

White Paper

AI War: Unleashing the Power of LLM and Prompt Engineering for AI Framework Fuzz

Sihang Hu, Zhihui Lin, Tianxiang Li

ABSTRACT

During the AI boom, fuelled by applications such as ChatGPT, thousands of models were trained, new applications were built on top of those models, a much wider audience was reached, the attack surface was expanded, and the overall landscape of the AI security transformed. PyTorch and TensorFlow are the two most popular AI frameworks, serving as the very foundation of building and training models today. Security issues, such as the insecure deserialisation with Python Pickle module that PyTorch relies on [1], hidden in these frameworks pose serious security threats to the entire AI ecosystem.

Unsurprisingly, upon rigorous analysis of 400+ security advisories from TensorFlow [2], our findings suggest that the majority of vulnerabilities previously discovered in standard fuzzing campaign leveraged failures in individual APIs. That is, the state-of-the-art fuzzer only scratched the surface without learning about API dependencies and contextual semantics.

In this talk, we open up a new paradigm by introducing the first AI framework fuzzer TaiSi and advocate the importance of prompt engineering. TaiSi harnesses the power of dual LLMS using a set of fine-tuned prompts tailored to the proposed security prompt framework TaiBai, under which the first LLM deals with generating a large number of high-quality seeds, and the second LLM performs a cost-effective mutation of them. The results demonstrate that our proposed tool outperforms naïve fuzzers without the ability to understand API dependencies and contextual semantics. To date, our fuzzer had discovered more than 100 weaponisable security bugs and vulnerabilities buried deep in several AI frameworks. Furthermore, thanks to its modularity, the proposed tool can be efficiently applied to other domains for context-aware vulnerability detection.

1 INTRODUCTION

The past few months witnessed a seismic shift in the realm of Artificial Intelligence (AI), predominantly powered by the emergence and explosive growth of Large Language Models (LLMs). From building AI applications to facilitating training and inference processes, AI frameworks such as PyTorch and TensorFlow serve as the backbone for the LLM revolution, thereby underlining the importance of their security. In fact, they possess potential security vulnerabilities that present significant threats to the broader AI ecosystem. An illustration of such insecurity is the Python Pickle module, on which PyTorch heavily relies, that has known issues with insecure deserialization [1].

Enhancing the robustness of AI frameworks hinges on the identification and rectification of hidden bugs within their thousands of APIs. Conventional test cases often falter at this hurdle due to complex dependencies and exceptional boundary conditions, presenting a significant challenge to securing these frameworks. Responding to this challenge, the security research community has proposed several fuzzing tools. For instance, Baidu Security presented AIModel-Mutator Fuzzer [3] at Blackhat Europe 2021, which unearthed many security flaws resulting from insufficient API verification. Other representative security organisations such as the Aivul Team from Qihoo 360 and the Secure Systems Lab from Brown University have also found a large number of related bugs, with the number of disclosed security advisories in TensorFlow’s project page reaching 400+ as of 25 July, 2023 [2]. However, these advancements barely scratch the surface of the problem, as they focus primarily on individual APIs, while overlooking the semantics and dependencies in chained API sequences. Namely, the ability to gain an understanding of API dependencies and contextual semantics is the key to avoid surface-level fuzzing that do not fully capture the underlying vulnerabilities present in AI frameworks.

In an evolution mirroring that of traditional fuzzers like AFL++ [4], it is time for AI framework fuzzers to evolve as well, by unleashing the prowess of LLMs. After learning from billions of code snippets, LLMs can generate complex and chained API calls of professional quality. To enable this for AI framework fuzzing, the underlying LLM first need to learn the complex syntax and semantic constraints that govern the chaining of APIs. This introduces the concept of Prompt Engineering and underscores its significance in the context of LLM-empowered fuzzing. In recognition of this need, we introduce TaiBai, a security prompt framework specifically tailored for the fuzzing domain. Building upon TaiBai, we further propose the first integrated AI framework fuzzer, TaiSi, that harnesses the capabilities of general LLMs (e.g., Llama [5], Llama 2 [6]) and code infilling LLMs (e.g., starCoder [7], InCoder [8]) to manage all stages of the fuzzing lifecycle, including sample generation, scheduling, mutation, execution, and logging.

By employing dual large language models (LLMs), TaiSi pushes the boundaries of what we know as conventional fuzzing. The first LLM (Llama 2) is used for the generation of a multitude of high-quality seeds, while the second is tasked with their cost-effective mutation, all under the proposed security prompt framework, TaiBai. It is also worth noting that a variant of the

Adversarial Multi-Armed Bandit (VAMAB) [9] algorithm is adopted in TaiSi to flexibly and intelligently select the seeds to be mutated with appropriate mutation strategy. Our results affirm the superior performance of TaiSi over traditional AI framework fuzzers, demonstrating its ability to understand API dependencies and contextual semantics. This pioneering approach has allowed us to discover over 100 weaponisable security bugs and vulnerabilities hidden within several AI frameworks. In addition to this, the tool’s modularity allows for efficient adaptation to other domains, showcasing its potential for context-aware vulnerability detection.

The subsequent sections of this paper detail our prompt framework, fuzz framework and the broader implications for the AI and security communities. Our ultimate goal is to reinforce the importance of robust security mechanisms in AI frameworks, contributing to the ongoing dialogue on AI security. This study is a call to arms for researchers and developers alike, urging for a more concerted effort in securing our AI future.

2 PROMPT FRAMEWORK

This section focuses on the Security Prompt Framework TaiBai, crucial for the successful integration of LLMs in the fuzzing process. We provide insight into the essence of prompt engineering, an understanding of the variations in the prompt, and a thorough exploration of the TaiBai Prompt Framework specifically designed for AI Framework Fuzzing. It is noteworthy that the name TaiBai is inspired by a famous ancient Chinese poet, Li Taibai. He is regarded as the immortal poet, a title that reflects his extraordinary talent and influence in the history of Chinese literature. We aspire to be the immortal poet of prompts in the security field.

2.1 THE IMPORTANCE OF PROMPT ENGINEERING

Prompt engineering is the art of designing effective inputs for LLMs, such as ChatGPT, to elicit specific outputs. It is a crucial skill for LLM applications, as it can enhance the quality, relevance, and consistency of the model’s responses. Proudly speaking, we are among the early adopters of prompt engineering, and we have explored its potential and limitations in fuzzing and other various security scenarios. In essence, prompt engineering involves creating and tuning the initial set of instructions given to an LLM to perform a task. The quality of these prompts can significantly impact the model’s ability to generate seeds and samples used in fuzzing campaigns. Unlike traditional fuzzing tools where each task (e.g., seed generation and mutation) requires explicit programming, LLMs are designed to comprehend a broad spectrum of tasks, with the prompt acting as the guiding input. Just as a different piece of function is required for a different task in traditional programming, different prompts are necessary for different fuzzing tasks performed by LLMs. Therefore, the ability to engineer a high-quality prompt that succinctly and accurately communicates the intended subtask is vital for harnessing the full potential of LLMs in AI framework fuzzing.

It is of note that even slight variations in prompts can lead to considerable differences in the seeds and samples generated by our chosen LLM. For example, the choice of wording, the amount of API context provided, and the explicitness of the instructions can all influence the

nature of the generated/mutated seeds. Hence, prompt engineering is not merely about creating a prompt but also involves fine-tuning it iteratively based on model performance.

2.2 DESIGN OF TAIBAI

The TaiBai Prompt Framework has been designed with the abovementioned principles in mind, to make the fuzzing process more effective, efficient, and contextually aware.

The first critical component of TaiBai is a comprehensive prompt library, cataloguing a wide range of prompts for different fuzzing tasks. Each prompt is carefully designed with to represent a specific aspect of the fuzzing process such as the generation of seeds. Given a template prompt from the prompt library, TaiBai then employs a process of Prompt Augmentation with the assistance of dedicated API/code miners. This involves enhancing the base task with additional context and constraints, tailored to the specific requirements of the fuzzing target. This may include, for example, specifics of the API under test, API usage example or requirements for the output. To give our reader an intuitive example, Figure 1 demonstrates how our framework can be applied to construct a sophisticated prompt for the initial seed generation of a fuzzing campaign. The figure shows the actions performed by the user, the LLM and the code logic in the prompt construction process.

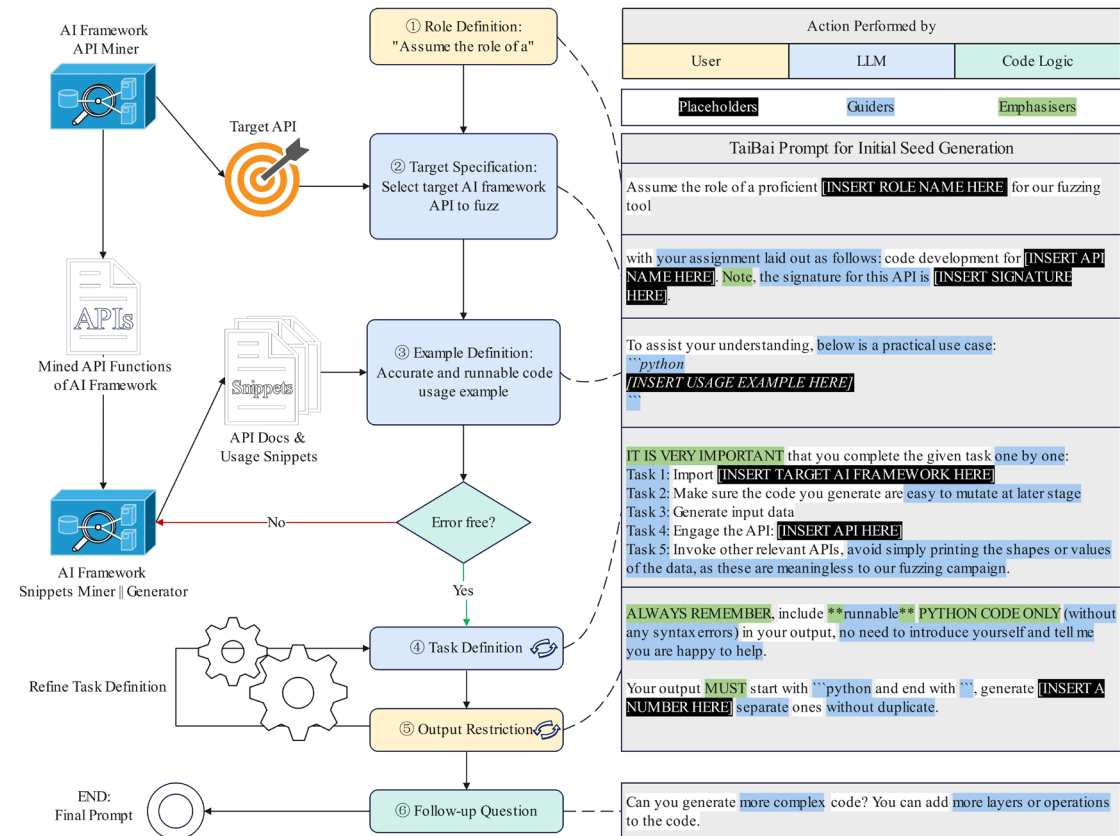


Figure 1. An illustration of using TaiBai prompt framework to construct a prompt for the initial seed generation of a fuzzing campaign, where AI framework API/snippets miner leverages search tools of LangChain, snippets generator leverages Llama2.

As our readers may have noticed, the constructed prompt is composed of five core components,

(1) Role Definition, (2) Target Specification, (3) Example Definition, (4) Task Definition, and (5) Output Restriction, to encapsulate the underlying semantics and complexities of the fuzzing domain. Role Definition component provides the role within which the LLM operates. It is essentially a constraint of the persona/function that the LLM is expected to embody for the fuzzing task. For example, the LLM might be defined as a seasoned developer for AI frameworks or an advanced seed generator. This definition provides a valuable baseline, helping to guide the LLM's responses to subsequent prompts; Target Specification sets the scene by explicitly defining the target AI framework the fuzzing is focused on. This could be a particular API sequence or a specific element within the chosen AI framework. By providing the LLM with a clear understanding of the target, this component enhances the precision of the fuzzing campaign; Example Definition component serves as a valuable learning source for the LLM. By showcasing an instance of a well-constructed usage scenario, the LLM learns to model similar but novel scenarios. This approach, harnessing the incredible pattern-recognition capabilities of LLMs, greatly enhances the quality and relevance of the generated fuzzing seeds; The fourth component, Task Definition, outlines the explicit tasks the LLM needs to perform. This could range from importing relevant packages to invoking chained APIs. The task definition acts as a guiding beacon for the LLM, setting clear objectives and providing a tangible goal for its actions; Lastly, Output Restriction specifies the boundaries of the acceptable outputs from the LLM. This might include defining the structure of the output, or specifying the level of novelty required. This final component ensures that the output from the LLM meets the specific needs of the fuzzing campaign, helping to ensure its success.

At times, the user may find the generated seed too simple or inadequate for their purpose. In such situations, we can pose a follow-up question to the model to prompt it to generate an initial seed with enhanced quality. In essence, TaiBai empowers LLMs to generate complex and semantically-rich API sequences that are extremely valuable for AI framework fuzzing.

TaiBai Prompt for Initial Seed Generation

Role Definition	Assume the role of a proficient 'initial seed generator' for our fuzzing tool, with your assignment laid out as
Target Specification	follows: code development for [INSERT API NAME HERE]. Note, the signature for this API is [INSERT SIGNATURE HERE]. To assist your understanding, below is a practical use case:
Example Definition	```python [INSERT USAGE EXAMPLE HERE] ```
Task Definition	IT IS VERY IMPORTANT that you complete the given task one by one: Task 1: Import [INSERT TARGET AI FRAMEWORK HERE] Task 2: Make sure the code you generate are easy to mutate at later stage Task 3: Generate input data Task 4: Engage the API: [INSERT API HERE] Task 5: Invoke other relevant APIs, avoid simply printing the shapes or values of the data, as these are meaningless to our fuzzing campaign.
Output Restriction	ALWAYS REMEMBER, include **runnable** PYTHON CODE ONLY (without any syntax errors) in your output, no need to introduce yourself and tell me you are happy to help.

Your output **MUST** start with `python` and end with `'''`, generate `[INSERT A NUMBER HERE]` separate ones without duplicate.

Listing 1. Prompt constructed using TaiBai for initial seed generation, black backgrounded tokens serve as placeholders, blue backgrounded tokens serve as guiders, green backgrounded tokens serve as emphasisers.

2.3 AUTONOMOUS PROMPT MUTATION

As we continue to explore the direction of AI for Security, we firmly believe that autonomous AI agents are the future. In line with this vision, we have also constructed a prompt (Listing 2) to support Autonomous Prompt Mutation as part of our TaiBai framework. This feature empowers the selected LLM to autonomously mutate the given prompts, creating a virtuous cycle of learning and adapting using live API documentations. This capability not only fosters diversity and novelty in the fuzzing scenarios, but also significantly accelerates the fuzzing process, thereby augmenting the overall effectiveness of our AI framework fuzzing toolkit.

TaiBai Prompt for Autonomous Prompt Mutation

Role Definition	Assume the role of a proficient 'prompt mutator', with your assignment laid out as follows: explore the latest
Target Specification	version of <code>[TensorFlow/Pytorch/...]</code> 's API and documentation, use information given in <code>square brackets</code> of the following prompt template as hint and <code>infill</code> all of them. <code>===TART OF THE PROMPT TEMPLATE===</code> <code>{To save space, please refer to the prompt given in Listing 2.1.1.}</code> <code>===END OF THE PROMPT TEMPLATE===</code>
Example Definition	To assist your understanding, below is a mutated example based on the given prompt template: <code>===START OF AN EXAMPLE OUTPUT===</code> <code>{To save space, please refer to the prompt given in Listing 3.1.1.}</code> <code>===END OF AN EXAMPLE OUTPUT===</code>
Task Definition	IT IS VERY IMPORTANT that you complete the given task one by one: Task 1: Replace <code>[INSERT API NAME HERE]</code> with the a <code>existing</code> API name you found Task 2: Replace <code>[INSERT SIGNATURE HERE]</code> with the <code>accurate</code> API signature Task 3: Replace <code>[INSERT USAGE EXAMPLE HERE]</code> with a <code>working</code> usage example Task 4: Replace <code>[INSERT API HERE]</code> with the target API you choose Task 5: Replace <code>[INSERT A NUMBER HERE]</code> with a number between 1-10
Output Restriction	ALWAYS REMEMBER , only include MUTATED PROMPT in your output, <code>no need to introduce yourself</code> and <code>tell me you are happy to help</code> . Your output MUST start with <code>===MUTATED PROMPT START===\n</code> and end with <code>\n===MUTATED PROMPT END===</code> , generate <code>[INSERT A NUMBER HERE]</code> separate ones without duplicate.

Listing 2. Prompt constructed using TaiBai for autonomous prompt mutation, black backgrounded tokens serve as placeholders, blue backgrounded tokens serve as guiders, green backgrounded tokens serve as emphasisers.

Please note that while the prompting strategies and methodologies presented here are grounded in AI framework fuzzing, they hold broader applicability across various security field. We encourage readers to adopt and adapt TaiBai to their respective domains for innovative applications.

3 FUZZING FRAMEWORK

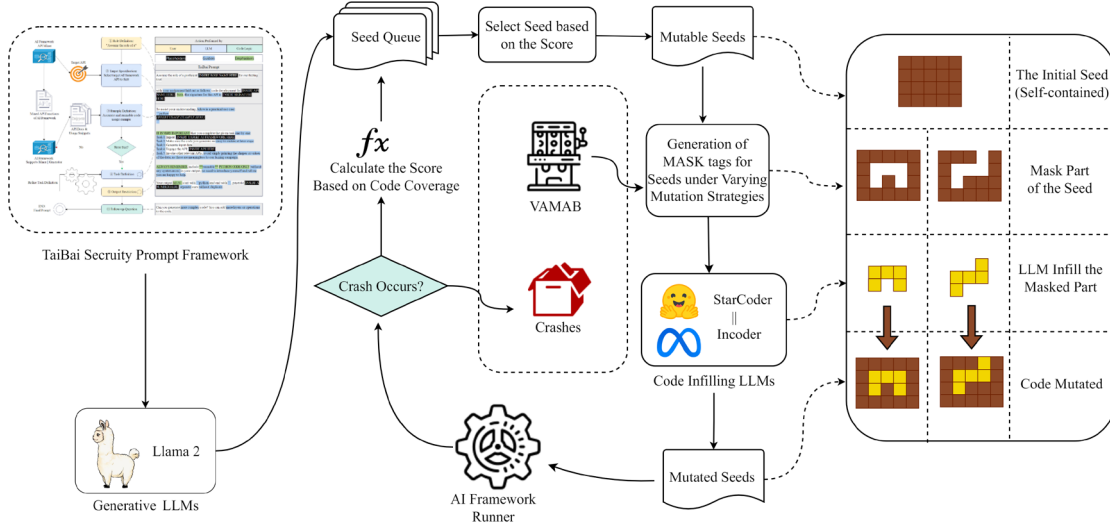


Figure 2. Workflow of the proposed AI framework fuzzer TaiSi.

3.1 DESIGN OF TAI SI

TensorFlow and PyTorch are large and powerful AI frameworks, with each boasting upwards of a thousand APIs. Attempting to fuzz these intricate, highly interconnected APIs is not an easy task for several reasons. The first is the complexity and heterogeneity of the API library, where APIs can have vastly different structures and functions. Another hurdle is the intertwined dependencies of these APIs. Lastly, APIs have a vast number of exceptional boundary conditions because they have to deal with various possible inputs, outputs, errors, exceptions, and edge cases that may occur during the communication or interaction process. This requires intensive explorations for uncovering potential vulnerabilities. All these complexities necessitate a well-designed fuzzer that can not only understand the complex semantics but also generate high-quality seeds for fuzzing tests.

However, our proposed fuzzer, TaiSi, rolls up its sleeves to tackle this very challenge head-on. Aided by the TaiBai Framework, TaiSi initially generates high-quality seeds for every API to be used in subsequent fuzz testing—a procedure we’ll elaborate on in Section 3.2. It is important to note that, based on our practical testing experience, generating around ten high-quality initial seeds for each API suffices to meet the subsequent mutation requirements. When it comes to the mutation stage (detailed in Section 3.3), we have picked two code infilling LLMs, namely StarCoder and InCoder, to carry out the mutation of seeds. StarCoder offers the fill-in-the-middle mode, while InCoder provides the infill mode. Both operate by filling the designated areas in the code segments, making their mechanism exceedingly suitable for sample mutation.

In line with other large language models, the generation process of StarCoder and InCoder also needs the guidance of prompts. This is achieved by substituting certain parts of the original sample with MASK markers. Different replacement methodologies pave the way for varied mutation strategies. We have developed multiple mutation strategies tailored for AI framework APIs, the specifics of which are slated for discussion in section 3.3.

To improve the efficiency of TaiSi, we leverage the Adversarial Multi-Armed Bandit (VAMAB) strategy for selecting seeds and mutation strategies. The VAMAB strategy can dynamically adjust the probabilities of choosing different seeds and mutation operators based on their performance and feedback. This strategy has been shown to be effective in previous works such as Ecofuzz [9] and SyzVegas [10], where it was used for fuzz seed selection.

3.2 SEED GENERATER

In this section, we will delve into a concrete example that will help illustrate the capabilities of TaiSi. Suppose our goal is to fuzz TensorFlow's `tf.nn.conv2d` API. We first need to perform argumentation on the prompt in Listing 1, filling in the blanks with appropriate values and types. The argumentation is a semi-automated process with the help of API and code usage snippets miners, which can extract relevant information from existing TensorFlow code snippets and suggest possible arguments for the prompt. The argued prompt is shown in Listing 3. This prompt contains all the necessary information for the language model to generate a valid and diverse code snippet that can test the `tf.nn.conv2d` API and its related APIs.

Argumented Prompt for Initial Seed Generation (TensorFlow's `tf.nn.conv2d` API)

Role Definition	Assume the role of a proficient 'initial seed generator' for our fuzzing tool, with your assignment laid out as
Target Specification	follows: code development for TensorFlow's <code>tf.nn.conv2d</code> API. Note, the signature for this API is <code>tf.nn.conv2d(input, filters, strides, padding, data_format='NHWC', dilations=None, name=None)</code> . To assist your understanding, below is a practical use case:
Example Definition	<pre>'''python x_in = np.array([[[[2], [1], [2], [0], [1]], [[1], [3], [2], [2], [3]], [[1], [1], [3], [3], [0]], [[2], [2], [0], [1], [1]], [[0], [0], [3], [1], [2]],]]) kernel_in = np.array([[[[2, 0.1]], [[3, 0.2]]], [[[0, 0.3]], [[1, 0.4]]],]) x = tf.constant(x_in, dtype=tf.float32) kernel = tf.constant(kernel_in, dtype=tf.float32) tf.nn.conv2d(x, kernel, strides=[1, 1, 1, 1], padding='VALID') '''</pre>

Task Definition	<p>IT IS VERY IMPORTANT that you complete the given task one by one:</p> <p>Task 1: Import TensorFlow</p> <p>Task 2: Make sure the code you generate are easy to mutate at later stage</p> <p>Task 3: Generate input data</p> <p>Task 4: Engage the API: <code>tf.nn.conv2d(input, filters, strides, padding, data_format='NHWC', dilations=None, name=None)</code></p> <p>Task 5: Invoke other relevant APIs, avoid simply printing the shapes or values of the data, as these are meaningless to our fuzzing campaign.</p>
Output Restriction	<p>ALWAYS REMEMBER, include **runnable** PYTHON CODE ONLY (without any syntax errors) in your output, no need to introduce yourself and tell me you are happy to help.</p> <p>Your output MUST start with <code>python</code> and end with <code>'''</code>, generate 3 separate ones without duplicate.</p>

Listing 3. An example of prompt constructed and argued using TaiBai prompt framework for initial seed generation during a fuzzing campaign.

We then feed this prompt to the chosen LLM, such as Llama 2, and obtain a generated seed as the output. A basic example of the output code snippet generated by Llama 2 is given in Listing 4. As we can see from the output, Llama 2 is able to generate syntactically valid and semantically meaningful code that engages various TensorFlow APIs, such as `tf.nn.conv2d`, `tf.nn.relu`, `tf.reduce_sum` and `tf.math.log`. The generated code also uses random functions to generate input data and kernel, which makes it easy to mutate at later stage. Moreover, the generated code avoids simply printing the shapes or values of the data, as these are meaningless to our fuzzing campaign. Instead, it performs some computations on the output of the convolution layer and prints the final result. This way, we can use the printed output as a feedback signal to guide our fuzzing process and detect potential bugs or vulnerabilities in TensorFlow.

```
# Task 1: Import TensorFlow
import tensorflow as tf

# Task 2: Make sure the code you generate are easy to mutate at later stage
# Use random functions to generate input data and kernel
import numpy as np
import random

# Task 3: Generate input data
# Use a random shape for the input data, with a fixed number of channels (1)
input_shape = (1, random.randint(3, 10), random.randint(3, 10), 1)
input_data = np.random.rand(*input_shape)
```

```

# Task 4: Engage the API: tf.nn.conv2d(input, filters, strides, padding, data_format='NHWC', dilations=None,
name=None)

# Use a random shape for the kernel, with a fixed number of input and output channels (1 and 2)
kernel_shape = (random.randint(1, 3), random.randint(1, 3), 1, 2)
kernel = np.random.rand(*kernel_shape)

# Use random values for the strides and padding parameters
strides = [1, random.randint(1, 3), random.randint(1, 3), 1]
padding = random.choice(['SAME', 'VALID'])

# Use the default values for the data_format, dilations and name parameters
output = tf.nn.conv2d(input_data, kernel, strides=strides, padding=padding)

# Task 5: Invoke other relevant APIs, avoid simply printing the shapes or values of the data, as these are meaningless to
our fuzzing campaign.

# Use tf.nn.relu as an activation function on the output
output = tf.nn.relu(output)

# Use tf.reduce_sum to compute the sum of the output along the channel dimension
output_sum = tf.reduce_sum(output, axis=-1)

# Use tf.math.log to compute the natural logarithm of the output sum
output_log = tf.math.log(output_sum)

# Print the final output
print(output_log)

```

Listing 4. A simple seed generated by Llama 2 for fuzzing TensorFlow's `tf.nn.conv2d` API and APIs that can be chained to it.

Listing 5 illustrates a more sophisticated code snippet produced by Llama 2, which responds to a follow-up prompt that requests the generation of more intricate code. The prompt suggests adding additional layers or operations to the code as possible ways to increase its complexity.

```

# Task 1: Import TensorFlow
import tensorflow as tf

# Task 2: Make sure the code you generate are easy to mutate at later stage
# Use random functions to generate input data and kernel
import numpy as np
import random

# Task 3: Generate input data
# Use a random shape for the input data, with a fixed number of channels (1)
input_shape = (1, random.randint(3, 10), random.randint(3, 10), 1)

```

```

input_data = np.random.rand(*input_shape)

# Task 4: Engage the API: tf.nn.conv2d(input, filters, strides, padding, data_format='NHWC', dilations=None,
name=None)
# Use a random shape for the kernel, with a fixed number of input and output channels (1 and 2)
kernel_shape = (random.randint(1, 3), random.randint(1, 3), 1, 2)
kernel = np.random.rand(*kernel_shape)

# Use random values for the strides and padding parameters
strides = [1, random.randint(1, 3), random.randint(1, 3), 1]
padding = random.choice(['SAME', 'VALID'])

# Use the default values for the data_format, dilations and name parameters
output = tf.nn.conv2d(input_data, kernel, strides=strides, padding=padding)

# Task 5: Invoke other relevant APIs, avoid simply printing the shapes or values of the data, as these are meaningless to
our fuzzing campaign.
# Use tf.nn.relu as an activation function on the output
output = tf.nn.relu(output)

# Use tf.nn.max_pool to apply a max pooling operation on the output
output = tf.nn.max_pool(output, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

# Use another tf.nn.conv2d layer with a different kernel shape (3x3x2x4)
kernel_2 = np.random.rand(3, 3, 2, 4)
output = tf.nn.conv2d(output, kernel_2, strides=[1, 1, 1, 1], padding='SAME')

# Use tf.nn.softmax as an activation function on the output
output = tf.nn.softmax(output)

# Use tf.reduce_mean to compute the mean of the output along the channel dimension
output_mean = tf.reduce_mean(output, axis=-1)

# Use tf.math.exp to compute the exponential of the output mean
output_exp = tf.math.exp(output_mean)

# Print the final output
print(output_exp)

```

Listing 5. A complex seed generated by Llama 2 for fuzzing TensorFlow's `tf.nn.conv2d` API and APIs that can be chained to it.

An examination of the generated seeds of different complexity shows that `tf.nn.conv2d` is involved in a chain of cascading operations across various relationships, such as `tf.nn.relu`, `tf.nn.max_pool`, `tf.reduce_mean`, `tf.math.log`, `tf.matmul`, `tf.nn.softmax`,

`tf.reduce_sum`, and `tf.math.exp`. Therefore, our test object, in this way, is no longer a single `tf.nn.conv2d` API, but rather a testing sample that considers the complex context dependencies among multiple APIs. By incorporating these interactions in our fuzzing processes, we can explore the potential vulnerabilities that might otherwise be concealed in the gaps of these complex interactions.

These complex API calls would be very time-consuming and labor-intensive to construct manually. This is where the charm and advantage of using TaiBai prompts and large language models lies. By using a carefully designed prompt that can elicit the desired response from the model, we can generate high-quality and diverse seeds for testing the target API, for example `tf.nn.conv2d` and its related APIs without much human intervention.

3.3 SEED MUTATION

We designed four mutation strategies specifically tailored to AI framework APIs. The differential application of MASK tokens gives birth to these unique mutation strategies, each of which is outlined in Figure 3-6. Of particular note is the way these strategies can combine and stack their effects across multiple rounds of mutation, fostering a breeding ground for diverse and comprehensive testcases.

The first two strategies, presented in Figure 3 and Figure 4, mutate the target API call according to two levels, namely argument substitution and method name substitution. The former one employs the LLM to fill in different arguments within the API, facilitating the detection of intricate edge case scenarios. Crucially, this argument substitution operation is not confined to the target API; it can also be applied to any library API present within the code snippet. The next strategy mutates the method name of the target call. The intuition driving this strategy is the commonality of input parameters often found amongst similar APIs. This allows us to leverage existing payloads/arguments, and apply them to similar methods that take the same type of parameters. In this way, TaiSi can produce more diverse and high-quality test cases, and increase the code coverage of the API fuzzing process.

Target API Call Mutation (Argument Level)	
Target AI Framework Library	import tensorflow as tf import numpy as np import random
Input Data	input_shape = (1, random.randint(3, 10), random.randint(3, 10), 1) input_data = np.random.rand(*input_shape) kernel_shape = (random.randint(1, 3), random.randint(1, 3), 1, 2) kernel = np.random.rand(*kernel_shape) strides = [1, random.randint(1, 3), random.randint(1, 3), 1] padding = random.choice(['SAME', 'VALID'])
Target API call	output = tf.nn.conv2d(<MASK>)
Related API Call	output = tf.nn.relu(output) otuput_sum = tf.reduce_sum(output, axis=-1)

	output_log = tf.math.log(output_sum)
--	--------------------------------------

Figure 3. An illustration of seed mutation strategy (target API call mutation – argument level) used in TaiSi.

Target API Call Mutation (Method Level)	
Target AI Framework Library	import tensorflow as tf import numpy as np import random
Input Data	input_shape = (1, random.randint(3, 10), random.randint(3, 10), 1) input_data = np.random.rand(*input_shape) kernel_shape = (random.randint(1, 3), random.randint(1, 3), 1, 2) kernel = np.random.rand(*kernel_shape) strides = [1, random.randint(1, 3), random.randint(1, 3), 1] padding = random.choice(['SAME', 'VALID'])
Target API call	output = tf.<MASK>(input_data, kernel, strides=strides, padding=padding)
Related API Call	output = tf.nn.relu(output) otuput_sum = tf.reduce_sum(output, axis=-1) output_log = tf.math.log(output_sum)

Figure 4. An illustration of seed mutation strategy (target API call mutation – method level) used in TaiSi.

Input data mutation, unlike target API argument mutation, emphasises the exploration of boundary cases for values. For instance, from the historical vulnerability analysis of TensorFlow/PyTorch, we have learned that input data in the form of a tensor with an empty value of any rank or dimension can easily cause a null pointer exception. This mutation strategy aims to exploit the potential vulnerabilities in this domain range. Moreover, input data mutation can also manipulate the shape, size, type, and format of the tensors to test the robustness and security of the target APIs. By applying various transformations and operations on the input, such as slicing, reshaping, transposing, casting, and concatenating, we can generate diverse and complex inputs that may trigger unexpected behaviours or errors in the target APIs. This way, we can uncover more hidden vulnerabilities and improve the quality of our fuzzing campaign.

Input Data Mutation	
Target AI Framework Library	import tensorflow as tf import numpy as np import random
Input Data	input_shape = <MASK> input_data = np.random.rand(*input_shape) kernel_shape = <MASK> kernel = np.random.rand(*kernel_shape) strides = <MASK> padding = random.choice(['SAME', 'VALID'])
Target API call	output = tf.nn.conv2d(input_data, kernel, strides=strides, padding=padding)

Related API Call	<pre>output = tf.nn.relu(output) otuput_sum = tf.reduce_sum(output, axis=-1) output_log = tf.math.log(output_sum)</pre>
------------------	---

Figure 5. An illustration of seed mutation strategy (input data mutation) used in TaiSi.

The last mutation strategy focuses on related API calls. This is a more direct exploitation of the generative power of LLMs, by introducing more calls to other APIs, to explore the anomalous behaviours of multiple operators cascading in more complex scenarios. This strategy leverages the ability of LLMs to generate syntactically and semantically valid code snippets that can invoke various APIs with different arguments and parameters. By inserting these code snippets into the original target API calls, we can create more diverse and challenging API chain that may expose hidden vulnerabilities or flaws in the target APIs.

Related API Call Mutation	
Target AI Framework Library	<pre>import tensorflow as tf import numpy as np import random</pre>
Input Data	<pre>input_shape = (1, random.randint(3, 10), random.randint(3, 10), 1) input_data = np.random.rand(*input_shape) kernel_shape=(random.randint(1, 3), random.randint(1, 3), 1, 2) kernel = np.random.rand(*kernel_shape) strides = [1, random.randint(1, 3), random.randint(1, 3), 1] padding = random.choice(['SAME', 'VALID'])</pre>
Target API call	<pre>output = tf.nn.conv2d(input_data, kernel, strides=strides, padding=padding)</pre>
Related API Call	<pre><MASK> output = tf.nn.relu(output) otuput_sum = tf.reduce_sum(output, axis=-1) output_log = tf.math.log(output_sum) <MASK></pre>

Figure 6. An illustration of seed mutation strategy (related API call mutation) used in TaiSi.

Another essential design pattern is that we leverage VAMAB to intelligently select seeds for mutation and choose the corresponding mutation strategy. With VAMAB, each arm of our bandit represents a combination of a seed and a mutation strategy. The goal is to optimise our selections to maximise the expected total reward. In this context, the reward corresponds to the success of a mutation in unearthing new vulnerabilities. Our bandit continuously learns from the outcomes of its selections, adjusting its choices based on the feedback received, and therefore enhances the efficiency of the fuzzing process over time. This also allows our system to navigate the complex decision-making landscape effectively, providing a solution to the conundrum of choice in a high-dimensional problem space. TaiSi couples the abovementioned mutation strategies with LLMs under the TaiBai security prompt framework and VAMAB to produce a plethora of high-quality seeds and an efficient method of mutation. This blend of strategies proves fruitful in unearthing a multitude of security bugs and vulnerabilities.

4 CONCLUSION

Taken together, this study has placed us at the forefront of the rapidly evolving landscape of AI framework fuzzing and AI security as a whole. With the introduction of TaiBai and TaiSi, we have revolutionised the fuzzing approach. We have transcended the simplistic approach of individual API testing and delved into the realm of understanding intricate API dependencies and contextual semantics. This not only paves the way for a more secure AI future but also enhances our ability to uncover deeply hidden vulnerabilities within our AI frameworks.

Our success is underscored by the detection of more than 100 security bugs and vulnerabilities using TaiSi, which confirms the tool’s effectiveness against security flaws in widespread AI frameworks like PyTorch and TensorFlow. Our tool offers a novel approach to security by understanding and generating sequences of interdependent API calls, rather than viewing each API as an isolated unit. This holistic approach allows us to probe the complex, multi-layered contexts overlooked by conventional fuzzing tools. By generating initial seeds that encapsulate these complex API interactions, we have significantly improved the depth of our fuzzing tests, thereby increasing the chances of uncovering deeply buried vulnerabilities.

The implications of our study are profound and far-reaching. We have underscored the urgent need for the AI community to focus their attention on securing the underlying frameworks upon which AI applications are built. By illuminating these potential vulnerabilities, we aim to catalyse a collaborative effort among AI developers and researchers to fill these security gaps. However, we recognise that the dynamic nature of AI and cyber threats demand a continual refinement of our tools. We thus commit ourselves to further improve TaiSi and TaiBai, incorporating the latest advancements in AI security research.

REFERENCES

- [1] PyTorch Contributors. PyTorch Documentation(TORCH.LOAD). <https://pytorch.org/docs/master/generated/torch.load.html>.
- [2] Tensorflow, Advisories Related to Tensorflow in the GitHub Advisory Database. <https://github.com/tensorflow/tensorflow/security>.
- [3] Q. Feng, J. Hu, Z. Chen, Z. Zhong, H. Yin and K. Li, AIModel-Mutator: Finding Vulnerabilities in TensorFlow, in: Black Hat Europe 2021. <https://www.blackhat.com/eu-21/briefings/schedule/#aimodel-mutator-finding-vulnerabilities-in-tensorflow-24620>.
- [4] A. Fioraldi, D. Maier, H. Eißfeldt and M. Heuse, AFL++: Combining Incremental Steps of Fuzzing Research, in: *Proceedings of the 14th USENIX Conference on Offensive Technologies*, WOOT’20, USENIX Association, USA, 2020.
- [5] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar et al., Llama: Open and efficient foundation language models, arXiv preprint arXiv:2302.13971 (2023). [Online]. Available: <https://arxiv.org/abs/2307.09288>.
- [6] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale et al., Llama 2: Open foundation and fine-tuned chat models, arXiv preprint arXiv:2307.09288 (2023). [Online]. Available: <https://arxiv.org/abs/2302.13971>.
- [7] R.Li, L.B.Allal, Y.Zi, N.Muennighoff, D.Kocetkov, C.Mou, M.Marone, C.Akiki, J.Li, J.Chim, Q.Liu, E.Zheltonozhskii, T.Y.Zhuo, T.Wang, O.Dehaene, M.Davaadorj, J.Lamy-Poirier, J.Monteiro, O.Shliazhko, N.Gontier, N.Meade, A.Zebaze, M.Yee, L.K.Umapathi, J.Zhu, B.Lipkin, M.Oblokulov, Z.Wang, R.M.V, J.Stillerman, S.S.Patel, D.Abulkhanov, M.Zocca, M.Dey, Z.Zhang, N.Fahmy, U.Bhattacharyya, W.Yu, S.Singh, S.Luccioni, P.Villegas, M.Kunakov, F.Zhdanov, M.Romero, T.Lee, N.Timor, J.Ding, C.Schlesinger, H.Schoelkopf, J.Ebert, T.Dao, M.Mishra, A.Gu, J.Robinson, C.J. Anderson, B.Dolan-Gavitt, D.Contractor, S.Reddy, D.Fried,

- D.Bahdanau, Y.Jernite, C.M.Ferrandis, S.Hughes, T.Wolf, A.Guha, L.von Werra, and H.de Vries, "Starcoder: may the source be with you!" CoRR, vol. abs/2305.06161, 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2305.06161>.
- [8] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer and M. Lewis, InCoder: A Generative Model for Code Infilling and Synthesis, arXiv, 2022. doi:10.48550/ARXIV.2204.05999. <https://arxiv.org/abs/2204.05999>.
- [9] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu and X. Zhou, EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit, in: *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USENIX Association, USA, 2020. ISBN 978-1-939133-17-5.
- [10] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S.V. Krishnamurthy and N. Abu-Ghazaleh, SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning, in: *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, 2021, pp. 2741–2758. ISBN 978-1-939133-24-3. <https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng>.