Shared Prime Vulnerability Analysis for Pseudo-Random Number Generator Implementation Proposed by AI Models

Ajay Singh RATHORE, Samuel TĂNASE
University of Science and Technologies Politehnica of Bucharest
ajay.rathore.sr@gmail.com, tanase.samuel.00@gmail.com

Abstract: Large Language Models have become popular for performing different tasks like summarization, translations, classification and code generation. This has led to different types of research where models are observed and their architecture is improved for better performance, including the tasks mentioned above. Other studies branch into finding a better way to generate responses through these LLMs while treating them as a black box. There are studies where the models are specially analyzed for the code generation task. We follow this category of studies to analyze the generated code for security weaknesses like CWEs. A few different publicly available models were used and prompted to provide the implementation of pseudo-random number generation for prime numbers. The purpose of this study is to find out if these generated prime numbers will be secure enough to be used for cryptography algorithms such as RSA.
Keywords: Large Language Models, Code Generation, Security Weaknesses, CWEs, RSA.

INTRODUCTION

Ron Rivest, Adi Shamir, and Leonard Adleman developed the popular algorithm: RSA. RSA stands for the initials of the authors. This algorithm is widely used for cryptographic tasks. Such an algorithm relies on the generation of two prime numbers: p and q. These numbers are used to compute the public and private key pairs of the key (Mlanov, 2009). A public key contains the product of its prime numbers, called the public modulus ($n = p * q$). The public modulus is in line with the idea that factoring the public key will be a computationally intensive task.

During the practical implementation of this algorithm, it is very important that previously generated random numbers are not repeated, otherwise, it can cause two issues:

First, for two given keys k_1, k_2 which are generated using p_1, q_1, p_2, q_2 respectively, then if

$$p_1 = q_1 \text{ and } p_2 = q_2 \Rightarrow k_1 = k_2.$$

As k_1 and k_2 are the same key, in that case, if k_1 is produced by the attacker, then a private key related to k_1 will allow decryption of all the messages encrypted by k_2.

This idea has been explored on a large scale in “Ron was wrong, Whit is right” (Lenstra et al., 2012). They found that, due to the repetition of previous random choices, 0.27 million keys were vulnerable, as they had the same moduli. This number represented 4% of the total collected RSA keys.

Second, for two given keys k_1 and k_2, which are generated using p_1, q_1, p_1, q_2 respectively, then both q_1 and q_2 can be obtained using the greatest common divisor:

$$GCD(k_1, k_2) = p_1$$

$$q_1 = k_1/p_1$$

$$q_2 = k_2/p_2$$

This implies that for both keys k_1 and k_2, private keys can now be derived. Hence, in this scenario, where two public keys contain one common prime, both keys can be compromised.

Through this study, the code generated when a large LLM is prompted to write an implementation for generating random prime numbers is analyzed. The three used models are GPT-3.5 (OpenAI, 2023), LLama2 (Meta, 2023), Microsoft/phi1(Microsoft, 2023). The paper is organised as follows. The following section briefly discusses the related work, and the next one presents the setup of the present experiment. The next section illustrates the results obtained, and the final one provides the conclusion of the present work and some prospects for the development of the proposed research. Appendix is added with the code generated by the models.

RELATED WORK

Similar studies are conducted on the quality of responses generated by AI. Such studies include analyzing capacity to write academic writing (Buruk, 2023), security weaknesses in Copilot generated code (Fu et al., 2023), or analyzing if the model is as bad as humans at introducing vulnerabilities (Asare et al., 2023). They show various findings, such as that 33.3% of the analyzed cases (Owura et al., 2023) produced vulnerable code, while Fu et al. (2023) show that up to 39.4% of snippets produced vulnerable code in Python and 46.1% in C++.

A common theme in these studies is that they treat the model as a black box, i.e., studies don’t try to train the model or change the parameters of the model. They focus on analyzing the model as if it were a service that could be prompted. A similar idea is used in this study as well, but the focus is more specifically on generating random prime numbers.

AI popularity has also triggered studies where this black box theme is used to analyze the performance of prompting. Studies show that following a set of mechanisms (Yuan et al., 2022) can generate different results and that models can be guided to provide better results. Similarly, Brown et al. (2020) present the increase in model accuracy, as zero-shot, one-shot and few-shot strategies are used.

EXPERIMENT SETUP

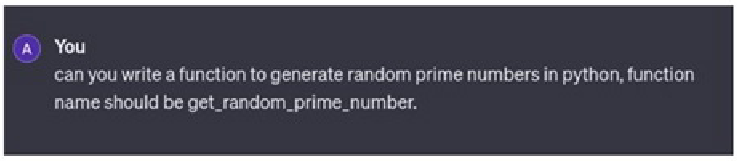
Different prompting strategies are taken into consideration when generating different code snippets. The same strategy was used for the models GPT-3.5 (OpenAi, 2023) and LLama2 (Meta, 2023), while, for Microsoft/phi1, a different strategy was used. The first two model prompts were categorized as such:

- Zero-shot: prompt was provided as a single sentence.

- One-shot: prompt included a single sentence and an example of the code snippet to guide a secure random prime generation.
- Few-shot: prompt was provided in multistep, i.e., a simple sentence was followed by a response, and then the response was added back, in order to generate the final code snippet.
- Few-shot-CoT (Chain of Thought): prompt was provided as a chain of thought to explain why certain examples were better than others.

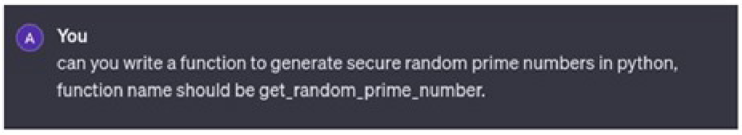
Some examples of these prompts are provided in Figure 1.

A



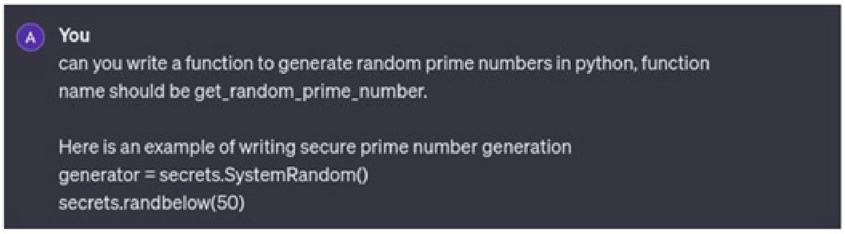
You
can you write a function to generate random prime numbers in python, function name should be get_random_prime_number.

B



You
can you write a function to generate secure random prime numbers in python, function name should be get_random_prime_number.

C



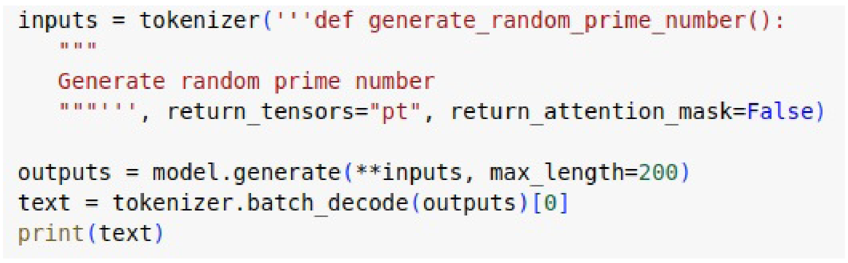
You
can you write a function to generate random prime numbers in python, function name should be get_random_prime_number.

Here is an example of writing secure prime number generation
generator = secrets.SystemRandom()
secrets.randbelow(50)

***Figure 1**. (A) Zero-shot prompt; (B) Zero-shot prompt with the word secure in it; (C) One-shot prompt with one example*

The Microsoft/phi1 model was loaded on a machine as no public interface was provided. The model was downloaded from Hugging Face and loaded using the torch library. Then prompts were created following the example provided on the model main page (Microsoft, 2023).

Figure 2 shows a prompt from Microsoft/phi1. Similarly, two other prompts were used [see Appendix].



```
inputs = tokenizer('''def generate_random_prime_number():
   """
   Generate random prime number
   """''', return_tensors="pt", return_attention_mask=False)

outputs = model.generate(**inputs, max_length=200)
text = tokenizer.batch_decode(outputs)[0]
print(text)
```

***Figure 2.** Prompt for the Microsoft/phi1 model*

Once the prompt output was collected, it was analyzed by humans to identify the following characteristics:

C1: Are large primes generated? i.e., the number is greater than or equal to 1024.

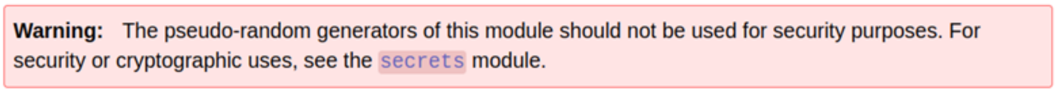
C2: Does code contain cryptographically secure methods? i.e., any CWE is present.

C3: Can the generated scripts run? If the code is valid, then a few code fixes are made in order to make the code run. If many changes are needed, it means that the code is not valid. If the code can run in a reasonable time, then it is valid.

C4: Run the valid scripts to generate 10K prime numbers and detect if the prime numbers are repeated.

RESULTS

The experiment showed clearly that in several instances, the functions generated by the AI models use Python's default random API. This API is not cryptographically secure (Python Software Foundation, 2024). Figure 3 shows the warning in the documentation provided for Cpython. This API is also marked (Ruohonen et al., 2021) to be associated with CWE-338 (MITRE, 2020).



Warning: The pseudo-random generators of this module should not be used for security purposes. For security or cryptographic uses, see the `secrets` module.

Figure 3. *The warning presented by the Cpython documentation of version 3.12.1.*

Some results showed that the scripts generated were not suitable to be run, such as:

```
import random

def get_random_prime_number(n):
   prime_numbers = []
   for i in range(n):
      prime_number.append(random.randint(1,100...
```

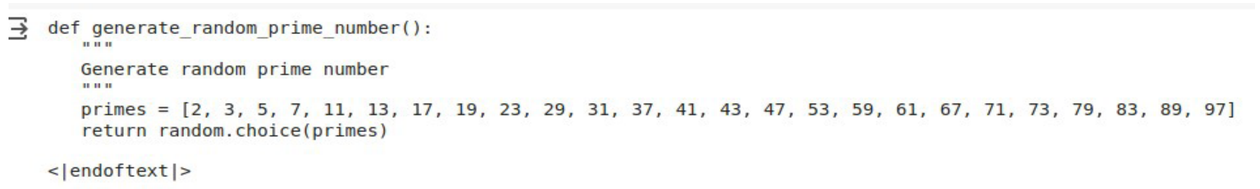
The dots after 100..., indicate a lengthy continuation of zeros. This is the case where it seems that model is just repeating itself after providing some tokens. In this instance, the result is not very useful and it would require the developers to write a more extended code to make the program functional.

There were also successful instances where the model generated Python code that used the cryptographically secure API "secrets".

```
def get_random_prime_number(bit_length=1024):
   """Generate a random prime number with the specified bit length."""
   while True:
       candidate = secrets.randbits(bit_length)
       if candidate % 2 == 0:
           candidate + = 1 # Ensure the number is odd
       if is_prime(candidate):
           return candidate
```

The generated code is a valid python script. Even though, it has the possibility to run for a long time if the candidate is not a prime number. Adding a default value of 1024 for bit_length ensures that, in case of clear declaration, a sufficiently large number is still obtained. It also handles the case where the prime candidate is even, by incrementing with 1, thus increasing the chances of getting a prime number. It seems that the model considered some logical cases and added code to handle them, even though no information for handling such cases was provided.

The case of the Microsoft/phi1 generated stub was the least desirable, as it is selected only a prime number from a hard-coded list:



```
def generate_random_prime_number():
    """
    Generate random prime number
    """
    primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
    return random.choice(primes)

<|endoftext|>
```

For Pseudo-Random Number Generator (PRNG), this code would be very limiting because it only has limited prime numbers from which it will make a choice. Repetition in prime numbers generated in this case will be guaranteed, if more than 25 calls will be made to this function. The prime numbers generated by this function are too small to be used for security purposes. Although the function is not helpful for the work of the present paper, it is worth noticing that the model was able to create a valid list of prime numbers between 1 and 100.

There is another notable thing that should be taken into consideration.

The Large Language models are trained on a corpus and then are prompted to autoregressively predict the next token. The generated text above shows the capacity of these models to produce algorithms by predicting the next token based on the previous generated tokens. Discussing different sampling strategies that can influence the process of which next token is selected would be out of the scope of this study.

Detailed generated code are attached in Appendix.

A summary of results is presented in Table 1 and Table 2.

Table 1. *Results for GPT-3.5 and LLama2*

	C1		C2		C3		C4	
Prompt	GPT-3.5	LLama2	GPT-3.5	LLama2	GPT-3.5	LLama2	GPT-3.5	LLama2
Zero-Shot Prompt 1	No	No	No	No	Yes	No	Yes	-
Zero-Shot Prompt 2	Yes	No	No	No	Yes	Yes*	No	Yes
One-Shot Prompt 1	Yes	No	Yes	Yes	Yes	No	No	-
One-Shot Prompt 2	Yes	No	Yes	No	No	No	-	-
Few-Shot Prompt 1	Yes	-	Yes	No	No	No	-	-
Few-Shot-CoT Prompt 1	No	Yes	No	No	Yes	No	No	-

"-" indicates that no relevant results can be defined
*script runs only for small primes

Table 2. *Results for Microsoft/phi1*

Microsoft/phi1	C1	C2	C3	C4
Prompt 1	No	No	Yes	Yes
Prompt 2	Yes	No	No	-
Prompt 3	No	No	No	-

"-" indicates that no relevant results can be defined

For C1, the capacity of the models to write a function that will generate sufficiently large primes is looked at. As it can be observed from the two tables, LLama2 was able to recommend a large prime only in one instance. Similarly, Microsoft/Phi1 was able to recommend large primes in just one instance. GPT-3.5 performed much better on C1 with 4 prompts out of 6 recommending large prime numbers. It is important to notice that some instances of the prompts used only the word "secure" and didn't explicitly mention that the generated result should be above 1024 bits.

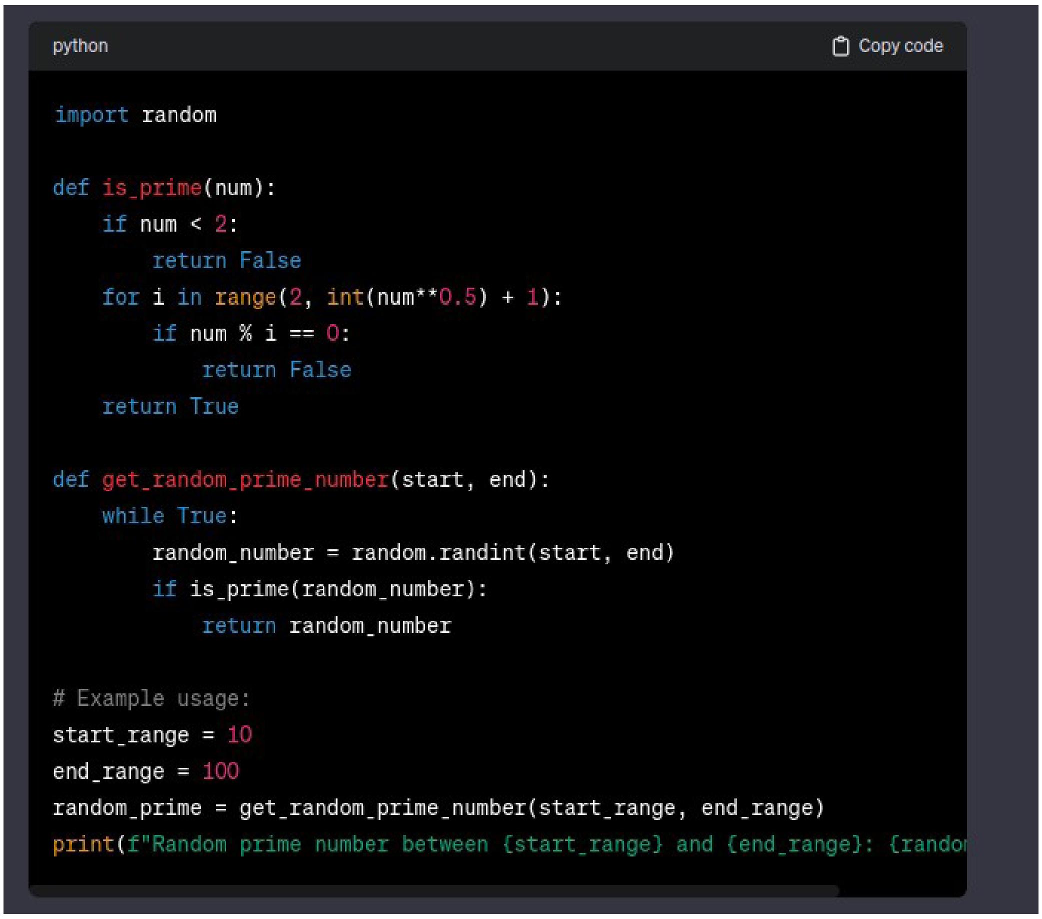
For C2, the capacity of the models to write a function that is free of known vulnerabilities is looked at. GPT-3.5 performed better with help from human input on the one-shot strategy and the few-shot strategy which involved giving it an example of using such APIs. LLama2 used the secure API just once, with a one-shot prompt.

Microsoft/Phi1 didn't use the vulnerability free "secrets" API at all.

For C3, we look at how valid is the code written by the model and if it could run successfully. In most cases, GPT-3.5 produced valid scripts. In the case of LLama2, only one script was runnable with the condition that only small primes can be generated. For Microsoft/Phi1, the generated code selected only the primes below 100 from a predefined list.

For C4, the scripts were run on Linux OS and 10K prime numbers were generated from the valid scripts. These numbers were then analyzed to count the duplicates. Only the script that didn't use large primes, i.e. C1, produced duplicates. GPT-3.5 output for the zero-shot prompt 1 wrote a script that used a range passed in the function as arguments named **start_range and end_range.**

Then it suggested start_range = 10 and end_range = 100, as it can be seen in Figure 4. If the suggested range is used and then 10k primes are generated, a lot of duplicates will be obtained, as there are very limited choices for prime numbers within the range 10 to 100. In the case of the zero-shot prompt 2 with LLama2 model, the generated code worked only for small range of prime numbers. The generated function also takes an argument n, used as the upper bound of the range. The value of n = 65536 was passed and the function did not stop in a reasonable time. The code used for generating these 10k prime numbers and the code generated by both LLama2 and GPT-3.5 is available in Appendix.



```python
import random

def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

def get_random_prime_number(start, end):
    while True:
        random_number = random.randint(start, end)
        if is_prime(random_number):
            return random_number

# Example usage:
start_range = 10
end_range = 100
random_prime = get_random_prime_number(start_range, end_range)
print(f"Random prime number between {start_range} and {end_range}: {rando
```

***Figure 4.** Prompt for the Microsoft/phi1 model*

CONCLUSION

The purpose of this study is to see how the AI models would perform on the criteria mentioned in this article. This study shows that not all the models perform well on these criteria, and special prompt strategies should be used to ensure that vulnerability free code is generated.

It is clear that LLMs can generate code using a simple natural language prompt. Although the quality of code can depend on the how well the prompt describes the expected output. After adding the word “secure” and providing samples of secure source code, GPT-3.5 was able to generate a better code.

From the findings of the present work, it can be observed that GPT-3.5 is the best in regards to the four criteria taken into account, i.e. it generates a valid Python code that can be successfully executed, the code calls cryptographically secure functions, the generated primes are sufficiently large and there are no duplicates. GPT-3.5 has even one instance (namely, **One-Shot Prompt 1**, as it could be seen in Table 1) where all the four criteria are valid. At the same time, if prompts are not given correctly, then it also has an instance (namely, **Zero-Shot Prompt 1** as it could be seen in Table 1), where three criteria, i.e., C1, C2, C4, are negative.

LLama2 did not have very good results overall.

Just one out of 6 attempts generated code that was valid (C3 criteria).

It also didn’t produce many positive results for C1 and C2, with just one case that was good.

Microsoft/phi-1 model have similar results with LLama2 with the exception at the C2 criteria, where the model did not generate any good sample.

The impact of generating predictable or repeated prime numbers would result in a weak or vulnerable RSA key pair generation. Hence, caution should be followed when including code that was generated by AI models.

The computer programmer should consult with computer security specialists or test the code with static code analysis methods and other security scans.

A future study can be made to analyze the performance of AI-generated code against different types of vulnerabilities. A classification framework can be proposed to identify the AI model's performance in the security field.

Further research can be made in order to fine-tune such models and update the model parameters so that the generated code is as vulnerability-free as possible.

REFERENCE LIST

Asare, O., Nagappan, M. & Asokan, N. (2023) Is GitHub's Copilot as Bad as Humans at Introducing Vulnerabilities in Code?. *Empirical Software Engineering.* [Preprint] https://arxiv.org/abs/2204.04741 [Accessed 27th December 2023].

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A. ,Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I. & Amodei, D. (2020) Language Models are Few-Shot Learners. *arXiv.* [Preprint] https://arxiv.org/abs/2005.14165 [Accessed 27th December 2023].

Buruk, O.O. (2023) Academic Writing with GPT-3.5 (ChatGPT): Reflections on Practices, Efficacy and Transparency. In: *Proceedings of the 26th International Academic Mindtrek Conference, 3-6 October 2023, Tampere, Finland.* New York, NY, USA, Academic Mindtrek. pp. 144–153.

Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M.& Yu, J. (2023) Security Weaknesses of Copilot Generated Code in GitHub. *arXiv.* [Preprint] https://arxiv.org/abs/2310.02059 [Accessed 27th December 2023].

Lenstra, A.K., Hughes, J.P., Augier M., Bos, J.W., Kleinjung, T. & Wachter, C. (2012) Ron was wrong, Whit is right. In: Safavi-Naini, R. & Canetti. R. (eds.) Advances in Cryptology – CRYPTO 2012. CRYPTO 2012. Lecture Notes in Computer Science, vol 7417. Berlin, Heidelberg, *Springer*. 626–642. doi: 10.1007/978-3-642-32009-5_37.

Microsoft. (2023) *microsoft/phi-1.* https://huggingface.co/microsoft/phi-1 [Accessed 27th December 2023].

The MITRE Corporation. (2020) *CWE-338 Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG).* https://cwe.mitre.org/data/definitions/338.html [Accessed 27th December 2023].

Mlanov, E. (3 June 2009) *The RSA Algorithm.* https://sites.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf [Accessed 4th January 2024].

Meta. (2023) *Code Llama 13B Chat.* https://huggingface.co/spaces/codellama/codellama-13b-chat [Accessed 27th December 2023].

OpenAI. (2023) *ChatGPT.* https://chat.openai.com/ [Accessed 27th December 2023].

Python Software Foundation. (2024) *random — Generate pseudo-random numbers.* https://docs.python.org/3/library/random.html [Accessed 27th December 2023].

Ruohonen, J., Hjerppe, K. & Rindell, K. (2021) A Large-Scale Security-Oriented Static Analysis of Python Packages in PyPI. In: *Proceedings of the 18th Annual International Conference on Privacy, Security and Trust* (PST 2021), *13-15 December 2021, Auckland, New Zealand.* New Jersey, USA, IEEE. pp. 1-10.

Yuan, X., Wang, T., Wang, Y., Fine, E., Abdelghani, R, Lucas, P., Sauzéon, H. & Oudeyer, P. (2022). Selecting Better Samples from Pre-trained LLMs: A Case Study on Question Generation. *arXiv.* [Preprint] https://arxiv.org/abs/2209.11000 [Accessed 27th December 2023].

APPENDIX

Rathore, A. (2024) *rsaTest.* https://github.com/AjaySRathore/rsaTest [Accessed 27th December 2023].