Testing the Limits: Unusual Text Inputs Generation for Mobile App Crash Detection with Large Language Model

Zhe Liu[1,2],Chunyang Chen[3], Junjie Wang[1,2,*], Mengzhuo Chen[1,2], Boyu Wu[2,4], Zhilin Tian[5], Yuekai Huang[1,2], Jun Hu[1,2], Qing Wang[1,2,6,*]
[1]State Key Laboratory of Intelligent Game, Beijing, China
Institute of Software Chinese Academy of Sciences, Beijing, China;
[2]University of Chinese Academy of Sciences, Beijing, China; [*]Corresponding author;
[3]Technical University of Munich, Munich, Germany; [4]DiDi Global Inc;
[5]The Pennsylvania State University; Work done during the internship at ISCAS;
[6]Science & Technology on Integrated Information System Laboratory
liuzhe181@mails.ucas.ac.cn,Chunyang.chen@monash.edu,junjie@iscas.ac.cn,wq@iscas.ac.cn

ABSTRACT

Mobile applications have become a ubiquitous part of our daily life, providing users with access to various services and utilities. Text input, as an important interaction channel between users and applications, plays an important role in core functionality such as search queries, authentication, messaging, etc. However, certain special text (e.g., -18 for Font Size) can cause the app to crash, and generating diversified unusual inputs for fully testing the app is highly demanded. Nevertheless, this is also challenging due to the combination of explosion dilemma, high context sensitivity, and complex constraint relations. This paper proposes InputBlaster which leverages the LLM to automatically generate unusual text inputs for mobile app crash detection. It formulates the unusual inputs generation problem as a task of producing a set of test generators, each of which can yield a batch of unusual text inputs under the same mutation rule. In detail, InputBlaster leverages LLM to produce the test generators together with the mutation rules serving as the reasoning chain, and utilizes the in-context learning schema to demonstrate the LLM with examples for boosting the performance. InputBlaster is evaluated on 36 text input widgets with cash bugs involving 31 popular Android apps, and results show that it achieves 78% bug detection rate, with 136% higher than the best baseline. Besides, we integrate it with the automated GUI testing tool and detect 37 unseen crashes in real-world apps.

KEYWORDS

Android GUI testing, Large language model, In-context learning

ACM Reference Format:
Zhe Liu[1,2],Chunyang Chen[3], Junjie Wang[1,2,*], Mengzhuo Chen[1,2], Boyu Wu[2,4], Zhilin Tian[5],, Yuekai Huang[1,2], Jun Hu[1,2], Qing Wang[1,2,6,*]. 2024. Testing the Limits: Unusual Text Inputs Generation for Mobile App Crash Detection with Large Language Model. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639118

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639118

1 INTRODUCTION

Mobile applications (apps) have become an indispensable component of our daily lives, enabling instant access to a myriad of services, information, and communication platforms. The increasing reliance on these applications necessitates a high standard of quality and performance to ensure user satisfaction and maintain a competitive edge in the fast-paced digital landscape. The ubiquity of mobile applications has led to a constant need for rigorous testing and validation to ensure their reliability and resilience against unexpected user inputs.

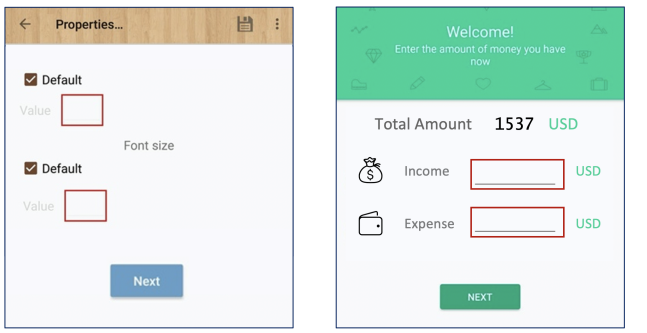
Text input plays a crucial role in the usability and functionality of mobile applications, serving as a primary means for users to interact with and navigate these digital environments [43, 44]. From search queries and form submissions to instant messaging and content creation, text input is integral to the core functionality of numerous mobile applications across various domains. The seamless handling of text input is essential for delivering a positive user experience, as it directly impacts the ease of use, efficiency, and overall satisfaction of the users.

Given the unexpected input, the program might suffer from memory leakage, data corruption, falling into the dead loop, resulting in the application stuck, crash, or other serious issues [14, 27, 28, 66]. Even worse, these buggy texts can only demonstrate a tiny difference from the normal text, or they themselves are normal text in other contexts, which makes the issue easily occur and difficult to spot. There has been a fair amount in the news about the crash of iOS and Android systems caused by a special text input [1], which has greatly affected people's daily lives. For example, in July 2020, a specific character of the Indian language caused iOS devices constantly crash. It has affected a wide range of iOS applications, including iMessage, WhatsApp, and Facebook Messenger [2], and as long as certain text inputs contain the character, these apps would crash.

Taken in this sense, automatically generating unusual inputs for fully testing the input widgets and uncovering bugs is highly demanded. Existing automated GUI testing techniques focus on generating the valid text input for passing the GUI page and conducting the follow-up page exploration [6, 8, 27, 43, 44, 65, 66], e.g., QTypist [44] used GPT-3 to generate semantic input text to improve the coverage of the test. They could not be easily adapted to this task, since the unusual inputs can be more diversified and

follow different rationales from the valid inputs. There are also studies targeting at generating strings that violate the constraints (e.g., string length) with heuristic analysis or finite state automaton techniques [37, 42, 67]. Yet they are designed for specific string functions like concatenation and replacement, and could not be generalized in this task.

Nevertheless, it is very challenging for the automatic generation of diversified unusual inputs. The first challenge is the combination explosion. There can be numerous input formats including text, number, date, time, currency, and innumerable settings, e.g., different character sets, languages and text lengths, which makes it quite difficult if not impossible to enumerate all these variants. The second challenge is context sensitivity. The unusual inputs should closely relate to the context of the input widgets to effectively trigger the bug, e.g., a negative value for font size (as shown in Figure 1), an extremely large number to potentially violate the widget for people's height. The third challenge is the constraint relation within and among the input widgets. The constraints can be that a widget only accepts pure numbers (without characters), or the sum of item values smaller/bigger than the total (as shown in Figure 1), which requires an exact understanding of the related widgets and these constraints so as to generate targeted variation. What's more difficult is that certain constraints only appear when interacting with the apps (i.e., dynamic hints in terms of the incorrect texts), and static analysis cannot capture these circumstances.



Inputs triggering crash: -3.2, -18
Illustration: only negative number can trigger the crash

Inputs triggering crash: 30, 1878
Illustration: only when 'expense' larger than the sum of 'total amount' and 'income' , the crash can be triggered

Figure 1: Example bugs triggered by unusual inputs.

Large Language Models (LLMs) [10, 17, 61, 69, 73] trained on ultra-large-scale corpus have exhibited promising performance in a wide range of tasks. ChatGPT[61], developed by OpenAI, is one such LLM with an impressive 175 billion parameters, trained on a vast dataset. Its ability to comprehend and generate text across various domains is a testament to the potential of LLMs in interacting with humans as knowledgeable experts. The success of ChatGPT is a clear indication that LLMs can understand human knowledge and can do well in providing answers to various questions.

Inspired by the fact that the LLM has made outstanding progress in email reply, abstract extraction, etc. [10, 16, 35, 71], we propose an approach, InputBlaster[1], to automatically generate the unusual text inputs with LLM which uncover the bugs[2] related to the text input widgets. Instead of directly generating the unusual inputs by LLM which is of low efficiency, we formulate the unusual inputs generation problem as a task of producing a set of test generators (a code snippet), each of which can yield a batch of unusual text inputs under the same mutation rule (i.e., insert special characters into a string), as demonstrated in Figure 4 ⑤.

To achieve this, InputBlaster leverages LLM to produce the test generators together with the mutation rules which serve as the reasoning chains for boosting the performance. In detail, InputBlaster first leverages LLM to generate the valid input which can pass the GUI page and serves as the target for the follow-up mutation (Module 1). Based on it, it then leverages LLM to produce mutation rules, and asks the LLM to follow those mutation rules and produce the test generator, each of which can yield a batch of unusual text inputs (Module 2). To further boost the performance, we utilize the in-context learning schema to demonstrate the LLM with useful examples from online issue reports and historical running records (Module 3).

To evaluate the effectiveness of InputBlaster, we carry out experiments on 36 text input widgets with cash bugs involving 31 popular Android apps in Google Play. Compared with 18 common-used and state-of-the-art baselines, InputBlaster can achieve more than 136% boost in bug detection rate compared with the best baseline, resulting in 78% bugs being detected. In order to further understand the role of each module and sub-module of the approach, we conduct ablation experiments to further demonstrate its effectiveness. We also evaluate the usefulness of InputBlaster by integrating it with the automated GUI testing tool and detecting unseen crash bugs in real-world apps from Google Play. Among 131 apps, InputBlaster detects 37 new crash bugs with 28 of them being confirmed and fixed by developers, while the remaining are still pending.

The contributions of this paper are as follows:

- We are the first to propose a novel LLM-based approach InputBlaster[3] for the automatic generation of unusual text inputs for mobile app testing.
- We conduct the first empirical categorization of the constraint relationships within and among text input widgets, which provides clues for the LLM in effective mutation, and facilitates the follow-up studies on this task.
- We carry out the effectiveness and usefulness evaluation of InputBlaster, with a promising performance largely outperforming baselines and 37 new detected bugs.

2 MOTIVATIONAL STUDY AND BACKGROUND

To better understand the constraints of text inputs in real-world mobile apps, we carry out a pilot study to examine their prevalence. We also categorize the constraints, to facilitate understanding and the design of our approach for generating unusual inputs violating the constraints.

[1]Our approach is named as InputBlaster considering it likes a blaster which ignites the following production of the unusual inputs.

[2]Note that, like existing studies [38, 40, 56], this paper focuses on the crash bug, which usually causes more serious effects and can be automatically observed, and we interchangeably use the term bug and crash.

[3]https://github.com/franklinbill/InputBlaster provides the dataset, source code of InputBlaster, and the detailed experimental results of this paper.

2.1 Motivational Study

2.1.1 ***Data Collection.*** The dataset is collected from one of the largest Android GUI datasets Rico [19], which has a great number of Android GUI screenshots and their corresponding view hierarchy files [45, 46]. These apps belong to diversified categories such as news, entertainment, medical, etc. We analyze the view hierarchy file according to the package name and extract the GUI page belonging to the same app. A total of 7,136 apps with each having more than 3 GUI pages are extracted. For these apps, we first randomly select 136 apps with 506 GUI pages and check their text inputs through view hierarchy files. We summarize a set of keywords that indicate the apps have text inputs widgets [30], e.g., *EditText, hint-text, AutoCompleteTextView, etc*[3]. We then use these keywords to automatically filter the view hierarchy files from the remaining 7,000 apps, and obtain 5,761 candidate apps with at least one potential text input widget. Four authors then manually check them to ensure that they have text inputs until a consensus is reached. In this way, we finally obtain 5,013 (70.2%) apps with at least one text input widget, and there are 3,723 (52.2%) apps having two or more text input widgets. Please note that there is no overlap with the evaluation dataset.

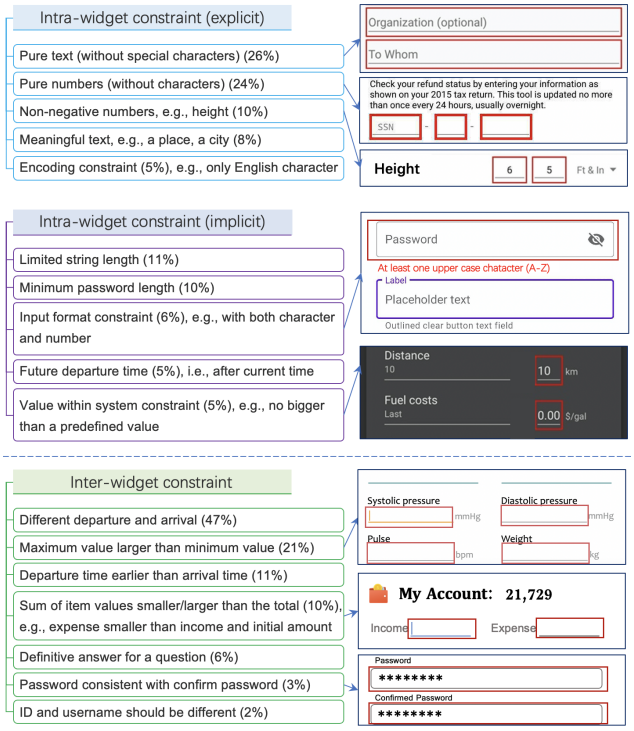


Figure 2: The category of constraints.

2.1.2 ***The Constraint Categories of Text Inputs.*** We randomly select 2000 apps with text inputs and conduct manual categorization to derive the constraint types of input widgets. Following the open coding protocol [62], two authors individually examine the content of the text input, including the app name, activity name, input type and input content. Then each annotator iteratively merges similar codes, and any disagreement of the categorization will be handed over to the third experienced researcher for double checking. Finally, we come out with a categorization of the constraints within (intra-widget) and among the widgets (inter-widget), with details summarized in Figure 2.

Intra-widget constraint. Intra-widget constraints depict the requirements of a single text input, e.g., a widget for a human's height can only input the non-negative number. There are explicit and implicit sub-types. The former accounts for 63%, which manifests as the requirement to display input directly on the GUI page. And the latter account for 37%, mainly manifested as the feedback when incorrect text input is received, e.g., after inputting a simple password, the app would remind the user "at least one upper case character (A-Z) is required" as demonstrated in Figure 2.

Inter-widget constraint. Inter-widget constraints depict the requirements among multiple text input widgets on a GUI page, for example, the diastolic pressure should be less than systolic pressure as shown in Figure 2.

Summary. As demonstrated above, the text input widgets are quite common in mobile apps, e.g., 70.2% apps with at least one such widget. Furthermore, considering the diversity of inputs and contexts, it would require significant efforts to manually build a complete set of mutation rules to fully test an input widget, and the automated technique is highly demanded. This confirms the popularity of text inputs in mobile apps and the complexity of it for full testing, which motivates us to automatically generate a batch of unusual text inputs for effective testing and bug detection.

2.2 Background of LLM and In-context Learning

The target of this work is to generate the input text, and the Large Language Model (LLM) trained on ultra-large-scale corpus can understand the input prompts (sentences with prepending instructions or a few examples) and generate reasonable text. When pre-trained on billions of samples from the Internet, recent LLMs (like ChatGPT [61], GPT-3 [10] and T5 [59]) encode enough information to support many natural language processing tasks [50, 63, 71].

Tuning a large pre-trained model can be expensive and impractical for researchers, especially when limited fine-tuned data is available for certain tasks. In-context Learning (ICL) [11, 25, 54] offers a new alternative that uses Large Language Models to perform downstream tasks without requiring parameter updates. It leverages input-output demonstration in the prompt to help the model learn the semantics of the task. This new paradigm has achieved impressive results in various tasks, including code generation and assertion generation.

3 APPROACH

This paper aims at automatically generating a batch of unusual text inputs which can possibly make the mobile apps crash. The common practice might directly produce the target inputs with LLM as existing studies in valid input generation [44] and fuzzing deep learning libraries [20, 21]. Yet, this would be quite inefficient for our task, because each interaction with the LLM requires a few seconds waiting for the response and consumes lots of energy. Instead, this paper proposes to produce the test generators (a code snippet) with LLM, each of which can generate a batch of unusual text inputs under the same mutation rule (e.g., insert special characters into a string), as demonstrated in Figure 4 ⑤.

Zhe Liu[1,2],Chunyang Chen[3], Junjie Wang[1,2,*], Mengzhuo Chen[1,2], Boyu Wu[2,4], Zhilin Tian[5], Yuekai Huang[1,2], Jun Hu[1,2], Qing Wang[1,2,6,*]

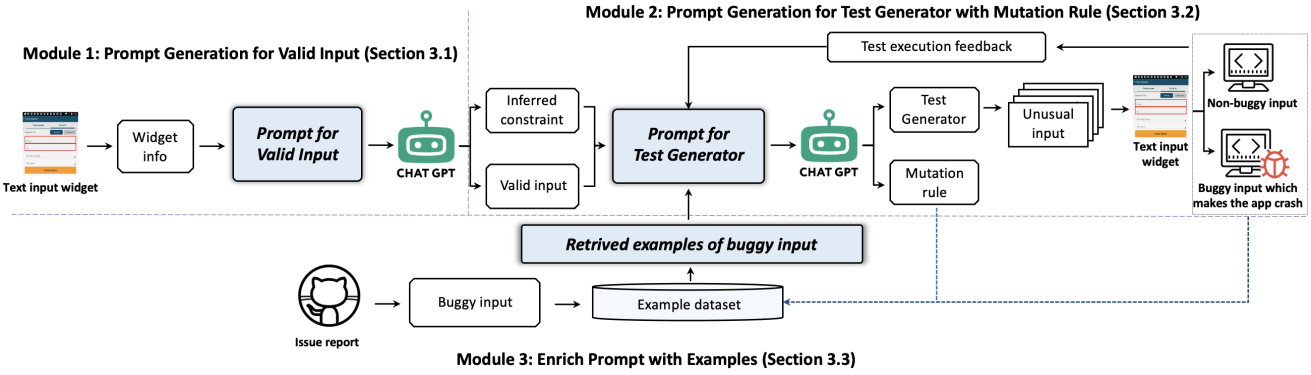


Figure 3: Overview of InputBlaster.

To achieve this, we propose InputBlaster which leverages LLM to produce the test generators together with the mutation rules which serve as the reasoning chains for boosting the performance, and each test generator then automatically generates a batch of unusual text inputs, as shown in Figure 3. In detail, given a GUI page with text input widgets and its corresponding view hierarchy file, we first leverage LLM to generate the valid text input which can pass the GUI page (Sec 3.1). We then leverage LLM to produce the test generator which can generate a batch of unusual text inputs, and simultaneously we also ask the LLM to output the mutation rule which serves as the reasoning chain for guiding the LLM in making the effective mutations from valid inputs (Sec 3.2). To further boost the performance, we utilize the in-context learning schema to provide useful examples when querying the LLM, from online issue reports and historical running records (Sec 3.3).

3.1 Prompt Generation for Valid Input

InputBlaster first leverages LLM to generate the valid input which will serve as the target towards which the following mutation can be conducted. The context information relates to the input widgets and its belonged GUI page can provide important clues about what the valid input should be, therefore we input this information into LLM (in Section 3.1.1). In addition, we also include the dynamic feedback information when interacting with the input widgets (in Section 3.1.2), and the constraint categories we summarized in the previous section (in Section 3.1.3) to improve the performance. Furthermore, besides the valid text input, we also ask LLM to output its inferred constraints for generating the valid input which will facilitate the approach to generating the mutation rules in the next section. We summarize all the extracted information with examples in Table 1.

3.1.1 ***Context Extraction.*** The context information is extracted from the view hierarchy file, which is easily obtained by automated GUI testing tools [26, 51, 52, 64]. As shown in Table 1, we extract the text-related field of the input widget which indicates how the valid input should be. In detail, we extract the "hint text", "resource id", and 'text' fields of the input widget, and utilize the first non-empty one among the above three fields.

We also extract the activity name of the GUI page and the mobile app name, and this global context further helps refine the understanding of the input widget. In addition, we extract the local context of the input widget (i.e., from nearby widgets) to provide thorough viewpoints and help clarify the meaning of the widget. The candidate information source includes the parent node widgets, the leaf node widget, widgets in the same horizontal axis, and fragment of the current GUI page. For each information source, we extract the "text" field (if it is empty, use the "resource-id" field), and concatenate them into the natural-language description with the separator (';').

3.1.2 ***Dynamic Hint Extraction.*** When one inputs an incorrect text into the app, there are some feedbacks (i.e., dynamic hints) related to the inputs, e.g., the app may alter the users that the password should contain letters and digits. The dynamic hint can further help LLM understand what the valid input should look like.

We extract the dynamic hints via differential analysis which compares the differences of the GUI page before and after inputting the text, and extracts the text field of the newly emerged widgets (e.g., a popup window) in the later GUI page, with examples shown in Figure 2. We also record the text input which makes the dynamic hint happens, which can help the LLM to understand the reason behind it.

3.1.3 ***Candidate Constraints Preparation.*** Our pilot study in Section 2.1.2 summarizes the categories of constraints within and among the widgets. The information can provide direct guidance for the LLM in generating the valid inputs, for example, the constraint explicitly requires the input should be pure text (without special characters). We provide this list of all candidate constraints described in natural language as in Section 2.1.2 to the LLM.

3.1.4 ***Prompt Generation.*** With the extracted information, we use three kinds of information to generate prompts for inputting into the LLM, as shown in Table 1. Generally speaking, it first provides the context information and the dynamic hints (if any) of the input widgets, followed by the candidate constraints, and then queries the LLM for the valid input. Due to the robustness of LLM, the generated prompt sentence does not need to fully follow the grammar.

After inputting the prompt, the LLM will return its recommended valid text input and its inferred constraints, as demonstrated in Figure 4 ②. We then input it into the widget, and check whether it can make the app transfer to the new GUI page (i.e., valid input). If the app fails to transfer, we iterate the process until the valid input is generated.

Table 1: The example of extracted information and linguistic patterns of prompts for Module 1.

Extracted information			
Id	**Attribute**	**Description**	**Examples**
I1	AppName	The name of testing app	AppName = "Wallet"
I2	PageName	Activity name of the current GUI page	PageName = "User"
I3	InputWidget	The text input widget(s) denoted with the textual related fields	InputWidget = "Please input user name"
I4	NearbyWidget	Nearby widgets denoted with their textual related fields	NearbyWidget = "your income: [SEP] $ "
I5	DynamicHint	Feedbacks in terms of an incorrect input	DynamicHint = "password should contain letters"
I6	CandidateConstraints	Candidate constraints within or among widget(s) summarized in pilot study, organized into intra-constraint(explicit), intra-constraint(implicit), and inter-constraint	CandidateConstraints = "intra-constraints(explicit): (1) Pure text (without special characters) ... "
Linguistic patterns of prompts			
Id	**Target**	**Pattern**	**Examples**
P1	Provide context information of the text input widgets	We want to test the text input widgets on $\langle PageName \rangle$ page of $\langle AppName \rangle$ app which has $\langle \#NumOfInputWidget \rangle$ text inputs. The first input widget is $\langle InputWidget \rangle$, its context is $\langle InputWidget \rangle$, and its dynamic hint is $\langle DynamicHint \rangle$. The second input	We want to test the text input widgets on User page of Wallet app which has 3 text inputs. The first input widget is 'username', its context is 'Welcome to ...', and its dynamic hint is 'Username already in use'. ...
P2	Provide candidate constraints	There are 5 explicit intra-constraints: $\langle intra-constraint(explicit) \rangle$; 5 implicit intra-constraints: $\langle intra-constraint(implicit) \rangle$; 7 inter-constraints: $\langle inter-constraint \rangle$	There are 5 explicit intra-constraints: (1) Pure text ... ; 5 implicit intra-constraints: (1) Limited string length ...; 7 inter-constraints: (1) Departure and Arrival ...
P3	Query LLM	Please generate a valid input based on the above information and provide the inferred constraints of each input.	

3.2 Prompt Generation for Test Generator with Mutation Rule

Based on the valid input in the previous section, InputBlaster then leverages LLM to produce the test generator together with the mutation rule. As demonstrated in Figure 4 ⑤, the test generator is a code snippet that can generate a batch of unusual inputs, while the mutation rule is the natural language described operation for mutating the valid inputs which automatically output by LLM based on our prompt and serves as the reasoning chain for producing the test generator. Note that the mutation rule here is output by LLM.

Each time when a test generator is produced, we can obtain a batch of automatically generated unusual text inputs, and will input them into the text widgets to check whether they have successfully made the mobile app crash. This test execution feedback (in Section 3.2.2) will be incorporated in the prompt for querying the LLM which can enable it more familiar with how the mutation works and potentially produce more diversified outcomes. We also include the inferred constraints in the previous section in the prompt (in Section 3.2.1), since the natural language described explanation would facilitate the LLM in producing effective mutation rules, for example, the inferred constraint is that the input should be in pure text (without special characters) and the LLM would try to insert certain characters to violate the constraint.

3.2.1 ***Inferred Constraints and Valid Input Extraction.*** We have obtained the inferred constraints and valid input from the output of the LLM in the previous section, here we extract this information from the output message and will input it into the LLM in this section. We design a flexible keyword matching method to automatically extract the description between the terms like 'constraints' and 'the input' and treat it as the inferred constraints, and extract the description after the terms like 'input is' and treat it as the valid input, as demonstrated in Figure 4 ②.

3.2.2 ***Test Execution Feedback Extraction.*** After generating the unusual text inputs, we input them into the mobile app and check whether they can successfully trigger the app crash. This test execution information will be inputted into the LLM to generate more effective and diversified text inputs. We use the real buggy text inputs and the other unusual inputs (which don't trigger bugs) to prompt LLM in the follow-up generation. The former can remind the LLM to avoid generating duplicate ones, while the latter aims at telling the LLM to consider other mutation rules.

Besides, we also associate the mutation rules with the text input to enable the LLM to better capture its semantic meaning. As shown in Figure 4 ⑤, we extract the content between the keywords "Mutation rule" and "Test generator" as mutation rules.

3.2.3 ***Prompt Generation.*** With the extracted information, we design linguistic patterns of the prompt for generating the test generator and mutation rules. As shown in Figure 4 ④, the prompt includes four kinds of information, namely inferred constraints, valid input, text execution feedback, and question. The first three kinds of information are mainly based on the extracted information as described above, and we also add some background illustrations to let the LLM better understand the task, like the inferred constraint in Figure 4 ④. For the question, we first ask the LLM to generate the mutation rule for the valid input, then let it produce a test generator following the mutation rule. Due to the robustness of LLM, the generated prompt sentence does not need to follow the grammar completely.

3.3 Enriching Prompt with Examples

It is usually difficult for LLM to perform well on domain-specific tasks as ours, and a common practice would be employing the in-context learning schema to boost the performance. It provides the LLM with examples to demonstrate what the instruction is, which enables the LLM better understand the task. Following the schema, along with the prompt for the test generator as described in Section 3.2, we additionally provide the LLM with examples of the unusual inputs. To achieve this, we first build a basic example dataset of buggy inputs (which truly trigger the crash) from the issue reports of open-source mobile apps, and continuously enlarge it with the running records during the testing process (in Section 3.3.1). Based on the example dataset, we design a retrieval-based example selection method (in Section 3.3.2) to choose the most suitable examples in terms of an input widget, which further enables the LLM to learn with pertinence.

Zhe Liu[1,2],Chunyang Chen[3], Junjie Wang[1,2,*], Mengzhuo Chen[1,2], Boyu Wu[2,4], Zhilin Tian[5], Yuekai Huang[1,2], Jun Hu[1,2], Qing Wang[1,2,6,*]

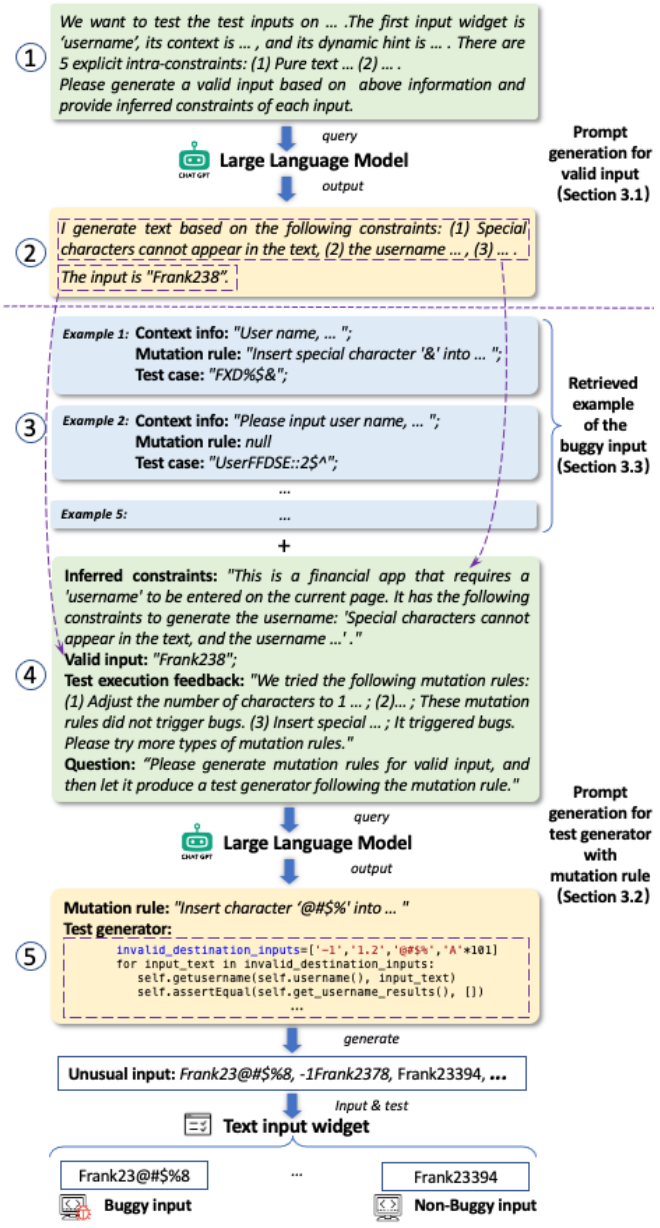


Figure 4: Example of how InputBlaster works.

3.3.1 ***Example Dataset Construction.*** We collect the buggy text inputs from GitHub and continuously build an example dataset that serves as the basis for in-context learning. For each data instance, as demonstrated in 4 ③, it records the buggy text inputs and the mutation rules which facilitate the LLM understanding of how the buggy inputs come from. It also includes the context information of the input widgets which provides the background information of the buggy inputs, and enables us to select the most suitable examples when querying the LLM.

Mining buggy text inputs from GitHub. First, we automatically crawl the issue reports and pull requests from the Android mobile apps in GitHub (updated before September 2022). Then we use keyword matching to filter these related to the text inputs (e.g., EditText) and have triggered crashes. Following that, we then employ manual checking to further determine whether there is a crash triggered by the buggy text inputs by running the app. In this way, we obtain 50 unusual inputs and store them in the example dataset (There is no overlap with the evaluation datasets.). We then extract the context information of the input widget with the method in Section 3.1.1, and store it together with the unusual input. Note that, since these buggy inputs don't associate with the mutation rules, we set them as null.

Enlarging the dataset with buggy text inputs during testing. We enrich the example dataset with the newly emerged unusual text inputs which truly trigger bugs during InputBlaster runs on various apps. Specifically, for each generated unusual text input, after running it in the mobile apps, we put the ones which trigger crashes into the example dataset. We also add their associated mutation rules generated by the LLM, as well as the context information extracted in Section 3.1.1.

3.3.2 ***Retrieval-based Example Selection and In-context Learning.*** Examples can provide intuitive guidance to the LLM in accomplishing a task, yet excessive examples might mislead the LLM and cause the performance to decline. Therefore, we design a retrieval-based example selection method to choose the most suitable examples (i.e., most similar to the input widgets) for LLM.

In detail, the similarity comparison is based on the context information of the input widgets. We use Word2Vec (Lightweight word embedding method) [53] to encode the context information of each input widget into a 300-dimensional sentence embedding, and calculate the cosine similarity between the input widget and each data instance in the example dataset. We choose the top-K data instance with the highest similarity score, and set K as 5 empirically.

The selected data instances (i.e., examples) will be provided to the LLM in the format of context information, mutation rule, and buggy text input, as demonstrated in Figure 4 ③.

3.4 Implementation

We implement InputBlaster based on the ChatGPT which is released on the OpenAI website[4]. It obtains the view hierarchy file of the current GUI page through UIAutomator [68] to extract context information of the input widgets. InputBlaster can be integrated by replacing the text input generation module of the automated GUI testing tool, which automatically extracts the context information and generates the unusual inputs.

4 EXPERIMENT DESIGN

4.1 Research Questions

- **RQ1: (Bugs Detection Performance)** How effective of InputBlaster in detecting bugs related to text input widgets?

For RQ1, we first present some general views of InputBlaster for bug detection, and then compare it with commonly-used and state-of-the-art baseline approaches.

- **RQ2: (Ablation Study)** What is the contribution of the (sub-) modules of InputBlaster for bug detection performance?

[4]https://beta.openai.com/docs/models/chatgpt

For RQ2, We conduct ablation experiments to evaluate the impact of each (sub-) module on the performance.

- **RQ3: (Usefulness Evaluation)** How does our proposed InputBlaster work in real-world situations?

For RQ3, we integrate InputBlaster with the GUI testing tool to make it automatically explore the app and detect unseen input-related bugs, and issue the detected bugs to the development team.

4.2 Experimental Setup

For RQ1 and RQ2, we crawl 200 most popular open-source apps from F-Droid [3], and only keep the latest ones with at least one update after September 2022 (this ensures the utilized apps are not overlapped with the ones in Sec 3.3). Then we collect all their issue reports on GitHub, and use keywords (e.g., EditText) to filter those related to text input. Finally, we obtain 126 issue reports related to 54 apps. Then we manually review each issue report and the mobile app, and filter it according to the following criteria: (1) the app wouldn't constantly crash on the emulator; (2) it can run all baselines; (3) UIAutomator [68] can obtain the view hierarchy file for context extraction; (4) the bug is related to text input widgets; (5) the bug can be manually reproduced for validation; (6) the app is not used in the motivational study or example dataset construction. Please note that we follow the name of the app to ensure that there is no overlap between the datasets. Finally, 31 apps with 36 buggy text inputs remain for further experiments.

We measure the bug detection rate, i.e., the ratio of successfully triggered crashes in terms of all the experimental crashes (i.e., buggy inputs), which is a widely used metric for evaluating GUI testing [8, 27, 43]. Specifically, with the generated unusual input, we design an automated test script to input it into the text input widgets, and automatically run the "submit" operation to check whether a crash occurs. If no, use the script to go back to GUI page with the input widget if necessary, and try the next generated unusual input. As long as a crash is triggered for a text input widget, we treat it as successful bug detection and will stop the generation for this widget. Note that our generated unusual input is not necessarily the same as the one provided in the issue report, e.g., -18 vs. -20, as long as a crash is triggered after entering the unusual inputs, we treat it as a successful crash detection.

For a fair comparison with other approaches, we employ two experimental settings, i.e., 30 attempts (30 unusual inputs) and 30 minutes. We record the bug detection rate under each setting (denoted as "Bug (%)" in Table 2 to Table 5), and also record the actual number of attempts (denoted as "Attempt (#)") and the actual running time (denoted as "Min (#)") when the crash occurs to fully understanding the performance.

For RQ3, we further evaluate the usefulness of InputBlaster in detecting unseen crash bugs related to text input. A total of 131 apps have been retained. We run Ape [26] (a commonly-used automated GUI testing tool) integrated with InputBlaster, for exploring the mobile apps and getting the view hierarchy file of each GUI page. We use the same configurations as the previous experiments. Once a crash related to text input is spotted, we create an issue report by describing the bug, and report them to the app development team through the issue reporting system or email.

4.3 Baselines

Since there are hardly any existing approaches for the unusual input generation of mobile apps, we employ 18 baselines from various aspects to provide a thorough comparison.

First, we directly utilize *ChatGPT* [61] as the baseline. We provide the context information of the text input widgets (as described in Table 1 P1), and ask it to generate inputs that can make app crash.

Fuzzing testing and mutation testing can be promising techniques for generating invalid inputs, and we apply several related baselines. Feldt et al. [24] proposed a testing framework called *GoldTest*, which generates diverse test inputs for mobile apps by designing regular expressions and generation strategies. In 2017, they further proposed an invalid input generation method [58] based on probability distribution (PD) parameters and regular expressions, and we name this baseline as *PDinvalid*. Furthermore, we reuse the idea of traditional random-based fuzzing [13, 41], and develop a *RandomFuzz* for generating inputs for text widgets. In addition, based on the 50 buggy text inputs from the GitHub dataset in Section 3.3.1, we manually design 50 corresponding mutation rules to generate the invalid input, and name this baseline as *ruleMutator*.

Furthermore, we include the string analysis methods as the baselines, i.e., *OSTRICH* [15] and *Sloth* [14]. They aim at generating the strings that violate the constraints (e.g., string length, concatenation, etc), which is similar to our task. *OSTRICH*'s key idea [15] is to generate the test strings based on heuristic rules. *Sloth* [14] proposes to exploit succinct alternating finite-state automata as concise symbolic representations of string constraints.

There are constraint-based methods, i.e., *Mobolic* [8] and *TextExerciser* [27], which can generate diversified inputs for testing the app. For example, *TextExerciser* utilizes the dynamic hints to guide it in producing the inputs.

We also employ two methods (*RNNInput* [43] and *QTypist* [44]) which aim at generating valid inputs for passing the GUI page. In addition, we use the automated GUI testing tools, i.e., *Stoat* [64], *Droidbot* [39], *Ape* [26], *Fastbot* [12], *ComboDroid* [70], *TimeMachine* [23], *Humanoid* [40], *Q-testing* [56], which can produce inputs randomly or following rules to make app running automatically.

We design the script for each baseline to ensure that it can reach the GUI page with the text input widget, and run them in the same experimental environment (Android x64) to mitigate potential bias.

5 RESULTS AND ANALYSIS

5.1 Bugs Detection Performance (RQ1)

Table 2 presents the bug detection performance of InputBlaster. With the unusual inputs generated by InputBlaster, the bug detection rate is 0.78 (within 30 minutes), indicating 78% (28/36) of the bugs can be detected. In addition, the bugs can be detected with an average of 13.52 attempts, and the average bug detection time is 9.64 minutes, which is acceptable. This indicates the effectiveness of our approach in generating unusual inputs for testing the app, and facilitating the uncovering of bugs related to input widgets.

Figure 5 demonstrates examples of InputBlaster's generated unusual inputs and the inputs that truly trigger the crash. We can see that our proposed approach can generate quite diversified inputs which mutate the valid input from different aspects, e.g., for the price in the first example which should be a non-negative value, the

Zhe Liu[1,2],Chunyang Chen[3], Junjie Wang[1,2,*], Mengzhuo Chen[1,2], Boyu Wu[2,4], Zhilin Tian[5], Yuekai Huang[1,2], Jun Hu[1,2], Qing Wang[1,2,6,*]

Table 2: Result of bugs detection performance. (RQ1)

Method	Setting 1 (30 attempts) Bug(%)	Setting 1 (30 attempts) Attempt(#)	Setting 2 (30 minutes) Bug(%)	Setting 2 (30 minutes) Min(#)
InputBlaster	**0.72**	**13.52**	**0.78**	**9.64**
ChatGPT	0.25	25.91	0.28	23.28
Mutation or fuzzing methods				
GoldTest	0.08	29.22	0.08	28.73
PDinvalid	0.19	28.65	0.19	22.73
RandomFuzz	0.25	22.31	0.25	21.55
ruleMutator	0.28	21.42	0.28	20.53
String analysis methods				
Sloth	0.25	23.61	0.25	22.61
OSTRICH	0.22	24.14	0.22	23.41
Constraint-based methods				
Mobolic	0.17	25.83	0.17	25.09
TextExerciser	0.31	22.11	0.33	20.18
Valid input generation methods				
RNNInput	0.06	28.67	0.06	28.64
QTypist	0.08	27.78	0.11	27.31
Automated GUI testing methods				
Ape	0.08	28.11	0.11	26.88
DroidBot	0.06	28.39	0.06	28.34
Stoat	0.08	27.94	0.08	27.58
TimeMachine	0.11	26.92	0.11	26.69
ComboDroid	0.14	26.11	0.14	25.85
Q-testing	0.11	27.06	0.11	26.70
Humanoid	0.11	26.92	0.14	25.85

Notes: "Bug (%)" is the average bug detecting rate, "Attempt (#)" is the average number of unusual inputs before triggering the crash, "Min (#)" is the average running time (minutes) before triggering the crash.

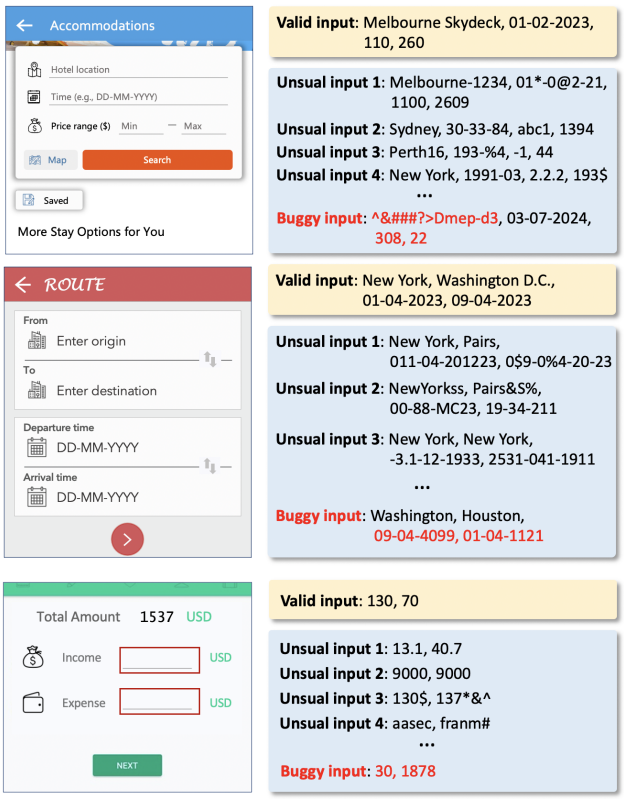


Figure 5: Example of InputBlaster's output.

generated unusual inputs range from negative values and decimals to various kinds of character strings. Furthermore, it is good at capturing the contextual semantic information of the input widgets and their associated constraints, and generating the violations accordingly. For example, for the minimum and maximum price in the first example, it generates the unusual inputs with the minimum larger than the maximum, and successfully triggers the crash.

We further analyze the bugs that could not be detected by our approach. A common feature is that they need to be triggered under specific settings, e.g., only under the user-defined setting, the input can trigger the crash, in the environment we tested, it may not have been possible to trigger a crash due to the lack of user-defined settings in advance. We have manually compared the unusual inputs generated by our approach with the ones in the issue reports. We find in all cases, InputBlaster can generate the satisfied buggy inputs within 30 attempts and 30 minutes, which further indicates its effectiveness.

Performance comparison with baselines. Table 2 also shows the performance comparison with the baselines. We can see that our proposed InputBlaster is much better than the baselines, i.e., 136% (0.78 vs. 0.33) higher in bug detection rate (within 30 minutes) compared with the best baseline TextExerciser. This further indicates the advantages of our approach. Nevertheless, the TextExerciser can only utilize the dynamic hints in input generation which covers a small portion of all situations, i.e., a large number of input widgets donot involve such feedback.

Without our elaborate design, the raw ChatGPT demonstrates poor performance, which further indicates the necessity of our approach. In addition, the string analysis methods, which are designed specifically for string constraints, would fail to work for mobile apps. In addition, since the input widgets of mobile apps are more diversified (as shown in Section 2.1.2) compared with the string, the heuristic analysis or finite-state automata techniques in the string analysis methods might be ineffective for our task. The baselines for automated GUI testing or valid text input generation are even worse, since their main focus is to increase the coverage through generating valid inputs. This further implies the value of our approach for targeting this unexplored task.

5.2 Ablation Study (RQ2)

5.2.1 ***Contribution of Modules.*** Table 3 shows the performance of InputBlaster and its 2 variants respectively removing the first and third module. In detail, for *InputBlaster w/o validInput* (i.e., without Module 1), we provide the information related to the input widgets (as Table 1 P1) to the LLM in Module 2 and set other information from Module 1 as "null". For *InputBlaster w/o enrichExamples* (i.e., without Module 3), we set the examples from Module 3 as "null" when querying the LLM. Note that, since Module 2 is for generating the unusual inputs which is indispensable for this task, hence we donot experiment with this variant.

Table 3: Contribution of different modules (RQ2)

Method	30 attempts		30 minutes	
	Bug(%)	Attempt(#)	Bug(%)	Min(#)
InputBlaster (Base)	**0.72**	**13.52**	**0.78**	**9.64**
w/o Module 1	0.31	22.75	0.39	19.15
w/o Module 3	0.47	22.19	0.53	20.15

Notes: The two variants respectively denote InputBlaster removing module 1 (valid input generation) and module 3 (enriched examples in prompt).

Table 4: Contribution of different sub-modules (RQ2)

Method	30 attempts		30 minutes	
	Bug(%)	Attempt(#)	Bug(%)	Min(#)
InputBlaster (Base)	**0.72**	**13.52**	**0.78**	**9.64**
w/o inferCons	0.53	19.94	0.56	15.11
w/o mutateRule	0.36	21.31	0.42	20.71
w/o feedback	0.58	16.64	0.58	14.40
w/o generator	0.61	16.86	0.36	24.37
w/o retriExample	0.56	19.11	0.56	23.44

Notes: The five variants respectively denote InputBlaster removing inferred constraint, mutation rule, test execution feedback, test generator, retrieved examples of buggy input.

Table 5: Result of different number of examples. (RQ2)

Exmample (#)	Setting 1 (30 attempts)		Setting 2 (30 minutes)	
	Bug(%)	Attempt(#)	Bug(%)	Min(#)
1	0.50	20.19	0.50	22.98
2	0.53	19.36	0.56	18.31
3	0.61	16.86	0.64	14.93
4	0.69	14.36	0.69	11.14
5(InputBlaster)	**0.72**	**13.52**	**0.78**	**9.64**
6	0.61	16.86	0.58	15.48
7	0.53	19.69	0.53	17.15
8	0.44	21.86	0.42	20.47
9	0.38	23.53	0.36	22.34
10	0.36	24.36	0.31	23.81

We can see that InputBlaster's bug detection performance is much higher than all other variants, indicating the necessity of the designed modules and the advantage of our approach.

Compared with InputBlaster, *InputBlaster w/o validInput* results in the largest performance decline, i.e., 50% drop (0.39 vs. 0.78) in bug detection rate within 30 minutes. This further indicates that the generated valid inputs and inferred constraints in Module 1 can help LLM understand what the correct input looks like and generate the violated ones.

InputBlaster w/o enrichExamples also undergoes a big performance decrease, i.e., 32% (0.53 vs. 0.78) in bug detection rate within 30 minutes, and the average testing time increases by 109% (9.64 vs. 20.15). This might be because without the examples, the LLM would spend more time understanding user intention and criteria for what kinds of answers are wanted.

5.2.2 ***Contribution of Sub-modules.*** Table 4 further demonstrates the performance of InputBlaster and its 5 variants. We remove each sub-module of the InputBlaster in Figure 3 separately, i.e., inferred constraint, mutation rule, text execution feedback, test generator and retrieved examples of buggy input. For removing the test generator, we directly let the LLM generate the unusual inputs, and for removing retrieved examples, we use the random selection method. For other variants, we set the removed content as "null".

The experimental results demonstrate that removing any of the sub-modules would result in a noticeable performance decline, indicating the necessity and effectiveness of the designed sub-modules.

Removing the mutation rules (*InputBlaster w/o-mutateRule*) have the greatest impact on the performance, reducing the bug detection rate by 50% (0.36 vs. 0.72 within 30 attempts). Remember that, InputBlaster first lets the LLM to generate the mutation rules (how to mutate the valid inputs), then asks it to produce the test generator following the mutation rule. With the generated mutation rules serving as the reasoning chain, the unusual input generation can be more effective, which further proves the usefulness of our design.

We also notice that, when removing the test generator (*InputBlaster w/o-generator*), the bug detection rate does not drop much (0.72 vs. 0.61) when considering 30 attempts, yet it declines a lot (0.78 vs. 0.36) when considering 30 minutes of testing time. This is because our proposed approach lets the LLM produce the test generator which can yield a batch of unusual inputs. This means interacting with the LLM once can generate multiple outcomes. However, if asking the LLM to directly generates unusual inputs (i.e., *InputBlaster w/o-generator*), it requires interacting with LLM frequently, and could be quite inefficient. This further demonstrates we formulate the problem as producing the test generator task is efficient and valuable.

In addition, randomly selecting the examples (*InputBlaster w/o-retriExample*) would also largely influence the performance, and decrease the bug detection rate by 22% (0.56 vs. 0.72 within 30 attempts). This indicates that by providing similar examples, the LLM can quickly think out what should the unusual inputs look like. Nevertheless, we can see that, compared with the variant without enriched examples in prompt (Table 3), the randomly selected examples do take effect (0.47 vs 0.56 in bug detection rate within 30 attempts), which further indicates the demonstration can facilitate the LLM in producing the required output.

5.2.3 ***Influence of Different Number of Examples.*** Table 5 demonstrates the performance under the different number of examples provided in the prompt.

We can see that the number of detected bugs increases with more examples, reaching the highest bug detection rate with 5 examples. And after that, the performance would gradually decrease even increasing the examples. This indicates that too few or too many examples would both damage the performance, because of the tiny information or the noise in the provided examples.

5.3 Usefulness Evaluation (RQ3)

Table 6 shows all bugs spotted by Ape integrated with our InputBlaster, and more detailed information on detected bugs can be seen in our website[3]. For the 131 apps, InputBlaster detects 43 bugs in 32 apps, of which 37 are newly-detected bugs. Furthermore, these new bugs are not detected by the Ape without InputBlaster.

We submit these 37 bugs to the development team, and 28 of them have been fixed/confirmed so far (21 fixed and 7 confirmed), while the remaining are still pending (none of them is rejected). This further indicates the effectiveness and usefulness of our proposed InputBlaster in bug detection.

When confirming and fixing the bugs, some Android app developers express thanks such as "*Very nice! You find an invalid input we*

Table 6: Confirmed or fixed bugs. (RQ3)

Id	APP Name	Category	Download	Status
1	OTOMU	Music	100M+	fixed
2	KWork	Tool	50M+	confirmed
3	NoxSecu	Tool	50M+	fixed
4	EarnMon	Finance	50M+	fixed
5	RewardM	Finance	50M+	confirmed
6	AttaPOl	Tool	10M+	confirmed
7	ISAY	Commun	10M+	fixed
8	Ipsos	Commun	10M+	fixed
9	MediaFire	Product	5M+	confirmed
10	DRBUs	Navig	500K+	fixed
11	MyTransp	Travel	500K+	fixed
12	MMDR	Utilities	500K+	fixed
13	Genting	Travel	500K+	fixed
14	Fair	Health	500K+	confirmed
15	ClassySha	Tool	500K+	fixed
16	Linphone	Commun	50K+	confirmed
17	IvyWall	Finance	50K+	fixed
18	Monefy	Finance	50K+	fixed
19	Spend	Finance	50K+	fixed
20	NYBA	Tool	50K+	fixed
21	OneTravel	Travel	50K+	fixed
22	Passpor	Travel	50K+	fixed
23	Thatch	Travel	50K+	confirmed
24	Click	Utilities	50K+	fixed
25	GGBN	Utilities	50K+	fixed
26	Vived	Utilities	50K+	fixed
27	Bizbazar	Finance	50K+	fixed
28	Flowx	Tool	50K+	fixed

thought was too insignificant to cause crashes."(i.e., Ipsos). Furthermore, some developers also express their thought about the buggy text input "*Handling different inputs can be tricky, and I admit we couldn't test for every possible scenario. It has given me a fresh appreciation for the complexity of user inputs and the potential bugs they can introduce.* "(i.e., DRBUs). Some developers also present valuable suggestions to facilitate the further improvement of InputBlaster. For example, some of them hope that we can find the patterns of these bugs and design repair methods.

6 DISCUSSION AND THREATS TO VALIDITY

6.1 Generality Across Platforms

The primary idea of InputBlaster is to generate unusual inputs for text widgets with the context information when running the apps. Although we only experiment with Android mobile apps, since other platforms have these similar types of information, InputBlaster can be used to conduct the testing of input widgets for other platforms. We conduct a small-scale experiment for another two popular platforms, and experiment on 10 iOS apps with 15 bugs and 10 Web apps with 18 bugs, with details on our website. Results show that InputBlaster's bug detection rate is 80% for iOS apps and 78% for Web apps within 30 minutes testing time. This further demonstrates the generality and usefulness of InputBlaster, and we will conduct more thorough experiments in the future.

6.2 Threats of Validity

The first threat concerns the representativeness of the experimental apps. We have selected popular and active apps which can partially reduce this threat.

The second threat relates to the baseline selection. Since there are hardly any existing approaches for the unusual input generation of mobile apps, we employ 18 approaches from various aspects for a thorough comparison. There are inputs generation techniques for Web apps [5, 6, 65, 66], yet because they need to analyze the web code which is different from mobile apps considering the different rendering mechanism, and cannot be directly applied in our task, hence we don't include them as the baselines.

The third threat is that we only focus on the crash bugs, since they cause more serious effects and can be automatically observed, and existing studies also only explore this type of bug [38, 40, 56].

The fourth threat might lie in the process of manual categorization in Section 2.1.2. The process involves multiple practitioners and double-checking for the final decision. Also note that, the derived categorization is only for illustration, rather than serving as the ground truth for evaluation.

The Fifth threat may exist in the uncertainty of LLM output results. LLM may not generate the corresponding output as expected, and we also design in-context learning and feedback mechanisms to ensure the output format and content of LLM.

Last but not least, InputBlaster gradually builds the example dataset (Section 3.3.1) as the test goes on. This indicates the performance can be influenced by the testing order, e.g., when arranged in the first place, the crash could not be detected, yet when arranged after 10 apps are tested, the crash can be revealed, since the example dataset has accumulated more knowledge. In this paper, we use a random order of the experimental apps and would explore more in the future.

7 RELATED WORK

Testing Related with Text Inputs. There have been many automated GUI testing techniques for mobile apps [7, 9, 12, 22, 23, 26, 39, 47–49, 51, 52, 60, 64, 70], yet they mainly focus on how to plan the exploration paths to fully cover the app activities and states. There are also studies [27, 43, 44] that aim at generating valid inputs to pass the GUI pages and are used to enrich the automated testing tools for higher coverage. None of them can conduct the testing of text input widgets.

For Web apps, SWAT [5] and AWET [65] generated the unusual inputs based on the pre-defined template. ACTEve [6] and S3 [66] first used symbolic execution to extract input constraints in the source code and then employ a solver to generate the inputs. They need to analyze the web code and can't be directly applied to Android apps which have quite different rendering mechanisms. In addition, some constraints are dynamically generated (as shown in Section 2.1.2), and couldn't be extracted from the source code.

There are some string analysis methods for generating the strings that violate the constraints (e.g., string length) [14, 15, 18, 28, 33, 34, 37, 42, 67]. Although they are effective for string constraints, yet the inputs of mobile apps are more diversified, and they cannot work well in our task.

LLM for Software Engineering. With the breakthrough of LLMs, studies have proposed to explore how LLMs can be used to assist developers in a variety of tasks, such as code generation [57, 72], program repair [29, 31, 55], and code summarization [4, 72]. There is also a growing trend of applying LLM for software testing, e.g., fuzzing deep learning libraries [20], unit test generation [36], bug reproduction [32], valid input generation [44], etc, and achieves

significant performance improvement. This work explores a different task, i.e., unusual text input generation for mobile apps, which provides new insights into how LLM can enhance the software testing practice.

8 CONCLUSION

Automated testing is crucial for helping improve app quality. Despite the dozens of mobile app GUI testing techniques, how to automatically generate the diversified unusual text inputs for fully testing mobile apps remains a challenge. This paper proposes InputBlaster which leverages the LLM to produce the unusual inputs together with the mutation rules which serve as the reasoning chains. It formulates the unusual inputs generation problem as a task of producing a set of test generators, each of which can yield a batch of unusual text inputs under the same mutation rule. The evaluation is conducted for both effectiveness and usefulness, with 136% higher bug detection rate than the best baselines, and uncovering 37 new crashes.

In the future, we plan to further analyze the root causes and repair strategy of these input-related bugs, and design automated bug repair methods.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China Grant No.62232016, No.62072442 and No.62272445, Youth Innovation Promotion Association Chinese Academy of Sciences, Basic Research Program of ISCAS Grant No. ISCAS-JCZD-202304, and Major Program of ISCAS Grant No. ISCAS-ZD-202302.

REFERENCES

[1] 2022. Crash bug text. https://www.theguardian.com/technology/iphone-crash-bug-text-imessage-ios.
[2] 2022. Crash bug text in ios. https://tech.hindustantimes.com/tech/news/be-careful-a-new-text-bomb-is-making-whatsapp-crash-and-will-hang-your-phone-71599532897852.html.
[3] 2022. F-droid. https://f-droid.org/.
[4] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. *ASE* (2022).
[5] Nadia Alshahwan and Mark Harman. 2011. Automated web application testing using search based software engineering. In *ASE*. IEEE, 3–12.
[6] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.
[7] Yauhen Leanidavich Arnatovich, Minh Ngoc Ngo, Tan Hee Beng Kuan, and Charlie Soh. 2016. Achieving high code coverage in android ui testing via automated widget exercising. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 193–200.
[8] Yauhen Leanidavich Arnatovich, Lipo Wang, Ngoc Minh Ngo, and Charlie Soh. 2018. Mobolic: An automated approach to exercising mobile application GUIs using symbiosis of online testing technique and customated input generation. *Software: Practice and Experience* 48, 5 (2018), 1107–1142.
[9] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 641–660.
[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[12] Tianqin Cai, Zhao Zhang, and Ping Yang. 2020. Fastbot: A Multi-Agent Model-Based Test Generation System Beijing Bytedance Network Technology Co., Ltd.. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*. 93–96.
[13] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. 2018. A systematic review of fuzzing techniques. *Computers & Security* 75 (2018), 118–137.
[14] Taolue Chen, Alejandro Flores-Lamas, Matthew Hague, Zhilei Han, Denghang Hu, Shuanglong Kan, Anthony W Lin, Philipp Rümmer, and Zhilin Wu. 2022. Solving string constraints with Regex-dependent functions through transducers with priorities and variables. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–31.
[15] Taolue Chen, Matthew Hague, Jinlong He, Denghang Hu, Anthony Widjaja Lin, Philipp Rümmer, and Zhilin Wu. 2020. A decision procedure for path feasibility of string manipulating programs with integer data type. In *Automated Technology for Verification and Analysis: 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19–23, 2020, Proceedings*. Springer, 325–342.
[16] Ting Chen, Simon Kornblith, Kevin Swersky, Mohammad Norouzi, and Geoffrey E Hinton. 2020. Big self-supervised models are strong semi-supervised learners. *Advances in neural information processing systems* 33 (2020), 22243–22255.
[17] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).
[18] Joel D Day, Thorsten Ehlers, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. 2019. On solving word equations using SAT. In *Reachability Problems: 13th International Conference, RP 2019, Brussels, Belgium, September 11–13, 2019, Proceedings 13*. Springer, 93–106.
[19] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *UIST*.
[20] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT. *ISSTA* (2023).
[21] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing deep-learning libraries via automated relational API inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 44–56.
[22] Android Developers. 2012. Ui/application exerciser monkey.
[23] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of android apps. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 481–492.
[24] Robert Feldt and Simon Poulding. 2013. Finding test data with specific properties via metaheuristic search. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 350–359.
[25] Shivam Garg, Dimitris Tsipras, Percy S Liang, and Gregory Valiant. 2022. What can transformers learn in-context? a case study of simple function classes. *Advances in Neural Information Processing Systems* 35 (2022), 30583–30598.
[26] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280.
[27] Yuyu He, Lei Zhang, Zhemin Yang, Yinzhi Cao, Keke Lian, Shuai Li, Wei Yang, Zhibo Zhang, Min Yang, Yuan Zhang, et al. 2020. TextExerciser: feedback-driven text input exercising for android applications. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1071–1087.
[28] Lukav Holik, Petr Jank, Anthony W Lin, Philipp Rmmer, and Tom Vojnar. 2017. String constraints with concatenation and transducers solved efficiently. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–32.
[29] Yang Hu, Umair Z Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-factoring based program repair applied to programming assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 388–398.
[30] Text input. 2022. Introduction about text input on Android Developer website. https://developer.android.google.cn/reference/android/widget/EditText?hl=en.
[31] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. *ICSE* (2023).
[32] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. *ICSE* (2023).
[33] Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J Guo, Pieter Hooimeijer, and Michael D Ernst. 2013. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21, 4 (2013), 1–28.
[34] Sebastian Krings, Joshua Schmidt, Patrick Skowronek, Jannik Dunkelau, and Dierk Ehmke. 2020. Towards constraint logic programming over strings for test data generation. In *Declarative Programming and Knowledge Management: Conference on Declarative Programming, DECLARE 2019, Unifying INAP, WLP, and WFLP, Cottbus, Germany, September 9–12, 2019, Revised Selected Papers 22*. Springer, 139–159.

[35] Misha Laskin, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas. 2020. Reinforcement learning with augmented data. *Advances in neural information processing systems* 33 (2020), 19884–19895.
[36] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *ICSE*.
[37] Guodong Li and Indradeep Ghosh. 2013. PASS: String solving with parameterized array and interval automaton. In *Hardware and Software: Verification and Testing: 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings 9*. Springer, 15–31.
[38] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: A Lightweight UI-Guided Test Input Generator for Android *(ICSE-C '17)*.
[39] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 23–26.
[40] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: a deep learning-based approach to automated black-box Android app testing. In *ASE*. IEEE, 1070–1073.
[41] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.
[42] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2014. A DPLL (T) theory solver for a theory of strings and regular expressions. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer, 646–662.
[43] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. 2017. Automatic text input generation for mobile testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 643–653.
[44] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Fill in the Blank: Context-aware Automated Text Input Generation for Mobile GUI Testing. *arXiv preprint arXiv:2212.04732* (2022).
[45] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl Eyes: Spotting UI Display Issues via Visual Understanding. In *ASE*. IEEE. https://doi.org/10.1145/3324884.3416547
[46] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Nighthawk: Fully Automated Localizing UI Display Issues via Visual Understanding. *IEEE Transactions on Software Engineering* (2022), 1–16. https://doi.org/10.1109/TSE.2022.3150876
[47] Zhe Liu, Chunyang Chen, Junjie Wang, Yuhui Su, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Ex pede Herculem: Augmenting Activity Transition Graph for Apps via Graph Convolution Network. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1983–1995.
[48] Zhe Liu, Chunyang Chen, Junjie Wang, Yuhui Su, and Qing Wang. 2022. NaviDroid: a tool for guiding manual Android testing via hint moves. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 154–158.
[49] Zhe Liu, Chunyang Chen, Junjie Wang, and Qing Wang. 2022. Guided Bug Crush: Assist Manual GUI Testing of Android Apps via Hint Moves. In *CHI 2022*. https://doi.org/10.1145/3491102.3501903
[50] Li Lucy and David Bamman. 2021. Gender and representation bias in GPT-3 generated stories. In *Proceedings of the Third Workshop on Narrative Understanding*. 48–55.
[51] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 224–234.
[52] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 94–105.
[53] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *Computer Science* (2013).
[54] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the Role of Demonstrations: What Makes In-Context Learning Work? *arXiv preprint arXiv:2202.12837* (2022).
[55] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*.
[56] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–164.
[57] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *ICLR* (2022).
[58] Simon Poulding and Robert Feldt. 2017. Generating controllably invalid and atypical inputs for robustness testing. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 81–84.
[59] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67.
[60] Vaibhav Rastogi, Yan Chen, and William Enck. 2013. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*. 209–220.
[61] J Schulman, B Zoph, C Kim, J Hilton, J Menick, J Weng, JFC Uribe, L Fedus, L Metz, M Pokorny, et al. 2022. ChatGPT: Optimizing language models for dialogue.
[62] Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering* 25, 4 (1999), 557–572.
[63] Mike Sharples. 2022. Automated Essay Writing: An AIED Opinion. *International Journal of Artificial Intelligence in Education* (2022), 1–8.
[64] 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*. 245–256.
[65] Nezih Sunman, Yiğit Soydan, and Hasan Sözer. 2022. Automated web application testing driven by pre-recorded test cases. *Journal of Systems and Software* (2022), 111441.
[66] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1232–1243.
[67] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2017. Model counting for recursively-defined strings. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II 30*. Springer, 399–418.
[68] UIAutomator. 2021. Python wrapper of Android uiautomator test tool. https://github. com/xiaocong/uiautomator.
[69] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* (2017).
[70] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. Combodroid: generating high-quality test inputs for android apps via use case combinations. In *ICSE*. 469–480.
[71] Zhengyuan Yang, Zhe Gan, Jianfeng Wang, Xiaowei Hu, Yumao Lu, Zicheng Liu, and Lijuan Wang. 2022. An empirical study of gpt-3 for few-shot knowledge-based vqa. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 3081–3089.
[72] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 39–51.
[73] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).