The Journal of Systems and Software 216 (2024) 112113

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Automating the correctness assessment of AI-generated code for security contexts[☆]

Domenico Cotroneo, Alessio Foggia, Cristina Improta, Pietro Liguori [*], Roberto Natella

University of Naples Federico II, Naples, Italy

ARTICLE INFO

Dataset link: https://github.com/dessertlab/ACCA, https://github.com/dessertlab/Shellcode_IA32

Keywords:
Code correctness
AI code generators
Assembly
Offensive security
Symbolic execution

ABSTRACT

Evaluating the correctness of code generated by AI is a challenging open problem. In this paper, we propose a fully automated method, named *ACCA*, to evaluate the correctness of AI-generated code for security purposes. The method uses symbolic execution to assess whether the AI-generated code behaves as a reference implementation. We use *ACCA* to assess four state-of-the-art models trained to generate security-oriented assembly code and compare the results of the evaluation with different baseline solutions, including output similarity metrics, widely used in the field, and the well-known ChatGPT, the AI-powered language model developed by OpenAI.

Our experiments show that our method outperforms the baseline solutions and assesses the correctness of the AI-generated code similar to the human-based evaluation, which is considered the ground truth for the assessment in the field. Moreover, *ACCA* has a very strong correlation with the human evaluation (Pearson's correlation coefficient $r = 0.84$ on average). Finally, since it is a full y automated solution that does not require any human intervention, the proposed method performs the assessment of every code snippet in ~ 0.17 s on average, which is definitely lower than the average time required by human analysts to manually inspect the code, based on our experience.

1. Introduction

Artificial Intelligence (AI) code generators use Neural Machine Translation (NMT) models to turn natural language (NL) descriptions into programming code. They represent a powerful asset in the arsenal of cybersecurity professionals and malicious programmers. Indeed, AI (*offensive*) code generators are becoming an attractive solution to creating *proof-of-concept* exploits in order to assess the exploitability and severity of software vulnerabilities (Yang et al., 2023; Liguori et al., 2021b; Ruan et al., 2023), letting the AI helping developers to generate low-level (i.e., assembly) and complex code, with a reduced effort and improved effectiveness.

Despite the dramatic increase in the adoption of AI code generators, they still have limitations and potential drawbacks. For example, they may not always generate code that is *correct*, i.e., code that performs what is required from the NL description, as they may struggle with more complex programming tasks that require human creativity and problem-solving skills, or may incorrectly interpret developers' descriptions. Furthermore, AI code generators can introduce security vulnerabilities if not properly tested and validated (Pearce et al., 2022; Siddiq et al., 2022; Tony et al., 2023). For these reasons, assessing the correctness of AI-generated code becomes a crucial challenge.

From the existing literature, it comes out that one of the most effective ways to assess the correctness of generated code is to perform a manual code review (i.e., *human evaluation*) (Parvez et al., 2021; Kononenko et al., 2016). This involves having a human expert review the code and identify any errors or inconsistencies with the NL description. However, human evaluation has several limitations. First, manual analysis can be a time-consuming process. Indeed, reviewers must carefully examine each line of code and thoroughly test the software to ensure that it meets the intended requirements and NL specifications. This process also requires reviewers to be highly knowledgeable about the programming language, development environment, and intended functionality of the code to provide accurate assessments. Moreover, the analysis can be subjective, as different reviewers may have different interpretations of the code and its intended functionality, depending on the expertise and experience of the reviewer. This can lead to inconsistent assessments of code correctness. Last but not least, manual analysis is prone to human error, as reviewers may miss subtle errors

[☆] Editor: Burak Turhan.

[*] Corresponding author.

E-mail addresses: cotroneo@unina.it (D. Cotroneo), alessio.foggia@unina.it (A. Foggia), cristina.improta@unina.it (C. Improta), pietro.liguori@unina.it (P. Liguori), roberto.natella@unina.it (R. Natella).

https://doi.org/10.1016/j.jss.2024.112113
Received 21 October 2023; Received in revised form 10 March 2024; Accepted 23 May 2024
Available online 24 May 2024

or inconsistencies in the code, or may introduce errors and biases into their assessments due to factors such as fatigue, distractions, or subjective opinions. From the above considerations, it is clear that what we gained from the help of AI, we lost due to the manual review.

Unfortunately, currently no fully automated solution can perform the semantic assessment of AI-generated code comparably to human evaluation. In fact, although existing automated testing and code analysis tools can effectively identify errors or inconsistencies in code, they do not provide any insights into whether the code is what is actually required by developers (Ayewah et al., 2008; Bessey et al., 2010; Liu et al., 2019a; Pylint, 2024). Moreover, these solutions often require in inputs entire, compilable programs (e.g., entire functions) rather than single code snippets, which is instead often the case with AI-generated code. Indeed, code snippets produced by models are often not complete programs but rather fragments or components of code that, on their own, might not be directly compilable. This characteristic is due to the nature of the training data available and the current capabilities of AI code generation models, which are adept at generating specific functional code snippets rather than entire, complex applications. This highlights the limitation in evaluating the correctness and functionality of these snippets using traditional software testing methods (e.g., unit-level testing) (Araujo and Rizzo Vincenzi, 2020; Lukasczyk et al., 2023).

Besides the automated solution issue, there is a more important one, i.e., how to evaluate the correctness of AI-generated code. Indeed, previous studies proposed a large number of *output similarity* metrics, i.e., metrics computed by comparing the textual similarity of generated code with a ground-truth reference implementation (Reiter and Belz, 2009; Shterionov et al., 2018; Liguori et al., 2023). The major advantage of the proposed metrics is that they are reproducible, easily tuned, and time-saving. However, in the context of programming code generation, existing metrics are not able to fully reflect the correctness of the code.

As illustrated in the next section, generated code can be different from the reference but still be correct (e.g., the assembly conditional jumps `jz` and `je` are different instructions that can be used to perform the same operation); or, there can be subtle differences between the generated and the reference code, which can be similar yet produce different outputs (e.g., the assembly conditional jumps `je` and `jne` are syntactically similar instructions, but they perform the opposite operation). Hence, it is crucial to develop novel, more accurate methods for automatically evaluating the correctness of AI-generated code.

This paper proposes a method, named *ACCA* (*Assembly Code Correctness Assessment*), to automatically assess the correctness of assembly AI-generated code without any human effort. More precisely, our solution leverages *symbolic execution*, i.e., a state-of-the-art solution for program analysis based on abstract execution, which consists of simulating the execution of a program providing symbolic values to evaluate its behavior, to assess whether the generated code behaves as a reference implementation, despite syntactic differences between the reference and the generated code.

We apply *ACCA* to assess four state-of-the-art NMT models in the generation of security-oriented code in assembly language starting from NL descriptions in the English language and compare the results of *ACCA* with the human evaluation and several baseline assessment solutions, including a wide range of output similarity metrics and the well-known ChatGPT by OpenAI. We show that the proposed method provides an almost perfect assessment of the code's correctness and has a very strong correlation with the human evaluation, outperforming all the baseline assessment solutions.

In the following, Section 2 introduces a motivating example; Section 3 describes *ACCA*; Section 4 presents the experimental setup; Section 5 shows the experimental results; Section 7 presents the related work; Section 8 concludes the paper.

2. Motivating example

In code generation tasks, datasets play a crucial role in training and evaluating AI models. These datasets typically consist of pairs of inputs and outputs, where the input is a natural language (NL) description of a coding task, and the output is the code snippet that fulfills the described task. NL descriptions refer to human-readable explanations or specifications of what a particular piece of code is intended to achieve. These descriptions serve a similar purpose to code comments but are used as input for AI code generation models to produce corresponding code snippets.

These corpora are commonly split into *training data*, i.e., the data used to feed the model, *validation data*, i.e., the data used to tune the model's parameters, and *test data*, i.e., the data used to evaluate the model in the generation of the code starting from new NL descriptions (i.e., the NL intents in the test data are never seen by the model in the train and validation data).

The most practical solution to assess the performance of the NMT models in the code generation is to compare, for every NL description of the test data (i.e., the input), the model's prediction with the code snippet (i.e., the output) in the test set, which is considered the *ground-truth* for the evaluation. To this aim, state-of-the-art provides a set of metrics that estimate the similarity between the code generated by NMT models and the code snippets in the test set. However, output similarity metrics cannot properly assess whether two pieces of code are different 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).

State-of-the-art also proposed solutions that apply contrastive learning to discern between semantically similar and dissimilar code samples (Wu and Zhang, 2022; Massarelli et al., 2021; Ullah and Oh, 2021). Code embedding models via contrastive learning are designed to learn rich, high-dimensional representations of code snippets by embedding them into a vector space. While they present an advanced method for capturing the semantic features of code snippets, several challenges limit their immediate applicability as a primary evaluation solution for AI-generated code. First, they require significant computational resources not only for training but also for inference. Evaluating large datasets or conducting extensive experiments can become computationally expensive and time-consuming, limiting the scalability of this solution. Moreover, the effectiveness of code embedding models is heavily dependent on the quality and diversity of the training data. Models trained on limited or biased datasets may not accurately capture the full spectrum of semantic equivalences, potentially skewing the evaluation results. This issue is further exacerbated in our case study, where there is a lack of corpora used for offensive code security, especially in low-level languages. On the contrary, output similarity metrics, which estimate the textual similarity of the model's predictions with respect to a ground truth reference, are easy to use, time-saving, and require low resource requirements, and represent the most common solution to assess models in the SOTA.

For the aforementioned reasons, *human evaluation* is considered the golden standard for assessing the correctness of the code generated by the models (Evtikhiev et al., 2023). Through manual inspection of every model's predictions, human evaluators assess if the code generated by the models is *semantically correct*, i.e., if the output is the exact translation of the NL intent into the target programming language. Semantic correctness implies *syntax correctness*, i.e., a code prediction that performs what is described in the NL intent must also adhere to the syntax rules of the target programming languages. Human evaluation classifies the code as correct or incorrect by assigning a value equal to 1 or 0, respectively.

As a simple example, consider the intent "*transfer EAX contents into EDX register*", which translates, on the 32-bit version of the x86 instruction set architecture (IA-32), to the assembly snippet:

```
mov EDX, EAX
```

An alternative method to copy the contents of a register into another is by pushing and popping its value onto the stack. Therefore, a semantically equivalent implementation of this copy is the code:

```
push EAX
pop EDX
```

Despite the model's prediction being both syntactically and semantically correct, output similarity metrics are not able to grasp the equivalence between the two snippets since they base their calculation on character and/or token similarity. Therefore, this translation results in low scores[1] for several output similarity metrics widely used in the field (see Section 4.3), such as *BLEU-4* (0.11) and *Edit Distance* (0.31).

The opposite occurs with the intent "*clear the EDX register and move 5 in the lowest byte of the register*", which translates to the assembly snippet:

```
xor EDX, EDX
mov DL, 5
```

If the model generates the snippet:

```
xor EDX, EDX
mov BL, 5
```

then prediction and reference differ by a single character, yet the code does not accomplish the same task. Indeed, the lowest byte of EDX is stored in the DL register, while BL contains the lowest byte of EBX. Automatic metrics fail to account for situations like this. For instance, the *Edit Distance* between these two pieces of code is 0.96, while the *BLEU-4* is 0.65, which are considered high values. Differently, a human evaluator would appropriately classify this snippet as semantically incorrect, since it does not perform the intended operation, although it properly respects the syntax of the assembly language.

However, since the human analyst needs to check the syntax and the semantics of every output generated by the models, human evaluation is often unfeasible. Indeed, the huge amount of data to scrutinize makes the analysis time-consuming and prone to errors.

3. Proposed method

To overcome the limitations in the assessment of AI-generated assembly code described in the previous section, we propose *ACCA*, a method to automatically assemble and symbolically execute both the reference (i.e., the ground truth) and predicted snippets (i.e., 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.

First, *ACCA* compares, at the string level, the prediction with the reference to assess its correctness, because, if the prediction is equal to the reference, then we assume the code prediction is correct. If this is not the case, the method assesses whether the code is syntactically correct. Indeed, if a code prediction is not structured according to the rules of the target programming languages, it is classified as incorrect.

If this is not the case, i.e., if the prediction differs from the ground truth reference and the prediction is syntactically correct, *ACCA* generates two source code files, one for the reference and one for the predicted snippet. It then assembles and links them to produce the executable files needed for the symbolic execution. At this point, the method symbolically executes both files resulting from the assembling process to assess whether they are equivalent.

Finally, *ACCA* returns the pair of syntactic correctness value (SYN) and the semantic correctness value (SEM) of the code predicted by the model, which is equal to 1 when the correctness is verified, 0 otherwise. Fig. 1 shows a detailed flowchart of the syntactic and semantics correctness evaluation process.

In the rest of this section, we detail the phases of the workflow. For the sake of simplicity, we showcase examples of assembly code for the Linux OS running on IA-32 architecture to describe the evaluation process, although the proposed method is not restricted to this specific context, and it can be applied to different instruction set architectures and operating systems.

3.1. String comparison

ACCA first checks whether the predicted code snippet perfectly matches the ground truth by performing a string comparison. If they match (i.e., the code generated by the model is equal to ground truth), the prediction is considered both syntactically and semantically correct (i.e., SYN = 1, SEM = 1) and the evaluation ends. Otherwise, i.e., if they differ, the method proceeds with the evaluation of the syntactic and semantic correctness. The preliminary string comparison is done to speed up the evaluation process by skipping the symbolic execution process when not needed, i.e., when the prediction perfectly matches the reference snippet and is, therefore, correct.

3.2. Assembling process

The purpose of the assembling process is to assess whether each code snippet generated by the model adheres to the syntax rules of the programming language it is written in, i.e., to check whether it is compilable. Since NMT is still far from producing entire complex programs, the output predicted by the models is a portion of an entire program (i.e., a single-line or multi-line statement). Thus, *ACCA* constructs a complete program by adding the necessary entry point and exit point code. For instance, consider the following code snippets for IA-32:

```
cmp EAX, EBX
je loop
pop ECX
```

This code compares the contents of two registers and, based on the result, either performs a jump operation or reads from the stack and saves the value into another register. The snippet is syntactically correct according to the assembly language, yet it is incomplete for the execution since code snippets need to contain a minimal set of information to be properly executed.

In the case of Linux OS, these instructions are kept in the `text` section, which must begin with the declaration `global _start` to define the initial program entry point. Moreover, to make sure the program terminates correctly, the proposed method also includes a fictitious label that represents the exit address the code jumps to at the end of its execution. This label is declared in the `data` section, which contains initialized read-write data.

Therefore, to have a complete program, the code is modified as follows:

```
section .data
exit_addr db 0x56
section .text
global _start
_start:
        cmp EAX, EBX
        je loop
        pop ECX
        jmp exit_addr
```

[1] Scores of output similarity metrics range between 0 and 1.

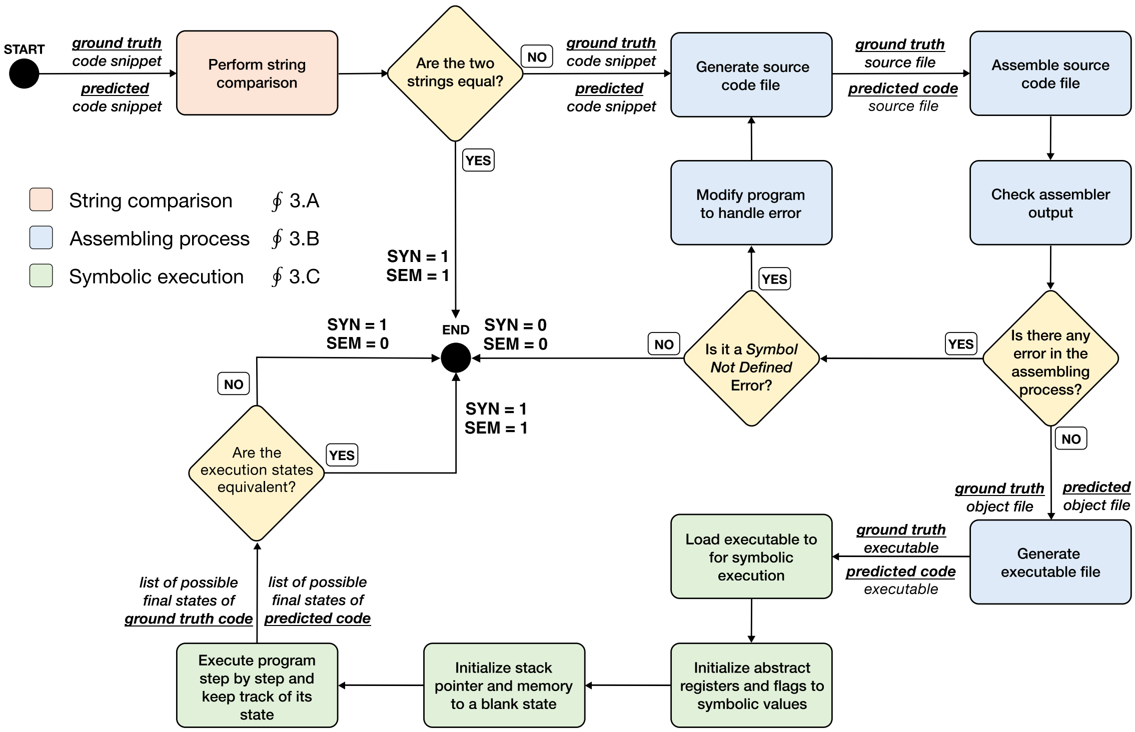


Fig. 1. Detailed flowchart of *ACCA*.

Once the whole program is created, *ACCA* generates a source file and leverages an assembler to assess its syntactic correctness. Indeed, if the programs compile, then all the instructions of the programs are syntactically correct, therefore also the code generated by the model respects the structure of assembly programming language. There are three possible output scenarios for the compilation:

- *No errors*, in which the assembling process is completely correct;
- *Warnings*, in which the assembler reports some type of warning (e.g., word data exceeds bounds, invalid register size specification ignored, etc.), but the compilation still terminates without errors;
- *Errors*, in which the assembling process results in an error that prevents the code from being assembled.

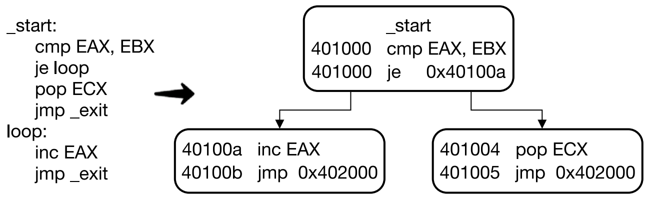
In the first two cases, the output produced by the model is considered syntactically correct (i.e., SYN = 1). Warnings are considered acceptable since they indicate issues involving bad practice, but are not severe enough to prevent the code compilation. Indeed, compiler warnings are raised for potential issues that do not violate the language's syntax rules (e.g., uninitialized space declared in .text section, invalid register size specification ignored). Moreover, these warnings are mainly due to the lines of code added by our method to perform the assessment. Therefore, distinguishing between code that compiles with warnings and code that compiles without warnings introduces a level of granularity that goes beyond the scope of syntax rule adherence as their presence does not mean that the code is syntactically incorrect.

When the compilation produces an error (i.e., the third case), we investigate the nature of the error. More precisely, we focus on a specific category of the error raised by the compiler, the *Symbol-Not-Defined (SND) errors*, which occur when the code contains a symbol (e.g., a label or variable) that has not been previously defined or initialized. We handle this category of errors appropriately. Indeed, since the predicted snippets contain only one or a few instructions, they might reference a label or variable defined in a different portion of the program, which leads to an assembling error even when the program is syntactically correct. Indeed, in the context of AI-generated code, it is common to deal with incomplete code instances or to assume context that is not explicitly provided within the snippet itself. This is particularly true for assembly language, where labels and jump instructions play a critical role in control flow. Our decision to specifically manage SND errors stems from their common occurrence in AI-generated assembly code and the nature of the code generation tasks we are evaluating. Unlike other compilation errors that might indicate fundamental syntactic misunderstandings by the AI (e.g., incorrect instruction syntax), SND errors can often be attributed to the context-dependent and fragmentary nature of the generated code snippets. Therefore, by addressing SND errors, the tool can evaluate the syntactic correctness of a code snippet to mitigate the incomplete context typically associated with AI-generated code.

For instance, consider, again, the previous code snippet: the first instruction compares the contents of the `EAX` and `EBX` registers and, if they are equal, the execution jumps to the `loop` label. However, this symbol has not been defined yet. To handle these cases, we analyze the assembler output to determine the missing symbol's name and include it in the source code file as a fictitious label. This label simply points to a jump operation to the previously defined exit address (i.e., `myExitAddr`). Indeed, the destination of the jump is not significant for the evaluation since we are only interested in checking the correctness of the instructions generated by the model. Therefore, after a SND error, *ACCA* further modifies the program as follows:

```
section .data
myExitAddr db 0x56
section .text
global _start
_start:
        cmp EAX, EBX
        je loop
        pop ECX
        jmp myExitAddr
loop:
        jmp myExitAddr
```

Once the source code file is modified accordingly, *ACCA* repeats the assembling process as before to check if the compilation ends with no



```
_start:
    cmp EAX, EBX
    je loop
    pop ECX
    jmp _exit
loop:
    inc EAX
    jmp _exit
```

Fig. 2. Example of a code snippet represented as a sequence of basic blocks.

errors or warnings. In this case, *ACCA* assigns the SYN score equal to 1 and continues the evaluation to check the semantic correctness.

When the compilation ends with errors different from the SND, such as an invalid combination of opcode and operands, expression syntax error, etc., then *ACCA* labels the model's prediction as syntactically incorrect (SYN = 0). Since a snippet syntactically incorrect is also semantically incorrect, then the evaluation process terminates, assigning the SEM score equal to zero. A source code file is generated and assembled for both the ground truth and the predicted code snippet.

3.3. Symbolic execution

To evaluate the semantic correctness, *ACCA* leverages the symbolic execution. To this aim, the method needs the program executable. If the assembling process ends correctly, the assembler outputs an object file, which is then fed to the linker to complete the process.

Since the same operation may be correctly implemented in different ways, a simple textual comparison with the reference is not enough to assess the semantic correctness of a program. We still consider the ground truth as the reference for the correct implementation of the intended code. However, we do not limit the comparison to a textual similarity, but we examine the *system state* at the end of the execution of both the reference and the generated code. Indeed, two programs that implement the same functionality using different operations can be considered semantically equivalent if they result in the same final system state. Since the final execution state depends on the inputs and initial state of the program, we need to compare the state produced by both programs for every possible combination of inputs and initial state.

Symbolic execution is a state-of-the-art solution for program analysis based on abstract execution. It consists in simulating the execution of a program providing symbolic values as its input instead of concrete ones. The result of this execution is a set of output values expressed as functions of the input symbols. *ACCA* uses symbolic execution to determine all the existing execution paths and all possible corresponding output system states. It then compares the set of output system states of the generated program with the set of output system states of the ground truth program: if they match, then the programs are semantically equivalent (SEM = 1), otherwise, the method classifies the model's prediction and the ground-truth as not semantically equivalent (SEM = 0).

To symbolically execute the programs, we use a *binary analysis platform* (BAP) that loads each executable and provides a complete abstract representation of the target architecture, CPU registers, memory address space, and stack. The program is conceived as a sequence of *basic blocks*, i.e., a straight-line code sequence with no branches, and the interconnections between the blocks represent the jump operations. An example is shown in Fig. 2: the program compares the contents of two registers and, if they are equal, the execution jumps to a specific address, otherwise, it jumps to the next instruction. Each possible branch is the entry point of a different basic block, which contains a sequence of operations that can be executed in order and one last instruction that causes the execution to move to another basic block.

ACCA begins the execution by initializing the abstract registers and flags to the same symbolic values; the method also sets the value of the stack pointer and initializes the memory to a blank state. Then, it executes the program by simulating each instruction step by step and keeping track of its state at each given step. During the execution of the programs, performing operations, such as arithmetic operations, comparisons, assignments, etc., that involve a variable (e.g., `if X> 10`) will yield an *execution tree*, i.e., a tree of operations formed by all the possible paths of the program, which encode all branch decisions taken up to a specific point. Execution trees are then translated into constraints, i.e., formulas that express a set of assumptions on the symbolic outputs of the execution. These constraints are finally solved by a *satisfiability modulo theories* (SMT) solver, which is typically employed in program verification to verify whether constraints can be satisfied by some assignment of concrete values to the program's symbolic arguments (Baldoni et al., 2018). As an example, consider the following simple constraints:

$$\begin{gathered} x > y \\ y > 2 \\ 10 > x \end{gathered}$$

The SMT solver treats them as assertions that must be satisfied with the valid values of symbolic variables. It, therefore, outputs a value that is consistent with all the constraints (e.g., x = 4, y = 3).

ACCA symbolically executes both the reference code and the predicted snippet: assuming that the program state at the beginning of the execution is identical, if the programs are semantically equivalent, then their state is also identical at the end of the execution. Therefore, to assess the semantic correctness of the generated code compared to the ground truth, the proposed method checks whether the states of the architecture are equal at the end of both executions. The program state includes:

- *state of the registers*, i.e., the contents of the abstract CPU registers;
- *state of the flags*, i.e., the abstract status register that keeps track of the CPU state, including the result of arithmetic operations (e.g., carry flag, overflow flag, etc.) and interruptions;
- *values on stack*, i.e., the contents of the memory area used to store temporary values;
- *path constraints*, i.e., the condition of input values, defined over the previous items, that leads to the corresponding final state.

Discerning between register states, flag states, stack values, and path constraints is integral in our method to accurately assess the state of execution and, by extension, the semantic equivalence of code snippets. The state of execution of a program at any point includes not just the data it operates on (registers and stack values) but also the conditions under which different operations are executed (flags and path constraints). Hence, each component provides unique insights into how the program behaves and interacts with the system's resources. For instance, register states give information about the current data being processed, flag states indicate the outcomes of previous operations (e.g., whether the last arithmetic operation resulted in a zero or a carry), stack values reveal the function call and return addresses along with local variables, and path constraints help understand the decision-making process within the program. Together, these components offer a complete picture of the program's execution path and its interactions with the system's memory and CPU. Moreover, each component's state is interrelated and influences the others during execution. For example, the outcome of a conditional branch (determined by flag states) will affect which path the program takes, subsequently influencing register and stack states. Assessing these components in isolation would lose the context of how the program transitions from one state to another, leading to a fragmented understanding of the program's behavior.

ACCA compares the state of each *leaf* node, i.e., the final *states* at the end of each path in the execution tree representing the program, of both executables. To compare the leaf nodes, the method constructs a set of lists for every final basic block of the two programs. Each set contains a list of register values, flag values, boolean constraints, and stack values. For example, a reference program whose execution tree ends with two basic blocks (leaf nodes) has two sets of lists. Each set contains all the values that represent the system state for that particular execution path. If the execution tree of the generated program has the same number of leaf nodes, then each list of the two sets is compared with each list of two sets of the reference program. If there is a correspondence between each leaf of the first program and each leaf of the second program, then they are semantically equivalent (SEM=1) and the evaluation process ends. Contrarily, if the leaf nodes of the two program execution trees do not match, then we conclude that the predicted code is not semantically equivalent to the reference snippet (SEM=0) and the process ends.

Since the total number of states can grow exponentially in the number of branches, one of the main challenges of symbolic execution is the path explosion problem (Baldoni et al., 2018). Indeed, keeping track of a large number of pending branches to be explored impacts both the running time and the space requirements of symbolic execution. The primary causes of path explosion are loops and function calls. One common method to deal with this problem is to bind the path exploration to a limited number of iterations. To handle programs whose symbolic execution does not terminate, we set a maximum number of execution steps. Since AI-generated code is typically concise and consists of a few assembly instructions, a correct program concludes its execution in a few execution steps. If it runs for more than `max_steps`, then the symbolic execution stops, and the generated program is classified as incorrect (SEM=0).

3.4. Implementation details

To assemble the programs and generate the executable files, we rely on the wide set of available open-source software. For the previous examples (on the IA-32), we used the Netwide Assembler (NASM) (NASM, 2024), an 80x86 and x86-64 assembler that supports a wide range of executable formats, including Linux, BSD, and Windows operating system formats. As a binary analysis platform, we use ANGR (Shoshitaishvili et al., 2016). ANGR provides support for a variety of CPU architectures, including ARM, MIPS, PPC, and x86 processors. It comprises a series of sub-components that implement the different steps necessary for the symbolic execution: to disassemble the executables and lift the binary code to an intermediate representation; to simulate the program state and execution, including registers and memory; and to solve the generated constraints, using the z3 (De Moura and Bjørner, 2008) SMT solver as a backend. We set the maximum number of execution steps `max_steps` to 100 to avoid infinite loops. Our implementation runs on both Linux and Windows OS. We publicly shared the implementation of *ACCA* on GitHub.[2]

4. Experimental setup

4.1. AI-code generation

To perform code generation and assess the tool on the AI-generated code, we adopted four state-of-the-art NMT models.

■ **Seq2Seq** is a model that maps an input of sequence to an output of sequence. Similar to the encoder–decoder architecture with attention mechanism (Bahdanau et al., 2015), we use a bi-directional LSTM as the encoder to transform an embedded intent sequence into a vector of hidden states with equal length. We implement the Seq2Seq model using *xnmt* (Neubig et al., 2018). We use an Adam optimizer (Kingma and Ba, 2015) with $\beta_1 = 0.9$ and $\beta_2 = 0.999$, while the learning rate α is set to 0.001. We set all the remaining hyper-parameters in a basic configuration: layer dimension = 512, layers = 1, epochs = 200, beam size = 5.

■ **CodeBERT** (Feng et al., 2020) is a large multi-layer bidirectional Transformer architecture (Vaswani et al., 2017) 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 (Liu et al., 2019b), with 12 attention heads, hidden layer dimension of 768, 12 encoder layers, and 514 for the size of position embeddings. We set the learning rate $\alpha = 0.00005$, batch size = 32, and beam size = 10.

■ **CodeT5+** (Wang et al., 2023) 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 (Raffel et al., 2020). It has an encoder–decoder architecture with 12 decoder layers, each with 12 attention heads and hidden layer dimension of 768, and 512 for the size of position embeddings. We set the learning rate $\alpha = 0.00005$, batch size = 16, and beam size = 10.

■ **PLBart** (Ahmad et al., 2021) is a multilingual encoder–decoder (sequence-to-sequence) model primarily intended for code-to-text, text-to-code, code-to-code tasks. The model is pre-trained on a large collection of Java and Python functions and natural language descriptions collected from GitHub and StackOverflow. We use the PLBart-large architecture with 12 encoder layers and 12 decoder layers, each with 16 attention heads. We set the learning rate $\alpha = 0.00005$, batch size = 16, and beam size = 10.

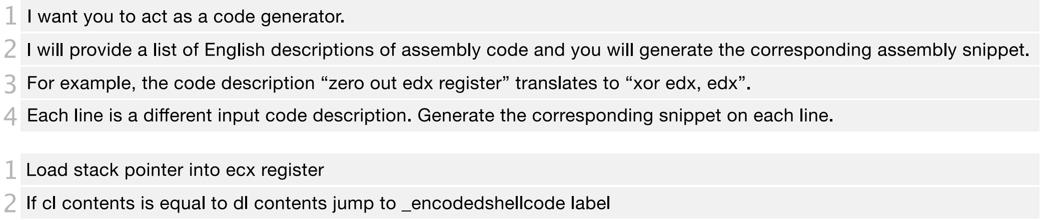
■ **ChatGPT** (OpenAI, 2023) is a popular large language model by OpenAI which extends the GPT model series. It can provide detailed responses given natural language prompts and examples of the desired output. We used the latest free version of the ChatGPT-3.5 model (OpenAI, 2024), which we queried via web interface and was trained on data up to April 2023. We prompted ChatGPT using the query format illustrated in Fig. 3. We first provided a detailed description of the required task, i.e., the generation of assembly code snippets starting from NL descriptions, and included an example of input and the desired output. Then, we provided a list of natural language code descriptions and asked ChatGPT to automatically generate the corresponding assembly instructions. We set the *temperature* parameter, which controls the degree of model randomness, at its default value of 1. By adopting the default configuration, we test the model's typical behavior, without forcing it to be more predictable or more creative (Buscemi, 2023).

To choose the models' hyperparameters that differ from the standard configuration (Yang et al., 2023; Liguori et al., 2021b) we followed the state-of-the-art in automatic exploit generation (see Section 7). We also evaluated the validation set every 320 steps.

We followed the best practices in the field of code generation by supporting NMT models with data processing operations. The data processing steps are usually performed both before translation (*pre-processing*), to train the NMT model and prepare the input data, and after translation (*post-processing*), to improve the quality and the readability of the code in output.

Our pre-processing operations start with the *stopwords filtering*, i.e., we remove a set of custom-compiled words (e.g., *the*, *each*, *onto*) from the intents to include only relevant data for machine translation. Next, we use a *tokenizer* to break the intents into chunks of text containing space-separated words (i.e., the *tokens*). To improve the performance of the machine translation (Li et al., 2018; Modrzejewski et al., 2020; Liguori et al., 2021b), we *standardize* the intents (i.e., we reduce the randomness of the NL descriptions) by using a *named entity tagger*, which returns a dictionary of *standardizable* tokens, such as

[2] https://github.com/dessertlab/ACCA



```
I want you to act as a code generator.
I will provide a list of English descriptions of assembly code and you will generate the corresponding assembly snippet.
For example, the code description “zero out edx register” translates to “xor edx, edx”.
Each line is a different input code description. Generate the corresponding snippet on each line.
```

```
Load stack pointer into ecx register
If cl contents is equal to dl contents jump to _encodedshellcode label
```

Fig. 3. Example of prompt used for code generation with ChatGPT.

Table 1
Dataset statistics.

Metric	NL intents	Assembly code snippets
Unique lines	5,740	3,316
Unique tokens	2855	1770
Avg. tokens per line	9.18	5.83

Table 2
Examples of assembly code with NL descriptions from our dataset.

Code snippet	English intent
`xor bl, 0xBB \n jz formatting \n mov cl, byte [esi]`	*Perform the xor between BL register and 0xBB and jump to the label formatting if the result is zero else move the current byte of the shellcode in the CL register.*
`xor ecx, ecx \n mul ecx`	*Zero out the EAX and ECX registers.*

specific values, label names, and parameters, extracted through regular expressions. We replace the selected tokens in every intent with “*var#*”, where # denotes a number from 0 to $|l|$, and $|l|$ is the number of tokens to standardize. Finally, the tokens are represented as real-valued vectors using *word embeddings*.

The pre-processed data is used to feed the NMT model. Once the model is trained, we perform the code generation from the NL intents. Therefore, when the model takes as inputs new intents, it generates the related code snippets based on its knowledge (*model's prediction*). As for the intents, also the code snippets predicted by the models are processed (*post-processing*) to improve the quality and readability of the code. Finally, the dictionary of standardizable tokens is used in the *de-standardization* process to replace all the “*var#*” with the corresponding values, names, and parameters.

In the data pre-processing phase, we employ the *nltk word tokenizer* (Bird, 2006) to tokenize the NL intents and the Python *tokenize* package (Python, 2024) for the code snippets. To facilitate the standardization of NL intents, we implement a named entity tagger using *spaCy*, an open-source, NL processing library written in Python and Cython (spaCy, 2024).

4.2. Dataset

To feed the models for the generation of security-oriented code, we extended the publicly available *Shellcode_IA32* dataset (Liguori et al., 2021a, 2022) for automatically generating *shellcodes* from NL descriptions. A shellcode is a list of machine code instructions to be loaded in a vulnerable application at runtime. The traditional way to develop shellcodes is to write them using the assembly language, and by using an assembler to turn them into *opcodes* (operation codes, i.e., a machine language instruction in binary format, to be decoded and executed by the CPU) (Foster, 2005; Megahed, 2018). Common objectives of shellcodes include spawning a system shell, killing or restarting other processes, causing a denial-of-service (e.g., a fork bomb), leaking secret data, etc.

The dataset consists of instructions in assembly language for *IA-32* collected from publicly available security exploits (Exploit-db, 2024; Shell-storm, 2022), manually annotated with detailed English descriptions. In total, it contains 3200 unique pairs of assembly code snippets/English intents. We further enriched the dataset with additional samples of shellcodes collected from publicly available security exploits, reaching 5900 unique pairs of assembly code snippets/English intents. To the best of our knowledge, the resulting dataset is the largest collection of shellcodes in assembly available to date.

Our dataset also includes 1374 intents (∼23% of the dataset) that generate multiple lines of assembly code, separated by the newline character *\n*. These multi-line snippets contain many different assembly instructions (e.g., whole functions) that can range between 2 and 18. For example, the copy of the ASCII string *“/bin//sh”* into a register is a typical operation to spawn a shell, which requires three distinct assembly instructions: push the hexadecimal values of the words *“/bin”* and *“//sh”* onto the stack register before moving the contents of the stack register into the destination register. Further examples of multi-line snippets include conditional jumps, tricks to zero out the registers without generating null bytes, etc. Table 2 shows two further examples of multi-line snippets with their natural language intents.

Table 1 summarizes the statistics of the dataset used in this work, including the unique examples of NL intents and assembly code snippets, the unique number of tokens, and the average number of tokens per snippet and intent. The dataset is publicly available on GitHub.[3]

To perform the experiments, we split the dataset into training, validation, and test sets using a common 80%/10%/10% ratio (Kim and MacKinnon, 2018; Mashhadi and Hemmati, 2021). Hence, they contain 4720, 590 and 590 samples, respectively.

4.3. Baseline assessment solutions

As a baseline for the evaluation, we used the following output similarity metrics, which are widely used to assess the performance of AI generators in many code generation tasks (Liguori et al., 2023), including the generation of assembly code for security contexts (Yang et al., 2022, 2023; Ruan et al., 2023; Liguori et al., 2021b, 2022):

- **Compilation Accuracy (CA)**. It indicates whether each code snippet produced by the model is compilable according to the syntax rules of the target language. CA value is either 1, when the snippet's syntax is correct, or 0 otherwise. To compute the *compilation accuracy*, we used the *Netwide Assembler* (NASM) assembler (NASM, 2024).
- **Bilingual Evaluation Understudy (BLEU) score** (Papineni et al., 2002). It measures the degree of n-gram overlapping between the string of each code snippet produced by the model and the reference. This metric also takes into account a *brevity penalty* to penalize predictions shorter than the references. BLEU value ranges between 0 and 1, with higher scores corresponding to a better quality of the prediction. Similar to previous studies, we use the BLEU-4 score (i.e., we set $n = 4$). We implemented BLEU score computation employing the `bleu_score` module contained in the open-source Python suite Natural Language Toolkit (NLTK) (NLTK, 2024).

[3] https://github.com/dessertlab/Shellcode_IA32

```
I want you to act as a human evaluator to assess the correctness of a list of code snippets.
I will provide a list of English descriptions of assembly code and the corresponding assembly snippet.
If the code is the semantically correct implementation of the English description, the score is ‘1’, otherwise it’s ‘0’.
Each line is a different pair of code description and code translation. Generate the corresponding score for each line.
E.g., considering the description “move 0x1 into lower byte of the eax register”, the snippet “mov al, 0x1” has a score equal to 1, while “mov bl, 0x1” has a score equal to 0.
```

```
“jump to _decode label if flag zero is equal to 0” - “jnz _decode”
“swap the values of esp and esi registers” - “xchg esp, esi”
```

Fig. 4. Example of prompt used for code correctness assessment with respect to the English description using ChatGPT.

```
I want you to act as a human evaluator to assess the correctness of a list of code snippets.
I will provide a list of reference implementations of assembly code and the corresponding hypothesis snippet.
If the hypothesis is semantically equivalent to the reference implementation, the score is ‘1’, otherwise it’s ‘0’.
Each line is a different pair of reference code and hypothesis code. Generate the corresponding score for each line.
E.g., considering the reference “xchg esp, esi”, the snippet “xchg esi, esp” has a score equal to 1, while the snippet “add esp, esi” has a score equal to 0.
```

```
“rol edx, 0x4” - “rol edx, 4”
“add eax, 1” - “inc eax”
```

Fig. 5. Example of prompt used for code correctness assessment with respect to the reference implementation using ChatGPT.

- **SacreBLEU** (Post, 2018). This is a different implementation of the BLEU score which differs from the traditional one because it uses different tokenization techniques. We used the implementation available on Hugging Face (Face, 2024)
- **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` (pylcs, 2024).

As a further baseline for the comparison, we again adopted the famous **ChatGPT** (OpenAI, 2023). As for the code generation process, we used the latest free version of the ChatGPT-3.5 model (OpenAI, 2024), which we queried via the web interface and was trained on data up to April 2023. We kept the default temperature value equal to 1 to test the model’s typical behavior.

For every code generated by the models, we asked ChatGPT to assess the code by assigning a value of 1 when it is correct, 0 otherwise. We performed two different evaluations:

- **ChatGPT-NL**: ChatGPT evaluates if the code generated by the models is the translation in assembly code of what is required in the natural language intents, similar to what a human evaluator does during the manual code review;
- **ChatGPT-GT**: ChatGPT evaluates if the code generated by the models is semantically equivalent to the ground truth used as a reference for the evaluation, similar to the assessment performed by output similarity metrics.

The prompt used to query the model in both settings are shown in Fig. 4 and 5, respectively. For the ChatGPT-NL evaluation, we provided a list of NL code descriptions and the corresponding code snippet generated by each model and asked ChatGPT to assign a score of 1 when the code is the semantically correct implementation of the description, 0 otherwise. For the ChatGPT-GT evaluation, we provided a list of reference implementations of the code snippets, i.e., the ground truth, and the corresponding code snippet generated by each model, and asked ChatGPT to assign a score of 1 when the generated code is semantically equivalent to the reference code, 0 otherwise.

Different from output similarity metrics and our tool, ChatGPT does not have a deterministic behavior. Therefore, we executed multiple assessments of code correctness with the OpenAI model by performing three repetitions. We found that in the 97% of the cases, there were no discrepancies among the different repetitions. In the few discrepancy cases, since the correctness score is 1 (correct) or 0 (incorrect), we adopted a majority voting mechanism (i.e., 2 out of 3).

4.4. Human evaluation

To ensure a robust and thorough assessment of both *ACCA* and baseline approaches in evaluating AI-generated code, we conducted a meticulous comparison against human evaluation, which serves as ground truth for our analysis.

In the human evaluation, a code prediction was deemed successful if it accurately translated the NL description into the assembly language and adhered to the established assembly programming language rules, warranting a score of 1. Any deviation resulted in a score of 0.

The manual evaluation was conducted by a diverse group consisting of 3 human evaluators, all with a computer science background and expertise in assembly language and cybersecurity. The group included individuals with varying degrees of professional experience and educational qualifications. In particular, 2 Ph.D. students with a master’s degree and a researcher with a Ph.D. in information technologies. The diversity and expertise of our evaluators ensured the reliability of our human evaluation process.

To fortify the integrity of our evaluation and minimize the potential for human error, the 3 human evaluators independently scrutinized each code snippet generated by the models. Any discrepancies that arose were attributed to human oversight and promptly rectified, culminating in unanimous consensus across all cases in the human evaluation, demonstrating a resounding 100% alignment.

Table 3
Code correctness assessment of *ACCA* with respect to the human evaluation.

	Seq2Seq	CodeBERT	CodeT5+	PLBart	ChatGPT	Average
Matching Rate	0.94	0.90	0.93	0.92	0.95	0.93
ACCA	0.60	0.68	0.69	0.57	0.71	0.65
Human Eval.	0.66	0.78	0.76	0.64	0.73	0.71

5. Experimental results

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

In the following, we assess how *ACCA* aligns with the human evaluation for the correctness of the code generated by models and compare the results with the baseline assessment solutions by performing both a quantitative (Section 5.1) and correlation analysis (Section 5.3). We also show representative examples to investigate discrepancy cases between *ACCA* and the human evaluation (Section 5.2). Finally, we assess the computational cost of running *ACCA* on the code predictions to provide an estimate of the average time needed to assess code correctness (Section 5.4).

5.1. Quantitative analysis

For all the models, we compared the average code correctness values computed by *ACCA* over all the examples of the test set with respect to the average semantic correctness assessed with the human evaluation and the alignment between the two solutions.

Table 3 shows the results.

The quantitative analysis is of limited use if we do not consider the matching rate between *ACCA* and the human evaluation, i.e., the percentage of code snippets that are equivalently considered correct or incorrect by both methods. For instance, both *ACCA* and the human evaluation can assess the code snippets as correct for 50% of the cases but still have 100% of discrepancy cases (i.e., the code snippets considered correct by *ACCA* and human evaluation are disjoint sets). The table highlights that the results provided by *ACCA* are very close to the human evaluation. Indeed, our method and human evaluation provide, on average, the same classification in the ∼ 93% of models' predictions (min 90%, max 95%). These results suggest that the proposed approach aligns well with human evaluations.

The table also shows the results of *ACCA* and the human evaluation for each model. The method classifies, on average, 65% of the generated code snippets by all the models as correct. On the other hand, according to our manual code review, we found that, on average, the 71% of the generated code snippets are semantically correct. Hence, although they are very close, *ACCA* underestimates the code correctness when compared to human evaluation. This is because human evaluators are not always able to account for all the factors that contribute to the program state. For example, the instructions `ADD EAX, 1` and `INC EAX` perform a similar operation, i.e., incrementing the `EAX` register, yet alter different flags, i.e., `INC` does not affect the Carry flag, hence they are not equivalent. Moreover, some snippets exceeded the constraint we imposed on the maximum number of symbolic execution steps to perform (i.e., *max_step*), hence leading to a correctness score equal to 0. However, it is important to notice that these cases represent a minimal fraction of the total examples (16 out of 2950 predictions, i.e., the 0.54%).

To better appreciate the evaluation provided by *ACCA*, we compared the results of the human evaluation with the results provided by the baseline solutions (described in Section 4.3). To this aim, we computed an *offset value*, i.e., the difference between the optimal value represented by the human evaluation and the value provided by different assessing solutions. The lower the offset, the closer the result is to the human evaluation. Table 4 shows the results.

The average offset of the output similarity metrics ranges between a minimum (best) value of 0.08 (for edit distance) and a maximum (worst) value of 0.25 (for BLEU-4). ChatGPT provided results similar to the best-performing output similarity metrics, with an average offset equal to 0.09 over all the models when the correctness of models' predictions is computed with respect to the NL intent (ChatGPT-NL), and equal to 0.12 when the predictions are compared to the ground truth (ChatGPT-GT). *ACCA* provided the lowest offset in 4 out of 5 models and, an average offset equal to 0.06, which results to be the lowest value, i.e., the code correctness computed by the proposed method is, on average, the closest to the human evaluation.

In our experiments and analysis, we observed that compilation errors encountered during the assembling process were exclusively attributable to the syntactic inaccuracies within the code snippets generated by the models. The occurrence of compilation errors due to syntactical inaccuracies underscores the challenges of AI code generation models in generating assembly code.

5.2. Qualitative analysis

We performed a manual inspection of the cases of discrepancy to examine when the method provides different results from the human evaluation. We have a discrepancy case when the method assesses the code as correct but the human evaluation does not, or when the method assesses the code as incorrect it is semantically correct according to the human evaluation.

As shown in Table 3, the method underestimates the performance of the models. In fact, an in-depth inspection of the results revealed that ∼ 99% of the discrepancy cases were due to examples classified as correct by human evaluation (value 1) but incorrect *ACCA* (value 0). To better discuss these discrepancy cases, Table 5 illustrates four representative examples of mismatch between *ACCA* and the human evaluation.

The first two rows of the table showcase two model predictions that are correctly labeled by human evaluation, but considered incorrect by our method. These misclassifications were due to the ambiguity of the code snippets since the same NL description can be expressed by semantically different code snippets. For instance, to zero out the stack (row # 1), a programmer can reset any register and then push the contents of the register (i.e., 0) into the stack register. Also, to move the contents of a register into a different one (row #2), a programmer can use the `mov` instruction to transfer a value from a source to a destination, or, equivalently, the `xchg` instruction, to swap the contents of the registers. Both the code snippets generated by the model accomplish what is required in the NL intent, but at the end of the symbolic execution, the state of the registers is different from the one obtained with the code in the ground truth (`EAX` is reset instead of `EDX` in the row # 1, while, in row # 2, `EAX` contains the value of `ESI`, instead of his original value). Therefore, *ACCA* provides the SEM score equal to zero, even if the snippets are semantically correct.

The last two rows of the table show examples of incorrect predictions that are wrongly classified as correct by the tool. As already remarked, these cases are very limited in numbers and represent situations in which, although the symbolic execution of predictions and ground-truth reference lead to the same state of the registers at the end of the execution, the model's prediction is not what is required by the NL description. For instance, in row # 3, the prediction contains what is described in the NL intents except for the label `L1`. The label

Table 4
Offset with respect to the human evaluation. The best performance (lower offset) is blue, while the worst performance (higher offset) is red.

Evaluation	Seq2Seq	CodeBERT	CodeT5+	PLBart	ChatGPT	Average
Compilat. Acc.	0.28	0.15	0.14	0.21	0.24	0.20
BLEU-4	0.22	0.20	0.29	0.24	0.29	0.25
SacreBLEU	0.15	0.05	0.07	0.09	0.09	0.09
Edit Distance	0.15	0.07	0.07	0.09	0.05	0.08
Exact Match	0.25	0.17	0.27	0.24	0.28	0.24
ChatGPT-NL	0.08	0.02	0.07	0.17	0.24	0.12
ChatGPT-GT	0.06	0.13	0.10	0.10	0.05	0.09
ACCA	0.06	0.10	0.07	0.07	0.02	0.06

Table 5
Examples of mismatches *ACCA* and human evaluation.

NL intent	Ground truth	Model's prediction	Human eval.	*ACCA*
Push zero into the stack	`xor edx, edx \n push edx`	`xor eax, eax \n push eax`	1	0
Save eax contents in esi contents	`mov esi, eax`	`xchg esi, eax`	1	0
In L1 jump short to esp	`L1: jmp short esp`	`jmp short esp`	0	1
Restore the top of the stack into the ecx register then decrement the ecx register and jump to the l1 label if the contents of the ecx register is not zero	`pop ecx \n loop l1`	`pop ecx \n dec ecx \n jmp l1`	0	1

does not affect the state of the registers during the execution of the code, but it will impact the behavior of the whole program (unless the label is never used by other instructions). Row #4, instead, showcases a more complex example in which the correct instruction `loop`, which decrements the value of the counter register `ECX` and jumps to the args (i.e., the `l1` label) if the counter is not zero, is replaced, in the model's prediction, by the decrement of the counter (`dec ecx` and an unconditional jump (`jmp`). In this case, although the instructions led to the same state of the registers because the counter was not zero after the decrement, the prediction is incorrect since the unconditional jump does not take into account the condition on the `ecx` register specified in the NL intent.

5.3. Correlation analysis

Additionally, we performed a statistical analysis by computing the correlation of *ACCA* with the human evaluation overall the code snippets of the test set (i.e., we considered the values on the single predictions).

To this aim, we computed the *Pearson* correlation coefficient r, which measures the strength of association (i.e., the linear relationship) between two variables in a correlation analysis and is defined as the covariance of the two variables divided by the product of their respective standard deviations (Pearson, 1895). The correlation coefficient is a unit-free value between −1 and 1, which represents *perfect* correlation, *negative*, and *positive*, respectively. Positive values indicate a positive correlation, i.e., the values of both variables tend to increase together, while negative values indicate a negative correlation, i.e., the values of one variable tend to increase when the values of the other variable decrease. A high value of the coefficient indicates that there is a strong correlation with the human evaluation. On the contrary, a small value indicates that there is a weak correlation with human evaluation. To provide context for the evaluation, we also computed the correlation coefficients between the baseline solutions and the human evaluation. Table 6 shows the results.

Confirming previous work (Liguori et al., 2023), the analysis shows that Edit Distance and Exact Match are the output similarity metrics most correlated to the semantic correctness for security-oriented code, with both Pearson's r coefficients equal to 0.69 and 0.59, respectively. The output similarity metric that is less correlated to human evaluation is the compilation accuracy (r = 0.43), showing that the syntactic correctness of the code is not highly correlated to its semantic correctness.

An important takeaway from our experiments is that, despite ChatGPT-based assessments providing results close to the human evaluation in the quantitative analysis (see 4), they have a correlation coefficient lower than the best-performing output similarity metric. Indeed, ChatGPT-GT has a correlation coefficient equal to 0.64, while ChatGPT-NL has a very poor correlation with human evaluation, resulting in the lowest value among all the baseline solutions (r = 0.34). This is a consequence of the high number of discrepancy cases between these solutions and the human evaluation, which is even more exacerbated in the ChatGPT-NL solution.

Finally, *ACCA* provides the highest correlation coefficient over all the five models, with an average value r = 0.84, hence being the only one to have a *very strong* correlation with human evaluation (Akoglu, 2018), surpassing by far all other metrics for all models.

5.4. Computational cost

We assessed the computational cost of *ACCA* in assessing the code correctness. Since the method skips the symbolic execution process for all generated snippets that are identical to the ground truth or that are not syntactically correct, we performed a thorough analysis considering three different cases: the evaluation of all the predictions in the test set (i.e., 590 code snippets), the evaluation of the subset of generated snippets that do not match the ground truth (i.e., "PR ≠ GT"), and evaluation of the subset of generated snippets that do not match the ground truth and are also syntactically correct (i.e., "PR ≠ GT & SYN = 1").

Table 7 presents a detailed analysis of the computational cost of *ACCA*. The table shows the average, median, and standard deviation of our method's cost, in terms of time (seconds) to assess a single code snippet, for each model. The last row shows the average, standard deviation, and median of our method's cost per snippet, across all models.

Regarding the evaluation of the entire test set, the average time required to assess a snippet is, as expected, lower than in other cases, with an average time equal to ∼ 0.20 s, a standard deviation of ∼ 1.28 s and a median value of 0.0. This is because predictions matching the ground truth are included in the analysis. In fact, in these cases, both syntactic assessment and symbolic execution for the semantic assessment are skipped.

Table 6
Pearson correlation coefficient *r* with human evaluation. For every model, the best values are blue, while the worst are red.

Evaluation	Seq2Seq	CodeBERT	CodeT5+	PLBart	ChatGPT	Average
Compilat. Acc.	0.36	0.44	0.59	0.52	0.22	0.43
BLEU-4	0.56	0.55	0.56	0.64	0.50	0.56
SacreBLEU	0.53	0.52	0.60	0.66	0.51	0.57
Edit Distance	0.67	0.61	0.75	0.76	0.64	0.69
Exact Match	0.60	0.67	0.55	0.61	0.55	0.59
ChatGPT-NL	0.42	0.42	0.37	0.44	0.07	0.34
ChatGPT-GT	0.67	0.61	0.70	0.68	0.56	0.64
ACCA	0.87	0.78	0.85	0.85	0.87	0.84

Table 7
Avg., median and std. dev. of the computational time (seconds) of *ACCA* per snippet on the whole data, the subset of predictions not matching the reference, and the subset that is also syntactically correct.

	All predictions		PR≠ GT		PR ≠ GT & SYN = 1	
	Avg & SD	Median	Avg & SD	Median	Avg & SD	Median
Seq2Seq	0.20 ± 1.22	0.00	0.33 ± 1.58	0.22	0.37 ± 1.67	0.22
CodeBERT	0.14 ± 1.12	0.00	0.29 ± 1.57	0.20	0.33 ± 1.68	0.21
CodeT5+	0.11 ± 0.97	0.00	0.22 ± 1.36	0.08	0.27 ± 1.51	0.20
PLBart	0.21 ± 1.56	0.00	0.35 ± 1.99	0.19	0.46 ± 2.30	0.21
ChatGPT	0.32 ± 1.51	0.00	0.67 ± 2.13	0.28	0.70 ± 2.17	0.29
All Models	0.20 ± 1.28	0.00	0.37 ± 1.73	0.21	0.43 ± 1.87	0.22

When we consider the subset of samples in which PR ≠ GT, i.e. when we exclude from our analysis the predictions that perfectly match the ground truth, the mean time per snippet increases on all five models. Indeed, in this case, *ACCA* requires on average ∼ 0.37 s to assess the code correctness, with a standard deviation of ∼ 1.73 s and a median value of ∼ 0.21 s.

In the last scenario, PR ≠ GT & SYN = 1, the value increases again because snippets that were labeled as syntactically incorrect during the assembling process (see Section 3.2) are excluded from the analysis. Therefore, this evaluation concerns only the predictions that went through the symbolic execution, i.e., all the evaluation steps of the proposed method. Interestingly, the mean time to assess the correctness of a code snippet produced by ChatGPT is slightly higher than other models. This is because the other models generate less syntactically correct code snippets than ChatGPT, which are therefore excluded from the analysis, i.e., 553, 550, 531, 499 for Seq2Seq, CodeBERT, CodeT5+ and PLBart, respectively, against the 577 by ChatGPT (out of 590, i.e., the examples in the test set). Overall, regardless of the model, *ACCA* needs on average 0.43 s to symbolically execute the code and perform the evaluation, with a standard deviation of ∼ 1.87 s and a median value of ∼ 0.22 s.

Another aspect that influences the total computational cost required for the analysis is the type of operation performed by the code snippet. For instance, while logical operations (e.g., `and`, `xor`, `not`) and instructions that handle register contents (e.g., `inc`, `dec`, `mov`) are fast computed (i.e., ∼ 0.25 s), instructions used to iterate over a variable, to compare two registers, or to perform conditional jumps (e.g., `cmp`, `loop`, `jns`) are less time-efficient. This is because arithmetical and logical operations are often simpler to implement because they involve basic bit-level manipulation. Contrarily, comparisons usually involve comparing values from different registers or memory locations, and conditional jumps depend on the result of these comparisons. This complexity can lead to longer execution times compared to simple logical operations. Table 8 presents two examples of outliers in the computational cost analysis, *ACCA*'s result, and the time needed for their evaluation.

Both ground truth snippets perform similar operations: a comparison between two registers or a numerical value, a conditional jump based on the previous result, and an unconditional jump to a specific label. While in row # 1 the code generated by the model has the same code complexity, in the second one the prediction exhibits lower complexity since the last jump is missing (i.e., `jmp while`). Both predictions are classified as incorrect by *ACCA* and take ∼ 24 s and ∼ 12 s, respectively.

While these outliers do increase the computational cost of our method, they represent a minimal fraction of all the generated samples, as in 17 out of 2950 predictions, i.e., only the 0.58%. Specifically, there are 3, 2, 1, 4 and 7 outliers for Seq2Seq, CodeBERT, CodeT5+, PLBart, and ChatGPT, respectively. The inclusion in our analyses of the outliers underscores the rigorousness of our methodology and highlights areas for future optimization in ACCA's execution efficiency.

Finally, to provide a context for the evaluation, we compared the computational cost of *ACCA* with the ones of the output similarity metrics, which are automatic and time-saving solutions, and the ChatGPT-based assessment solutions. Unsurprisingly, we found that the output similarity metrics provide an average estimate of similarity in a very limited amount of time (∼ 0.004 s on average per snippet), ranging from 0.001 s for the exact match to ∼ 0.01 s for the SacreBLEU metric. ChatGPT is also time-efficient, needing only ∼ 0.003 s to evaluate the correctness of the generated code with respect to the code description (ChatGPT-NL) and ∼ 0.001 for the comparison between predicted and ground truth snippets (ChatGPT-GT). As a result, the computational costs of *ACCA*, are higher than one of the baseline solutions since it depends on the non-negligible time needed by the binary execution.

However, it is important to stress again that the output similarity metrics provide only an estimate of the code similarity rather than evaluating the code's correctness. Moreover, although ChatGPT provides limited computational time for the assessment, it is not an automated solution as it requires a non-trivial manual effort, including detailed instructions and several iterations with the human operator. On the contrary, our method is fully automated as it does not require any human intervention for the assessment.

Finally, it is worth noticing that the computational times of *ACCA* are definitely lower than the average time required by human analysts to manually inspect the code, based on our experience. Indeed, since the human analyst needs to check both the NL description and the code snippet predicted by the models, in our experiments, the assessment of the semantic correctness required ∼ 20 s on average per code snippet.

6. Threats to validity

Relevance to Security. *ACCA* is designed as a tool to evaluate the semantic correctness of assembly code generated by AI models. Its primary function is to determine if AI-generated code behaves as intended

Table 8
Examples of code snippets with a high computational cost.

Ground truth	Model's prediction	*ACCA*	Time (s)
`cmp BYTE al, 2 \n je do_inject \n jmp while`	`cmp al, 2 \n jne while \n jmp do_inject`	0	23.92
`cmp ax, bx \n jne l3 \n jmp while`	`cmp ax, bx \n jne while`	0	11.98

when compared to a reference implementation. This involves assessing whether the generated code and the reference lead to the same final system state under the same initial conditions, thus ensuring semantic equivalence. While *ACCA* is applicable in various contexts where assembly code generation is involved, our study specifically targets security-oriented code due to the critical importance of correctness in this domain. The choice to focus on security-oriented code, exemplified by using the extended version of the Shellcode_IA32 dataset, stems from the challenges associated with generating assembly code for security purposes, e.g., exploit development. It is crucial to remark that *ACCA* does not perform a security assessment of the code itself; rather, it assesses the code's semantic correctness within security contexts. The tool does not identify vulnerabilities, exploits, or other security flaws in the generated code. Instead, it is a novel solution to automatically assess the semantics correctness of the code generated by models.

Relevance to the oracle problem in SE. We acknowledge that the Oracle problem is a central challenge in software engineering, particularly in the domain of automated testing (Giamattei et al., 2024) and code generation (Dinella et al., 2022; Hossain et al., 2023). The difficulty in establishing a definitive oracle that can accurately predict the correct output for any given input underlies many of the challenges in assessing the semantic correctness of AI-generated code. Our work indirectly addresses this problem by assessing semantic equivalence between AI-generated assembly code and a reference implementation. While we do not claim to solve the oracle problem, our approach provides a practical methodology for evaluating whether AI-generated code meets the intended specifications as represented by the reference code. This is especially relevant in contexts where the correctness of the code is critical, such as security-oriented applications.

Automation Claims and Reference Implementation. *ACCA*'s automation is predicated on the availability of a reference implementation, a common prerequisite in the field for assessing AI code generators. The reference implementations used for evaluation (i.e., the test set) are part of the same dataset that is used for training and validation purposes. Indeed, it is common practice in the field to randomly split the fine-tuning dataset into training, validation, and test sets, ensuring that the models are exposed to a diverse range of examples during the learning process. Hence, the reference for assessment is always available as long as there is a dataset to fine-tune models. In this work, we operate under the assumption that the reference implementation used for evaluation is reliable and accurately reflects the intended functionality as specified in the natural language descriptions. This assumption is consistent with standard practices in the field of code generation and evaluation, where output similarity metrics and other assessment methods rely on a trusted reference to determine the correctness of generated code. If we were to question the reliability of the reference implementations used for evaluation, it would logically extend to questioning the integrity of the entire dataset, including the portions used for training models. Such a scenario would imply that the models have been fine-tuned based on incorrect data, fundamentally challenging the premise of our study. However, this is beyond the scope of our work, which focuses on automating the assessment of the semantic correctness of AI-generated assembly code.

Assessment of AI code generators. Our tool is deterministic in its assessment process. The evaluation criteria and procedures ACCA employs are consistent across all evaluations, ensuring that any given code snippet's assessment outcome remains the same under identical conditions. Hence, performing multiple training of the models, given that the models' training is inherently stochastic, would only showcase the variability in AI model outputs, but this is not the focus of our work. The benefit of executing multiple models' training would be limited to a more comprehensive set of predictions to use for assessing the tool's performance. However, we remark that our work performs an extensive evaluation of *ACCA* by using the code generated by 5 distinct state-of-the-art models, for a total of 2950 predictions to evaluate (i.e., 590, the number of snippets of the test set, multiplied by the number of models, 5), of which 1369 are unique pairs reference-predictions. This diverse set of models ensures that our evaluation encompasses a broad spectrum of AI-generated code, thereby enhancing the generalizability of our results. Finally, we highlight the computational resources to run models and the considerable time to perform manual analysis on all code snippets generated by the models, which would make our experiments prohibitive with multiple repetitions.

Dataset: This work targets the assessment of AI-based solutions in the generation of offensive code for software exploits. The dataset used for our experiments fits perfectly with the scope of this work since it is the largest collection of offensive code available to date for code generation. This manually curated dataset contains high-quality and detailed descriptions of code, that are often not available in larger corpora for code generation. Indeed, the dataset provides NL descriptions both at the block and statement levels that are closer to the descriptions needed by the models for complex programming tasks.

Generalization and Offensive Code Generation: Although offensive code is different from general-purpose ones in terms of programming languages and characteristics, the proposed method can be applied to different assembly code generation scenarios. The decision to focus on offensive code generation is driven by the critical importance of code correctness assessment in this specific field. The use of models to generate offensive code, a research topic that is gaining increasing interest in software security (Yang et al., 2023; Botacin, 2023; Natella et al., 2024), poses unique challenges, requiring code generated by models to be properly evaluated. By focusing on exploits, the study addresses a specific and high-impact application of code generation, providing insights into the challenges of assessing the correctness of models in scenarios with stringent requirements.

7. Related work

Automatic Program Evaluation. Traditionally, the problem of automatic program assessment has been largely addressed for educational purposes, aiming to assist educators in the student work evaluation process. Insa and Silva (2018, 2015) presented a tool to assess Java programs by automatically validating different properties, such as the use of interfaces and class hierarchy. Romli et al. (2014) developed *FaSt-Gen*, a framework of test data generation to cover both the functional and structural testing of programs for automatic assessment. Li et al. (2016) leveraged random testing and dynamic symbolic execution (DSE), i.e., a software testing technique that simulates the execution of a program by providing symbolic inputs instead of concrete values. They generated test inputs and ran programs on these test inputs to compute values of behavioral similarity. Arifi et al. (2016) proposed a method to grade C programs in an educational context automatically. They measured the similarity between programs by comparing the outputs of their symbolic execution. *CASM-VERIFY* (Lim and Nagarakatte, 2019) is a tool to automatically check the equivalence of optimized assembly implementations of cryptographic algorithms. The tool decomposes the equivalence checking problem into several small sub-problems using a combination of concrete and symbolic evaluation. The use of symbolic execution to evaluate code similarity has been explored also for security applications. Luo et al. (2017) introduced a binary code similarity comparison method for code theft detection. Gao et al. (2008) presented *BinHunt*, a method to identify the semantic differences between an executable and its patched version, revealing the vulnerability that the patch eliminates. Scalabrino et al. focused on automatically assessing the understandability of code snippets by combining 121 existing and new metrics, including code-related, documentation-related, and developer-related metrics. They concluded, however, that these metrics are not suited to capture the complexity of code in practical applications. Ullah and Oh (2022) proposed a neural network-based solution to perform *binary diffing* on x86 architecture binaries, i.e., the process of discovering the differences and similarities in functionality between two binary programs. Leveraging symbolic execution to check semantic equivalence has been proposed and used in the area of compiler validation since compilers should preserve semantics. For example, Béra et al. (2016) applied symbolic execution on the bytecode produced by the compilation with optimizations and that produced by the compilation without optimizations, in order to detect compiler bugs. Hawblitzel et al. (2013) detected compiler bugs by comparing assembly language outputs through symbolic execution. These solutions, however, require entire programs as input and do not work on portions of code (i.e., code snippets), which is often the case with AI-generated code since NMT is still far from generating entire complex functions, particularly in the context of offensive security.

Programming Language Code-oriented Metrics. In addition to state-of-the-art textual similarity metrics used as a baseline for the evaluation (see Section 4.3), recent work proposed a set of novel code-oriented metrics, i.e., metrics created ad-hoc for specific programming languages, to automatically assess the correctness of the generated code. Examples of code-oriented metrics are CodeBLEU (Ren et al., 2020) and RUBY (Tran et al., 2019), which were introduced to evaluate programs written in Java and C#. However, these solutions rely on deeper program analysis, including syntax and dataflow match, and require compilable code to function, which prevents them from being language-agnostic. Indeed, none of the available code-oriented metrics is designed for low-level programming languages such as assembly. Previous work on code generation also resorted to *functional correctness* to evaluate the quality of the generated programs, where a code sample is considered correct if it passes a set of unit tests. Kulal et al. (2019) used an evaluation metric based on functional correctness to address the problem of producing correct code starting from pseudocode. They generated k code samples per problem and assessed the ratio of problems in which any of the k samples passed the set of unit tests. Chen et al. (2021) proposed `pass@k`, an unbiased and numerically stable implementation of this metric. They generated $n \geq k$ samples per task ($n = 200$ and $k \leq 100$), counted the number of correct samples $c \leq n$ that pass unit tests, and calculated an unbiased estimator to benchmark their models in the generation of Python programs from docstrings. To estimate the functional correctness of a program, however, a set of unit tests needs to be manually constructed. This requires a significant effort that is often unfeasible for large amounts of generated code.

AI Generative for Security. Automatic exploit generation (AEG) research challenge consists of automatically generating working exploits (Avgerinos et al., 2014). This task requires technical skills and expertise in low-level languages to gain full control of the memory layout and CPU registers and attack low-level mechanisms (e.g., heap metadata and stack return addresses). Given their recent advances, AI-code generators have become a new and attractive solution to help developers and security testers in this challenging task. Liguori et al. (2022) released a dataset containing NL descriptions and assembly code extracted from software exploits. The authors 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 security-oriented code used to obfuscate software exploits from systems' protection mechanisms (Liguori et al., 2021b). Yang et al. (2022) proposed a data-driven approach to software exploit generation and summarization as a dual learning problem. The approach exploits the symmetric structure between the two tasks via dual learning and uses a shallow Transformer model to learn them simultaneously. Yang et al. (2023) proposed a novel template-augmented exploit code generation approach. The approach uses a rule-based template parser to generate augmented NL descriptions and uses a semantic attention layer to extract and calculate each layer's representational information. The authors show that the proposed approach outperforms the state-of-the-art baselines from the previous studies of automatic code generation. Ruan et al. (2023) designed an approach for software exploit generation based on prompt tuning. The solution aids the generation process by inserting trainable prompt tokens into the original input to simulate the pre-training stage of the model to take advantage of its prior knowledge distribution. Xu et al. (2023) introduced an artifact-assisted AEG solution that automatically summarizes the exploit patterns from artifacts of known exploits and uses them to guide the generation of new exploits. The authors implemented AutoPwn, an AEG system that automates the generation of heap exploits for Capture-The-Flag *pwn* competitions. Recent work also explored the role of GPT-based models, including ChatGPT and Auto-GPT, in the offensive security domain. Botacin (2023) found that, by using these models, attackers can both create and deobfuscate malware by splitting the implementation of malicious behaviors into smaller building blocks. Pa Pa et al. (2023) and Gupta et al. (2023) proved the feasibility of generating malware and attack tools through the use of reverse psychology and *jailbreak prompts*, i.e., maliciously crafted prompts able to bypass the ethical and privacy safeguards for abuse prevention of AI code generators like ChatGPT. Gupta et al. (2023) also examined the use of AI code generators to improve security measures, including cyber defense automation, reporting, threat intelligence, secure code generation and detection, attack identification, and malware detection. Natella et al. (2024) built a security-oriented evaluation benchmark to discuss potential use cases of AI code generators for offensive security. These use cases encompass attack surface analysis, OSINT, vulnerability exploitation, and post exploitation activities. The authors concluded that cybersecurity professionals must embrace AI code generators to prevent attacks more efficiently. All previous work uses state-of-the-art output similarity metrics or performs manual analysis to assess the correctness of AI-generated code/programs.

Our work is complementary to previous ones. Indeed, this work proposes a method that leverages symbolic execution to automatically assess the correctness of low-level code snippets used in security contexts. Since the method does not necessarily require full programs in inputs, it is suitable for assessing AI-generated code because they are often incomplete or non-compilable programs. Moreover, the proposed method does not require any human intervention, yet, differently from traditional text similarity metrics, which are commonly used to assess the performance of AI-generated code, its accuracy is comparable to human evaluation.

8. Conclusion

In this paper, we addressed the automatic correctness of the code generated by AI code generators. We proposed a fully automated method, named *ACCA*, that uses symbolic execution to assess the correctness of security-oriented code without any human effort.

We used our method to evaluate the performance of four state-of-the-art code generators in the generation of offensive assembly from NL descriptions and compared the results with the human evaluation and different baseline solutions, including state-of-the-art output similarity metrics and the well-known ChatGPT.

Our experiments showed that *ACCA* provides results almost equal and is the most correlated assessment solution to human evaluation, which is considered the golden standard in the field. Moreover, the analysis of the computational cost revealed that the time to assess every code snippet is ~ 0.17 s on average, which is lower than the average time required by human analysts to manually inspect the code, based on our experience.

In future work, integrating *ACCA* with advanced static code analysis and binary matching approaches could significantly enhance its assessment capabilities. For instance, static code analysis could provide additional insights into potential vulnerabilities or inefficiencies in the generated code (Sui and Xue, 2016), while binary matching could extend *ACCA*'s applicability to compiled binaries (Gui et al., 2022), offering a more holistic view of code correctness and security.

Future work also includes supporting the automatic correctness assessment for other programming languages. We are actively exploring the extension of *ACCA* to evaluate code in high-level programming languages such as Python. This expansion will address the growing need for semantic correctness assessment in diverse coding environments, further enhancing *ACCA*'s applicability and impact in the assessment of code generation models.

CRediT authorship contribution statement

Domenico Cotroneo: Conceptualization, Funding acquisition, Project administration, Resources, Supervision, Writing – review & editing. **Alessio Foggia:** Validation, Visualization, Investigation, Data curation, Formal analysis. **Cristina Improta:** Writing – review & editing, Conceptualization, Formal analysis, Methodology, Software, Validation, Visualization, Writing – original draft, Investigation. **Pietro Liguori:** Conceptualization, Investigation, Methodology, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Roberto Natella:** Conceptualization, Funding acquisition, Investigation, Methodology, Resources, Supervision, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The source code required to reproduce the experiments is available to download from https://github.com/dessertlab/ACCA. The raw data required to reproduce the above findings are available to download from https://github.com/dessertlab/Shellcode_IA32.

Acknowledgments

This work has been partially supported by the University of Naples Federico II in the frame of the MUR PRIN 2022 program, project FLEGREA, CUP E53D23007950001 (https://flegrea.github.io). We are grateful to our former student Emiliano Fiorenza for his help in the early stages of this work.

References

Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K., 2021. Unified pre-training for program understanding and generation. In: Toutanova, K., Rumshisky, A., Zettlemoyer, L., Hakkani-Tür, D., Beltagy, I., Bethard, S., Cotterell, R., Chakraborty, T., Zhou, Y. (Eds.), Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. NAACL-HLT 2021, Online, June 6-11, 2021, Association for Computational Linguistics, pp. 2655–2668. http://dx.doi.org/10.18653/V1/2021.NAACL-MAIN.211.

Akoglu, H., 2018. User's guide to correlation coefficients. Turk. J. Emerg. Med. 18 (3), 91–93.

Araujo, F.S., Rizzo Vincenzi, A.M., 2020. How far are we from testing a program in a completely automated way, considering the mutation testing criterion at unit level? In: Proceedings of the XIX Brazilian Symposium on Software Quality. pp. 1–9.

Arifi, S.M., Zahi, A., Benabbou, R., 2016. Semantic similarity based evaluation for c programs through the use of symbolic execution. In: 2016 IEEE Global Engineering Education Conference. EDUCON, pp. 826–833. http://dx.doi.org/10.1109/EDUCON.2016.7474648.

Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M., Brumley, D., 2014. Automatic exploit generation. Commun. ACM 57 (2), 74–84.

Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J.D., Penix, J., 2008. Using static analysis to find bugs. IEEE Softw. 25 (5), 22–29.

Bahdanau, D., Cho, K., Bengio, Y., 2015. Neural machine translation by jointly learning to align and translate. In: Bengio, Y., LeCun, Y. (Eds.), 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.

Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I., 2018. A survey of symbolic execution techniques. ACM Comput. Surv. 51 (3), 1–39.

Béra, C., Miranda, E., Denker, M., Ducasse, S., 2016. Practical validation of bytecode to bytecode JIT compiler dynamic deoptimization. J. Object Technol. 15 (2), 1.

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D., 2010. A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM 53 (2), 66–75.

Bird, S., 2006. NLTK: the natural language toolkit. In: Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions. pp. 69–72.

Botacin, M., 2023. GPThreats-3: Is automatic malware generation a threat? In: 2023 IEEE Security and Privacy Workshops. SPW, IEEE, pp. 238–254.

Buscemi, A., 2023. A comparative study of code generation using ChatGPT 3.5 across 10 programming languages. arXiv preprint arXiv:2308.04477.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al., 2021. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

De Moura, L., Bjørner, N., 2008. Z3: An efficient SMT solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. In: TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, pp. 337–340.

Dinella, E., Ryan, G., Mytkowicz, T., Lahiri, S.K., 2022. Toga: A neural method for test oracle generation. In: Proceedings of the 44th International Conference on Software Engineering. pp. 2130–2141.

Evtikhiev, M., Bogomolov, E., Sokolov, Y., Bryksin, T., 2023. Out of the bleu: how should we assess quality of the code generation models? J. Syst. Softw. 203, 111741.

Exploit-db, 2024. Exploit database shellcodes. https://www.exploit-db.com/shellcodes?platform=linux_x86/.

Face, H., 2024. SacreBLEU. https://huggingface.co/spaces/evaluate-metric/sacrebleu.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. 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, In: Findings of ACL, vol. EMNLP 2020, Association for Computational Linguistics, pp. 1536–1547. http://dx.doi.org/10.18653/v1/2020.findings-emnlp.139.

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

Gao, D., Reiter, M.K., Song, D., 2008. Binhunt: Automatically finding semantic differences in binary programs. In: Information and Communications Security: 10th International Conference, ICICS 2008 Birmingham, UK, October 20-22, 2008 Proceedings 10. Springer, pp. 238–255.

Giamattei, L., Guerriero, A., Pietrantuono, R., Russo, S., 2024. Automated functional and robustness testing of microservice architectures. J. Syst. Softw. 207, 111857. http://dx.doi.org/10.1016/J.JSS.2023.111857.

Gui, Y., Wan, Y., Zhang, H., Huang, H., Sui, Y., Xu, G., Shao, Z., Jin, H., 2022. Cross-language binary-source code matching with intermediate representations. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 601–612.

Gupta, M., Akiri, C., Aryal, K., Parker, E., Praharaj, L., 2023. From ChatGPT to ThreatGPT: Impact of generative AI in cybersecurity and privacy. IEEE Access.

Hawblitzel, C., Lahiri, S.K., Pawar, K., Hashmi, H., Gokbulut, S., Fernando, L., Detlefs, D., Wadsworth, S., 2013. Will you still compile me tomorrow? static cross-version compiler validation. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 191–201.

Hossain, S.B., Filieri, A., Dwyer, M.B., Elbaum, S., Visser, W., 2023. Neural-based test oracle generation: A large-scale evaluation and lessons learned. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 120–132.

Insa, D., Silva, J., 2015. Semi-automatic assessment of unrestrained Java code: a library, a DSL, and a workbench to assess exams and exercises. In: Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education. pp. 39–44.

Insa, D., Silva, J., 2018. Automatic assessment of Java code. Comput. Lang. Syst. Struct. 53, 59–72.

Kim, D., MacKinnon, T., 2018. Artificial intelligence in fracture detection: transfer learning from deep convolutional neural networks. Clin. Radiol. 73 (5), 439–445.

Kingma, D.P., Ba, J., 2015. Adam: A method for stochastic optimization. In: Bengio, Y., LeCun, Y. (Eds.), 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.

Kononenko, O., Baysal, O., Godfrey, M.W., 2016. Code review quality: How developers see it. In: Proceedings of the 38th International Conference on Software Engineering. pp. 1028–1038.

Kulal, S., Pasupat, P., Chandra, K., Lee, M., Padon, O., Aiken, A., Liang, P.S., 2019. Spoc: Search-based pseudocode to code. Adv. Neural Inf. Process. Syst. 32.

Li, Z., Wang, X., Aw, A., Chng, E.S., Li, H., 2018. Named-entity tagging and domain adaptation for better customized translation. In: Proceedings of the Seventh Named Entities Workshop. pp. 41–46.

Li, S., Xiao, X., Bassett, B., Xie, T., Tillmann, N., 2016. Measuring code behavioral similarity for programming and software engineering education. In: Proceedings of the 38th International Conference on Software Engineering Companion. ICSE '16, Association for Computing Machinery, New York, NY, USA, pp. 501–510. http://dx.doi.org/10.1145/2889160.2889204.

Liguori, P., Al-Hossami, E., Cotroneo, D., Natella, R., Cukic, B., Shaikh, S., 2021a. Shellcode_IA32: A dataset for automatic shellcode generation. In: Proceedings of the 1st Workshop on Natural Language Processing for Programming. NLP4Prog 2021, Association for Computational Linguistics, Online, pp. 58–64. http://dx.doi.org/10.18653/v1/2021.nlp4prog-1.7, URL https://aclanthology.org/2021.nlp4prog-1.7.

Liguori, P., Al-Hossami, E., Cotroneo, D., Natella, R., Cukic, B., Shaikh, S., 2022. Can we generate shellcodes via natural language? An empirical study. Autom. Softw. Eng. 29 (1), 1–34.

Liguori, P., Al-Hossami, E., Orbinato, V., Natella, R., Shaikh, S., Cotroneo, D., Cukic, B., 2021b. EVIL: exploiting software via natural language. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 321–332.

Liguori, P., Improta, C., Natella, R., Cukic, B., Cotroneo, D., 2023. Who evaluates the evaluators? On automatic metrics for assessing AI-based offensive code generators. Expert Syst. Appl. 225, 120073. http://dx.doi.org/10.1016/j.eswa.2023.120073.

Lim, J.P., Nagarakatte, S., 2019. Automatic equivalence checking for assembly implementations of cryptography libraries. In: 2019 IEEE/ACM International Symposium on Code Generation and Optimization. CGO, IEEE, pp. 37–49.

Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F., 2019a. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 1–12.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V., 2019b. RoBERTa: A robustly optimized BERT pretraining approach. CoRR abs/1907.11692. URL http://arxiv.org/abs/1907.11692.

Lukasczyk, S., Kroiß, F., Fraser, G., 2023. An empirical study of automated unit test generation for Python. Empir. Softw. Eng. 28 (2), 36.

Luo, L., Ming, J., Wu, D., Liu, P., Zhu, S., 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. IEEE Trans. Softw. Eng. 43 (12), 1157–1177. http://dx.doi.org/10.1109/TSE.2017.2655046.

Mashhadi, E., Hemmati, H., 2021. Applying codebert for automated program repair of java simple bugs. In: 18th IEEE/ACM International Conference on Mining Software Repositories. MSR 2021, Madrid, Spain, May 17-19, 2021, IEEE, pp. 505–509. http://dx.doi.org/10.1109/MSR52588.2021.00063.

Massarelli, L., Di Luna, G.A., Petroni, F., Querzoni, L., Baldoni, R., 2021. Function representations for binary similarity. IEEE Trans. Dependable Secure Comput. 19 (4), 2259–2273.

Megahed, H., 2018. Penetration Testing with Shellcode: Detect, Exploit, and Secure Network-Level and Operating System Vulnerabilities. Packt Publishing.

Modrzejewski, M., Exel, M., Buschbeck, B., Ha, T.-L., Waibel, A., 2020. 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.

NASM, 2024. Netwide assembler (NASM). https://www.nasm.us.

Natella, R., Liguori, P., Improta, C., Cukic, B., Cotroneo, D., 2024. AI code generators for security: Friend or foe? IEEE Secur. Privacy.

Neubig, G., Sperber, M., Wang, X., Felix, M., Matthews, A., Padmanabhan, S., Qi, Y., Sachan, D.S., Arthur, P., Godard, P., Hewitt, J., Riad, R., Wang, L., 2018. XNMT: the extensible neural machine translation toolkit. In: Cherry, C., Neubig, G. (Eds.), Proceedings of the 13th Conference of the Association for Machine Translation in the Americas, AMTA 2018, Boston, MA, USA, March 17-21, 2018 - Volume 1: Research Papers. Association for Machine Translation in the Americas, pp. 185–192, URL https://aclanthology.org/W18-1818/.

NLTK, 2024. Natural language toolkit (NLTK), bleu_score module. https://www.nltk.org/api/nltk.translate.bleu_score.html.

OpenAI, 2023. ChatGPT. https://chat.openai.com/chat. (Accessed 10 January 2023).

OpenAI, 2024. ChatGPT release notes. https://help.openai.com/en/articles/6825453-chatgpt-release-notes. (Accessed 3 January 2024).

Pa Pa, Y.M., Tanizaki, S., Kou, T., Van Eeten, M., Yoshioka, K., Matsumoto, T., 2023. 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.

Papineni, K., Roukos, S., Ward, T., Zhu, W., 2002. 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, ACL, pp. 311–318. http://dx.doi.org/10.3115/1073083.1073135.

Parvez, M.R., Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K., 2021. Retrieval augmented code generation and summarization. In: Moens, M., Huang, X., Specia, L., Yih, S.W. (Eds.), Findings of the Association for Computational Linguistics. EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021, Association for Computational Linguistics, pp. 2719–2734. http://dx.doi.org/10.18653/v1/2021.findings-emnlp.232.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R., 2022. Asleep at the keyboard? Assessing the security of GitHub copilot's code contributions. In: 43rd IEEE Symposium on Security and Privacy. SP 2022, San Francisco, CA, USA, May 22-26, 2022, IEEE, pp. 754–768. http://dx.doi.org/10.1109/SP46214.2022.9833571.

Pearson, K., 1895. Notes on regression and inheritance in the case of two parents. Proc. R. Soc. Lond. 58, 240–242, K Pearson.

Post, M., 2018. A call for clarity in reporting BLEU scores. In: Proceedings of the Third Conference on Machine Translation: Research Papers. Association for Computational Linguistics, Belgium, Brussels, pp. 186–191.

pylcs, 2024. Python library pylcs. https://pypi.org/project/pylcs/.

Pylint, 2024. Pylint. https://www.pylint.org.

Python, 2024. tokenize. https://docs.python.org/3/library/tokenize.html.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J., 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. J. Mach. Learn. Res. 21, 140:1–140:67.

Reiter, E., Belz, A., 2009. An investigation into the validity of some metrics for automatically evaluating natural language generation systems. Comput. Linguist. 35 (4), 529–558.

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., Ma, S., 2020. CodeBLEU: a method for automatic evaluation of code synthesis. CoRR abs/2009.10297. arXiv:2009.10297.

Romli, R., Sulaiman, S., Zamli, K.Z., 2014. Test data generation framework for automatic programming assessment. In: 2014 8th. Malaysian Software Engineering Conference. MySEC, pp. 84–89. http://dx.doi.org/10.1109/MySec.2014.6985993.

Ruan, X., Yu, Y., Ma, W., Cai, B., 2023. Prompt learning for developing software exploits. In: Proceedings of the 14th Asia-Pacific Symposium on Internetware. pp. 154–164.

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

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al., 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 138–157.

Shterionov, D., Superbo, R., Nagle, P., Casanellas, L., O'dowd, T., Way, A., 2018. Human versus automatic quality evaluation of NMT and PBSMT. Mach. Transl. 32 (3), 217–235.

Siddiq, M.L., Majumder, S.H., Mim, M.R., Jajodia, S., Santos, J.C., 2022. An empirical study of code smells in transformer-based code generation techniques. In: 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation. SCAM, IEEE, pp. 71–82.

spaCy, 2024. Industrial-strength natural language processing. https://spacy.io/.

Sui, Y., Xue, J., 2016. SVF: interprocedural static value-flow analysis in LLVM. In: Proceedings of the 25th International Conference on Compiler Construction. pp. 265–266.

Tony, C., Mutas, M., Ferreyra, N.E.D., Scandariato, R., 2023. LLMSecEval: A dataset of natural language prompts for security evaluations. In: 20th IEEE/ACM International Conference on Mining Software Repositories. MSR 2023, Melbourne, Australia, May 15-16, 2023, IEEE, pp. 588–592. http://dx.doi.org/10.1109/MSR59073.2023.00084.

Tran, N.M., Tran, H., Nguyen, S., Nguyen, H., Nguyen, T.N., 2019. Does BLEU score work for code migration? In: Guéhéneuc, Y., Khomh, F., Sarro, F. (Eds.), Proceedings of the 27th International Conference on Program Comprehension. ICPC 2019, Montreal, QC, Canada, May 25-31, 2019, IEEE / ACM, pp. 165–176. http://dx.doi.org/10.1109/ICPC.2019.00034.

Ullah, S., Oh, H., 2021. BinDiff NN: Learning distributed representation of assembly for robust binary diffing against semantic differences. IEEE Trans. Softw. Eng. 48 (9), 3442–3466.

Ullah, S., Oh, H., 2022. BinDiffNN: Learning distributed representation of assembly for robust binary diffing against semantic differences. IEEE Trans. Softw. Eng. 48 (9), 3442–3466. http://dx.doi.org/10.1109/TSE.2021.3093926.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. In: Advances in Neural Information Processing Systems. pp. 5998–6008.

Wang, Y., Le, H., Gotmare, A., Bui, N.D.Q., Li, J., Hoi, S.C.H., 2023. CodeT5+: Open code large language models for code understanding and generation. In: Bouamor, H., Pino, J., Bali, K. (Eds.), Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. EMNLP 2023, Singapore, December 6-10, 2023, Association for Computational Linguistics, pp. 1069–1088.

Wu, B., Zhang, X., 2022. Contrastive learning for multi-modal automatic code review. In: Peng, R., Pantoja, C.E., Kamthan, P. (Eds.), The 34th International Conference on Software Engineering and Knowledge Engineering. SEKE 2022, KSIR Virtual Conference Center, USA, July 1 - July 10, 2022, KSI Research Inc., pp. 54–59. http://dx.doi.org/10.18293/SEKE2022-022.

Xu, D., Chen, K., Lin, M., Lin, C., Wang, X., 2023. AutoPwn: Artifact-assisted heap exploit generation for CTF PWN competitions. IEEE Trans. Inf. Forensics Secur..

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

Yang, G., Zhou, Y., Chen, X., Zhang, X., Han, T., Chen, T., 2023. ExploitGen: Template-augmented exploit code generation based on CodeBERT. J. Syst. Softw. 197, 111577.

Domenico Cotroneo is a Professor at the University of Naples Federico II, Italy. His current research interests include software reliability and security, field failure data analysis, and software fault injection. Domenico Cotroneo received a Ph.D. degree from the Department of Computer Science and System Engineering of the University of Naples Federico II. He is chair of the IEEE technical community on dependable computing and fault tolerance, he is an IEEE senior member and elected member of IFIP WG 10.4 on dependable computing and fault tolerance.

Alessio Foggia is a research fellow at the University of Naples Federico II, Italy. His research interests include offensive security, AI code generation, and the security of machine learning systems. Alessio Foggia received a M.Sc. degree in Computer Engineering at the University of Naples Federico II.

Cristina Improta is a Ph.D. student at the University of Naples Federico II, Italy. Her current research interests include offensive security, AI code generation, and the security of machine learning systems. Cristina Improta received a M.Sc. degree in Computer Engineering at the University of Naples Federico II.

Pietro Liguori is an Assistant Professor at the University of Naples Federico II, Italy. His current research interests include automatic exploit generation and the robustness and security of AI code generators. Pietro Liguori received a Ph.D. degree in Information Technology and Electrical Engineering from the University of Naples Federico II.

Roberto Natella is an Associate Professor at the University of Naples Federico II, Italy. His current research interests are in the field of software security and dependability. The main recurring theme of his research activity is the experimental injection of faults, attacks, and stressful conditions. Roberto Natella received a Ph.D. degree in Information Technology and Electrical Engineering from the University of Naples Federico II. He is an IEEE senior member.