Why Am I Seeing Double? An Investigation of Device Management Flaws in Voice Assistant Platforms

Muslum Ozgur Ozmen
Arizona State University
moozmen@asu.edu

Mehmet Oguz Sakaoglu
Purdue University
msakaogl@purdue.edu

Jackson Bizjak
Purdue University
jbizjak@purdue.edu

Jianliang Wu
Simon Fraser University
wujl@sfu.ca

Antonio Bianchi
Purdue University
antoniob@purdue.edu

Dave (Jing) Tian
Purdue University
daveti@purdue.edu

Z. Berkay Celik
Purdue University
zcelik@purdue.edu

Abstract

In Voice Assistant (VA) platforms, when users add devices to their accounts and give voice commands, complex interactions occur between the devices, skills, VA clouds, and vendor clouds. These interactions are governed by the *device management capabilities* (DMC) of VA platforms, which rely on device names, types, and associated skills in the user account. Prior work studied vulnerabilities in specific VA components, such as hidden voice commands and bypassing skill vetting. However, the security and privacy implications of device management flaws have largely been unexplored.

In this paper, we introduce DMC-Xplorer, a testing framework for the automated discovery of VA device management flaws. We first introduce VA description language (VDL), a new domain-specific language to create VA environments for testing, using VA and skill developer APIs. DMC-Xplorer then selects VA parameters (device names, types, vendors, actions, and skills) in a combinatorial approach and creates VA environments with VDL. It issues real voice commands to the environment via developer APIs and logs event traces. It validates the traces against three formal security properties that define the secure operation of VA platforms. Lastly, DMC-Xplorer identifies the root cause of property violations through intervention analysis to identify VA device management flaws.

We exercised DMC-Xplorer on Amazon Alexa and Google Home and discovered two design flaws that can be exploited to launch four attacks. We show that malicious skills with default permissions can eavesdrop on privacy-sensitive device states, prevent users from controlling their devices, and disrupt the services on the VA cloud.

Keywords

IoT, voice assistant platforms, privacy, security

Proceedings on Privacy Enhancing Technologies 2025(2), 719–733

https://doi.org/10.56553/popets-2025-0084

1 Introduction

Voice Assistant (VA) platforms, such as Amazon Alexa [5] and Google Home [27], enable voice-based commands to control and monitor smart devices. To achieve these, they provide developers APIs to develop voice-enabled skills[1] [13, 42].

To control a smart device with voice, a user first registers their device to its vendor (e.g., through a mobile app). The user then installs the vendor's skill to the VA platform, allowing the skill to add the registered device to the user's account. When the user gives a voice command, the VA device (e.g., Amazon Echo) sends it to the VA cloud for speech recognition and intent extraction. The VA cloud sends the extracted intent to the relevant skill that runs on a back-end server. The skill relays the command to the device vendor cloud, which handles the request and notifies the VA platform.

These complex interactions are governed by the *device management capabilities* (DMC) of VA platforms, which includes adding devices to a user account and controlling them with a skill until they are removed from the user account. During these interactions, the DMC of VA platforms provides three security properties. First, it ensures voice command *confidentiality* by determining the devices the user refers to and sending the command only to the skills that control those devices to ensure unauthorized skills do not receive privacy-sensitive voice commands and device states. Second, it offers voice command *integrity* by sending the correct command extracted from voice to the devices. Lastly, it provides voice command *availability* by ensuring the voice command is executed in a time delivering an acceptable quality-of-service ($\leq$ 8 secs [3]).

Design flaws in the DMC of VA platforms, however, may cause violations of these security properties. Our main observation is that an adversary can exploit such violations by developing a malicious skill installed by the user. Prior work has shown that malicious skills can be (1) published in the skill marketplaces due to the weaknesses in skill vetting [18, 43, 67, 71] and (2) installed in user accounts through hidden voice injection [14, 17, 21, 76, 81] and skill squatting attacks [41, 44, 59, 83, 85]. After installation, malicious skills (*by default*) have permission to add devices to the user's VA account and exploit the design flaws in the DMC of VA platforms.

[1] Voice-enabled apps are called skills in Alexa and actions in Google Home. We use the term *skill* to refer to an app for any VA platform.

Then, the adversary can exploit the DMC security property violations for three purposes. First, the adversary can access privacy-sensitive device states, e.g., whether the door is unlocked, and infer the user's routine, e.g., when they go to sleep and when they are not at home. Second, the adversary can manipulate the commands issued to the devices, e.g., causing a door to `unlock` instead of `lock`. Lastly, the adversary can cause the devices to be unresponsive to voice commands, which may put users and their physical environment at risk, e.g., a door not getting unlocked during a fire.

However, discovering design flaws in the DMC of VA platforms has three key challenges. (1) The VA platforms lack end-to-end testing environments to explore their DMC in a realistic and scalable manner. The closed-source nature of VA platforms further exacerbates this challenge. (2) VA platforms have a large parameter space, where VA environments with different numbers of devices that have various names, types, skills, and vendors can be created. (3) It requires formal definitions of the security properties that represent the correct and secure operation of VA platforms. Such definitions allow identifying property violations during testing to expose design flaws in the DMC of VA platforms.

In this paper, we address these challenges through DMC-Xplorer, an automated testing framework that explores VA operation with different skills controlling multiple devices to discover DMC design flaws in the DMC of VA platforms.

To create VA environments suitable for testing, we design a concise VA description language (VDL) with a grammar in Backus–Naur Form (BNF). When a VDL code, including a VA device and a set of smart devices with their names, types, actions, vendors, and skills, is executed, it uses VA and skill developer APIs to create a VA environment. Such APIs allow developers to create and test skills without any differences compared to when these skills are deployed in real smart homes. Therefore, integrating these APIs into VDL ensures DMC-Xplorer to *directly test the real DMC of VA platforms* in a scalable manner. DMC-Xplorer then leverages a combinatorial testing algorithm to configure VA parameters (e.g., number of devices, names, skills). It translates the parameters into VDL to create a VA environment and issues *real* voice commands using developer APIs. After each command, it logs event traces, including device information, issued command, and the time the command is received at the VA device and skill back-ends. It validates the traces against three security properties formally expressed in metric temporal logic (MTL) to identify violations. Lastly, it conducts an intervention-based root cause analysis by identifying the parameters that cause property violations to discover design flaws in the DMC of VA platforms.

We apply DMC-Xplorer to two popular VA platforms, Amazon Alexa and Google Home. DMC-Xplorer discovered that both platforms violate the voice command availability property. Through DMC-Xplorer's root cause analysis, in both platforms, we identified two design flaws that a malicious skill with default permissions can exploit. First, VA platforms allow a skill to add *duplicate devices* with the same name or type as other devices on the user's account. A malicious skill can exploit this flaw to (1) *eavesdrop* on privacy-sensitive device states when the user gives a voice command with device name or type to Google Home and a voice command with device type to Alexa, and (2) *intercept* voice commands intended for other skills when the user gives a voice command with device name

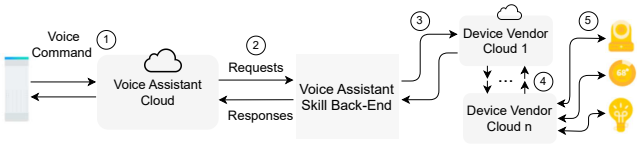

Figure 1: The interactions among VA platform components, from a given voice command to its execution.

to Alexa. Second, VA platforms *do not have a proper bound check* on the number of devices a skill can add to a user's account. This allows a malicious skill to add many devices to the user's account to (1) *delay* voice commands given to the Alexa, (2) *block* voice commands given to Google Home, and (3) overload both Google Home's and Alexa's computational resources at their cloud back-ends. We physically confirmed both design flaws exist in Alexa and Google Home by adding 1K duplicate devices to the user accounts. Unfortunately, preventing these attacks is difficult, as any restriction to skill permissions can prevent benign skills from functioning properly, causing a security-utility trade-off.

In summary, we make the following contributions.

- We design a domain-specific language (VDL) in BNF grammar to create VA environments for different VA platforms using their developer APIs.
- We introduce DMC-Xplorer, a VA operation testing framework that discovers DMC design flaws in VA platforms. DMC-Xplorer leverages VDL to create VA environments, automatically issues voice commands, logs event traces, and validates the traces against three formal security properties.
- We use DMC-Xplorer[2] on Alexa and Google Home and expose two previously unknown design flaws in their DMC.
- We develop four attacks exploiting the identified design flaws and show that a malicious skill with default permissions can eavesdrop on privacy-sensitive device states, prevent users from controlling devices, and overload the VA cloud's computational resources. We then propose three proactive and reactive countermeasures to mitigate these attacks.

Responsible Disclosure. During our testing, we use a test-user account on the VA platforms and limit our testing parameters to prevent flooding the VA cloud and sending superfluous requests.

We have sent an initial report about the design flaws to Amazon using the Amazon Vulnerability Research Program on Hackerone [7] and Google using their Vulnerability Reward Program [29]. Both Amazon and Google acknowledged the flaws and recommended users be cautious about the skills they install (i.e., installing skills from trusted vendors). DMC-Xplorer is an important step towards informing the public of the risks that skills present.

2 Background

VA platforms provide a set of APIs for third-party developers to develop skills that monitor and control devices. Users can then install skills through VA marketplaces with their mobile or desktop apps or by issuing voice commands to their VA devices.

[2]We make DMC-Xplorer available at https://github.com/purseclab/DMC-Xplorer to foster future research on the security of VA platforms.

VA platforms allow adding devices to user accounts in three ways [19, 20]: (*a*) Users can manually add devices through mobile or desktop apps; (*b*) VA platforms can detect and add devices without user interaction; (*c*) Skills have *default* permissions to add devices they detect in the user's physical environment. Similarly, users can remove devices from their accounts [56, 57] through mobile or desktop apps and by uninstalling a skill, which in turn removes all devices added by that skill.

We refer to the process of adding devices to a user's VA account and controlling them via a particular skill with a voice command until the device's removal from the user's account as the *device management capabilities* (DMC) of VA platforms. During this process, multiple VA components, both internal to the VA platform and external, interact with each other, as shown in Fig. 1.

Skill Back-end and Vendor Clouds. After the user gives a voice command, the VA device relays it to the VA cloud (①). The VA cloud converts voice into text, extracts the user's intent, determines which device the user wants to control, and sends a request to the corresponding skill back-end (②). The skill back-end then communicates with the device vendor cloud, which resolves the request (③) or relays it to other vendor clouds until the one interacting with the device is reached (④).

Voice Command Execution. The vendor cloud sends commands to devices in two ways. First, for devices connected to the Internet, the vendor cloud directly issues the command (⑤). The vendor cloud then responds to the skill back-end, which notifies the VA cloud that the command is executed. Second, for devices connected to a VA device over short-range communications (e.g., BLE), the vendor cloud forms a command message and sends it to the skill back-end. The skill back-end sends it to the VA cloud, which sends it to the VA device. The VA device issues the command to the device.

3 Problem Statement

By studying the developer documentation of popular VA platforms (e.g., Alexa [2] and Google Home [28]), we observe that the DMC of VA platforms ensures three key security properties: *Voice Command (VC) Confidentiality*, *VC Integrity*, and *VC Availability*. These properties ensure, respectively, that the VA platform does not send requests to unintended skills, sends the user-intended command to the device, and issues the commands to skill back-ends in a time delivering an acceptable quality-of-service. VA platforms achieve these properties using the device information stored in the user account, which is governed by the DMC of VA platforms.

These security properties, however, can be violated by skills due to potential design flaws in the DMC of VA platforms. The property violations express deviations from the expected execution of voice commands, indicating that commands are sent to the unintended skills to cause privacy issues, the VA sends an incorrect command to a device, and the VA fails to execute voice commands. We show in Sec. 6 that malicious skills with default permissions can exploit these violations and launch attacks on Amazon Alexa and Google Home to jeopardize the security and privacy of smart homes.

Voice Command Confidentiality (P1). This property ensures each user-intended skill receives the voice command request while others do not. For instance, when the user gives a voice command `lock the front door`, VC confidentiality ensures only the skill that controls the lock receives a request. VA platforms ensure VC confidentiality by correctly determining the device(s) the user intends to control and only issuing commands to their associated skills.

The adversary can exploit VC confidentiality violations to obtain privacy-sensitive information about smart home device states. For instance, when the user gives a voice command to turn off their bedroom lights, the adversary receives this command. Such information could help the adversary to infer the user's routine (e.g., when they go to sleep) and location (e.g., whether they are at home).

Voice Command Integrity (P2). This property holds if the VA sends the user-intended commands to devices. For instance, when the user gives a voice command `lock the front door`, VC integrity ensures the VA platform does not issue an incorrect command (e.g., `unlock`). VA platforms offer VC integrity by issuing the correct command extracted from the user's voice command to devices.

The adversary can leverage VC integrity violations to manipulate the commands given to devices. For instance, the user gives the voice command `lock the front door`, but the adversary causes the VA platform to issue the `unlock` command.

Voice Command Availability (P3). This property ensures the user's voice command is executed within an acceptable time, e.g., $\leq$ 8 secs [3]. VA platforms offer VC availability by extracting the user's intent from a voice command, finding the corresponding devices in the user's VA account, and issuing the commands within a given time frame, to guarantee the desired quality of service.

VC availability violations allow an adversary to deny or delay voice commands, preventing users from controlling their devices. This poses a security risk, especially when VAs are used to control safety-critical devices, such as door locks, cameras, and fire alarms.

3.1 Threat Model

We consider an adversary who aims to trigger the security property violations in the DMC of VA platforms to obtain privacy-sensitive information, manipulate device commands, and prevent users from controlling their devices. We assume the adversary achieves this by developing a skill installed to the user's VA account. To be installed by a victim, the adversary's skill must be first approved for publication on the skill marketplace. Unfortunately, VA platforms' skill vetting process cannot prevent the attacker's skill from being published since the adversary's skill uses the default permissions in the DMC of VA platforms. Additionally, prior work has shown the weaknesses in the skill vetting process and demonstrated that skill vetting can be bypassed and skills are only vetted when initially submitted but not routinely afterward [18, 43, 67, 71].

After vetting, the adversary can distribute the skill to many users by publishing it on the VA skill marketplace [18, 44]. To ensure that the skill is installed by users, the adversary can leverage skill squatting attacks and give their skills similar names to popular skills [41, 44, 59, 83, 85]. Additionally, the adversary can also target a specific user to install their skill. To this end, the adversary can (1) conduct hidden voice injection attacks [14, 16, 17, 21, 76, 81] (e.g., Alexa installs a skill when it is given a voice command as 'Alexa, enable [Skill Name]'), or (2) trick the user into installing the skill via phishing and social engineering methods [72].

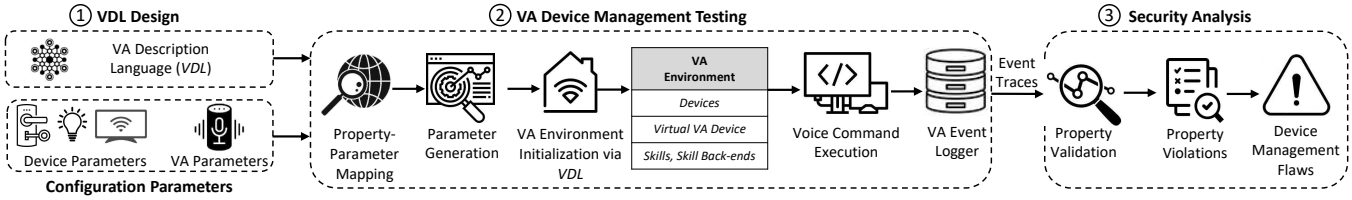

Figure 2: Overview of DMC-Xplorer architecture.

3.2 Design Challenges

C1: Creating a VA Testing Environment. Discovering design flaws in the DMC of VA platforms requires a VA testing environment that allows generating different user accounts with various parameters (e.g., devices with different names, types, actions, and skills) and sending voice commands to validate security properties. One may consider conducting tests with physical VA devices; however, multiple tests with different parameters are impractical due to the manual effort required. Unfortunately, the closed-source nature of VA platforms prevents us from directly testing their DMC software to expose design flaws. Although VA and skill developer APIs exist for black-box testing, simply sending inputs through such APIs is not enough to expose DMC design flaws. This is because identifying the design flaws in the DMC of VA platforms requires creating different VA environments with various parameters.

To address this, we design a domain-specific language that uses VA and skill developer APIs to generate real VA environments that enable creating different tests and issuing real voice commands in an automated and scalable manner. However, designing a generalizable and practical domain-specific language for VAs is a challenging task because VA environments can include diverse devices (e.g., light, camera, door lock) with different actions (e.g., on/off, dimming).

C2: Large Parameter Space. Various user VA accounts can be generated with different numbers of devices with different names, types, and skills. One may use falsification [8, 52] or fuzzing [49] to identify the user account parameters (e.g., number of devices and their names, types, and skills) that violate the security properties. Yet, falsification needs numerical feedback for parameter selection. Unfortunately, VA security properties are boolean-typed (satisfied or violated) and, thus, cannot guide falsification. Similarly, most fuzzing techniques require feedback from prior tests (e.g., code coverage). Since VA platforms are closed-source, they are infeasible.

One may use black-box fuzzing [74] without guidance. Yet, it would generate random parameters, which may not expose property violations. There are also black-box fuzzing techniques with guided input mutation, such as the skill squatting attacks [41] that mutates voice to find phonetically similar words. However, these mutation strategies are designed specifically to expose different types of flaws and vulnerabilities, and therefore, cannot be applied to identify DMC design flaws. To address these, we first identify the VA parameters that influence security properties. We then introduce a combinatorial approach to test parameter combinations and discover design flaws in the DMC of VA platforms. To our knowledge, our mutation strategy is the first to create semantically different VA environments by mutating VA parameters.

C3: Formally Checking the Security Properties. After determining the user account parameters, we must send voice commands and check if the security properties hold in each test. This requires formally representing the properties amenable for validation. Prior work on VA security has explored properties to identify and prevent privacy-sensitive information leakages [30, 47, 62] and check which skill is invoked given a voice command [41, 85]. Although such properties can be useful in detecting VC confidentiality violations, they cannot detect VC integrity and availability as these properties require reasoning about the specific voice command sent to devices and the temporal aspects of VC execution. To address this, we formally define new security properties through MTL, which extends the operators of linear temporal logic (LTL) with timing constraints [40]. Lastly, VA platforms do not generate logs that would allow property validation. Thus, we build a logging mechanism that records events during VA platform operation to validate the security properties after each voice command.

4 DMC-Xplorer Design

We introduce DMC-Xplorer, an automated VA operation testing framework that identifies design flaws in the DMC of VA platforms. Fig. 2 illustrates the overview of DMC-Xplorer.

We first design a domain-specific language VDL (VA Description Language), addressing **C1** (①, Sec. 4.1). VDL provides a concise syntax to create an environment with a VA controlling multiple devices parameterized by their names, types, vendors, actions, and associated skills. When a VDL code is executed, it generates a VA environment by leveraging VA and skill developer APIs to add devices and skills to a user account and associate them with a virtual VA device and skill back-ends. DMC-Xplorer then addresses **C2** by selecting VA parameters through a combinatorial algorithm that generates different numbers of devices with various names and types, controlled by single or multiple skills (②, Sec. 4.2). It next uses VDL to create a VA environment from the parameters.

DMC-Xplorer then issues real voice commands to the created VA environment through developer APIs. This ensures direct testing of the DMC of VA platforms when a voice command is given. After each voice command, DMC-Xplorer logs event traces with device configurations, issued voice commands, and the requests each skill receives. Lastly, it validates the traces against three security properties formally represented in MTL and conducts an intervention-based root cause analysis to discover the design flaws in the DMC of VA platforms, addressing **C3** (③, Sec. 4.3).

Target Application of DMC-Xplorer. We introduce DMC-Xplorer for VA developers to identify the design flaws in the DMC of their platforms. Identifying such flaws during development can allow developers to make more informed design choices and revise their

```
setupInst := Setup (vaList)
vaList := vaInst | vaList
vaInst := VoiceAssistant (vaProperties)
vaProperties := vaPlatform, vaDevType
devList := devInst | devInst @ actList | devList
deviceDefn := class Device[(SmartHome)]:
              (ID: devID, vendor: vName, skill: skillID)
devInst := Device (devProperties)
devProperties := ID, vendor, skill
actList := action | action @ actPrmList | actList
action := string
actPrmList := actionParameter | actPrmList
actionParameter := string | integer
Basic data types;
vaPlatform := string
vaDevType := string
devID := string
vName := string
skillID := string
For completeness;
string := char | char string
char := 0x00 | 0x01 | . . . | 0xFF
integer := ['+' | '-'] naturalNumber
naturalNumber := digit | digit naturalNumber
digit := '0' | '1' | . . . | '9'
```

Listing 1: Formal VDL grammar.

implementation of the DMC of their platforms. Although the intended users are developers, VA end users can also use it to identify such flaws, informing them about possible risks, as it doesn't require VA platform source code. We note that during testing, DMC-Xplorer does not consider any malicious skills (considers all skills are benign) while identifying the design flaws. Yet, in Sec. 6, we show how such design flaws can be exploited through malicious skills.

4.1 VA Description Language (VDL)

We design a domain-specific language, VDL, which provides a concise syntax to simplify the task of creating VA environments. VDL is a key component of DMC-Xplorer since it allows creating VA environments for different platforms (e.g., Alexa and Google Home) with different sets of devices controlled by multiple skills. DMC-Xplorer uses these environments to issue voice commands and validate VA platforms' behaviors against security properties to discover the design flaws in their DMC (detailed in Sec. 4.2).

4.1.1 VDL Grammar. Listing 1 presents the VDL grammar in the BNF notation. VDL describes VA environments with a VA device and a set of devices in the user account. To determine the attributes/parameters of the VA environments, we study the developer documentations of popular VA platforms and analyze how they store the VA and device information. We define a VA device with two attributes: platform (e.g., Alexa) and type (e.g., Echo). Each device in the user account has a type, name, actions, vendor name, and associated skill. VDL includes default actions as `turn on` and `turn off`, and allows users to define other actions. Here, device actions can include numerical values (e.g., dim the light to 10%). Thus, VDL includes an action parameter list that supports integer values.

Table 1: Built-in smart home device classes and their actions.

Smart Home Classes	Supported Actions
Lights	turnOn, turnOff, brighten, dim
Thermostat	setTemp, increase, decrease
Lock	lock, unlock, status
Cooler	increaseSpeed, decreaseSpeed
Camera	answer, show, hide
Washing Machine	status, pause, resume
Refrigerator	checkTemp, setTemp
Television	setVolume, play, pause

```
1  class waterHeater(smartDevice):
2    def __init__(self, devID, vName, skillID ):
3      smartDevice.__init__(self, devID, vName, skillID )
4      waterHeaterInst = waterHeater("3", "ManuC", "Skill3")
```

Listing 2: An example of a new class definition in VDL.

@ Operator. We define a linkage (@) operator in VDL to associate action parameters to actions, actions to devices, and devices to VAs while constructing a VA environment.

Classes and Objects. We integrated eight common device types as built-in classes for VDL. Table 1 outlines the built-in device classes and their actions. We define these smart home device types based on a generic `Device` superclass that contains `ID`, `vendor`, `skill` class variables. Since VAs have similar capabilities, we implemented a single VA class named `VoiceAssistant` that has two class variables, the VA platform (i.e., Alexa, Google Home) and VA device type (i.e., Echo, Nest). We also define a `Setup` class that is called to create the VA environment through API calls.

The code block in Listing 2 shows the implementation of a new smart home device class and the creation of an instance of the newly implemented device. In this example, the user constructs a new device class (`water heater`) that is not built-in and creates an instance that can be used in testing the DMC of VA platforms. This feature gives users a variety of testing options since new smart home device classes can be easily introduced.

Functions. To improve the usability of VDL, we develop functions to make it concise and clear. Since DMC-Xplorer's DMC testing includes repetitions of the parameters, we introduce a set of functions to facilitate efficient grammar.

- **Group.** DMC-Xplorer generates VA parameters in which a group of smart devices has the same name (detailed in Sec. 4.2). To reduce code repetition, we propose a group function to facilitate defining such device groups. The group function takes a smart device class, a list of device IDs, vendors, skills, and the linked VA object as parameters and returns a list of corresponding device objects.
- **Compare.** This `setup` class function generates the same set of devices for two different VAs. It can be used to compare the behavior of two different VAs for the same set of devices.
- **All.** This `setup` class function generates a sample device from all device classes. This function only requires defining a VA and setup class. It then generates devices of all types and assigns a valid value for the parameters of each device.

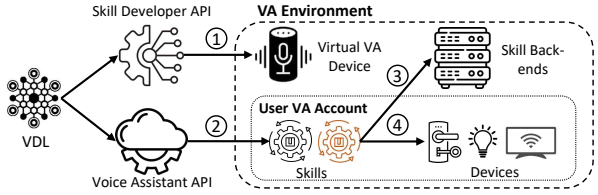

Figure 3: VDL executor's VA testing environment creation.

```
1  alexaEnv = VoiceAssistant("Alexa","Echo")
2  dev1 = Light("bedroom light","Philips", "Skill1")
3  dev2 = Lock("front door lock","Schlage","Skill2")
4  LightActionList = ["turnOn","Dim@10","turnOff"]
5  LockActionList = ["lock","unlock"]
6  dev1@LightActionList
7  dev2@LockActionList
8  alexasetup = Setup(alexaEnv)
9  alexasetup@dev2
10 alexasetup@dev1
```

Listing 3: An example VDL code that corresponds to a VA testing environment with Alexa, two devices, and two skills.

4.1.2 VDL Execution. Figure 3 shows how VDL executor creates a VA environment, including a virtual VA device, a set of devices, and a set of skills with back-ends, using VA and skill developer APIs. These APIs are provided to developers to create and test their skills without any differences when these skills are deployed in real smart home environments. Therefore, integrating such APIs into VDL allows us to *test the real DMC of VA platforms*.

Given a VDL code, VDL executor first links the virtual VA device to a user account using the developer APIs (①). Through the virtual VA device, VDL executor enables sending real voice commands to the VA cloud. VDL executor next builds the skills and their back-ends, which handle the voice command requests (②-③). It uses VA APIs to set the skill type and the workflow between the VA cloud and the skill back-end. It specifies an API endpoint on a web server as the skill back-end. The skill back-ends directly respond to requests from the VA cloud to control devices, mimicking the operation of vendor clouds.

After creating skills and their back-ends, VDL executor associates each skill with an authorization code, which grants the skill permission to add devices to the user's VA account. Thus, it leverages the skills to add devices with names, types, actions, and vendor names provided in the VDL code to the user account (④).

Example. We present a sample VDL code in Listing 3, which represents an Alexa-controlled VA environment with a light and door lock. Given this code, VDL executor first defines a virtual VA device for `Alexa` (Line 1). It then instantiates two devices with built-in `light` and `lock` classes (Line 2-3). For instance, from Line 2, it sets the device's type as `light`, name as `bedroom light`, vendor as `Philips`, and skill as `Skill 1`. It then creates device actions (Line 4-5), e.g., the `lock` and `unlock` actions for the front door lock.

Here, we leverage the linkage operator (`@`) to associate illuminance `action parameter` to an `action` (bedroom light's dim) (Line 4) and the device action lists to the devices (Line 6-7). Lastly, VDL executor sets up the Alexa environment (Line 8) and links the devices to the VA setup (Line 9-10).

Algorithm 1 DMC-Xplorer VA Device Management Testing

Input: Security Properties (L_p), VA Parameter-Property Mapping ($M[v,p]$), List of voice commands (L_{vc})
Output: Data Traces (**D**), Property Violations (L_{vio})
1: **function** VA_Explore(L_p, $M[v,p]$, L_{vc})
2: **for each** p in L_p **do**
3: $L_v = \arg_j(M[j,p] = 1)$
4: params ← Combinatorial(L_v)
5: VDL ← create_vdl(params)
6: va_env ← vdl.execute()
7: D_i ← va_env.execute(L_{vc})
8: $\mathbf{D} = \mathbf{D} \cup \{D_i, \text{params}\}$
9: **if** $D_i \not\models p$ **then** $L_{vio} = L_{vio} \cup$ params
10: **end if**
11: **end for**
12: **return D**, L_{vio}
13: **end function**

4.2 VA Device Management Testing

DMC-Xplorer uses VDL to create VA environments and issues real voice commands to discover design flaws in the DMC of VA platforms. For each environment, DMC-Xplorer must determine the VA parameters, namely, the number of devices and each device's type, name, vendor, skill, and actions. These parameters encompass all parameters that a VA uses in the user account for their DMC. As detailed in **C2**, Sec. 3.2, unfortunately, prior parameter generation methods may not uncover DMC design flaws in VA platforms.

To address this, we first map the security properties to the VA parameters that may influence them. VA parameters are the variables in VDL (number of devices and each device's name, type, vendor, skill, and actions), and properties are VC confidentiality, integrity, and availability, as detailed in Sec. 3. DMC-Xplorer then selects the parameters in a combinatorial approach, creating parameter combinations that comprehensively cover the parameter space. It next creates testing environments through VDL and issues real voice commands, as shown in Algorithm 1.

The algorithm takes, as input, security properties, a list of voice commands, and the VA parameter-property mapping. For each security property, the algorithm finds the parameters that impact it from their mapping (Line 3). It then generates VA parameters through a combinatorial approach (Line 4) and translates them into VDL (Line 5). It executes the VDL code to create a VA environment through VA developer APIs (Line 6). It then issues real voice commands to the devices (Line 7) and logs event traces, including timestamped commands and their skill back-end requests (Line 8). It uses these traces to validate formally represented security properties and discover design flaws in the DMC of VA platforms (Line 9-10) (See Sec. 4.3).

4.2.1 VA Parameter and Security Property Mapping. We construct a binary matrix, $M[v,p]$, by studying VA developer documentations, to map each VA parameter (v) to the security properties (p) that they may impact. In the matrix, $M[v,p] = 1$ means v influences p and 0 means it does not. This matrix allows selectively mutating the parameters for a scalable exploration of the DMC of VA platforms.

First, the device types and names may influence all security properties as they are included in voice commands. Second, the number of devices may also influence all properties. as more devices in the user account may increase the request delays of voice commands and complicate correctly determining the skills and actions. Third, the device actions may impact VC integrity, as they may cause mismatches between the given voice command and the device's

supported actions. Yet, actions do not impact the voice command delay and which devices receive the given command. Fourth, the skills may affect all properties because they receive the given voice commands. Lastly, the device vendors do not impact any property since they are not referred to in voice commands, and do not influence the request delay or actions devices receive.

4.2.2 VA Parameter Selection. DMC-Xplorer generates different sets of parameters, guided by the parameter to security property mapping, to create VA environments for property validation. It adopts a hierarchical approach to selecting VA parameters. It first determines the number of devices, and for each device, it creates its types, names, skills, and actions.

Selection of the Number of Devices. DMC-Xplorer determines the max number of devices it can add to a user's VA account, and if the VA platform does not impose a limit, it stops at a user-defined limit to enable scalable exploration. DMC-Xplorer then uses a scaling parameter (ρ) to divide the max number of devices into specific values. For instance, if the max value is 1000 and $\rho = 6$, it creates $\{2, 200, 400, 600, 800, 1000\}$ devices in each test. Here, the min number of devices is 2 as a single device would generate trivial tests.

Selection of Device Parameters. DMC-Xplorer next selects each device's parameters by focusing on the property-parameter mapping. Our observation is that, for the DMC of VA platforms, the specific device types, names, skills, and actions are insignificant. Instead, the number of devices having the same parameter may impact security properties since it determines the number of devices a voice command intends to control. For instance, while creating two devices, generating a `light` and `lock` is indifferent from generating a `camera` and `TV` as each device type is unique and the command will only refer to one device. Thus, instead of mutating parameter values, DMC-Xplorer mutates the number of devices that have the same property. This ensures that it generates semantically meaningful parameters for comprehensive and scalable testing.

To this end, DMC-Xplorer uses another scaling parameter (ω) indicating the portion of devices that have the same name, type, skills, or actions. For instance, if the number of devices is 20 and $\omega = 5$, it creates $\{1, 5, 10, 15, 20\}$ devices with the same type in each test and sends the voice command to the given type (See Sec. 4.2.3). The higher values of the scaling parameters (ρ, ω) provide a more fine-grained search, whereas their lower values offer higher scalability. We set these parameters in Sec. 6 to values that enable a reasonable trade-off. While determining each test's parameters, DMC-Xplorer starts with the min number of devices and mutates each parameter until a property violation occurs or the parameters are completed.

4.2.3 Voice Command Execution. After determining VA parameters, DMC-Xplorer leverages VDL to create a VA environment and issues voice commands for testing. Particularly, DMC-Xplorer parses each set of VA parameters into a VDL code. DMC-Xplorer then executes the VDL code to generate a VA environment. Lastly, it issues real voice commands to the created environment.

4.2.4 VA Event Logger. DMC-Xplorer logs events for each issued voice command (`t`) and for each request received by skill back-ends (`d`), enabling it to validate security properties.

Voice Command Events. We define each voice command event (`t`) as `t` = (`devices`, `vc`, `ts`, `exNum`). The `devices` is an array of devices that the voice command (`vc`) intends to control. The `vc` has three attributes: command, type, and name, e.g., the command `turn on all lights` contains `vc.command` = `turn on`, `vc.type` = `Light`, `vc.name` = `null` attributes. The `ts` is the timestamp when the voice command is issued. The `exNum` is a unique voice command ID used to relate voice command and skill back-end events.

```
1   D1 = {t1, d1,1, d1,2} // Events recorded
2   t1 = (devices:[bedroom light, kitchen light],
    vc:[type:Light, command:turn on], ts:5:00:00,
     exNum:1) // [turn on all lights] command given at 5:00:00
3   d1,1 =(Skill 1, cmd:turn on, ts:5:00:02, ex:1)
4   d1,2 =(Skill 2, cmd:turn on, ts:5:00:04, ex:1)
```

Listing 4: Example logs collected by DMC-Xplorer.

Skill Back-end Events. DMC-Xplorer logs events at skill back-ends (`d`) for each request received from the VA cloud. This enables DMC-Xplorer to analyze the requests the VA cloud issues after it receives the voice command. We express each skill back-end event as `d` = (`skill#`, `command`, `ts`, `ex`). The `skill#` is an identifier. The `command` is the request type received from the VA cloud, e.g., `turn on`, and the `ts` is the request's timestamp. Lastly, the `ex` is the last issued voice command's ID, which is used to associate each voice command event with skill back-end events.

The logs contain a data trace for each issued voice command as $D_i = \{t_i, d_{i,1}, d_{i,2}, \ldots, d_{i,m}\}$, where `m` is the number of requests received across all skill back-ends after the voice command t_i is issued. Each event $d_{i,j}$ is one request issued from the voice command `i` and received by the skill back-end `j`.

Example. Listing 4 shows example events DMC-Xplorer logs. Here, DMC-Xplorer adds two devices to the user's VA account, (1) a device named `kitchen` with type `Light`, and associated with `Skill 1`, (2) and a device named `bedroom` with type `Light`, and associated with `Skill 2`. DMC-Xplorer then issues a voice command as `turn on all lights`. It logs an event to record the voice command (t_1), and two events to record the requests received by the two skill back-ends ($d_{1,1}$ and $d_{1,2}$).

4.3 Security Analysis

4.3.1 Security Property Formalization. We represent the security properties (detailed in Sec. 3) with MTL. This formalization transforms high-level security properties into verifiable formulas that allow formally validating them on event traces. Table 2 presents the security properties, their formalization, and their security goals.

VC Confidentiality (`P1`). `P1` checks whether each user-intended skill receives a request for the user's voice command and other skills do not. For this, `P1` validates whether each skill that receives the voice command controls the device(s) referred by the user. To determine if a skill controls a specific device, this property checks the list of devices that the skill has added to the user account. `P1` violations mean that a skill that does not control any of the devices the voice command refers to, indicating an unauthorized skill has received the voice command.

VC Integrity (`P2`). `P2` checks whether the correct command given by the user is issued to the device. To this end, it validates if the command in the voice command event (`t`) is equal to the command

Table 2: Descriptions and formal representations of security properties ensured by the DMC of VA platforms.

ID	Security Property	Description	Formal Representation†	Security Goal
P1	VC Confidentiality	Each user-intended skill must receive the VC request while the others must not.	$\square$(d.skill.devices $\in$ t.devices)	Prevent adversaries from eavesdropping on privacy-sensitive device states.
P2	VC Integrity	The correct commands intended by the user must be sent to devices.	$\square$(d.command = vc.command)	Prevent adversaries from sending unintended commands to devices.
P3	VC Availability	The user's voice command must be executed within a quality of service time.	$\square$(d.command $\rightarrow \diamond_{[0,thld]}$ vc.command)	Prevent adversaries from blocking or delaying voice commands.

† $\square$ means always. t.devices are the devices the voice command refers to, and d.skill.devices represents the devices controlled by the skills that receive a request for the given command. d.command is the command the device receives, and vc.command is the voice command. $\diamond_{[0,thld]}$ means eventually in [0, thld] secs.

in the skill back-end event (d). P2 violations mean the VA platform issues unintended commands to devices.

VC Availability (P3). P3 validates if the given voice command is executed within a delay threshold (i.e., quality of service time). We determine the delay threshold (thld) as 8 secs based on Alexa and Google Home developer documentations and empirically validate it. If P3 holds, the voice command is executed without delay.

4.3.2 Security Property Validation. DMC-Xplorer validates MTL formulas on the recorded event traces after executing each voice command. Our formalization allows using any MTL checker for property validation. In the DMC-Xplorer prototype, we integrated PSY-Taliro [68] to validate the MTL formulas on event traces through a dynamic programming-based algorithm. However, it depends on additional libraries. For a stand-alone implementation, we leverage the intuitiveness of our security properties and implement a custom property-checking algorithm. After validating each security property, DMC-Xplorer outputs the VA parameters, logs, and whether each property is violated or satisfied.

4.3.3 Intervention-based Root Cause Analysis. DMC-Xplorer's output allows analyzing the root causes of the violations by determining the common VA parameters that cause the violations. This approach is similar to the intervention analysis in causal inference, which enables distinguishing causal structures based on observations from intervening in a variable [31].

In our context, when DMC-Xplorer mutates a VA parameter to different values while keeping other parameters the same, and it does not cause a property violation, then this parameter cannot be the root cause. In contrast, if changing the parameter (while keeping others the same) causes a violation, then this parameter's change is the root cause. Particularly, we compare two probabilities,

$$\mathsf{Prob}(\mathsf{p}|\mathsf{do}(\mathsf{v_i} = \mathsf{V_{i,1}})) \stackrel{?}{=} \mathsf{Prob}(\mathsf{p}|\mathsf{do}(\mathsf{v_i} = \mathsf{V_{i,2}}))$$

where p is a security property, do represents the interventions, $\mathsf{v_i}$ represents the VA parameters, and $\mathsf{V_i}$ represents specific parameter values. If a parameter is the root cause for violations, its changes cause a significant increase in the probability of the security property being violated, allowing us to identify the root causes.

Since DMC-Xplorer's VA device management testing algorithm mutates parameters individually, its output enables us to directly conduct intervention-based root cause analysis without running additional tests. In turn, the identified root causes reveal the design flaws in the DMC of VA platforms.

5 Implementation

We have implemented DMC-Xplorer for Alexa [5] and Google Home [27] as they are the most popular VA platforms. DMC-Xplorer requires minimal manual effort before automated testing: (1) creating a user account and a virtual VA device, (2) building skills through smart home skill templates, (3) setting up an event logger, and (4) determining the request delay threshold. DMC-Xplorer then automates VA device management testing on Alexa and Google Home APIs by sending pre-recorded audio to the virtual VA device. To detail, we first create a test VA account on both platforms to enable VDL executor to install skills that add devices to the user account. We then set up a virtual VA device using the VA SDKs. For each skill, VDL executor uses the VA APIs to create their back-end.

As the VA event logger, we configure a relational database on a cloud service [6]. For both VA platforms, we determine the *request delay threshold* required to validate VC availability (P3) through developer documentations [3]. These documentations state that VA platforms time out and give users an error message after 8 secs. Thus, we set the delay threshold at 8 secs. We empirically confirmed this threshold by issuing a voice command and delaying the response from our skill back-end to record the time Alexa and Google wait before giving an error. We detail the manual effort required for porting DMC-Xplorer to other VA platforms in Sec. 7.2.

5.1 Platform-specific Implementation Details

Amazon Alexa. We implement Alexa skill back-ends in Node.js on AWS lambda [10] to handle `Discovery` (initial connection), `AcceptGrant` (account sign-in), `ReportState` (state query), and `PowerController` (device control) queries [73]. VDL executor adds devices proactively, so it does not report any devices in its response to Discovery messages. In `ReportState`, VDL executor reports the device is currently in a hard-coded state. For `PowerController`, it reports the device's new state is whichever state Alexa required the skill to change it to. Each skill back-end includes approximately 200 LoC used to handle the requests. We use the same code for each skill back-end except for a hard-coded skill number.

For account sign-in, we leverage the Login With Amazon [48] authentication model. We get an access code and a refresh code rather than one authorization code. The access code is used in proactive requests to the Alexa API, identifying the user, skill, and authorizing requests to add devices. Although the code expires after an hour, sending the refresh code to the Login With Amazon API retrieves a new access code.

VDL executor uses the Alexa `/v3/events` API [73] to report the devices from each skill. This API limits the number of reported devices per message. We space the requests by 100 ms to comply

Table 3: Design flaws discovered by DMC-Xplorer and the attacks that can be performed on Alexa and Google Home.

Exploited Design Flaw					Impacted Platform	
Duplicate Devices	**No Proper Bound Check**	**Attack**	**User Voice Command**	**Attack Consequence**	**Alexa**	**Google**
✓	✗	Eavesdropping	[Cmd] [Type]	Malicious skill learns the voice command sent to a benign skill	✓	✓
			[Cmd] [Name]		✗	✓
✓	✗	Intercepting	[Cmd] [Name]	Malicious skill intercepts the voice command sent to a benign skill	✓	✗
✓	✓	Delaying	[Cmd] [Type]	Malicious skill delays the voice command sent to a benign skill	✓	✗
✓	✓	Blocking	[Cmd] [Name/Type]	Malicious skill blocks the voice command sent to a benign skill	✗	✓
✓	✓	VA Overloading	[Cmd] [Type]	Malicious skill overloads the Alexa servers	✓	✗

with these checks. To set up a virtual VA device, we use Alexa's code-based linking [9] to get an authorization code and issue requests to the speech API [4] on behalf of the user. DMC-Xplorer then sends the voice commands, containing the authorization code and an audio file, to the `SpeechRecognizer` API.

Google Home. We set up a Firebase function [26] for the skill back-end of Google Home actions to handle the login procedure (initial connection and account sign-in), `onQuery` (state query), `onExecute` (device control), and `SYNC` (add devices) requests [36]. The Firebase function also gets authorization to use the `HomeGraph` API, which allows issuing `RequestSync` messages to the Google Home cloud. Each Google skill has the same back-end, around 300 lines of Node.js code, with just a hard-coded skill number changed.

There is no Google Home API to add devices directly. Thus, VDL executor uses `RequestSync` and `SYNC` messages to accomplish this. A skill sending a `RequestSync` message to Google Home causes Google to reply with a `SYNC` message. The skill then responds to this message specifying the devices it is adding. For this reason, the skill back-end creates the device configuration when it receives a `SYNC` request. To this aim, it reads the current execution number from the database, and it uses the current execution number and input arrays to determine the devices that it needs to send in the report. VDL executor invokes each skill to send a `RequestSync` message to Google Home and updates the execution number in the database instead of adding devices directly.

6 Evaluation

We leverage DMC-Xplorer to discover design flaws in the DMC of two most popular VA platforms, Amazon Alexa and Google Home.

DMC-Xplorer selects the VA parameters through a combinatorial approach, with the scaling parameters ρ and ω set as 6. We select these parameters since they enable a scalable but comprehensive exploration. We found that a more fine-grained exploration does not reveal additional design flaws while increasing the testing overhead. DMC-Xplorer then translates the parameters into VDL and executes the VDL code to generate VA environments. To control the devices, DMC-Xplorer issues voice commands as `turn on all [device type]` and `turn on [device name]`.

We run our experiments on a computer with an Intel Core i5 CPU running at 3.9 GHz with 16 GB RAM.

Property Violations. DMC-Xplorer finds that both Alexa and Google Home violate the `P3` (VC availability) security property. These violations indicate that the voice command execution is delayed by more than 8 secs. DMC-Xplorer also finds that neither platform violates the `P1` and `P2` properties, indicating that (1) the skills that receive voice commands have added the user-referred devices to the user's VA account, and (2) the commands delivered to the skill back-ends correctly match the issued voice commands. To ensure that the parameters eliminated with the parameter-property mapping matrix (See Sec. 4.2.1) do not cause missed violations, we conducted experiments while mutating those parameters and found that they do not cause additional violations.

Root Causes. DMC-Xplorer's root cause analysis indicates that the property violations on Amazon Alexa are due to devices with the same name or type being in the user account or a high number of devices. For instance, DMC-Xplorer found that when it adds 200 devices with the same name to the user account from two skills and issues the command `turn on [device name]`, Alexa only sends a request to the last skill that added the device.

For Google Home, DMC-Xplorer found that all violations occur when 1K devices with the same name or type are added to the user account. DMC-Xplorer discovered that although Google Home allows a skill to add 1K devices to the user account, it does not send a request to skill back-ends when the user issues a voice command if the devices have the same name or type, violating `P3`.

Thus, DMC-Xplorer's root cause analysis found there are two design flaws in the DMC of Alexa and Google Home: (1) duplicate devices in the user account and (2) no proper bound on the number of devices added by a skill. *We physically validated that both design flaws exist in Alexa and Google Home by adding 1K duplicate devices to the user VA accounts.* Table 3 summarizes the design flaws and the attacks we introduce to exploit them through malicious skills.

6.1 Duplicate Devices in the User's VA Account

We refer to duplicate devices as devices in the user's VA account that have the same name and/or type. DMC-Xplorer's root cause analysis found that both platforms allow skills to add duplicate devices to user accounts. Although duplicate devices may have unique ids that distinguish them in the user account, when a user issues a voice command, the VA platforms fail to distinguish them.

The VA platforms do not prevent skills from adding duplicate devices for two main reasons. First, users may have multiple devices with the same type (e.g., multiple light devices) or name (e.g., multiple living room lights) in their home. Second, a user may register the same device to multiple skills (e.g., the user registers a Hue light through a Philips skill and the Apple HomeKit skill), which in turn causes these skills to add duplicate devices to the user account.

To investigate the impact of duplicate devices in user accounts, we leverage the DMC-Xplorer event traces and analyze the number of requests received by skill back-ends. We found that when the user issues a voice command `[Cmd] [Device Type]` (e.g., `turn on the lights`), both platforms send a request to all devices of the given type. On Alexa, when the user controls devices with a specific name,

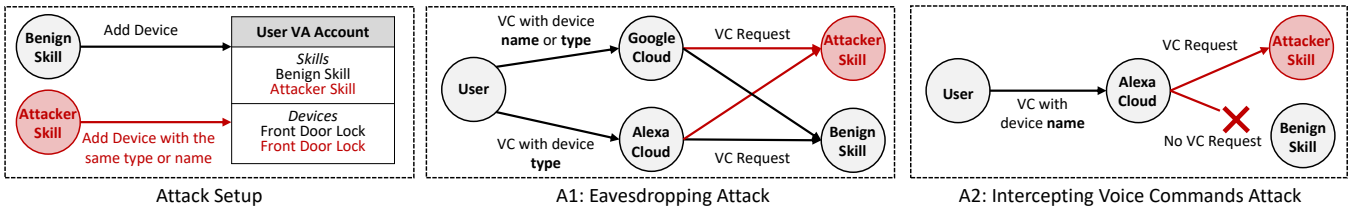

Figure 4: Illustration of eavesdropping and intercepting voice command (VC) attacks with a finite state machine. Each node represents a VA component, and the edges represent their interactions.

it sends a request to the most recently added device with that name. For example, if there are multiple devices named `bedroom light` and the user issues the voice command `turn on bedroom light`, only the last added `bedroom light` receives the request. In contrast, when the user issues a voice command to a device name on Google Home, it sends a command to every device with that name.

Our findings on Alexa and Google Home show that skills can add duplicate devices to user accounts with any device information *without notifying the user*. An attacker can leverage malicious skills and exploit such DMC permissions to launch two different attacks.

6.1.1 A1: Eavesdropping on the Privacy-Sensitive Device States. We design an attack in which a malicious skill (installed by a user) adds devices with the same type or name as another device to eavesdrop on the user's voice commands. The attacker can launch this attack by the following three steps, as illustrated in Fig. 4.

❶ The attacker skill adds a device with the same name or type as another device to the user's VA account.

`Benign Skill` → `door`
`Attacker Skill` → `door`

❷ The user issues a voice command to a device type or name (`[Cmd] [Type/Name]`), where an attacker has added device(s) with that device type/name to the user account.

`User` → `unlock all doors`

❸ The attacker skill learns the user's voice command.

`unlock-door` → `Benign Skill Back-end`
`unlock-door` → `Attacker Skill Back-end`

Attack Impact. When a user issues a voice command to a device type (e.g., `lock the doors`) or name (e.g., `lock the patio door`), the attacker skill back-end receives the request to change device states (Table 3, first row). Here, the attacker can infer if the user's doors are locked or not. Since the attacker is not limited in the number of devices they can add to a user account, they can add many devices with common types and names and obtain device states every time a user issues a voice command (detailed in Sec. 6.1.3).

Additionally, this attack is stealthy, as benign skills and physical devices receive voice commands and operate as usual. For instance, when the user issues the `unlock all doors` command, the doors physically get unlocked. However, the attacker also receives this command, unbeknownst to the user.

The eavesdropping attack does not apply to Alexa when the user issues a voice command to a device with its name. This is because, in such cases, Alexa only sends the command to the last added duplicate device, which makes Alexa vulnerable to our voice command interception attack, as detailed next.

6.1.2 A2: Intercepting Voice Commands. We design an attack on Alexa where an attacker intercepts the voice commands sent to devices that a user intends to control, as shown in Fig. 4. While **A2** has similar steps with **A1**, here, the attacker needs to add a device to a user account with the same *name* already added by the user.

❶ The attacker skill adds a device with the same name as another device on the Alexa user account.

`Benign Skill` → `front door`
`Attacker Skill` → `front door`

❷ The user issues a voice command to a device name where an attacker has added a device with that name.

`User` → `lock the front door`

❸ The attacker skill intercepts the user's voice command.

`lock-front-door` ↛ `Benign Skill Back-end`
`lock-front-door` → `Attacker Skill Back-end`

Attack Impact. When a user issues a voice command to a device name, the attacker's skill intercepts the request, preventing the physical device from receiving the command (Table 3, second Row). This attack does not apply to Google Home because it sends a request to every device with the same name rather than the most recently added device, making it susceptible to the eavesdropping attack, as detailed in Sec. 6.1.1.

The users can notice this attack since their voice command will not be executed by the physical device. Yet, the attacker can carefully select which commands they intercept to remain stealthy. For instance, intercepting a `turn off the bedroom light` command can be noticed. On the contrary, users may be less likely to notice if the attacker intercepts the commands given to cameras (e.g., `turn on the front camera`) and door locks (e.g., `lock the front door`). To be more stealthy, the attacker's skill back-end can also send fake confirmation messages to the user, stating the command has been executed, while in reality, it has not.

6.1.3 Selecting Device Types and Names. The eavesdropping and intercepting attacks require the attacker to add a duplicate device to the user's VA account. We conduct additional analysis to understand how an attacker can determine which device types and names they can add to users' VA accounts to maximize their chances of eavesdropping on and intercepting voice commands.

We found there are 17 supported device types on Alexa [5] and 79 on Google Home [27]. Thus, *to conduct an eavesdropping attack, the attacker only needs to add* 17 *devices to Alexa and* 79 *devices to Google Home.* This would allow the attacker to eavesdrop on *any* voice command given as `[Cmd] [Device Type]` for both platforms.

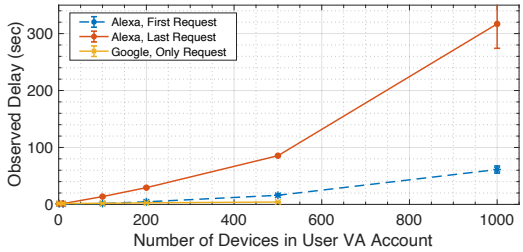

Figure 5: Delay between a voice command and request to control different numbers of devices.

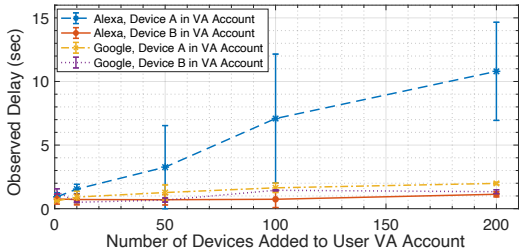

Figure 6: Delay of a voice command given to Device A with varying # of devices and device types in the user VA account.

Selecting device names is more challenging since users can name their devices differently. To understand how users name their devices on VA platforms, we studied posts on official and third-party smart home forums. We found that users usually give predictable names to their devices in the form of `[Location] [Device Type]` [11, 12, 35, 69]. For instance, they name the light in the bedroom as `bedroom light` and the one in the kitchen as `kitchen light`. This naming convention allows users to remember the device names and distinguish them while giving commands. Thus, the attacker can select the device names they add to user accounts as different combinations of locations and device types. This would maximize their chances of conducting eavesdropping attacks on Google Home and intercepting attacks on Alexa when the user issues a voice command in the form of `[Cmd] [Device Name]`.

6.2 No Bound Check on Number of Devices

DMC-Xplorer's root cause analysis indicated that both Alexa and Google Home allow a skill to add a large number of devices to a user account. We conducted additional experiments with a varying number of devices (1 to 1K) configured with different names and types to understand the behavior of both VA platforms.

In our first experiment, we add an increasing number of duplicate devices from a skill and issue the voice command `turn on all [device type]`. Figure 5 shows the response time with an increasing number of devices. On Alexa, we measure the delay between when a voice command is issued and when a skill back-end receives the first and last requests. On Google Home, we only show one delay measurement as it sends a single request containing a list of devices that the voice command intends to control.

On Alexa, adding 1K devices causes an average delay of 61.2 secs before the first request received by the skill back-end and a 5-minute delay on the last request. On Google Home, the max number of devices we could test was 500 because, with 1K devices, the skill back-end does not receive a request at all. Yet, on Google Home, the same number of devices yields a lower delay, e.g., adding 500 devices causes an average delay of 4.15 secs. Overall, we observe that delay increases with the number of devices the voice command refers to, and it is not affected by the device type or name.

In our second experiment, we aim to understand if a skill can delay requests to devices added by another skill. Fig. 6 shows the delay between a user issuing the voice command `turn on all Device A` and `skill-1` back-end receiving the request, while `skill-2` adds a different number of `Device A` or `B` to the user account. On Alexa, there is a delay of 10.8±3.86 secs with 200 `Device A` added by `skill-2`, but there is minimal delay when 200 `Device B` are added. On Google Home, there is a smaller increase in the delay on Alexa, up to 1.99 secs with 200 `Device A` and 1.33 secs with 200 `Device B`.

In our last experiment, we analyzed the limit on the number of devices a skill can add. DMC-Xplorer was able to add 3K devices, the maximum amount we tested, to Alexa without any restriction. For Google Home, DMC-Xplorer's request to add 2K devices was accepted, but the request to add 3K resulted in no devices being added.

Our analysis revealed three key observations. First, Alexa has a large delay with an increasing number of devices. Second, Google Home does not send a request to skill back-ends if the voice command controls more than 500 devices in the user account. Lastly, Alexa allows skills to add many devices, whereas Google Home limits it to 2K. Based on these observations, we design two attacks that exploit the *no proper bound check* on Alexa and Google Home.

6.2.1 A3: Delaying/Blocking Voice Commands. We design an attack in which a malicious skill adds 1K duplicate devices to delay voice commands on Alexa and block them on Google Home. This attack can be conducted by following the steps shown in Fig. 7.

❶ The attacker skill adds many devices with the same name or type as another device on the user account.

$$\texttt{Benign Skill} \rightarrow \texttt{door}$$
$$\texttt{Attacker Skill} \rightarrow \texttt{door (x1000)}$$

❷ The user issues a voice command for that specific device type or name (`[Cmd] [Type/Name]`).

$$\texttt{User} \rightarrow \texttt{lock all doors}$$

❸a Benign skill back-end receives a delayed command in Alexa.

$$\texttt{lock-door} \xrightarrow{\text{delay}} \texttt{Benign Skill Back-end}$$
$$\texttt{lock-door} \xrightarrow{\text{delay}} \texttt{Attacker Skill Back-end}$$

❸b Google Home blocks the given voice command's transmission to all skill back-ends.

$$\texttt{lock-door} \not\rightarrow \texttt{Benign Skill Back-end}$$
$$\texttt{lock-door} \not\rightarrow \texttt{Attacker Skill Back-end}$$

Attack Impact. When a user issues a voice command to a device type on Alexa, this attack causes the command to be executed with a long delay (e.g., 61.8 secs delay with 1K duplicate devices). Yet, the delaying attack is not successful when the user issues a voice command to device names. This is because, in such cases, the voice command is sent to the most recently added device, resulting in the interception attack (**A2**). On Google Home, when a user issues a

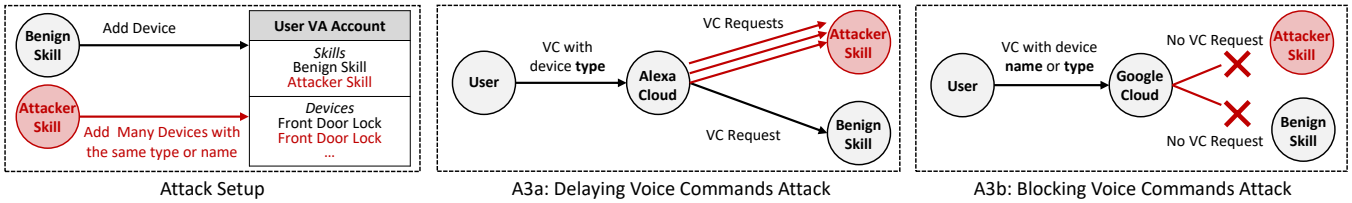

Figure 7: Illustration of delaying and blocking voice commands.

voice command to a device type or name, this attack blocks the request from reaching any skill back-end. This attack may be noticed by users as their commands are delayed or blocked. However, similar to **A2**, the attacker can carefully select which device commands they delay or block to remain stealthy (e.g., cameras, door locks).

6.2.2 A4: Overloading VA's Computational Resources. We design an attack that targets the VA cloud instead of users *without requiring a skill to be added to the marketplace*, as the skill is only added to the attacker's VA account. The attacker adds many devices to the VA account(s) they created and sends voice commands to overload the VA cloud. This attack can be launched through the following steps.

❶ The attacker creates a skill and adds it to the VA account(s) they created.

❷ The attacker repeatedly adds new devices (with the same type or name) to their account(s) through VA APIs.

`Attacker Skill` → `door (x3000)`

❸ The attacker issues a command to all devices.

`Attacker` → `lock all doors`

❹ In turn, the VA device sends thousands of requests to the VA cloud, overloading its servers.

Attack Impact. When the attacker gives a voice command, the Alexa cloud performs significantly higher computation compared to the attacker, as the attacker issues a single API request in milliseconds, whereas the Alexa cloud takes over a minute to issue requests. This attack can easily scale, as the attacker can open multiple VA accounts. With such a large number of devices, this attack can cause significant usage of Alexa computational resources. We note that this attack does not work on Google Home as it does not allow over 2K devices and has a lower delay.

6.3 Testing Performance

Testing Time Overhead. DMC-Xplorer's testing time includes (1) running the testing algorithm, (2) recording events, and (3) validating them against security properties. DMC-Xplorer's VA testing algorithm selects VA parameters, translates them into VDL, executes it, and issues voice commands. The first two steps require time in milliseconds. After issuing each voice command, DMC-Xplorer waits until the request delay threshold (8 secs), which dominates the total testing time. Thus, the testing time increases linearly with the number of voice commands sent. In our experiments, DMC-Xplorer takes, on average, ≈ 130 minutes on both platforms.

Event Log Storage Overhead. DMC-Xplorer records one event per voice command, and the skill back-end records an event for each received request from the VA cloud. Each event is, on average, 16 bytes. For instance, if we add 10 devices (5 lights and 5 cameras) and issue the voice command `turn on the lights`, the skill back-end records 5 events, a total of 80 bytes. DMC-Xplorer records ≈ 2 MB of event traces during our experiments.

7 Limitations and Discussion

7.1 Countermeasures

The attacks (**A1**-**A4**) introduced in Sec. 6 stem from skills being allowed to add duplicate devices in the user account and add many devices without a proper bound check. To conduct these attacks, the only capability required by the attacker's skill is adding devices to the user account, the permission automatically granted to all smart home skills on Alexa and Google Home.

One may consider using an access control mechanism to prevent our attacks. Yet, traditional IoT access control mechanisms [33, 61, 63, 80] cannot prevent the attacks we introduced. The reason is that skills all have the same permissions and any restrictions can prevent benign skills from functioning properly. Thus, we provide three user-centered countermeasures against our attacks.

User Approval for Duplicate Devices. DMC-Xplorer exposed that skills, with default permissions, can add duplicate devices to a user account. This allows an attacker to eavesdrop on voice commands (**A1**) and intercept them (**A2**) with a malicious skill.

Unfortunately, preventing skills from adding duplicate devices by enforcing unique names and types hurt VA usability (e.g., a user may own multiple devices of the same type or install multiple skills to control a device, resulting in these skills adding the device with the same name [55]). Thus, to mitigate these attacks, VA platforms may ask users for confirmation through run-time prompts in mobile VA apps when a duplicate device is added to their VA account. To implement this countermeasure, VA developers can check each request to add a device against existing devices in the user account and send runtime prompts to the user. This interaction prevents an attacker from adding duplicate devices without the user's notice. However, the continuous runtime prompts may create permission fatigue in the users, causing them to allow duplicate devices without checking their validity. Therefore, it is critical to design intuitive runtime prompts and evaluate their usability and effectiveness with large-scale user studies in future work.

Bound Checking on the Number of Devices. Blocking and delaying voice commands (**A3**) and overloading the computational resources of the VA cloud (**A4**) are caused by skills being able to add a high number of devices. To address this, VA platforms can limit the number of devices a skill can add to a lower amount (e.g., 200) and have users opt-in to a higher limit. VA developers can implement this by checking if the number of existing devices in the user's

VA account is less than the limit before adding a new device. This would prevent skills from adding any number of devices to the user account by default but would require users to approve the higher device limit. As a user-centric defense, similar to the user approval for duplicate devices, this countermeasure can also be evaluated in terms of usability and effectiveness through user studies.

VA Account Vetting. The first two countermeasures offer proactive defenses to prevent malicious skills from adding duplicate or a high number of devices. As a reactive measure, VA platforms can vet user accounts to detect suspicious activity. For instance, VA platforms can vet user accounts by validating whether the account includes a large number of devices or any duplicate devices and sending warning messages to the users. To this end, VA platforms can conduct large-scale measurement studies to check whether such symptoms exist in user VA accounts.

Unfortunately, such countermeasures are currently lacking in VA platforms, allowing adversaries to eavesdrop on privacy-sensitive device states, prevent users from controlling their devices, and overload the computational resources of the VA cloud.

7.2 Practical Considerations

Broader Impact of DMC-Xplorer. DMC-Xplorer can be used to explore different design flaws and privacy issues in VA platforms and reproduce existing VA attacks. For example, DMC-Xplorer can issue hidden adversarial voice commands to observe if such commands, combined with a set of devices on a user account, cause VA platforms to send incorrect requests. As another example, DMC-Xplorer's parameter selection can be extended to smart homes where multiple VAs control common devices. For this analysis, events from each VA can be correlated through common devices they control to explore conflicting device states. Our MTL formulas are generalizable for these analyses to identify confidentiality, integrity, and availability violations due to different design flaws.

Porting DMC-Xplorer to other VA Platforms. DMC-Xplorer and VDL can be generalized to other VA platforms with the following steps: (*a*) create skills and their back-ends that respond to requests, (*b*) parse the VDL code using VA APIs to create a VA testing environment from VA parameters, and (*c*) issue voice commands and collect event logs to discover security property violations.

Some VA platforms may structure VA components differently than Alexa and Google Home. For instance, Mycroft [24] allows each skill to keep track of a user's devices and determine which device to send requests to. In such cases, instead of adding devices through VA APIs, DMC-Xplorer can leverage skills to store different devices on the Mycroft device. It can then issue voice commands to the Mycroft device to validate the security properties.

8 Related Work

VA Security and Privacy. There has been a growing interest in the security and privacy of VA platforms [75]. A line of work revealed weaknesses in the skill vetting of the VA platforms [18, 44, 66] and identified the policy-violating skills [30, 47, 62, 78]. Another line of work studied skill squatting attacks, in which an attacker exploits the frequent misinterpretations of the user's voice commands to stealthily mislead the user into installing and using a malicious skill [41, 44, 59, 83, 85]. Prior works also showed attacks against speech recognition [14, 16, 17, 21, 76, 81] and proposed defenses [1, 22, 34, 46, 50, 77, 82, 84] against them.

These works differ from DMC-Xplorer as they focus on skill management and speech recognition vulnerabilities, and therefore, they (1) do not create VA environments with skills, back-ends, devices, and a VA device, and (2) formalize and validate security properties. In contrast, DMC-Xplorer discovers design flaws in the DMC of VA platforms using their APIs by creating VA environments and issuing real voice commands for testing. The created VA environment with each VA component enables DMC-Xplorer to discover design flaws in the DMC of VA platforms that can only be revealed when multiple devices from multiple skills are added to user accounts.

Security of IoT Component Interactions. Prior work explores the security of the interactions between IoT components [25, 37, 51, 53, 58, 86]. For instance, recent work [86] showed that an attacker could create a phantom device to acquire privacy-sensitive information and intercept commands, yielding similar consequences to DMC-Xplorer's eavesdropping and intercepting attacks.

However, these works do not consider the interactions between IoT devices and VA components. In contrast, DMC-Xplorer discovers design flaws in the DMC of VA platforms by exploring the interactions between the VA cloud, skills, and skill back-ends. We show that an attacker can leverage *malicious skills* (rather than phantom devices) to exploit such design flaws and conduct various attacks.

Fuzzing. Traditional software fuzzers have been used to expose memory corruption vulnerabilities [15, 32, 54, 65, 79]. Black-box fuzzers and testing tools have also been proposed to link trigger-action platform rules [72] and find vulnerabilities in IoT device firmware [23]. Instead, DMC-Xplorer identifies DMC design flaws in VA platforms through VDL, new formal security properties, and a parameter mutation algorithm. Recent fuzzers identify semantic bugs with oracles that define the correct operation of robotic vehicles [38, 39] and autonomous driving software [45, 60, 64, 70] to discover vulnerabilities. Yet, they cannot be extended for DMC design flaws of VA platforms because, instead of program inputs, we explore different VA parameters and leverage VA security properties as opposed to code-level (quantitative) feedback for testing.

9 Conclusions

We introduce DMC-Xplorer, a VA operation testing framework that discovers design flaws in the DMC of VA platforms. DMC-Xplorer creates VA environments with a set of devices that have different names, types, and skills through a new domain-specific language VDL. It then sends voice commands to the devices using VA developer APIs. After sending each voice command, DMC-Xplorer logs events, validates them against a set of security properties formally represented with MTL, and conducts intervention-based root cause analysis to determine the VA parameters causing property violations. We use DMC-Xplorer on Alexa and Google Home and discover design flaws in their DMC that a skill with default permissions can exploit. We design four attacks exploiting the design flaws in which a malicious skill can eavesdrop on privacy-sensitive device states, intercept, block, or delay voice commands, and overload the VA's computational resources. Lastly, we propose proactive and reactive countermeasures to mitigate these attacks.

Acknowledgments

We thank our revision editor and the anonymous reviewers for their comments and suggestions. This work has been partially supported by the National Science Foundation (NSF) under grants CNS-2144645 and CNS-2145744. The views expressed are those of the authors only.

References

[1] Shimaa Ahmed, Ilia Shumailov, Nicolas Papernot, and Kassem Fawaz. 2022. Towards more robust keyword spotting for voice assistants. In *USENIX Security*.
[2] Alexa Developer Documentation 2024. https://developer.amazon.com/en-US/docs/alexa/documentation-home.html. [Online; accessed 20-April-2024].
[3] Alexa Response Time 2024. https://developer.amazon.com/en-US/docs/alexa/device-apis/alexa-response.html. [Online; accessed 20-April-2024].
[4] Alexa SpeechRecognizer 2.3 2024. https://developer.amazon.com/en-US/docs/alexa/alexa-voice-service/speechrecognizer.html. [Online; accessed 10-April-2024].
[5] Amazon Alexa 2024. https://alexa.amazon.com/. [Online; accessed 22-April-2024].
[6] Amazon Relational Database Service (RDS) 2024. https://aws.amazon.com/rds/. [Online; accessed 10-April-2024].
[7] Amazon Vulnerability Research Program 2024. https://hackerone.com/amazonvrp?type=team. [Online; accessed 25-April-2024].
[8] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. 2011. S-taliro: A tool for temporal logic falsification for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.
[9] Authorize an AVS Device Through Code-Based Linking 2024. https://developer.amazon.com/en-US/docs/alexa/alexa-voice-service/authorize-cbl.html. [Online; accessed 10-April-2024].
[10] AWS Lambda 2024. https://aws.amazon.com/lambda/. [Online; accessed 10-April-2024].
[11] Best Practices for Naming Devices 2019. https://en.community.sonos.com/amazon-alexa-and-sonos-229102/best-practices-for-naming-of-devices-6819283. [Online; accessed 10-May-2024].
[12] Best Way to Name Devices 2018. https://community.smartthings.com/t/best-way-to-name-devices-in-regards-to-alexa-2018/138257. [Online; accessed 10-May-2024].
[13] Build your skill 2024. https://developer.amazon.com/en-US/docs/alexa/build/build-your-skill-overview.html. [Online; accessed 10-April-2024].
[14] Nicholas Carlini, Pratyush Mishra, Tavish Vaidya, Yuankai Zhang, Micah Sherr, Clay Shields, David Wagner, and Wenchao Zhou. 2016. Hidden voice commands. In *USENIX Security*.
[15] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*.
[16] G. Chen, S. Chen, L. Fan, X. Du, Z. Zhao, F. Song, and Y. Liu. 2021. Who is Real Bob? Adversarial Attacks on Speaker Recognition Systems. In *IEEE Symposium on Security and Privacy (S&P)*.
[17] Guangke Chen, Yedi Zhang, Zhe Zhao, and Fu Song. 2023. QFA2SR: Query-Free Adversarial Transfer Attacks to Speaker Recognition Systems. In *USENIX Security*.
[18] Long Cheng, Christin Wilson, Song Liao, Jeffrey Young, Daniel Dong, and Hongxin Hu. 2020. Dangerous Skills Got Certified: Measuring the Trustworthiness of Skill Certification in Voice Personal Assistant Platforms. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[19] Connect Smart Home Devices in the Google Home App 2024. https://support.google.com/googlenest/answer/9159862. [Online; accessed 10-April-2024].
[20] Connect Smart Home Devices to Alexa 2024. https://www.amazon.com/gp/help/customer/display.html?nodeId=G3RKPNRKF33ECTW7. [Online; accessed 10-April-2024].
[21] Sergio Esposito, Daniele Sgandurra, and Giampaolo Bella. 2022. Alexa versus Alexa: Controlling Smart Speakers by Self-Issuing Voice Commands. In *ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*.
[22] Huan Feng, Kassem Fawaz, and Kang G. Shin. 2017. Continuous Authentication for Voice Assistants. In *International Conference on Mobile Computing and Networking (MobiCom)*.
[23] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. 2021. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[24] Kris Gesling. 2023. Mycroft Development Setup. https://mycroft-ai.gitbook.io/docs/skill-development/introduction. [Online; accessed 15-April-2024].
[25] Furkan Goksel, Muslum Ozgur Ozmen, Michael Reeves, Basavesh Shivakumar, and Z Berkay Celik. 2021. On the safety implications of misordered events and commands in IoT systems. In *IEEE Security and Privacy Workshops (SPW)*.
[26] Google Firebase 2024. https://firebase.google.com/. [Online; accessed 10-April-2024].
[27] Google Home 2024. https://assistant.google.com/. [Online; accessed 12-April-2024].
[28] Google Home Developer Documentation 2024. https://developers.home.google.com/docs. [Online; accessed 20-April-2024].
[29] Google VRP 2024. https://www.google.com/about/appsecurity/reward-program/. [Online; accessed 10-April-2024].
[30] Zhixiu Guo, Zijin Lin, Pan Li, and Kai Chen. 2020. SkillExplorer: Understanding the Behavior of Skills in Large Scale. In *USENIX Security*.
[31] York Hagmayer, Steven A Sloman, David A Lagnado, and Michael R Waldmann. 2007. Causal reasoning through intervention. *Causal learning: Psychology, philosophy, and computation*.
[32] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Security*.
[33] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. 2018. Rethinking Access Control and Authentication for the Home Internet of Things (IoT).. In *USENIX Security*.
[34] Ashish Hooda, Matthew Wallace, Kushal Jhunjhunwalla, Earlence Fernandes, and Kassem Fawaz. 2022. SkillFence: A Systems Approach to Practically Mitigating Voice-Based Confusion Attacks. *ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)* (2022).
[35] How To Name Your Smart Home Devices for Better Voice Control 2019. https://www.howtogeek.com/404609/how-to-name-your-smarthome-devices-for-better-voice-control/. [Online; accessed 10-May-2024].
[36] Intent Fulfillment 2024. https://developers.google.com/assistant/smarthome/develop/process-intents. [Online; accessed 10-April-2024].
[37] Yan Jia, Luyi Xing, Yuhang Mao, Dongfang Zhao, XiaoFeng Wang, Shangru Zhao, and Yuqing Zhang. 2020. Burglars' IoT paradise: Understanding and mitigating security risks of general messaging protocols on IoT clouds. In *IEEE Symposium on Security and Privacy (S&P)*.
[38] Hyungsub Kim, Muslum Ozgur Ozmen, Antonio Bianchi, Z Berkay Celik, and Dongyan Xu. 2021. PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles. In *Network and Distributed System Security Symposium (NDSS)*.
[39] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. 2019. RVFUZZER: finding input validation bugs in robotic vehicles through control-guided testing. In *USENIX Security*.
[40] Ron Koymans. 1990. Specifying real-time properties with metric temporal logic. In *Real-time systems*.
[41] Deepak Kumar, Riccardo Paccagnella, Paul Murley, Eric Hennenfent, Joshua Mason, Adam Bates, and Michael Bailey. 2018. Skill squatting attacks on Amazon Alexa. In *USENIX Security*.
[42] Launch your smart home action 2024. https://developers.google.com/assistant/smarthome/develop/launching. [Online; accessed 10-April-2024].
[43] Tu Le, Dongfang Zhao, Zihao Wang, XiaoFeng Wang, and Yuan Tian. 2024. Alexa, is the skill always safe? Uncover Lenient Skill Vetting Process and Protect User Privacy at Run Time. In *International Conference on Software Engineering (ICSE)*.
[44] Christopher Lentzsch, Sheel Jayesh Shah, Benjamin Andow, Martin Degeling, Anupam Das, and William Enck. 2021. Hey Alexa, is this Skill Safe?: Taking a Closer Look at the Alexa Skill Ecosystem. In *Network and Distributed System Security Symposium (NDSS)*.
[45] Guanpeng Li, Yiran Li, Saurabh Jha, Timothy Tsai, Michael Sullivan, Siva Kumar Sastry Hari, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2020. AV-FUZZER: Finding safety violations in autonomous driving systems. In *International Symposium on Software Reliability Engineering (ISSRE)*.
[46] Xinfeng Li, Xiaoyu Ji, Chen Yan, Chaohao Li, Yichen Li, Zhenning Zhang, and Wenyuan Xu. 2023. Learning Normality is Enough: A Software-based Mitigation against Inaudible Voice Attacks. In *USENIX Security*.
[47] Song Liao, Long Cheng, Haipeng Cai, Linke Guo, and Hongxin Hu. 2023. SkillScanner: Detecting Policy-Violating Voice Applications Through Static Analysis at the Development Phase. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[48] Login with Amazon for Websites Overview 2024. https://developer.amazon.com/docs/login-with-amazon/web-docs.html. [Online; accessed 20-April-2024].
[49] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. In *IEEE Transactions on Software Engineering*.
[50] Yan Meng, Jiachun Li, Matthew Pillari, Arjun Deopujari, Liam Brennan, Hafsah Shamsie, Haojin Zhu, and Yuan Tian. 2022. Your Microphone Array Retains Your Identity: A Robust Voice Liveness Detection System for Smart Speakers. In *USENIX Security*.
[51] TJ OConnor, Reham Mohamed, Markus Miettinen, William Enck, Bradley Reaves, and Ahmad-Reza Sadeghi. 2019. HomeSnitch: behavior transparency and control for smart home IoT devices. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*.

[52] Muslum Ozgur Ozmen, Xuansong Li, Andrew Chu, Z Berkay Celik, Bardh Hoxha, and Xiangyu Zhang. 2022. Discovering IoT physical channel vulnerabilities. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[53] Muslum Ozgur Ozmen, Ruoyu Song, Habiba Farrukh, and Z Berkay Celik. 2023. Evasion attacks and defenses on smart home physical event verification. In *Network and Distributed System Security Symposium (NDSS)*.

[54] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy (S&P)*.

[55] Samantha Reig, Elizabeth Jeanne Carter, Lynn Kirabo, Terrence Fong, Aaron Steinfeld, and Jodi Forlizzi. 2021. Smart home agents and devices of today and tomorrow: Surveying use and desires. In *International Conference on Human-Agent Interaction*.

[56] Remove Google Nest Devices from the Google Home App 2024. https://support.google.com/googlenest/answer/9691327. [Online; accessed 20-April-2024].

[57] Remove Smart Home Devices from Alexa 2024. https://www.amazon.com/gp/help/customer/display.html?nodeId=GH7J6YW8GMWE7BZY. [Online; accessed 20-April-2024].

[58] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. 2017. IoT Goes Nuclear: Creating a Zigbee Chain Reaction. In *IEEE Symposium on Security and Privacy (S&P)*.

[59] Aafaq Sabir, Evan Lafontaine, and Anupam Das. 2022. Hey Alexa, Who Am I Talking to?: Analyzing Users' Perception and Awareness Regarding Third-Party Alexa Skills. In *CHI Conference on Human Factors in Computing Systems*.

[60] Ivan F Salgado, Nicanor Quijano, Daniel J Fremont, and Alvaro A Cardenas. 2022. Fuzzing Malicious Driving Behavior to find Vulnerabilities in Collision Avoidance Systems. In *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*.

[61] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. 2018. Situational access control in the internet of things. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[62] Faysal Hossain Shezan, Hang Hu, Gang Wang, and Yuan Tian. 2020. VerHealth: Vetting Medical Voice Applications through Policy Enforcement. In *ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)*.

[63] Amit Kumar Sikder, Leonardo Babun, Z Berkay Celik, Abbas Acar, Hidayet Aksu, Patrick McDaniel, Engin Kirda, and A Selcuk Uluagac. 2020. Kratos: Multi-user multi-device-aware access control system for the smart home. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*.

[64] Ruoyu Song, Muslum Ozgur Ozmen, Hyungsub Kim, Raymond Muller, Z Berkay Celik, and Antonio Bianchi. 2023. Discovering Adversarial Driving Maneuvers against Autonomous Vehicles. In *USENIX Security*.

[65] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *Network and Distributed System Security Symposium (NDSS)*.

[66] Dan Su, Jiqiang Liu, Sencun Zhu, Xiaoyang Wang, and Wei Wang. 2020. Are you home alone?" "Yes" Disclosing Security and Privacy Vulnerabilities in Alexa Skills. In *arXiv preprint arXiv:2010.10788*.

[67] Dan Su, Jiqiang Liu, Sencun Zhu, Xiaoyang Wang, and Wei Wang. 2022. Alexa Skills: Security Vulnerabilities and Countermeasures. In *IEEE Conference on Communications and Network Security (CNS)*.

[68] Quinn Thibeault, Jacob Anderson, Aniruddh Chandratre, Giulia Pedrielli, and Georgios Fainekos. 2021. Psy-taliro: A python toolbox for search-based test generation for cyber-physical systems. In *Formal Methods for Industrial Critical Systems (FMICS)*.

[69] Voice Control - How to Name Your Smart Devices 2023. https://www.smarthome.com.au/voice-control-how-to-name-your-smart-devices/. [Online; accessed 10-May-2024].

[70] Ziwen Wan, Junjie Shen, Jalen Chuang, Xin Xia, Joshua Garcia, Jiaqi Ma, and Qi Alfred Chen. 2022. Too Afraid to Drive: Systematic Discovery of Semantic DoS Vulnerability in Autonomous Driving Planning under Physical-World Attacks. In *Network and Distributed System Security Symposium (NDSS)*.

[71] Dawei Wang, Kai Chen, and Wei Wang. 2021. Demystifying the vetting process of voice-controlled skills on markets. In *ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)*.

[72] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A Gunter. 2019. Charting the attack surface of trigger-action IoT platforms. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[73] What is the Alexa Skills Kit 2024. https://developer.amazon.com/en-US/docs/alexa/ask-overviews/what-is-the-alexa-skills-kit.html. [Online; accessed 20-April-2024].

[74] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[75] Chen Yan, Xiaoyu Ji, Kai Wang, Qinhong Jiang, Zizhi Jin, and Wenyuan Xu. 2022. A Survey on Voice Assistant Security: Attacks and Countermeasures. In *ACM Computing Surveys*.

[76] Qiben Yan, Kehai Liu, Qin Zhou, Hanqing Guo, and Ning Zhang. 2020. SurfingAttack: Interactive Hidden Attack on Voice Assistants Using Ultrasonic Guided Wave. In *Network and Distributed System Security Symposium (NDSS)*.

[77] Qiang Yang, Kaiyan Cui, and Yuanqing Zheng. 2023. VoShield: Voice Liveness Detection with Sound Field Dynamics. In *IEEE INFOCOM*.

[78] Jeffrey Young, Song Liao, Long Cheng, Hongxin Hu, and Huixing Deng. 2022. SkillDetective: Automated Policy-Violation Detection of Voice Assistant Applications in the Wild. In *USENIX Security*.

[79] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security*.

[80] Eric Zeng and Franziska Roesner. 2019. Understanding and Improving Security and Privacy in Multi-User Smart Homes: A Design Exploration and In-Home User Study.. In *USENIX Security*.

[81] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. 2017. Dolphinattack: Inaudible voice commands. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[82] Linghan Zhang, Sheng Tan, Zi Wang, Yili Ren, Zhi Wang, and Jie Yang. 2020. VibLive: A Continuous Liveness Detection for Secure Voice User Interface in IoT Environment. In *Annual Computer Security Applications Conference (ACSAC)*.

[83] Nan Zhang, Xianghang Mi, Xuan Feng, XiaoFeng Wang, Yuan Tian, and Feng Qian. 2019. Dangerous skills: Understanding and mitigating security risks of voice-controlled third-party functions on virtual personal assistant systems. In *IEEE Symposium on Security and Privacy (S&P)*.

[84] Shaohu Zhang, Zhouyu Li, and Anupam Das. 2023. VoicePM: A Robust Privacy Measurement on Voice Anonymity. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*.

[85] Yangyong Zhang, Lei Xu, Abner Mendoza, Guangliang Yang, Phakpoom Chinprutthiwong, and Guofei Gu. 2019. Life after speech recognition: Fuzzing semantic misinterpretation for voice assistant applications. In *Network and Distributed System Security Symposium (NDSS)*.

[86] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. 2019. Discovering and understanding the security hazards in the interactions between IoT devices, mobile apps, and clouds on smart home platforms. In *USENIX Security*.