This is a postprint of an article whose final and definitive form is available online at:

Improta, C., Liguori, P., Natella, R. *et al.* Reading between the Lines: Context-Aware AI-based generation of software exploits. *Empir Software Eng* 31, 60 (2026). https://doi.org/10.1007/s10664-025-10796-x.

manuscript No.
(will be inserted by the editor)

Reading between the Lines: Context-Aware AI-based Generation of Software Exploits

Cristina Improta* · **Pietro Liguori** · **Roberto Natella** · **Bojan Cukic** · **Domenico Cotroneo**

Received: date / Accepted: date

*Corresponding author

Abstract AI-based code generators have transformed offensive security by translating natural language descriptions into executable exploits. However, the semantic variability and implicit assumptions in NL descriptions limit their robustness and usability in this domain. This study evaluates nine state-of-the-art DL models, including fine-tuned models and instruction-tuned LLMs, under varying contextual information conditions to assess their ability to handle ambiguity, leverage useful context, and filter irrelevant information. Using a manually-curated dataset of real-world shellcodes and rigorous evaluations, we find that fine-tuned encoder-decoder models excel with related context, decoder-only indirectly benefit from unrelated context to better comprehend the task at hand, while instruction-tuned LLMs struggle to utilize context effectively, regardless of the prompting setting. These results underline the importance of optimized contextual strategies and task-specific fine-tuning for

Cristina Improta
University of Naples Federico II, Italy
E-mail: cristina.improta@unina.it

Pietro Liguori
University of Naples Federico II, Italy
E-mail: pietro.liguori@unina.it

Roberto Natella
University of Naples Federico II, Italy
E-mail: roberto.natella@unina.it

Bojan Cukic
University of North Carolina at Charlotte, USA
E-mail: bcukic@charlotte.edu

Domenico Cotroneo
University of North Carolina at Charlotte, USA
E-mail: dcotrone@charlotte.edu

advancing AI-driven exploit generation for high-stakes applications in software security.

Keywords AI Offensive Code Generation · Offensive Security · Assembly · Large Language Models · Contextual Information

1 Introduction

The automation of code generation has revolutionized software development, bridging the gap between human intent expressed in natural language (NL) and the generation of executable code. By simplifying traditionally effort-intensive software development processes, AI-driven tools have proven to be invaluable across diverse programming tasks. This advancement has had a significant impact on domains that demand high technical skills, such as *offensive security*, where the automation of coding tasks promises to boost productivity and innovation [47,50,86].

Offensive security focuses on proactively identifying and exploiting vulnerabilities by developing *proof-of-concept* attacks, providing critical insights into system weaknesses and driving timely security patches and mitigations [3]. Despite its importance, the process of crafting software exploits remains highly demanding, requiring in-depth knowledge of low-level programming languages (e.g., assembly), memory layouts, and system architectures. This complexity, combined with the significant time required, underscores the necessity for AI-driven tools for automatic exploit generation (AEG). By translating intents detailing the exploitation of system vulnerabilities into actionable code, AI code generators, relying on deep learning (DL) models, can automate and accelerate exploit development, enabling security analysts to focus on higher-level strategic tasks [50,38].

Despite their potential, the application of AI code generators to offensive security presents specific challenges. Effective exploit development relies on precise descriptions of low-level system operations, yet natural language specifications often lack explicit details or rely on implicit assumptions. The expertise, terminology and usage of NL by developers vary widely [77,68], making it difficult for AI code generators to generate accurate and functional code when faced with missing or implied information.

Human developers naturally fill in these gaps by leveraging domain knowledge and prior information, but DL models struggle to replicate this task-specific reasoning. These models tend to process inputs in isolation, limiting their ability to incorporate sequential information, a crucial capability for handling real-world offensive security tasks that frequently rely on assumed knowledge and contextual dependencies.

For instance, a common sequence in system exploitation involves loading a value into a register, modifying it, and then storing the result in memory. In exploit development, certain registers serve specific purposes, such as storing pointers, holding return addresses, or facilitating arithmetic operations, and this knowledge is often assumed rather than explicitly stated. Given the NL

description "*store the result in memory*", a human developer would intuitively infer that "the result" refers to the previously computed value, likely stored in a register commonly used for such operations. However, a DL model, lacking contextual awareness and an understanding of these implicit conventions, may fail to establish this connection, leading to incorrect code generation.

This illustrates a fundamental challenge in AI-driven exploit generation: the inherent variability of NL, coupled with the inability of DL models to recognize implicit dependencies, undermines their robustness to implicit or missing information. Without mechanisms to enhance robustness and contextual understanding, the usability of these models in security-critical applications, such as exploit development, remains limited [45, 26].

This study addresses this challenge by systematically evaluating: *(i)* the robustness of DL models to natural language descriptions that lack explicit details or rely on implicit knowledge; *(ii)* the impact of contextual information, whether relevant or not directly related, in mitigating NL ambiguities and improving model performance in automatic exploit generation. To this end, we design a *context-aware training and prompting strategy* that feeds models with contextual information by using the concatenation of inputs, i.e., by merging previous NL descriptions with the current one [69, 74, 65, 72]. Our aim is to critically examine how AI code generators respond to varying types of contextual input, including scenarios with no additional context, useful related information, and unrelated or irrelevant context, aiming to assess their ability to handle ambiguity, infer missing information, and disregard worthless details.

Our evaluation spans nine state-of-the-art DL models, encompassing fine-tuned encoder-decoder and decoder-only architectures, alongside instruction-tuned LLMs evaluated under both zero-shot and few-shot (4-shot) prompting strategies. Central to this analysis is a carefully constructed dataset of real-world shellcodes, curated to reflect the complexities and nuances of authentic offensive security tasks. This corpus features NL descriptions tailored to challenge the models by varying levels of detail and context dependency, testing their capacity for contextual understanding and robustness. To ensure a rigorous assessment of the quality of offensive code, we employ a combination of five state-of-the-art output similarity metrics and symbolic execution via *ACCA* [10], evaluating both syntactic and semantic correctness.

We explore how DL models respond to missing information, the impact of additional context, and the influence of irrelevant inputs, addressing four research questions that examine these factors' effects on code generation accuracy and robustness. The findings provide actionable insights into selecting and optimizing DL models for offensive security, highlighting strategies to enhance their performance in real-world applications.

These key findings can be summarized as follows:

- **Missing Information.** Models struggle more with highly technical and specific commands that lack relevant details, underscoring the need for detailed NL descriptions in security-related tasks.

- **Relevant Contextual Information.** Introducing additional context significantly improves fine-tuned models' performance, with a notable enhancement in generating complex code. Instruction-tuned LLMs, although flexible, do not benefit from additional information to the same extent, as their general-purpose training often lacks the domain-specific focus needed to fully understand complex task-specific relationships. In all cases, there is a diminishing return on performance when extending context beyond a single preceding intent, indicating an optimal level of contextual information for effective generation.
- **Unrelated Information.** Not directly related information does not degrade fine-tuned models' performance: encoder-decoder models are able to maintain discrete performance, while decoder-only models significantly enhance their capabilities, indicating the effective ability to indirectly benefit from additional cues and filter out unnecessary information. Conversely, LLMs under both prompting strategies show marked sensitivity to unrelated inputs, with performance deteriorating sharply compared to the baseline.
- **Syntactic & Semantic Correctness.** Contextual information, both related and unrelated, significantly impacts the syntactic and semantic correctness of AI-generated offensive security code, with clear trends across model categories, underscoring the need for a tailored training and prompting strategy based on the target DL model.

This work extends our conference paper published at ISSRE 2024 [37], significantly expanding the scope and depth of the study with key contributions and new results. We introduce three large language models, enabling comparisons under both zero-shot and few-shot prompting settings, and a broader comparative analysis across different model categories. We extend the evaluation methodology by including a new code-specific similarity metric, CrystalBLEU, and by assessing both the syntactic and semantic correctness of generated code using *ACCA*, a symbolic execution framework, which forms the basis of a new research question (RQ_4). Our new findings reveal that incorporating additional contextual information significantly enhances the performance of task-specific fine-tuned models, which show notable improvements in generating syntactically and semantically correct code. Encoder-decoder models (e.g., CodeT5+) excel when guided by contextually relevant information, while decoder-only models (e.g., CodeGen) benefit from non-directly related information to better focus on the target task. In contrast, instruction-tuned LLMs (e.g., DeepSeek-Coder, Qwen2.5-Coder, StableCode) remain weaker overall than fine-tuned models, with their effectiveness strongly influenced by prompt design and model size. Few-shot prompting does not consistently close this performance gap, further emphasizing the importance of task-specific fine-tuning for reliable deployment in high-stakes domains like offensive security.

In the following, Section 2 discusses related work; Section 3 describes our research study; Section 4 details our experimental setup; Section 5 presents the experimental evaluation; Section 6 discusses our findings and practical

implications; Section 7 discusses threats to validity; Section 9 concludes the paper.

2 Related Work

The automatic exploit generation (AEG) research aims at automatically developing working exploits [3]. This task is inherently complex since it requires technical expertise in low-level languages to manipulate memory layouts and CPU registers, and to attack mechanisms like heap metadata and stack return addresses, tasks that are inaccessible through high-level programming. Since the time and expertise required for these operations make AEG particularly demanding, AI code generators have emerged as promising tools to help developers and security analysts face these challenges.

Liguori *et al.* [34] 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 [35]. Yang *et al.* [82] addressed software exploit generation and summarization as a dual learning problem and later extended their work with a template-augmented approach, employing rule-based parsers and semantic attention layers to enhance NL descriptions [83]. Similarly, Ruan *et al.* [64] proposed *PT4Exploits*, utilizing prompt tuning to build relationships between English comments and corresponding code snippets, simulating pre-training to leverage prior knowledge. Xu *et al.* [79] introduced *AutoPwn*, an artifact-assisted automatic exploit generation (AEG) system for heap exploits, guiding generation through patterns extracted from known exploits.

Recent work also explored GPT-based models in offensive security. Botacin [6] demonstrated their use in creating and deobfuscating malware by segmenting malicious behaviors into smaller tasks. Pa *et al.* [54] and Gupta *et al.*[23] showed that attackers could use *jailbreak prompts* to bypass safeguards in AI code generators, facilitating malware creation. Gupta *et al.* also highlighted the potential of AI in enhancing security measures, such as cyber defense automation, threat intelligence, and secure code generation.

Although these solutions have shown high accuracy in the generation of software exploits starting from NL descriptions, their robustness to incomplete or missing information has not been thoroughly explored. Indeed, these studies do not take into account that, since developers have different levels of expertise and writing styles, real-world code descriptions may be missing some information or reference previous instructions that can negatively affect the performance of AI code generators. Improta *et al.* [26] proposed a data augmentation method to assess and enhance AI code generators' robustness to unseen synonyms and missing information by introducing variations in NL descriptions to simulate real-world conditions. However, their work heavily re-

lies on synthetically generated perturbations, which may not fully capture the diversity of real-world NL descriptions, and does not address the mechanisms required for models to infer missing context dynamically from surrounding descriptions, an essential capability for robust exploit generation.

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 [69,65,1]. 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 [31,72], or to encode the global context of the document [84,74]. In software engineering, Yu *et al.* [80] utilized contextual encoders for code comment generation, combining function-level and class-level features. Wang *et al.* [73] translated code differences (*diffs*) into commit messages using retrieval-based guidance.

Recent research demonstrates that incorporating additional contextual information can significantly enhance performance in source code-related tasks. Ciniselli *et al.* [9] achieved a boost in code completion of programs written in Java by incorporating coding, process, and developer context. Pan *et al.* [55] introduced CatCoder, a framework that enhances repository-level code generation by integrating type dependencies and retrieved code for statically typed programming languages such as C, C++, Java, and Rust. Kapu *et al.* [27] proposed DemoCraft, which leverages demonstration selection and latent concept learning to improve Python and C++ code generation.

Current work also investigated the impact of designing program semantics- and execution-aware NL descriptions in improving code generation. Ding *et al.* [14] introduced *SemCoder*, a semantics-aware framework that enriches code pre-training with control-flow and data-flow reasoning. Similarly, Ni *et al.* [51] proposed *NExT*, which teaches LLMs to reason about execution by predicting intermediate states and traces.

Both SemCoder and NExT place NL at the center of their methodology, but in ways that differ fundamentally from ours. SemCoder pairs executable Python code with functional NL descriptions and monologue-style semantic rationales that are automatically checked against test cases, while NExT couples buggy programs with execution traces and chain-of-thought rationales describing how to correct them. In contrast, our dataset concerns low-level assembly shellcode, where executing code to obtain traces is unsafe and often infeasible due to the issues related to the execution of malicious payloads. We therefore rely on expert-validated contextual NL descriptions that explicitly encode relevant or irrelevant preceding instructions without requiring runtime execution. Although all three studies share the goal of enriching code with NL explanations, our focus is on the importance of the contextual information in

Table 1 Example of incorrect prediction. In red, the word implying the register name that the model fails to derive.

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

the NL inputs. In contrast, SemCoder and NExT emphasize execution-trace-driven functional reasoning for high-level languages.

Our work extends previous research by investigating how contextual information from surrounding sentences can improve robustness in code generation, particularly for the challenging task of software exploit generation. We concatenate NL code descriptions to provide additional context, enabling DL models to generate more accurate offensive code even when input descriptions are incomplete or missing critical details. Our approach focuses on dynamically inferring missing details directly from the surrounding context across varying settings, providing a comprehensive analysis of how different types of DL models adapt to contextual challenges.

3 Study Design

A major limitation of existing AI code generators lies in their inability to handle the inherent variability and ambiguity of NL descriptions [26]. Exploit descriptions frequently use ad hoc terminology and vary significantly based on developers' expertise, writing styles, and assumed knowledge. Developers frequently omit details they consider implicit, leading to ambiguous inputs that hinder model robustness and real-world applicability. As a result, models struggle to infer unspoken connections or dynamically resolve references to prior operations.

Table 1 illustrates this issue in an extract of a proof-of-concept shellcode that opens a shell on a victim system (see Table 3, id 1 for the complete code). Although the model correctly interprets the first instruction to subtract a value from the ESI register, it fails when the subsequent instruction refers to the previous result without explicitly naming the register. A human developer would instinctively associate the "result" with the same register, but the model defaults to a placeholder ("var"), highlighting its difficulty in processing implicit references.

To address these limitations, we design our research study to evaluate how missing information affects the ability of DL models to generate correct code, 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. We investigate how different types of models, rang-

ing from task-specific fine-tuned models to general-purpose LLMs, interpret contextual cues, distinguishing relevant from irrelevant information.

3.1 Research Questions

We designed this research study with the aim of answering the following research questions (RQs):

▷ **RQ$_1$:** *How do DL models perform in generating offensive security code from NL descriptions when faced with missing information?*
The goal of this research question is to assess the ability of deep learning models to generate accurate offensive security code when critical details are absent from the natural language descriptions, a common scenario in real-world applications. By examining their capability to infer missing or implicit details and generate functional code despite incomplete inputs, the study aims to assess their reliability and practical utility in the offensive security domain.

▷ **RQ$_2$:** *Does related contextual information enhance the robustness of DL models in the generation of offensive security code?*
With this research question, we investigate the potential of leveraging previous intents and contextual information to improve model performance. This analysis focuses on the models' ability to use additional, contextually relevant information to understand and accurately execute the current task. By exploring the impact of additional information, we aim to determine whether incorporating preceding intents as context can effectively mitigate the challenges posed by sparse or unclear NL descriptions.

▷ **RQ$_3$:** *Does unrelated information negatively impact the performance of DL models in the generation of offensive security code?*
The third question addresses the potential drawbacks of contextual information, particularly when it is not directly related 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 unrelated context, we aim to reveal insights into how DL models manage information overload and identify strategies for optimizing their training to improve focus and relevance in code generation tasks.

▷ **RQ$_4$:** *To what extent does contextual information (or lack thereof) influence the syntactic and semantic correctness of AI-generated offensive security code?*
To address this research question, we conduct an in-depth analysis of the exploit code generated by the models, evaluating its syntactic and semantic correctness against a reference implementation. This examination goes beyond conventional performance evaluation using state-of-the-art similarity metrics. We evaluate the impact of related and unrelated contextual information on the

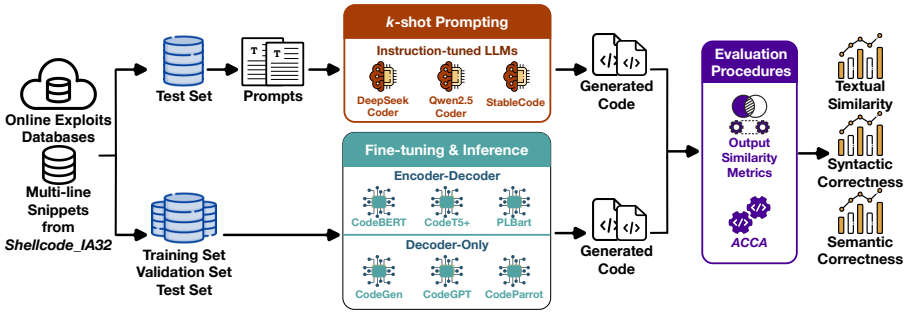

Fig. 1 Overview of the methodology.

models' code generation capabilities by determining whether the generated code is *syntactically correct* (i.e., compilable) and *semantically correct* (i.e., an accurate implementation of the NL description). This assessment leverages *ACCA* [10], which employs symbolic execution to rigorously analyze and validate the outputs.

3.2 Leveraging Contextual Information

To enhance the ability of deep learning models to interpret the complexities of natural language descriptions for exploit generation, our study employs a method that integrates preceding instructions as additional input context. This approach enriches the model's understanding of the sequence and relevance of operations described in NL, improving its capability to generate accurate and contextually appropriate code snippets. Unlike traditional learning approaches that treat each instruction in isolation, our solution enables the model to make informed inferences about ambiguous or incomplete descriptions by inferring the implicit or missing information from the surrounding intents.

To enforce context comprehension in DL models, we design two complementary strategies that are tailored to the training paradigm of the models, including fine-tuned models and k-shot prompted LLMs. Fig. 1 shows an overview of our methodology. For fine-tuned models, we adopt a context-enriched fine-tuning approach. This involves concatenating preceding intents with the current NL description, separated by a custom delimiter token, the *_BREAK* token [69], during the training phase. By incorporating these contextual context-enriched data into the training process, the models' parameters are updated to enhance their ability to leverage context effectively during inference, resulting in more accurate and semantically coherent code generation.

For large language models, we implement a context-aware prompting strategy. Instead of modifying the model's parameters, we concatenate previous intents with the current input at inference time to provide additional context. This k-shot prompting technique relies on the inherent capabilities of instruction-tuned LLMs to process structured inputs and interpret context.

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

Context Length	Context-Aware NL Description	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`

The same delimiter token (*_BREAK*) serves as a clear separator, ensuring the models can differentiate between the current instruction and its surrounding context. While this strategy does not involve training or fine-tuning, it enables LLMs to leverage their pre-trained knowledge to address tasks that require contextual reasoning.
To effectively integrate context, we employ two configurations:

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

Table 2 showcases examples of how the concatenation technique is applied to provide DL models with contextual information. 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" 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 of which 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`.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 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 helps the model understand the series of operations that lead 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.

This dual concatenation approach allows for flexible integration of context across different model types, without the need for alterations to their fundamental architecture. Fine-tuned models benefit from incorporating contextual information during training, leveraging the self-attention mechanism of Transformer architectures to dynamically adjust focus and prioritize relevant information within concatenated inputs [1]. In contrast, prompted LLMs utilize structured input modifications during inference, providing a lightweight yet effective solution for tasks that do not permit fine-tuning or require rapid adaptability. Using both fine-tuning and prompting strategies, this study demonstrates how context can be systematically used to improve model performance across diverse settings. These approaches enable models to address the challenges posed by ambiguous or incomplete NL descriptions, advancing the state of AI-driven code generation in offensive security applications.

3.3 Dataset Construction

To evaluate the impact of contextual information on the generation of offensive security code from NL descriptions, we curated a dataset of real-world shellcodes sourced from reputable online databases and developer resources [18,66]. Since effective code generation requires understanding individual instructions and recognizing their inter-dependencies within a given context, the dataset was designed to consist of sequences of contextually interconnected code.

In offensive code generation, shellcode generation represents a critical and widely studied topic [83,35,64]. A shellcode is a series of machine code instructions to be loaded into a vulnerable application at runtime. Traditionally, shellcodes are written in assembly language and compiled into *opcodes* (binary machine language instructions) to be executed by the CPU [20,46]. Common objectives include spawning system shells, killing or starting processes, causing denial-of-service (e.g., fork bombs), or extracting sensitive data.

To construct the dataset, we began with 20 real shellcodes previously used to test models [35], totaling 528 lines of code. These shellcodes, sourced from public repositories, encompass diverse functionalities and complexities, ensuring a comprehensive evaluation of the models' capabilities. Table 3 lists the shellcodes along with their source URLs and line counts.

Table 3 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

Table 4 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`

We further extended the dataset by incorporating the 510 *multi-line snippets* from *Shellcode_IA32* [33], a corpus comprising instructions in assembly language for IA-32 architectures from publicly-available security exploits and described in English. A multi-line sample represents a sequence of instructions that it would be meaningless to consider as separate because of the strong contextual relation between them. Hence, 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 4 shows an example drawn from the original dataset.

We carefully reviewed the dataset to identify and remove any overlaps between the multi-line snippets and the initial 20 shellcodes, ensuring a diverse, non-redundant collection of samples for training and evaluation.

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-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 NL descriptions of the dataset, we manually described the code collected from the shellcodes and selected samples from Shellcode_IA32 to

train and test the capabilities of the DL models in a wide range 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 unrelated information.

All NL descriptions were carefully created and validated by a small team of security and assembly experts, building on prior experience with the Shellcode_IA32 dataset [33,34]. In that earlier effort, we established a rigorous methodology: multiple authors independently described assembly snippets in English, cross-checking one another and incorporating comments written by the original shellcode developers or extracted from well-known assembly tutorials and textbooks to capture the natural variability of human descriptions. Only after a consistent descriptive style had emerged did we move on to real shellcodes, writing new descriptions when no documentation was available. We applied the same principles here. Each snippet was independently described by one of three annotators (one Ph.D. student and two Ph.D. researchers, all with assembly and offensive-security expertise); disagreements or ambiguities were resolved through group discussion until a consensus was reached. This multi-annotator, consensus-driven process, together with systematic checks for consistency and strict alignment with the reference code, ensures that every NL-code pair is precise, internally consistent, and reproducible.

Listing 1 presents a short extract from a real-world shellcode to exemplify the different types of contextual information we encompassed. This assembly snippet is part of a *shellcode decoder* (Table 3, id 13), a small program designed to decrypt or unpack an encoded shellcode before executing it, typically used to evade detection or bypass security mechanisms.

```
1  ; Move the word at the address [EDI + 1 + ECX] into AX
2  mov     ax, word [edi + 1 + ecx]
3  ; Perform XOR between AX and BX
4  xor     ax, bx
5  ; Move the result into the word at [EDI]
6  mov     word [edi], ax
7  ; Increment ECX to point to the next byte
8  inc     ecx
9  ; Load the address [EDI + 2] into EDI
10 lea     edi, [edi + 2]
```

Listing 1 Assembly code extract.

The first three lines form a logically interconnected sequence where each operation depends on the outcome of the previous step. For instance, the instruction `mov word [edi], ax` (line 5) stores a value in memory, but its meaning remains ambiguous unless the origin of `ax` is known. Without additional context, a generic NL description like "Move the result into the word at [EDI]" provides insufficient information for a model to determine the correct computation path. This scenario represents the *no context* setting, where missing details make code generation challenging. However, when preceding oper-

ations are considered, such as loading a value into `ax` (line 1) and performing an XOR operation on it (line 3), the description becomes more meaningful, forming a *related context* that allows the model to interpret and generate the code correctly.

Conversely, some instructions frequently co-occur in offensive security exploits without being directly dependent on one another. For example, the increment operation `inc ecx` (line 8) and the address computation `lea edi, [edi + 2]` (line 10) serve distinct purposes but often appear together in exploit development. This exemplifies the *unrelated context* scenario, which tests a model's ability to filter out coincidental patterns and focus only on meaningful dependencies for accurate code generation.

Taking into account these considerations about directly and indirectly interconnected operations, we generated NL descriptions for the code snippets using the following notation:

■ **No Context**: Instructions without added contextual information, representing the baseline for model performance assessment, in which NL descriptions may have implicit or incomplete information. This category includes 963 lines (∼ 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 Related Context**: Incorporating the immediate preceding contextually related instruction to provide context, accounting for 360 lines (∼ 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 Related Context**: Further extending context by including the two contextually related instructions preceding the current one, comprising 238 lines (∼ 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 Unrelated Context**: Featuring a single previous instruction that does not logically link to the current task, this scenario covers 303 lines (∼ 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 Unrelated Context**: Similar to 2to1 Unrelated Context but with two preceding instructions, also making up 303 lines (∼ 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 were carefully aligned with the relational structure of the code snippets derived from the shellcodes. In the **No Context** scenario, designed to evaluate the models' ability to generate code based solely on standalone instructions, we provided descriptions without any contextual dependencies. To better reflect real-world variability, we intentionally modified the highly detailed intents from the Shellcode_IA32 dataset to be less precise, simulating the inaccuracies commonly found in developer-provided inputs.

For scenarios requiring the learning of relational dependencies (**2to1 Related Context** and **3to1 Related Context**), we selected sequences of instructions that depended on one another to complete a coherent task. These descriptions could only be fully understood with knowledge of the preceding one or two instructions, emphasizing the role of contextual relationships. In contrast, for evaluating the models' ability to handle unrelated information (**2to1 Unrelated Context** and **3to1 Unrelated Context**), we deliberately included instructions with no direct connection to the current task. 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 is available on GitHub for reproducibility and future research [1].

3.4 Code Generation Task

To rigorously evaluate how additional contextual information, related and unrelated, affects the translation ability of deep-learning models, we follow established best practices in automatic code generation. This process is designed to analyze how different models interpret different types of contextual information and assess their robustness under varying conditions. Specifically, we aim to understand whether models are robust when faced with missing information and whether the inclusion of additional useful information enhances their performance. For this purpose, we evaluate nine state-of-the-art models, representative of three key categories: fine-tuned encoder-decoder models, fine-tuned decoder-only models, and instruction-tuned large language models (LLMs). This selection allows us to compare the strengths and limitations of different architectures and training paradigms, offering a comprehensive understanding of how task-specific fine-tuned models and general-purpose LLMs handle contextual cues in offensive security code generation.

For fine-tuned models, the process begins by constructing a corpus to train the DL models. The training data undergoes *pre-processing* operations, starting with *stopwords filtering*, where irrelevant words (e.g., *the*, *each*, *onto*) are removed to focus on content relevant for translation. Next, a *tokenizer* breaks the intents into *tokens*. To improve the performance of the machine translation [32,48,34], we *standardize* the intents (i.e., we reduce the randomness of

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

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, tokens are converted into real-valued vectors using *word embeddings*, which are used to train the model.

After training, the fine-tuned models generate code snippets for unseen NL descriptions. During *post-processing*, to improve the quality and readability of the code, the predicted code snippets undergo a *de-standardization* process using the dictionary of standardizable tokens to replace all the "*var#*" with the corresponding values, names, and parameters.

For k-shot prompted models, the workflow differs. Instead of training, we construct tailored prompts that incorporate NL descriptions along with any related or unrelated contextual information. These prompts are designed to effectively guide the models, leveraging their pre-trained capabilities. In this setting, post-processing involves cleaning up the models' outputs to remove unnecessary wording or responses generated by the LLMs, ensuring the final code snippet is concise and accurate.

This dual approach allows us to comprehensively evaluate how well different types of models (encoder-decoder, decoder-only, and instruction-tuned LLMs) handle varying levels of information. By comparing their performance, we aim to identify trends in robustness, particularly in how models cope with missing information and whether additional context improves their ability to generate accurate, syntactically correct, and semantically meaningful code.

3.5 Evaluation Procedures

To evaluate the performance of the models, we followed best practices in the field by adopting *output similarity metrics*. These metrics, which compare the generated code with a ground-truth, reference code, are widely used to assess the performance of AI assistants in many code generation tasks [36], including the generation of assembly code for security contexts [82, 83, 64, 35, 34]. Popular similarity metrics, such as METEOR [29] and ROUGE [39], are drawn from the natural language processing domain and adopted in software engineering as they are easily computed and enable automated evaluation. State-of-the-art also proposed solutions tailored to source code, which take into account the differences between natural language and programming languages, including program structure and coding conventions. Prevalent examples are CrystalBLEU [15] and CodeBLEU [62].

While output similarity metrics, including metrics specifically designed to assess code, are the most practical and fast solution to measure the model performance, they are not well-suited to address the strict and highly technical nature of programming code, especially in the offensive security domain. Indeed, they cannot verify whether the generated code is syntactically correct (hence, executable) nor properly assess whether two pieces of code are dif-

ferent but semantically equivalent, i.e., they provide the same output and/or effects although they use different operations (e.g., `jz label` and `je label` are different assembly instructions performing the same conditional jump).

For example, while alternative implementations of the same intent (e.g., transferring EAX contents to EDX using mov versus push and pop) are semantically equivalent, these metrics yield low similarity scores due to their reliance on character or token overlap. Conversely, a syntactically similar but semantically incorrect prediction (e.g., using mov BL, 5 instead of mov DL, 5) can achieve higher scores despite failing to meet the intended functionality. This underscores the need for evaluation metrics that better account for syntactic and semantic correctness.

A common alternative in code generation research is the *pass@k* metric, which measures whether at least one of k generated candidates passes predefined test cases [8]. However, we note that pass@k is poorly suited to our domain. Defining robust, reliable test cases for shellcode is inherently nontrivial, due to tight dependencies on system architecture, register state, and instruction-level side effects such as syscall behavior, flag mutations, and control flow decisions. In such settings, correctness cannot be assessed solely based on output (e.g., whether a shell is spawned), but instead requires a deeper notion of instruction-level and semantic equivalence.

For the aforementioned reasons, human evaluation is considered the gold standard for assessing the correctness of the code generated by AI models [17]. Manual inspection ensures a comprehensive evaluation by leveraging the domain-specific expertise of security analysts to analyze both the syntax and semantics of the generated outputs. However, this approach is often impractical and error-prone due to its significant demands on time and expertise and becomes increasingly unfeasible as the volume of data grows.

To overcome these limitations, we rely on *ACCA* (*Assembly Code Correctness Assessment*) [10], an automated framework designed to comprehensively evaluate AI-generated code for security contexts by assessing both its syntactic and semantic correctness [26,37]. First, *ACCA* verifies whether the generated code correctly adheres to the syntax rules of the target language (i.e., assembly) using the NASM assembler. Next, it leverages *symbolic execution*, i.e., a state-of-the-art solution for program analysis based on abstract execution [81, 61]. It consists of simulating the execution of a program providing symbolic values to evaluate its behavior, enabling an assessment of whether the generated code behaves as a reference implementation, despite syntactic differences between the reference and the generated code.

More precisely, *ACCA* automatically assembles and symbolically executes both the reference code (i.e., the ground truth) and the code generated by the models. Through symbolic execution, the method simulates the execution of both programs and determines whether, starting from the same state, they terminate producing equivalent results and effects. To do so, *ACCA* compares final program states across registers, flags, stack memory, and path constraints for all execution paths. If these match across corresponding execution branches, the code is considered semantically equivalent; otherwise, it is not.

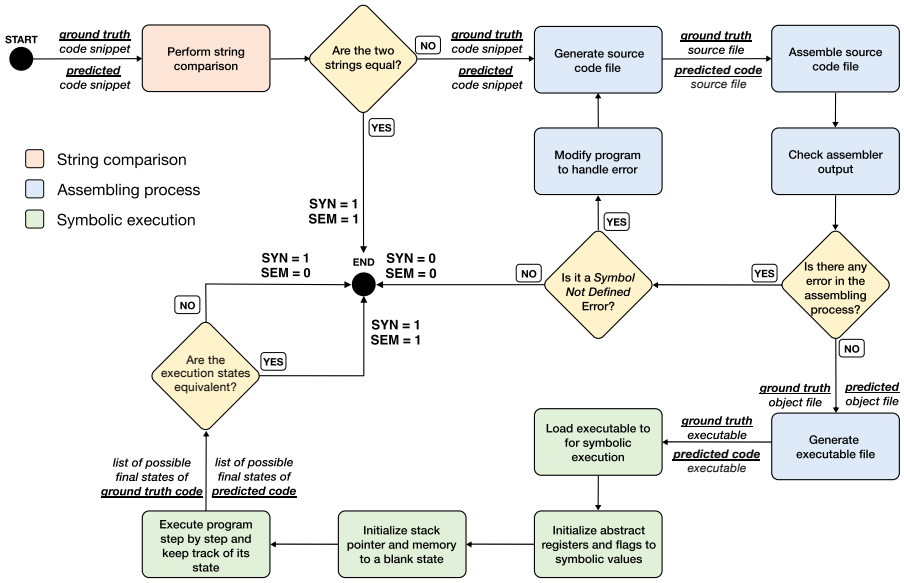

Fig. 2 Detailed flowchart of ACCA [10].

Fig. 2 presents a detailed flowchart illustrating how the method works. *ACCA*'s final output for each ⟨ *prediction*, *ground-truth* ⟩ pair is a syntactic correctness score (i.e., *SYN*) and a semantic correctness score (i.e., *SEM*), as detailed in § 4.2.

This automated solution allows us to provide a thorough analysis of the models' performance in the code generation task without requiring any manual intervention, yet retaining a very strong correlation with the human evaluation (Pearson's correlation coefficient $r = 0.84$ on average) [10]. Furthermore, symbolic execution's extensive adoption in diverse fields, including AI-generated code verification [75] and smart-contracts analysis [40], reinforces its credibility as a robust methodology. By automating the assessment process, *ACCA* eliminates human subjectivity and provides a scalable, reproducible solution for evaluating the performance of DL models in generating offensive code.

4 Experimental Setup

Table 5 summarizes the experimental setup across different analyses designed to evaluate the performance of DL models in generating offensive security code from NL descriptions under varying conditions. Each analysis aims at exploring a specific aspect of the model's capabilities, such as handling missing information, leveraging contextual information, and distinguishing useful information from unrelated context.

The dataset was structured to support two main objectives within this framework. First, for each experimental setting, the data was systematically

Table 5 Dataset statistics and analysis objectives for contextual and unnecessary information experiments.

Analysis	**Objective**	**Set**	**No Ctx**	**2to1 Rel. Ctx**	**3to1 Rel. Ctx**	**2to1 Unrel. Ctx**	**3to1 Unrel. Ctx**	**Total**
Missing Information (No Context)	Establishing a baseline for model performance without context to assess the ability to handle standalone instructions.	Train	770 (80%)	-	-	-	-	770 (80%)
		Dev	96 (10%)	-	-	-	-	96 (10%)
		Test	96 (10%)	-	-	-	-	96 (10%)
Related Information (2to1 Related Context)	Simulating realistic coding scenarios where previous information impacts operations.	Train	867 (90%)	180 (50%)	-	-	-	1047 (80%)
		Dev	48 (5%)	90 (25%)	-	-	-	138 (10%)
		Test	48 (5%)	90 (25%)	-	-	-	138 (10%)
Related Information (3to1 Related Context)	Emulating more complex coding scenarios to assess 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%)
Unrelated Information (2to1 Unrel. Context)	Determining the ability to 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%)
Unrelated Information (3to1 Unrel. Context)	Evaluating the capacity to ignore multiple irrelevant 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%)

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). These splits were employed to fine-tune six DL models for the target task of generating offensive security code. The *No Context* set serves as the foundation for comparing the impact of additional related or unrelated context, while the proportional inclusion of contextual scenarios across the train, validation, and test sets allows for a comprehensive assessment of how well DL models learn from varying degrees of contextual information.

The distribution of samples across the dataset categories naturally influenced the absolute size of the splits. Standalone instructions (No Context) are inherently more frequent than sequences of two dependent intents (2to1), and three dependent intents (3to1) are even rarer, although they are particu-

larly challenging for models to comprehend. This reflects how exploit code is actually written: shellcodes tend to be compact, with short sequences of dependent instructions rather than long contextual chains. Artificially equalizing the training data (e.g., by oversampling, generating synthetic 3to1 examples or even cutting out useful data) would have introduced biases and potentially reduced the validity of the study. Instead, we chose to preserve the dataset's realistic proportions, ensuring that the models were evaluated under conditions that mirror the scarcity and distribution of contexts encountered in practice. This decision is crucial, especially in the offensive security domain, where the goal is to test models in realistic scenarios rather than artificially balanced datasets.

To guarantee fairness across models, we adopted a uniform 80%/10%/10% split ratio for training, validation, and testing, consistent with established practices in empirical software engineering and AI for code [28,44,33]. This ensures that every model was trained and evaluated under exactly the same conditions. Furthermore, the test set was deliberately standardized at ∼100 samples per category, regardless of training set size, to provide a robust and controlled basis for comparison. Thus, while absolute training sizes differ between categories (an unavoidable outcome of the domain's intrinsic characteristics), the relative fairness is fully preserved: all models are exposed to the same distributions and evaluated on the same balanced test sets.

We also note that the dataset size used for the experiments, approximately one thousand samples, aligns with other state-of-the-art corpora used for fine-tuning models in related domains [85]. In fact, in state-of-the-art code generation, models are not trained from scratch, but existing pre-trained models (that were already trained on millions of publicly available lines of code) are fine-tuned in a supervised way, to achieve transfer learning for the specific task (in our case, generating offensive code).

Second, the same test sets were refined to evaluate the capabilities of three instruction-tuned LLMs by constructing prompts based on the NL descriptions. We adopted a *persona-based prompting* technique in two settings: *zero-shot* and *k-shot*. The zero-shot setting evaluates the models' ability to generate accurate outputs without relying on additional demonstrations, ensuring that performance reflects only their interpretation of the contextual information (or lack thereof) provided. In contrast, the k-shot setting introduces k demonstrations as in-context examples to guide the model in the task comprehension and generation process. This dual setup allows us to directly compare LLMs' baseline reasoning with their capacity to leverage exemplar demonstrations under controlled conditions.

Listing 2 shows an example of a zero-shot prompt containing an NL description with *2to1 Related Context*. In particular, we specify the role to explicitly instruct the model to act as an expert in IA-32 assembly code generation and include a detailed task description. To help the model distinguish between provided contextual information (if present) and the current NL description, the `_BREAK` token was introduced to mark the boundary between previous context and the current task, ensuring focus on the latest NL description. The example

illustrates the structured format of the prompt: *i)* a "template" explaining the task, equal for all prompts; *ii)* the specific prompt "instruction", indicating the NL description of the code to be generated, different for each prompt; *iii)* the "response" section, prompting the model to produce the corresponding assembly code. The dataset and code are available on GitHub [2].

```
You are an expert in IA-32 assembly code generation. You have to
generate an IA-32 assembly code snippet starting from its natural
language description. I will provide a code description, generate
just the corresponding assembly code. The _BREAK tokens separate
previous code descriptions from the current code description. Just
 translate the last description.

### Instruction:
Description: Move eax to edx _BREAK Right shift the register by
byte 16 _BREAK Add the result to eax

### Response:
Assembly Code:
```

Listing 2 Example of zero-shot prompt for generating IA-32 assembly code with LLMs.

In the few-shot setting, we adopted k=*4* demonstrations, following recent findings that small- to medium-sized LLMs achieve the best balance of performance and stability with around four in-context examples. Prior work on code intelligence tasks shows that increasing k beyond four often leads to diminishing or unstable returns due to prompt length and input noise [21]. We used a task-level demonstration approach, where the same four examples were included for every test input within each dataset. To ensure fairness and domain alignment, the four demonstrations were randomly sampled from the corresponding training split (e.g., demonstrations drawn from the 2to1 related context training set when testing on 2to1 related context tasks) in order to balance reproducibility with coverage of realistic task diversity, while avoiding bias toward a small subset of curated examples.

Listing 3 shows an example of a 4-shot prompt, where demonstrations precede the new task to be solved.

```
You are an expert in IA-32 assembly code generation. You have to
generate an IA-32 assembly code snippet starting from its natural
language description. Below are several examples of natural
language descriptions and their corresponding IA-32 assembly code
implementations. Use these examples to understand how the
descriptions are translated into assembly code. The _BREAK tokens
separate previous code descriptions from the current code
description. Just translate the last description.

### Description and Assembly Code Examples:

Description: increment ecx _BREAK repeat
Assembly Code: inc ecx
Description: clear ecx _BREAK subtract the result from ebx
Assembly Code: sub ebx,ecx
```

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

```
Description: move 876189623 in edi _BREAK doubles its value
Assembly Code: add edi,edi
Description: push the pointer to stack for later execution _BREAK
move it to edi
Assembly Code: mov edi, esi

### New Task:
New Description: Right shift the register by byte 16 _BREAK
Add the result to eax

### Response:
Assembly Code:
```

Listing 3 Example of 4-shot prompt with task-level demonstrations for IA-32 assembly code generation

Using the same test set for both the fine-tuned models and for constructing prompts for the LLMs ensured a consistent and reliable framework for comparative analysis under uniform test conditions.

4.1 Deep-Learning Models

To perform the code generation task, we consider a comprehensive set of state-of-the-art models covering different types of model architectures and training procedures, which are described in the following.

■ **CodeBERT** [19] is a multi-layer bidirectional Transformer architecture [71] 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 [41], with 12 attention heads, hidden layer dimension of 768, 12 encoder layers, and 514 for the size of position embeddings.

■ **CodeT5+** [76] 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 [60]. 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.

■ **PLBart** [2] 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.

■ **CodeGen** [52] 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 [52], a large-scale dataset of multiple programming languages from GitHub repositories, which consists of 119.2B tokens and includes C, C++, Go, Java, JavaScript, and Python.

■ **CodeGPT** [42] 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 adopt 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 follow previous work for the implementation [42].

■ **CodeParrot** [24] 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 adopt the *codeparrot-small* version with $110M$ parameters.

■ **DeepSeek-Coder-Instruct** [22] is part of a series of code language models, each trained from scratch on 2T tokens, with a composition of 87% code and 13% natural language. The base model is pre-trained on project-level code corpus by employing a window size of 16K and an extra fill-in-the-blank task. We utilize a 4-bit quantized, instruction-tuned model with 6.7 billion parameters, which was further fine-tuned on 2B tokens of instruction data.

■ **Qwen2.5-Coder-Instruct** [25] is the latest series of Qwen large language models specifically designed for code-related tasks. These models have been pre-trained on an extensive dataset exceeding 5.5 trillion tokens, achieving state-of-the-art performance in code generation, completion, reasoning, and repair tasks. We employ a 4-bit quantized model with 7 billion parameters, which was further trained for instruction-tuning through a multi-stage fine-tuning process.

■ **StableCode-Instruct** [57] is a 2.7B parameter decoder-only language model pre-trained on 1.3 trillion tokens of diverse textual and code datasets spanning over 18 programming languages. The instruct model was trained on a mix of publicly available and synthetic datasets using Direct Preference Optimization. We adopt the 4-bit quantized version of this model.

The first three (i.e., CodeBERT, CodeT5+, and PLBart) are fine-tuned models 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 second three (i.e., CodeGen, CodeGPT, and CodeParrot) are fine-tuned *decoder-only* models, which read the input sequence and predict subsequent words one at a time, well-suited for generation tasks. The last three (i.e., DeepSeek-Coder-Instruct, Qwen2.5-Coder-Instruct, and StableCode-Instruct) are instruction-tuned LLMs with 6.7B, 7B, and 3B trainable parameters, respectively, which feature advanced capabilities for reasoning and executing complex, user-specific prompts.

When fine-tuning DL models, during data pre-processing, we tokenize the NL intents using the *nltk word tokenizer* [5] and code snippets using the Python *tokenize* package [59]. We use *spaCy*, an open-source, NL processing library written in Python and Cython [67], to implement the named entity tagger for the standardization of the NL intents. To ensure accurate comparison with instruction-tuned LLMs, we clean up their predicted outputs by extracting only the generated code, removing any unnecessary responses such as explanations or formatting artifacts. This process involves applying regular expressions to isolate code segments from the broader output. Additionally, we manually inspect the cleaned samples to verify correctness and ensure the extracted code is consistent with the intended task. This meticulous approach minimizes noise and standardizes outputs for reliable evaluation.

The experimental campaign was conducted on a Debian Linux system with 20 vCPUs, 32 GBs RAM, and one Nvidia GeForce GTX 1660 GPU.

4.2 Evaluation Metrics

To automatically assess the performance of the models, we compared the generated code with the ground-truth code using five widely recognized output similarity metrics. Four of these metrics are commonly used in neural machine translation tasks across various domains, including natural language and source code generation. The fifth metric, CrystalBLEU, is specifically tailored for assessing the quality of code, providing a more domain-specific evaluation. In particular, we adopted the following:

■ **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` [58].

■ **METEOR** [29]. 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 [16].

■ **ROUGE-L** [39]. 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). We computed the ROUGE-L metric using the Python package `rouge` [63].

■ **CrystalBLEU** [15]. It is a variant of the BLEU score [56] designed to better assess similarity among code snippets. While the BLEU score measures similarity based on the overlap of n-grams between the predicted and reference outputs, CrystalBLEU refines this approach by excluding the most frequent (i.e., "trivially shared") n-grams in a given programming language. Indeed, in the case of programming languages, these frequent n-grams are mostly due to the syntactic constructs and coding conventions of the language, thus inflating and flattening the computed similarity scores. CrystalBLEU scores range between 0 and 1, with higher scores corresponding to greater overlaps between n-grams. We computed the CrystalBLEU metric using the Python package `crystalbleu` [12].

As mentioned in § 3.5, these similarity metrics do not take into account whether the generated code is syntactically correct and semantically equivalent to the reference code. Therefore, we introduce two additional metrics to provide a comprehensive evaluation of the generated exploit code and leverage *ACCA* to compute them. They are:

■ **Syntactic Accuracy (SYN)**. It indicates whether each code snippet produced by the model is compilable according to the syntax rules of the target language. SYN value is either 1, when the snippet's syntax is correct, or 0 otherwise. To compute syntactic correctness, we used the implementation provided by *ACCA* which relies on the *Netwide Assembler* (NASM) assembler [49]. Syntactic accuracy is computed as the number of syntactically correct predictions over the total number of predictions (i.e., the total number of code snippets generated by the model).

■ **Semantic Accuracy (SEM)**. It indicates whether the code snippet is the exact translation of the NL intent into the target programming language (i.e., assembly). SEM value is either 1, when the snippet is semantically equivalent to the reference code, or 0 otherwise. To assess this, we rely on *ACCA* to evaluate the semantic equivalence between the generated code and the ground truth code by symbolically executing both. Semantic accuracy is computed as the number of semantically correct predictions over the total number of predictions (i.e., the total number of code snippets generated by the model).

5 Experimental Results

This section presents the experimental results obtained from evaluating nine deep-learning models on the task of generating assembly shellcodes from NL descriptions. The models include six fine-tuned models, i.e., three encoder-decoder architectures and three decoder-only architectures, and three instruction-tuned LLMs. For the LLMs, we considered both a zero-shot setting and an additional few-shot setting (i.e., 4-shot prompting) in order to assess whether providing exemplar demonstrations improves their ability to handle security-relevant code generation tasks. The experiments were designed to evaluate the

models' capabilities across three dimensions: handling scenarios with missing information, effectively leveraging extended context, and distinguishing between related and unrelated contextual information. First, we evaluated the models' performance using a comprehensive set of output similarity metrics, then we performed a thorough analysis of the syntactic and semantic correctness of the generated exploit code.

5.1 Missing Information

To address $\mathbf{RQ}_1$, we evaluated the performance of DL models in generating offensive security code from natural language descriptions without providing any additional contextual information. This scenario mirrors real-world situations where developers might provide incomplete or ambiguous descriptions, often due to oversights or assumptions about implicit knowledge, and sets the baseline for our experimental evaluation. Assessing how different types of models handle such missing information is essential for evaluating their practical utility in automated code generation tasks, especially in such high-stakes domains as offensive security development. Table 6 shows the results, highlighting in **bold** the best score for each metric. A comparative analysis reveals significant differences in performance among fine-tuned encoder-decoder models (*E-D*), fine-tuned decoder-only models (*D-O*), and instruction-tuned LLMs, both in a zero-shot (*0-shot*) and a few-shot (*4-shot*) setting, highlighting the strengths and weaknesses of each approach.

Fine-tuned encoder-decoder models such as CodeBERT and CodeT5+ consistently outperform the others across all metrics, with CodeT5+ achieving the highest scores among all models (METEOR: 74.87, ROUGE-L: 73.44, CrystalBLEU: 60.35), demonstrating strong generalization capabilities even in the absence of additional information. CodeBERT follows closely, performing well on EM (45.99) and ED (77.30). In contrast, PLBart underperforms, showing the lowest scores across all models. This suggests that PLBart struggles with incomplete descriptions, potentially due to its pre-training limitations or less effective fine-tuning.

Fine-tuned decoder-only models exhibit more variability and generally perform worse when compared to E-D models. Among them, CodeGen achieves moderate results (EM: 30.92, METEOR: 52.72, CrystalBLEU: 38.54), indicating its ability to handle missing information to a certain extent. However, CodeGPT and CodeParrot struggle significantly, with CodeGPT showing the lowest METEOR (38.60) and ROUGE-L (34.17) scores in this category. These results suggest that D-O models, which lack the bidirectional context modeling inherent in encoder-decoder architectures, face challenges in capturing complex relationships between NL inputs and generated code, particularly in scenarios with incomplete descriptions. Between E-D and D-O, model size does not seem to significantly affect performance, with CodeBERT having fewer trainable parameters (125M) than PLBart (140M), CodeParrot less than CodeGPT (110M *vs.* 124M), and CodeGen being the biggest model (350M).

Table 6 Model performance on the generation task with missing information. Best scores are in **bold**.

	Model	EM	ED	METEOR	ROUGE-L	Crystal BLEU
E-D	*CodeBERT*	45.99	77.30	67.41	65.75	54.30
	CodeT5+	**59.35**	**81.68**	**74.87**	**73.44**	**60.35**
	PLBart	7.44	32.59	21.41	27.04	13.19
D-O	*CodeGen*	30.92	60.97	52.72	48.76	38.54
	CodeGPT	16.60	46.95	38.60	34.17	28.00
	CodeParrot	21.76	53.15	43.78	40.71	32.20
0-shot	*DeepSeek-Coder*	39.69	67.30	61.45	62.03	48.81
	Qwen2.5-Coder	32.06	68.29	58.46	55.44	47.36
	StableCode	10.88	40.43	36.78	29.85	24.26
4-shot	*DeepSeek-Coder*	34.16	64.96	61.19	61.85	47.14
	Qwen2.5-Coder	32.06	65.68	50.28	56.02	44.28
	StableCode	20.23	52.52	42.76	40.20	34.58
	All Models	29.26	59.32	50.81	49.61	39.42

Instruction-tuned LLMs present an even higher variability. In the 0-shot setting, DeepSeek-Coder demonstrates competitive performance, achieving the third-best performance after CodeT5+ and CodeBERT (EM: 39.69, METEOR: 61.45, ROUGE-L: 62.03, CrystalBLEU: 48.81). This indicates the potential of some instruction-tuned LLMs to correctly handle missing information in the inputs even without fine-tuning. Qwen2.5-Coder follows closely, with slightly lower but consistent scores across metrics, showcasing moderate robustness. In contrast, StableCode underperforms significantly (EM: 10.88, CrystalBLEU: 24.26), highlighting the limitations of zero-shot prompting for complex code generation tasks.

In the few-shot setting, results diverge. DeepSeek-Coder and Qwen2.5-Coder see a slight decline in performance across most metrics (e.g., DeepSeek-Coder CrystalBLEU: 48.81 $\rightarrow$ 47.14), suggesting that providing multiple examples may introduce noise or confusion when the examples are not directly aligned with the current task. By contrast, StableCode shows notable improvements with 4-shot prompting (EM: 10.88 $\rightarrow$ 20.23; CrystalBLEU: 24.26 $\rightarrow$ 34.58), although it remains behind both the stronger LLMs and most fine-tuned models.

These findings suggest that while zero-shot prompting can already yield competitive performance for stronger LLMs, few-shot prompting does not guarantee improvements and may even hinder results if the model is sensitive to prompt length or example relevance. On the other hand, smaller LLMs may benefit more from structured few-shot examples, as seen with StableCode. Moreover, in this setting, the number of parameters strongly impacts performance, with StableCode being half the size of the other two LLMs (3B against $\sim$7B).

The average scores across all models (EM: 29.26, ED: 59.32, METEOR: 50.81, ROUGE-L: 49.61, CrystalBLEU: 39.42) reflect a collective moderate proficiency in dealing with NL descriptions that are missing some information.

To contextualize the evaluation, prior research with more detailed NL descriptions [83] reported EM scores of 48.52 for CodeGPT and 51.80 for CodeBERT, setting a benchmark for assembly code generation under richer input conditions. In our experiments, without contextual information, CodeGPT and CodeBERT achieved EM scores of 16.60 and 45.99, respectively, highlighting a significant drop for CodeGPT and a smaller decline for CodeBERT. This discrepancy emphasizes the impact of missing information on model performance, with CodeBERT showing greater robustness under these conditions.

RQ$_1$: *How do DL models perform in generating offensive security code from NL descriptions when faced with missing information?*

DL models display varying levels of effectiveness in generating offensive security code from less detailed NL descriptions. Fine-tuned encoder-decoder architectures demonstrate the best ability to manage missing information and generate correct code, with CodeT5+ showing the best performance among all, followed by CodeBERT. Their explicit training on task-specific datasets appears to provide a significant advantage, despite their significantly smaller size when compared to other LLMs (e.g., Qwen2.5-Coder is more than 30 times bigger than CodeT5+). On the other hand, zero-shot prompted LLMs show surprising potential without the need for time-consuming fine-tuning, particularly DeepSeek-Coder, but few-shot prompting does not consistently improve their performance, with only smaller models like StableCode benefiting from additional examples. Finally, fine-tuned decoder-only models generally lag behind, suggesting that their architecture may be less suited for more complex tasks, especially in scenarios with incomplete information.

5.2 Related Contextual Information

Next, to answer **RQ$_2$**, we focused on analyzing to what extent the presence of additional related contextual information in the NL descriptions impacts the performance of DL models in generating assembly shellcodes. 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 and the 3to1 Context datasets, respectively (see Table 5). This investigation is particularly important to determine how much contextual information can effectively aid in enhancing model performance, especially in comparison to baseline performance with missing information, and how different models can interpret and benefit from it.

Table 7 Comparison of models' performance with 2to1 and 3to1 related context against the baseline without contextual information (None). Best scores are in **bold**.

	Model	Context Type	EM	ED	METEOR	ROUGE-L	Crystal BLEU
E-D	*CodeBERT*	None	45.99	77.30	67.41	65.75	54.30
		2to1	61.07	80.77	74.30	72.29	59.08
		3to1	47.90	75.34	67.19	65.84	52.93
	CodeT5+	None	59.35	81.68	74.87	73.44	60.35
		2to1	62.40	**82.72**	**77.58**	**75.50**	**61.68**
		3to1	**62.79**	81.70	77.06	75.14	61.22
	PLBart	None	7.44	32.59	21.41	27.04	13.19
		2to1	13.55	35.99	29.36	33.21	22.23
		3to1	10.31	24.25	21.63	29.50	17.13
D-O	*CodeGen*	None	30.92	60.97	52.72	48.76	38.54
		2to1	43.70	66.15	60.87	55.91	44.08
		3to1	36.26	62.07	53.91	51.43	38.37
	CodeGPT	None	16.60	46.95	38.60	34.17	28.00
		2to1	25.38	53.71	44.59	41.14	32.79
		3to1	22.14	50.83	41.20	37.96	29.93
	CodeParrot	None	21.76	53.15	43.78	40.71	32.20
		2to1	27.67	57.98	48.56	45.24	35.51
		3to1	20.99	51.52	42.20	37.16	29.35
0-shot	*DeepSeek-Coder*	None	39.69	67.30	61.45	62.03	48.81
		2to1	41.03	64.53	65.05	58.52	47.03
		3to1	40.46	64.56	64.60	60.17	47.62
	Qwen2.5-Coder	None	32.06	68.29	58.46	55.44	47.36
		2to1	30.73	64.23	61.35	51.55	44.14
		3to1	34.92	66.17	63.71	56.51	47.06
	StableCode	None	10.88	40.43	36.78	29.85	24.26
		2to1	12.40	38.07	38.48	29.59	23.30
		3to1	11.45	38.52	39.05	31.85	23.89
4-shot	*DeepSeek-Coder*	None	34.16	64.96	61.19	61.85	47.14
		2to1	28.63	58.96	59.41	52.91	43.80
		3to1	39.89	63.66	64.19	55.61	48.27
	Qwen2.5-Coder	None	32.06	65.68	50.28	56.02	44.28
		2to1	23.66	60.94	51.83	47.85	43.35
		3to1	41.03	69.50	57.79	56.79	49.18
	StableCode	None	20.23	52.52	42.76	40.20	34.58
		2to1	14.69	48.21	41.59	37.83	31.75
		3to1	22.90	52.07	45.39	40.33	35.33
	All Models	None	29.26	59.32	50.81	49.61	39.42
		2to1	32.08	59.36	54.41	50.13	40.73
		3to1	32.59	58.35	53.16	49.86	40.02

Table 7 provides a comparative analysis of the nine DL models across different context settings. First fundamental observation is that all fine-tuned models benefit from related context, where the inclusion of additional contextual information, whether 2to1 or 3to1, consistently improves performance compared to the baseline with no context.

Fine-tuned encoder-decoder models (E-D) demonstrate the most substantial gains, while decoder-only models (D-O) and instruction-tuned LLMs (0- and 4-shot) show more moderate or unstable improvements. Among E-D models, scores consistently improve in the 2to1 setting and are better or stable in the 3to1 setting, with CodeT5+ demonstrating the most important gains across all models (+3.05 in EM). Similarly, CodeBERT and PLBart benefit significantly from contextual information, achieving their highest EM scores (61.07 and 13.55, respectively).

Decoder-only models also show consistent improvements from using contextual information. CodeGen, which remains the best-performing model in the group, improves its EM from 30.92 (None) to 43.70 (2to1) and 36.26 (3to1), achieving better performance than the baseline in both settings. CodeGPT and CodeParrot follow a similar trend, with CodeGPT's EM rising from 16.60 (None) to 25.38 (2to1) and 22.14 (3to1). Although these models exhibit diminishing performance with excessive context, their code generation capabilities remain better than the baseline, underscoring the consistent benefit of enriching the NL descriptions with contextual details. The only model that deviates from the trend in the 3to1 setting is CodeParrot, which is not capable of handling longer sequences.

Finally, instruction-tuned LLMs reveal a more mixed pattern and, while flexible and robust to varying levels of contextual information, consistently underperform compared to fine-tuned models in absolute terms. Instruction-tuned LLMs reveal a more mixed pattern. In the zero-shot setting, DeepSeek-Coder and Qwen2.5-Coder see small but consistent gains from context, though they remain behind fine-tuned E-D models in absolute performance. StableCode shows only marginal improvement. By contrast, in the few-shot (4-shot) setting, performance becomes more variable: DeepSeek-Coder worsens with 2to1 context but partially recovers with 3to1, Qwen2.5-Coder improves substantially in 3to1 (EM: 41.03), and StableCode shows modest but steady gains across both contexts.

The primary reason for this gap lies in the lack of task-specific fine-tuning, as instruction-tuned models rely solely on pre-trained general-purpose capabilities and prompt engineering. This limits their ability to fully align with the specialized requirements of offensive security code generation, particularly in low-level languages like assembly. Moreover, their heavy dependence on the structure and length of the prompt can exacerbate performance issues in complex tasks.

Overall, extending the contextual information provided in NL descriptions enhances model robustness across categories, but the scale of the improvement depends strongly on the model type. Fine-tuned encoder–decoder architectures remain the most effective at leveraging context, decoder-only models

show more modest benefits, and instruction-tuned LLMs, especially under few-shot prompting, exhibit greater variability, reflecting their reliance on prompt quality and their size-dependent performance.

> **RQ$_2$**: *Does related contextual information enhance the robustness of DL models in the generation of offensive security code?*
>
> Extending the contextual information provided in the NL descriptions consistently enhances the robustness of deep learning models in generating offensive security code, though the extent varies by model type. Fine-tuned encoder-decoder architectures, particularly CodeT5+ and CodeBERT, show the most significant improvements across all metrics, with consistent gains in both 2to1 and 3to1 settings. Decoder-only models also benefit, though with diminishing returns in the 3to1 context. Instruction-tuned LLMs show smaller and more variable gains: while zero-shot prompting yields modest improvements, few-shot prompting amplifies variability, with some models benefiting (Qwen2.5-Coder, StableCode) and others deteriorating (DeepSeek-Coder in 2to1). These findings highlight that while context is beneficial overall, fine-tuned models are far more effective at systematically leveraging it, whereas LLMs remain sensitive to prompt design and scaling conditions, emphasizing the importance of task-specific fine-tuning for leveraging contextual information effectively.

5.3 Unrelated Contextual Information

To investigate **RQ$_3$**, we explored the impact of incorporating unrelated contextual information (Unrelated Context 2to1 and 3to1) on the performance of the different types of DL models in generating offensive security code (see Table 5). This setup allows us to assess the models' ability to generate precise code in the presence of information that is not directly related to the current task. In this setting, we incorporate into the prompts preceding NL descriptions that do not directly contribute to the logical flow of the current instruction but may appear together in offensive security exploits. Table 8 shows the results.

Considering fine-tuned models, the overall pattern is clear: encoder–decoder models deteriorate significantly with unrelated context, while decoder-only models improve substantially. For example, CodeBERT's EM drops from 45.99 (None) to 30.15 (2to1), while CodeT5+ falls from 59.35 (None) to 45.42 (2to1). These models appear to be adversely affected by unrelated information, which disrupts their ability to generate accurate code, suggesting their struggle in filtering out unhelpful details effectively. Only PLBart shows a significant gain in 2to1 context (CrystalBLEU 13.19 $\rightarrow$ 23.51), in which even though the NL descriptions include unnecessary details, they ensure that some useful signals are still embedded within the input.

Table 8 Comparison of models' performance with 2to1 and 3to1 additional unrelated context against the baseline without contextual information (None). Best scores are in **bold**.

	Model	Context Type	EM	ED	METEOR	ROUGE-L	Crystal BLEU
E-D	*CodeBERT*	None	45.99	77.30	67.41	65.75	54.30
		2to1	30.15	63.51	52.03	48.71	42.40
		3to1	35.69	65.08	55.68	52.98	44.25
	CodeT5+	None	59.35	81.68	74.87	73.44	60.35
		2to1	45.42	71.78	61.99	60.75	51.70
		3to1	47.33	74.24	63.59	61.75	52.81
	PLBart	None	7.44	32.59	21.41	27.04	13.19
		2to1	16.98	36.37	29.41	32.50	23.51
		3to1	7.25	26.37	17.78	21.28	14.15
D-O	*CodeGen*	None	30.92	60.97	52.72	48.76	38.54
		2to1	**61.83**	**81.74**	**67.92**	**71.11**	**55.58**
		3to1	50.76	73.37	59.78	59.51	47.99
	CodeGPT	None	16.60	46.95	38.60	34.17	28.00
		2to1	46.76	72.79	57.35	59.47	46.15
		3to1	37.02	66.37	49.70	49.87	39.42
	CodeParrot	None	21.76	53.15	43.78	40.71	32.20
		2to1	54.39	77.22	63.25	64.97	49.87
		3to1	34.16	66.03	49.70	47.56	39.08
0-shot	*DeepSeek-Coder*	None	39.69	67.30	61.45	62.03	48.81
		2to1	5.73	36.36	52.09	39.09	23.33
		3to1	4.58	26.50	51.70	34.27	17.15
	Qwen2.5-Coder	None	32.06	68.29	58.46	55.44	47.36
		2to1	5.34	39.45	52.63	40.31	26.05
		3to1	3.63	28.23	51.11	36.99	18.48
	StableCode	None	10.88	40.43	36.78	29.85	24.26
		2to1	1.72	23.63	30.53	21.36	12.08
		3to1	0.38	16.45	25.68	16.72	6.73
4-shot	*DeepSeek-Coder*	None	34.16	64.96	61.19	61.85	47.14
		2to1	8.40	36.79	41.24	29.64	23.58
		3to1	5.15	29.27	47.12	30.56	19.87
	Qwen2.5-Coder	None	32.06	65.68	50.28	56.02	44.28
		2to1	17.37	48.90	49.38	47.33	32.64
		3to1	21.56	46.56	45.13	44.15	30.53
	StableCode	None	20.23	52.52	42.76	40.20	34.58
		2to1	5.53	31.97	34.32	23.49	19.96
		3to1	4.39	25.32	32.24	20.68	15.28
	All Models	None	29.26	59.32	50.81	49.61	39.42
		2to1	24.97	51.71	49.35	44.89	33.90
		3to1	20.99	45.32	45.77	39.69	28.81

By contrast, decoder-only models not only withstand unrelated information but thrive: CodeGen improves from EM 30.92 (None) to 61.83 (2to1), its best overall performance across all settings. CodeGPT and CodeParrot show similar gains, with CodeGPT reaching EM 46.76 in 2to1, and CodeParrot achieving its highest EM score (54.39) in the 2to1 setting and maintaining better-than-baseline performance even in the 3to1 setting (EM: 34.16). These results suggest that decoder-only models are more resilient to unrelated context and can extract useful information from noisy input, using them as a form of regularization. Although additional unrelated information does not directly influence the current instruction, these operations may often occur together in offensive code and contribute to the generation of the correct snippet. Indeed, CodeGen and CodeParrot exhibit the second-best performance, achieving scores comparable (or even higher on some metrics) to CodeT5+ and CodeBERT in the 2to1 Related Context scenario, which is the best overall performance.

Instruction-tuned LLMs, however, show the opposite trend. In the zero-shot setting, DeepSeek-Coder, Qwen2.5-Coder, and StableCode all collapse under unrelated context, with EM values dropping close to zero (e.g., DeepSeek-Coder: 39.69 $\rightarrow$ 5.73 in 2to1). In the few-shot (4-shot) setting, the trend remains: DeepSeek-Coder drops from EM 34.16 (None) to 8.40 (2to1), Qwen2.5-Coder falls to 17.37, and StableCode performs even worse. This indicates that few-shot prompting does not mitigate the effect of providing the models with unrelated information and may even exacerbate it, making these models more brittle when faced with unclear prompts. Taken together, these results highlight important differences across model families. Encoder–decoder models are hindered by information not directly related to the current input, decoder-only models can benefit from it, and LLMs, although with a slight improvement with few-shot prompting, remain highly vulnerable.

Indeed, these models rely exclusively on pre-trained general-purpose capabilities and prompt engineering at inference time, without the benefit of task-specific fine-tuning, therefore, they lack the fine-grained alignment required for domain-specific goals such as shellcode generation. Their application of general knowledge to a highly specialized task often leads them to an over-reliance on surface-level patterns in the input. These results underscore the need for tailored context-handling mechanisms to maximize performance in code generation tasks, particularly for models sensitive to irrelevant information.

RQ$_3$: *Does unrelated information negatively impact the performance of DL models in the generation of offensive security code?*

While fine-tuned encoder-decoder models, such as CodeBERT and CodeT5+, experience significant performance drops in the presence of unrelated context, decoder-only models consistently improve when compared to the baseline, as seen with CodeGen and CodeGPT, which leverage not directly related inputs to enhance their performance, pos-

sibly using them as a form of regularization to better focus and discern useful information. Notably, CodeGen and CodeParrot achieve scores comparable to the best overall performance (i.e., CodeT5+ and CodeBERT in the Related Context setting). Instruction-tuned LLMs, by contrast, perform worse than their baseline, as they rely solely on general-purpose pre-training and lack the task-specific knowledge needed to filter out unnecessary details effectively. These findings highlight the varying robustness of DL models to unrelated context and emphasize the importance of designing tailored mechanisms to manage superfluous and excessive information in code generation tasks.

5.4 Syntactic & Semantic Correctness

Finally, to address $\mathbf{RQ}_4$, we further explored the models' capabilities in generating security exploits by assessing the syntactic and semantic correctness of their outputs. As mentioned in § 3.5, we employed *ACCA* to assemble and symbolically execute the generated exploit code across varying types of contextual information.

Fig. 3 presents the syntactic correctness of the DL models. To more clearly identify distinct performance trends, models are grouped by color: fine-tuned encoder-decoders are represented in different shades of blue, fine-tuned decoder-only shades of purple, zero-shot prompted LLMs (i.e., DeepSeek-Coder, Qwen2.5-Coder, and StableCode) in orange and 4-shot prompted LLMs (i.e., DeepSeek-Coder-4S, Qwen2.5-Coder-4S, and StableCode-4S) in shades of green.

Encoder-decoder models like CodeBERT and CodeT5+ show high SYN scores both when faced with missing information and when aided by additional related context, with CodeT5+ leading consistently. However, performance declines when dealing with unrelated information, especially for PLBart, which is the worst-performing model (∼41% on average). Decoder-only models (i.e., CodeGen, CodeGPT, CodeParrot) perform robustly, with CodeGen achieving the highest SYN scores across all contexts (99%) and showing resilience even in unrelated context settings (>95%). In contrast, zero-shot prompted LLMs perform poorly, with scores between 70–85% which significant decline when the NL descriptions contain unrelated information, especially for StableCode, which is the second worst (∼49% on average). This decline is more severe in the few-shot prompted variants, which suffer substantial drops in performance as context becomes less relevant. For instance, Qwen2.5-Coder-4S and StableCode-4S fall below 30% in the 3to1 unrelated context setting.

Overall, fine-tuned models produce syntactically correct exploit code in over 85% of cases, maintaining robustness even when exposed to additional unrelated information. Zero- and few-shot prompted LLMs consistently underperform, and their susceptibility to prompt contamination suggests limitations in out-of-the-box usage for structured, syntax-critical code generation tasks.

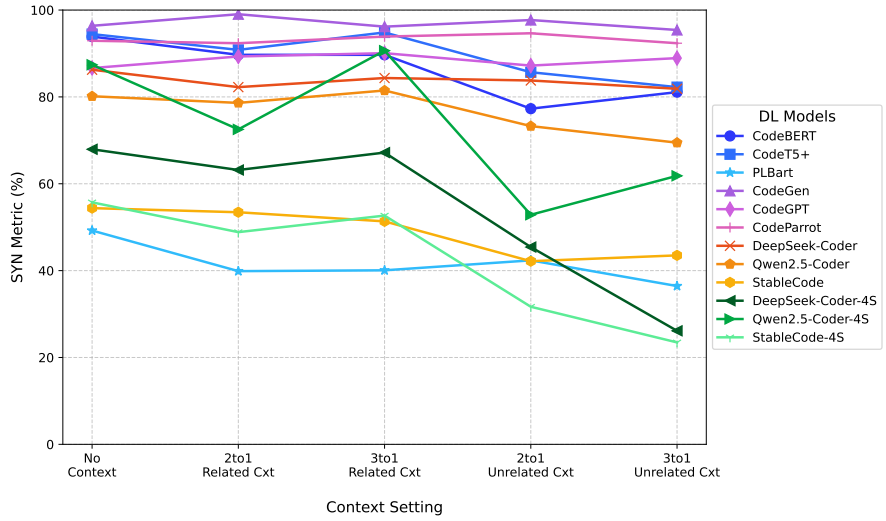

Fig. 3 Syntactic correctness (SYN) of the code generated by all models across different context settings.

Fig. 4 illustrates how well the models are capable of generating code that is semantically equivalent (i.e., implements the same functionality) to the reference code, under varying contextual conditions. Interestingly, similar models present similar behavior, with trends varying significantly across models and context settings, and distinct strengths and weaknesses evident in each.

Fine-tuned encoder-decoder models, represented in blue tones, demonstrate the strongest overall performance, especially in the None and related context settings, but are notably sensitive to unrelated context, as shown in RQ_2 5.2. CodeT5+ achieves the highest overall scores in this group, peaking in the 3to1 context setting (63.05%), improving from the baseline with missing information (59.35%). However, its performance drops significantly when dealing with unrelated context. Similarly, CodeBERT improves from 45.80% (No Context) to 61.86% (2to1 Related Ctx) but declines sharply to 29.75% (2to1 Unrelated Ctx), reflecting its difficulty in filtering unnecessary information. PLBart, which underperforms across all settings, shows slight improvements with useful context (14.12% in 2to1 Related Ctx) but fails with longer sequences with unrelated information, dropping to 7.25%. These results highlight the reliance of encoder-decoder models on clean, relevant input and their struggles with too much information.

Interestingly, although being an encoder-decoder model, PLBart shows a trend similar to decoder-only models. This may be attributed to the fact that, by design, once the bidirectional encoder processes the input sequence, the autoregressive decoder uses causal masking to enforce a unidirectional (left-to-right) generation process like GPT-style models [30, 2].

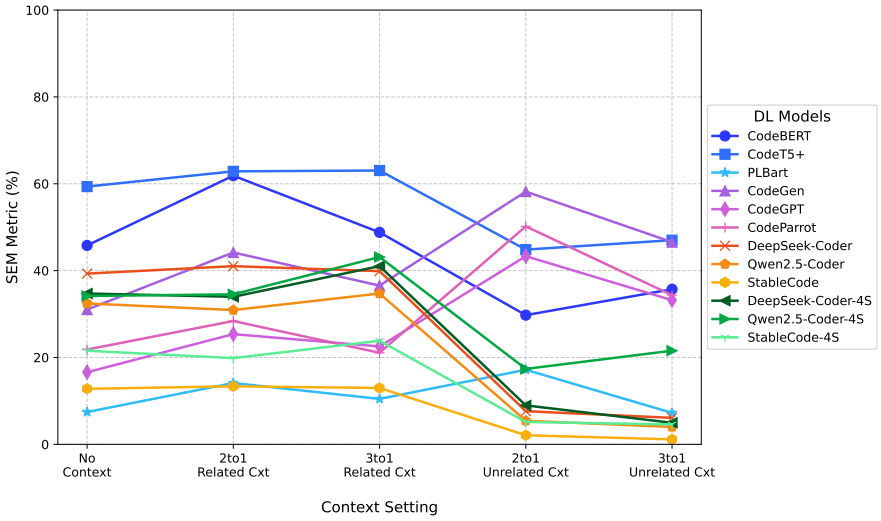

Fig. 4 Semantic correctness (SEM) of the code generated by all models across different context settings.

Decoder-only models, represented in purple, display a more consistent performance trend, particularly excelling in unrelated context settings, as demonstrated in RQ_3 (5.3). CodeGen leads this group, improving from 31.03% (None) to 44.18% (2to1 Related Ctx) and achieving its highest score in the 2to1 Unrelated Ctx setting (58.20%), showcasing its ability to benefit from not directly related context and dealing better without too much extended information. Similarly, CodeGPT improves from 16.60% (None) to 25.38% (2to1 Related Ctx) and reaches 43.32% in 2to1 Unrelated Ctx, indicating its ability to leverage noisy input effectively. CodeParrot also performs well, with scores peaking at 50.19% in 2to1 Unrelated Ctx, outperforming its scores when faced with relevant contextual information (28.40% in 2to1 Related Ctx). These results suggest that unrelated information may help decoder-only models in focusing on the relevant parts of the inputs, enabling them to maintain higher SEM scores in noisy conditions, comparable to the best overall performance (i.e., CodeT5+ and CodeBERT with Related Ctx).

Zero-shot prompted models, shown in orange, perform the worst across all context settings, with significant declines in unrelated context settings. DeepSeek-Coder maintains moderate performance (40–42%) in related and no-context scenarios, but its SEM score plummets to 7.6% and 6.1% under 2to1 and 3to1 Unrelated Cxt, respectively. Qwen2.5-Coder follows a similar trend, dropping from 32.44% (None) and 34.73% (3to1 Related Ctx) to just 5.43% (2to1 Unrelated Ctx) and 4.01% (3to1 Unrelated Ctx). StableCode shows a catastrophic decrease, starting at a weak 12.79% (None) and falling to 2.09% (2to1 Unrelated Ctx) and 1.14% (3to1 Unrelated Ctx), completely failing when met with unnecessary information. The 4-shot prompted variants

exhibit the same exact trend, collapsing when confronted with unrelated information in the inputs, except for Qwen2.5-Coder which makes a comeback in the 3to1 Unrelated Cxt scenario (∼20%). These results demonstrate the inability of instruction-tuned models to efficiently generate offensive code, both when faced with missing information or aided by additional context, due to their lack of task-specific fine-tuning.

> **RQ$_4$**: *To what extent does contextual information (or lack thereof) influence the syntactic and semantic correctness of AI-generated offensive security code?*
>
> Contextual information, both related and unrelated, significantly impacts the syntactic and semantic correctness of AI-generated offensive security code, with clear trends across different models. Fine-tuned encoder-decoder models, especially CodeT5+, perform best with related contextual information but their performance worsens with unrelated context, while decoder-only models, particularly CodeGen, are able to indirectly benefit from it. Instruction-tuned LLMs consistently underperform, struggling across all settings due to their lack of task-specific tuning. These results underscore the need for a tailored training and prompting strategy based on the target DL model.

6 Discussion

This study provides a detailed examination of how contextual information, or lack thereof, impacts the performance of different DL models in offensive code generation. Fine-tuned encoder-decoder models, such as CodeT5+ and CodeBERT, exhibit strong performance in leveraging structured, relevant context, and achieve high accuracy in generating syntactically and semantically correct code (i.e., up to 94% and 63%, for SYN and SEM, respectively). However, their sensitivity to unrelated information leads to significant performance degradation, highlighting their reliance on clean and well-structured input data. These models benefit from related context because their architecture processes the entire input sequence as a whole, enabling the encoder to build a comprehensive representation of the input NL description. This allows the decoder to leverage meaningful relationships between related NL descriptions and code, enhancing the model's ability to generate accurate outputs. Instead, they struggle with unrelated context because the encoder does not prioritize or filter unnecessary information, resulting in noisy or diluted representations.

On the other hand, fine-tuned decoder-only models, including CodeGen and CodeParrot, demonstrate unexpected robustness in noisy environments, often benefiting from unrelated context by leveraging their inherent attention mechanisms. This makes them particularly suitable for tasks involving inconsistent or ambiguous inputs. Indeed, they 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 not directly related, it likely receives lower attention weights during the generation process, allowing the model to focus more on the relevant target description. Conversely, when the previous sentence is related, the attention mechanism might distribute focus across both the previous and target sentences, leading to the blending of information. This ability to dynamically shift focus based on input quality may explain their relative robustness compared to other architectures, which makes them achieve the second-best overall performance across different context settings.

Finally, instruction-tuned LLMs were tested under two prompting methods, zero-shot and 4-shot. Across both settings, they consistently underperform compared to task-specific fine-tuned models, even though they are significantly larger in size. Zero-shot prompting sometimes yields competitive results (e.g., DeepSeek-Coder in the baseline scenario), but these gains are not stable across tasks. Few-shot prompting, rather than systematically improving performance, often introduces variability: some models such as StableCode benefit from additional examples, while larger ones like DeepSeek-Coder and Qwen2.5-Coder sometimes deteriorate. This highlights that LLMs remain strongly dependent on prompt design and sensitive to input length and relevance. Moreover, model size alone does not guarantee robustness: fine-tuned models with hundreds of millions of parameters outperform LLMs with billions of parameters, reinforcing the importance of domain adaptation.

The results also emphasize the importance of balancing the quality and quantity of contextual information. Our experiments show that while moving from a no-context baseline to a 2to1 context improves performance, extending to a 3to1 context often provides little additional benefit and can even degrade results. This suggests an early saturation point: exploit-related shellcodes are typically short, with only one or two preceding instructions contributing relevant context. Beyond that, additional descriptions introduce redundancy or noise rather than useful signal. Indeed, we also observed that identifying sufficient 3to1 examples in real datasets was already challenging, reflecting the way offensive code is structured in practice. From a broader perspective, this indicates that the optimal context window for offensive assembly code generation may be narrower than for other software engineering tasks. Unlike higher-level programming, where longer function or module histories may provide meaningful dependencies, shellcodes are compact, instruction-focused, and highly sensitive to low-level state changes. As a result, extending the window length beyond two preceding intents does not enhance reasoning but risks diluting useful signals.

From a practical perspective, these findings highlight key considerations for deploying AI models for offensive security testing. Fine-tuned models excel in tasks requiring precision and structured input, while decoder-only models indirectly benefit from unrelated additional information or unpredictable environments. Instruction-tuned LLMs, despite their flexibility, require enhancements like fine-tuning or adaptive context filtering to perform effectively in specialized domains. Hybrid approaches combining fine-tuning and instruction-tuning

could bridge the gap between precision and adaptability. This research also demonstrates the potential of a streaming framework for contextual reasoning. By maintaining a sliding window of previous instructions as context, models do not need to repeat earlier prompts but can dynamically infer the information necessary for efficient code generation. Such advancements pave the way for more robust and reliable AI-driven tools for software security development.

7 Threats to Validity

Internal Validity. Internal validity concerns the accuracy of the results and whether they are influenced by confounding factors. One potential threat arises from the construction process of the dataset. 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, we selected shellcodes covering diverse functionalities, complexities, and objectives to ensure a balanced representation of real-world offensive security tasks. A further threat arises from the natural imbalance across dataset categories (e.g., 3to1 samples being rarer than no-context instructions). Rather than artificially balancing the dataset, we preserved these proportions to reflect the way shellcodes are structured in practice. Fairness was maintained by applying uniform split ratios (80/10/10) and by standardizing test sets to $\sim$100 samples per category, ensuring controlled and comparable evaluation conditions.

Another possible concern is that concatenating natural language intents into contextualized inputs (e.g., 2to1 or 3to1 settings) could introduce spurious correlations, i.e., forcing the model to perceive links between instructions that are not truly dependent. We stress that this is not an unintended artifact but the very foundation of our experimental design. Real exploits naturally contain both *(i)* sequences of strictly dependent instructions (e.g., clearing a register before moving a value) and *(ii)* instructions that co-occur without direct dependency (e.g., incrementing a counter before setting up a syscall). To reflect these authentic conditions, we deliberately created two categories: related context (2to1, 3to1), where preceding instructions are genuinely required to interpret the target line, and unrelated context (2to1, 3to1), where preceding instructions come from the same shellcode but serve distinct purposes.

This design allows us to test whether models can exploit meaningful dependencies while ignoring irrelevant or misleading ones. The results show that models behave differently across categories, surfacing model-specific differences in how context is processed. Encoder-decoder models gain accuracy with related context but drop on unrelated context, decoder-only models remain more robust, and instruction-tuned LLMs prove the most sensitive. These patterns indicate that models are not simply exploiting accidental co-occurrence, but are being challenged to distinguish genuine from spurious signals.

A final threat is related to the manual annotation process used to create NL descriptions for the dataset. Our aim was to mimic the variability in specificity and writing styles of developers 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. Finally, we employed cross-validation and collaborative resolution of inconsistencies. This multi-faceted approach to dataset annotation seeks to capture a realistic range of expression and technical detail, also enhancing the external validity of our findings.

External Validity. External validity pertains to the generalizability of the findings beyond the experimental setup, which, in our case, might be impacted by the choice of models. To mitigate this threat, we carefully selected a variety of nine state-of-the-art models with diverse architectures and training procedures, including fine-tuned encoder-decoder and decoder-only models and instruction-tuned LLMs, ensuring a representation of current advancements in the field [42,78,70,11,43]. This careful selection aims to ensure that our findings reflect broader trends in model performance for code generation tasks.

One limitation, however, is that our study primarily focuses on small to medium-sized models (up to 7B parameters). We acknowledge that model size is an important factor influencing contextual reasoning capabilities in LLMs. Our decision was guided by practical constraints, including GPU memory limitations and model deployment complexity. While larger models such as DeepSeek-Coder-33B may offer performance improvements, recent technical reports indicate that scaling from 6.7B to 33B parameters yields only a modest average gain of +3% across 8 programming languages in the Multilingual HumanEval Benchmark [13]. Given the 4–5× increase in model size, we believe this marginal benefit does not justify the substantial increase in computational cost and reproducibility challenges.

Recent large-scale studies on local LLM deployment also show that developers typically favor smaller models (<7B parameters) because they are more practical to run locally, requiring fewer computational resources, easier to reproduce, and able to preserve competitive performance despite their reduced scale [4]. Furthermore, we note that other state-of-the-art LLMs, such as CodeLlama, are licensed to explicitly prohibit exploit or shellcode generation, even in academic contexts, which restricts their applicability to our study. Future work plans to explore the integration of larger-scale models such as DeepSeek-Coder-33B.

We deliberately excluded other public AI models such as GitHub Copilot and OpenAI ChatGPT for similar reasons: *(i)* they impose strong ethical safeguards that prevent generation of offensive code, even in controlled research contexts, and *(ii)* their terms of use explicitly prohibit employing them to generate malicious code, even when intended for *proof-of-concept* purposes in offensive security research. These restrictions made them unsuitable for this

study, which seeks to evaluate the practical capabilities of models in generating security exploits.

Construct Validity. Construct validity examines whether the methods and metrics used accurately measure the intended phenomena. Our evaluation relies on a comprehensive set of five automated similarity metrics (e.g., METEOR, ROUGE-L, CrystalBLEU), which, while commonly used in the field, may not fully capture the nuances of syntactic and semantic correctness in assembly code generation. These metrics often fail to account for scenarios where semantically equivalent outputs differ in syntax or where syntactically similar outputs fail semantically. To address this limitation, we supplemented similarity metrics with *ACCA* [10], a state-of-the-art method based on symbolic execution, enabling a deeper assessment of whether the generated code correctly replicates the intended functionality. No single metric is perfect, but analyzing them collectively allows for a comprehensive evaluation of the code.

A final threat is related to the prompting strategies used for instruction-tuned LLMs, since LLM performance is highly sensitive to the prompt design and number, quality, and relevance of in-context examples. A poorly designed prompt can introduce noise or misalignment, leading to substantial variability in results. This dependency poses a validity threat because performance differences may partly reflect prompt engineering choices rather than intrinsic model capability. To mitigate this, we carefully crafted prompts following best practices in the field, evaluating two different prompting strategies, i.e., zero-shot and few-shot prompting. We used multiple iterations of refinement to minimize ambiguity, and applied a consistent persona-based prompting style across models. Nevertheless, we acknowledge that prompt sensitivity remains an inherent limitation when evaluating LLMs in security-critical tasks.

8 Ethical Considerations

Offensive security is a sub-field of security research that tests security measures from an adversary's perspective, employing ethical hackers to probe systems for vulnerabilities [7,53]. We follow this line of work by investigating the automation of exploit generation, with the goal of surfacing critical weaknesses before they can be abused by attackers. Automating exploit development can both simplify the process of identifying vulnerabilities and provide insights into the technical skills, degree of experience, and intent of adversaries, thereby supporting the development of more effective detection and prevention mechanisms (e.g., jailbreak prevention, misuse detection).

Without promoting or facilitating malicious activity, this research involved generating shellcode and related exploit constructs with the goal of evaluating the behavior and limitations of different AI-based models, i.e., encoder-decoder, decoder-only and large language models, in the context of offensive security. Our objective is to understand how these models interpret incomplete or ambiguous natural language inputs when tasked with generating low-level,

security-relevant code. All experiments were conducted strictly for academic research, and no live, production, or external systems were targeted or affected.

This work contributes to the growing body of literature on AI in security, aligning with recent efforts such as Pa *et al.* [54] and Gupta *et al.* [23], who studied prompt-based misuse of LLMs for exploit and malware generation. We hope our findings will aid in future AI safety efforts and responsible LLM development.

Our dataset consists exclusively of shellcodes that are already publicly available from well-known sources such as Exploit-DB, ShellStorm, and other community-maintained repositories [33, 66, 18]. We did not create or release new exploits; rather, we curated and annotated existing examples to evaluate model performance under varying contextual conditions. The dataset and code are made available solely for reproducibility purposes within the academic community. We believe transparency is essential to enable reproducibility of results. No exploit code or system-specific payloads generated in this study have been released publicly or distributed.

The models evaluated in this study were DeepSeek-Coder-6.7B-Instruct, Qwen2.5-Coder-7B-Instruct, and StableCode-3B-Instruct. We selected these models based on their open availability, instruction-tuned capabilities, and licensing terms that permit security research. Specifically:

- **Qwen2.5-Coder-7B-Instruct** is released under the Apache-2.0 license, which imposes no restrictions on content or usage domain.
- **StableCode-3B-Instruct** is also released under the Apache-2.0 license, enabling unrestricted use for research and experimentation.
- **DeepSeek-Coder-6.7B-Instruct** is distributed under the DeepSeek Model License, which includes clauses requiring lawful and non-harmful use. Our usage complied fully with these terms. All generated assembly snippets were analyzed strictly within a contained research environment. No generated code was deployed on unauthorized systems or used to perform live exploitation.

Importantly, we deliberately excluded models whose licenses prohibit code generation for security testing. For instance, CodeLlama's license explicitly forbids generating malware or exploit code, even for academic purposes. We therefore did not include CodeLlama in our evaluation to ensure compliance with its intended use policy.

9 Conclusion

This study provides a comprehensive evaluation of deep learning models in the challenging domain of offensive security, focusing on their ability to generate accurate code from NL descriptions under varying contextual conditions. By analyzing nine SOTA models, including fine-tuned encoder-decoder and decoder-only architectures alongside instruction-tuned LLMs, we uncover critical insights into how these models handle missing, related, and irrelevant contextual information.

Our findings highlight the clear benefits of leveraging relevant contextual information, particularly for fine-tuned encoder-decoder models (e.g., CodeT5+ and CodeBERT), which show notable improvements in syntactic and semantic correctness when guided by structured, related inputs. However, we observe diminishing returns when extending context beyond a single preceding instruction, suggesting an optimal level of context for effective code generation. Decoder-only models (e.g., CodeGen) are able to indirectly benefit from unrelated information, often using it constructively and achieving the second-best performance after CodeT5+ and CodeBERT with related information. Finally, instruction-tuned LLMs (e.g., DeepSeek-Coder) exhibit limited robustness to contextual variations, emphasizing the need for task-specific optimization in high-stakes domains like offensive security.

This work underscores the critical role of context-aware training and prompting strategies in advancing AI-driven tools for exploit generation. By addressing the limitations of current models in interpreting ambiguous or incomplete NL descriptions, our study offers practical guidance for the design and selection of DL models, contributing to more reliable and efficient applications in offensive security.

Declarations

Author Contributions

Cristina Improta: Writing – original draft, Writing – review & editing, Conceptualization, Formal analysis, Methodology, Software, Validation, Visualization, Investigation. *Pietro Liguori*: Conceptualization, Investigation, Methodology, Supervision, Validation, Writing – review & editing. *Roberto Natella*: Conceptualization, Investigation, Methodology, Resources, Supervision, Writing – review & editing. *Bojan Cukic*: Resources, Writing – review & editing, Supervision. *Domenico Cotroneo*: Conceptualization, Project administration, Resources, Supervision, Writing – review & editing.

Data Availability Statement

The dataset and code to replicate the experiments are available at the following URL: `https://github.com/dessertlab/Software-Exploits-with-Contextual-Information`.

Funding

The authors have no relevant financial or non-financial interests to disclose.

Conflict of interest

The authors declare that they have no conflict of interest.

Ethical Approval

Not applicable.

Informed Consent

Not applicable.

Acknowledgements This work has been partially supported by the MUR PRIN 2022 program, project *FLEGREA*, CUP E53D23007950001, and by the *IDA—Information Disorder Awareness* Project funded by the European Union-Next Generation EU within the SERICS Program through the MUR National Recovery and Resilience Plan under Grant PE00000014.

References

1. Agrawal, R.R., Turchi, M., Negri, M.: 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, pp. 11–20 (2018)
2. Ahmad, W., Chakraborty, S., Ray, B., Chang, K.W.: 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, pp. 2655–2668 (2021)
3. Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M., Brumley, D.: Automatic exploit generation. Communications of the ACM **57**(2), 74–84 (2014)
4. Belcak, P., Heinrich, G., Diao, S., Fu, Y., Dong, X., Muralidharan, S., Lin, Y.C., Molchanov, P.: Small language models are the future of agentic ai. arXiv preprint arXiv:2506.02153 (2025)
5. Bird, S.: NLTK: the natural language toolkit. In: Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions, pp. 69–72 (2006)
6. Botacin, M.: GPThreats-3: Is Automatic Malware Generation a Threat? In: 2023 IEEE Security and Privacy Workshops (SPW), pp. 238–254. IEEE (2023)
7. Bratus, S., Arce, I., Locasto, M.E., Zanero, S.: Why offensive security needs engineering textbooks. Yale Law & Policy Review p. 2 (2013)
8. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.D.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
9. Ciniselli, M., Pascarella, L., Bavota, G.: Deep Learning-based Code Completion: On the Impact on Performance of Contextual Information. In: 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 211–223. IEEE (2024)
10. Cotroneo, D., Foggia, A., Improta, C., Liguori, P., Natella, R.: Automating the correctness assessment of AI-generated code for security contexts. Journal of Systems and Software p. 112113 (2024)
11. Cotroneo, D., Improta, C., Liguori, P., Natella, R.: Vulnerabilities in AI code generators: Exploring targeted data poisoning attacks. In: Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, pp. 280–292 (2024)
12. crystalbleu: Python CrystalBLEU Score Implementation (2022). URL `https://pypi.org/project/crystalbleu/`
13. DeepSeek-AI: DeepSeek-Coder Technical Report. `https://github.com/deepseek-ai/DeepSeek-Coder?tab=readme-ov-file#6-detailed-evaluation-results`
14. Ding, Y., Peng, J., Min, M., Kaiser, G., Yang, J., Ray, B.: Semcoder: Training code language models with comprehensive semantics reasoning. Advances in Neural Information Processing Systems **37**, 60275–60308 (2024)
15. Eghbali, A., Pradel, M.: CrystalBLEU: precisely and efficiently measuring the similarity of code. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–12 (2022)
16. evaluate: Python library evaluate (2022). URL `https://pypi.org/project/evaluate/`
17. Evtikhiev, M., Bogomolov, E., Sokolov, Y., Bryksin, T.: Out of the BLEU: how should we assess quality of the code generation models? Journal of Systems and Software **203**, 111741 (2023)
18. Exploit-db: Exploit Database Shellcodes. `https://www.exploit-db.com/shellcodes?platform=linux_x86/` (2023)

19. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M.: 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, *Findings of ACL*, vol. EMNLP 2020, pp. 1536–1547. Association for Computational Linguistics (2020). DOI 10.18653/v1/2020.findings-emnlp.139. URL https://doi.org/10.18653/v1/2020.findings-emnlp.139
20. Foster, J.: Sockets, Shellcode, Porting, and Coding: Reverse Engineering Exploits and Tool Coding for Security Professionals. Elsevier Science (2005). URL https://books.google.it/books?id=ZNI5dvBSfZoC
21. Gao, S., Wen, X.C., Gao, C., Wang, W., Zhang, H., Lyu, M.R.: What makes good in-context demonstrations for code intelligence tasks with llms? In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 761–773. IEEE (2023)
22. Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y., et al.: DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. arXiv preprint arXiv:2401.14196 (2024)
23. Gupta, M., Akiri, C., Aryal, K., Parker, E., Praharaj, L.: From ChatGPT to ThreatGPT: Impact of generative AI in cybersecurity and privacy. IEEE Access (2023)
24. huggingface: CodeParrot. https://huggingface.co/codeparrot/codeparrot (2024)
25. Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., et al.: Qwen2.5-coder technical report. arXiv preprint arXiv:2409.12186 (2024)
26. Improta, C., Liguori, P., Natella, R., Cukic, B., Cotroneo, D.: Enhancing robustness of AI offensive code generators via data augmentation. Empirical Software Engineering **30**(1), 7 (2025)
27. Kapu, N.J., Sreejith, M.: DemoCraft: Using In-Context Learning to Improve Code Generation in Large Language Models. arXiv preprint arXiv:2411.00865 (2024)
28. Kim, D., MacKinnon, T.: Artificial intelligence in fracture detection: transfer learning from deep convolutional neural networks. Clinical radiology **73**(5), 439–445 (2018)
29. Lavie, A., Agarwal, A.: 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, StatMT '07, p. 228–231. Association for Computational Linguistics, USA (2007)
30. Lewis, M.: Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. arXiv preprint arXiv:1910.13461 (2019)
31. Li, B., Liu, H., Wang, Z., Jiang, Y., Xiao, T., Zhu, J., Liu, T., Li, C.: Does multi-encoder help? A case study on context-aware neural machine translation. In: D. Jurafsky, J. Chai, N. Schluter, J.R. Tetreault (eds.) Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020, pp. 3512–3518. Association for Computational Linguistics (2020). DOI 10.18653/v1/2020.acl-main.322. URL https://doi.org/10.18653/v1/2020.acl-main.322
32. Li, Z., Wang, X., Aw, A., Chng, E.S., Li, H.: Named-entity tagging and domain adaptation for better customized translation. In: Proceedings of the seventh named entities workshop, pp. 41–46 (2018)
33. Liguori, P., Al-Hossami, E., Cotroneo, D., Natella, R., Cukic, B., Shaikh, S.: Shellcode_IA32: A dataset for automatic shellcode generation. In: Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021), pp. 58–64. Association for Computational Linguistics, Online (2021). DOI 10.18653/v1/2021.nlp4prog-1.7. URL https://aclanthology.org/2021.nlp4prog-1.7
34. Liguori, P., Al-Hossami, E., Cotroneo, D., Natella, R., Cukic, B., Shaikh, S.: Can we generate shellcodes via natural language? An empirical study. Automated Software Engineering **29**(1), 30 (2022)
35. Liguori, P., Al-Hossami, E., Orbinato, V., Natella, R., Shaikh, S., Cotroneo, D., Cukic, B.: Evil: exploiting software via natural language. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), pp. 321–332. IEEE (2021)
36. Liguori, P., Improta, C., Natella, R., Cukic, B., Cotroneo, D.: Who evaluates the evaluators? On automatic metrics for assessing AI-based offensive code generators. Expert Systems with Applications **225**, 120073 (2023). DOI https://doi.org/

10.1016/j.eswa.2023.120073. URL https://www.sciencedirect.com/science/article/pii/S0957417423005754
37. Liguori, P., Improta, C., Natella, R., Cukic, B., Cotroneo, D.: Enhancing ai-based generation of software exploits with contextual information. In: 2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE), pp. 180–191. IEEE (2024)
38. Liguori, P., Marescalco, C., Natella, R., Orbinato, V., Pianese, L.: The power of words: Generating PowerShell attacks from natural language. In: 18th USENIX WOOT Conference on Offensive Technologies (WOOT 24), pp. 27–43. USENIX Association, Philadelphia, PA (2024). URL https://www.usenix.org/conference/woot24/presentation/liguori
39. Lin, C.Y.: Rouge: A package for automatic evaluation of summaries. In: Text summarization branches out, pp. 74–81 (2004)
40. Lin, S.W., Tolmach, P., Liu, Y., Li, Y.: Solsee: a source-level symbolic execution engine for solidity. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1687–1691 (2022)
41. Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V.: Roberta: A robustly optimized BERT pretraining approach. CoRR **abs/1907.11692** (2019). URL http://arxiv.org/abs/1907.11692
42. Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al.: Codexglue: A machine learning benchmark dataset for code understanding and generation. In: J. Vanschoren, S. Yeung (eds.) Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual (2021). URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html
43. Lyu, C., Yan, L., Xing, R., Li, W., Samih, Y., Ji, T., Wang, L.: Large language models as code executors: An exploratory study. arXiv preprint arXiv:2410.06667 (2024)
44. Mashhadi, E., Hemmati, H.: 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, pp. 505–509. IEEE (2021). DOI 10.1109/MSR52588.2021.00063
45. Mastropaolo, A., Pascarella, L., Guglielmi, E., Ciniselli, M., Scalabrino, S., Oliveto, R., Bavota, G.: 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, pp. 2149–2160. IEEE (2023). DOI 10.1109/ICSE48619.2023.00181. URL https://doi.org/10.1109/ICSE48619.2023.00181
46. Megahed, H.: Penetration Testing with Shellcode: Detect, exploit, and secure network-level and operating system vulnerabilities. Packt Publishing (2018)
47. Mirsky, Y., Demontis, A., Kotak, J., Shankar, R., Gelei, D., Yang, L., Zhang, X., Pintor, M., Lee, W., Elovici, Y., et al.: The threat of offensive AI to organizations. Computers & Security p. 103006 (2022)
48. Modrzejewski, M., Exel, M., Buschbeck, B., Ha, T.L., Waibel, A.: Incorporating external annotation to improve named entity translation in NMT. In: Proceedings of the 22nd Annual Conference of the European Association for Machine Translation, pp. 45–51 (2020)
49. NASM: Netwide Assembler (NASM). https://www.nasm.us (2024)
50. Natella, R., Liguori, P., Improta, C., Cukic, B., Cotroneo, D.: AI Code Generators for Security: Friend or Foe? IEEE Security & Privacy **22**(5) (2024). DOI 10.1109/MSEC.2024.3355713
51. Ni, A., Allamanis, M., Cohan, A., Deng, Y., Shi, K., Sutton, C., Yin, P.: Next: Teaching large language models to reason about code execution. arXiv preprint arXiv:2404.14662 (2024)
52. Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C.: 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). URL https://openreview.net/pdf?id=iaYcJKpY2B_

53. Oakley, J.G.: The state of modern offensive security. In: Professional Red Teaming: Conducting Successful Cybersecurity Engagements, pp. 29–41. Springer (2019)
54. Pa Pa, Y.M., Tanizaki, S., Kou, T., Van Eeten, M., Yoshioka, K., Matsumoto, T.: An Attacker's Dream? Exploring the Capabilities of ChatGPT for Developing Malware. In: Proceedings of the 16th Cyber Security Experimentation and Test Workshop, pp. 10–18 (2023)
55. Pan, Z., Hu, X., Xia, X., Yang, X.: Enhancing Repository-Level Code Generation with Integrated Contextual Information. arXiv preprint arXiv:2406.03283 (2024)
56. Papineni, K., Roukos, S., Ward, T., Zhu, W.: Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA, pp. 311–318. ACL (2002). DOI 10.3115/1073083.1073135. URL https://aclanthology.org/P02-1040/
57. Pinnaparaju, N., Adithyan, R., Phung, D., Tow, J., Baicoianu, J., Datta, A., Zhuravinskyi, M., Mahan, D., Bellagente, M., Riquelme, C., et al.: Stable code technical report. arXiv preprint arXiv:2404.01226 (2024)
58. pylcs: Python library pylcs (2023). URL https://pypi.org/project/pylcs/
59. Python: tokenize (2023). URL https://docs.python.org/3/library/tokenize.html
60. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J.: Exploring the limits of transfer learning with a unified text-to-text transformer. J. Mach. Learn. Res. **21**, 140:1–140:67 (2020). URL http://jmlr.org/papers/v21/20-074.html
61. Ramos, D.A., Engler, D.: {Under-Constrained} symbolic execution: Correctness checking for real code. In: 24th USENIX Security Symposium (USENIX Security 15), pp. 49–64 (2015)
62. Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., Ma, S.: Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297 (2020)
63. rouge: Python ROUGE Score Implementation (2021). URL https://pypi.org/project/rouge/
64. Ruan, X., Yu, Y., Ma, W., Cai, B.: Prompt Learning for Developing Software Exploits. In: Proceedings of the 14th Asia-Pacific Symposium on Internetware, pp. 154–164 (2023)
65. Scherrer, Y., Tiedemann, J., Loáiciga, S.: Analysing concatenation approaches to document-level NMT in two different domains. In: Proceedings of the Fourth Workshop on Discourse in Machine Translation (DiscoMT 2019), pp. 51–61. Association for Computational Linguistics, Hong Kong, China (2019). DOI 10.18653/v1/D19-6506. URL https://aclanthology.org/D19-6506
66. Shell-storm: Shellcodes database for study cases. http://shell-storm.org/shellcode/ (2022)
67. spaCy: Industrial-Strength Natural Language Processing (2023). URL https://spacy.io/
68. Steidl, D., Hummel, B., Juergens, E.: Quality analysis of source code comments. In: 2013 21st international conference on program comprehension (icpc), pp. 83–92. Ieee (2013)
69. Tiedemann, J., Scherrer, Y.: Neural machine translation with extended context. In: B.L. Webber, A. Popescu-Belis, J. Tiedemann (eds.) Proceedings of the Third Workshop on Discourse in Machine Translation, DiscoMT@EMNLP 2017, Copenhagen, Denmark, September 8, 2017, pp. 82–92. Association for Computational Linguistics (2017). DOI 10.18653/v1/w17-4811. URL https://doi.org/10.18653/v1/w17-4811
70. Tipirneni, S., Zhu, M., Reddy, C.K.: Structcoder: Structure-aware transformer for code generation. ACM Transactions on Knowledge Discovery from Data **18**(3), 1–20 (2024)
71. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. In: Advances in neural information processing systems, pp. 5998–6008 (2017)
72. Voita, E., Serdyukov, P., Sennrich, R., Titov, I.: Context-aware neural machine translation learns anaphora resolution. In: I. Gurevych, Y. Miyao (eds.) Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 1264–1274. Association for Computational Linguistics, Melbourne, Australia (2018). DOI 10.18653/v1/P18-1117. URL https://aclanthology.org/P18-1117

73. Wang, H., Xia, X., Lo, D., He, Q., Wang, X., Grundy, J.: Context-aware retrieval-based deep commit message generation. ACM Transactions on Software Engineering and Methodology (TOSEM) **30**(4), 1–30 (2021)
74. Wang, L., Tu, Z., Way, A., Liu, Q.: Exploiting cross-sentence context for neural machine translation. In: M. Palmer, R. Hwa, S. Riedel (eds.) Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017, pp. 2826–2831. Association for Computational Linguistics (2017). DOI 10.18653/v1/d17-1301. URL `https://doi.org/10.18653/v1/d17-1301`
75. Wang, W., Liu, K., Chen, A.R., Li, G., Jin, Z., Huang, G., Ma, L.: Python Symbolic Execution with LLM-powered Code Generation. arXiv preprint arXiv:2409.09271 (2024)
76. Wang, Y., Le, H., Gotmare, A.D., Bui, N.D., Li, J., Hoi, S.C.: Codet5+: Open code large language models for code understanding and generation. arXiv preprint arXiv:2305.07922 (2023)
77. Wang, Z.: 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, pp. 29–35. Springer (2013)
78. Wei, Y., Xia, C.S., Zhang, L.: 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, pp. 172–184 (2023)
79. Xu, D., Chen, K., Lin, M., Lin, C., Wang, X.: AutoPwn: Artifact-assisted Heap Exploit Generation for CTF PWN Competitions. IEEE Transactions on Information Forensics and Security (2023)
80. Xu, X., Huang, Q., Wang, Z., Feng, Y., Zhao, D.: Towards context-aware code comment generation. In: Findings of the Association for Computational Linguistics: EMNLP 2020, pp. 3938–3947. Association for Computational Linguistics (2020)
81. Yadegari, B., Debray, S.: Symbolic execution of obfuscated code. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 732–744 (2015)
82. Yang, G., Chen, X., Zhou, Y., Yu, C.: 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, pp. 361–372. IEEE (2022)
83. Yang, G., Zhou, Y., Chen, X., Zhang, X., Han, T., Chen, T.: ExploitGen: Template-augmented exploit code generation based on CodeBERT. Journal of Systems and Software **197**, 111577 (2023)
84. Zheng, Z., Yue, X., Huang, S., Chen, J., Birch, A.: Towards making the most of context in neural machine translation. In: C. Bessiere (ed.) Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020, pp. 3983–3989. ijcai.org (2020). DOI 10.24963/ijcai.2020/551. URL `https://doi.org/10.24963/ijcai.2020/551`
85. Zhou, C., Liu, P., Xu, P., Iyer, S., Sun, J., Mao, Y., Ma, X., Efrat, A., Yu, P., Yu, L., Zhang, S., Ghosh, G., Lewis, M., Zettlemoyer, L., Levy, O.: LIMA: less is more for alignment. CoRR **abs/2305.11206** (2023). DOI 10.48550/ARXIV.2305.11206. URL `https://doi.org/10.48550/arXiv.2305.11206`
86. Zhu, X., Zhou, W., Han, Q.L., Ma, W., Wen, S., Xiang, Y.: When Software Security Meets Large Language Models: A Survey. IEEE/CAA Journal of Automatica Sinica **12**(2), 317–334 (2025)