IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 11 September 2024, accepted 22 October 2024, date of publication 30 October 2024, date of current version 11 November 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3488204

RESEARCH ARTICLE

Adversarial Attacks Against Binary Similarity Systems

GIANLUCA CAPOZZI, DANIELE CONO D'ELIA, GIUSEPPE ANTONIO DI LUNA, AND LEONARDO QUERZONI

Department of Computer, Control, and Management Engineering Antonio Ruberti, Sapienza University of Rome, 00185 Rome, Italy

Corresponding author: Gianluca Capozzi (capozzi@diag.uniroma1.it)

This work was supported in part by the Ministero dell' Università e della Ricerca (MUR) National Recovery and Resilience Plan funded by the European Union–NextGenerationEU through the Project SEcurity and RIghts In the CyberSpace (SERICS) under Grant PE00000014 and through the project Rome Technopole under Grant ECS00000024, in part by Sapienza Ateneo under Project RM1221816C1760BF and Project AR1221816C754C33, and in part by the Amazon Web Services (AWS) Cloud Credit Program.

ABSTRACT Binary analysis has become essential for software inspection and security assessment. As the number of software-driven devices grows, research is shifting towards autonomous solutions using deep learning models. In this context, a hot topic is the binary similarity problem, which involves determining whether two assembly functions originate from the same source code. However, it is unclear how deep learning models for binary similarity behave in an adversarial context. In this paper, we study the resilience of binary similarity models against adversarial examples, showing that they are susceptible to both targeted and untargeted (w.r.t. similarity goals) attacks performed by black-box and white-box attackers. We extensively test three state-of-the-art binary similarity solutions against (i) a black-box greedy attack that we enrich with a new search heuristic, terming it *Spatial Greedy*, and (ii) a white-box attack in which we repurpose a gradient-guided strategy used in attacks to image classifiers. Interestingly, the target models are more susceptible to black-box attacks than white-box ones, exhibiting greater resilience in the case of targeted attacks.

INDEX TERMS Adversarial attacks, binary analysis, binary code models, binary similarity, black-box attacks, greedy, white-box attacks.

I. INTRODUCTION

An interesting problem that currently is a hot topic in the security and software engineering research communities [1], [2], [3], is the *binary similarity problem*. That is, to determine if two functions in assembly code are compiled from the same source code [4]: if so, the two functions are said *similar*. This problem is far from trivial: it is well-known that different compilers and optimization levels radically change the shape of the generated assembly code.

Binary similarity has many applications, including plagiarism detection, malware detection and classification, and vulnerability detection [5], [6], [7]. It can also be a valid aid for a reverse engineer as it helps with the identification of functions taken from well-known libraries or open-source software. Recent research [4] shows that techniques for binary similarity generalize, as they are able to find similarities between semantically similar functions.

We can distinguish binary similarity solutions between the ones that use deep neural networks (DNNs), like [4], [8], and [9], and the ones that do not, like [1] and [10]. Nearly all of the most recent works rely on DNNs, which offer in practice state-of-the-art performance while being computationally inexpensive. This aspect is particularly apparent when compared with solutions that build on symbolic execution or other computationally intensive techniques.

A drawback of DNN-based solutions is their sensitivity to adversarial attacks [11] where an adversary crafts an innocuously looking instance with the purpose of misleading the target neural network model. Successful adversarial attacks are well-documented for DNNs that process, for example, images [12], [13], [14], audio and video samples [15], and text [16].

The associate editor coordinating the review of this manuscript and approving it for publication was Mahmoud Elish.

In spite of the wealth of works identifying similar functions with ever improving accuracy, we found that an extensive study on the resilience of (DNN-based) binary similarity solutions against adversarial attacks is missing. Indeed, we believe binary similarity systems are an attractive target for an adversary. As examples, an attacker: (1) may hide a malicious function inside a firmware by making it similar to a benign white-listed function, as similarly done in malware misclassification attacks [17]; (2) may make a plagiarized function dissimilar to the original one, analogously to source code authorship attribution attacks [18]; or, we envision, (3) may replace a function—entirely or partially, as in forward porting of bugs [19]—with an old version known to have a vulnerability and make the result dissimilar from the latter.

In this context, we can define an attack **targeted** when the goal is to make a rogue function be the most similar to a target, as with example (1). Conversely, an attack is **untargeted** when the goal is to make a rogue function the most dissimilar from its original self, as with examples (2) and (3). In both scenarios, the adversarial instance has to preserve the semantics (i.e., execution behavior) of the rogue function as in the original.

In this paper, we aim to close this gap by proposing and evaluating techniques for **targeted** and **untargeted** attacks using both **black-box** (where adversaries have access to the similarity model without knowing its internals) and **white-box** (where they know also its internals) methods.

For the black-box scenario, we adopt a greedy optimizer to modify a function by inserting a single assembly instruction to its body at each optimization step. Where applicable, we consider an enhanced gray-box [17] variant that, leveraging limited knowledge of the model, chooses only between instructions that the model treats as distinct.

We then enrich the greedy optimizer with a novel black-box search heuristic, where we transform the discrete space of assembly instructions into a continuous space using a technique based on instruction embeddings [20]. We call this enhanced black-box attack *Spatial Greedy*. When using our heuristic, the black-box attack is on par or outperforms the gray-box greedy attack, *without requiring any knowledge of the model*. For the white-box scenario, we repurpose a method for adversarial attacks on images that relies on gradient descent [21] and use it to drive instruction insertion decisions.

We test our techniques against three binary similarity systems—Gemini [9], GMN [22], and SAFE [4]—focusing on three research questions: **(RQ1)** determining whether the target models are more robust against targeted or untargeted attacks; **(RQ2)** assessing whether the target models exhibit greater resilience to black-box or white-box approaches; and **(RQ3)** exploring how target models influence the effectiveness of our attacks. Our results indicate that all the three models are inherently more vulnerable to untargeted attacks. In the targeted scenario, the best attack technique mislead the target models in 31.6% of instances for Gemini, 59.68% for GMN, and 60.68% for SAFE. However, in the untargeted scenario, these percentages increased to 53.89% for Gemini, 93.81% for GMN, and 90.62% for SAFE. Our analysis shows that all target models are more resilient to our white-box procedure; we believe this is largely due to the inherent challenges of conducting gradient-based attacks on models that use discrete representations.

A. CONTRIBUTIONS

This paper proposes the following contributions:

- we propose to study the problem of adversarial attacks against binary similarity systems, identifying targeted and untargeted attack opportunities;
- we investigate black-box attacks against DNN-based binary similarity systems, exploring an instruction insertion technique based on a greedy optimizer. Where applicable, we enhance it in a gray-box fashion for efficiency, using partial knowledge of the model sensitivity to instruction types;
- we propose Spatial Greedy, a fully black-box attack that matches or outperforms gray-box greedy by using a novel search heuristic for guiding the choice of the candidates' instructions used during the attack;
- we investigate white-box attacks against DNN-based binary similarity systems, exploring a gradient-guided search strategy for inserting instructions;
- we conduct an extensive experimental evaluation of our techniques in different attack scenarios against three systems backed by largely different models and with high performance in recent studies [23].

II. RELATED WORKS

In this section, we first discuss loosely related approaches for attacking image classifiers and natural language processing (NLP) models; then, we describe attacks against source code models. Finally, we discuss prominent attacks against models for binary code analysis.

A. ATTACKS TO IMAGE CLASSIFIERS AND NLP MODELS

Historically, the first adversarial attacks targeted image classifiers. The crucial point for these attacks is to insert inside a clean image instance a perturbation that should not be visible to the human eye while being able to fool the target model, as first pointed out by [12] and [13].

Most of the attacks modify the original instances using gradient-guided methods. In particular, when computing an adversarial example, they keep the weights constant while altering the starting input in the direction of the gradient that mimizes (or maximizes, depending on whether the attack is targeted or untargeted) the loss function of the attacked model. The FGSM attack [13] explicitly implements this technique. Other attacks, such as the Carlini-Wagner [14] one, generate a noise that is subject to L_p-norm constraints to preserve similarity to original objects.

As observed in Section III-B, adversarial examples generation is possibly easier in the image domain than in the textual one, due to the continuous representation of the original objects. In the NLP domain, the inputs are discrete objects, a fact that prevents any direct application of gradient-guided methods for adversarial examples generation. Ideally, check perturbations to fool deep models for language analysis should be grammatically correct and semantically coherent with the original instance.

One of the earliest methodologies for attacking NLP models is presented in [16]. The authors propose attacks to mislead deep learning-based reading comprehension systems by inserting perturbations in the form of new sentences inside a paragraph, so as to confuse the target model while maintaining intact the original correct answer. The attacks proposed in [24] and [25] focus on finding replacement strategies for words composing the input sequence. Intuitively, valid substitutes should be searched through synonyms; however, this strategy could fall short in considering the context surrounding the word to substitute. Works like [26] and [27] further investigate this idea using BERT-based models for identifying accurate word replacements.

B. ATTACKS AGAINST MODELS FOR SOURCE CODE ANALYSIS

This section covers some prominent attacks against models that work on source code.

The general white-box attack of [28] iteratively substitutes a target variable name in all of its occurrences with an alternative name until a misclassification occurs. The attack against plagiarism detection from [18] uses genetic programming to augment a program with code lines picked from a pool and validated for program equivalence by checking that an optimizer compiler removes them. The attack against clone detectors from [29] combines several semantics-preserving perturbations of source code using different optimization heuristic strategies.

We highlight that these approaches have limited applicability in the binary similarity scenario, as their perturbations may not survive compilation (e.g., variable renaming) or result in marginal differences in compiled code (e.g., turning a while-loop into a for-loop).

C. ATTACKS AGAINST MODELS FOR BINARY CODE ANALYSIS

We complete our review of related works by covering research on evading ML-based models for analysis of binary code.

1) ATTACKS AGAINST MALWARE DETECTORS

Attacks such as [30] and [31] to malware detectors based on convolutional neural networks add perturbations in a new non-executable section appended to a Windows PE binary. Both use gradient-guided methods for choosing single-byte perturbations to mislead the model in classifying the whole binary. We emphasize that binary similarity systems analyze executable code, meaning these attacks are ineffective in our scenario.

Pierazzi et al. [17] explore transplanting binary code gadgets into a malicious Android program to avoid detection. The attack follows a gradient-guided search strategy based on a greedy optimization. In the initialization phase, they mine from benign binaries code gadgets that modify features that the classifier uses to compute its classification score. In the attack phase, they pick the gadgets that can mostly contribute to the (mis)classification of the currently analyzed malware sample; they insert gadgets in order of decreasing negative contribution, repeating the procedure until misclassification occurs. To preserve program semantics, gadgets are injected into never-executed code portions. Differently from our main contribution, their attack is only applicable in a targeted white-box scenario.

Lucas et al. [32] target malware classifiers analyzing raw bytes. They propose a functionality-preserving iterative procedure viable for both black-box and white-box attackers. At every iteration, the attack determines a set of applicable perturbations for every function in the binary and applies a randomly selected one (following a hill-climbing approach in the black-box scenario or using the gradient in the white-box one). Done via binary rewriting, the perturbations are local and include instruction reordering, register renaming, and replacing instructions with equivalent ones of identical length. The results show that these perturbations can be effective even against (ML-based) commercial antivirus products, leading the authors to advocate for augmenting such systems with provisions that do not rely on ML. In the context of binary similarity, though, we note that these perturbations would have limited efficacy if done on a specific pair of functions: for example, both instruction reordering and register renaming would go completely unnoticed by Gemini and GMN (Section VIII-A and VIII-B). Furthermore, since [32] is mainly designed for models that classify binary programs, it is not directly applicable in our scenario, where the output of the model is a real value representing the distance between the two inputs.

MAB-Malware [33] is a reinforcement learning-based approach for generating adversarial examples against PE malware classifiers in a black-box context. Adversarial examples are generated through a multi-armed bandit (MAB) model that has to keep the sample in a single, non-evasive state when selecting actions while learning reward probabilities. The goal of the optimization strategy is to maximize the total reward. The set of applicable perturbations (which can be considered as actions) are standard PE manipulation techniques from prior works: header manipulation, section insertion and manipulation (e.g., adding trailing byte), and in-place randomization of an instruction sequence (i.e., replacing it with a semantically equivalent one). Each action is associated with a specific content—a payload—added to the malware when the action is selected. An extensive evaluation is conducted on two popular ML-based classifiers

and three commercial antivirus products. As outlined for other works, our scenario does not allow for the application of this approach for two primary reasons. Firstly, this attack is specifically designed to target classifiers. Secondly, many of the proposed transformations are ineffective when applied to binary similarity systems.

2) ATTACKS AGAINST BINARY SIMILARITY MODELS

Concurrently to our work, a publicly available technical report proposes FuncFooler [34] as a black-box algorithm for attempting untargeted attacks against ranking systems (i.e., top-k most similar functions) based on binary similarity. The key idea behind the attack is to insert instructions likely to push the source function below the top results returned by the search engine. Insertion points are fixed: specifically, CFG nodes that dominate the exit points of a function. The algorithm picks the instructions directly from those functions with the least similarity in the pool under analysis, then it compensates for their side effects through additional insertions. Differently from their goal to attack binary similarity-based ranking systems, our goal is to directly attack the similarity function implemented by the target model; additionally, differently from their black-box approach designed only for untargeted attacks, we propose methodologies for assessing the robustness of the considered systems against both targeted and untargeted attacks, extending the evaluation to white-box attacks.

III. BACKGROUND

In this section, we provide background knowledge for adversarial attacks against models for code analysis. Then, we introduce a categorization of semantics-preserving perturbations for binary functions.

A. ADVERSARIAL KNOWLEDGE

We can describe a deep learning model through different aspects: training data, layers architecture, loss function, and weights parameters. Having complete or partial knowledge about such elements can facilitate an attack from a computational point of view. According to seminal works in the area [17], [35], we can distinguish between:

- **white-box** attacks, where the attacker has *perfect knowledge* of the target model, including all the dimensions mentioned before. These type of attacks are realistic when the adversary has direct access to the model (e.g., an open-source malware classifier);
- **gray-box** attacks, where the attacker has *partial knowledge* of the target model. For example, they have knowledge about feature representation (e.g., categories of features relevant for feature extraction);
- **black-box** attacks: the attacker has *zero knowledge* of the target model. Specifically, the attacker is only aware of the task the model was designed for and has a rough idea of what potential perturbations to apply to cause some feature changes [35].

Different attack types may suit different scenarios best. A white-box attack, for example, could be attempted on an open-source malware classifier. Conversely, a black-box attack would suit also a model hosted on a remote server to interrogate, as with a commercial cloud-based antivirus.

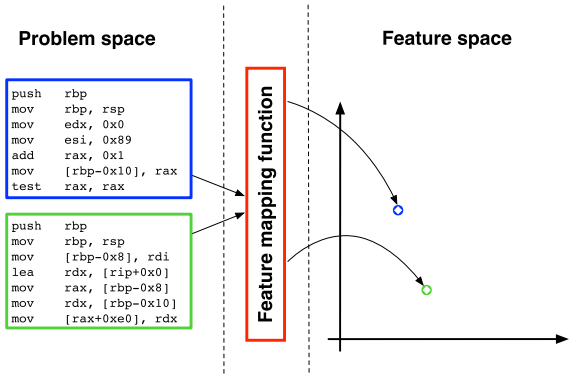

FIGURE 1. A feature mapping function maps problem-space objects into feature vectors. The two boxed binary functions implement similar functionalities and are mapped to two points close in the feature space.

B. INVERSE FEATURE MAPPING PROBLEM

In the following, we refer to the input domain as *problem space* and to all its instances as *problem-space* objects.

Deep learning models can manipulate only continuous problem-space objects. When inputs have a discrete representation, a first phase must map them into continuous instances. The phase usually relies on a *feature mapping function* (Figure 1) whose outputs are *feature vectors*. The set of all possible feature vectors is known as the *feature space*.

Traditional white-box attacks against deep learning models solve an optimization problem in the feature space by minimizing an objective function in the direction following its negative gradient [21]. When the optimization ends, they obtain a feature vector that corresponds to a problem-space object representing the generated adversarial example.

Unfortunately, given a feature vector, it is not always possible to obtain its problem-space representation. This issue is called the **inverse feature mapping** problem [17].

For code models, the feature mapping function is neither invertible nor differentiable. Therefore, one cannot understand how to modify an original problem-space object to obtain the given feature vector. In particular, the attacker has to employ approximation techniques that create a feasible problem-space object from a feature vector. Ultimately, mounting an attack requires a manipulation of a problem-space object via perturbations guided by either gradient-space attacks (as in the white-box case above) or "gradient-free" optimization techniques (as with black-box attacks). We discuss perturbations specific to our context next.

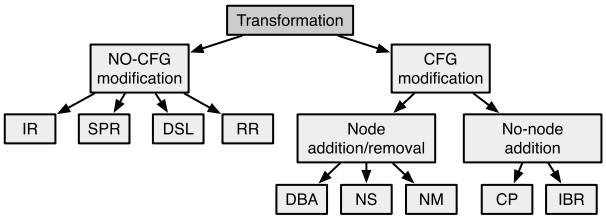

FIGURE 2. Taxonomy of semantics-preserving perturbations suitable for the proposed attacks. Acronyms are spelled out in the body of the paper.

C. SEMANTICS-PRESERVING PERTURBATIONS OF PROBLEM-SPACE OBJECTS

In this section, we discuss how to modify problem-space objects in the specific case of binary code models working on functions. To this end, we review and extend perturbations from prior works [17], [32], [33], identifying those suitable for adversarial manipulation of functions.

For our purpose, we seek to modify an original binary function f into an adversarial binary example f_{adv} that preserves the semantics of f; intuitively, this restricts the set of available perturbations for the adversary. We report a taxonomy of possible *semantics-preserving* perturbations in Figure 2, dividing them according to how they affect the binary layout of the function's control-flow graph (CFG).

(a) Original example

```
push ebp
subl esp, 0x12
xorl ebx, ebx
notl ebx
test eax, eax
movl ebp, ebx
sarl ebp, 0x8
xorl ebp, [esp+0x8]
decl edi
```

(b) IR

```
push ebp
subl esp, 0x12
xorl ebx, ebx
notl ebx
movl ebp, ebx
test eax, eax
sarl ebp, 0x8
xorl ebp, [esp+0x8]
decl edi
```

(c) SPR

```
push ebp
addl esp, -0x12
xorl ebx, ebx
notl ebx
movl ebp, ebx
test eax, eax
sarl ebp, 0x8
xorl ebp, [esp+0x8]
decl edi
```

(d) RR

```
push ebp
subl esp, 0x12
xorl ecx, ecx
notl ecx
movl ebp, ecx
test eax, eax
sarl ebp, 0x8
xorl ebp, [esp+0x8]
decl edi
```

FIGURE 3. Examples of semantics-preserving perturbations that do not alter the binary CFG layout. We modify the assembly snippet in (a) by applying, in turn, (b) Instruction Reordering, (c) Semantics-Preserving Rewriting, and (d) Register Renaming. Altered instructions are in red.

Among CFG-preserving perturbations, we identify:

- **(IR) Instruction Reordering**: reorder independent instructions in the function;
- **(SPR) Semantics-Preserving Rewriting**: substitute a sequence of instructions with a semantically equivalent sequence;
- **(DSL) Modify the Data-Section Layout**: modify the memory layout of the `.data` section and update all the global memory offsets referenced by instructions;
- **(RR) Register Renaming**: change all the occurrences of a register as instruction operand with a register currently not in use or swap the use of two registers.

Figure 3 shows examples of their application. As for perturbations that affect the (binary-level) CFG layout, we can identify the ones that involve adding or deleting nodes:

- **(DBA) Dead Branch Addition**: add dead code in a basic block guarded by an always-false branch;
- **(NS) Node Split**: split a basic block without altering the semantics of its instructions (e.g., the original block will jump to the one introduced with the split);
- **(NM) Node Merge**: merge two basic blocks when semantics can be preserved. For example, by using predicated execution to linearize branch-dependent assignments as conditional `mov` instructions [36].

And the ones that leave the graph structure unaltered:

- **(CP) Complement Predicates**: change the predicate of a conditional branch and the branch instruction with their negated version;
- **(IBR) Independent Blocks Reordering**: change the order in which independent basic blocks appear in the binary representation of the function.

IV. THREAT MODEL AND PROBLEM DEFINITION

In this section, we define our threat model together with the problem of attacking binary similarity models.

A. THREAT MODEL

The focus of this work is to create adversarial instances that attack a model at inference time (i.e., we do not investigate attacks at training time). Following the description provided in Section III-A, we consider two different attack scenarios: respectively, a black-box and a white-box one. In the first case, the adversary has no knowledge of the target binary similarity model; nevertheless, we assume they can perform an unlimited number of queries to observe the output produced by the model. In the second case, we assume that the attacker has perfect knowledge of the target binary similarity model.

B. PROBLEM DEFINITION

Let *sim* be a similarity function that takes as input two functions, f_1 and f_2, and returns a real number, the *similarity score* between them, in [0, 1].

We define two binary functions to be *semantically equivalent* if they are two implementations of the same abstract functionality. We assume that there exists an adversary that wants to attack the similarity function. The adversary can mount two different kind of attacks:

- **Targeted attack**. Given two binary functions, f_1 (identified as *source*) and f_2 (identified as *target*), the adversary wants to find a binary function f_{adv} semantically equivalent to f_1 such that: $sim(f_{adv}, f_2) \geq \tau_t$, where τ_t is a

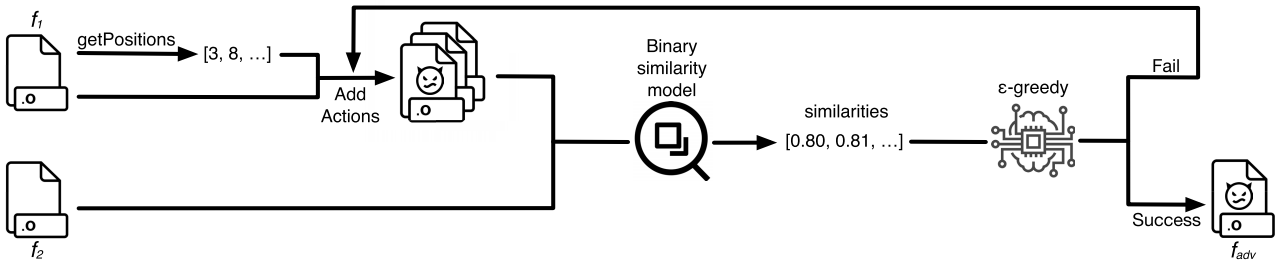

FIGURE 4. Overall workflow of the *black-box* ε-greedy perturbation-selection strategy in the *targeted* scenario.

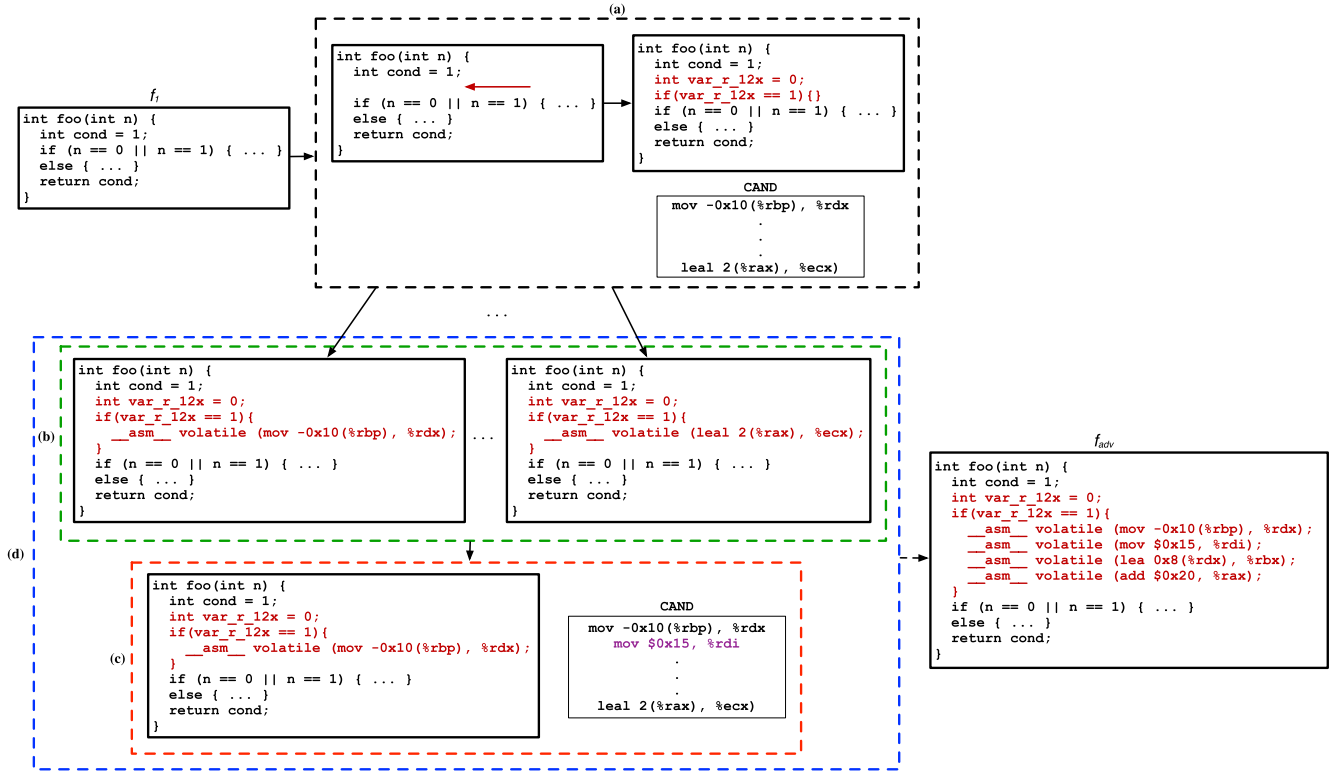

FIGURE 5. Toy example describing how the *source* function f_1 is modified during the various steps of our Spatial Greedy attack. We first identify the set of available positions and initialize the candidates' set CAND **(a). Then, we enumerate all the possible perturbations (b) and choose one according to the** ε**-greedy strategy while updating** CAND **according to the Spatial Greedy heuristic (c). This process (d) is repeated until a successful adversarial example is generated or we reach a maximum number of iterations.**

success threshold[1] chosen by the attacker depending on the victim at hand.

- **Untargeted attack**. Given a binary function f_1, the adversary goal consists of finding a binary function f_{adv} semantically equivalent to f_1 such that: $sim(f_1, f_{adv}) \leq \tau_u$. The threshold τ_u is the analogous of the previous case for the untargeted attack scenario.

Loosely speaking, in case of **targeted attack**, the attacker wants to create an adversarial example that is as similar as possible to a specific function, as in the example scenario (1) presented in Section I. In case of **untargeted attack**, the goal of the attacker consists of creating an adversarial example that is as dissimilar as possible from its original version, as in the example scenarios (2) and (3) also from Section I.

C. PERTURBATION SELECTION

Given a binary function f_1, our attack consists in applying to it perturbations that do not alter its semantics.

To study the feasibility of our approach, we choose *dead branch addition* (DBA) among the suitable perturbations outlined in Section III-C. We find DBA adequate for this study for two reasons: it is sufficiently expressive so as to affect heterogeneous models (which may not hold for others[2]) and its implementation complexity for an attacker is fairly

[1] Although f_{adv} and f_2 are similar for the model, they are not semantically equivalent: this is precisely the purpose of an attack that wants to fool the model to consider them as such, while they are not.

[2] For example, basic block-local transformations such as IR and RR would have limited efficacy on models that study an individual block for its instruction types and counts or other coarse-grained abstractions. This is the case with Gemini and GMN that we attack in this paper.

limited. Nonetheless, other choices remain possible, as we will further discuss in Section XIII.

At each application, our embodiment of DBA inserts in the binary code of f_1 one or more instructions in a new or existing basic block guarded by a branch that is never taken at runtime (i.e., we use an always-false branch predicate).

Such a perturbation can be done at compilation time or on an existing binary function instance. For our study, we apply DBA during compilation by adding placeholder blocks as inline assembly, which eases the generation of many adversarial examples from a single attacker-controlled code. State-of-the-art binary rewriting techniques would work analogously over already-compiled source functions.

We currently do not attempt to conceal the nature of our branch predicates for preprocessing robustness, which [17] discusses as something that attackers should be wary of to mount stronger attacks. We believe off-the-shelf obfuscations (e.g., opaque predicates, mixed boolean-arithmetic expressions) or more complex perturbation choices may improve our approach in this respect. Nevertheless, our main goal was to investigate its feasibility in the first place.

V. BLACK-BOX ATTACK: SOLUTION OVERVIEW

In this section, we describe our black-box attack. We first introduce our baseline (named **Greedy**), highlighting its limitations. We then move to our main contribution in the black-box scenario (named **Spatial Greedy**). Figure 4 depicts a general overview of our black-box approach.

A. GREEDY

The baseline black-box approach we consider for attacking binary function similarity models consists of an iterative perturbation-selection rule that follows a greedy optimization strategy. Starting from the original sample f_1, we iteratively apply perturbations $T_1, T_2, \ldots, T_k$ selected from a set of available ones, generating a series of instances $f_{adv_1}, f_{adv_2}, \ldots, f_{adv_k}$. This procedure ends upon generating an example f_{adv} meeting the desired similarity threshold, otherwise the attack fails after $\bar{\delta}$ completed iterations.

For instantiating Greedy using DBA, we reason on a set of positions `BLK` for inserting dead branches in function f_1 and a set of instructions `CAND`, which we call the *set of candidates*. Each perturbation consists of a $\langle$`bl`, `in`$\rangle$ pair made of the branch `bl` $\in$ `BLK` and an instruction `in` $\in$ `CAND` to check insert in the dead code block guarded by `bl`.

The naive perturbation-selection rule (i.e., greedy) at each step selects the perturbation that, in case of targeted attack, locally maximizes the relative increase of the objective function. Conversely, for an untargeted attack, the optimizer selects the perturbation that locally maximizes the relative decrease of the objective function.

This approach, however, may be prone to finding local optima. To avoid this problem, we choose as our Greedy baseline an ε-greedy perturbation-selection rule. Here, we select with a small probability ε a suboptimal perturbation instead of the one that the standard greedy strategy picks, and with probability $1 - \varepsilon$ the one representing the local optimum.

In case of targeted attack, the objective function is the similarity between f_{adv} and the target function f_2 (formally, $sim(f_{adv}, f_2)$) while it is the negative of the similarity between f_{adv} and the original function in case of untargeted attack (formally, $-sim(f_1, f_{adv})$). In the following, we only discuss the maximization strategy followed by targeted attacks; mutatis mutandis, the same rationale holds for untargeted attacks.

1) LIMITATIONS OF THE COMPLETE ENUMERATION STRATEGY

At each step, Greedy enumerates all the applicable perturbations computing the marginal increase of the objective function, thus resulting in selecting an instruction `in` by enumerating all the possible instructions of the considered set of candidates `CAND` for each position `bl` $\in$ `BLK`.

Unfortunately, the Instruction Set Architecture (ISA) of a modern CPU may consist of a large number of instructions. To give an example, consider the x86-64 ISA: according to [37], it has 981 unique mnemonics and a total of 3,684 instruction variants (without counting register operand choices for them). Therefore, it would be unfeasible to have a `CAND` set that covers all possible instructions of an ISA.

This means that the size of `CAND` must be limited. One possibility is to use hand-picked instructions. However, this approach has two problems. Such a set could not cover all the possible behaviors of the ISA, missing fundamental aspects (for example, leaving vector instructions uncovered); furthermore, this effort has to be redone for a new ISA. There is also a more subtle pitfall: a set of candidates fixed in advance could include instructions that the specific binary similarity model under attack deems as not significant.

On specific models, it may still be possible to use a small set of candidates profitably, enabling a **gray-box** attack strategy for Greedy. In particular, one can restrict the set of instructions to the ones that effectively impact the features extracted by the attacked model (which obviously requires knowledge of the features it uses; hence, the gray-box characterization). In such cases, this strategy is equivalent to the black-box Greedy attack that picks from all the instructions in the ISA, but computationally much more efficient.

B. SPATIAL GREEDY

In this section, we extend the baseline approach by introducing a fully black-box search heuristic. To differentiate between the baseline solution and the heuristic-enhanced one, we name the latter *Spatial Greedy*.

When using this heuristic, the black-box attack overcomes all the limitations discussed for Greedy using an adaptive procedure that dynamically updates the set of candidates according to a feedback from the model under attack *without requiring any knowledge* of it.

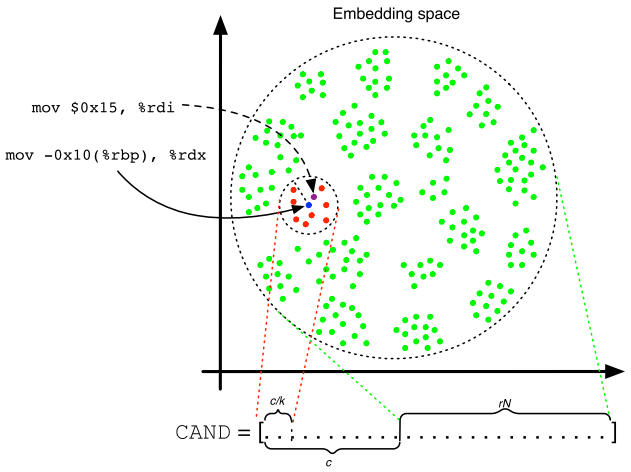

FIGURE 6. Dynamic update of the set of candidates. The `mov` instruction is the greedy action for the current iteration and is mapped to the blue point in the instruction embedding space. The set of candidates is updated selecting c/k neighbours of the considered *top-k* perturbation (represented in red), $c - c/k$ instructions among the closest neighbours of the remaining *top-k* greedy perturbations, and rN random instructions.

In Spatial Greedy, we extend the ε-greedy perturbation-selection strategy by adaptively updating the set of candidates that we use at each iteration. Using instructions embedding techniques, we transform each instruction `in` $\in$ `CAND` into a vector of real values. This creates vectors that partially preserve the semantics of the original instructions.

Chua et al. [38] showed that such vectors may be grouped by instruction semantics, creating a notion of proximity between instructions: for example, vectors representing arithmetic instructions are in a cluster, vectors representing branches in another, and so on.

Here, at each step, we populate a portion of the set of candidates by selecting the instructions that are close, in the embedding metric space, to instructions that have shown a good impact on the objective function. The remaining portion of the set is composed of random instructions. We discuss our choices for instruction embedding techniques and dynamic candidates selection in the following.

In the experimental section, for the black-box realm, we will compare Spatial Greedy against the Greedy approach, opting for the computationally efficient gray-box flavor of Greedy when allowed by the specific model under study.

1) INSTRUCTION EMBEDDING SPACE

We embed assembly instructions into numeric vectors using an instruction embedding model [20]. Given such a model M and a set I of assembly instructions, we map each $i \in I$ to a vector of real values $\vec{i} \in \mathbb{R}^n$, using M. The model is such that, for two instructions having similar semantics, the embeddings it produces will be close in the metric space.

2) DYNAMIC SELECTION OF THE SET OF CANDIDATES

The process for updating the set of candidates for each iteration of the ε-greedy perturbation-selection procedure represents the focal point of Spatial Greedy.

Algorithm 1 Spatial Greedy procedure (targeted case)

Input: source function f_1, target function f_2, similarity threshold τ_t, max number of dead branches B, max number of instructions to be inserted δ, max number of instructions to be tested N, max number of random instructions r, max number of neighbours c, probability of selecting a random perturbation ε.
Output: adversarial sample f_{adv}.
Definitions:

- The function getPositions(f_1, B) identifies B positions inside f_1 where it is possible to insert dead branches.
- The function getRandomInstructions(N) samples uniformly N instructions from the entire ISA.
- The operator $\oplus$ indicates the insertion into a function of a certain instruction into a specific block.
- The function selectGreedy($\cdot$) takes as input a vector of pairs $\langle\langle$`bl, in`$\rangle, currSim\rangle$ and returns the $\langle$`bl, in`$\rangle$ perturbation associated to the maximum *currSim* value.
- The function selectRandom($\cdot$) takes as input a vector of pairs $\langle\langle$`bl, in`$\rangle, currSim\rangle$ and returns a perturbation uniformly sampled.
- The function getTopK($\cdot, K$) takes as input a vector of pairs $\langle\langle$`bl, in`$\rangle, currSim\rangle$ and returns the instructions associated to the top-K greedy actions.
- The function updateInstructions($\cdot, r, c$) takes as input a vector of instructions and returns a vector containing c of their neighbours and r instructions sampled uniformly at random.

```
1: f_adv ← f_1
2: instr ← 0
3: BLK ← getPositions(f_1, B)                                   } a
4: CAND ← getRandomInstructions(N)                              }
5: sim ← sim(f_adv, f_2)
6: while sim ≤ τ_t AND instr < δ do                             } d
7:     iterSim ← sim
8:     iterBlock ← ⟨⟩
9:     testedPerts ← [ ]
10:    for ⟨bl, in⟩ ∈ BLK × CAND do                             } b
11:        f̄_adv ← f_adv ⊕ ⟨bl, in⟩
12:        currSim ← sim(f̄_adv, f_2)
13:        testedPerts.append(⟨⟨bl, in⟩, currSim⟩)
14:    prob ← uniform(0, 1)                                     } c
15:    if prob < ε then
16:        iterPert, iterSim ← selectGreedy(testedPerts)
17:    else
18:        iterPert, iterSim ← selectRandom(testedPerts)
19:    f_adv ← f_adv ⊕ iterPert
20:    elected ← getTopK(testedPerts, K)
21:    CAND ← updateInstructions(elected, r, c)
22:    sim ← iterSim
23:    instr ← instr + 1
24: return f_adv
```

Let N be the size of the set of candidates `CAND`. Initially, we fill it with N random instructions. Then, at each iteration of the ε-greedy procedure, we update `CAND` by replacing the current instructions with rN random instructions, where $r \in [0, 1)$, and c instructions we select among the closest neighbors of the instructions composing the top-k greedy actions of the last iteration.

In case of a targeted attack, the *top-k greedy perturbations* are the k perturbations that, at the end of the last iteration, achieved the highest increase of the objective function. To keep the size of the set stable at value N, we take the closest c/k neighbors of each top-k action.[3]

The rationale of having r random and c selected instructions is seeking a balance between *exploration* and

[3]We also apply rounding so that we can work with integer numbers.

exploitation. With the random instructions, we randomly sample the solution space to escape from a possibly local optimum found for the objective function. With the selected instructions, we exploit the part of the space that in the past has brought the best solutions. Figure 6 provides a pictorial representation of the update procedure.

We present the complete description of Spatial Greedy in case of targeted attack in Algorithm 1 together with a simplified execution example in Figure 5. The first step **(a)** consists in identifying the positions `BLK` where to introduce dead branches (function getPositions(f_1, B) at line 3) and initializing the set of candidates `CAND` with N random instructions (function getRandomInstructions(N) at line 4). Then, during the iterative procedure **(d)**, we first enumerate all the possible perturbations **(b)**. Then **(c)**, we apply the perturbation-selection rule according to the value of ε, and we get the *top-k greedy perturbations* (line 20) as depicted in Figure 6. Finally, we update the set of candidates (line 21).

VI. WHITE-BOX ATTACK: SOLUTION OVERVIEW

As pointed out in Section III-A, in a white-box scenario the attacker has a perfect knowledge of the target deep learning model, including its loss function and gradients. We discuss next how we can build on them to mount an attack.

A. GRADIENT-GUIDED CODE ADDITION METHOD

White-box adversarial attacks have been largely investigated against image classifiers by the literature, resulting in valuable effectiveness [13]. Our attack strategy for binary similarity derives from the design pattern of the PGD attack [21], which iteratively targets image classifiers.

We call our proposed white-box attack *Gradient-guided Code Addition Method* (**GCAM**). It consists in applying a set of perturbations using a gradient-guided strategy. In the case of a targeted attack, our goal is to minimize the loss function of the attacked model on the given input while keeping the perturbation size small and respecting the semantics-preserving constraint. We achieve this by using the L_p-norm as soft constraint. On the other hand, for an untargeted attack, we aim to maximize the loss function while also keeping the size of the perturbation small.

Because of the *inverse feature mapping* problem, gradient optimization-based approaches cannot be directly applied in our context (Section III-B). We need a further (hard) constraint that acts on the feature-space representation of the input binary function. This constraint strictly depends on the target model: we will further investigate its definition in Section VIII. In the following, we focus on the loss minimization strategy argued for targeted attacks. As before, we can easily adapt the same concepts to the untargeted case.

We can describe a DNN-based model for binary similarity as the concatenation of the two functions λ and sim_v. In particular, λ is the function that maps a problem-space object to a feature vector (i.e., the feature mapping function discussed in Section III-B), while sim_v is the neural network computing the similarity given the feature vectors.

Given two binary functions f_1 and f_2, we aim to find a perturbation δ that minimizes the loss function of sim_v, which corresponds to maximize $sim_v(\lambda(f_1)) + \delta, \lambda(f_2))$. To do so, we use an iterative strategy where, during each iteration, we solve the following optimization problem:

$$\min \mathcal{L}(sim_v(\lambda(f_1) + \delta, \lambda(f_2)), \theta) + \epsilon ||\delta||_p, \tag{1}$$

where $\mathcal{L}$ is the loss function, θ are the weights of the target model, and ϵ is a coefficient in $[0, \infty)$.

We randomly initialize the perturbation δ and then update it at each iteration by a quantity given by the negative gradient of the loss function $\mathcal{L}$. The vector δ has several components equal to zero and it is crafted so that it modifies only the (dead) instructions in the added blocks. The exact procedure depends on the target model: we return to this aspect in Section VIII.

Notice that the procedure above allows us to find a perturbation in the feature space, while our final goal is to find a problem-space perturbation to modify the function f_1. Therefore, we derive from the perturbation δ a problem-space perturbation δ_p. The exact technique is specific to the model we are attacking, as we further discuss in Section VIII.

The common idea behind all technique instances is to find the problem-space perturbation δ_p whose representation in the feature space is the *closest* to δ. Essentially, we use a *rounding-based inverse* strategy to solve the inverse feature mapping problem that accounts to *rounding* the feature space vector to the closest vector that corresponds to an object in the problem space. The generated adversarial example is $f_{adv} = f_1 + \delta_p$. As for the black-box scenario, the process ends whenever we reach a maximum number of iterations or the desired threshold for the similarity value.

VII. COMPARISON BETWEEN THE ATTACKS

In this section, we present a more direct comparison between the three proposed attack methodologies.

We summarize in Table 1 the key differences according to four interesting aspects: attacker's knowledge, perturbation type, usage of the candidates' set, and usage of an additional instruction embedding model.

From a technical perspective, GCAM is a white-box attack that assumes an attacker having a complete knowledge of the target model's internals. Contrarily, both Spatial Greedy and Greedy are black-box approaches, meaning that they can be easily adapted to attack any binary similarity model, without having any prior knowledge. This distinction according to the attacker's knowledge underlines a more subtle difference among the approaches; indeed, while the two black-box attacks operate in the problem space producing valid adversarial examples, GCAM initially produces perturbations in the feature space, which must then be converted into problem space objects using a rounding process.

Looking at more practical aspects, both Greedy and Spatial Greedy depend on the concept of candidates' set, while GCAM leverages the internals of the target model to guide the choice of the instructions to insert into the function

TABLE 1. Comparison and underlying principles of the three attack techniques.

	Greedy	Spatial Greedy	GCAM
Knowledge	Black-box / Gray-box	Black-box	White-box
Perturbation type	Problem-space	Problem-space	Feature-space + rounding
Candidates' set	Static	Dynamic	Not required
Instructions embedding model	Not required	Required	Not required

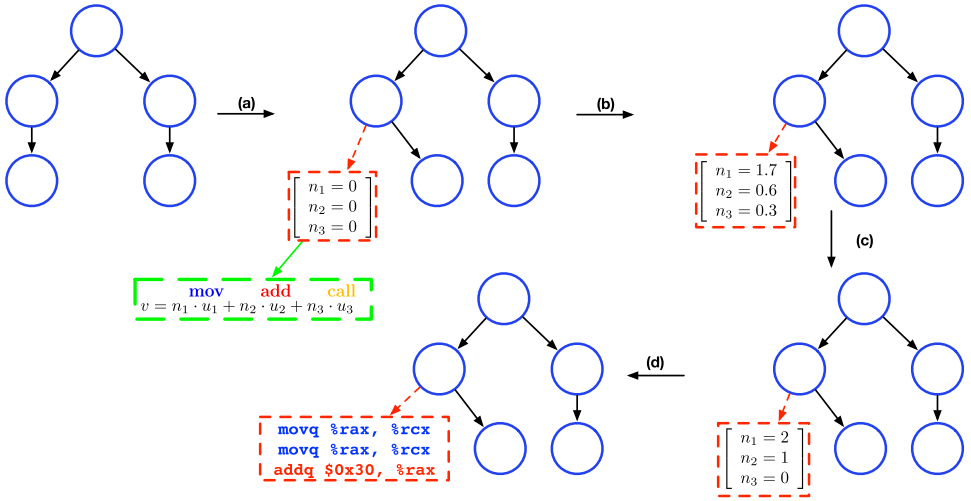

FIGURE 7. GCAM attack against Gemini. Once obtained the initial CFG of the function f_1, we initialize an empty dead branch in one of the available positions (a). In particular, each node is represented as a feature vector v, which is the linear combination of three embedding vectors corresponding to three different categories of instructions (green block). We then iteratively apply the gradient descent to modify the coefficients n_j associated to the instruction vectors (b), obtaining a vector of non-integer values. Finally, we round the obtained coefficients to the closest integer values (c) and, (d), we insert into the dead branch as many instructions belonging to the class j as specified by the coefficient n_j.

according to the objective function. Specifically, GCAM can potentially utilize the entire set of instructions encountered by the target model during training, while the black-box methods are constrained to a predetermined set of instructions that can be tested during each iteration. As highlighted in Section V-A1, the usage of a manually-crafted candidates' set represents the main weakness of the Greedy procedure, which we addressed with the Spatial Greedy heuristic proposing an adaptive set based on the usage of instruction embeddings.

Finally, when considering Spatial Greedy, it is important to note that one should train from scratch an instruction embedding model to effectively apply the embedding based search heuristic. However, we remark that the model has to be trained only once and then it can be reused for all the attacks against binary for a certain ISA.

VIII. TARGET SYSTEMS

In this section, we illustrate the three models we attacked: Gemini [9], GMN [22], and SAFE [4].

We selected the models by conducting a literature review [23] to identify plausible candidates. We then analyzed the characteristics of existing binary similarity systems, choosing models that are fundamentally different from one another. This approach allows us to test the generality of our solution. Specifically, the three models we selected can be distinguished by the following features:

- **NN architecture**: Both Gemini and GMN are GNN-based models while SAFE is a RNN-based one.
- **Input representation**: Both Gemini and GMN represent functions through their CFGs while SAFE uses the linear disassembly.
- **Feature mapping process**: Both Gemini and GMN use manual features from the CFG nodes, while SAFE learns features using an instruction embedding model.

In the following, we provide an overview of the internal workings of the models and then discuss specific provisions for the Greedy (Section V-A) and GCAM (Section VI) attacks. Notably, Spatial Greedy needs no adaptations.

A. GEMINI

Gemini [9] represents functions in the problem space through their Attributed Control Flow Graph (ACFG). An ACFG is a control flow graph where each basic block consists of a vector of manual features (i.e., node embeddings).

The focal point of this approach consists of a graph neural network (GNN) based on the Structure2vec [39] model

that converts the ACFG into an embedding vector, obtained by aggregating the embedding vectors of individual ACFG nodes. The similarity score for two functions is given by the cosine similarity of their ACFG embedding vectors.

1) GREEDY ATTACK

Each ACFG node contributes a vector of 8 manually selected features. Five of these features depend on the characteristics of the instructions in the node, while the others on the graph topology. The model distinguishes instructions from an ISA only for how they contribute to these 5 features. This enables a gray-box variant of our Greedy attack: we measure the robustness of Gemini using a set of candidates `CAND` of only five instructions, carefully selected for covering the five features. Later in the paper, we use this variant as the baseline approach for a comparison with Spatial Greedy.

2) GCAM ATTACK

As described in the previous section, some of the components of a node feature vector v depend on the instructions inside the corresponding basic block. As Gemini maps all possible ISA instructions into 5 features, we can associate each instruction with a deterministic modification of v represented as a vector u. We select five categories of instructions and for each category c_j we compute the modification u_j that will be applied to the feature vector v. We selected the categories so as to cover the aforementioned features.

When we introduce in the block an instruction belonging to category c_j, we add its corresponding u_j modification to the feature vector v. Therefore, inserting instructions inside the block modifies the feature vector v by adding to it a linear combination vector $\sum_j n_j u_j$, where n_j is the number of instructions of category c_j added. Our perturbation δ acts on the feature vector of the function only in the components corresponding to the added dead branches, by modifying the coefficients of the linear combination above.

Since negative coefficients are meaningless, we avoid them by adding to the optimization problem appropriate constraints. Moreover, we solve the optimization problem without forcing the components of δ to be integers, as this would create an integer programming problem. Therefore, at the end of the iterative optimization process, we get our problem-space perturbation δ_p by *rounding* to the closest positive integer value each component of δ. It is immediate to obtain from δ_p the problem-space perturbation to insert in our binary function f_1. Indeed, in each dead block, we must add as many instructions belonging to a category as the corresponding coefficient in δ_p. We report a simplified example of the GCAM procedure against Gemini in Figure 7.

B. GMN

Graph Matching Network (GMN) [22] computes the similarity between two graph structures. When functions are represented through their CFGs, GMN offers state-of-the-art performance for the binary similarity problem [22], [23].

Differently from solutions based on standard GNNs (e.g., Gemini), which compare embeddings built separately for each graph, GMN computes the distance between two graphs as it attempts to match them. In particular, while in a standard GNN the embedding vector for a node captures properties of its neighborhood only, GMN also accounts for the similarity with nodes from the other graph.

1) GREEDY ATTACK

Similarly to the case of Gemini, each node of the graph consists of a vector of manually-engineered features. In particular, each node is a bag of 200 elements, each of which represents a class of assembly instructions, grouped according to their mnemonics. The authors do not specify why they only consider these mnemonics among all the available ones in the `x86-64` ISA. Analogously to Gemini, when testing the robustness of this model against the Greedy approach we devise a gray-box variant by considering a set of candidates `CAND` of 200 instructions, each of which belonging to one and only one of the considered classes.

2) GCAM ATTACK

Our white-box attack operates analogously to what we presented in Section VIII-A2 and illustrated in Figure 7. Similarly to the Gemini case, each dead branch adds a node to the CFG while the feature mapping function transforms each CFG node into a feature vector. The feature vector is a bag of the instructions contained in the node, where assembly instructions are divided into one of 200 categories using the mnemonics.

C. SAFE

SAFE [4] is an embedding-based similarity model. It represents functions in the problem space as sequences of assembly instructions. It first converts assembly instructions into continuous vectors using an instruction embedding model based on the word2vec [20] word embedding technique. Then, it supplies such vectors to a bidirectional self-attentive recurrent neural network (RNN), obtaining an embedding vector for the function. The similarity between two functions is the cosine similarity of their embedding vectors.

1) GREEDY ATTACK

The Greedy attack against SAFE follows the black-box approach described in Section V-A. Since SAFE does not use manually engineered features, we cannot select a restricted set of instructions that generates all vectors of the feature space for a gray-box variant. We test its resilience against the Greedy approach considering a carefully designed list of candidates `CAND` composed of random and hand-picked instructions, meaning that the baseline is a black-box attack.

2) GCAM ATTACK

In the feature space, we represent a binary function as a sequence of instruction embeddings belonging to a

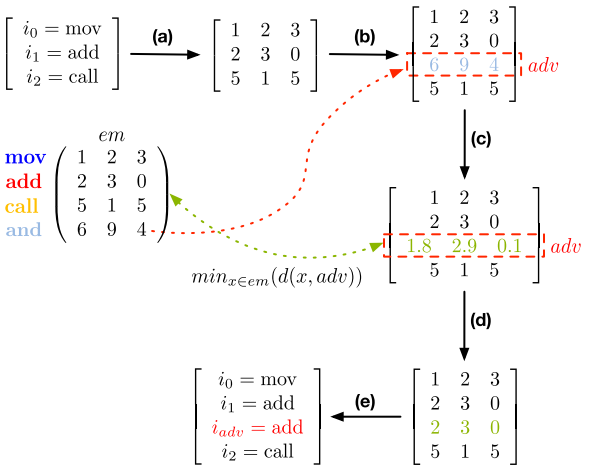

FIGURE 8. GCAM attack against SAFE. Once obtained the initial linear disassembly of the function f_1, we map each instruction to its embedding (a) using the embedding matrix *em*, obtaining the feature space representation of f_1. Then, we initialize the perturbation by inserting into the feature space representation of f_1 the embedding vector *adv* associated to a real instruction (b) uniformly sampled from the embeddings in *em*. We then iteratively modify *adv* by applying the gradient descent (c). Finally, we approximate the obtained *adv* vector to the closest embedding in *em* (d) and we insert its corresponding instruction into f_1 (e).

predefined metric space. The perturbation δ is a sequence of real-valued vectors initialized with embeddings of real random instructions; each dead block contains four of such vectors. In the optimization process, we modify each embedding $i_j \in \delta$ by a small quantity given by the negative gradient of the loss function $\mathcal{L}$. In other words, every time we optimize the objective function, we alter each $i_j \in \delta$ by moving it in the negative direction identified through the gradient.

Since during optimization we modify instruction embeddings in terms of their single components, we have no guarantee that the obtained vectors are embeddings of real instructions. For this reason, after the optimization process, we compute the problem-space perturbation δ_p by *approximating*, at each iteration, the vectors in δ to the closest embeddings in the space of real instruction embeddings. At this point, it is straightforward to obtain from the approximated perturbation δ_p the instructions that should be added to the binary function f_1; indeed, each vector in δ_p corresponds to the embedding of a real instruction that will be inserted into the function f_1. We report a simplified example of the **GCAM** procedure against SAFE in Figure 8.

IX. DATASETS AND IMPLEMENTATION

In this section, we discuss the evaluation datasets and the corpus for training the embedding model of Spatial Greedy.

A. ATTACK DATASET

We test our approaches by considering pairs of binary functions randomly extracted from 6 open-source projects written in C language: binutils, curl, gsl, libconfig, libhttp, and openssl. We compile the programs for an `x86-64` architecture using the `gcc` 9.2.1 compiler with `-O0` optimization level on Ubuntu 20.04. We filter out all functions with less than six instructions. As a result, we obtain a dataset of code representative of real-world software, with source programs used in the evaluation or training of binary similarity solutions (e.g., [4], [8], [9], [23]), and that could be potential targets for the exemplary scenarios outlined in Section I.

To evaluate the robustness of the three target models against our proposed approaches, we used a dataset made of 500 pairs of binary functions sampled from the general dataset. The dataset, which we call **Targ**, consists of pairs of random functions. In its construction, a function cannot be considered more than once as a source function but may appear multiple times as a target. The functions within a pair differ at most by 1345 instructions, and on average by 135.27 instructions.

In the **untargeted** scenario, source and target functions have to coincide. For these attacks, we use the dataset **Untarg** composed by the 500 functions used as source in the **Targ** dataset. Being pairs made of identical functions, they are trivially balanced for instructions and CFG nodes.

B. DATASET USED FOR SPATIAL GREEDY

As described in Section V-B1, in Spatial Greedy we use an instruction embedding model to induce a metric space over assembly instructions. We opt for word2vec [20]; the reader may wonder whether this choice may unfairly favor Spatial Greedy when attacking SAFE, which uses word2vec in its initial instruction embedding stage. We conducted additional experiments for SAFE using two other models, GloVe [40] and fastText [41]. The three models perform almost identically in targeted attacks, while in untargeted ones fastText occasionally outperforms the others by a small margin. For the sake of generality, in the paper evaluation we will report and discuss results for word2vec only. For each of the considered models, we use the following parameters during training: embedding size 100, window size 8, word frequency 8 and learning rate 0.05. We train these models using assembly instructions as tokens. We use as training set a corpus of 23,181,478 assembly instructions, extracted from 291,688 binary functions collected by compiling various system libraries with the same setup of the previous section.

One aspect worth emphasizing is that Spatial Greedy uses embeddings unrelated to the binary similarity model being targeted. We trained the Spatial Greedy embedding model using distinct dataset and parameters compared to SAFE, whereas neither GMN nor Gemini incorporate a layer that converts a single instruction into a feature vector. Spatial Greedy relies on embeddings to enhance the instruction insertion process during the attack by clustering the instruction space, independently of the underlying model being attacked.

C. IMPLEMENTATION DETAILS

We implement our attacks in Python in about 3100 LOC. In the hope to foster research in the area, we make the code available upon request to fellow researchers.

An aspect that is worth mentioning for the black-box attacks involves the application of the perturbation $\langle \texttt{bl}, \texttt{in} \rangle$ chosen at each iteration. Modifying the binary representation of the function every time incurs costs (recompilation in our case; binary rewriting in alternative implementations) that we may avoid through a simulation. In particular, we directly modify the data structures that the target models use for feature mapping when parsing the binary, simulating the presence of newly inserted instructions. The authors implemented these models in tensorflow or pytorch, which allows us to keep our modifications rather limited. In preliminary experiments, we have verified that the similarity values from our simulation are comparable with those we would have obtained by recompiling the modified functions output by our attacks. Finally, to avoid recalculating the adversarial examples for various thresholds, we selected two fixed values for our optimizer to satisfy: $\tau_t = 0.96$ for the targeted case and $\tau_u = 0.50$ for the untargeted one. For the evaluation in Section X, we consider the adversarial examples generated by inserting the perturbations obtained at the end of the simulation process into the corresponding functions and compiling them into object files.

Finally, we want to highlight that each black-box iteration could be considered as a single query to the target model. This is possible because we are querying the model in batch mode, giving it in input a set of functions that are processed together. This implies that when setting a maximum number of iterations, we are implicitly limiting the number of queries that the attacker can perform, following the approaches adopted in [42] and [43].

X. EVALUATION

In this section, we evaluate our attacks and investigate the following research questions:

> **RQ1**: *Are the three target models more robust against targeted or untargeted attacks?*
> **RQ2**: *Are the three target models more robust against black-box or white-box approaches?*
> **RQ3**: *What is the impact of feature extracting methodologies and model architectures on the performance and the behaviour of our attacks?*

Performance Metrics: Our main evaluation metric is the **Attack success rate (A-rate)**, that is the percentage of adversarial samples that successfully mislead the target model. We complement our investigation by collecting a set of support metrics to gain qualitative and quantitative insights into the attacking process:

- **Modification size (M-size)**: number of inserted inline assembly instructions;
- **Average Similarity (A-sim)**: obtained final similarity values;
- **Normalized Increment (N-inc)**: similarity increments normalized with respect to the initial value; only used for targeted attacks;
- **Normalized Decrement (N-dec)**: similarity decrements normalized with respect to the initial value; only used for untargeted attacks.

Support metrics are computed over the set of samples that successfully mislead the model (according to the success conditions outlined in Section X-A).

As an example, let us consider a targeted attack against three pairs of functions with initial similarities 0.40, 0.50, and 0.60. After the attack we reach final similarities that are 0.75, 0.88, and 0.94, by inserting respectively 4, 7, and 12 inline assembly instructions. We deem an attack as successful if the final similarity is above $\tau_t = 0.80$ (the reason will be clear in the next section). In this example, we have an **A-rate** of 66.66%, a **M-size** of 9.5, an **A-sim** of 0.91, and a **N-inc** of 0.81.

The **N-inc** is the average of the formula below over the samples that successfully mislead the model:

$$\frac{final\ similarity - initial\ similarity}{1 - initial\ similarity} \tag{2}$$

The denominator for the fraction above is the maximum possible increment for the analyzed pair: we use it to normalize the obtained increment. Intuitively, the value of this metric is related with the initial similarities of the successfully attacked pair. Consider a targeted attack where a pair exhibits a final similarity of 0.80. When the normalized increment is 0.7, their initial similarity is 0.33 (from Equation 2); when the normalized increment is 0.3, we have a much higher 0.7 initial similarity.

The comparison between **A-sim** and the success threshold gives us insights on the ability of the attack to reach high similarity values. In the aforementioned example, the **A-sim** value of 0.91 shows that when the attack is able to exceed the success threshold, it has actually an easy time to bring the similarity around the value of 0.91.

Evaluation Outline: We test our black-box and white-bock attacks against each target model in both the targeted and the untargeted scenario. As discussed in Section IX-A, we use dataset Targ for the former and dataset Untarg for the latter.

A. SETUP

In this section, we describe the attack parameters selected for our experimental evaluation.

Successful Attacks:

An attack is successful depending on the similarity value between the adversarial example and the target function. For a targeted attack, the similarity score has to be increased during the attack until it trespasses a success threshold τ_t. For an untargeted attack, this score, which is initially 1, has to decrease until it is below a success threshold τ_u. Operatively, the values of such thresholds are determined by the way

TABLE 2. Evaluation metrics with $\tau_t = 0.80$ relative to the black-box attacks against the three target models in the targeted scenario. Spatial Greedy (SG) is evaluated using parameters $\varepsilon = 0.1$ and $r = 0.75$. Greedy (G) is evaluated using $\varepsilon = 0.1$. G* is the gray-box version of Greedy: when such a version is available (Section VIII), we show it instead of G. When examining G against SAFE, a set of candidates of size 400 is considered.

	Target	Attack	A-rate (%) ($\tau_t = 0.80$)				M-size ($\tau_t = 0.80$)				A-sim ($\tau_t = 0.80$)				N-inc ($\tau_t = 0.80$)			
			C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4
Targ	Gemini	G*	15.36	21.96	24.55	**27.94**	12.82	27.47	40.28	53.21	0.84	0.85	0.86	0.86	0.18	0.25	0.26	0.26
		SG	15.77	22.55	26.55	27.54	13.36	27.65	40.34	51.85	0.84	0.85	0.86	0.86	0.18	0.24	0.26	0.27
	GMN	G*	26.40	43.31	51.29	59.08	11.48	22.02	32.19	41.89	0.92	0.92	0.92	0.93	0.74	0.76	0.77	0.78
		SG	31.13	45.77	54.71	**59.68**	5.62	22.23	32.21	40.99	0.92	0.92	0.92	0.93	0.75	0.77	0.78	0.79
	SAFE	G	34.33	48.70	54.49	56.88	13.44	24.66	32.75	38.75	0.88	0.91	0.91	0.92	0.46	0.52	0.53	0.53
		SG	37.13	51.90	58.08	**60.68**	13.54	24.63	32.76	40.31	0.89	0.92	0.92	0.92	0.47	0.52	0.54	0.55

the similarity score is used in practice. In our experimental evaluation, we choose the thresholds as follows. We compute the similarity scores that our attacked systems give over a set of similar pairs and over a set of dissimilar pairs. For the first set, the average score is 0.79 with a standard deviation of 0.15. For the second set, these values are respectively 0.37 and 0.17. We thus opted for a success threshold $\tau_u = 0.5$ for untargeted attacks and $\tau_t = 0.8$ for targeted ones. Both τ_u and τ_t are within one standard deviation distant from the average similarity value measured for the relevant set for the attack. For the charts, we plot $\tau_u \in [0.46, 0.62]$ and $\tau_t \in [0.74, 0.88]$.

To fully understand the performance of the attacks, we also measure the amount of function pairs in a dataset already meeting a given threshold. For the targeted scenario, we plot it as a line labeled **C0**. As our readers can see (Figure 9), their contribution is marginal: hence, we do not discuss them in the remainder of the evaluation. For the untargeted scenario, no such pair can exist by construction.

Black-box Attacks: To evaluate the effectiveness of Spatial Greedy against the black-box baseline Greedy, we select a maximum perturbation size $\bar{\delta}$ (namely, the number of inserted instructions) and a number of dead branches B in four settings: **C1** ($\bar{\delta} = 15, B = 5$), **C2** ($\bar{\delta} = 30, B = 10$), **C3** ($\bar{\delta} = 45, B = 15$), and **C4** ($\bar{\delta} = 60, B = 20$).

We set $\varepsilon = 0.1$ in all greedy attacks. For Spatial Greedy and black-box Greedy, we test two sizes for the set of candidates: 110 and 400. For Greedy, we pick 110 instructions manually and then randomly add others for a total of 400; for Spatial Greedy, we recall that the selection is dynamic (Section V-B2). The larger size brought consistently better results in both attacks, hence we present results only for it. Finally, for Spatial Greedy, we use $c = 10$ and $r \in \{0.25, 0.50, 0.75\}$, with $r = 0.75$ being the most effective choice in our tests (thus, the only one presented next). For the gray-box Greedy embodiments for Gemini and GMN, we refer to Section VIII-A1 and VIII-B1, respectively.

White-box Attack: We evaluate GCAM considering four different values for the number B of dead branches inserted: **C1** ($B = 5$), **C2** ($B = 10$), **C3** ($B = 15$), and **C4** ($B = 20$). For each model, we use the number of iterations that brings the attack to convergence.

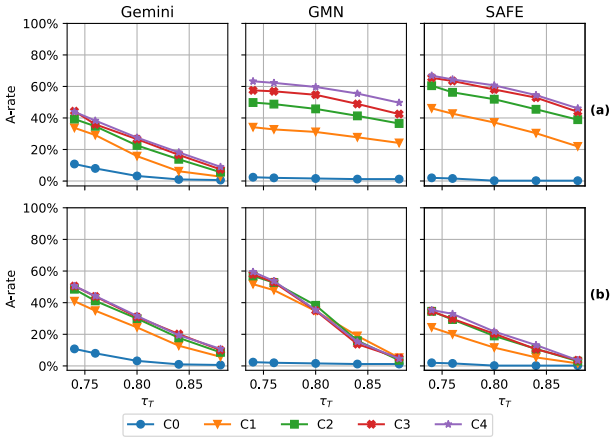

FIGURE 9. (a) Black-box targeted attack with Spatial Greedy against the three target models while varying the success threshold $\tau_t \in [0.74, 0.88]$, and settings C0 to C4. We use a set of candidates of 400 instructions, $\varepsilon = 0.1$, and $r = 0.75$. (b) White-box targeted attack against the three target models while varying the success threshold $\tau_t \in [0.74, 0.88]$ and settings C0 to C4. *Left*: GCAM attack with 20k iterations against GEMINI. *Center*: GCAM attack with 1k iterations against GMN. *Right*: GCAM attack with 1k iterations against SAFE.

B. COMPLETE ATTACK RESULTS

This section provides complete results for our black-box and white-box attacks on the three target models. For brevity, we focus only on Spatial Greedy when discussing black-box targeted and untargeted attacks, leaving out the results for the baseline Greedy. The two will see a detailed comparison later, with Spatial Greedy emerging as generally superior.

1) BLACK-BOX TARGETED ATTACK

Considering an attacker with black-box knowledge in a targeted scenario, the three target models show a similar behavior against Spatial Greedy.

The attack success rate **A-rate** is positively correlated with the number B of dead branches and the maximum number $\bar{\delta}$ of instructions introduced in the adversarial example. Fixing at $\tau_t = 0.80$ the success threshold for the attack, we have an **A-rate** that on Gemini goes from 15.77% (setting **C1**) up to 27.54% (setting **C4**). The other target models follow this

TABLE 3. Evaluation metrics with $\tau_u = 0.50$ relative to the black-box attacks against the three target models in the untargeted scenario. Spatial Greedy (SG) is evaluated using parameters $\varepsilon = 0.1$ and $r = 0.75$. Greedy (G) is evaluated using $\varepsilon = 0.1$. Similarly to Table 2, G* is the gray-box version of Greedy where applicable. When examining G against SAFE, a set of candidates of size 400 is considered.

	Target	Attack	A-rate (%) ($\tau_u = 0.50$)				M-size ($\tau_u = 0.50$)				A-sim ($\tau_u = 0.50$)				N-dec ($\tau_u = 0.50$)			
			C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4
Untarg	Gemini	G*	23.60	37.40	47.0	50.60	2.73	5.24	9.16	11.78	0.47	0.48	0.48	0.48	0.53	0.52	0.52	0.52
		SG	22.95	40.32	48.10	**53.89**	3.32	5.80	9.09	11.35	0.48	0.48	0.48	0.48	0.52	0.52	0.52	0.52
	GMN	G*	68.40	86.20	91.02	**93.81**	2.55	3.77	3.82	3.71	0.27	0.25	0.22	0.21	0.73	0.75	0.77	0.79
		SG	65.87	83.23	88.13	91.62	2.75	3.21	4.12	4.14	0.27	0.24	0.24	0.23	0.73	0.76	0.76	0.77
	SAFE	G	44.71	74.65	82.83	88.22	6.59	8.28	8.77	9.04	0.40	0.42	0.42	0.42	0.40	0.44	0.45	0.45
		SG	56.49	80.83	87.42	**90.62**	6.67	7.74	7.77	7.64	0.39	0.41	0.42	0.42	0.60	0.59	0.58	0.59

behavior, as the **A-rate** for GMN goes from 31.13% up to 59.68%, and from 37.13% up to 60.68% for SAFE. This trend holds for other success thresholds as visible in Figure 9. From these results, it is evident that the higher the values of the two parameters, the lower the robustness of the attacked models. Table 2 presents a complete overview of the results.

The other metrics confirm the relationship between the parameters B and $\bar{\delta}$ and the effectiveness of our attack. In particular, when increasing the perturbation size, as highlighted by the modification size **M-size** metric, both **A-sim** and the normalized increment **N-inc** increase, suggesting that incrementing the perturbation size is always beneficial.

2) BLACK-BOX UNTARGETED ATTACK

Considering an attacker with black-box knowledge in a untargeted scenario, all the three target models are vulnerable to Spatial Greedy, with different robustness.

The observations highlighted in Section X-B1 also hold in this scenario. Incrementing B and $\bar{\delta}$ is beneficial for the attacker. As visible in Figure 10 and in Table 3, the attack success rate **A-rate** for $\tau_u = 0.50$ in setting **C1** is 22.95% for Gemini, 65.87% for GMN, and 56.49% for SAFE. The metric increases across settings, peaking at 53.89% for Gemini, 91.62% for GMN, and 90.62% for SAFE in setting **C4**.

Table 3 also reports the results for modification size metric **M-size**. In this case, we can see the effectiveness of Spatial Greedy as a small number of inserted instructions is needed against each of the considered target models. Indeed, considering setting **C4**, which is the one that modifies the function most, the **M-size** at $\tau_u = 0.50$ is 11.35 for Gemini, 4.14 for GMN, and 7.64 for SAFE.

3) WHITE-BOX TARGETED ATTACK

With an attacker with white-box knowledge in a targeted scenario, the three target models show different behaviors. Table 4 presents a complete overview of the results.

Both Gemini and SAFE show a higher robustness to our GCAM attack if compared to GMN.

As visible in Figure 9, when attacking Gemini and SAFE, there is a positive correlation between the number B of locations (i.e., dead branches) where to insert instructions

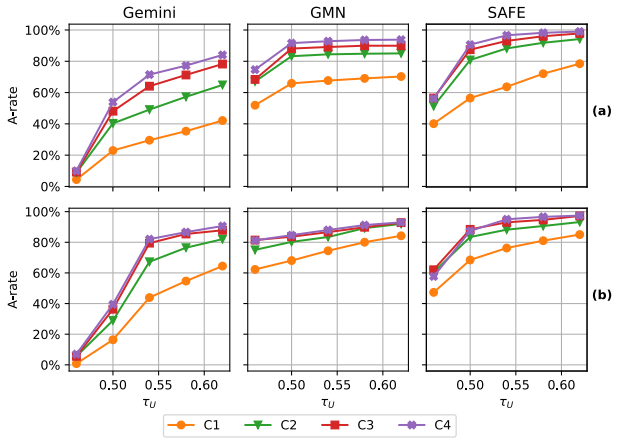

FIGURE 10. (a) Black-box untargeted attack with Spatial Greedy against the three target models while varying the success threshold $\tau_u \in [0.46, 0.62]$, and the settings **C1**, **C2**, **C3**, and **C4**. We use a set of candidates of 400 instructions, $\varepsilon = 0.1$, and $r = 0.75$. (b) White-box untargeted attack against the three target models while varying the success threshold $\tau_u \in [0.46, 0.62]$, and the settings **C1**, **C2**, **C3**, and **C4**. *Left*: GCAM attack with 40k iterations against GEMINI. *Center*: GCAM attack with 1k iterations against GMN. *Right*: GCAM attack with 1k iterations against SAFE.

and the attack success rate **A-rate**. When considering setting **C1**, the **A-rate** for $\tau_t = 0.80$ is 24.35% for Gemini, and 11.57% for SAFE; moving to **C4**, it increases up to 31.60% for Gemini, and 21.76% for SAFE. On the contrary, GMN does not show a monotonic **A-rate** increase for an increasing B value, as the peak **A-rate** is 38.32% in setting **C2**.

We now discuss the modification size **M-size** metric: fixing $\tau_t = 0.80$ and considering the setting where **A-rate** peaks, we measure an **M-size** value of 38.90 for SAFE (**C4**), 133.84 for Gemini (**C4**), and 350.50 for GMN (**C2**): SAFE is the model that sees the insertion of fewer instructions. This is not surprising: due to the feature-space representation of SAFE, the embeddings we alter in the attack for it (Section VIII-C2) refer to a number of instructions that is fixed.

4) WHITE-BOX UNTARGETED ATTACK

Figure 10 and Table 5 report the results for our attacks with white-box knowledge in the untargeted scenario.

TABLE 4. Evaluation metrics with $\tau_t = 0.80$ for the white-box targeted attack against the three target models. The GCAM attack is executed up to 20k iterations for Gemini and up to 1k for GMN and SAFE.

	Target	A-rate (%) ($\tau_t = 0.80$)				M-size ($\tau_t = 0.80$)				A-sim ($\tau_t = 0.80$)				N-inc ($\tau_t = 0.80$)			
		C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4
Targ	Gemini	24.35	29.94	30.94	**31.60**	53.67	86.53	111.83	133.84	0.85	0.86	0.86	0.86	0.54	0.60	0.63	0.62
	GMN	34.73	**38.32**	34.93	35.33	212.79	350.5	439.03	461.08	0.85	0.84	0.84	0.84	0.78	0.78	0.77	0.79
	SAFE	11.57	18.96	20.36	**21.76**	18.62	31.87	38.27	38.90	0.84	0.85	0.85	0.85	0.67	0.71	0.70	0.71

TABLE 5. Evaluation metrics with $\tau_u = 0.50$ for the white-box untargeted attack against the three target models. The GCAM attack is executed up to 20k iterations for Gemini and up to 1k for GMN and SAFE.

	Target	A-rate (%) ($\tau_t = 0.50$)				M-size ($\tau_t = 0.50$)				A-sim ($\tau_t = 0.50$)				N-dec ($\tau_t = 0.50$)			
		C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4	C1	C2	C3	C4
Untarg	Gemini	16.37	28.94	36.32	**39.52**	46.63	79.76	103.86	117.21	0.48	0.48	0.48	0.47	0.51	0.52	0.52	0.53
	GMN	68.06	80.24	83.63	**84.63**	298.34	541.32	718.57	859.53	0.23	0.19	0.18	0.18	0.78	0.81	0.82	0.82
	SAFE	68.46	83.43	**88.42**	87.42	17.52	30.01	36.0	38.49	0.39	0.38	0.39	0.40	0.56	0.58	0.58	0.58

Gemini looks more robust than the other models: for example, fixing $\tau_u = 0.50$, we measure the highest attack success rate **A-rate** as 39.52% in the **C4** setting. On the contrary, for the same τ_u, the highest **A-rate** for SAFE is 88.42% (setting **C3**) and 84.63% for GMN (setting **C4**).

The general trend of having a positive correlation of B and the **A-rate** is still observable (with a sharp increase of the **A-rate** from setting **C1** to **C2**). The **M-size** shows that SAFE is the most fragile model in terms of instructions to add, as they are much fewer than with the other two models.

5) GREEDY VS. SPATIAL GREEDY

We now compare the performance of Spatial Greedy against the Greedy baseline, until now left out of our discussions for brevity. Figure 11 shows the results for a targeted attack on the Targ. Additional data points are available in Table 2.

We discuss Gemini and GMN first. We recall that we could exploit their feature extraction process to reduce the size of the set of candidates, devising a gray-box Greedy procedure. Spatial Greedy is instead always black-box.

Considering the **A-rate** at $\tau_t = 0.80$, Spatial Greedy always outperform the gray-box baseline, except for setting **C4** on Gemini (although the two perform similarly: 27.94% for Greedy and 27.56% for Spatial Greedy). Looking at the other metrics, we can see that our black-box approach based on instructions embeddings is almost on par or improves on the results provided by the gray-box baseline.

Moving to SAFE, we recall that only a black-box Greedy is feasible. Considering the **A-rate**, we can notice that increasing both $\bar{\delta}$ and B produces a more noticeable difference between the baseline technique and Spatial Greedy. In the **C1** setting, the **A-rate** at $\tau_t = 0.80$ is 34.33% for Greedy and

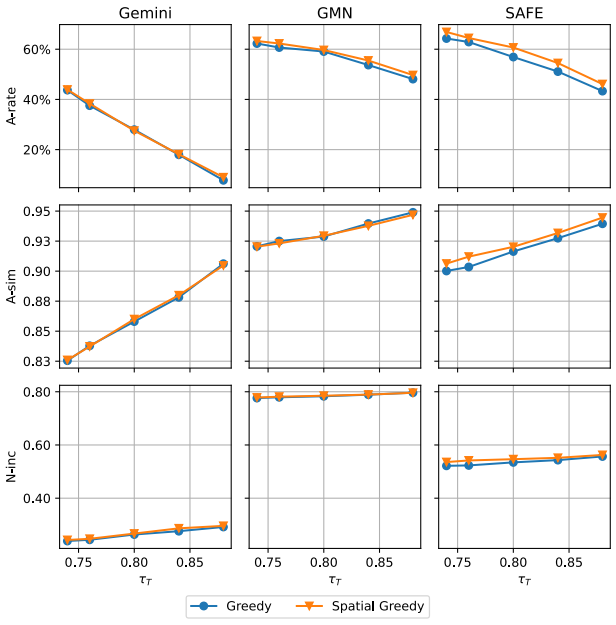

FIGURE 11. Greedy and Spatial Greedy targeted attacks against the three models while varying the success threshold $\tau_t \in [0.74, 0.88]$, considering the setting C4. For both, we consider $\varepsilon = 0.1$ and $|\mathrm{CAND}| = 400$. For Spatial Greedy, we also set $r = 0.75$.

37.13% for Spatial Greedy; then, it increases up to 56.89% for Greedy and 60.68% for Spatial Greedy when considering the **C4** scenario.

The other metrics confirm this behavior. Considering the average similarity **A-sim**, regardless of the chosen $\bar{\delta}$ and B from the setting, we observe that adversarial pairs generated through Spatial Greedy present a final average similarity that is higher than the one relative to the pairs generated using the baseline solution. The effectiveness of Spatial Greedy is

further confirmed by the normalized increment **N-inc** metric; at a comparison of the results, the impact of the candidates selected using Spatial Greedy is more consistent if compared to the one of the candidates selected using the baseline approach. We omit a discussion of the untargeted case for brevity.

Comparing Spatial Greedy with Greedy, we measure on the Targ dataset an average **A-rate** increase of 2.27 and a decreased **M-size** by 0.46 instructions across all configurations and models. When considering the Untarg, Spatial Greedy sees an average A-rate increase of 1.75, whereas the average M-size is smaller by 0.16 instructions. We report detailed results in Table 6.

Restricted Set Experiments: Finally, we perform a further experiment between the black-box version of Greedy and Spatial Greedy, considering a candidates' set of smaller size; in particular, we consider a set of 50 instructions which, in the case of Greedy, is a subset of the one considered for the previously detailed experiments. Our hypothesis is that the smaller the size of the candidates' set the higher the difference in terms of **A-rate** in favour of Spatial Greedy. We highlight that we applied the black-box version of Greedy also when targeting Gemini and GMN. In the following, we refer to the results obtained in the targeted case when considering the **C4** scenario.

When targeting SAFE, there is a significant difference, with an **A-rate** of 30.74% for Greedy and 49.70% for Spatial Greedy. A similar trend is seen with Gemini, where Greedy shows an **A-rate** of 9.58% compared to 25.55% for Spatial Greedy, and with GMN, where Greedy's **A-rate** is 37.13% versus 49% for Spatial Greedy.

> **Takeaway:** Spatial Greedy is typically superior, and always at least comparable, to a Greedy attack even when an efficient gray-box Greedy variant is possible. The results suggest that our dynamic update of the set of candidates, done at each iteration of the optimization procedure, can lead to the identification of new portions of the instruction space (and, consequently, a new subset of the ISA) that can positively influence the attack results. Finally, the smaller the size of the candidates' set the higher the effectiveness of Spatial Greedy compared to Greedy.

C. RQ1: TARGETED VS. UNTARGETED ATTACKS

From the previous sections, the attentive reader may have noticed that all our approaches are much more effective in an **untargeted** scenario for all models and proposed metrics.

When looking at the **A-rate** for all thresholds of similarities, the three target models are less robust against untargeted attacks (rather than targeted ones) regardless of the adversarial knowledge. For the best attack among black-box and white-box configurations, in the targeted scenario, the peak **A-rate** at $\tau_t = 0.80$ is 27.54% for Gemini, 59.68% for GMN, and 60.68% for SAFE. For the untargeted scenario, the peak **A-rate** at $\tau_u = 0.50$ is 53.89% for Gemini, 91.62% for GMN, and 90.62% for SAFE.

The number of instructions **M-size** needed for generating valid adversarial examples further confirms the weak resilience of the target models to untargeted attacks. When considering the worst setting according to **M-size** (i.e., **C4**), while we need only few instructions for untargeted attacks at $\tau_u = 0.50$ (i.e., 11.35 for Gemini, 4.14 for GMN, and 7.64 for SAFE), we need a significantly higher number of added instructions for targeted attacks (i.e., 51.85 for Gemini, 40.99 for GMN, and 40.31 for SAFE) at $\tau_t = 0.80$.

> **Takeaway:** On all the attacked models, both targeted and untargeted attacks are feasible, especially using Spatial Greedy (see also RQ2). Their resilience against untargeted attacks is significantly lower.

D. RQ2: BLACK-BOX VS. WHITE-BOX ATTACKS

An interesting finding from our tests is that the white-box strategy does not always outperform the black-box one.

Figure 12 depicts a comparison in the targeted scenario between Spatial Greedy and GCAM for the attack success rate **A-rate**, average similarity **A-sim**, and normalized increment **N-inc** metrics. The figure shows how different values of the success attack threshold τ_t can influence the considered metrics. On GMN and SAFE, Spatial Greedy is

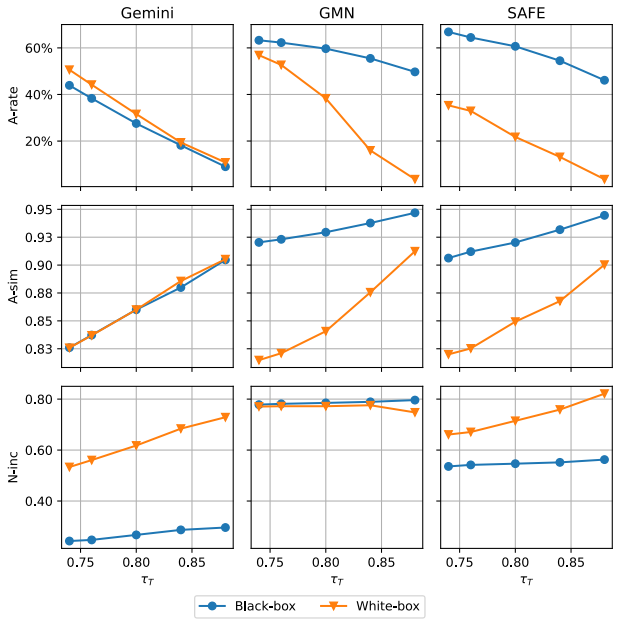

FIGURE 12. **Black-box and white-box targeted attacks against the three models while varying the success threshold $\tau_t \in [0.74, 0.88]$. In the black-box scenario, all the results refer to the Spatial Greedy approach ($\varepsilon = 0.1$, $r = 0.75$, and $|\text{CAND}| = 400$). In the white-box scenario, the results for Gemini are for a GCAM attack with 20k iterations while the ones for SAFE and GMN are for a GCAM attack with 1k iterations. We consider all approaches in their most effective parameter choice, being it always setting C4 except for the GCAM attack against GMN, for which we consider setting C2.**

TABLE 6. Difference between SG and G for the A-rate and M-size in the four settings C1-C4, averaged on the three models. Where applicable, we consider the gray-box version of Greedy.

	C1		C2		C3		C4	
	A-rate	M-size	A-rate	M-size	A-rate	M-size	A-rate	M-size
Targ	2.65	-1.74	2.08	0.12	3	0.03	1.33	-0.23
Untarg	2.87	0.29	2.04	-0.18	0.93	-0.26	1.17	-0.47

more effective than GCAM, resulting in significantly higher A-rate values, while the two perform similarly on Gemini.

Interestingly, in contrast with the evaluation based on the **A-rate** metric, both the **A-sim** and **N-inc** values highlight a coherent behavior among the three target models. Generally, adversarial examples generated using Spatial Greedy exhibit a higher **A-sim** value than the white-box ones (considering $\tau_t = 0.80$, we have 0.86 vs. 0.86 for Gemini, 0.93 vs. 0.84 for GMN, and 0.92 vs. 0.85 for SAFE). Looking at **N-inc**, we face a completely reversed situation; the metric is better in the adversarial samples generated using GCAM (0.62 for Gemini, 0.79 for GMN, and 0.71 for SAFE) compared to those from Spatial Greedy (0.27 for Gemini, 0.79 for GMN, and 0.55 for SAFE). These two observations lead us to the hypothesis that the black-box attack is more effective against pairs of binary functions that exhibit high initial similarity values and can potentially reach a high final similarity. On the other side, GCAM is particularly effective against pairs that are very dissimilar at the beginning.

For the untargeted scenario, our results (Tables 3 and 5) for the **A-rate** metric considering $\tau_u = 0.50$ show that Spatial Greedy has a slight advantage on GCAM. For Spatial Greedy, we have best-setting values of 53.89% for Gemini, 91.62% for GMN, and 90.62% for SAFE; for GCAM, we have 39.52% for Gemini, 84.63% for GMN, and 88.42% for SAFE.

In our experiments, GCAM performed worse than the black-box strategy, which may look puzzling since theoretically a white-box attack should be more potent than a black-box one. We believe this behavior is due to the inverse feature mapping problem. Hence, we conducted a GCAM attack exclusively in the feature space by eliminating all constraints needed to identify a valid potential sample in the problem space (i.e., non-negativity of coefficients for Gemini and GMN, rounding to genuine instruction embeddings for SAFE). As a result, GCAM achieved an **A-rate** between 92.90% and 99.81% in targeted scenarios and between 97.01% and 100% in untargeted ones.

Takeaway: In our tests, the Spatial Greedy black-box attack is on par or beats the white-box GCAM attack based on a rounding inverse strategy. Further investigation is needed to confirm if this result will hold for more refined inverse feature mapping techniques and when attacking other models.

E. RQ3: IMPACT OF FEATURES EXTRACTION AND ARCHITECTURES ON ATTACKS

As detailed in Section VIII, we can distinguish the target models according to three aspects (NN architecture, input representation, and feature mapping process). Here, we are interested in investigating whether these aspects can influence the performance of our attacks or not.

In the targeted black-box scenario (Figure 9 and Table 2), SAFE and GMN are the weakest among the three considered models, as the peak attack success rate **A-rate** at $\tau_t = 0.80$ is 60.68% for SAFE, 59.68% for GMN, and 27.54% for Gemini (**C4** setting). These results highlight that our attack is sensible to some aspects of the target model, in particular to the feature mapping process and the DNN architecture employed by the considered models. To assess this insight, we conduct different analyses to check whether our Spatial Greedy attack is exploiting some hidden aspects of the considered models to guide the update of the candidates' set. First, we check whether or not there exists a correlation between the number of initial instructions composing the function f_1 and the obtained final similarity value. This analysis is particularly interesting for SAFE, as this model computes the similarity between two functions by only considering their first 150 instructions; the results of this study are reported in Figure 13. From the plots it is visible that there exists a negative correlation between the final similarity and the initial number of instructions composing the function we are modifying, also confirmed by the Pearson's r correlation coefficient highlighting that this negative correlation is almost moderate for SAFE (with $r = -0.38$) while it is weak for both Gemini and GMN (with $r = -0.25$ and $r = -0.22$ respectively). These results confirm that when Spatial Greedy modifies a function that is initially small (particularly composed by less than 150 instructions), then our adversarial example and the function f_2 are more likely to have a final similarity value near 1 when targeting SAFE rather then the two other models.

Then, since both Gemini and GMN implements a feature mapping function which deeply looks at the particular assembly instructions composing the single blocks, we conduct a further analysis to assess whether or not the instructions inserted by Spatial Greedy trigger the features required by the two considered models. In particular, for each inserted instruction, we check whether or not it is mapped over the features considered by the two models and, for each adversarial example, we calculate the percentage of instructions

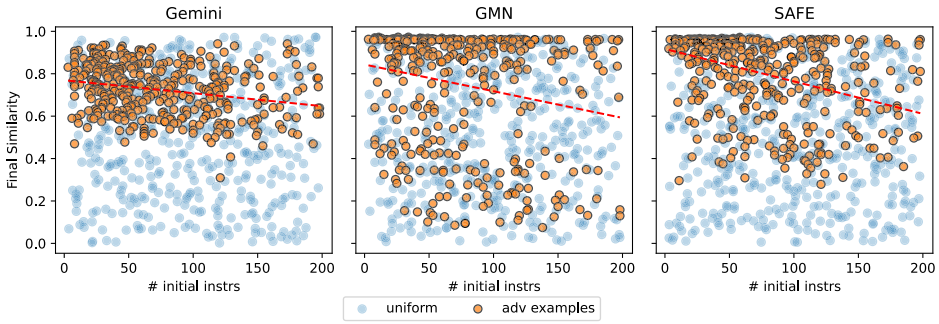

FIGURE 13. Correlation between initial number of instructions of the function f_1 and the similarity between the generated adversarial example f_{adv} and the target function f_2. The considered adversarial examples are generated in the targeted scenario, using our Spatial Greedy approach ($\varepsilon = 0.1$, $r = 0.75$, and $|\text{CAND}| = 400$) in setting C4. We also reported an equal number of samples randomly drawn from a uniform distribution.

satisfying this condition. In the targeted black-box scenario using Spatial Greedy, we find that on average, 90.83% of the inserted instructions for Gemini and 100% for GMN are mapped to the considered features.

To verify how the particular architecture implemented by the model affects the performance of Spatial Greedy, we checked how the instructions inserted by our procedure are distributed across the various dead branches. Our hypothesis is that when targeting GNN-based models (as Gemini and GMN), our attack should span the inserted instructions across the various dead branches; on the contrary, the position of the block should not influence the choice of the attack when targeting a RNN as SAFE. For all the considered models, the block where our attack inserts the majority of the instructions for each adversarial example is the one closest to beginning of the function. However, while this is evident for GMN and SAFE (where the first block contains most of the inserted instruction in 313 and 205 of the considered examples respectively), in Gemini the inserted instructions are more uniformly distributed across the various dead branches. We report the complete distribution in Figure 14. To further validate these results, we calculated the entropy of the generated adversarial examples, resulting in values of 2.94 for Gemini, 2.77 for GMN, and 2.68 for SAFE. Higher entropy suggests a more uniform distribution of inserted instructions across dead branches, while lower values indicate concentration in specific blocks. These entropy values reinforce our previous conclusions. We highlight that these results are partially coherent with our initial hypothesis; indeed, the first block is the closest to the prologue of the function, which plays a key role for both SAFE and GMN. Indeed, as mentioned in [44], SAFE primarily targets function prologues, which explains why our attack inserts most instructions into the first block, as it is closest to the function prologue; For GMN, since prologue instructions typically follow specific compiler patterns, the nodes containing these instructions are likely to match, prompting Spatial Greedy to insert most instructions into the dead branches closest to the prologue.

The small **M-size** results in the untargeted scenario prevent us to obtain meaningful results when running these analyses in this context, so we decided to not report the obtained results.

For the white-box attack, we believe that the different levels of robustness among the considered models are mainly due to their feature mapping processes. As mentioned in Section X-D, we evaluated a variant of the GCAM attack solely in the feature space, removing all constraints necessary for producing adversarial examples valid in the problem space. For both Gemini and GMN, we removed the non-negativity constraint of coefficients, and for SAFE, we eliminated the rounding to real instruction embeddings constraint. In the **C4** targeted scenario, the unconstrained version of GCAM increases the **A-rate** of the standard GCAM on Gemini from 31.60% to 96%, on GMN from 35.33% to 99.81%, and on SAFE from 21.76% to 92.90%. This demonstrates that the performance of our attack is primarily influenced by the feature mapping method rather than the specific model architectures. The results on the untargeted scenario confirm this hypothesis, as the unbounded GCAM reaches an **A-rate** near 100% for both GMN and SAFE, while a value of 97.01% against Gemini.

> **Take away:** When considering the black-box scenario, the particular architecture seems to influence the position where the instructions are inserted. In general, the particular feature mapping process adopted by the models seems playing a crucial role in the choice of the instructions.
> In the white-box scenario, the feature mapping processes adopted by the models prevent in reaching optimal **A-rate** results.

XI. MIRAI CASE STUDY

We complement our evaluation with a case study examining our attacks in the context of disguising functions from malware.

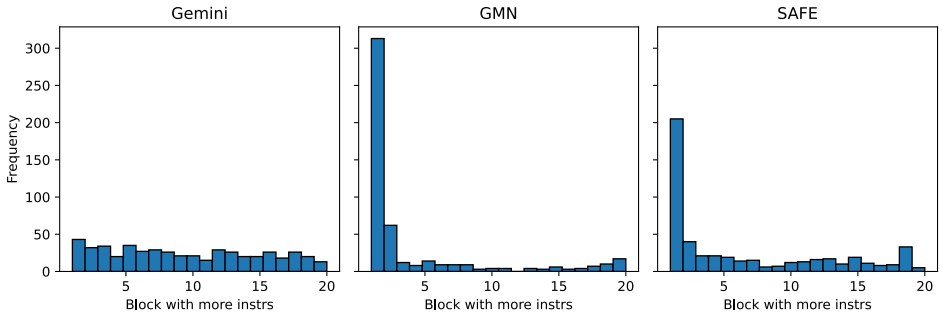

FIGURE 14. Distribution of the blocks where, for each adversarial example (successful or not), our Spatial Greedy attack inserts most of the instructions. The considered adversarial examples are generated in the targeted scenario, using our Spatial Greedy approach ($\varepsilon = 0.1$, $r = 0.75$, and $|\texttt{CAND}| = 400$) in setting C4.

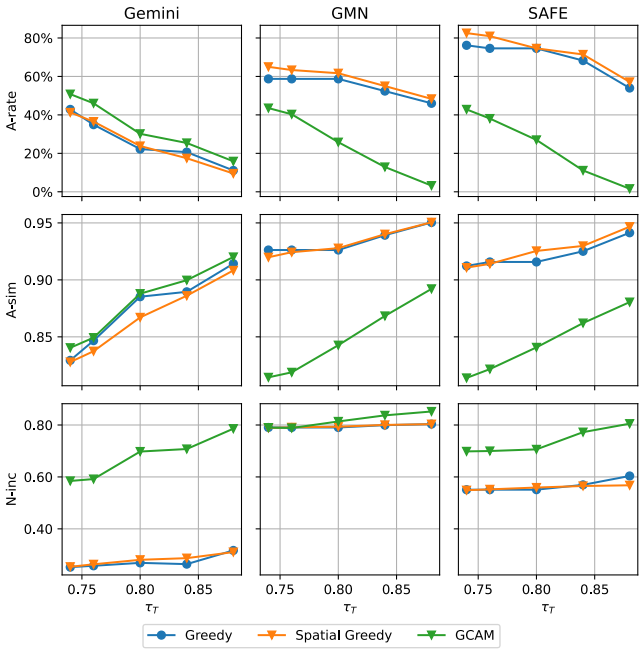

FIGURE 15. Experiments on the three models subject of black-box and white-box attackers in the targeted scenario, on the Mirai dataset for a different success threshold $\tau_t \in (0.74, 0.88)$ in setting C4. In case of black-box attacker, we test the Spatial Greedy approach against the target models with $\varepsilon = 0.1$, $r = 0.75$, $|\texttt{CAND}| = 400$. In case of white-box attacker, we test GCAM attack with 20k iterations against GEMINI, with 1k iterations against GMN, and with 1k iterations against SAFE.

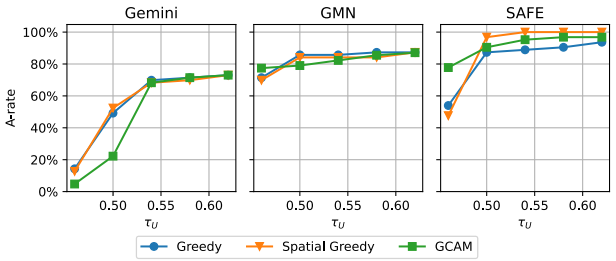

FIGURE 16. Resilience of the three models to black-box and white-box attackers in the untargeted scenario, on the Mirai dataset for a different success threshold $\tau_u \in [0.46, 0.62]$, considering the setting C4. In case of black-box attacker, we test the Spatial Greedy approach against the target models with $\varepsilon = 0.1$, $r = 0.75$, $|\texttt{CAND}| = 400$. In case of white-box attacker, we test GCAM attack with 20k iterations against GEMINI, with 1k iterations against GMN, and with 1k iterations against SAFE.

We consider the code base from a famous leak of the Mirai malware, compiling it `gcc` 9.2.1 with `-O0` optimization level on Ubuntu 20.04. After filtering out all functions with less than six instructions, we obtain a set of 63 functions. We build distinct datasets for the targeted and untargeted case. For the former, we pair malicious Mirai functions with benign ones from the Targ dataset from the main evaluation. For the latter, each of the 63 functions is paired with itself.

Figure 15 reports on our targeted attacks, comparing Greedy, Spatial Greedy, and the white-box GCAM for the metrics of **A-rate**, **A-sim** and **N-inc**. For brevity, we focus on the performant **C4** configuration from the main evaluation.

For the **A-rate**, when attacking GMN and SAFE, Spatial Greedy has an edge on both Greedy and GCAM, with the latter performing markedly worse than the two black-box ones. With Gemini, Spatial Greedy and Greedy perform similarly, with both resulting below GCAM. This behaviour is consistent with the main evaluation results (cf. Figure 12).

In more detail, with GMN, the average increase of **A-rate** for Spatial Greedy over Greedy is 3.73 (max. of 6.27 at $\tau_t = 0.74$, min. of 2.27 at $\tau_t = 0.88$). With SAFE, this increase is 3.81 (max. of 6.35 at $\tau_t = 0.74$; min. of zero at $\tau_t = 0.8$). With Gemini, GCAM is the best attack with an average 7.94 increase over Spatial Greedy (max. of 9.52% at $\tau_t = 0.74$; min. of 6.35% at $\tau_t = 0.88$). SAFE remains the easiest model to attack also on this dataset.

Regarding **A-sim** and **N-inc**, Spatial Greedy and Greedy perform similarly on GMN and SAFE, whereas on Gemini Spatial Greedy is slightly worse than Greedy for **A-sim** at lower thresholds. The relative performance of GCAM vs. the black-box attacks resembles the trends discussed in the main evaluation (cf. Figure 12).

Figure 16 reports on the experiments we conducted for the untargeted scenario. We note that Spatial Greedy outperforms the other attacks on SAFE (with the exception of GCAM when τ_u =0.46) and performs analogously to them on the

other two models. Compared to the main evaluation results, targeted attacks have worse performance than untargeted ones also on this dataset. Moreover, successful untargeted attacks continue to require fewer instructions: in particular, across all models, a successful black-box targeted attack needs on average 42.63 instructions, whereas the untargeted one adds on average 5.27 instructions.

XII. PRACTICAL IMPACTS AND POSSIBLE COUNTERMEASURES

In this section we discuss the practical impacts of our paper and possible countermeasures.

A. PRACTICAL IMPACTS

The findings in Section X-B reveal that the evaluated binary similarity systems are susceptible to both targeted and untargeted attacks, though their resilience differs. These systems show higher robustness against targeted attacks, with an average **A-rate** of 49.43%, whereas the average **A-rate** for untargeted attacks is 79.44%. From a practical perspective, as detailed in Section I we can consider the three main uses cases of binary similarity systems: vulnerability detection, plagiarism detection, and malware detection. The results in the untargeted scenario imply that when having an attacker that is trying to substitute a function with an older, vulnerable version or make a plagiarized function dissimilar to the original one, then they succeed in nearly 80% of cases. This suggests that current binary similarity models are unfit for tasks such as vulnerability detection or authorship identification when used in a context that could be subject to adversarial attacks (as example, but not limited to, when used in security sensitive scenarios). To remark on this our results in Section XI practically show that, when an attacker creates a new variant of a malicious function without targeting any specific benign function, then the models fail in recognising it as similar to any known malicious sample in nearly 78% of the cases. In contrast, the considered models show greater resistance when the attacker is trying to create a variant of its input matching a specific target function. This implies that the considered models are more resistant when facing an attacker trying to make a malicious function closely resemble a specific whitelisted function rather than when the attacker is hiding the malevolent function. However, it is important to note that even in this scenario, as reported in Section XI, an attacker can bypass the binary similarity detection system in more than half of the cases.

B. COUNTERMESURES

Given these results, even though our primary focus has been on the attack side, it is important to investigate potential defensive strategies.

A typical approach consists of using a classifier as detector to distinguish between clean examples and adversarial ones. In our context, one could use an anomaly detection model to check whether the input function's code follows common patterns of compiler-generated code or not, using models for checking compiler provenance [45], [46].

Adversarial training [13], [47] is the standard solution for increasing the robustness of an already trained model; however, while it could improve the robustness against our methodologies, there is no guarantee that the retrained models would be robust against zero-day attacks. To overcome these limitations, techniques based on randomized defenses [48] could be considered. In particular [48] proposes a methodology to increase the robustness of DNN classifiers against adversarial examples by introducing random noise inside the input representation during both training and inference. While originally designed for the computer vision scenario, this method has been adapted to the malware classification domain, by randomly substituting [49] and deleting [50] bytes from the input sample. However, the applicability of these approaches in the binary similarity domain has not been studied yet and must focus on manipulating directly assembly instructions or CFG nodes.

A more promising approach consists of analyzing only a subset of the instructions from the input functions; the rationale is that this could thwart the attack by partially destroying the pattern of instructions introduced by the adversary. Similarly to [51], one could learn the function representation by focusing only on some random portions of the input. A more refined approach could consist of filtering out instructions using techniques such as data-flow analysis and micro-trace execution, to concentrate solely on the ones with the highest semantic importance. However, one has to keep in mind that refined analyses at the pre-processing stage could introduce significant delays that would partially nullify the speed advantages of using DNNs solutions over symbolic execution ones.

Finally, one could use an ensemble of all the target models combined with a majority voting approach to determine the final similarity. As discussed in the evaluation, the various attacked models respond differently to our attacks. This suggests that an ensemble model could be a feasible defense.

XIII. LIMITATIONS AND FUTURE WORKS

In this paper, we have seen how adding dead code is a natural and effective way to realize appreciable perturbations for a selection of heterogeneous binary similarity systems.

In Section IV-C, we acknowledged how, in the face of defenders that pre-process code with static analysis, our implementation would be limited from having the inserted dead blocks guarded by non-obfuscated branch predicates. Furthermore, we highlight that all the approaches we propose consist of inserting into dead branches sequences of instructions that do not present any data-dependency, which make them easier to detect.

Our experiments suggest that, depending on the characteristics of a given model and pair of functions, the success of an attack may be affected by factors like the initial difference in code size and CFG topology, among others. In this respect, it could be interesting to explore how to

alternate our dead-branch addition perturbation, for example, with the insertion of dead fragments within existing blocks.

We believe both limitations could be addressed in future work with implementation effort, whereas the main goal of this paper was to show that adversarial attacks against binary similarity systems are a concrete possibility. To enhance our attacks, we could explore more complex patching implementation strategies based on binary rewriting or a modified compiler back-end. Such studies may then include also other performant similarity systems, such as Asm2Vec [8] or jTrans [52].

XIV. CONCLUSION

We presented the first study on the resilience of code models for binary similarity to black-box and white-box adversarial attacks, covering targeted and untargeted scenarios. Our tests highlight that current state-of-the-art solutions in the field (Gemini, GMN, and SAFE) are not robust to adversarial attacks crafted for misleading binary similarity models. Furthermore, their resilience against untargeted attacks appears significantly lower in our tests. Our black-box Spatial Greedy technique also shows that an instruction-selection strategy guided by a dynamic exploration of the entire ISA is more effective than using a fixed set of instructions. We hope to encourage follow-up studies by the community to improve the robustness and performance of these systems.

ACKNOWLEDGMENT

This work has been carried out while Gianluca Capozzi was enrolled in the Italian National Doctorate on Artificial Intelligence run by Sapienza University of Rome.

REFERENCES

[1] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (English version)," in *Proc. Symp. sur la sécurité des Technol. de l'information et des Commun. (SSTIC)*, 2005, vol. 5, no. 1, p. 3.

[2] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *Proc. 10th Work. Conf. Mining Softw. Repositories (MSR)*, May 2013, pp. 329–338.

[3] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "SIGMA: A semantic integrated graph matching approach for identifying reused functions in binary code," *Digit. Invest.*, vol. 12, pp. S61–S71, Mar. 2015.

[4] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Function representations for binary similarity," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 4, pp. 2259–2273, Jul. 2022.

[5] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2016, pp. 266–280.

[6] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2017, pp. 79–94.

[7] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *Proc. 23rd USENIX Secur. Symp. (SEC)*, 2014, pp. 303–317.

[8] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2 Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 472–489.

[9] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 363–376.

[10] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proc. 30th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2014, pp. 406–415.

[11] X. Yuan, P. He, Q. Zhu, and X. Li, "Adversarial examples: Attacks and defenses for deep learning," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 9, pp. 2805–2824, Sep. 2019.

[12] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," 2013, *arXiv:1312.6199*.

[13] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2014, *arXiv:1412.6572*.

[14] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 39–57.

[15] J. Li, S. Qu, X. Li, J. Szurley, J. Z. Kolter, and F. Metze, "Adversarial music: Real world audio adversary against wake-word detection system," in *Proc. 32nd Annu. Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2019, pp. 11908–11918.

[16] R. Jia and P. Liang, "Adversarial examples for evaluating reading comprehension systems," in *Proc. 22nd Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2017, pp. 2021–2031.

[17] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, "Intriguing properties of adversarial ml attacks in the problem space," in *Proc. 41st IEEE Symp. Secur. Privacy (SP)*, 2020, pp. 1332–1349.

[18] B. Devore-McDonald and E. D. Berger, "Mossad: Defeating software plagiarism detection," in *Proc. ACM Program. Lang. (OOPSLA)*, vol. 4, Jun. 2020, pp. 1–28.

[19] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," in *Proc. ACM Meas. Anal. Comput. Syst.*, 2020, vol. 4, no. 3, pp. 1–29.

[20] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. 27th Annu. Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2013, pp. 3111–3119.

[21] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," 2017, *arXiv:1706.06083*.

[22] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 3835–3845.

[23] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *Proc. 31st USENIX Secur. Symp. (SEC)*, 2022, pp. 2099–2116.

[24] N. Mrkšic, D. Ó Séaghdha, B. Thomson, M. Gašic, L. M. Rojas-Barahona, P.-H. Su, D. Vandyke, T.-H. Wen, and S. Young, "Counter-fitting word vectors to linguistic constraints," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Human Lang. Technol.*, 2016, pp. 142–148.

[25] S. Ren, Y. Deng, K. He, and W. Che, "Generating natural language adversarial examples through probability weighted word saliency," in *Proc. 57th Annu. Meeting Assoc. Comput. Linguistics*, 2019, pp. 1085–1097.

[26] D. Li, Y. Zhang, H. Peng, L. Chen, C. Brockett, M.-T. Sun, and B. Dolan, "Contextualized perturbation for textual adversarial attack," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Human Lang. Technol.*, 2021, pp. 5053–5069.

[27] L. Li, R. Ma, Q. Guo, X. Xue, and X. Qiu, "BERT-ATTACK: Adversarial attack against BERT using BERT," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2020, pp. 6193–6202.

[28] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," in *Proc. ACM Program. Lang. (OOPSLA)*, vol. 4, Jun. 2020, pp. 1–30.

[29] W. Zhang, S. Guo, H. Zhang, Y. Sui, Y. Xue, and Y. Xu, "Challenging machine learning-based clone detectors via semantic-preserving code transformations," *IEEE Trans. Softw. Eng.*, vol. 49, no. 5, pp. 3052–3070, May 2023.

[30] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, "Adversarial malware binaries: Evading deep learning for malware detection in executables," in *Proc. 26th Eur. Signal Process. Conf. (EUSIPCO)*, Sep. 2018, pp. 533–537.

[31] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet, "Adversarial examples on discrete sequences for beating whole-binary malware detection," 2018, *arXiv:1802.04528*.

[32] K. Lucas, M. Sharif, L. Bauer, M. K. Reiter, and S. Shintre, "Malware makeover: Breaking ML-based static analysis by modifying executable bytes," in *Proc. 16th ACM Asia Conf. Comput. Commun. Secur. (AsiaCCS)*, 2021, pp. 744–758.

[33] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, and H. Yin, "MAB-malware: A reinforcement learning framework for blackbox generation of adversarial malware," in *Proc. 17th ACM Asia Conf. Comput. Commun. Secur. (AsiaCCS)*, 2022, pp. 990–1003.

[34] L. Jia, B. Tang, C. Wu, Z. Wang, Z. Jiang, Y. Lai, Y. Kang, N. Liu, and J. Zhang, "FuncFooler: A practical black-box attack against learning-based binary code similarity detection methods," 2022, *arXiv:2208.14191*.

[35] B. Biggio and F. Roli, "Wild patterns: Ten years after the rise of adversarial machine learning," *Pattern Recognit.*, vol. 84, pp. 317–331, Dec. 2018.

[36] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida, "Constantine: Automatic side-channel resistance using efficient control and data flow linearization," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2021, pp. 715–733.

[37] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, "Stratified synthesis: Automatically learning the x86–64 instruction set," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Des. Implement. (PLDI)*, 2016, pp. 237–250.

[38] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *Proc. 26th USENIX Secur. Symp. (SEC)*, 2017, pp. 99–116.

[39] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proc. 33rd Int. Conf. Mach. Learn. (ICML)*, vol. 48, 2016, pp. 2702–2711.

[40] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proc. 19th Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2014, pp. 1532–1543.

[41] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Trans. Assoc. Comput. Linguistics*, vol. 5, pp. 135–146, Dec. 2017.

[42] C. Guo, J. R. Gardner, Y. You, A. G. Wilson, and K. Q. Weinberger, "Simple black-box adversarial attacks," in *Proc. 36th Int. Conf. Mach. Learn. (ICML)*, vol. 97, 2019, pp. 2484–2493.

[43] J. Chen, M. I. Jordan, and M. J. Wainwright, "HopSkipJumpAttack: A query-efficient decision-based attack," in *Proc. 41st IEEE Symp. Secur. Privacy (SP)*, 2020, pp. 1277–1294.

[44] W. K. Wong, H. Wang, Z. Li, and S. Wang, "BinAug: Enhancing binary similarity analysis with low-cost input repairing," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, vol. 9, Feb. 2024, pp. 1–13.

[45] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Proc. Workshop Binary Anal. Res.*, 2019, pp. 1–11.

[46] X. He, S. Wang, Y. Xing, P. Feng, H. Wang, Q. Li, S. Chen, and K. Sun, "BinProv: Binary code provenance identification without disassembly," in *Proc. 25th Int. Symp. Res. Attacks, Intrusions Defenses*, Oct. 2022, pp. 350–363.

[47] K. Lucas, S. Pai, W. Lin, L. Bauer, M. K. Reiter, and M. Sharif, "Adversarial training for raw-binary malware classifiers," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 1163–1180.

[48] J. Cohen, E. Rosenfeld, and J. Z. Kolter, "Certified adversarial robustness via randomized smoothing," in *Proc. 36th Int. Conf. Mach. Learn. (ICML)*, vol. 97, 2019, pp. 1310–1320.

[49] D. Gibert, G. Zizzo, and Q. Le, "Towards a practical defense against adversarial attacks on deep learning-based malware detectors via randomized smoothing," 2023, *arXiv:2308.08906*.

[50] Z. Huang, N. G. Marchant, K. Lucas, L. Bauer, O. Ohrimenko, and B. I. P. Rubinstein, "RS-Del: Edit distance robustness certificates for sequence classifiers via randomized deletion," in *Proc. 36th Annu. Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2023, pp. 1–36.

[51] D. Gibert, L. Demetrio, G. Zizzo, Q. Le, J. Planes, and B. Biggio, "Certified adversarial robustness of machine learning-based malware detectors via (De)Randomized smoothing," 2024, *arXiv:2405.00392*.

[52] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "JTrans: Jump-aware transformer for binary code similarity detection," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, 2022, pp. 1–13.

GIANLUCA CAPOZZI received the master's degree in engineering in computer science from the Sapienza University of Rome, Italy, in 2021, where he is currently pursuing the Ph.D. degree. His main research interest includes adversarial machine learning against neural network models for binary analysis.

DANIELE CONO D'ELIA received the Ph.D. degree in engineering in computer science from the Sapienza University of Rome, in 2016. He is currently a tenure-track Assistant Professor with the Sapienza University of Rome. His research activities span several fields across software and systems security, with contributions in the analysis of adversarial code and in the design of program analyses and transformations to make software more secure.

GIUSEPPE ANTONIO DI LUNA received the Ph.D. degree. After the Ph.D. study, he did a postdoctoral research with the University of Ottawa, Canada, working on fault tolerant distributed algorithms, distributed robotics, and algorithm design for programmable particles. In 2018, he started a postdoctoral research with Aix-Marseille University, France, where he worked on dynamic graphs. Currently, he is performing research on applying NLP techniques to the binary analysis domain. He is an Associate Professor with the Sapienza University of Rome, Italy.

LEONARDO QUERZONI received the Ph.D. degree with a thesis on efficient data routing algorithms for publish/subscribe middleware systems, in 2007. He is a Full Professor with the Sapienza University of Rome, Italy. He has authored more than 80 papers published in international scientific journals and conferences. His research interests range from cyber security to distributed systems, in particular binary similarity, distributed stream processing, dependability, and security in distributed systems. In 2017, he received the Test of Time Award from the ACM International Conference on Distributed Event-Based Systems for the paper TERA: Topic-Based Event Routing for Peer-to-Peer Architectures, published, in 2007.

Open Access funding provided by 'Università degli Studi di Roma La Sapienza 2' within the CRUI CARE Agreement