aforementioned limitations by automating much of the experimenting process. First, VETIOT incorporates a testbed generator to quickly and easily create an identical testbed to enable a controlled vetting environment required for reproducibility and comparative testing. Secondly, VETIOT employs an event sequence generator to automate the generation of testcases. Instead of manually feeding each event using the platform’s UI, VETIOT utilizes the external programming interface available on the platform (*e.g.*, REST API) to push each event of the testcase. Finally, VETIOT replaces manual inspection of the test outcomes with an automated comparator to report on the defense solution’s efficacy. The comparator makes decisions based on a heuristic algorithm that checks for discrepancy between the initial and final states of the testbed.

We developed a fully functional prototype of VETIOT using Python 3.9. VETIOT leverages virtual IoT devices as they cost nothing and require less time to reset and boot up as opposed to physical devices. To deploy the testbed, VETIOT utilizes the OpenHAB IoT platform [17], because it is not only open source but also entirely deployable on a local machine, as opposed to SmartThings [18] which is hosted on a proprietary cloud server. When VETIOT interacts with the testbed (*e.g.*, to push a testcase), it uses OpenHAB’s REST API interface.

We vetted three smart home defense solutions that enforce safety and security policies at runtime: *PatrIoT* [8], *ExPAT* [7], and *IoTGuard* [6]. We selected them for vetting because of their popularity in the research community. *ExPAT*<sup>1</sup> and *PatrIoT*<sup>2</sup> have open-source implementations for OpenHAB available on GitHub.com. *IoTGuard*, on the other hand, is designed for SmartThings, but its public repository on GitHub.com<sup>3</sup> does not contain the full implementation, which prompted us to develop an in-house implementation of *IoTGuard* for OpenHAB. Our implementation of *IoTGuard* faithfully followed its description [6] as closely as possible.

Using VETIOT, we evaluated each test-subject (*ExPAT*, *PatrIoT*, *IoTGuard*) by following three testing approaches: *fidelity*, *stress*, and *differential*. Technically, fidelity testings are used to assess VETIOT’s fidelity in reproducing the efficacy results of each subject by replicating the same testbed and testcases as mentioned in [6]–[8]. Next, we used VETIOT to stress test each subject separately to measure their own efficacy with respect to multiple test-suites comprising total 140 randomly generated testcases. Finally, VETIOT was used to perform a differential testing on *ExPAT*, *PatrIoT* and *IoTGuard* where we used an identical testbed along with the same auto-generated testcases. Our vetting revealed many nuanced corner cases where the test-subjects efficacy differed for unique reasons specific to their design choices (*e.g.*, which policies are selected to enforce at runtime). In differential testing, we observed that *IoTGuard* prevented 8 less unexpected actions than *ExPAT* and *PatrIoT*. VETIOT is available as open-source at <https://github.com/syne-lab/vetiot>.

**Contributions.** This paper makes the following contributions:

- We proposed a highly-automated uniform evaluation platform, dubbed VETIOT, for vetting IoT defenses that enforce policies at runtime. It automates much of the traditional evaluation process, which requires huge manual efforts from the researchers.
- We developed a fully functional prototype of VETIOT for the OpenHAB platform. To demonstrate VETIOT’s efficacy, we evaluated VETIOT by applying it to three IoT defenses (*ExPAT*, *PatrIoT*, and *IoTGuard*) for assessing their individual and comparative efficacy.
- To the best of our knowledge, VETIOT is the first automated platform that empirically evaluates IoT security defenses enforcing policies at runtime.

## II. PRELIMINARIES

**Platforms.** Smart home platforms (*e.g.*, OpenHAB, SmartThings) provide users with a common interface to control and manage IoT devices and enable users to automate numerous manual tasks with customized applications (aka, *apps*) that may interact with the physical world by operating the IoT devices. A platform is considered as the *brain* of a smart home, because the platform not only manages the devices connected to it and ensures communication between them but also executes the core automation logic. Depending on the platform’s architecture, a local or remote server (aka, *backend*) is employed to host the brain. For instance, SmartThings utilizes a proprietary cloud-based backend whereas OpenHAB allows users to spawn their own local backend.

**Apps.** Platforms typically allow numerous customization in automation apps, which can vary in complexity. For example, “When the sun sets, turn on the porch lights” is a simpler app and “When *Smoke\_Detector* detects smoke and if *Living\_room\_temperature* > 120°, turn on *Fire\_Sprinkler*” is a complex one. In general, these apps follow a *trigger-condition-action* paradigm. A trigger, which is usually a logical event occurred in the smart home (*e.g.*, “*Smoke\_Detector* detects smoke”), initiates the execution of an app. The action block includes a list of commands to be operated on the respective IoT device (*e.g.*, “turn on *Fire\_Sprinkler*”). Note that a command can change not only the state of the physical world (*e.g.*, turning on the fire sprinkler will spray water) but also the internal state of the device (*e.g.*, *Fire\_Sprinkler.state* = ON). A condition is an optional block of predicates (*e.g.*, *Living\_room\_temperature* > 120°), often expressing a situation that must be satisfied before the app can continue executing its action block.

**Defenses using Policy Enforcement.** Many safety and security concerns of smart home (or, IoT systems in general) often stem from faulty apps or unintended interaction and interference between apparently correctly functioning apps. It is a common threat model for smart homes. To curb the undesired behavior of apps, many solutions have been proposed, including a class of defenses that rely on enforcing policies at runtime to ensure safety and security of IoT systems [6]–[8], [10], [11]. A policy is basically a user’s expectation about the behavior of an IoT system. An example policy can be “The surveillance

<sup>1</sup><https://github.com/expat-paper/expat>

<sup>2</sup><https://github.com/yahyazadeh/patriot>

<sup>3</sup><https://github.com/BeerKay/SmartAppAnalysis>

camera can never be turned off.” For example, if an app issues an action to turn off the surveillance camera (*i.e.*, `camera.off()` command), a defense solution should block it as the action violates the camera policy. Thus, the defense attempts to keep the system aligned with the user’s expectations at runtime.

Furthermore, for policy enforcement, a defense solution needs to deploy hooks (called policy information point (**PIP**)) at several locations in each app to collect additional information at runtime, necessary to decide about policies using a policy decision function (**PDF**). The defense also uses a policy enforcement point (**PEP**) where the defense checks if the contemplated actions will satisfy or violate the user-provided policies.

Where to deploy their PEP, PIP and PDF varies among the defense mechanisms. While some solutions [7], [8] deploy all its PEP, PIP and PDF in the app’s source code, solutions like [6], [10] keep PIP and PEP in the app but leverage a remote offshore server to deploy their PDF. Therefore, they all require an access to the apps’ source code for instrumentation.

Irrespective of how PEP, PIP, and PDF are deployed, whenever the execution of an app reaches its PEP (*e.g.*, its action block), the instrumented code in PEP invokes its PDF by supplying all the information collected from its PIP. Once the PDF returns with a response, the PEP simply allows the actions if the response is positive or blocks the actions if the response is negative.

### III. OVERVIEW OF VETIoT

In this section, we present an abstract model of our testing platform and our problem definition.

#### A. Abstract Models

**Programmable IoT Systems.** At a high-level, a programmable IoT system  $\mathbb{I}$  can be viewed a labeled transition system (LTS) defined as  $\mathbb{I} = \langle \mathcal{V}, \mathcal{S}, \mathcal{A}, \Lambda, \mathcal{R}, \mathcal{T} \rangle$ . Here,  $\mathcal{V}$  represents a finite set of *typed* variables, which can further be decomposed into two mutually exclusive sets  $\mathcal{V}_{env}$  and  $\mathcal{V}_{dev}$ .  $\mathcal{V}_{env}$  refers to a set of environment variables (*e.g.*, temperatures) and  $\mathcal{V}_{dev}$  denotes a set consisting of the internal state variables of each device deployed in this system (*e.g.*, the state of a bedroom light).  $\mathcal{S}$  is a non-empty finite set of system-states such that each  $s \in \mathcal{S}$  is a tuple  $\langle d_0, d_1, \dots, d_{|\mathcal{V}|} \rangle$  where  $d_i$  is an assigned value to  $v_i \in \mathcal{V}$  taken from  $v_i$ ’s finite domain  $\mathcal{D}_i$ . For example, a state  $s$  of an IoT system composed of two devices – a front-porch light and a smart lock – can be  $\langle \text{On}, \text{Locked} \rangle$  at a given instant of time.  $\mathcal{A}$  in  $\mathbb{I}$  refers to a finite set of all possible activities supported by the underlying IoT platform. Technically, an IoT platform provides a list of `action_commands` to change each  $v \in \mathcal{V}$ . For example,  $\{\text{Lock}, \text{Unlock}\}$  are two possible actions allowed to change the status of a smart lock.

The rest of the components of  $\mathbb{I}$  is relatively complex. Whenever there is a change in the value of  $v_i$  (*e.g.*, from  $d_{i_j}$  to  $d_{i_k}$ , where  $d_{i_j}, d_{i_k} \in \mathcal{D}_i$ ), the device/sensor sends  $\lambda_i$  (aka, a `status_update` message) to the backend. In fact, there is a one-to-one mapping between  $v_i$  and  $\lambda_i$ , and hence  $\Lambda = \bigcup_{i=0}^n \lambda_i$ . The change in  $v_i$  can happen *after* (i) the device

executes an appropriate command  $a_x \in \mathcal{A}$  received from the backend (*e.g.*, the `Unlock` command), or (ii) a user physically interacts with the device (*e.g.*, the user unlocks the door with a key).

$\mathcal{R}$  defines a set of all possible automation apps allowed by the underlying platform and can be viewed as  $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{C} \times 2^{\mathcal{A}}$ . Here,  $\mathcal{E}$  is a finite set of all possible methods to interact with  $\mathbb{I}$ . These interactions can be either through physical interactions with the actual device(s) or using the platform’s UI. The former method causes the device to send  $\lambda_i$  to the backend for the change occurred in  $v_i$ , while the latter method asks the backend to send  $a_x$  to the device which may change  $v_i$ . Hence,  $\mathcal{E}$  is essentially the possible set of **triggering events**, defined as  $\mathcal{E} = \mathcal{A} \cup \Lambda$  (Recall that  $\mathcal{A}$  and  $\Lambda$  are mutually exclusive. Some platforms, *e.g.*, OpenHAB, allow apps that can be triggered right before executing an action from  $\mathcal{A}$ ).  $\mathcal{C}$  is a finite set of conditions/predicates such that each element  $c \in \mathcal{C}$  is a Boolean expression with logical and relational operators over  $\mathcal{V}$ . In other words, a condition dictates a specific situation under which the app in question can be executed. For any  $e \in \mathcal{E}$ ,  $c \in \mathcal{C}$  and  $\alpha \in 2^{\mathcal{A}}$ , if an app  $\langle e, c, \alpha \rangle \in \mathcal{R}$ , then it signifies that after observing a triggering event  $e$  under a situation where  $c$  evaluates to true, the backend will execute the actions listed in  $\alpha$ .

$\mathcal{T}$  is another relation and defined as  $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{E} \times \mathcal{S}$ . This signifies how the system  $\mathbb{I}$  transitions from one state to another upon receiving a triggering event from a device. We consider  $\mathcal{T}$  to be left-total and  $\mathbb{I}$  to be a deterministic LTS.

**Target Defense Solutions (TDS).** In this paper, our target defense solutions are those that aspire to ensure the safety and security of an IoT system by enforcing policies at runtime [6]–[8]. A target defense system (TDS)  $\mathbb{T}$  is defined as  $\mathbb{T} = \langle \Psi, \Delta, \aleph \rangle$ . Here,  $\Psi$  is a finite set of policy statements (say,  $\psi_1, \dots, \psi_n$ ),  $\Delta$  dictates the logic behind its policy enforcement (*i.e.*, PIP, PEP, and PDF), and  $\aleph$  is a special function to embed PEP, PIP and PDF in each app before installing it to  $\mathbb{I}$ .

Assume for each app  $r \in \mathcal{R}$ ,  $\xi(r)$  denotes the syntactical representation of  $r$  written in the language supported by the platform and  $\mathcal{L}(r)$  denotes the semantics of  $r$  defined by the language’s type system and the supported logical formula. Now we can defined  $\aleph$ , which is in theory,  $\aleph : \mathcal{R} \mapsto \mathcal{R}$ . In other words, for each  $r \in \mathcal{R}$ , there exists  $r' \in \mathcal{R}$  such that both  $r$  and  $r'$  are semantically the same (*i.e.*,  $\mathcal{L}(r) \equiv \mathcal{L}(r')$ ) but syntactically different (*i.e.*,  $\xi(r) \neq \xi(r')$ ). The syntactical difference between  $r$  and  $r'$  is merely due to incorporating PIP, PEP, and PDF inside  $r'$ .

The target system is often equipped with its own domain specific language that the user utilizes to write each policy. Internally, each policy  $\psi_j$  is first converted to a logical formula (preferably, quantifier-free first-order logic (QF-FOL) or similar). In most cases, the collection of policies  $\Psi$  is treated as a logical conjunctive formula over all  $\psi_j$ s (*i.e.*,  $\Psi \equiv \bigwedge_{j=0}^n \psi_j$ ).

The policy decision logic  $\Delta$  expresses the underlying mechanism to enforce  $\Psi$  at runtime. This mechanism varies with each TDS. For example, given  $\Psi$ ,  $\Delta_{\text{ExpAT}}$  used by ExpAT [7] influences  $\mathcal{T}$  to include  $\langle s_i, \lambda, s_j \rangle$  only if  $s_j \models \Psi$ . Informally,  $\Delta_{\text{ExpAT}}$  will allow a contemplated action  $a \in \mathcal{A}$

in  $s_i$  only if the status update  $\lambda$  that will be resulted after executing  $a$  will take the system to  $s_j$ , which satisfies  $\Psi$  (i.e.,  $s_2 \models \Psi$ , aka, safe).  $\Delta_{\text{IoTGuard}}$  used by IoTGuard [6] logically achieves the same outcome as  $\Delta_{\text{ExPAT}}$ . On the contrary, given  $\Psi$ ,  $\Delta_{\text{PatrIoT}}$  of PatrIoT [8] allows  $a \in \mathcal{A}$  in  $s_i$  only if  $s_i \models \Psi$  and thus includes  $\langle s_i, a, s_j \rangle \in \mathcal{T}$ . Informally,  $\Delta_{\text{PatrIoT}}$  allows  $a$  only if the current state  $s_i$  satisfies all policies relevant to  $a$ .

**Testbeds:** A testbed is defined as  $\Omega = \langle \mathbb{I}, \mathbb{T} \rangle$ , where  $\mathbb{I}$  and  $\mathbb{T}$  are defined as above. For example, when VETIoT instantiates a testbed for ExPAT, the test-bed is notified as  $\Omega_{\text{ExPAT}} = \langle \mathbb{I}_{\text{ExPAT}}, \mathbb{T}_{\text{ExPAT}} \rangle$ . Similarly, for PatrIoT and IoTGuard, VETIoT can instantiate  $\Omega_{\text{PatrIoT}} = \langle \mathbb{I}_{\text{PatrIoT}}, \mathbb{T}_{\text{PatrIoT}} \rangle$  and  $\Omega_{\text{IoTGuard}} = \langle \mathbb{I}_{\text{IoTGuard}}, \mathbb{T}_{\text{IoTGuard}} \rangle$ , respectively.

Note it is possible that the testbeds can have the same  $\mathbb{I}$  (i.e.,  $\mathbb{I}_{\text{ExPAT}} = \mathbb{I}_{\text{PatrIoT}} = \mathbb{I}_{\text{IoTGuard}}$ ), but their  $\mathbb{T}$ s must be different. Finally, when no target defense solution is selected, the testbed  $\Omega_v = \langle \mathbb{I}, \emptyset \rangle$  is considered as a **vanilla** testbed, where no instrumentation on apps and no policy enforcement at runtime are performed.

**Testcases:** Given a testbed  $\Omega$  (an instantiation for a  $\mathbb{I}$  and  $\mathbb{T}$ ), a testcase (aka, test scenario) is defined as  $\Gamma = \langle \Omega, \zeta \rangle$ , where  $\zeta$  is a sequence of triggering events (recall,  $\mathcal{E}$ ).

### B. Problem Definition

Consider a testcase  $\Gamma = \langle \Omega_x, \zeta \rangle$ , where  $\Omega_x$  be a testbed instantiated for a  $\mathbb{I}_x$  and an actual  $\mathbb{T}_x$  and  $\zeta$  be  $\langle e_1, e_2, \dots, e_n \rangle$  with  $n > 0$ . Assume  $\mathbb{I}$  begins at  $s_0$ . After pushing each  $e_j$  (where,  $1 \leq j \leq n$ ) to  $\Omega_x$ ,  $\mathbb{I}_x$  transitions through states  $s_j$  and eventually ends up in  $s_n$ . If any  $e_j \in \zeta$  triggers the app that in turn invokes some unexpected actions, VETIoT can detect if  $\mathbb{T}_x$  were successful in preventing the unexpected actions and mark  $\Gamma$  accordingly (i.e., success or failure). success signifies that  $\mathbb{T}_x$  prevented the unexpected actions whereas failure denotes  $\mathbb{T}_x$  failed to do so.

**Problem.** Given a TDS (say,  $P$ ), can VETIoT evaluate its efficacy? We designed VETIoT to evaluate the TDS using three testing approaches:

- **Fidelity testing:** VETIoT checks if it can reproduce the evaluation result reported in the TDS’s paper/repository. Formally, VETIoT instantiates  $\Omega_P$  using the same  $\mathbb{I}_P$  and  $\mathbb{T}_P$  as described in  $P$ ’s paper and tests the TDS against a series of testcases  $\Gamma_1, \Gamma_2, \dots$ , where each  $\Gamma_i = \langle \Omega_P, \zeta_i \rangle$  and  $\zeta_i$  be the same event sequence used in the paper.
- **Stress testing:** VETIoT measures the TDS’s efficacy against new testcases (not tested in the original paper). Formally, VETIoT utilizes the previously instantiated  $\Omega_P$  and tests against a series of testcases  $\Gamma_1, \Gamma_2, \dots$ , where each  $\Gamma_i = \langle \Omega_P, \zeta_i \rangle$  and  $\zeta_i$  be a randomly generated event sequence.
- **Differential testing:** VETIoT assesses how the given TDS ( $P$ ) fares against other TDSs (say,  $Q$ ). Formally, VETIoT first instantiates two almost identical testbeds  $\Omega_P$  and  $\Omega_Q$  where  $\mathbb{I}_P = \mathbb{I}_Q$  and  $\Psi_P = \Psi_Q$  and then tests both  $P$  and  $Q$  in their respective testbed using the same series of randomly generated event sequences. Finally, VETIoT reports the efficacy of  $P$  compared to  $Q$ .

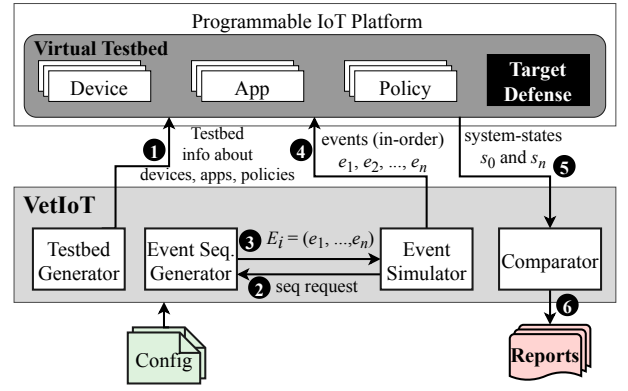


Fig. 1: VETIoT’s architecture and workflow

## IV. DESIGN OF VETIoT

We will describe a high-level workflow of VETIoT followed by the inner workings of its different components..

### A. Workflow

Figure 1 presents the architecture of VETIoT. VETIoT consists of 4 modules: Testbed Generator, Event Sequence Generator, Event Simulator, and Comparator. All four modules of the VETIoT work in concert to create a desired testbed and conduct experiments for evaluating an IoT defense solution.

Given a testbed configuration file provided by the user (aka, the evaluator), the testbed generator instantiates a virtual testbed – consisting of IoT devices – in a programmable IoT platform (e.g., OpenHAB) and then installs the specified automation apps. For the vanilla testbed  $\Omega_v$ , that is all required. However, to prepare a testbed  $\Omega_{\mathbb{T}}$  for a target defense solution ( $\mathbb{T}$ ), the testbed generator deploys the supplied policies in appropriate format and the solution itself. A testbed is instantiated automatically by invoking (external) APIs of the platform (1). Our generator can create not only an arbitrary testbed but also a customized testbed based on the supplied configuration. While the former mode offers versatility, the latter enables a controlled vetting environment essential for reproducibility and comparative analysis.

After that, the event simulator takes over the control and drives the execution of each testcase. Upon the request for the next testcase (2), the event sequence generator supplies a sequence of events ( $E_i = \langle e_1, \dots, e_n \rangle$ ) composing the testcase  $\Gamma_i$  (3).<sup>4</sup> What an event will be depends on the devices installed in the testbed. For example,  $\text{TV} = \text{ON}$  is a possible event for TV, indicating TV has been turned ON; similarly,  $\text{MotionSensor} = \text{ON}$  is possible event for the MotionSensor device, indicating the motion sensor has detected motion. Now whether the supplied sequence of events is randomly generated or selected for a predefined set depends on the provided configuration. The former is used for stress testing each TDS, and the latter is used for differential testing. Then the event simulator pushes each event  $e_j$  sequentially to the platform using the platform’s external API (4).

<sup>4</sup>Hereafter,  $E$  and  $\zeta$  are used interchangeably to denote a sequence of events.

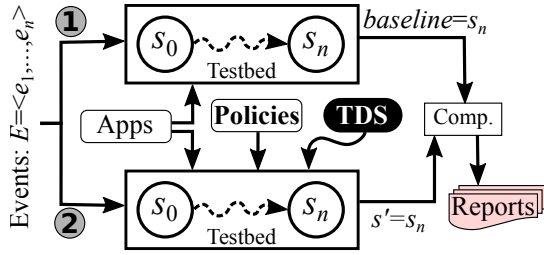


Fig. 2: Testing a target defense solution (TDS) using VETIoT

Upon processing each event  $e_j$ , the testbed logically changes from the system-state  $s_{j-1}$  to  $s_j$ . Note that  $s_{j-1}$  and  $s_j$  are not necessarily unique. After a short delay, the comparator collects the testbed’s initial  $s_0$  and final  $s_n$  system-states (5) and compares them to generate a report (6). This report indicates whether the  $\mathbb{T}$  was successful in preventing unexpected actions in case of an event causing any policy violation. VETIoT additionally provides a debug mode. If it is enabled, a trace of how the system-state is changing after each event is included in the report. Before the event simulator repeats the process for the next testcase from 2, VETIoT automatically resets the testbed to its initial. The testing cycle will terminate once there is no more testcases or the time budget expires.

### B. Automated Comparator

The goal of designing VETIoT is to evaluate dynamic policy enforcing smart home defenses automatically. To perform automated testing, VETIoT easily replaces most of the manual steps of the traditional evaluation: setting up testbed, generating testcases, and running testcases. But automating the manual inspection of test outcome is challenging, because neither the platform nor a TDS itself provides any output signal to indicate the success/failure of the TDS in preventing unexpected actions. One can argue to export the internal result of policy violation from inside the platform or the TDS, which we want to avoid as it will require us to modify either the platform or the TDS, limiting VETIoT’s portability. With these constraints, there exists no unequivocal way to externally measure the efficacy of the TDS.

To overcome this challenge, we adopted a heuristic approach to externally measure the efficacy of a TDS (see Figure 2). Technically, we run each testcase twice: once in the vanilla testbed  $\Omega_v$  (path 1) and once in the TDS’s testbed  $\Omega_{\mathbb{T}}$  (path 2). Recall that both  $\Omega_v$  and  $\Omega_{\mathbb{T}}$  both have the same set of devices and apps, but only  $\Omega_{\mathbb{T}}$  contains the policies and the solution code  $\mathbb{T}$ . The final system-state  $s_n$  of  $\Omega_v$  serves as the *baseline*, which describes what the final system-state would be without the defense solution  $\mathbb{T}$ . The *baseline* will be compared against the final system-state  $s_n$  of  $\Omega_{\mathbb{T}}$  (i.e.,  $s'$ ).

After collecting both *baseline* and  $s'$ , VETIoT uses the comparator (see Algorithm 1) which operates as follows: for each device  $i$ , it checks if the observed state of the device ( $d_i$ ) in *baseline* is different from that in  $s'$ . Any discrepancy indicates that some actions were allowed in the vanilla testbed  $\Omega_v$  but later blocked in the  $\mathbb{T}$ ’s testbed  $\Omega_{\mathbb{T}}$ , indicating the possibility of  $\mathbb{T}$  becoming successful in preventing unexpected actions. The comparator next checks if both  $s_0$  and  $s'$  have

---

### Algorithm 1 Comparator

---

**Require:** Initial system-state  $s_0 = \langle d_1, d_2, \dots, d_m \rangle$ ,  $baseline = \langle d_1, d_2, \dots, d_m \rangle$ ,  $s' = \langle d_1, d_2, \dots, d_m \rangle$ , assuming there exists  $m$  devices

- 1:  $policyViolation \leftarrow false$
- 2:  $indeterminate \leftarrow false$
- 3: **for**  $i \leftarrow 1$  to  $m$  **do**
- 4:   **if**  $baseline[d_i] \neq s'[d_i]$  **then**
- 5:     /\* Defense might work \*/
- 6:     **if**  $s'[d_i] = s_0[d_i]$  **then**
- 7:        $policyViolation \leftarrow true$  /\* Defense worked \*/
- 8:     **else**
- 9:        $indeterminate \leftarrow true$  /\* Unsure \*/
- 10:    **end if**
- 11: **end if**
- 12: **end for**
- 13: **if**  $indeterminate = true$  **then**
- 14:    Report “the outcome is indeterminate”
- 15: **else if**  $policyViolation = true$  **then**
- 16:    Report “the TDS prevented unexpected actions”.
- 17: **else**
- 18:    Nothing to report
- 19: **end if**

---

recorded the same state of the device. The equality here means that the initial state of the device did not change in the final system-state, implying the success  $\mathbb{T}$  in preventing unexpected actions on this device. However, if  $d_i$  of  $s_0$  and  $d_i$  of  $s'$  are unequal, then the comparator cannot be conclusive in deciding about the success of  $\mathbb{T}$ . Therefore, the comparator marks them as “indeterminate”.

To help an evaluator debug such indeterminate cases, VETIoT offers a debug mode, which is slow but generates a detail report to provide fine-grained insights of the testbed for the testcase in question. When the debug mode is enabled, the comparator collects the system-state of the testbed after pushing each event of the sequence, unlike the normal mode where it waits for the entire sequence to finish running. After running VETIoT in the debug mode for the testcase twice: once in  $\Omega_v$  and later in  $\Omega_{\mathbb{T}}$ , the comparator generates a trace file containing the pairwise differences between the system-state of  $\Omega_v$  and that of  $\Omega_{\mathbb{T}}$  at every step. The evaluator can utilize this detail report to manually inspect such cases.

## V. IMPLEMENTATIONS

We implemented a fully functional prototype of VETIoT using Python 3.9. As a the programmable IoT platform, VETIoT utilizes the OpenHAB platform [17] (precisely, OpenHAB-3.2.0 stable runtime edition). During an evaluation of a target defense solution (TDS), different components of VETIoT communicates with the platform using OpenHAB’s REST API. VETIoT accepts testbed configurations in toml format. It uses json format for internal usage and reporting results. The event simulator uses Python’s random module for randomly generating event sequences. VETIoT introduces some delays at several points during an evaluation. For example, the mechanism to install apps in OpenHAB does not

instantaneously load freshly installed apps. Therefore, VETIoT waits for 10 seconds (resp., 15 seconds) when it installs vanilla apps (resp., instrumented apps) in the testbed. To ensure that each injected event is processed by the platform and that the triggered apps have enough time to finish their execution, VETIoT waits for 5 seconds. All the delay times were selected based on our experiments with the platform where we gradually increased the delay times until we found a suitable duration when VETIoT could finish fidelity testing without any incomplete execution of the apps.

## VI. EVALUATION

We now demonstrate how our prototype implementation of VETIoT empirically assessed several IoT defenses that enforce policies at runtime. For our evaluation, we selected three test-subjects: ExPAT [7], PatrIoT [8], and IoTGuard [6]. While they are highly popular in the community and closely related in terms of their policy enforcement mechanism, none of them was empirically compared with each other. Another goal of this evaluation is to show how the developers of those defense solutions could have utilized VETIoT had it existed back then.

The research question we set out to answer is: *can VETIoT automatically evaluate each of three IoT defenses and compare them empirically?* To answer this question, we conducted the evaluation using three testing approaches – fidelity, stress and differential – discussed in § III-B. As our evaluation metrics, we used two counts: “violation” and “indeterminate”. While the former denotes the number of testcases for which VETIoT detects that the defense solution managed to prevent unexpected actions, the latter denotes the number of testcases for which VETIoT is unsure whether the defense worked or not. The remaining testcases are all considered as “no violation”. We also measured the performance overhead incurred by VETIoT.

### A. Setup

In our evaluation, we used the open-source implementations of ExPAT and PatrIoT from their respective public repository (*i.e.*, github.com). Despite IoTGuard’s popularity, unfortunately, a full implementation of IoTGuard is not publicly available. Therefore, we re-implemented IoTGuard by closely following [6]. Although IoTGuard was originally implemented for SmartThings, we chose the OpenHAB platform to implement IoTGuard as both ExPAT and PatrIoT were implemented for OpenHAB. While both ExPAT and PatrIoT has all their policy enforcement modules (*i.e.*, PIP, PEP, PDF) in the platform, IoTGuard employs a remote offshore server to host its PDF and exports data from the platform to the server for policy decision. Therefore, our in-house IoTGuard implementation spawns a separate server to host its PDF and intermediate data necessary for making policy decision (for details, see Appendix A).

While VETIoT can utilize a testbed with physical devices for experiments, we chose virtual devices for our evaluations, because they were used by our test-subjects in their original evaluation. For fidelity and stress testing, we replicated the same virtual smart home and policies as used in their respective paper (recall  $\Omega_{\text{ExPAT}}$ ,  $\Omega_{\text{PatrIoT}}$ ,  $\Omega_{\text{IoTGuard}}$  in § III-A). However,

for differential testing, we used an identical smart home ( $\mathbb{I}$ ) and the same set of policies ( $\Psi$ ) for all test-subjects. These  $\mathbb{I}$  and  $\Psi$  were chosen from ExPAT, the lowest common denominator.

The event sequence ( $\zeta$ ) used in a testcase depends on the testing approach. For fidelity testing, we hand-crafted each event sequence for VETIoT according to the testcases used in the test-subject’s original evaluation. For stress testing, VETIoT randomly generates a sequence of events for each testcase. The set of testcases used in stress testing is unique for each test-subject. On the contrary, for differential testing, all test-subjects were tested against the same testcases. For simplicity, the set of testcases generated for ExPAT’s stress testing is reused in the differential testing of all test-subjects.

We configured VETIoT to create 6 test-suites consisting of 5, 10, 15, 25, 35, and 50 different testcases. Each testcase can have at most 15 events, while the actual number of events was varied between 1 and 15. These event sequences were generated based on the testbed VETIoT was evaluating. All our experiments were conducted in a server machine equipped with 40 core 2.4 GHz Intel Xeon(R) Platinum 8260 CPU and 80GB RAM.

### B. Results

**Fidelity testing.** With this experiment, we wanted to assess the correctness of our VETIoT prototype by checking whether VETIoT could faithfully reproduce the evaluation results of each test-subject reported in [6]–[8]. These works conducted their evaluations manually where the authors interacted with the virtual devices using the platform UI to create a triggering event and inspected the outcome of each testcase. VETIoT replaced all the manual interaction and inspection with automated approaches while keeping the same smart home and policies as the prior work. In this evaluation, we observed that VETIoT tested ExPAT and PatrIoT with all the testcases (8 and 5, respectively) and were able to reproduce all results mentioned in [7], [8]. On the other hand, VETIoT reproduced the results of 4 out of 6 testcases for IoTGuard. Unfortunately, we could not even reproduce the results of the remaining 2 testcases manually using the platform’s UI. After close inspection, we found that those testcases could not be reproduced solely based on the description provided in [6].

**Stress testing.** Recall that in stress testing, we wanted to evaluate the efficacy of a given test-subject using VETIoT. For each subject, we replicated the testbed as mentioned in their paper but tested the subject against randomly generated 6 test-suites. Technically, we randomly generated a sequence of events for each testcase required for our test-suites (mentioned earlier). In other words, each subject was tested in a different testbed against different sets of test-suites.

Figure 3a shows the outcome of stress testing ExPAT against 6 test-suites. For example, we observed that ExPAT was able to prevent unexpected actions in 15 out of 50 testcases of test-suite 6 (testcases with policy violation). 33 testcases did not cause any policy violation. In the remaining 2 testcases, VETIoT noticed some changes in one or more devices but was unsure whether ExPAT was successful or not, and thus VETIoT reported them as indeterminate.

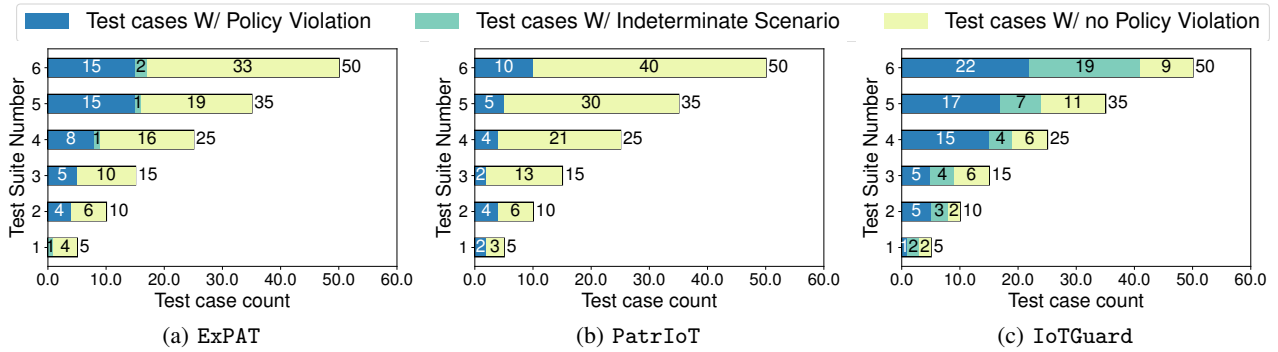


Fig. 3: Result of Stress Testing IoT defenses

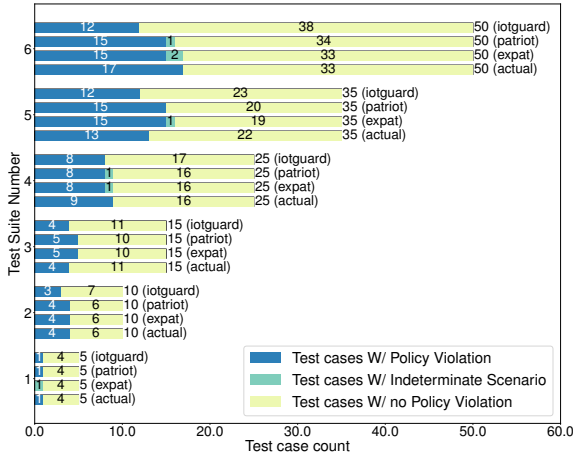


Fig. 4: Result of Differential Testing with VETIoT

Figure 3b shows the results of stress testing PatrIoT against 6 test-suites generated for PatrIoT. We observed that PatrIoT has no indeterminate cases, which can be attributed to the policy language of PatrIoT that allows fine-grained policies. We will explain this benefit with a case study later.

Figure 3c shows the results of stress testing IoTGuard. We observed a higher number of policy violation and indeterminate cases, which is due to the testbed of IoTGuard. The testbed has many apps that create a loop in multiple testcases (e.g., app1’s action triggers app2, then app2’s action triggers app1). In such cases, VETIoT cannot correctly generate a baseline that is required by the comparator algorithm.

**Differential testing.** In this experiment, we wanted to assess VETIoT’s capability to empirically compare the test-subjects. VETIoT used an identical testbed along with the same policies (written in their respective languages) and the same suites of testcases to evaluate each subject. We compared the test outcomes of each subject with that of others. The rationale was to measure how a subject fared compared to others in preventing unexpected actions. Figure 4 shows the comparison of differential testing, where actual denotes the ground truth (derived manually). We observed that ExPAT and PatrIoT behaved similarly in most cases. IoTGuard detected relatively lower number of policy violations, which we attribute to IoTGuard’s policy enforcement mechanism. This reason will be further explained in a case study later. In suites 3 and 5, both ExPAT and PatrIoT erroneously labeled more testcases as policy violations than the actual number. After close in-

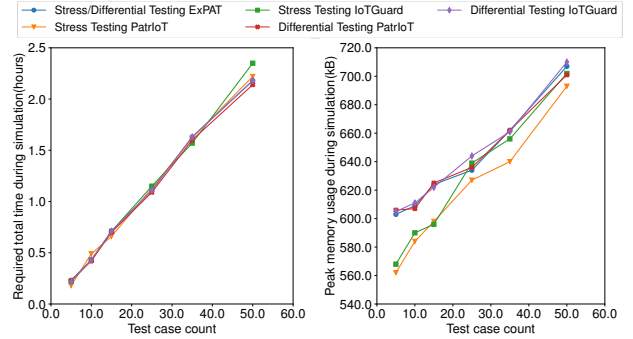


Fig. 5: Performance overhead incurred by VETIoT

spection, we found that testcases causing the IoT system to become unsafe due to events simulating physical interactions were labeled as policy violations by ExPAT and PatrIoT.

**Performance overhead.** To measure approximate performance overhead incurred by VETIoT, we recorded its peak memory usage and the total required time when running each test-suite. Figures 5a and 5b show that VETIoT required around 2.2 hrs maximum (no more than 6 seconds as CPU-time + delays) and less than 800kB recorded its peak memory usage. Also, the performance overhead increases linearly with the increase in the number of testcases.

### C. Case Studies

**Type1: Inconsistency between the IoTGuard’s policy server and the platform.** Differential testing revealed that IoTGuard failed to prevent policy-violating actions in multiple testcases, whereas ExPAT and PatrIoT prevented them. It is because the IoTGuard’s policy server and the platform sometimes go out-of-sync, resulting in erroneous policy enforcement. Note that this issue is specific to IoTGuard’s design.

Consider a testcase from test-suite 2 of differential testing. For this testcase, IoTGuard failed to prevent WaterValve from turning OFF when FireSprinkler was spraying water due to the smoke detected by SmokeDetector, even with this policy “PI1: WaterLeakDetector can turn off WaterValve only if SmokeDetector is OFF” in place. Upon investigation, we observed that when the platform received the event SmokeDetector=ON, no app was triggered and hence no information about this event was exported

to the IoTGuard’s policy server. Recall that IoTGuard exports data to the policy server right before an app’s action block (*i.e.*, PEP). Later when the platform received WaterLeakDetector=ON, an app was triggered that contemplated an action WaterValve=OFF. The PEP of this app exported all information (*i.e.*, the event WaterLeakDetector=ON and the contemplated action WaterValve=OFF). Since the policy server has no information about SmokeDetector=ON, the server erroneously decided that the contemplated action would not violate PI1 and thus allowed the action, causing WaterValve to be OFF at a wrong time.

**Type2: Efficacy depends on the policy selection process.** How a defense solution selects which policies to evaluate during its policy enforcement impacts its efficacy. While PatrIoT and IoTGuard select only those policies that are relevant to the contemplated actions to evaluate, ExPAT always evaluates all policies, which sometimes results in erroneous enforcement.<sup>5</sup>

Consider a part of a long testcase from test-suite 1 of differential testing, where the platform receives the event TV=ON followed by AC=ON and SleepMode=ON in that order. The TV=ON event triggered an app that opened LivingRoomWindow. Then when AC=ON (say, due to a physical action) was received, the system reached an unsafe state because it violated a policy “PI8: LivingRoomWindow can be opened only if both the heater and AC are off.” None of the defense solutions prevented it because it was due to a user interaction in the physical world.

Later when the SleepMode=ON event triggers an app that contemplated to turn off TV, ExPAT denied the action as it evaluated all policies, including PI8 that turned out to be false. On the contrary, PatrIoT and IoTGuard will allow TV.off() because they did not evaluate PI8 as this policy was not related to the contemplated action.

**Type3: Erroneous policy decision function.** IoTGuard’s policy decision function (PDF) employs a reachability analysis on the graph of events and actions that the policy server internally builds based on the data exported by the platform. During stress testing, IoTGuard could not prevent unexpected actions in some testcases due to an erroneous reachability analysis. Instead of explaining one of those testcases, we will use a simple scenario for the sake of brevity.

Consider a testcase where the event TV=ON is followed by the event LivingRoomTemp=90°. The first event triggered an app that opened LivingRoomWindow. Later the second event triggered an app that contemplated AC.on() because LivingRoomTemp>70°. Now as per the policy “PI8: LivingRoomWindow can be opened only if both the heater and AC are off”, a defense mechanism should not turn on AC because LivingRoomWindow=OPEN and PI8 is internally represented as “¬(LivingRoomWindow=OPEN) ∨ ((Heater=OFF) ∧ (AC=OFF))”.

<sup>5</sup>Recall that the collection of policies ( $\Psi$ ) is a conjunction of all policy statements ( $\psi_j$ ) (*i.e.*,  $\Psi \equiv \bigwedge_{j=0}^n \psi_j$ ). Therefore, a violation of any policy (*e.g.*,  $\psi_j = \text{false}$ ) will evaluate  $\Psi$  to false. While the policy selection of PatrIoT and IoTGuard may appear counter intuitive because they do not evaluate the non-relevant policies, the policy language and its semantics of PatrIoT and IoTGuard allow the non-relevant policies automatically evaluate to true, making the evaluation of  $\Psi$  only depends on the relevant ones.

Unfortunately, IoTGuard could not block AC.on() because of its reachability analysis. Since there was no direct interaction from the first app to the second app (*i.e.*, one app’s action does not trigger the second app), IoTGuard’s policy server did not have any path from the LivingRoomWindow=OPEN node to the AC=ON node. As a result, IoTGuard erroneously decided no violation of PI8 and allowed AC.on().

## VII. DISCUSSIONS

VETIoT depends on the baseline generation mechanism to measure the impact of the defense solution. However, a baseline cannot be generated with confidence when two or more apps create an execution loop (*e.g.*, two apps’ actions trigger each other, creating a loop).

Our in-house implementation of IoTGuard faithfully follows the description [6] as closely as possible, and we took some rational design decisions whenever the description was vague. Our evaluation results may differ if the original implementation of IoTGuard is used, since the implementation could include optimizations that were not mentioned in [6].

The defense solutions can prevent unexpected actions only when issued by apps. As a result, user interactions with the physical devices or on the platform UI can potentially move the system into an unsafe state, later causing the defense solutions enforce policy incorrectly. Our randomly generated testcases often include such events that push the system into an unsafe state, resulting in some indeterminate outcomes.

## VIII. RELATED WORK

The rapid growth of IoT innovations has resulted in multiple avenues of smart-home security research. Examples include IoT security relying on access control mechanism [19]–[24], static policy enforcement [2], [9], [25], [26] and dynamic policy enforcement [4], [6]–[8], [10], [11]. On the contrary, VETIoT provides an evaluation platform for defense solutions enforcing dynamic policies at runtime.

Testbeds are key component for any kind of research experiment. Creating testbeds to conduct large scale IoT experiments has become another research problem. Much of the prior research in this direction focuses on creating large scale physical IoT testbeds. OpenTestBed [12], LinkLab 2.0 [13], Fit IoT Lab [27], 1KT [28] are physical IoT testbed that can be accessed via web portal or ssh to conduct IoT experiments. Gotham Testbed [14] uses emulation to create IoT devices. These testbeds are used for IoT data generation and IoT network security experiments. On the other hand, our target is dynamic policy enforcement based defenses that aim to curb unexpected actions issued by automation apps, not devices. Therefore, physical testbeds and emulated devices are not essential. Hence, VETIoT utilizes an existing IoT platform equipped with virtual devices and communicates with the platform using programming APIs.

Uniform evaluation of IoT security mechanisms has been proposed for other types of IoT systems. BenchIoT [15] proposed a test suite and evaluation framework for evaluating defense mechanisms created for the micro-controllers used in IoT devices. SmartAttack [29] proposed a uniform adversarial



attack model framework to test security solutions that analyze network trace of IoT devices. Similarly, VETIoT provides an evaluation framework for policy enforcing IoT defenses.

Prior research on generating IoT events has been proposed for security evaluation and simulation purposes. Helion [30] proposes an event generation framework based on patterns found in IoT applications. Helion uses the event generation framework to define security policy. Alotaibi *et al.* [31] proposed a Smart Home Simulator, which can generate events to simulate human activity. But it requires manual UI interactions.

Event generation based evaluation approaches have been proposed for systems. Android Monkey [32] is an UI event generator built to test Android applications. Approaches like IoTFuzzer [33], DIANE [34] generate app-specific events to test smartphone companion apps for IoT.

## IX. CONCLUSION

We proposed VETIoT, a highly automated uniform evaluation platform for vetting IoT defense solutions that dynamically enforce security and safety policies at runtime to prevent unexpected actions issued by automation apps. VETIoT replaces much of the traditional experimentation process with an automated counterpart, including, testbed instantiation, testcase generation and execution, output collection and report generation. For evaluation, we demonstrated VETIoT on three existing IoT defense solutions: ExPAT, PatrIoT, and IoTGuard. Our results demonstrate that VETIoT can be effective in assessing IoT defense solutions. Researchers can leverage it to fine-tune their new defense solutions and empirically compare with existing solutions.

## ACKNOWLEDGMENT

This research was supported by the National Science Foundation under grants CNS-2006556 and CNS-2007512.

## REFERENCES

- [1] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *IEEE S&P*, 2016.
- [2] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action iot platforms," in *ACM CCS*, 2019.
- [3] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash, "ContextIoT: Towards Providing Contextual Integrity to Appified IoT Platforms," in *NDSS*, 2017.
- [4] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the internet of things," in *ISOC NDSS*, 2018.
- [5] H. Landi, "82% of healthcare organizations have experienced an iot-focused cyberattack, survey finds," <https://www.fiercehealthcare.com/tech/82-healthcare-organizations-have-experienced-iot-focused-cyber-attack-survey-finds>.
- [6] Z. B. Celik, G. Tan, and P. D. McDaniel, "Iotguard: Dynamic enforcement of security and safety policy in commodity iot." in *NDSS*, 2019.
- [7] M. Yahyazadeh, P. Podder, E. Hoque, and O. Chowdhury, "Expat: Expectation-based policy analysis and enforcement for appified smart-home platforms," in *ACM SACMAT*, 2019.
- [8] M. Yahyazadeh, S. R. Hussain, E. Hoque, and O. Chowdhury, "Patriot: Policy assisted resilient programmable iot system," in *RV*, 2020.
- [9] H. Chi, Q. Zeng, X. Du, and J. Yu, "Cross-app interference threats in smart homes: Categorization, detection and handling," in *DSN*, 2020.

- [10] W. Ding, H. Hu, and L. Cheng, "Iotsafe: Enforcing safety and security policy with real iot physical interaction discovery," in *NDSS*, 2021.
- [11] M. H. Mazhar, L. Li, E. Hoque, and O. Chowdhury, "Maverick: An app-independent and platform-agnostic approach to enforce policies in iot systems at runtime," in *ACM WiSec*, 2023.
- [12] J. Munoz, F. Rincon, T. Chang, X. Vilajosana, B. Vermeulen, T. Walcarius, W. van de Meerse, and T. Watteyne, "OpenTestBed: Poor Man's IoT Testbed," in *IEEE INFOCOM - CNERT*, 2019.
- [13] W. Dong, B. Li, H. Li, H. Wu, K. Gong, W. Zhang, and Y. Gao, "LinkLab 2.0: A multi-tenant programmable IoT testbed for experimentation with Edge-Cloud integration," in *USENIX NSDI*, 2023.
- [14] X. Sáez-de Cámara, J. L. Flores, C. Arellano, A. Urbieta, and U. Zurutuza, "Gotham testbed: A reproducible iot testbed for security experiments and dataset generation," *IEEE TDSC*, 2023.
- [15] N. S. Almkhdhub, A. A. Clements, M. Payer, and S. Bagchi, "Benchiot: A security benchmark for the internet of things," in *IEEE/IFIP DSN*, 2019.
- [16] "Iotbench-test-suite," <https://github.com/IoTBench/IoTBench-test-suite>.
- [17] OpenHAB, <https://www.openhab.org>.
- [18] "Smarthings," <https://www.smarthings.com/>.
- [19] A. Rahmati, E. Fernandes, K. Eykholt, and A. Prakash, "Tyche: A risk-based permission model for smart homes," in *IEEE SecDev*, 2018.
- [20] S. Lee, J. Choi, J. Kim, B. Cho, S. Lee, H. Kim, and J. Kim, "Fact: Functionality-centric access control system for iot programming frameworks," in *ACM SACMAT*, 2017.
- [21] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague, "Smartauth: User-centered authorization for the internet of things," in *USENIX Security*, 2017.
- [22] A. K. Sikder, L. Babun, Z. B. Celik, A. Acar, H. Aksu, P. McDaniel, E. Kirde, and A. S. Uluagac, "Kratos: Multi-user multi-device-aware access control system for the smart home," in *ACM WiSec*, 2020.
- [23] Y. Jia, B. Yuan, L. Xing, D. Zhao, Y. Zhang, X. Wang, Y. Liu, K. Zheng, P. Crnjak, Y. Zhang, D. Zou, and H. Jin, "Who's in control? on security risks of disjointed iot device management channels," in *ACM CCS*, 2021.
- [24] G. Goyal, P. Liu, and S. Sural, "Securing smart home iot systems with attribute-based access control," in *ACM SaT-CPS*, 2022.
- [25] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated IoT safety and security analysis," in *USENIX ATC*, 2018.
- [26] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. Colbert, and P. McDaniel, "Iotsan: fortifying the safety of iot systems," in *ACM CoNEXT*, 2018.
- [27] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, and T. Watteyne, "Fit iot-lab: A large scale open experimental iot testbed," in *IEEE WF-IoT*, 2015.
- [28] M. Banaszek, W. Dubiel, J. Łysiak, M. Dundefinedbski, M. Kisiel, D. Łazarczyk, E. Głogowska, P. Gumienny, C. Siłuszzyk, P. Ciołkosz, A. Paszkowska, I. Rüb, M. Matraszek, S. Acedański, P. Horban, and K. Iwanicki, "1kt: A low-cost 1000-node low-power wireless iot testbed," in *ACM MSWiM*, 2021.
- [29] K. Yu and D. Chen, "Smartattack: Open-source attack models for enabling security research in smart homes," in *IGSC*, 2020.
- [30] S. Manandhar, K. Moran, K. Kafle, R. Tang, D. Poshyanyk, and A. Nadkarni, "Towards a natural perspective of smart homes for practical security and safety analyses," in *IEEE S&P*, 2020.
- [31] A. Alotaibi and C. Perera, "Smart home human activity simulation tool for openhab-based research," Cardiff University, Tech. Rep., 2019.
- [32] "Android monkey," <https://developer.android.com/studio/test/other-testing-tools/monkey>.
- [33] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in *NDSS*, 2018.
- [34] N. Redini, A. Continella, D. Das, G. De Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, "Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices," in *IEEE S&P*, 2021.

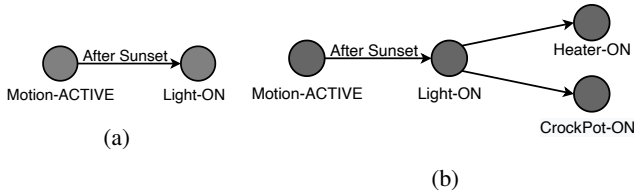


Fig. A1: (a) A dynamic model of an IoT app and (b) the unified dynamic model of interacting IoT apps

#### APPENDIX A. IOTGUARD RE-IMPLEMENTATION

IoTGuard [6] is a defense mechanism built to protect IoT system from malicious IoT applications (hereafter, apps) and unintended interactions among seemingly benign IoT applications. To ensure safety and security, IoTGuard enforces policies on IoT applications at runtime.

Policy enforcement mechanism of IoTGuard is built using three modules: *code instrumentor*, *data collector*, and *security service*. To the best of our knowledge, implementations of data collector and security service are not publicly available. IoTGuard’s code instrumentor was built for the SmartThings platform. Since VETIoT is developed for OpenHAB, we could not reuse the publicly available code instrumentor of IoTGuard.<sup>6</sup> Therefore, we had to re-implement all three modules of IoTGuard for OpenHAB.

The purpose of the code instrumentor module of IoTGuard is to instrument an IoT app with necessary hooks, which guard each action to be taken by the app. At runtime, when an instrumented app is triggered and starts executing, these hooks enable IoTGuard’s data collector to collect necessary data and invoke security service to verify safety and security of taking the contemplated action. According to [6], when an app is about to take multiple actions in the same context, the code instrumentor must deploy one common hook for all such actions. In our implementation of the code instrumentor, we first parsed the given app for OpenHAB and inserted those policy enforcing hooks before the action statement of the app. Our instrumentor follows the design mentioned in [6].

IoTGuard’s data collector collects necessary information (*i.e.*, the occurred events and the contemplated actions) from the instrumented IoT apps at runtime and stores this information in a dynamic model consisting of states and transitions. Each state represents an attribute of a device. Each transition from one state to another state represents the condition under which the state change has occurred. At a program level, this dynamic model is essentially a mutable directed graph where nodes represent states and edges denotes transitions. For example, consider an IoT app named `light-control`: “when `motion-active` after `sunset`, turn on `light`.” Upon receiving data from the instrumented `light-control` app, the data collector will create a dynamic model which has an event node “`motion=ACTIVE`” and an action node “`Light=ON`” along with an edge between them with the condition “after `sunset`” (see Figure A1a).

If multiple applications interact with each other, the data

collector will create a unified dynamic model. Consider this example. If the “`Light=ON`” event due to the action from the `light-control` app triggers another app, say, `heat-on` app: “Turn on the heater and crockpot when light is on.”, the dynamic models of the `light-control` app and the `heat-on` app will be merged to generate a unified dynamic model (see Figure A1b). In our implementation, we developed the data collector module using the Python’s Networkx library and ensured this module has all the features of the original design [6].

IoTGuard’s security service module reads safety and security policies, enforces those policies on the dynamic models generated by the data collector, and conveys the result of policy enforcement to the instrumented IoT apps. After receiving the data (*i.e.*, event, action) from an instrumented IoT app, the data collector updates (or create) the dynamic model and invokes security service to enforce policies on the latest dynamic model.

IoTGuard [6] supports three types of policies: General Policies, Application Specific Policies, and Trigger-Action Specific Policies. General policies are enforced directly on the dynamic model by applying graph-based algorithms (*e.g.*, an cycle detection algorithm in the graph). Application specific policies are written in a specific policy language described in [6]. We implemented a policy parser to read policies written in the IoTGuard’s policy language. To enforce application specific policy, our implementation of security service followed the description provided in [6].

Each application specific policy is essentially a logical implication between a premise and a conclusion. A premise/-conclusion is a logical expression consisting of one or more atoms combined using logical connectives (*e.g.*,  $\wedge$ ), where each atom denotes a proposition on a device attribute (*e.g.*, comparing a device attribute with a constant using a relational operator). Recall that device attributes (*e.g.*, `Light=ON`) are represented as nodes in the dynamic model. To enforce an application specific policy, the security service searches for a path in the dynamic model – from the node that satisfies the premise to a node that satisfies the conclusion. If there exists such a path in the dynamic model and if it is a “restrict” policy, the security service labels it as a policy violation. Then, the security service removes the nodes that were recently added by the data collector prior to this round of policy enforcement and returns a “deny” response to the instrumented IoT app.

To enforce a trigger-action specific policy, we tagged each physical device of the testbed as trusted and secure and all virtual triggers (*e.g.*, `EmailSent`, `GoogleAssistantActivated`) as untrusted and insecure. When enforcing a trigger-action specific policy, the security service searches the dynamic model for a path from a node with the untrusted/insecure tag to a node with the trusted/secure tag. If there exists such a path, the security service denies the contemplated actions and removes the recently added nodes from the model.

<sup>6</sup><https://github.com/BeerKay/SmartAppAnalysis>