Enhancing AI-based Generation of Software Exploits with Contextual Information

Pietro Liguori*, Cristina Improta*, Roberto Natella*, Bojan Cukic† and Domenico Cotroneo*

*University of Naples Federico II, Naples, Italy

{pietro.liguori, cristina.improta, roberto.natella, cotroneo}@unina.it

†University of North Carolina at Charlotte, Charlotte, NC

bcukic@charlotte.edu

Abstract—This practical experience report explores Neural Machine Translation (NMT) models' capability to generate offensive security code from natural language (NL) descriptions, highlighting the significance of contextual understanding and its impact on model performance. Our study employs a dataset comprising real shellcodes to evaluate the models across various scenarios, including missing information, necessary context, and unnecessary context. The experiments are designed to assess the models' resilience against incomplete descriptions, their proficiency in leveraging context for enhanced accuracy, and their ability to discern irrelevant information. The findings reveal that the introduction of contextual data significantly improves performance. However, the benefits of additional context diminish beyond a certain point, indicating an optimal level of contextual information for model training. Moreover, the models demonstrate an ability to filter out unnecessary context, maintaining high levels of accuracy in the generation of offensive security code. This study paves the way for future research on optimizing context use in AI-driven code generation, particularly for applications requiring a high degree of technical precision such as the generation of offensive code.

Index Terms—Contextual Learning, AI Offensive Code Generation, Neural Machine Translation, Assembly, Software Exploits

I. INTRODUCTION

In recent years, the integration of AI-code generators into the programming workflow has marked a paradigm shift, significantly streamlining coding tasks and facilitating the interpretation of complex instructions through natural language (NL) using advanced ML models. This evolution has been particularly transformative in fields that demand high technical skills, such as *offensive security*, where the automation of coding tasks promises to boost productivity and innovation [1].

Offensive security, with its emphasis on developing *proof-of-concept* attacks to uncover and exploit vulnerabilities, occupies a critical junction in cybersecurity efforts. It aids in understanding the mechanics of attacks, motivating timely patches and mitigations [2]. However, the manual crafting of exploit code, given its dependence on deep system-level knowledge, presents a considerable bottleneck. Herein lies the appeal of automatic exploit generation (AEG): an AI-driven solution to enhance the efficiency of security analysts by generating functional exploits for security assessments.

Recent advancements have seen AI-based code generators, employing Neural Machine Translation (NMT) models, make significant strides in translating NL descriptions (*intents*) into precise programming *code snippets* [3], [4]. This capability is particularly noteworthy within the offensive security domain, where code generators translate intents detailing the exploitation of system vulnerabilities into actionable code [5], [6].

Yet, this domain's complexity introduces distinct challenges. The need for a stringent vocabulary to articulate low-level operations essential for exploiting system vulnerabilities means that the variability inherent in technical descriptions can severely impact the functionality of generated code. Variations in developers' technical knowledge, terminology use, and description specificity can lead to significant semantic discrepancies [7], [8]. Such variability underscores the necessity for code generators to exhibit *robustness*, ensuring effective operation across a wide spectrum of sentence structures, vocabularies, and detail levels. The absence of this robustness could limit code generators' real-world applicability, constraining their usbaility in practice [9], [10]. Hereby, exploit generation is pivotal in bridging the gap between theoretical NMT capabilities and practical, high-stakes security applications.

This practical experience report offers a critical examination of the potential and limitations of AI-code generators when faced with NL descriptions of varying accuracy and detail in the realm of applications requiring high levels of precision and contextual awareness, such as in the development of offensive security code. More precisely, this work delves into the potential benefits of training the models to comprehend the context of a sentence, aiming to discern the extent to which context understanding can compensate for the ambiguities of the NL. To this aim, we employ a prompt-engineering solution that feeds models with context information (*contextual learning*) by using the concatenation of inputs, i.e., by merging previous NL descriptions with the current one [11]–[14].

By harnessing the power of contextual learning, we demonstrate how AI code generators can significantly improve in translating NL descriptions into code for software exploits. Our study performs an extensive evaluation involving six state-of-the-art NMT models, each assessed for its ability to navigate the complexities of generating code from NL descriptions that vary in detail and context. Furthermore, the construction of a unique dataset from real-world exploits for model training and evaluation underscores the practical relevance and applicability of our findings.

arXiv:2408.02402v3 [cs.SE] 6 Sep 2024

TABLE I: Main findings of our research study.

Analysis	Main Findings
Missing Information	• Models struggled more with highly technical or specific operational commands when details were absent, underscoring the need for detailed NL descriptions in security-related tasks.
Contextual Learning	• Introducing additional context significantly improved model performance, with a notable enhancement in generating complex, multi-step security protocols. • There was a diminishing return on performance when extending context beyond a single preceding intent, indicating an optimal level of contextual information for effective learning.
Unnecessary Information	• Unnecessary information did not degrade model performance; models were able to maintain high performance, indicating effective filtering of irrelevant information.

To conduct our study, we identified three research questions (RQs) aimed at exploring the capabilities of NMT models specifically within the domain of offensive security. These questions focus on the models' ability to handle missing information, their ability to leverage contextual learning to enhance code generation, and the impact of unnecessary context on their performance. Central to our analysis is the examination of how these factors influence the accuracy of the generated code, pivotal in offensive security applications. Additionally, we investigate the potential of these models to discern and filter irrelevant information, i.e., a critical capability for maintaining high performance in complex security scenarios. TABLE I presents a concise summary of our key findings, offering insights into the optimization of NMT models for the requirements of offensive code generation.

In the following, Section II discusses related work; Section III illustrates the problem statement; Section IV describes our research study; Section V details our experimental setup; Section VI presents the experimental evaluation; Section VII discusses threats to validity; Section VIII concludes the paper.

II. RELATED WORK

Automatic exploit generation (AEG) research challenge consists of automatically generating working exploits [2]. This task requires technical skills and expertise in low-level languages to gain full control of the memory layout and CPU registers and attack low-level mechanisms (e.g., heap metadata and stack return addresses) not otherwise accessible through high-level programming languages. Given their recent advances, AI-code generators have become a new and attractive solution to help developers and security testers in this challenging task. Although these solutions have shown high accuracy in the generation of software exploits, their robustness against new inputs has never been studied before. Liguori *et al.* [15] released a dataset containing NL descriptions and assembly code extracted from software exploits. They performed an empirical analysis showing that NMT

models can correctly generate assembly code snippets from NL and that in many cases can generate entire exploits with no errors. The authors extended the analysis to the generation of Python offensive code used to obfuscate software exploits from systems' protection mechanisms [16]. Yang *et al.* [17] proposed a data-driven approach to software exploit generation and summarization as a dual learning problem. The approach exploits the symmetric structure between the two tasks via dual learning and uses a shallow Transformer model to learn them simultaneously. Yang *et al.* [18] proposed a novel template-augmented exploit code generation approach. The approach uses a rule-based template parser to generate augmented NL descriptions and uses a semantic attention layer to extract and calculate each layer's representational information. Ruan *et al.* [19] proposed *PT4Exploits*, an approach for software exploit generation via prompt tuning. They designed a prompt template to build the contextual relationship between English comment and the corresponding code snippet, simulating the pre-training stage of the model to take advantage of the prior knowledge distribution. Xu *et al.* [20] introduced an artifact-assisted AEG solution that automatically summarizes the exploit patterns from artifacts of known exploits and uses them to guide the generation of new exploits. The authors implemented *AutoPwn*, an AEG system that automates the generation of heap exploits for Capture-The-Flag *pwn* competitions. Recent work also explored the role of GPT-based models, including ChatGPT and Auto-GPT, in the offensive security domain. Botacin [21] found that, by using these models, attackers can both create and deobfuscate malware by splitting the implementation of malicious behaviors into smaller building blocks. Pa *et al.* [22] and [23] proved the feasibility of generating malware and attack tools through the use of reverse psychology and *jailbreak prompts*, i.e., maliciously crafted prompts able to bypass the ethical and privacy safeguards for abuse prevention of AI code generators like ChatGPT. Gupta *et al.* [23] also examined the use of AI code generators to improve security measures, including cyber defense automation, reporting, threat intelligence, secure code generation and detection, and attack and malware detection.

These studies do not take into account that, since developers have different levels of expertise and writing styles, code descriptions may be missing some information or reference previous instructions. The issue of inferring the context of the current sentence from the surrounding ones has been widely addressed for translation tasks between different natural languages. Proposed solutions vary from data-driven methods, to structural modifications to the model's architecture, or hybrid solutions. As for the former approach, previous studies concatenate previous and subsequent sentences, separated by a special token, to provide additional information to the model when translating the current phrase [11], [13], [24]. Regarding architectural solutions, these include the integration of multiple encoders to encode not only the source-sentence, but also the context, i.e., the previous or next sentences [14], [25], or to encode the global context of the document [12], [26].

These techniques have recently been applied also in the

software engineering domain, in particular to aid the automatic generation of code comments and commit messages starting from programs. Yu *et al.* [27] integrated local and class-level contextual information for code comment generation. They employ a local encoder, which extracts features from the target function, a global encoder, which exchanges information between all functions within the target class, and a decoder to aggregate local and global information. Wang *et al.* [28] proposed a method to translate *diffs*, i.e., the differences between two versions of code, that include both code changes and non-code changes into commit messages. They adopted retrieval-based solutions to retrieve the most similar commit from the training set to guide the commit message generation.

Our work builds upon previous studies to take advantage of contextual information coming from previous sentences also in the code generation domain. We use the concatenation of multiple intents to guide NMT models in generating more accurate assembly instructions even when the NL code descriptions provided by developers are missing important details.

III. PROBLEM STATEMENT

AI code generation, powered by NMT models, represents a significant leap forward in bridging the gap between the conceptual intent expressed in NL and its translation into code. These advanced tools, trained on extensive datasets of intent-code pairs, have the capability to predict code snippets from a single NL description, encompassing everything from simple instructions to complex multi-line code segments.

However, the deployment of NMT models for code generation is not without its challenges. A primary concern is the inherent variability of NL descriptions, which can significantly impact the models' effectiveness. Indeed, developers rarely provide exhaustive details in their code descriptions, often omitting information that they consider self-evident or redundant [10]. This tendency towards brevity or assumed knowledge poses a significant challenge for NMT models, which rely on the completeness and clarity of the input to generate accurate code. The example provided in TABLE II serves as a case in point, illustrating the discrepancies between human interpretation of NL instructions and the literal output of an NMT model. In this example, the NMT model correctly interprets the first instruction to subtract a value from a byte in the ESI register. However, it falters when presented with a subsequent instruction that refers to the operation's outcome simply as "the result." A human developer would instinctively understand this reference to imply the same register involved in the preceding operation, yet the model fails to make this connection, instead defaulting to a generic placeholder "var" due to the lack of explicit mention of the register's name.

This scenario underscores a critical limitation of current NMT models in code generation: their struggle with inferential reasoning and contextual understanding. For NMT models to be truly effective and reliable in a development setting, they must be capable of beyond-literal interpretation, grasping the unspoken implications of a given instruction based on its context within the broader scope of the code.

TABLE II: Example of incorrect prediction. In red, the word implying the register name that the model fails to derive.

English Intent	Reference Code	Predicted Code
Subtract 8 from the current byte in ESI	`sub byte [ESI], 8`	`sub byte [ESI], 8`
Negate the result	`not ESI`	`not var`

To address these challenges, our study evaluates the impact of missing information on the code generation capabilities of NMT models, exploring their capacity to utilize context from preceding intents and discern the relevance of provided information. This involves examining the extent to which models can compensate for sparse or ambiguous NL descriptions by leveraging additional context and determining the optimal amount of contextual information that aids in accurate code translation without introducing confusion or inaccuracies.

By focusing on these areas, our research aims to shed light on the complex dynamics of AI-driven code generation and identify strategies for enhancing the contextual awareness and inferential capabilities of NMT models. The ultimate goal is to advance the state of AI code generation, ensuring that NMT models can effectively navigate the complexities of software development tasks and contribute to the creation of more intuitive, efficient, and accessible programming environments.

IV. RESEARCH STUDY

To enhance the model's ability to decipher the complexity of NL descriptions for code generation, our study adopts a solution that enriches the contextual awareness (i.e., the *contextual learning*) of NMT models during their training phase. Contextual learning, within the domain of AI code generation, refers to the capacity of NMT models to interpret and utilize the broader sequence of operations implied by a series of coding instructions [29]. Unlike traditional learning approaches that treat each instruction in isolation, contextual learning emphasizes understanding each piece of information within the continuum of previous and forthcoming instructions. This approach enables the model to make informed inferences about ambiguous or incomplete descriptions by leveraging the context provided by surrounding intents.

To enforce contextual learning in NMT models, we adopt a prompt engineering strategy that concatenates preceding intents to the current source intent. This method involves the strategic integration of additional context directly with the current instruction set, demarcated by a specially designed token, the *_BREAK* token [11]. This token serves a pivotal role in delineating the segments of concatenated intents, allowing the model to distinguish between the current actionable intent and its contextual backdrop. Specifically, we design two strategies to provide contextual information:

- **2to1 context:** we concatenate the previous intent to the current source intent, separated by the *_BREAK* token, to form a single NL code description as the input; the corresponding target code snippet represents the output.

3

TABLE III: Examples of 2to1 and 3to1 context-aware intents. **Bold** refers to the current intent to be translated.

Context	English Intent	Reference Code Snippet
2to1	Clear the eax register _BREAK **Move 22 into the lower byte**	`mov al,22`
	Move esi in eax _BREAK **Increment it**	`inc eax`
3to1	Subtract 8 from the current byte in esi _BREAK Negate the result _BREAK **Move eax in it**	`move byte[esi],eax`
	Move eax to edx _BREAK Right shift the register by byte 16 _BREAK **Add the result to eax**	`add eax, edx`

- **3to1 context:** we concatenate the two previous intents to the current source intent, separated by the _*BREAK* token, to form a single NL code description as the input; the corresponding target code snippet represents the output.

TABLE III showcases examples of how the concatenation technique is applied to provide NMT models with contextual information, aiming to enhance their understanding and accuracy in code generation tasks. This technique is part of a strategy to improve model performance, particularly in scenarios where the NL description may lack detailed specificity. The examples are divided into two categories, based on the amount of preceding context provided: 2to1 and 3to1 contexts.

In the 2to1 context scenario, example # 1 shows the intent "Clear the eax register" is followed by _*BREAK* and then the current task "Move 22 into the lower byte." This provides a clear sequence of operations where the model is informed that the register is initially cleared before a new value is moved into it. The corresponding reference code snippet for the combined intents is `mov al, 22`, which directly corresponds to the current task, leveraging the context from the preceding intent. Example # 2 shows another instance combining "Move esi in eax" with the current intent "Increment it," separated by the _*BREAK* token. This contextually rich input instructs the model that the ESI register's value has been moved to EAX before the increment operation, leading to the code `inc eax`.

The 3to1 context setup extends the concept by concatenating two previous intents to the current source intent, again using the _*BREAK* token for separation. The first example starts with "Subtract 8 from the current byte in esi," followed by "Negate the result," and then the current intent "Move eax in it." This sequence offers a comprehensive scenario where a subtraction is performed, the result is negated, and then a move operation is described, culminating in the generation of the code `move byte[esi],eax`. The detailed context aids the model in understanding the series of operations leading to the final action. Example # 2 presents a progression from "Move eax to edx," through "Right shift the register by byte 16," to the current intent "Add the result to eax." This sequence provides a clear narrative of operations that the model can use to generate the corresponding code snippet `add eax,`

`edx`, demonstrating the model's capacity to follow a multi-step process informed by the provided context.

Employing concatenation as a means to provide contextual information necessitates no alterations to the fundamental architecture of the NMT models. This simplicity in implementation, combined with the transformative potential in enhancing code generation accuracy, underscores the effectiveness of the solution. Moreover, the self-attention mechanism inherent in Transformer-based architectures is particularly well-suited to this strategy, enabling the models to dynamically adjust focus and relevance across the concatenated intents to derive a coherent and contextually informed output [24].

A. Research Questions

We designed this research study with the aim of answering the following research questions (RQs):
▷ **R1:** *How do NMT models perform in generating offensive security code from NL descriptions when faced with missing information?*
This question seeks to understand the resilience of NMT models to incomplete or ambiguous NL descriptions, a common scenario in real-world applications. By evaluating the models' ability to fill in the gaps and make informed guesses about missing details, we aim to assess their practical utility in offensive security contexts, where the stakes of misinterpretation are high, and the cost of errors can be significant.
▷ **R2:** *Does contextual learning enhance the robustness of the NMT models in the generation of offensive security code?*
With RQ2, we delve into the potential of leveraging previous intents or contextual clues to improve model performance. This inquiry focuses on the models' ability to use additional, contextually relevant information to understand and accurately execute the current task. By exploring the impact of contextual learning, we seek to determine whether incorporating preceding intents as context can effectively mitigate the challenges posed by sparse or unclear NL descriptions.
▷ **R3:** *Does unnecessary information negatively impact the performance of the NMT models in the generation of offensive security code?*
The third question addresses the potential drawbacks of contextual information, particularly when it is irrelevant or superfluous. This aspect of the research is crucial for understanding the models' ability to discern and filter out unnecessary information, ensuring that their focus remains on relevant details crucial for the accurate generation of code. By investigating the impact of unnecessary context, we aim to reveal insights into how NMT models manage information overload and identify strategies for optimizing their training to improve focus and relevance in code generation tasks.

B. Code Generation Task

To perform a rigorous evaluation of how the use of contextual information coming from previous intents affects the translation ability of the models, we follow the best practices in the field. Hence, we support the models with *data processing* operations. Data processing is an essential step to support the

NMT models in the automatic code generation and refers to all the operations performed on the data used to train, validate and test the models. These operations strongly depend on the specific source and target languages to translate. The data processing steps are usually performed both before translation (*pre-processing*), to train the NMT model and prepare the input data, and after translation (*post-processing*), to improve the quality and the readability of the code in output.

First, we use a corpus to train the NMT models. The *training data* is pre-processed before being used to feed the model. The pre-processing starts with the *stopwords filtering*, i.e., we remove a set of custom-compiled words (e.g., *the*, *each*, *onto*) from the intents to include only relevant data for machine translation. Next, we use a *tokenizer* to break the intents into chunks of text containing space-separated words (i.e., the *tokens*). To improve the performance of the machine translation [15], [30], [31], we *standardize* the intents (i.e., we reduce the randomness of the NL descriptions) by using a *named entity tagger*, which returns a dictionary of *standardizable* tokens, such as specific values, label names, and parameters, extracted through regular expressions. We replace the selected tokens in every intent with "*var#*", where # denotes a number from 0 to $|l|$, and $|l|$ is the number of tokens to standardize. Finally, the tokens are represented as real-valued vectors using *word embedding*. The pre-processed data is used to feed the NMT model. Once the model is trained, we perform the code generation from NL. Therefore, when the model takes as inputs new intents from the *test data* (i.e., data of the corpora not used in the training phase), it generates the corresponding code snippets based on the knowledge inferred during the training (*model's prediction*). As for the intents, also the code snippets predicted by the models are processed (*post-processing*) to improve the quality and readability of the code. First, the dictionary of standardizable tokens is used in the *de-standardization* process to replace all the "*var#*" with the corresponding values, names, and parameters.

Finally, the code snippets generated during the model's prediction are evaluated to assess the quality of the code generation task. The evaluation can be performed through output similarity metrics or manual analysis (human evaluation). The former estimates the quality of the prediction by comparing the model's predictions with the ground truth reference in the test data, the latter, instead, assesses if the output predicted by the model is the correct translation of the NL intent into the generated code snippet.

C. Fine-tuning Dataset

To assess the impact of contextual learning in the generation of offensive security code from NL, we curated a dataset of real-world shellcodes from reputable online databases and developer resources [32], [33]. Since models require not just an understanding of individual code instructions but, crucially, how these instructions interact and depend on each other within a given context, our dataset consists of sequences of code that are contextually interconnected.

TABLE IV: The 20 shellcodes used to build the dataset for the experiments.

id	URL	Lines of Code
1	https://www.exploit-db.com/shellcodes/47564	17
2	https://www.exploit-db.com/shellcodes/47461	32
3	https://www.exploit-db.com/shellcodes/46994	27
4	https://www.exploit-db.com/shellcodes/46519	22
5	https://www.exploit-db.com/shellcodes/46499	16
6	https://www.exploit-db.com/shellcodes/46493	16
7	https://www.exploit-db.com/shellcodes/45529	32
8	https://www.exploit-db.com/shellcodes/43890	23
9	https://www.exploit-db.com/shellcodes/37762	24
10	https://www.exploit-db.com/shellcodes/37495	19
11	https://www.exploit-db.com/shellcodes/43758	29
12	https://www.exploit-db.com/shellcodes/43751	46
13	https://rastating.github.io/creating-a-custom-shellcode-encoder/	27
14	https://voidsec.com/slae-assignment-4-custom-shellcode-encoder/	18
15	https://snowscan.io/custom-encoder/#	42
16	https://github.com/Potato-Industries/custom-shellcode-encoder-decoder	19
17	https://medium.com/@d338s1/shellcode-xor-encoder-decoder-d8360e41536f	33
18	https://www.abatchy.com/2017/05/rot-n-shellcode-encoder-linux-x86	17
19	https://xoban.info/blog/2018/12/08/shellcode-encoder-decoder/	24
20	http://shell-storm.org/shellcode/files/shellcode-902.php	45

In the domain of offensive code generation, shellcode generation represents a critical and widely studied topic [16], [18], [19]. A shellcode is a list of machine code instructions to be loaded in a vulnerable application at runtime. The traditional way to develop shellcodes is to write them using the assembly language, and by using an assembler to turn them into *opcodes* (operation codes, i.e., a machine language instruction in binary format, to be decoded and executed by the CPU) [34], [35]. Common objectives of shellcodes include spawning a system shell, killing or restarting other processes, causing a denial-of-service (e.g., a fork bomb), leaking secret data, etc.

To build the dataset, we first used 20 real shellcodes adopted in previous work to test models [16]. The detailed source and number of lines of code (i.e., complexity), totaling 528 lines of code, of the shellcodes used to build our dataset are shown in TABLE IV. These programs encompass a wide range of functionalities and complexities, ensuring a comprehensive evaluation of the models' code-generation capabilities.

To further extend our collection, we selected the 510 *multi-line snippets* from *Shellcode_IA32* [36], a dataset comprising instructions in assembly language for IA-32 architectures from publicly-available security exploits and described in English. A multi-line sample represents a set of instructions that it would be meaningless to consider as separate because of the strong contextual relation between them. Hereby, these samples are perfectly suited for our training objectives since they embody the contextual relationships between consecutive code snippets. These lines correspond to code descriptions that generate multiple lines of shellcodes (separated by the newline character \n). TABLE V shows an example drawn from the original dataset. In order to ensure a diverse and non-redundant dataset, we carefully checked that there were no duplicates between the 20 collected shellcodes and Shellcode_IA32.

We opted not to include the entire Shellcode_IA32 dataset, as it primarily comprises *single-line* snippets that lack the contextual relationships (e.g., `mov eax, 1`). These snippets do not provide the sequential or logical linkages found in real-

TABLE V: Multi-line snippet from Shellcode_IA32.

Intent: *jump to the label recv http request if the contents of the* `eax` *register is not zero else subtract the value* `0x6` *from the contents of the* `ecx` *register*
Multi-line Snippet: `test eax, eax \n jnz recv_http request \n sub ecx, 0x6`

world programming tasks, which are crucial for testing and enhancing the contextual understanding capabilities of NMT models. Instead, we focused on using multi-line snippets that better represent the interconnected nature of offensive security code to build our dataset.

For the dataset's NL descriptions, we described the code collected from the shellcodes and selected samples from Shellcode_IA32 to test the NMT models' capabilities across a wide array of offensive security coding challenges, yielding a total of $2,167$ pairs of NL descriptions and corresponding code snippets. This process involved selecting instructions that either directly needed contextual understanding for code generation or, conversely, were deliberately disconnected to assess the models' ability to disregard irrelevant information. Specifically, we described the code snippets by using the following notations:

■ **No Context**: Instructions without added contextual information, forming the baseline for model performance assessment. This category includes 963 lines ($\sim 44\%$ of the dataset).
Example: An intent such as "Increment the value in the register" with the code snippet `inc eax`, without any preceding context. This intent is presented without auxiliary information, challenging the model to deduce the target register based on common conventions or inferred knowledge.

■ **2to1 Context**: Incorporating the immediate preceding instruction to provide context, accounting for 360 lines ($\sim 17\%$ of the dataset).
Example: "Clear the eax register **_BREAK** Move 22 into the lower byte" translates to `mov al, 22`, where the context of clearing the register is crucial for understanding the operation.

■ **3to1 Context**: Extending context further by including the two instructions preceding the current one, comprising 238 lines ($\sim 11\%$ of the dataset).
Example: "Subtract 8 from the current byte in esi **_BREAK** Negate the result **_BREAK** Move eax in it" results in `move byte[esi], eax`, showcasing the model's need to synthesize a broader sequence of operations.

■ **2to1 Unnecessary Context**: Featuring previous instructions that do not logically link to the current task, this scenario encompasses 303 lines ($sim 14\%$ of the dataset).
Example: "Define the decode label **_BREAK** Subtract 8 from the current byte of the shellcode" leads to `sub byte[esi], 8`, where the definition of a label is unrelated to the operation on the esi register.

■ **3to1 Unnecessary Context**: Similar to Unnecessary Context 2to1 but with two preceding instructions, also making up 303 lines ($\sim 14\%$ of the dataset).
Example: "Increment edi **_BREAK** Add 3 to al **_BREAK**

Jump short to switch" corresponds to `jmp short switch`, demonstrating the model's challenge in identifying relevant context from unrelated instructions.

The NL descriptions are tied to the relational structure of the code snippets derived from the shellcodes. For the **No Context** scenario, where the objective is to evaluate the models' ability to generate code based solely on standalone instructions, we provided descriptions that omitted contextual dependencies. To better simulate realistic and variable input scenarios, we intentionally modified the original NL descriptions drawn from the Shellcode_IA32 dataset, particularly precise and detailed, to be less accurate. This setup mimics the real-world inaccuracies often encountered in code descriptions provided by different developers. For contexts where the learning of relational dependencies was intended (**2to1 Context** and **3to1 Context**), we selected sequences of instructions that relied on each other to perform a coherent task. This required a description that could only be completed with the understanding of the previous one or two instructions, thus necessitating a detailed contextual description. Conversely, for scenarios meant to assess the models' handling of irrelevant information (**2to1 Unnecessary Context** and **3to1 Unnecessary Context**), we deliberately chose instructions that had no logical connection to each other. This setup aimed to test whether the models could effectively ignore context that does not contribute meaningfully to understanding the current task, thereby simulating conditions where contextual cues might mislead rather than aid in code generation.

The dataset's NL descriptions were independently crafted by three authors, all with a computer science background and expertise in assembly language and cybersecurity. The group included individuals with varying degrees of professional experience and educational qualifications. In particular, one Ph.D. student with a master's degree and two researchers with a Ph.D. in information technologies. The diversity and expertise of the authors ensured the reliability and variety of the dataset's NL descriptions construction process.

V. EXPERIMENTAL SETUP

In our experiments, we used a machine with a Debian-based distribution, 8 vCPU, 16 GB RAM, and one Nvidia T4 GPU.

TABLE VI summarizes the experimental setup across different analyses designed to evaluate the performance of NMT models in generating offensive security code from NL descriptions under various conditions. Each analysis aims to explore a specific aspect of the model's capabilities, such as handling missing information, leveraging contextual learning, and discerning unnecessary context. The dataset for each experiment is systematically divided into *training* (i.e., the data used to fine-tune the models), *validation* (i.e., the data used to tune the model's parameters), and *test* sets (i.e., the data used to evaluate the model in the generation of the code starting from new NL descriptions), adhering to a common ratio of $80\%/10\%/10\%$ [36]–[38], ensuring consistency and reliability in model evaluation.

TABLE VI: Summary of data used in the training (Train), validation (Dev), and testing (Test) sets across different analyses.

Analysis	Objective	Set	w/o Context	2to1 Context	3to1 Context	Unn. 2to1 Context	Unn. 3to1 Context	Total
Missing Information (w/o context)	Establishing a baseline for how well the models perform without any context to assess their innate ability to handle standalone instructions.	Train	770 (80%)	-	-	-	-	770 (80%)
		Dev	96 (10%)	-	-	-	-	96 (10%)
		Test	96 (10%)	-	-	-	-	96 (10%)
Contextual Learning (2to1 context)	Simulating a realistic coding scenario where previous information impacts the current operation.	Train	867 (90%)	180 (50%)	-	-	-	1047 (80%)
		Dev	48 (5%)	90 (25%)	-	-	-	138 (10%)
		Test	48 (5%)	90 (25%)	-	-	-	138 (10%)
Contextual Learning (3to1 context)	Emulating more complex coding scenarios to assess the model's ability in leveraging extended sequences of operations.	Train	867 (90%)	-	81 (34%)	-	-	948 (80%)
		Dev	48 (5%)	-	79 (33%)	-	-	127 (10%)
		Test	48 (5%)	-	79 (33%)	-	-	127 (10%)
Unnecessary Information (2to1 Unn. Context)	Determining the model's ability to discern and filter out unnecessary immediate context.	Train	867 (90%)	324 (90%)	-	103 (34%)	-	1293 (80%)
		Dev	48 (5%)	18 (5%)	-	100 (33%)	-	166 (10%)
		Test	48 (5%)	18 (5%)	-	100 (33%)	-	166 (10%)
Unnecessary Information (3to1 Unn. Context)	Evaluating the model's capacity to ignore multiple irrelevant preceding instructions.	Train	867 (90%)	-	214 (95%)	-	103 (34%)	1084 (80%)
		Dev	48 (5%)	-	12 (5%)	-	100 (33%)	160 (10%)
		Test	48 (5%)	-	12 (5%)	-	100 (33%)	160 (10%)

We employed a consistent split ratio across all experiments to ensure that each model is fine-tuned, parameter-tuned, and evaluated under uniform conditions. The *No Context* set serves as the foundation for comparing the impact of additional or unnecessary context, while the proportional inclusion of contextual scenarios across the train, validation, and test sets allows for a comprehensive assessment of how well NMT models learn from varying degrees of contextual information. The availability of samples for each category also influenced our split. No context samples are more frequent than 2to1 context samples. Similarly, 2to1 context samples are more common than 3to1 context samples. This natural distribution affected how we allocated samples across the splits. Finally, we aimed to have a balanced sampling among training, validation, and test sets for contextual (necessary and unnecessary) information. We also aimed to have a test set size of close to 100 samples for each category to ensure a robust assessment.

Finally, we note that the size of the data used for the experiments is in line with other state-of-the-art corpora used to fine-tune models, which are relatively limited, i.e., in the order of one thousand samples [39]. In fact, in state-of-the-art code generation, a model is not trained from scratch, but existing Large Language Models (that were already trained with millions of publicly available lines of code) are fine-tuned in a supervised way, to achieve transfer learning for the specific case (in our case, generating offensive code). We shared the dataset on GitHub [1].

A. NMT Models

To perform the code generation task, we consider a comprehensive set of state-of-the-art models, which are described in the following. The first three (i.e., CodeBERT, CodeT5+, and PLBart) are based on an *encoder-decoder* architecture, where the input sequence is encoded into a context vector and then decoded to generate the output sequence. The latter (i.e., CodeGen, CodeGPT, and CodeParrot) are *decoder-only* models, which read the input sequence and predict subsequent words one at a time, well-suited for generation tasks.

■ **CodeBERT** [40] is a large multi-layer bidirectional Transformer architecture [41] pre-trained on millions of lines of code across six different programming languages. Our implementation uses an encoder-decoder framework where the encoder is initialized to the pre-trained CodeBERT weights, and the decoder is a transformer decoder, composed of 6 stacked layers. The encoder follows the RoBERTa architecture [42], with 12 attention heads, hidden layer dimension of 768, 12 encoder layers, and 514 for the size of position embeddings. We set the learning rate $\alpha = 0.00005$, batch size = 32, and beam size = 10.

■ **CodeT5+** [43] is a new family of Transformer models pre-trained with a diverse set of pretraining tasks including causal language modeling, contrastive learning, and text-code matching to learn rich representations from both unimodal code data and bimodal code-text data. We utilize the variant with model size $220M$, which is trained from scratch following T5's architecture [44]. It has an encoder-decoder architecture with 12 decoder layers, each with 12 attention heads and hidden layer dimension of 768, and 512 for the size of position embeddings. We set the learning rate $\alpha = 0.00005$, batch size = 16, and beam size = 10.

■ **PLBart** [45] is a multilingual encoder-decoder (sequence-to-sequence) model primarily intended for code-to-text, text-to-code, code-to-code tasks. The model is pre-trained on a large collection of Java and Python functions and natural language descriptions collected from GitHub and StackOverflow. We use the PLBart-large architecture with 12 encoder layers and 12 decoder layers, each with 16 attention heads. We set the learning rate $\alpha = 0.00005$, batch size=16, and beam size=10.

■ **CodeGen** [46], is an autoregressive language model for program synthesis with an architecture that follows a standard transformer decoder with left-to-right causal masking. Specifically, we leverage *CodeGen-350M-Multi*, initialized from CodeGen-NL and further pre-trained on BigQuery [46], a large-scale dataset of multiple programming languages from GitHub repositories, which consists of 119.2B tokens and

[1] https://github.com/dessertlab/Software-Exploits-with-Contextual-Information

7

includes C, C++, Go, Java, JavaScript, and Python.

■ **CodeGPT** [47], is a Transformer-based language model pre-trained on millions of Python functions and Java methods. The decoder-only architecture consists of 12 layers of Transformer decoders with $124M$ parameters. We adopted the *CodeGPT-small-py-adaptedGPT2* version, which has the same GPT-2 vocabulary and natural language understanding ability to support text-to-code generation tasks. We followed previous work for the implementation [47].

■ **CodeParrot** [48] is a GPT-2 model with BPE tokenizer trained on Python code from the training split of the data, and a context length of 1024. This model was released as an educational tool for training large language models from scratch on code, with detailed tutorials and descriptions of the training process. We adopted the *codeparrot-small* version with $110M$ parameters.

During data pre-processing, we tokenize the NL intents using the *nltk word tokenizer* [49] and code snippets using the Python *tokenize* package [50]. We use *spaCy*, an open-source, NL processing library written in Python and Cython [51], to implement the named entity tagger for the standardization of the NL intents.

B. Metrics

Following best practices in the field, we adopted *output similarity metrics* to assess the performance of the models. These metrics, which compare the generated code with the code from the ground truth, are widely used to assess the performance of AI generators in many code generation tasks [52], including the generation of assembly code for security contexts [15]–[19]. In particular, we adopted the following metrics:

■ **Exact Match accuracy (EM)**. It indicates whether each code snippet produced by the model perfectly matches the reference. EM value is 1 when there is an exact match, 0 otherwise. To compute the exact match, we used a simple Python string comparison.

■ **Edit Distance (ED)**. It measures the *edit distance* 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 the reference. ED value ranges between 0 and 1, with higher scores corresponding to smaller distances. For the edit distance, we adopted the Python library `pylcs` [53].

■ **METEOR** [54]. It measures the *alignment* between each code snippet produced by the model and the reference. The alignment is defined as a mapping between unigrams (i.e., 1-gram), such that every unigram in each string maps to zero or one unigram in the other string, and no unigrams in the same string. METEOR value ranges between 0 and 1, with higher scores corresponding to greater alignment between strings. To calculate the METEOR metric, we relied on the Python library `evaluate` by HuggingFace [55].

■ **ROUGE-L**. It is a metric based on the longest common subsequence (LCS) between the model's output and the reference, i.e. the longest sequence of words (not necessarily consecutive, but still in order) that is shared between both. The metric ranges between 0 (perfect mismatch) and 1 (perfect matching).

TABLE VII: Model performance on the generation task with missing context.

Model	EM	ED	METEOR	ROUGE-L
CodeBERT	45.99	77.30	67.41	65.75
CodeT5+	59.35	81.68	74.87	73.44
PLBart	7.44	32.59	21.41	27.04
CodeGen	30.92	60.97	52.72	48.76
CodeGPT	16.60	46.95	38.60	34.17
CodeParrot	21.76	53.15	43.78	40.71
All Models	30.34	58.77	49.80	48.31

We computed the ROUGE-L metric using the Python package `rouge` [56].

VI. EXPERIMENTAL EVALUATION

This section describes the experimental setup and results obtained from evaluating several NMT models on the task of generating assembly shellcodes from NL descriptions. The experiments were designed to assess the models' ability to handle missing information, leverage extended context effectively, and discern and utilize unnecessary information.

A. Missing Information

First, we evaluated the performance of NMT models when generating offensive security code from NL descriptions with no additional contextual information (see TABLE VI). This scenario simulates real-world conditions where developers may provide incomplete or vague descriptions due to oversight or assumption of implicit knowledge. Understanding how NMT models cope with such missing information is crucial for assessing their practical applicability in automated code generation tasks. TABLE VII shows the results.

CodeBERT and CodeT5+ exhibit superior performance across all metrics, with CodeT5+ showing particularly high scores in EM (59.35), METEOR (74.87), and ROUGE-L (73.44). CodeBERT also performs well, with its best score in METEOR (67.41). This indicates a strong capability of the models in accurately translating NL descriptions to code, suggesting a robust understanding of the language and the task requirements even in the absence of context. PLBart struggles significantly in this setup, with much lower scores across the board, particularly in EM (7.44) and ROUGE-L (27.04). This may indicate difficulties in capturing the essence of the NL descriptions and translating them into accurate code snippets without contextual cues. CodeGen, CodeGPT, and CodeParrot display moderate performances, with CodeGen performing relatively better among the three, especially in terms of EM (30.92) and ED (60.97). These models seem to have a moderate capability in understanding and generating code from NL descriptions, with varying degrees of success in handling the missing context. The Average scores across models (EM: 30.34, ED: 58.77, METEOR: 49.80, ROUGE-L: 48.31) reflect a collective moderate proficiency in dealing with NL descriptions devoid of additional context. This underscores a variation in how different models process and generate code

based solely on the information contained within the NL instructions.

To provide context for the evaluation, it's instrumental to consider findings from previous research that tackled a similar task but with more detailed NL descriptions [18]. Specifically, prior work reported EM scores of 48.52 for CodeGPT and 51.80 for CodeBERT, indicating a benchmark for models generating assembly code under conditions of richer linguistic input. In our experiments, CodeGPT and CodeBERT achieved EM scores of 16.60 and 45.99, respectively, when operating without contextual information. This represents a notable decline for CodeGPT compared to the previous EM score of 48.52 and a slight underperformance for CodeBERT relative to its previous benchmark of 51.80. The discrepancy in performance between the current and previous studies, particularly for CodeGPT, underscores the impact of missing information on model output.

> **RQ1: How do NMT models perform in generating offensive security code from NL descriptions when faced with missing information?**
>
> NMT models exhibit varied levels of proficiency in generating offensive security code from less detailed NL descriptions. The comparative analysis reveals a marked impact of missing information on the accuracy of code generation, particularly for CodeGPT, which saw a significant performance dip compared to its prior achievements. Meanwhile, CodeBERT demonstrates a relative steadiness, albeit with room for improvement to match its earlier performance. This juxtaposition highlights the critical role of detailed NL descriptions in achieving high accuracy in code generation and also points to the essential need for enhancing NMT models' capabilities in dealing with sparse information. It emphasizes an ongoing research imperative to develop models that can more effectively bridge the gap between minimalistic NL instructions and the precise requirements of low-level programming tasks, particularly in the contextually rich domain of offensive security.

B. Contextual Learning

Then, we focused on analysing the impact of contextual learning on the performance of NMT models in generating assembly shellcodes from NL descriptions. The extended context was provided in two forms: one additional previous intent (2to1 context) and two additional previous intents (3to1 context) by using the 2to1 Context Test and the 3to1 Context Test, respectively (see TABLE VI). This investigation is particularly important for determining how much contextual information can effectively aid in enhancing model performance, especially in comparison to baseline performance without any extended context. TABLE VIII shows the results of the analysis.

The results show that all models improved performance in the 2to1 context compared to the no-context scenario, with CodeT5+ (EM: 62.40) and CodeBERT (EM: 61.07) leading the pack. This indicates a substantial benefit from including

TABLE VIII: Comparison of models' performance with 2to1 and 3to1 Context.

Model	Context Type	EM	ED	METEOR	ROUGE-L
CodeBERT	2to1	61.07	80.77	74.30	72.29
	3to1	47.90	75.34	67.19	65.84
CodeT5+	2to1	62.40	82.72	77.58	75.50
	3to1	62.79	81.70	77.06	75.14
PLBart	2to1	13.55	35.99	29.36	33.21
	3to1	10.31	24.25	21.63	29.50
CodeGen	2to1	43.70	66.15	60.87	55.91
	3to1	36.26	62.07	53.91	51.43
CodeGPT	2to1	25.38	53.71	44.59	41.14
	3to1	22.14	50.83	41.20	37.96
CodeParrot	2to1	27.67	57.98	48.56	45.24
	3to1	20.99	51.52	42.20	37.16
All Models	2to1	38.96	62.89	55.88	53.88
	3to1	33.40	57.62	50.53	49.51

immediate preceding context, as it likely provides crucial information missing from the standalone instructions. The average EM score increased by 8.62 points, METEOR by 6.08, and ROUGE-L by 5.57, underscoring the utility of adding a single contextual sentence in bridging the gap between the NL descriptions and the required code outputs.

In the 3to1 Context experiments, while CodeT5+ again excels (EM: 62.79), other models like CodeBERT and PLBart show decreased performance compared to the 2to1 context, suggesting a potential overload or diminishing returns from additional context for some models. Indeed, the transition from 2to1 to 3to1 context shows an overall decrease in performance averages across metrics, with a notable drop in EM (-5.56) and METEOR (-5.35). This implies that while some context is beneficial, too much can confuse models or dilute the relevant information. Despite mixed results when transitioning from 2to1 to 3to1 contexts, comparing the 3to1 context results to the no-context scenario reveals an overall enhancement in model performance. The average EM score shows an improvement (3.06 points increase), indicating that even with potential issues of information overload, the inclusion of an extended context offers a net benefit over standalone instructions.

Previous work involving detailed NL descriptions and the use of models like CodeGPT and CodeBERT achieved EM scores of 48.52 and 51.80, respectively. When considering our findings in the 2to1 context experiment, where Code-BERT and CodeT5+ exhibit significant performance boosts (61.07 and 62.40 in EM, respectively), it suggests a parallel in how contextual information can supplement detailed NL descriptions in enhancing model output. The ability of models to perform comparably or even better with added context against a backdrop of detailed descriptions underscores the value of contextual learning in bridging the gap between NL instructions and code generation tasks.

TABLE IX: Comparison of models' performance with 2to1 and 3to1 Unnecessary Context.

Model	Context Type	EM	ED	METEOR	ROUGE-L
CodeBERT	2to1	30.15	63.51	52.03	48.71
	3to1	35.69	65.08	55.68	52.98
CodeT5+	2to1	45.42	71.78	61.99	60.75
	3to1	47.33	74.24	63.59	61.75
PLBart	2to1	16.98	36.37	29.41	32.50
	3to1	7.25	26.37	17.78	21.28
CodeGen	2to1	61.83	81.74	67.92	71.11
	3to1	50.76	73.37	59.78	59.51
CodeGPT	2to1	46.76	72.79	57.35	59.47
	3to1	37.02	66.37	49.70	49.87
CodeParrot	2to1	54.39	77.22	63.25	64.97
	3to1	34.16	66.03	49.70	47.56
All Models	2to1	42.59	67.24	55.33	56.25
	3to1	35.37	61.91	49.37	48.83

RQ2: Does contextual learning enhance the robustness of the NMT models in the generation of offensive security code?

Contextual learning enhances the robustness of NMT models in generating offensive security codes, affirming the positive impact of incorporating contextual information. Specifically, the addition of a single preceding instruction (2to1 context) markedly improves model performance across various metrics, demonstrating that even minimal context can significantly aid models in understanding code generation tasks more accurately. Comparing the 3to1 context to the no-context scenario reveals that extended context still offers benefits, with all models performing better than without any context. This further corroborates the value of contextual information in enhancing model performance. However, the slight decline in metrics when moving from a 2to1 to a 3to1 context suggests an optimal level of contextual information beyond which the effectiveness of additional context may diminish or even detract from model performance.

C. Unnecessary Information

Finally, we explored the impact of incorporating irrelevant contextual information (Unnecessary Context 2to1 and 3to1) on the performance of NMT models in generating offensive security code (see TABLE VI). This setup allows us to investigate the ability of the models to generate precise code in the presence of unnecessary context. TABLE IX shows the results.

The overall performance increased by 12.25, 8.46, 5.53, and 7.94 points for EM, ED, METEOR, and ROUGE-L, respectively, over the no-context baseline, indicating that models are capable of filtering out irrelevant information to some extent and still improving upon the no-context scenario.

Decoder-only models like CodeGen, CodeGPT, and Code-Parrot showed significant improvements in EM scores (61.83,

46.76, and 54.39, respectively), even outperforming their results in the 2to1 context experiment. This suggests that, based on model architecture, the presence of unnecessary context does not hinder—and may even aid—their code generation capabilities. These models rely on causal attention, where each generated token attends to all previous tokens. This allows the model to weigh each input token's importance based on its contribution to the output sequence. When the previous sentence is unrelated, such is in this analysis, it likely receives lower attention weights during the generation process, allowing the model to focus more on the relevant target description.

The transition to unnecessary 3to1 context shows a notable impact on model performance. While CodeT5+ and Code-Gen maintain relatively high performance, there's a notable decrease across all models when directly compared to the unnecessary 2to1 context setup. This decrease underscores the models' challenges in managing more complex distractions. Despite the presence of unnecessary information, models generally maintain or slightly improve performance compared to the no-context baseline, albeit with some performance reduction compared to the 2to1 necessary context setup.

RQ3: Does unnecessary information negatively impact the performance of the NMT models in the generation of offensive security code?

Unnecessary information does not universally negatively impact the performance of NMT models in generating offensive security code. In the unnecessary 2to1 context setup, models demonstrate a remarkable ability to disregard irrelevant information, with decoder-only models even showing improved performance over both the no-context and necessary 2to1 context scenarios. This improvement suggests that models can extract useful patterns or ignore distractions, focusing on the task-relevant aspects of the input, also based on their architecture. However, as the complexity of unnecessary context increases (unnecessary 3to1), we observe a general performance decrease compared to unnecessary 2to1 context, highlighting a limit to the models' ability to filter out noise. This performance decline, though slight, emphasizes the challenge of managing more extensive unrelated information, which can obscure relevant details and impede accurate code generation.

VII. THREATS TO VALIDITY

Models Selection. The external validity of the study might be impacted by the choice of models. To mitigate this, we carefully selected models with distinct architectures and capabilities, ensuring a representation of current advancements in the field [47], [57]–[59]. This careful selection aims to ensure that our findings reflect broader trends in NMT model performance for code generation tasks. Moreover, we did not consider public AI models such as GitHub Copilot and OpenAI ChatGPT because they impose restrictions on malicious uses and, as a consequence, they often "refuse" to generate

offensive code, even if it is used for research or testing activities. Also, since both attackers and defenders need to avoid leaking their techniques and tactics to their counterparts (OPSEC), we consider the case of an attacker or defender that builds her own AI code generator by fine-tuning a model on a dataset of offensive code, thus circumventing usage policies of public AI code generators.

Evaluation Metrics. The reliance on output similarity metrics, although representing the most common solution in the field, poses a potential threat to construct validity, as these metrics may not fully encapsulate the correctness of the generated assembly code [60]. To address this issue, our evaluation strategy encompassed a comprehensive suite of metrics, each offering unique insights into the models' performance. By considering multiple metrics and aligning with common practices in code generation evaluation, we provided a well-rounded assessment. No single metric is perfect, but analyzing them collectively allows for a comprehensive evaluation of the code.

Dataset Construction and NL Description. The selection of 20 real shellcodes and the multi-line samples from Shellcode_IA32 to build the dataset aimed to provide a broad overview of common tasks and challenges in this domain. However, the inherent variability in exploit development and the continuous evolution of offensive techniques mean that no dataset can fully encapsulate the entire scope of offensive coding. To mitigate this threat, we ensured that the chosen shellcodes span a range of functionalities, complexities, and objectives, aiming for a balanced representation of real-world coding tasks in offensive security. Regarding the NL descriptions of the shellcodes, we aimed to mimic the variability in detail and specificity that might be encountered in real-world scenarios. Nevertheless, we acknowledge that the manual description of shellcodes introduces subjectivity and could potentially influence the models' performance. To minimize bias and ensure consistency, multiple authors independently described different samples of the dataset, and, where available, we used developers' original comments as NL descriptions. This multi-faceted approach to dataset annotation seeks to capture a realistic range of expression and technical detail, enhancing the external validity of our findings.

VIII. CONCLUSION

This work presented a comprehensive investigation into the capabilities of NMT models in the domain of offensive security code generation. Through a series of designed experiments, we investigated the complexities surrounding the models' interaction with missing information, their utilization of contextual learning, and their resilience in the face of unnecessary context.

Our results showed that contextual information substantially improved model performance, highlighting the value of contextual information in dealing with undetailed NL descriptions. However, the results also showed the diminishing returns associated with extensive context, pointing to the existence of an optimal context threshold that maximizes model performance. In scenarios involving unnecessary context, our

experiments demonstrated the models' ability to filter out irrelevant information, maintaining high performance in the code generation task.

REFERENCES

[1] Y. Mirsky, A. Demontis, J. Kotak, R. Shankar, D. Gelei, L. Yang, X. Zhang, M. Pintor, W. Lee, Y. Elovici *et al.*, "The threat of offensive ai to organizations," *Computers & Security*, p. 103006, 2022.

[2] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[3] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 25–36.

[4] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 336–347.

[5] R. Natella, P. Liguori, C. Improta, B. Cukic, and D. Cotroneo, "Ai code generators for security: Friend or foe?" *IEEE Security & Privacy*, pp. 2–10, 2024.

[6] P. Liguori, C. Marescalco, R. Natella, V. Orbinato, and L. Pianese, "The power of words: Generating PowerShell attacks from natural language," in *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 27–43. [Online]. Available: https://www.usenix.org/conference/woot24/presentation/liguori

[7] Z. Wang, "Study on the importance of cultural context analysis in machine translation," in *Proceedings of The Eighth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA), 2013*. Springer, 2013, pp. 29–35.

[8] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *2013 21st international conference on program comprehension (icpc)*. Ieee, 2013, pp. 83–92.

[9] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the robustness of code generation techniques: An empirical study on github copilot," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2149–2160. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00181

[10] C. Improta, P. Liguori, R. Natella, B. Cukic, and D. Cotroneo, "Enhancing robustness of ai offensive code generators via data augmentation," *arXiv preprint arXiv:2306.05079*, 2023.

[11] J. Tiedemann and Y. Scherrer, "Neural machine translation with extended context," in *Proceedings of the Third Workshop on Discourse in Machine Translation, DiscoMT@EMNLP 2017, Copenhagen, Denmark, September 8, 2017*, B. L. Webber, A. Popescu-Belis, and J. Tiedemann, Eds. Association for Computational Linguistics, 2017, pp. 82–92. [Online]. Available: https://doi.org/10.18653/v1/w17-4811

[12] L. Wang, Z. Tu, A. Way, and Q. Liu, "Exploiting cross-sentence context for neural machine translation," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, M. Palmer, R. Hwa, and S. Riedel, Eds. Association for Computational Linguistics, 2017, pp. 2826–2831. [Online]. Available: https://doi.org/10.18653/v1/d17-1301

[13] Y. Scherrer, J. Tiedemann, and S. Loáiciga, "Analysing concatenation approaches to document-level NMT in two different domains," in *Proceedings of the Fourth Workshop on Discourse in Machine Translation (DiscoMT 2019)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 51–61. [Online]. Available: https://aclanthology.org/D19-6506

[14] E. Voita, P. Serdyukov, R. Sennrich, and I. Titov, "Context-aware neural machine translation learns anaphora resolution," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, I. Gurevych and Y. Miyao, Eds. Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 1264–1274. [Online]. Available: https://aclanthology.org/P18-1117

[15] 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 Software Engineering*, vol. 29, no. 1, p. 30, 2022.

[16] P. Liguori, E. Al-Hossami, V. Orbinato, R. Natella, S. Shaikh, D. Cotroneo, and B. Cukic, "Evil: exploiting software via natural language," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 321–332.

[17] G. Yang, X. Chen, Y. Zhou, and C. Yu, "Dualsc: Automatic generation and summarization of shellcode via transformer and dual learning," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 2022, pp. 361–372.

[18] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Han, and T. Chen, "Exploitgen: Template-augmented exploit code generation based on codebert," *Journal of Systems and Software*, vol. 197, p. 111577, 2023.

[19] X. Ruan, Y. Yu, W. Ma, and B. Cai, "Prompt learning for developing software exploits," in *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, 2023, pp. 154–164.

[20] D. Xu, K. Chen, M. Lin, C. Lin, and X. Wang, "Autopwn: Artifact-assisted heap exploit generation for ctf pwn competitions," *IEEE Transactions on Information Forensics and Security*, 2023.

[21] M. Botacin, "Gpthreats-3: Is automatic malware generation a threat?" in *2023 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2023, pp. 238–254.

[22] Y. M. Pa 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 *Proceedings of the 16th Cyber Security Experimentation and Test Workshop*, 2023, pp. 10–18.

[23] 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*, 2023.

[24] R. R. Agrawal, M. Turchi, and M. Negri, "Contextual handling in neural machine translation: Look behind, ahead and on both sides," in *Proceedings of the 21st Annual Conference of the European Association for Machine Translation*, 2018, pp. 11–20.

[25] B. Li, H. Liu, Z. Wang, Y. Jiang, T. Xiao, J. Zhu, T. Liu, and C. Li, "Does multi-encoder help? A case study on context-aware neural machine translation," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, D. Jurafsky, J. Chai, N. Schluter, and J. R. Tetreault, Eds. Association for Computational Linguistics, 2020, pp. 3512–3518. [Online]. Available: https://doi.org/10.18653/v1/2020.acl-main.322

[26] Z. Zheng, X. Yue, S. Huang, J. Chen, and A. Birch, "Towards making the most of context in neural machine translation," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, C. Bessiere, Ed. ijcai.org, 2020, pp. 3983–3989. [Online]. Available: https://doi.org/10.24963/ijcai.2020/551

[27] X. Xu, Q. Huang, Z. Wang, Y. Feng, and D. Zhao, "Towards context-aware code comment generation," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, 2020, pp. 3938–3947.

[28] H. Wang, X. Xia, D. Lo, Q. He, X. Wang, and J. Grundy, "Context-aware retrieval-based deep commit message generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–30, 2021.

[29] Y. Hwang, H. Yun, and K. Jung, "Contrastive learning for context-aware neural machine translation using coreference information," in *Proceedings of the Sixth Conference on Machine Translation*, 2021, pp. 1135–1144.

[30] Z. Li, X. Wang, A. Aw, E. S. Chng, and H. Li, "Named-entity tagging and domain adaptation for better customized translation," in *Proceedings of the seventh named entities workshop*, 2018, pp. 41–46.

[31] M. Modrzejewski, M. Exel, B. Buschbeck, T.-L. Ha, and A. Waibel, "Incorporating external annotation to improve named entity translation in nmt," in *Proceedings of the 22nd Annual Conference of the European Association for Machine Translation*, 2020, pp. 45–51.

[32] Exploit-db, "Exploit Database Shellcodes," https://www.exploit-db.com/shellcodes?platform=linux_x86/, 2023.

[33] Shell-storm, "Shellcodes database for study cases," http://shell-storm.org/shellcode/, 2022.

[34] J. Foster, *Sockets, Shellcode, Porting, and Coding: Reverse Engineering Exploits and Tool Coding for Security Professionals*. Elsevier Science, 2005. [Online]. Available: https://books.google.it/books?id=ZNI5dvBSfZoC

[35] H. Megahed, *Penetration Testing with Shellcode: Detect, exploit, and secure network-level and operating system vulnerabilities*. Packt Publishing, 2018.

[36] P. Liguori, E. Al-Hossami, D. Cotroneo, R. Natella, B. Cukic, and S. Shaikh, "Shellcode_IA32: A dataset for automatic shellcode generation," in *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*. Online: Association for Computational Linguistics, Aug. 2021, pp. 58–64. [Online]. Available: https://aclanthology.org/2021.nlp4prog-1.7

[37] D. Kim and T. MacKinnon, "Artificial intelligence in fracture detection: transfer learning from deep convolutional neural networks," *Clinical radiology*, vol. 73, no. 5, pp. 439–445, 2018.

[38] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 2021, pp. 505–509.

[39] C. Zhou, P. Liu, P. Xu, S. Iyer, J. Sun, Y. Mao, X. Ma, A. Efrat, P. Yu, L. Yu, S. Zhang, G. Ghosh, M. Lewis, L. Zettlemoyer, and O. Levy, "LIMA: less is more for alignment," *CoRR*, vol. abs/2305.11206, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.11206

[40] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, ser. Findings of ACL, vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547. [Online]. Available: https://doi.org/10.18653/v1/2020.findings-emnlp.139

[41] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[42] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[43] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.

[44] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, pp. 140:1–140:67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html

[45] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021, pp. 2655–2668.

[46] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. [Online]. Available: https://openreview.net/pdf?id=iaYcJKpY2B_

[47] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue:

A machine learning benchmark dataset for code understanding and generation," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html

[48] huggingface, "CodeParrot," https://huggingface.co/codeparrot/codeparrot, 2024.

[49] S. Bird, "Nltk: the natural language toolkit," in *Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions*, 2006, pp. 69–72.

[50] Python, "tokenize," 2023. [Online]. Available: https://docs.python.org/3/library/tokenize.html

[51] spaCy, "Industrial-Strength Natural Language Processing," 2023. [Online]. Available: https://spacy.io/

[52] 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 Systems with Applications*, vol. 225, p. 120073, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417423005754

[53] pylcs, "Python library pylcs," 2023. [Online]. Available: https://pypi.org/project/pylcs/

[54] A. Lavie and A. Agarwal, "Meteor: An automatic metric for mt evaluation with high levels of correlation with human judgments," in *Proceedings of the Second Workshop on Statistical Machine Translation*, ser. StatMT '07. USA: Association for Computational Linguistics, 2007, p. 228–231.

[55] evaluate, "Python library evaluate," 2022. [Online]. Available: https://pypi.org/project/evaluate/

[56] rouge, "Python ROUGE Score Implementation," 2021. [Online]. Available: https://pypi.org/project/rouge/

[57] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 172–184.

[58] S. Tipirneni, M. Zhu, and C. K. Reddy, "Structcoder: Structure-aware transformer for code generation," *arXiv preprint arXiv:2206.05239*, 2022.

[59] D. Cotroneo, C. Improta, P. Liguori, and R. Natella, "Vulnerabilities in ai code generators: Exploring targeted data poisoning attacks," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 280–292.

[60] D. Cotroneo, A. Foggia, C. Improta, P. Liguori, and R. Natella, "Automating the correctness assessment of ai-generated code for security contexts," *Journal of Systems and Software*, p. 112113, 2024.