Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2022-IC-002

2022-IC-002

VITAS : Guided Model-based VUI Testing of VPA Apps

Suwan Li, Lei Bu, Guangdong Bai, Zhixiu Guo, Kai Chen, Hanlin Wei

Technical Report 2022

VITAS : Guided Model-based VUI Testing of VPA Apps

Suwan Li
State Key Laboratory for Novel Software Technology, Nanjing University
China
lisuwan@smail.nju.edu.cn

Lei Bu*
State Key Laboratory for Novel Software Technology, Nanjing University
China
bulei@nju.edu.cn

Guangdong Bai
School of ITEE, University of Queensland
Australia
g.bai@uq.edu.au

Zhixiu Guo
Institute of Information Engineering, Chinese Academy of Sciences
China
guozhixiu@iie.ac.cn

Kai Chen
Institute of Information Engineering, Chinese Academy of Sciences
China
chenkai@iie.ac.cn

Hanlin Wei
School of ITEE, University of Queensland
Australia
hanlin.wei@uq.edu.au

ABSTRACT

Virtual personal assistant (VPA) services, e.g. Amazon Alexa and Google Assistant, are becoming increasingly popular recently. Users interact with them through voice-based apps, e.g. Amazon Alexa skills and Google Assistant actions. Unlike the desktop and mobile apps which have visible and intuitive graphical user interface (GUI) to facilitate interaction, VPA apps convey information purely verbally through the voice user interface (VUI), which is known to be limited in its invisibility, single mode and high demand of user attention. This may lead to various problems on the usability and correctness of VPA apps.

In this work, we propose a model-based framework named Vitas to handle VUI testing of VPA apps. Vitas interacts with the app VUI, and during the testing process, it retrieves semantic information from voice feedbacks by natural language processing. It incrementally constructs the finite state machine (FSM) model of the app with a weighted exploration strategy guided by key factors such as the coverage of app functionality. We conduct a large-scale testing on 41,581 VPA apps (i.e., skills) of Amazon Alexa, the most popular VPA service, and find that 51.29% of them have weaknesses. They largely suffer from problems such as unexpected exit/start, privacy violation and so on. Our work reveals the immaturity of the VUI designs and implementations in VPA apps, and sheds light on the improvement of several crucial aspects of VPA apps.

ACM Reference Format:
Suwan Li, Lei Bu, Guangdong Bai, Zhixiu Guo, Kai Chen, and Hanlin Wei. 2022. VITAS : Guided Model-based VUI Testing of VPA Apps. In *37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556957

*Corresponding author.

ASE '22, October 10–14, 2022, Oakland Center, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556957

1 INTRODUCTION

The occurrence of smart devices is producing a revolution in the traditional graphics-based human computer interaction. They extensively embed voice-based user interaction to serve users in a more convenient way than screen tapping or mouse movement. On these devices, various virtual personal assistant (VPA) services such as the Amazon Alexa [1] and Google Assistant [2] have become easily accessible. At the same pace, an ecosystem centered around the VPA services is formulated. The third-party developers are enabled to create apps and release them through app stores. The end users can seamlessly invoke the released apps from their smart devices through verbal inputs. This model greatly stimulates the boom of VPA apps. The app stores continue flourishing, and various apps providing a wide range of functionalities from setting alarms, playing music, gaming, to controlling smart home devices become available. Taking Amazon Alexa as an example, the number of its apps (also called *skills*) has risen from 130 to over 100,000 within three years by September 2019 [6].

Despite the proliferation of VPA apps, the intrinsic *invisibility feature* of the voice-based user interaction may raise various concerns on them. On their *quality*, given that conversation is the mere way to interact with them, they should provide sufficient feedback for the users to understand their status. On their *security* property, since the apps strongly rely on the speech recognition algorithms, an attacker can impersonate an app with another that has a sound-similar invocation name (e.g., "full moon" v.s. "four moon"), as revealed by recent studies [5, 26]. On their *privacy* property, malicious apps may query personal information irrelevant to their functionalities, or eavesdrop the user's conversations stealthily [20, 34].

To tackle these concerns, some efforts have been committed. Amazon puts in place a certification process that validates and tests apps submitted to its app store. The details of this process is not revealed though, and recent studies show that apps with policy violations can still pass it [14]. Chatbot is a suitable technique for VPA app testing [21], given that the app to test is a blackbox from the perspective of the tester. It only relies on the current output of the app, and attempts to explore app behaviors by generating inputs with natural language processing (NLP) techniques. However,

without the context awareness, the generated test cases are often either redundant or irrelevant to trigger bugs.

In this work, we seek a systematic approach to test VPA apps and find their potential problems related to either quality, security or privacy. We focus on automating the *voice user interface (VUI) testing*, as much voice-based interaction has to be handled in the app testing process. A potential technique is the model-based testing, given that the app is usually designed with an explicit or implicit model [8]. Model-based testing maintains an abstract model to represent the internal control flow and behaviors of the system, and generates targeted test cases based on the current state that the system is in. The generated test cases can drive the execution towards preferable testing goals, e.g., a high coverage or locations deep in the system.

Despite the maturity of model-based testing and its demonstrated power in various domains, such as binary testing [29], real-time systems testing [28] and graphical user interface (GUI) testing [25, 31], new challenges arising from applying it to VUI testing are at least threefold. First, *how to construct a model to represent the behaviors of the app?* In traditional domains, clear and informative signals are available for the tester to determine the current state, such as the program counter and valuation of state variables in program testing, and the visible GUI elements in GUI testing. In contrast, a VUI tester has to fully rely on the audio outputs to determine the current state of the app. Second, *how to generate relevant test cases?* In traditional domains, the tester is often offered intuitive options to proceed, for example, the clickable buttons in the GUI testing. In VUI testing, the tester has to interpret the audio response and understand the context of the current state, in order to generate relevant inputs. Third, *how to handle the notorious path-explosion problem?* For a complex VPA app, there may be multiple possible inputs in each of its states, and thus to enumerate all of them is impractical. Without a proper reduction or guidance, the tester may easily sink into the endless exploration.

To address these challenges, we propose a *guided model-based testing* framework named Vitas. Vitas builds an FSM model to represent the behaviors of the VPA app under testing, and takes a strategy of *on-the-fly model construction*. It starts with a small number of inputs learnt from the app document using NLP techniques. During the app execution, it constructs states from the audio outputs, so as to expand the FSM model of the app. It also expands its input set by generating new inputs based on the responses of the app, which in turns are fed to the app to discover new states. The inputs Vitas gives to the app include not only functionality-level inputs (e.g., `start the game`, and `play some music`, but also system-level inputs (e.g., `pause`, `stop`, and `resume`). We design a weighted exploration strategy to guide the state exploration. The weight takes into consideration the input type, covering new functionalities / states and invocation times. The weighted exploration leads to a high coverage of the state space in an efficient manner.

To characterize the quality of existing VPA apps, we conduct a comprehensive study with Vitas on the VPA apps of Amazon Alexa, the most popular VPA service. We test 41,581 Amazon skills, and find that 51.29% of them have various weaknesses in their implementations (e.g., unexpected exit/start), or even violate user privacy. We also evaluate the Vitas's performance in terms of accuracy, coverage and efficiency, and reveal that it significantly outperforms the techniques that are based on random fuzzing and chatbot.

Contributions. The contributions of this work are summarized as follows.

- **Automatic VUI testing techniques.** We propose automatic techniques to handle the voice-based user interaction when testing VPA apps. Our approach works with only publicly available information of the app under testing, by taking the app as a blackbox.
- **Guided exploration.** We propose Vitas, which uses guided model-based testing to explore the state space of the app. Vitas is designed to feed the app with inputs that are potential to lead to large state coverage.
- **Practical results.** We apply our framework to a large-scaled testing of 41,581 Amazon skills. The experiments demonstrate its accuracy and efficiency performance. Our work manages to identify weaknesses from real-world VPA apps.

2 BACKGROUND

VPA services, e.g., Amazon Alexa and Google Assistant, are a class of software that provides services to users by voice-based interactions. The functionality of a VPA service can be greatly extended by VPA apps, e.g., skills to Amazon Alexa and actions to Google Assistant, which are developed by third-party developers.

The AI-enhanced speech recognition capability of the VPA service enables a convenient voice-based interaction. Figure 1 uses an Alexa skill named "90's Movie Trivia" to demonstrate the interaction between the user and the VPA app. The user says the invocation name of the skill to Alexa, who recognizes the skill being invoked and initiates the skill. If the skill is new to her, she may say the second command "Help" to understand what services the app provides. Based on the feedback provided by the app, the user can keep interacting with the skill through voice inputs to use its services. We can see that each input from the user can trigger the Alexa skill to give an output respectively. Therefore, we choose to model this interactive behavior using classical finite state machine (FSM), which will be illustrated in Section 3.1.

Below, we introduce the terms that are frequently used in the context of VPA apps.

- **Execution.** The process from initiating the app till the app fully stops is called a complete execution.
- **Inputs.** User's inputs to the VPA apps. The inputs are purely voice-based natural language, and are converted to texts through Alexa's voice recognition.
- **Outputs.** The outputs are the VPA app's outputs to the user. A decent VUI design should provide the user with sufficient context to come up with effective inputs.
- **Questions.** The question is a special type of output and it often expects a different type of input. The questions given by the VPA app can be roughly categorized into five types, as summarized in a recent study [21]. They are shown in Table 1.
- **Interaction Round.** During the execution, one conversation in which the user gives an input and the app gives an output is called an interaction round.

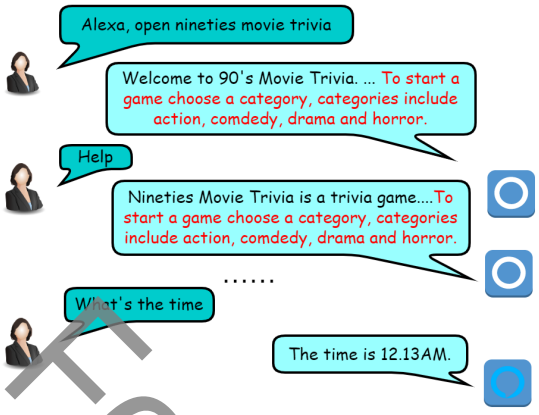


Figure 1: The communication between the user and the "90's Movie Trivia" skill.

Table 1: the five types of questions in VPA apps' outputs

Question type	Description
Yes-no questions	A yes-no question usually begins with an auxiliary verb followed by a subject and the expected inputs are "Yes" or "No".
Selection questions	A selection question offers a series of options to choose. The expected inputs are composed of these options.
Instruction questions	The verbs like "say" are the sign of instruction questions. The potential inputs appear after these verbs.
Wh questions	Wh questions begin with wh words. The keyword (usually the wh words or nouns) determines the range of possible inputs. A knowledge database for some keywords is built to get expected inputs to answer Wh questions.
Mixed questions	A mixed question is a mixed form of the above four questions. Its answers can be selected from the answers of above four questions.

3 APP MODEL AND APPROACH OVERVIEW

In this section, we define how the app behaviors are modeled by Vitas. Then we introduce the overall framework of Vitas.

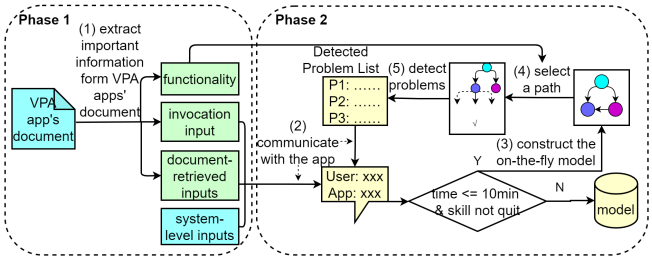


Figure 2: The Vitas framework.

3.1 VPA app behavior model

Vitas uses the finite state machine (FSM) to represent the behaviors of the app. An FSM model is defined as a five-tuple $M = (Q, \Sigma, \delta, s_0, F)$.

- Q represents the set of states.
- Σ represents the set of input events.
- F is the set of final states, and satisfies $F \subseteq Q$.
- s_0 is the initial state and satisfies $s_0 \in Q$.
- $\delta : Q \times \Sigma \rightarrow Q$ represents a transition function. The input event e that triggers the transition from the state s_0 to the states s_1 is represented as $\delta(s_0, e) = s_1$.

Vitas maps the app behavior to an FSM model based on the interaction rounds between the user and the app. In particular, it recognizes the app's outputs as states and users' inputs as input events. An interaction round, which triggers the app to output according to an input, can be modeled as a transition.

This straightforward strategy may lead to a large number of redundant states though. In our running example, the output "*To start a game choose a category, ...*" appears in two different interaction rounds, one with welcome messages and the other with some brief descriptions. Two different states would be created as these two rounds have different outputs. Nonetheless, both occurrences mean the same state from the user's perspective. To alleviate this problem, we split the outputs into independent sentences, and map each sentence to a state. In this way, the same sentence appearing in diverse outputs would reuse the same state.

This simplification leads to another problem. More than one state is created for an output if the output contains several sentences. To cope with this, Vitas selects only one sentence to create the state, so that each (unseen) output leads to only one state regardless of the length of the output. Our strategy to select the sentence will be discussed in Section 5.2.

3.2 Approach Overview

Figure 2 shows the overall framework of Vitas. It is composed of five main steps in two phases: (1) extract important information from VPA apps' document, (2) communicate with the app, (3) construct the on-the-fly model, (4) select a path, and (5) detect problems.

Phase 1: document processing. This phase aims to extract information to facilitate the testing (in terms of both user input generation and guidance) from the app's document. The app document usually includes its name, invocation name, and description. The invocation name is used for launching the app, which is mandated to be unique. Otherwise, users may open an unexpected app. Taking Alexa skills as an example, the invocation input is "Alexa, open + <invocation name>". The app's description is usually written by its developer to give users an introduction on its features, functionalities and usage. Therefore, in this phase, Vitas processes these fields to retrieve inputs and functionalities of the app. The app document processing is detailed in Section 4.

Phase 2: model-based testing and problem detection. Vitas constructs the model on the fly during the testing process. Initially, it uses the invocation input to launch the app. When receiving outputs from the app, Vitas analyzes them to extract new states and transitions, and expand the app's model. Then, it selects an input to feed into the app, based on dynamically-adjustable weights of all possible inputs at that state. The model construction, input generation and guided exploration are introduced in Section 5. During the exploration process, Vitas detects five types of problems, based on pre-defined rules, as presented in Section 6.

90's ***Movie Trivia*** is a ***game*** that will put your movie knowledge to the test. The game is simple to play, you pick a category and we ***put*** to together a list of 10 movies. Each trivia question is plot about different the movie, guess the movie title. ***Categories*** are comedy, drama, action and horror. Season 1 of 90's Movie Trivia includes 100 trivia questions, 25 per category. To start just say "Alexa open nineties movie trivia". To choose a category say "comedy" or "category comedy".

Figure 3: Document of our running example in Figure 1.

4 APP DOCUMENT PROCESSING

As the VPA app is a black-box to Vitas, we leverage its available documents to retrieve information to facilitate the testing.

4.1 Retrieving inputs

The inputs to the VPA app are in the form of verbal phrases. In light of this, we extract phrases with subjects related to users from the document and treat them as potential inputs in our testing.

One direct source of inputs are the phrases with quotation marks, as developers usually use them to mark examples of valid inputs. In addition, we resort to NLP techniques to extract inputs. We use SpaCy [7] to build a dependency-based parse tree for every sentence in the document, and from that Vitas can extract the part of speech (pos) and the dependency (dep) of each word. Then, we identify the verbal phrases in special forms, including *subject + verb* and *subject + verb + noun(s)*, and remove decorative phrases like adjectives. Finally, we check whether the subjects of these verb phrases are related to users (like "User" or "You"). If so, we add these verb phrases to the Vitas's input set. We note that not all inputs extracted in this way are valid, but they only take a few percentage of the input set. For example, in our running example, the invalid inputs only account for 5% of the input set.Meanwhile, we do not attempt to remove them in Vitas, as invalid inputs during testing may trigger abnormal behavior of the system if the system is not well designed. The generation of other types of inputs will be discussed in Section 5.2.

Figure 3 lists the document of our running example. From it, Vitas can extract four phrases (underlined) as inputs, and all except the first one ("*pick a category*") are confirmed as valid inputs in the later execution.

4.2 Extracting app functionalities

Understanding the functionalities of the app could guide the state exploration towards a direction of higher functionality coverage. Therefore, we use the TF-IDF [24] method to extract the list of functionalities from the document. TF-IDF is a statistical method to compute the importance of words in the document based on a corpus. In most cases, the keywords of the document can summarize the functionalities well. We thus obtain the importance of each word in the document and treat those words with top TF-IDF rate as the app's functionalities. If the frequency of a word is low in the corpus but high in the document, the TF-IDF value of the word will be large, which means the word is of more importance to the document.

To calculate a word's TF-IDF value, we have built a corpus composed of the documents of 20 representative apps from 20 different categories. Each of them is the most popular VPA app in their categories. In our method, the top five words with the highest TF-IDF values in an app's document are selected to represent the functionalities of the app. For example, in our running example, the words with top five TF-IDF values are identified (***bolded*** in Figure 3). They are "category", "movie", "trivia", "game" and "put", where the top four words can well summarize the functionalities of this app. These functionalities represented by these phrases are used to guide the state space exploration (detailed soon in Section 5).

Algorithm 1: On-the-fly VUI Testing-based VPA Model Exploration

Input: the outcomes of phase 1: invocation input *invoc*.
Output: VPA's model: $M = (Q, \Sigma, \delta, s_0, F)$

Initialize a weighted FSM $M' = (Q = \{< start >\}, \Sigma = \{invoc\}, \delta = \{\}, s_0 =< start >, F = \{\}, W = \{\})$;
for *times in* 1 *to n* **do**
  $s = s_0$, $cand\_in = \{invoc\}$, $in = invoc$;
  //s: the current state, *cand_in*: candidate input set, *in*: the actual input
  out = **execute**(in);
  repeat
    s = **getSentence**(out);
    if s is an unknown state **then**
      $cand\_in$ = **generateInpt**(s);
      initWeight($cand\_in, W$);
    end
    else
      $cand\_in$ = **getInpt**(s);
    end
    in = **getMaxWeight**($cand\_in, W$);
    out = **execute**(in);
    foreach event $a_0 \in cand\_in$ **do**
      updateWeight(a_0, out, W);
    end
    updateFSM($s, cand\_in, in$);
  until *VPA app exits* || *time limit is met*;
  $F = F \cup out$;
end
$M = M'$.**removeWeight**();
return M;

5 STATE SPACE EXPLORATION

During the testing process, Vitas builds the FSM model of the app on the fly. In this section, we detail our approach for phase 2.

5.1 Exploration Workflow

To explore as many behaviors as possible, Vitas uses a guided exploration strategy. We introduce a weight to each candidate input

in a particular state to indicate its current priority to be invoked. The weight is dynamically updated to guide the exploration. To facilitate this weight management process, we extend the FSM defined in Section 3.1 with the weight, into the weighted FSM represented by a six-tuple $M = (Q, \Sigma, \delta, s_0, F, W)$, where:

- $(Q, \Sigma, \delta, s_0, F)$ is the classical FSM defined in Section 3.1.
- $W : Q \times \Sigma \rightarrow float$ represents the weight of an input event (detailed in Section 5.3).

Using weighted FSM, Algorithm 1 conducts the model exploration as briefed below.

- A weighted FSM M' is initialized and the initial state s_0 is set to $< start >$ (line 1). Vitas uses the invocation input to open the app, and receives *out* as the reply (**execute**, line 5).
- In each round, Vitas selects a sentence to represent the current state (**getSentence**) from the *out* (line 7, detail in Section 5.2). The state is represented by *s*.
- If *s* is an unknown state, Vitas generates candidate inputs (**generateInpt**, line 9, detailed in Section 5.2) for *s*, initializes their weights, and saves the results to *W*. (**initWeight**, line 10, detailed in Section 5.3). If *s* is an existing state, it finds the previously saved candidate inputs for *s* (**getInpt**, line 12-14).
- The input with the highest weight (**getMaxWeight**, line 15) is selected as the actual input *in* and Vitas receives the *out* as the reply (**execute**, line 16).
- Vitas modifies the weight of candidate inputs under the state *s* according to the new *out*, and saves the results to *W* (**updateWeight**, line 17-19, detailed in Section 5.3).
- Then, the model is updated accordingly (**updateFSM**, line 20).
- A complete execution is over when the VPA app exits or the time limit is met (line 21). *F* saves the last outputs of VPA apps (line 22). For the sake of sufficient exploration, we can repeat such complete execution for *n* times in testing. According to our experience in the analysis, *n* is set to 3 by default to achieve a relatively satisfying coverage.
- Finally, the weighted FSM M' is transferred to a normal FSM *M* by removing the last tuple *W* (line 24- 25).

Figure 4 uses part of the execution of the "90's Movie Trivia" skill to demonstrate how our method works. The detail will be elaborated in the remaining of this section.

5.2 Candidate Inputs Generation

Selecting a sentence from the outputs (function **getSentence**). Given the output of the app, Vitas first looks for questions from it. If any question exists, non-wh questions have higher priorities to be chosen, considering the expected inputs to these questions are more restrict than those to wh-questions and more likely to be accepted. For example, in Figure 4, a non-wh question (selection question) is located, and Vitas continues its exploration from it. If there is no question in the output, Vitas selects the last sentence as the current state.

Generating candidate inputs (function **generateInpt**). After the question is determined, Vitas generates an input to interact with the app. Besides the document-retrieved inputs discussed in Section 4.1, we have identified another four input types. Below we sum up these five types of inputs, and show how these inputs can be generated by Vitas.

- **Invocation input.** The invocation input is specified by the invocation name in the app's document. For example, in Alexa skills, the invocation input is in the form of "Alexa, open <invocation name>". Such inputs can only be used to initiate an app, and cannot be used during the interaction.
- **System-level inputs.** The system-level inputs are supported by the VPA service, and independent of a specific app. Most system-level inputs are related to system behaviors such as the suspension, restart, and shutdown of the apps, like ["Pause", "Resume", "Restart", "Reboot", "Stop", "Exit", "Cancel"].
- **Document-retrieved inputs.** The document-retrieved inputs are extracted from the app's document (detailed in Section 4.1).
- **Embedded-help inputs.** The embedded-help inputs are embedded by the app developer as the introduction to the usage of the app's functionalities. They typically follow the format of instruction questions (see Table 1), e.g., "*You can say Stop to stop*". Vitas obtains them by giving the input "Help" to the app, and then derives embedded-help inputs specific to this app using the chat bot solution of Skillexplorer [21].
- **Context-related inputs.** The context-related inputs are created dynamically during the interaction with the app. They are composed of expected inputs that are generated according to the VPA app's questions. We can use off-the-shelf chat bot-style solutions, e.g., Skillexplorer [21], to understand the question and generate corresponding inputs.

5.3 Input Weight Management

Mutation based on weighted stochastic and probabilistic finite state machine [22] has been widely applied in the testing and exploration of complex software, especially in testing of the GUI and web pages [31]. In existing studies, testers have clear view of the current state of the system under testing and a deterministic set of operations to take.

In our problem domain of VUI testing, the interaction is conducted purely verbally, making it non-trivial to maintain state space and determine suitable inputs. To address this challenge, we design our weighted FSM based mutation method to guide the exploration of VUI apps. As we mentioned in Section 5.1, the weight of each candidate input in a particular state indicates its current priority to be invoked. In this section, we discuss the weight management strategies.

Initializing weights for candidate inputs (function **initWeight**). Once candidate inputs are generated, Vitas needs to select one to input to the app. To avoid repeatedly testing the same paths, a *weight* is introduced for each input to indicate its priority. We adopt different strategies to initialize weights based on the types of inputs.

- **System-level inputs.** Giving the app a system-level input at the beginning or halfway during the testing will affect the normal testing process. Therefore, a suitable invocation time for a system-level input is near the end of the testing. Thus, their weights should be very low at initial, and we set them to be 0.1.
- **Document-retrieved inputs.** If the document-retrieved inputs is quoted in the document, Vitas sets a high initial weight of them as 0.8 as they are usually useful inputs. Otherwise, the input comes from verbal phrases in the document. The initial weight of these verbal phrases depend on the importance of verbs.

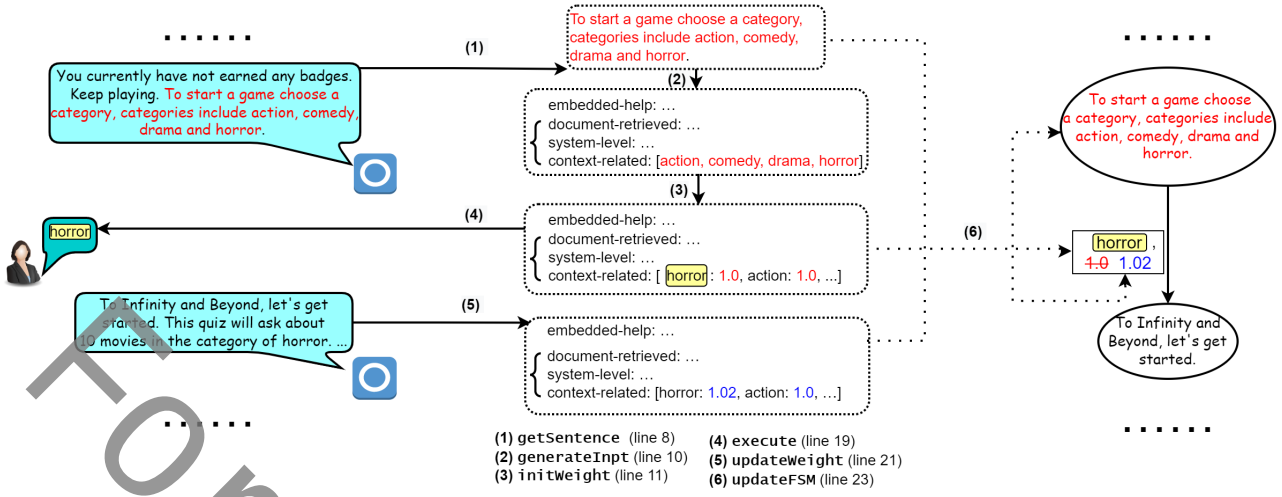


Figure 4: An example that demonstrates the algorithm using part of the execution of the "90's Movie Trivia" skill. (1) Vitas chooses the sentence ("To start a game ...") from the current outputs because it is a selection question. (2) Candidate inputs are generated for the sentence selected in (1). The generation of document-retrieved inputs are discussed in 4.1, and the other inputs are detailed in 5.2. (3) Weights are initialized (if not exist) for all candidate inputs found in (2). (4) The input with the highest weight (i.e., horror) is selected and sent to the VPA app, and the next outputs are received. (5) Based on the newly received outputs, Vitas updates the weights of all candidate inputs found in (2). (6) The FSM is updated according to the newly discovered states, inputs and transitions.

We set a list of typical VPA verbs that often appear, e.g., "*play*" and "*get*". We use SpaCy [7] again to calculate the similarity of the verb in the phrase with the verbs in our list, and directly use the highest similarity value as the initial weight. In this way, most invalid inputs are assigned with a low weight of about 0.5 to make sure they will not be invoked frequently.

- **Embedded-help inputs**. Embedded-help inputs are composed of expected inputs to VPA app's questions when user asks for "Help". The initial weight of embedded-help inputs is 0.7.
- **Context-related inputs**. The context-related inputs are often more closely related to the current state and are more likely to guide the discovery of new states. Therefore, the context-related inputs are initialized with the highest initial weight of 1.0.

Updating weights of candidate inputs (function `updateWeight`). To avoid the same paths being selected repeatedly, dynamically maintaining the candidate inputs' weights is of great importance to the exploration. We summarize four elements that can affect the weight updating as follows.

- **Input type.** The context-related inputs and embedded-help inputs have the highest priority because they are more related to the context and app functionality. Document-retrieved inputs are extracted from the document and they may not be valid, so they have a lower priority.
- **Covering a new functionality/state.** Covering a new state means that the VPA app's outputs contain a new sentence, and covering a new functionality means that the newly discovered state contains a word that represents certain functionality (see Section 4.2). If an input can trigger the app to cover a new functionality or a new state, it has the potential to explore more unknown states. Therefore, we increase its weight accordingly.
- **Number of invocation times.** The number of invocation times of an input is inversely proportional to its weight. The reason is that if Vitas has already executed one input for many times, the possibility that it can still lead to a new state is low.

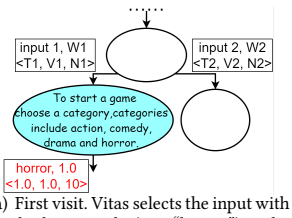
Suppose we are at a state s_i, and we express the input set of s_i as $C_i^* = \{C_i^1, C_i^2, ...C_i^n\}$ (n is the current size of C_i^*). After selecting the p-th input C_i^p from C_i^* as the input, we reach a state s_{i+1}. Now, we update the weights of every candidate input C_i^k from C_i^* ($1 \le k \le n$), represented as $W(s_i, C_i^k)$, by equation 1.

$$W(s_i, C_i^k) = \frac{\alpha T(s_i, C_i^k) + \beta V(s_i, C_i^k)}{\gamma N(s_i, C_i^k)} \tag{1}$$

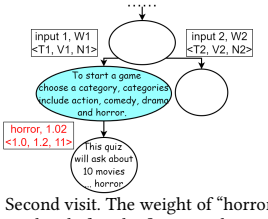
Here, T represents the input type factor. Context-sensitive and system-level inputs have the highest weight as the system is more likely to have meaningful responses to them. As we mentioned in Section 4.1, document-retrieved inputs are assigned with a relatively low weight as they can contain some invalid inputs. In order to prevent frequent interruptions in the execution, we set the minimum weight for system-level inputs.

$$T(s_i, C_i^k) = \begin{cases} 0.8 & C_i^k \text{ is a document-retrieved input} \\ 0.4 & C_i^k \text{ is a system-level input} \\ 1.0 & C_i^k \text{ is a context-related input} \\ 1.0 & C_i^k \text{ is a embedded-help input} \end{cases} \tag{2}$$

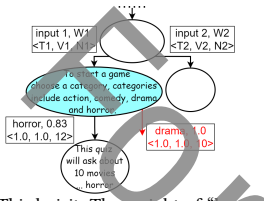
V represents the covering new states / functionalities factor. $V(s_i, C_i^k)$ is initialized to be 1.0. Denoting the value of $V(s_i, C_i^k)$ before executing the input C_i^p as $V'(s_i, C_i^k)$, the new value of $V(s_i, C_i^k)$ is calculated as follows:



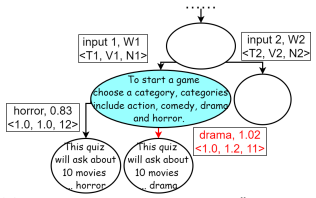
(a) First visit. Vitas selects the input with the highest weight (e.g., "horror") as the actual input.



(b) Second visit. The weight of "horror" was updated after the first visit, but still the highest.



(c) Third visit. The weight of "horror" was updated after the second visit. Now "drama" has the highest weight.



(d) Fourth visit. The weight of "drama" was updated after the third visit, but still the highest.

Figure 5: Adjustment of weights in each visit of the same states.

$$V(s_i, C_i^k) = \begin{cases} V'(s_i, C_i^k) + \Delta & C_i^k = C_i^p \\ V'(s_i, C_i^k) & C_i^k \neq C_i^p \end{cases} \tag{3}$$

$$\Delta = \begin{cases} 0.2 & \text{both a new state and a new functionality are covered} \\ 0.1 & \text{only a new state is covered} \\ -0.3 & s_i = s_{i+1}, \text{ it is a self loop} \\ -0.2 & s_{i+1} \text{ has been visited before} \end{cases} \tag{4}$$

N represents the invocation times factor. $N(s_i, C_i^k)$ is set to a constant number M for all the inputs in the initialization. $N(s_i, C_i^k)$ is increased when C_i^k is executed, and the N of other system-level inputs under s_i is decreased to make sure that the system-level inputs have a chance to be invoked near the end of the execution. Denoting the previous value of $N(s_i, C_i^k)$ before C_i^p is executed as $N'(s_i, C_i^k)$, the value of $N(s_0, C_i^k)$ can be calculated as follows:

$$N(s_0, C_i^k) = N'(s_0, C_i^k) + \theta \tag{5}$$

$$\theta = \begin{cases} 1 & C_i^k = C_i^p \\ -1 & C_i^k \text{ is a system-level input under } s_i \\ 0 & \text{other} \end{cases} \tag{6}$$

Figure 5 shows the change of the app model after each of four visits of the same states. The expansion of the model benefits from the proper adjustment of the candidate inputs' weights. In Figure 6, we demonstrate the "complete" model of "90's Movie Trivia" built by Vitas after executing and exploring the skill for three times. It is worth to note that since we only explore the skill for three times, this model does not cover all the potential states.

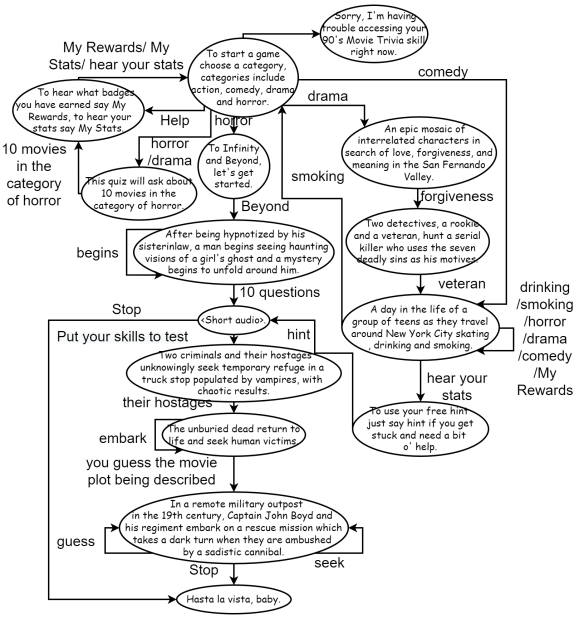


Figure 6: The FSM model of "90's Movie Trivia" generated by Vitas at last.

6 PROBLEM DETECTION

During the exploration, Vitas detects weaknesses from the app under testing. Every time new outputs are received, Vitas checks the outputs to detect problems. It mainly focuses on the following five types of problems that we summarize from the literature [21, 26, 32, 33] and our experience.

Problem 1: unexpected exit. This problem occurs when an app exits unexpectedly during the interaction, when no obvious exit signal is given.

Oracle of P1. When Vitas detects that the app is not alive—the app does not output within a time limit after being given an input—it checks its previous output. If there is no clear exit signs to remind the user, such as "goodbye" and "thank you", it raises an alert.

Problem 2: privacy violation. This problem occurs when an app asks the user for privacy information, e.g., name and address, as reported by Skillexplorer [21].

Oracle of P2: Vitas matches keywords related to privacy in the app outputs, such as "your name", "email", "gender", "address", etc. To reduce false positives, e.g., "*we will email you the steps*", it reports only when a keyword appears in the form of a noun. It also filters out the access that is included by the app's privacy permission.

Problem 3: unstoppable app. This problem occurs when the user gives a input of "Stop", "Cancel" or "Exit", the app does not exit. This problem is also targeted by Zhang et al. [33].

Oracle of P3: Vitas gives inputs accepted only by the VPA service, e.g., "What's the time?", following a termination input to the app. If the respondent is still the app, it reports an alert. Vitas will start another execution after detecting this problem.

Problem 4: unexpected app started. This problem occurs when some apps share the same invocation name, or multiple apps have confused invocation names, as reported by Kumar et al. [26]. This

problem leads the user to opening unexpected apps.

Oracle of P4: Vitas relies on the prompts from the VPA services to detect this problem. When opening an Alexa skill for the first time, Alexa prompts for the name of the opened skill, such as "Ok, Here is <skill's name>", or "Here is the skill <skill's name>, by <developer name>". Vitas uses the name of the opened VPA app to check whether an unexpected app is started. Vitas will stop the testing of this skill immediately after detecting this problem.

Problem 5: unstartable VPA app. This problem occurs when the invocation input is given, the called app is not started.

Oracle of P5: Vitas reports an alert when two situations occur. First, a series of alternative apps are provided, but none of them is the expected one. Second, the VPA service directly informs Vitas that the invocation input is not recognizable, although the app is listed in the app store. Vitas will also stop the testing of this skill immediately after detecting this problem.

7 EVALUATION

In this section, Vitas is evaluated on its coverage, efficiency, and accuracy of problem detection. To characterize the quality of existing VPA apps, we conduct a comprehensive study on Amazon skills, the apps of most popular VPA service. We also show that Vitas is applicable to other VPA apps such as Google Actions.

7.1 Settings

Settings. Since the speech-to-text and text-to-speech process is time-consuming and may bring additional errors. In our experiment, we use the simulator of skill [3], which is provided by Amazon, to interact with skills using texts instead of speech. The main difference between the simulator and the real machine is that the simulator also takes texts as inputs, but does not support flash-briefing skills.

Dataset. In the evaluation, we prepare two data sets. One is a Landscape set of Amazon skills online. Amazon skills are classified into 20 categories [4] (23 in Amazon store but three of them are covered by other 20). Since the skill simulator does not support the processing of skills in the flash-briefing category, we skip this category in our testing. We obtain all documents (e.g., name, invocation name, and description) of the rest skills into a large-scale data set. We randomly select 150 skills from our dataset as the benchmark dataset, on which we evaluate the performance of Vitas. In addition, to evaluate applicability of Vitas on other VPA apps, we also collect 150 Google Actions and apply Vitas on them.

Baselines. We compare Vitas with mainly two techniques including random testing, and Skillexplorer [21].

- **Random testing.** This strategy uses the same input set and model construction as Vitas, but selects input randomly at each state.
- **Skillexplorer [21].** Skillexplorer is a state-of-the-art model-free chatbot-style VPA app testing tool. It uses NLP-based input generation. It has no guidance during testing, and uses a pure DFS traversal.

The evaluation was run on the Ubuntu 18.04.5 machines with AMD EPYC 7702P 64-Core Processor CPU@1.996GHz and 2GB RAM. We make the source code and complete experimental results online, to faciliate future researh in VUI testing [10].

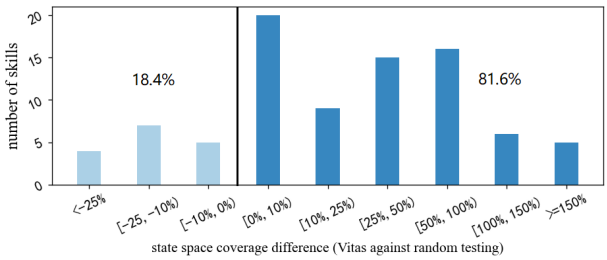


Figure 7: Comparison of state space coverage between Vitas and random testing.

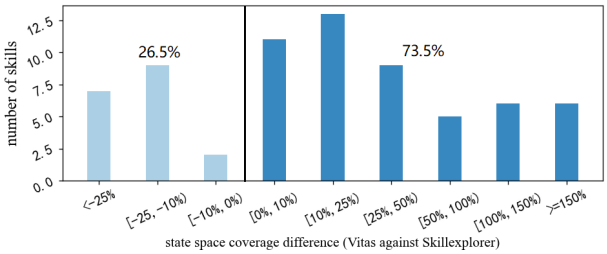


Figure 8: Comparison of state space coverage between Vitas and Skillexplorer.

7.2 The evaluation of Vitas

Our evaluation aims to answer the following research questions.

RQ1: Coverage. How large is the state space covered by Vitas's guided exploration strategy, compared with the baselines?

RQ2: Efficiency. How long does it take for Vitas to test an app on average? How many interaction rounds does it require for Vitas to achieve the same state space coverage compared with the baselines?

RQ3: Accuracy. How is the accuracy performance of Vitas in problems detection?

RQ4: Scalability. How large are the models generated by Vitas's state space exploration?

RQ5: Landscape. What is the *status quo* of the quality of existing skills?

RQ6: Applicability. Is Vitas applicable to other VPA apps?

In the following experiments, we set the value of variables mentioned in Section 5.3 as $\alpha = 0.4$, $\beta = 0.6$, $\gamma = 0.1$ and $M = 10$.

7.2.1 Study 1: Coverage of state space. To show the state space coverage of Vitas, we compare the size of the state space obtained by Vitas with the random testing and Skillexplorer on the benchmark dataset. In this experiment, we set a time limit of 10 minutes for the testing of each app as Skillexplorer [21] reports that the average testing time of a skill is about 627 seconds.

Figure 7 and 8 show the comparison of state space coverage between Vitas and baselines. Vitas reaches equal or higher coverage than random testing and Skillexplorer in 81.6% and 73.5% skills respectively[1].

[1]Due to issues like stability of internet connection and availability of specific apps, some apps may fail to be executed during testing. Therefore, we conduct the comparison on the set of apps that are successfully tested by all three techniques.

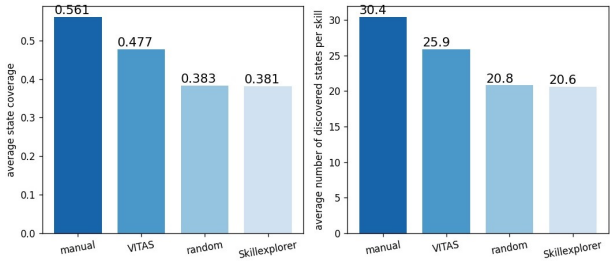


Figure 9: Average state coverage achieved by manual testing, Vitas, random testing, and Skillexplorer

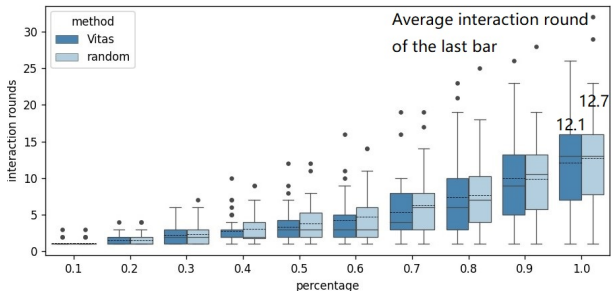


Figure 10: The number of interaction rounds required to reach a certain state space coverage between Vitas and random testing

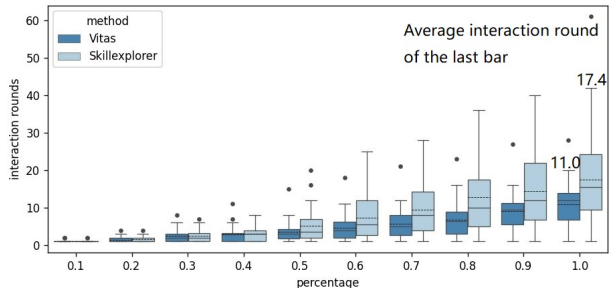


Figure 11: The number of interaction rounds required to reach a certain state space coverage between Vitas and Skillexplorer

As VPA apps are black-box services, it is difficult to obtain the exact size of their state space. We read the document of each app in our benchmark dataset, and manually test them for multiple times till it no longer gives new outputs. Then, we take the union of unique states that are generated by manual testing, Vitas, random testing, and Skillexplorer into a complete set. We take this set as the ground truth. We then report the average state space coverage of each technique in Fig. 9.

Our results show that VUI testing of VPA apps is a challenging task. Even we give manual testing no time limit, it can achieve only a low (56.1%) state space coverage. The reason might be that testers tend to give meaningful inputs, while some states can only be triggered by invalid inputs. We also observe that Vitas achieves a similar level of state space coverage with manual testing within only 10 minutes, and it outperforms other tools significantly.

Answer to RQ1: Vitas reaches equal or higher coverage than random testing and Skillexplorer in 81.6% and 73.5% skills respectively. This implies that Vitas could efficiently discover more behaviors of VPA apps than other techniques.

7.2.2 Study 2: Efficiency. In our experiment, we give techniques 10 minutes to test each app, so we skip the comparison on the testing time. Here we evaluate the efficiency from the aspect of the number of interaction rounds that are needed to achieve the same state space coverage for each technique.

Since two techniques may reach different coverage of the same app in 10 minutes' limit, directly comparing their interaction rounds may lose fairness. Thus, we only select those apps whose final state space size obtained by two techniques are within (-10%, 10%) to compare. Figure 10 and 11 show the number of interaction rounds required for Vitas and the baselines to achieve a certain state space coverage. We can see that Vitas needs fewer interaction rounds to reach the same coverage than other two, and keeps ahead to the end. Especially, Vitas spend 36.8% less interaction rounds than Skillexplorer to achieve the same level of coverage.

Answer to RQ2: Vitas requires fewer interaction rounds to achieve the same percentage of state space coverage than two baselines.

Table 2: The summary of testing results and accuracy on the benchmark skills

problem category	p1	p2	p3	p4	p5
simulator	28	3	9	3	2
verification	28	3	1	3	2
accuracy	100%	100%	11.1%	100%	100%

7.2.3 Study 3: Accuracy. After running experiments on the benchmark dataset, we record the number of apps that are detected with problems. We remove the results of those apps that are affected by network errors. We try to reproduce the results manually on a real machine to validate the results Vitas gets on the simulator. Table 2 is a summary of testing results and accuracy on the benchmark.

We can see that except the accuracy rate of p3 (unstoppable VPA apps) is relatively low, the accuracy of other problems are all 100%. We manually collect a set of apps that cannot exit normally on the real machine and test them with Vitas. The result shows that the apps marked as unstoppable by Vitas cover all these truly unstoppable apps. At the same time, those apps that are mistakenly recognized as unstoppable are truly unstoppable on the simulator. Therefore, we conclude that the inconsistency between the real machine and the simulator is the root cause of the low accuracy in detecting unstoppable VPA apps.

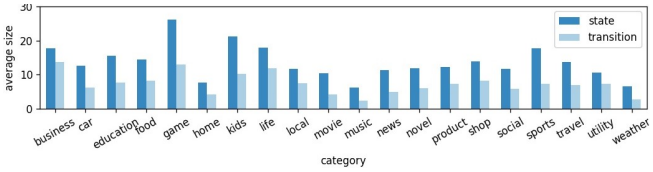


Figure 12: Average size of states and transitions of models constructed by Vitas on skills in different categories

Answer to RQ3: We have high accuracy for most of the problems detected. The accuracy for p3 (unstoppable VPA app) is low due to the inconsistent behaviors of the simulator and real machines, which is not a technical issue in Vitas.

7.2.4 Study 4: The scalability on large-scale dataset. The size of state space is determined by the number of different sentences of the skill in its all executions. The transition number is the number of different events of a user's input to trigger the transition from a state to another. The average size of state space and the average transition number of each category is shown in Fig. 12.

The average sizes of states and transitions vary from different categories. For the type 'home', 'music' and 'weather', state-space and transition number are relatively small. In the 'home' category, some skills require linking to a specific device to continue working. Due to the unavailability of such devices, Vitas terminates testing in an early stage. For 'music' and 'weather', the functionalities are relatively simple. Most skills in 'music' play some audio before exit, and most skills in 'weather' exit after reporting the weather.

Answer to RQ4: The average number of state and transition are 13.56 and 7.28 respectively for models on the large-scale dataset.

7.2.5 Study 5: Landscape on the large-scale data set. We have shown Vitas's performance on the state space coverage, efficiency and accuracy on the benchmark dataset. In this section, we conduct an experiment on the large-scale dataset to understand the *status quo* of the VPA apps in the wild. We remove those results of apps that may be inaccurate because of network exceptions. We list the numbers of apps that are detected as flawed by Vitas in Table 3. Due to space limitation, we release our testing log of each app in our website [10].

We can see that apps with p1 account for the largest proportion. Some of those apps quit without any obvious exit signs, but they complete the claimed functionality before exiting. So we further divide p1 into two categories, including those completing the task and those not. From the results, we can see that besides of those 10,881 apps that complete their tasks and exit without giving notices, 8,086 apps quit suddenly even before their tasks are completed. From our experiments, we find that the developers' careless designs may be the root cause for the high problem rate.

Table 3: The summary of problems detected by Vitas in different categories.

category	total	p (p1 + … + p5)	p1	p1 (complete)	p1 (not complete)	p2	p3	p4	p5
business	2093	893	788	294	494	180	53	15	20
car	123	47	46	15	31	1	1	0	1
education	5549	3134	2866	1632	1234	80	317	109	148
food	1231	674	631	257	374	48	25	7	29
game	5813	2411	2079	973	1106	154	202	97	114
home	882	454	431	56	375	10	7	11	13
kids	3558	1000	646	375	271	14	17	118	298
local	1060	451	411	181	230	44	14	12	16
movie	804	631	602	493	109	10	24	10	15
music	5802	4206	3898	3128	770	12	189	166	151
news	1146	601	538	315	223	47	11	12	19
novel	2698	1731	1631	993	638	17	33	90	54
product	4544	1975	1784	679	1105	76	55	91	57
shop	351	113	105	40	65	16	5	1	3
social	1606	608	547	255	292	34	11	20	23
sports	1201	578	527	260	267	4	18	34	12
travel	1127	389	350	137	213	48	14	12	13
utility	1346	769	610	380	230	31	13	18	9
weather	647	688	477	418	59	4	5	9	6
total	41581	21326	18967	10881	8086	830	1014	832	1001

Table 4: The summary of testing results and accuracy of Vitas on 150 Google Actions.

problem category	p1	p2	p3	p4	p5
simulator	10	4	1	0	61
verification	8	4	1	0	60
accuracy	80%	100%	100%	-	98.4%

Answer to RQ5: We test a total of 41,581 skills, and find that 51.29% of them have problems. Most apps suffer from p1 (expected exit). We further break down the apps with p1 into two types according to the completeness of their tasks. It is surprising to see 42.6% of apps with p1 exit their execution without completing their tasks.

7.2.6 Study 6: The applicability of Vitas on other VPA apps. To evaluate the applicability of Vitas for other VPA apps other than Alexa skills, we apply it on 150 Google Actions randomly downloaded from the app store. Similarly, we use Google's "simulator" as the testing platform [9].

Our testing results for these Actions are shown in Table 4. We confirm the results manually on the real machine to check the accuracy. We can see that Vitas is fully applicable to Google Actions.

Answer to RQ6: Vitas is also applicable to other VPA apps like Google Actions. It achieves a high accuracy of problems detection on Google Actions.

7.3 Discussion of Limitation and Threats

On some of the benchmarks, the state space coverage achieved by Vitas is comparatively low. After investigating this problem, we find that it is caused by the inaccuracy of detecting question types and generating inputs. For example, "Ready?" is a Yes-No question, but the verb and subject are omitted. The complete question should be "Are you ready?". Such incomplete sentences make it difficult to generate inputs, and further, affect the expansion of the state space. We would explore advanced NLP techniques to help with sentence understanding in the future work.

The accuracy of problem detection can also be improved. As mentioned in Section 7.1, we use simulator of skill [3] to interact with skills. For some skills, the simulator uses a version newer than the one on the real devices. It is not mature enough such that we find more problems on the simulator than on real devices. Furthermore, the simulator sometimes may have network exceptions, so the test results may be affected.

Last, our model can have redundant states that include different sentences but are semantically equivalent. We will conduct semantic analysis to simplify the model in future work.

8 RELATED WORK

Model-based Testing. Model-based testing is a widely used approach for various testing scenarios [30]. The most important and challenging task in model-based testing is to extract the model of the tested system [18]. In this regard, several studies have been conducted to automate the model extraction process. Chow [15] uses the FSM to model the software internal logic. AndroidRipper [11], AMOLA [13] and Stoat [31] are typical tools that use the FSM to represent GUI behavior.

Generating test cases from the built models is another key component of model-based testing. Various coverage criteria are extensively used for evaluating the adequacy of tests [27] and selecting suitable test cases [12]. Instead of randomly generating test cases that satisfy the coverage criteria [12], Hierons et al. [23] propose to use a stochastic model and extended mutation testing to model-based testing. Test cases are generated on the mutated model. Su et al. [31] also use the stochastic model to describe software behavior, but the test cases generated from the model are dynamically mutated according to the execution feedback. Besides, W [15], H [19], SPY [17] and P [16] methods are proposed to generate n-complete test suits based on a pre-built model.

VPA app Testing. Some recent studies have been conducted on testing VPA apps. Skillexplorer [21] analyzes the skills' outputs and generates potential commands by NLP techniques. After that, it performs a DFS-based exploration of the state space to detect privacy violations. LipFuzzer [35] is another VPA app tester. It generates voice commands that are likely to cause a semantic inconsistency and mutates the voice commands. All these commands are fed to the VPA apps to test whether the apps can understand them correctly. Different from these studies, Vitas proposes a model-based VUI testing method for VPA apps. It conducts model construction and exploration guided by weights of each command. In addition, Vitas targets five types of problems in VPA apps.

Security of VPA apps. More studies focus on the security issues in VPA apps. Kumar et al. [26] discover the skill squatting attack, with a focus on speech interpretation errors. By leveraging the predictable speech misinterpretation in the speech recognition process, users can be routed to a malicious skill. For example, a malicious skill with name "four moon" can be invoked even if the users expect to wake up a skill called "full moon". Zhang et al. [33] propose the skill masquerading attack. Malicious skills pretend to terminate execution by saying conclusions like "Bye", but still eavesdrop on the users' conversations. Some studies [20, 34] find that VPA apps eavesdrop and upload users' daily conversations. Long et al. [14] discover the unreliability of the skill certification by successfully getting 234 (100%) policy-violating skills certified. Recently in 2022, Skipper [32] reports that skills may collect data types inconsistent with that declared in the privacy policy. It will be an interesting work to extend our model-based VUI testing method to handle such security issues in the future.

9 CONCLUSION

In this paper, we propose Vitas, a guided model-based VUI testing method to test VPA apps automatically. Vitas retrieves semantic information from voice feedbacks by natural language processing. Based on the outputs of the app, it incrementally constructs the FSM model of the app to represent the app behaviors, and then employs a weighted exploration strategy guided by key factors such as the coverage of app functionalities. Our experiments show that Vitas can achieve higher state coverage than the state-of-the-art tools in a more efficient manner. We evaluate Vitas on 41,581 Alexa skills and find 51.29% of them are with problems. Our work reveals that the current VUI designs and implementations are not mature enough, and sheds light on the improvement of VPA apps.

ACKNOWLEDGMENTS

We are grateful for the constructive feedback of all the anonymous reviewers to improve this manuscript. The authors from Nanjing University are supported in part by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (No. BK20202001), the National Natural Science Foundation of China (No. 62172200 and No.62032010) and the Fundamental Research Funds for the Central Universities (No. 020214380094). The authors from IIE, Chinese Academy of Science, are supported in part by NSFC U1836211, Beijing Natural Science Foundation (No.M22004), the Anhui Department of Science and Technology under Grant 202103a05020009, Youth Innovation Promotion Association CAS, Beijing Academy of Artificial Intelligence (BAAI). The authors from University of Queensland are supported in part by UQ's NSRSG grant and Oracle Labs Australia under the CR grant.

REFERENCES

[1] 2014. Amazon.com: Alexa. https://www.amazon.com/b?node=21576558011.
[2] 2016. Google Assistant, your own personal Google. https://assistant.google.com.
[3] 2017. Alexa Simulator limitations. https://developer.amazon.com/en-US/docs/alexa/devconsole/test-your-skill.html#use-simulator.
[4] 2017. Amazon.com: Alexa Skills. https://www.amazon.com/s?i=alexa-skills.
[5] 2018. Portland Family Says Their Amazon Alexa Recorded Private Conversations. https://www.wweek.com/news/2018/05/26/portland-family-says-their-amazon-alexa-recorded-private-conversations-and-sent-them-to-a-random-contact-in-seattle/.
[6] 2019. Total number of Amazon Alexa skills from January 2016 to September 2019. https://www.statista.com/statistics/912856/amazon-alexa-skills-growth.
[7] 2020. *Industrial-Strength Natural Language Processings.* https://spacy.io

[8] 2021. Scenes|Conversational Actions|Google Developers. https://developers.google.com/assistant/conversational/scenes.

[9] 2021. Simulator|Actions console|Google Developers. https://developers.google.com/assistant/console/simulator.

[10] 2022. Vitas. https://vitas000.github.io/tool/.

[11] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, Michael Goedicke, Tim Menzies, and Motoshi Saeki (Eds.). ACM, 258–261.

[12] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Softw.* 32, 5 (2015), 53–59.

[13] Young Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 238–249.

[14] 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 *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 1699–1716.

[15] Tsun S. Chow. 1978. Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. Software Eng.* 4, 3 (1978), 178–187.

[16] Adenilso da Silva Simão and Alexandre Petrenko. 2010. Fault Coverage-Driven Incremental Test Generation. *Comput. J.* 53, 9 (2010), 1508–1522.

[17] Adenilso da Silva Simão, Alexandre Petrenko, and Nina Yevtushenko. 2009. Generating Reduced Tests for FSMs with Extra States. In *Testing of Software and Communication Systems, 21st IFIP WG 6.1 International Conference, TESTCOM 2009 and 9th International Workshop, FATES 2009, Eindhoven, The Netherlands, November 2-4, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5826)*, Manuel Núñez, Paul Baker, and Mercedes G. Merayo (Eds.). Springer, 129–145.

[18] Siddhartha R. Dalal, Ashish Jain, Nachimuthu Karunanithi, J. M. Leaton, Christopher M. Lott, Gardner C. Patton, and Bruce M. Horowitz. 1999. Model-Based Testing in Practice. In *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*, Barry W. Boehm, David Garlan, and Jeff Kramer (Eds.). ACM, 285–294.

[19] Rita Dorofeeva, Khaled El-Fakih, and Nina Yevtushenko. 2005. An Improved Conformance Testing Method. In *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3731)*, Farn Wang (Ed.). Springer, 204–218.

[20] Marcia Ford and William Palmer. 2019. Alexa, are you listening to me? An analysis of Alexa voice service network traffic. *Pers. Ubiquitous Comput.* 23, 1 (2019), 67–79.

[21] Zhixiu Guo, Zijin Lin, Pan Li, and Kai Chen. 2020. SkillExplorer: Understanding the Behavior of Skills in Large Scale. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2649–2666.

[22] Robert M Hierons and Mercedes G Merayo. 2009. Mutation testing from probabilistic and stochastic finite state machines. *Journal of Systems and Software* 82, 11 (2009), 1804–1818.

[23] Robert M. Hierons and Mercedes G. Merayo. 2009. Mutation testing from probabilistic and stochastic finite state machines. *J. Syst. Softw.* 82, 11 (2009), 1804–1818.

[24] Karen Spärck Jones. 2004. A statistical interpretation of term specificity and its application in retrieval. *J. Documentation* 60, 5 (2004), 493–502.

[25] Onur Kilinççeker, Alper Silistre, Fevzi Belli, and Moharram Challenger. 2021. Model-Based Ideal Testing of GUI Programs-Approach and Case Studies. *IEEE Access* 9 (2021), 68966–68984.

[26] Deepak Kumar, Riccardo Paccagnella, Paul Murley, Eric Hennenfent, Joshua Mason, Adam Bates, and Michael Bailey. 2018. Skill Squatting Attacks on Amazon Alexa. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 33–47.

[27] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. 2001. Coverage criteria for GUI testing. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*, A Min Tjoa and Volker Gruhn (Eds.). ACM, 256–267.

[28] Marius Mikucionis, Kim Guldstrand Larsen, and Brian Nielsen. 2004. T-UPPAAL: Online Model-based Testing of Real-Time Systems. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria*. IEEE Computer Society, 396–397.

[29] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2016. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 543–553.

[30] Muhammad Shafique and Yvan Labiche. 2010. A systematic review of model based testing tool support. *SCE Technical Reports* (2010).

[31] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 245–256.

[32] Fuman Xie, Yanjun Zhang, Chuan Yan, Suwan Li, Lei Bu, Kai Chen, Zi Huang, and Guangdong Bai. 2022. Scrutinizing Privacy Policy Compliance of Virtual Personal Assistant Apps. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*.

[33] 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 *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1381–1396.

[34] Nan Zhang, Xianghang Mi, Xuan Feng, Xiao Feng 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. *IEEE Symposium on Security and Privacy* (2019).

[35] 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 *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.