AI Code Generators for Security: Friend or Foe?

Roberto Natella, Pietro Liguori, and Cristina Improta | University of Naples Federico II
Bojan Cukic | University of North Carolina at Charlotte
Domenico Cotroneo | University of Naples Federico II

Recent advances of artificial intelligence (AI) code generators are opening new opportunities in software security research, including misuse by malicious actors. We review use cases for AI code generators for security and introduce an evaluation benchmark.

Large language models (LLMs) represent the latest breakthrough in machine learning and are going to have a significant impact on supporting people in various tasks. These models can automatically generate streams of humanlike text, as they are trained on huge volumes of text scraped from the web and books, using highly scalable deep learning architectures. Most notably, these models are also artificial intelligence (AI) code generators, as they can create computer programs using a programming language. Given, as input, a description of a program in natural language (e.g., plain English), AI can generate a corresponding program as a sequence of output tokens.

Computer security is also going to be affected by the advent of these AI code generators. They can represent a new threat, as malicious actors can use them to write new malicious software, bringing more diversity and agility to their attacks. AI code generators are also easily available to any malicious party through public services, such as GitHub Copilot and Amazon CodeWhisperer, which leverage the same technology behind the well-known ChatGPT and can convert natural language (e.g., in a code comment) into entire methods and functions from within the development environment.

Digital Object Identifier 10.1109/MSEC.2024.3355713
Date of current version: 1 February 2024

At the same time, security analysts can (and should) leverage AI code generators. We believe in the need for an open discussion on the uses of this technology for security applications. Since the dawn of the Internet, security analysts have been debating whether to publicly share information about vulnerabilities and programs to exploit them since this information can be misused even by inexperienced attackers (e.g., "newbies" and "script kiddies"). Attackers will inevitably take any opportunity to use AI code generators; cybersecurity professionals should also strive to benefit from these tools to better prevent and mitigate intrusions.

The field of generative AI for security is still a young one. Recent studies analyzed this technology in the context of generating malware, malicious content for social engineering, and a few more use cases. However, research on generative AI is limited by the availability of labeled datasets for security use cases, which are needed for fine-tuning LLMs since these models are trained only in a nonsupervised way. Moreover, datasets are needed to support research on new emerging LLMs by enabling rigorous experimental evaluations.

In this article, we study the application of AI code generators for creating synthetic attacks. First, we discuss potential benign applications of synthetic attack generation across many use cases in the context of

penetration testing. Then, we present a dataset and an experimental evaluation of three popular LLMs (GitHub Copilot, Amazon CodeWhisperer, and CodeBERT) for generating synthetic attacks. Our novel dataset includes a set of security-oriented programs in Python, which we annotate with descriptions in natural language. Our experiments show that the LLMs can generate security-oriented programs with high accuracy, although with less accuracy compared to general-purpose programs. We find that the best results are obtained with natural language descriptions at a fine grain (i.e., individual statements rather than whole functions) and by fine-tuning CodeBERT with our dataset. This dataset and the experiments can serve as benchmarks for future research.

Security Uses (and Misuses) of AI Code Generators

Many professional roles can benefit from AI code generators, including penetration testers, red teams, incident handlers, threat analysts, and more, as all these roles rely on writing custom software to automate complex tasks. Such tasks include the assessment of attack surfaces, the collection and analysis of intelligence, and the emulation of exploitations and adversarial behaviors. Moreover, AI code generators can assist newcomers (e.g., students) in writing code for security, which requires advanced coding skills on how to exploit software vulnerabilities. This barrier is a limiting factor to the growing demand for cybersecurity professionals, more of whom are needed to flatten the learning curve of security-oriented coding and to open the field to a wider and less experienced community. Thus, ethical hacking can greatly benefit from AI code generators.

Both the defensive and offensive sides invest significant efforts to write programs for automating common tasks and for scaling large systems and amounts of data. Scripting programming languages, such as Python, are a typical choice for task automation. These tasks include the following:

- *Attack surface analysis*: This involves the discovery of technical assets that are reachable from outside the target network. Assets include Internet Protocol (IP) addresses, servers, domain names, networks, and Internet of Things (IoT) objects. These assets are potentially affected by software vulnerabilities and misconfigurations that can be exploited by an attacker. For example, both defenders and attackers can write tools to enumerate subdomains, scan network ports, crawl web pages, and query search engines (e.g., Shodan) to identify vulnerable hosts and services, such as code repositories, admin panels, shared files, and e-mail and chat servers, which can be prone to data leaks (e.g., source code and authentication tokens) and can allow unauthorized access if not protected. Automated tools accelerate the analysis of multiple types of assets using different sources of data.
- *Open source intelligence*: This relates to the discovery of pieces of information about people in a target organization, such as names, e-mail addresses, phone numbers, and social network accounts, by looking into publicly reachable sources. Again, attacker-written tools can automate web crawling and parsing to extract this information. This information can be leveraged for attacks, such as for social engineering and brute forcing. For example, in brute-force attacks, a tool can include personal information to generate tentative usernames and passwords for logging into a system. In social engineering, attacker can use a tool to craft spear phishing e-mails from templates and send them to multiple targets. Similarly, defenders need to collect open source intelligence (OSINT) to learn about information leaks from their organization and to perform social engineering attacks for assessment purposes.
- *Vulnerability exploitation*: Attacks rely on automation to trigger vulnerabilities. Once a vulnerability has been discovered, malicious attackers use scripts ("exploits") to quickly exploit multiple targets (e.g., different organizations or different hosts in the same organization). Writing exploits is of high interest to security analysts too. They need exploits to test that their systems are indeed protected from a known attack. Moreover, exploits are often needed to demonstrate the impact and actual exploitability of a vulnerability ("proof of concept") to motivate vendors and users to patch their systems. In the worst case, writing an exploit can show that a vulnerability allows a remote attacker to execute arbitrary code in the target host; in other cases, the attacker may get access to data, cause a denial of service (e.g., killing a process and consuming resources), or exploit more vulnerabilities. It is challenging for vendors to tell apart vulnerabilities that are indeed exploitable; for example, Common Vulnerabilities and Exposures data are not technically verified and often misleading.
- *Postexploitation activities*: Getting a foothold through an exploited vulnerability is only the initial step of an attack (the "cyberkill chain"). Both attackers and security analysts ("red teams") leverage automated tools for lateral movement and privilege escalation by stealing credentials from sniffed traffic or compromised hosts, for persistence by installing back doors and remote-control tools to provide access and maintain it over time (e.g., after reboots), and for data theft and exfiltration by logging keystrokes and screens and transmitting the stolen data to an external network.

Attackers write custom programs for all these activities to tailor an attack for a specific victim and for evading antivirus, network monitoring, and endpoint detection and response solutions. For example, malware is often delivered as an encrypted payload to be launched with a decryption program (an "unpacker"); attackers apply their own custom-made (and even simplistic) encryption to differentiate from other attacks and evade malware detection signatures. Similarly, red teams emulate real attacks by using custom-made software to realistically assess the effectiveness of procedures and solutions for attack detection. Moreover, security analysts can use AI-generated code for automating incident response actions.

These use cases show that offensive security is a software-intensive area, but writing offensive code takes its toll on the time budget. Moreover, it can be a technically difficult activity. For example, in exploit development, "shellcode" payloads are typically written in assembly language to perform low-level operations and to gain full control of the layout of code and data in stack and heap memory so as to make the shellcode more compact and obfuscated. However, programming in assembly is time-consuming and has low productivity. In testing antimalware solutions, writing malware requires working with (and the abuse of) the complex C++ application programming interfaces (APIs) of the Microsoft Windows operating system and related products. Higher-level languages, such as Python, make it easier to write offensive code but provide less flexibility and can still require a significant amount of time to write code.

We make the case that security analysts need to leverage AI code generators to get support for defensive tasks. In this case, developers would translate a description of a piece of code in English (an "intent") into a corresponding code snippet. For example, security analysts can query AI for code snippets that they cannot recall or that they are not yet confident to write themselves, in a similar way to querying a search engine, with the additional benefit that the generated code is tailored for the specific application. Moreover, working with security code, such as in assembly language, can be a barrier for newcomers in security, which is a limiting factor to the growing demand for security analysts able to work with low-level attacks. Thus, AI code generators can flatten the learning curve with natural language processing. Finally, as malicious actors reap the benefits of AI code generators (e.g., to develop more diverse malware in larger quantities), security analysts also need to leverage AI to keep up with the pace.

Table 1. The violent-python dataset.

	Individual Lines	Multiline Blocks	Functions
Penetration testing	490	48	21
Forensic analysis	342	47	13
Network traffic analysis	375	43	20
OSINT and social engineering	553	55	25
Total	**1,129**	**171**	**72**

Experimental Evaluation

We experimented with AI code generators in the context of several security tasks. For evaluation purposes, we built our own manually curated dataset (violent-python, https://github.com/dessertlab/violent-python), where a sample contains a piece of code from offensive software (in a programming language) and its corresponding description in natural language (plain English).

We built the dataset by using the popular book *Violent Python*, by T. J. O'Connor,[1] which presents several examples of offensive programs using the Python language. Our dataset covers multiple areas of offensive security, including penetration testing (e.g., an automated exploit for a Server Message Block vulnerability, a port scanner, and a Secure Socket Shell botnet), forensic analysis (e.g., geolocating individuals, recovering deleted items, inspecting the Windows registry, examining metadata in documents and images, and analyzing data from mobile and desktop applications), network traffic analysis (e.g., capturing packets and geolocating IP addresses, identifying distributed denial-of-service toolkits, discovering decoy scans, analyzing botnet traffic, and foiling intrusion detection systems), and OSINT and social engineering (e.g., anonymously browsing the web, working with developer APIs, scraping popular social media sites, and creating a spear phishing e-mail).

The dataset consists of 1,372 unique samples, as shown in Table 1. Note that the row total indicates the total number of unique examples (i.e., we did not report replicated pairs of natural language intent/code snippets). This dataset is complementary to our previous datasets (Shellcode_IA32 and EVIL), where we included code snippets from shellcodes in assembly language[2] and from exploits in mixed Python and assembly language.[3]

The size of our dataset is in line with other state-of-the-art corpora used to fine-tune machine learning models. In fact, in state-of-the-art code generation, a model is not trained from scratch. Instead, existing LLMs (that were already trained with millions of publicly available lines of code) are fine-tuned in a

supervised way to achieve transfer learning for a specific case (in our case, generating offensive code). Typically, the datasets for fine-tuning are relatively limited, on the order of 1,000 samples.[4]

In our evaluation, we considered several approaches to describe offensive code in natural language since this is an important factor to determine the usability of AI code generators. One approach is to describe individual lines of code with an English statement, which is typical of other datasets in the field of code generation. This approach can provide the highest accuracy of the generated code since the developer guides the AI code generator with a fine-grained description. However, this approach is also the most verbose and demanding one for the developer. Therefore, we also consider other two approaches, where, respectively, we use an English statement to describe multiple lines of code ("blocks") and entire functions. In the case of blocks and functions, multiple code snippets are joined by the newline character "\n". Overall, the dataset consists of 82% individual lines, 12% multiline blocks, and 6% entire functions. For every script in the dataset, we manually described it in the three alternative ways. We based the descriptions on the contents of the chapter around each script and on comments in the code where available. Table 2 lists examples of descriptions of the three alternative granularities.

To evaluate AI code generators for security purposes, we start from CodeBERT (https://github.com/microsoft/CodeBERT), a pretrained language model for programming languages. CodeBERT is a model representative of the state of the art, which has achieved high performance in several software engineering tasks, including the generation of offensive code. It is a multiprogramming-lingual model, which has been pretrained on pairs of natural language intents and code snippets, across six different programming languages. CodeBERT represents the state of the art for several code-related tasks in the software engineering field, such as code search and the generation of code and other artifacts, such as comments, documentation, and commit messages. According to the best practices for using pretrained models, we use part of our dataset as training data to fine-tune the model for the specific task of generating offensive code. Moreover, we run the model along with data processing operations[3] both before translation to prepare the input data and after translation to improve the quality and readability of the code in output. For our experiments, we used a machine with a Debian-based distribution, with eight virtual CPUs, 16 GB of random-access memory, and two Nvidia T4 GPUs.

We assessed the model's ability to generate offensive code from different styles of natural language according to the three different levels of details in the descriptions of the dataset (i.e., lines, blocks, and functions). We split the dataset into sets for training (the samples for fine-tuning the model), validation (to tune the hyperparameters of the models), and testing (for the evaluation), using a random selection with the common 80%–10%–10% ratio.

To estimate the correctness of the AI-generated code, the gold standard is represented by a manual code review, where a human evaluator checks whether the code generated by the models is semantically correct, i.e., that it performs exactly what is described in the natural language intent. However, manual review is often infeasible due to the large amount of data to scrutinize, which makes the process time-consuming and prone to errors.

Therefore, the most common practice is to adopt output similarity metrics to assess the similarity of the code generated by the models with a reference ground truth. Among the large set of available output similarity metrics, we choose the edit distance (ED). We based this choice on our previous work,[5] where we systematically analyzed several similarity metrics for both Python

Table 2. Examples of intents in natural language.

Code	Individual Lines Description	Multiple Lines (Block) Description	Entire Function Description
`def connScan(tgtHost, tgtPort)`	Define function connScan with parameters tgtHost and tgtPort.	Try to create the socket with parameters AF_INET and SOCK_STREAM, connect to tgtHost on tgtPort, send the message "ViolentPython," receive the response, and acquire the lock.	Send the message "ViolentPython" to the host tgtHost on the port tgtPort and receive the response.
`try:`	Start the try block.		
`connSkt = socket(AF_INET, SOCK_STREAM)`	Create the socket with parameters AF_INET and SOCK_STREAM.		
`connSkt.send(ViolentPython \r\n)`	Send the message "ViolentPython."		
`results = connSkt.recv(100)`	Receive the response.		

and assembly code and analyzed the correlation of these metrics with semantic correctness. This metric measures the ED between two strings, i.e., the minimum number of operations on single characters required to make each code snippet produced by the model equal to a reference code snippet from the dataset, which is used as ground truth for the evaluation. The ED ranges between zero and one, with higher scores corresponding to smaller distances.

Including metrics that assess whether the code is compilable would not yield any useful information since these metrics assess the syntactical correctness rather than the semantic one. In fact, a code can be syntactically correct (i.e., compilable) but still not perform what is described in the intent (i.e., semantically incorrect). As a matter of fact, metrics such as compilation accuracy have shown to be less correlated to the semantic correctness of security-oriented code for both Python and assembly languages.[5]

To understand how the model fine-tuning impacts the performance, we compare the results against the performance of the model without any fine-tuning, also known as *zero-shot learning*. Figure 1 describes the results in terms of the ED. Unsurprisingly, the results highlight that fine-tuning the model on offensive code always provides performance higher than that obtained without fine-tuning. The boost in performance is more evident when the models generate individual lines (19.05% versus 77.89%) and becomes closer when the model generates blocks (18.23% versus 43.45%) and functions (22.31% versus 37.14%). This happens mainly for two reasons: 1) the fine-tuned model, as expected, is less accurate at generating complex code (e.g., code blocks and functions) than individual lines and 2) the model without fine-tuning (i.e., zero-shot learning) is insensitive to the complexity of the code to be generated. Even better, during zero-shot learning, CodeBERT generates functions with higher performance than that obtained during the generation of blocks and single lines. Most likely, the data used to pretrain CodeBERT contained several examples of complex code, such as entire functions rather than simple code snippets.

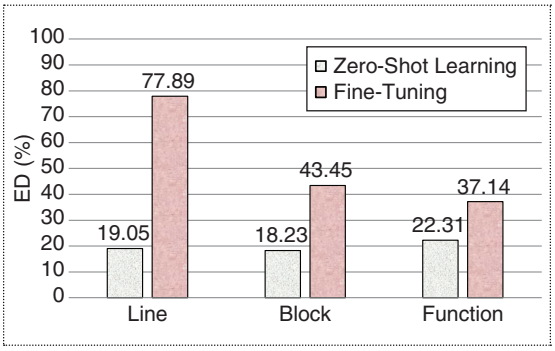

Figure 1. Zero-shot learning versus offensive code fine-tuning.

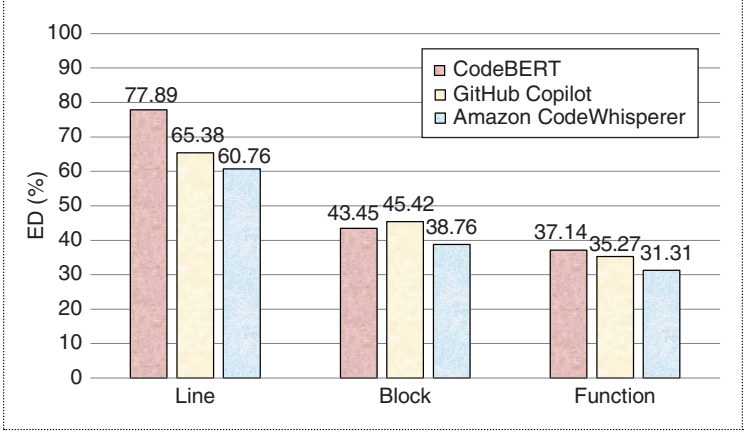

Figure 2. A comparison of AI code generators in the generation of offensive Python code.

Then, we compare the fine-tuned CodeBERT against two popular and widely used public AI code generators: GitHub Copilot (https://github.com/features/copilot) and Amazon CodeWhisperer (https://aws.amazon.com/codewhisperer). They are both public services that empower AI code assistants within the development environment by providing code suggestions from comments in natural language and from existing code. They were trained on billions of lines of code from open source projects. These solutions are accessible via APIs.

We compare the performance of the three AI code generators on the same test set. We used the training data only for CodeBERT since it is not possible to further fine-tune public AI code generators. Figure 2 presents the results, in terms of the ED, of the AI code generators in the generation of single lines, code blocks, and entire functions of security-oriented Python code. First, the figure shows that the performance decreases from single lines to code blocks and from code blocks to entire functions, regardless of the code generator. This is an expected result due to the increasing complexity of the code to be generated. Let us analyze the results thoroughly. For the simplest task, i.e., the generation of single lines, CodeBERT (77.89%) provides the best performance, followed by Copilot (65.38%) and CodeWhisperer (60.76%). We attribute this to the fine-tuning of the model since the process boosts performance at generating offensive code. For blocks and functions, where the number of samples in the dataset is limited, we found that CodeBERT and Copilot have similar performance (43.45% versus 45.52% for blocks and 37.14% versus 35.27% for functions); hence, the fine-tuning does not boost

the performance of CodeBERT when compared to public AI code generators. CodeWhisperer, instead, shows lower performance than the two competitors (38.76% and 31.31% for blocks and functions, respectively). The reasons can be attributed to the fact that CodeWhisperer, differing from a general-purpose tool, such as Copilot, caters first and foremost to development use cases associated with Amazon platforms, such as Amazon Web Services, i.e., to writing code related to Amazon APIs.

Finally, to provide context about the results achieved by the AI code generators, we can contrast them with the performance of state-of-the-art LLMs in the generation of general-purpose (i.e., nonoffensive) Python code, which is equal to 86.7%.[6] This result is comparable with the performance in the generation of individual lines of offensive code. Therefore, current language models provide good potential for generating offensive code, although they need to be fine-tuned for this task and be guided by fine-grained descriptions from developers. As the scale and complexity of LLMs grow, we can expect that their overall performance in security applications will improve over time.

To provide more practical insights into the code generated by the code generators, Table 3 provides a qualitative analysis using cherry-picked examples from our test sets. It is important to remark that the code we collected for the dataset is always related to security use cases. Unlike regular code generation that focuses on nonsecurity programs (e.g., create, read, update, and delete applications and small algorithms on data structures), security-oriented code contains a large number of low-level arithmetic logic operations and bit-level slices.[7] Given that security code consists of several lines of code, some versions can appear as "general purpose" code if considered out of context. However, all the examples in our dataset are indeed part of a real security use case.

Table 3. Examples of generated code and expected code.

Type	Natural Language Intent	Ground Truth Reference	CodeBERT Output	GitHub Copilot Output	Amazon CodeWhisperer Output
Line	Assign 0 to seqNum.	**seqNum = 0**	**seqNum = 0**	**seqNum = 0**	**seqNum = 0**
Block	If city is not equal to an empty string, concatenate the strings city, "," and country to geoLoc.	**if city != '': \n geoLoc = city + ', ' + country**	**if**(cityFile = None): \n cities = geoLoc	**if city != '': \n geoLoc = city + ', ' + country** \n print(geoLoc)	city = "London" \n if city: \n geoLoc = city + ", UK" \n print(geoLoc) \n else: \n print("City is not defined")
Function	Connect to the database skypeDB and execute the command "SELECT displayname, skypename, city, country, phone_ mobile, birthday FROM Contacts;."	**def printContacts (skypeDB): \n conn = sqlite3.connect (skypeDB) \n c = conn.cursor() \n c.execute ("SELECT displayname, skypename, city, country, phone_ mobile, birthday FROM Contacts;" \n**	**def** printCookies (cookiesDB): \n try: \n **conn = sqlite3.connect** (cookiesDB) **\n c = conn.cursor()** \n **c.execute("SELECT displayname, skypename, city, country, phone_ mobile, birthday FROM Contacts;") \n** for row in c: \n ...	**def** connect_ skypeDB(): \n try: \n **conn = sqlite3. connect** ('skypeDB') **\n cur = conn.cursor() \ncur.execute ("SELECT displayname, skypename, city, country, phone_mobile, birthday FROM Contacts;") \n** rows = cur.fetchall() \n for row in rows: \n ...	**def** connect_ db(): \n conn = sqlite3.connect ('skypeDB') \n return conn

Generated code that perfectly matches the reference is in bold.

The first row of the table shows how the models can correctly generate individual line code snippets by performing a simple operation, such as the assignment of a variable. The second row shows the ability of Copilot to generate a whole multiline block by generating a correct sequence of an if-then statement. CodeBERT and Amazon CodeWhisperer, instead, both fail to generate the correct output. Indeed, the former generates something close to the expected output, yet the result is incomplete. The latter, on the other hand, produces a verbose snippet that is syntactically correct but that diverges from the natural language description. Finally, the third row shows how the models deal with the generation of whole complex functions from a single natural language description. CodeBERT and Copilot prove their potential by generating several lines of code snippets; hence, they prove to be a valid solution as code assistants for more complex tasks. The code generated by CodeWhisperer again generates a syntactically correct function that, however, contains only a subset of the operation required to accomplish what is required by the natural language prompt.

Related Work

Given their recent advances, AI-based solutions have become an attractive solution for different tasks in the field of software security. Table 4 examines the related work.

Our work uses AI code generators for the generation of offensive code. Differing from previous work on generative AI, we adopt AI-based code generators to support several types of synthetic attacks in the context of penetration testing. Indeed, due to the lack of corpora containing security-oriented code to train AI-based solutions, previous work focused on other more specific use cases, such as the generation of exploits with low-level languages, malware generation, and malicious content for social engineering. Therefore, our work aims to expand the scope of generative AI for security by introducing a novel dataset and experimental baselines for research in this area.

Through this experience with LLMs, and in building a security-oriented evaluation benchmark, we learned about potential use cases for offensive security. These use cases encompass attack surface analysis, OSINT, vulnerability exploitation, and postexploitation. We believe that cybersecurity professionals must embrace AI code generators to prevent attacks more efficiently.

Overall, the results of our experiments on current AI code generators emphasize the importance of a careful

Table 4. Related work.

Year	Author	Contribution
2022	Liguori et al. [2]	Use of neural machine translation models to generate software exploits in assembly language from natural language
2022	Kim et al. [8]	Security surveillance toward AI-enabled digital twin service, which provides ecofriendly security through the active participation of low-resource devices
2022	Yang et al. [9]	Use of a shallow transformer model that performs code generation and summarization to generate software exploits
2023	Yang et al. [7]	Generation of software exploits using a rule-based template parser to generate augmented natural language descriptions and a semantic attention layer to extract and calculate each layer's representational information
2023	Xiao et al. [10]	Use of neural network-based API completion techniques to capture program dependencies
2023	McIntosh et al. [11]	Use of a state-of-the-art LLM in generating cybersecurity policies to deter and mitigate ransomware attacks that perform data exfiltration
2023	Pa et al. [12]	Development of malware programs and attack tools using public AI generative models
2023	Gupta et al. [13]	Use of public AI code generator to create social engineering attacks, phishing attacks, automated hacking, payload attacks, and malware
2023	Botacin et al. [14]	Use of public AI code generator to generate malware
2023	Grigoriadou et al. [15]	Detection and mitigation of IoT cyberattacks by using an AI-powered intrusion detection and prevention system

choice of which one to use. In particular, the experiments showed the importance of fine-tuning the models for security-oriented applications. In fact, when trained with samples of security-oriented code, CodeBERT can outperform popular public AI code generators and achieve performance close to nonsecurity-oriented applications. Unfortunately, the availability of data for security applications is very limited, and the creation of corpora from scratch is a difficult and time-consuming task, as it requires significant manual effort supported by high expertise and technical skills in the field.

When security-oriented code to fine-tune the models is not available, the usage of public AI code generators is a potential solution, although with a performance loss. The choice of a public generator strongly depends on the application, but the experiments we performed suggest that a general-purpose tool, such as GitHub Copilot, that was trained with more diverse programming languages and projects can better deal with the generation of offensive code than generators tailored for specific APIs and architectures, such as Amazon CodeWhisperer. ■

Acknowledgment

This work has been partially supported by Ministry of University and Research Projects of Significant National Interest 2022 project Federated Learning for Generative Emulation of Advanced Persistent Threats, CUP E53D23007950001 (https://flegrea.github.io/).

References

1. T. O'Connor, *Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers*. Oxford, U.K.: Newnes, 2012.
2. P. Liguori, E. Al-Hossami, D. Cotroneo, R. Natella, B. Cukic, and S. Shaikh, "Can we generate shellcodes via natural language? An empirical study," *Automated Softw. Eng.*, vol. 29, no. 1, p. 30, Mar. 2022, doi: 10.1007/s10515-022-00331-3.
3. P. Liguori et al., "EVIL: Exploiting software via natural language," in *Proc. 32nd IEEE Int. Symp. Softw. Rel. Eng. (ISSRE)*, Wuhan, China, Z. Jin, X. Li, J. Xiang, L. Mariani, T. Liu, X. Yu, and N. Ivaki, Eds., Piscataway, NJ, USA: IEEE, Oct. 25–28, 2021, pp. 321–332, doi: 10.1109/ISSRE52982.2021.00042.
4. C. Zhou et al., "LIMA: Less is more for alignment," 2023, arXiv:2305.11206.
5. P. Liguori, C. Improta, R. Natella, B. Cukic, and D. Cotroneo, "Who evaluates the evaluators? On automatic metrics for assessing AI-based offensive code generators," *Expert Syst. Appl.*, vol. 225, Sep. 2023, Art. 120073, doi: 10.1016/j.eswa.2023.120073.
6. A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "IntelliCode compose: Code generation using transformer," in *Proc. 28th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds., ACM, Nov. 8–13, 2020, pp. 1433–1443, doi: 10.1145/3368089.3417058.
7. G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Han, and T. Chen, "ExploitGen: Template-augmented exploit code generation based on codeBERT," *J. Syst. Softw.*, vol. 197, Mar. 2023, Art. no. 111577, doi: 10.1016/j.jss.2022.111577.
8. H. Kim and J. Ben-Othman, "Eco-friendly low resource security surveillance framework toward green ai digital twin," *IEEE Commun. Lett.*, vol. 27, no. 1, pp. 377–380, Jan. 2023, doi: 10.1109/LCOMM.2022.3218050.
9. G. Yang, X. Chen, Y. Zhou, and C. Yu, "DualSC: Automatic generation and summarization of shellcode via transformer and dual learning," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Honolulu, HI, USA, Mar. 15–18, 2022, pp. 361–372, doi: 10.1109/SANER53432.2022.00052.
10. Y. Xiao, W. Song, J. Qi, B. Viswanath, P. D. McDaniel, and D. Yao, "Specializing neural networks for cryptographic code completion applications," *IEEE Trans. Softw. Eng*, vol. 49, no. 6, pp. 3524–3535, Jun. 2023, doi: 10.1109/TSE.2023.3265362.
11. T. R. McIntosh et al., "Harnessing GPT-4 for generation of cybersecurity GRC policies: A focus on ransomware attack mitigation," *Comput. Secur.*, vol. 134, Nov. 2023, Art. no. 103424, doi: 10.1016/j.cose.2023.103424.
12. Y. M. P. Pa, S. Tanizaki, T. Kou, M. van Eeten, K. Yoshioka, and T. Matsumoto, "An attacker's dream? Exploring the capabilities of ChatGPT for developing malware," in *Proc. Cyber Secur. Experimentation Test Workshop (CSET)*, Marina del Rey, CA, USA. New York, NY, USA: ACM, Aug. 7–8, 2023. pp. 10–18, doi: 10.1145/3607505.3607513.
13. M. Gupta, C. Akiri, K. Aryal, E. Parker, and L. Praharaj, "From ChatGPT to ThreatGPT: Impact of generative AI in cybersecurity and privacy," *IEEE Access*, vol. 11, pp. 80,218–80,245, Aug. 2023, doi: 10.1109/ACCESS.2023.3300381.
14. M. Botacin, "GPThreats-3: Is automatic malware generation a threat?" in *Proc. IEEE Secur. Privacy Workshops (SPW)*, San Francisco, CA, USA. Piscataway, NJ, USA: IEEE, May 25, 2023. pp. 238–254, doi: 10.1109/SPW59333.2023.00027.
15. S. Grigoriadou et al., "Hunting IoT cyberattacks with AI - Powered intrusion detection," in *Proc. IEEE Int. Conf. Cyber Secur. Resilience (CSR)*, Venice, Italy. Piscataway, NJ, USA: IEEE, Jul. 31/Aug. 2, 2023. 2023, pp. 142–147, doi: 10.1109/CSR57506.2023.10224981.

Roberto Natella is an associate professor at the University of Naples Federico II, 80125 Naples, Italy. His research interests include software security and dependability, with a focus on the experimental injection of faults, attacks, and stressful conditions. Natella received a Ph.D. in information technology and electrical engineering from the University of Naples Federico II. Contact him at roberto.natella@unina.it.

Pietro Liguori is an assistant professor at the University of Naples Federico II, 80125 Naples, Italy. His research interests include automatic exploit generation and the robustness and security of large language models. Liguori received a Ph.D. in information technology and electrical engineering from the University of Naples Federico II. Contact him at pietro.liguori@unina.it.

Cristina Improta is a Ph.D. student at the University of Naples Federico II, 80125 Naples, Italy. Her research interests include offensive security, artificial intelligence code generation, and the security of machine learning systems. Improta received an M.Sc. in computer engineering from the University of Naples Federico II. Contact her at cristina.improta@unina.it.

Bojan Cukic is a professor in and the dean of the College of Computing and Informatics, University of North Carolina at Charlotte, Charlotte, NC 28223 USA. His research interests include information assurance and biometrics; software engineering, with an emphasis on verification and validation; and resilient computing. Cukic received a Ph.D. in computer science from the University of Houston, and an honorary Ph.D. from the University of Rijeka. Contact him at bcukic@charlotte.edu.

Domenico Cotroneo is a professor at the University of Naples Federico II, 80125 Naples, Italy. His research interests include software reliability and security, field failure data analysis, and software fault injection. Cotroneo received a Ph.D. from the Department of Computer Science and System Engineering, University of Naples Federico II. He is the chair of the IEEE Computer Society Technical Community on Dependable Computing and Fault Tolerance and an elected member of International Federation for Information Processing Working Group 10.4 on Dependable Computing and Fault Tolerance. He is a Senior Member of IEEE. Contact him at cotroneo@unina.it.