On the Lack of Robustness of Binary Function Similarity Systems

Gianluca Capozzi[1], Tong Tang[2], Jie Wan[2], Ziqi Yang[2,3], Daniele Cono D'Elia[1], Giuseppe Antonio Di Luna[1], Lorenzo Cavallaro[4], and Leonardo Querzoni[1]

[1]Sapienza University of Rome, Italy, {capozzi,delia,diluna,querzoni}@diag.uniroma1.it
[2]The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, {tong.tang, wanjie, yangziqi}@zju.edu.cn
[3]Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, China
[4]University College London, UK, l.cavallaro@ucl.ac.uk

Abstract

Binary function similarity, which often relies on learning-based algorithms to identify what functions in a pool are most similar to a given query function, is a sought-after topic in different communities, including machine learning, software engineering, and security. Its importance stems from the impact it has in facilitating several crucial tasks, from reverse engineering and malware analysis to automated vulnerability detection. Whereas recent work cast light around performance on this long-studied problem, the research landscape remains largely lackluster in understanding the resiliency of the state-of-the-art machine learning models against adversarial attacks. As security requires to reason about adversaries, in this work we assess the robustness of such models through a simple yet effective black-box greedy attack, which modifies the topology and the content of the control flow of the attacked functions. We demonstrate that this attack is successful in compromising all the models, achieving average attack success rates of 57.06% and 95.81% depending on the problem settings (targeted and untargeted attacks). Our findings are insightful: top performance on clean data does not necessarily relate to top robustness properties, which explicitly highlights performance-robustness trade-offs one should consider when deploying such models, calling for further research.

Keywords: Adversarial machine learning, binary analysis, binary function similarity

1 Introduction

A fruitful and long-standing research trend involves applying Deep Neural Networks (DNNs) to solve binary analysis problems. These solutions typically provide end-to-end capabilities for handling complex tasks across entire binaries (prototypical examples include malware/benign classification solutions). More recently, the focus has narrowed to specific binary analysis challenges that could immediately assist a reverse engineer, such as decompiling binary functions [8], identifying the signature and boundaries of a function [2,36], and detecting the toolchain used to generate a specific binary [32].

The binary function similarity problem

Among these tasks, one that has been predominately studied involves identifying when two binary functions are obtained from the same source code compiled with different compilers or optimization flags. This is known in the literature as the binary function similarity problem [1, 13, 18, 23]. This problem plays a key role in several security-sensitive scenarios [31,33,46], and is especially effective in detecting previously analyzed functions using a reference database. This includes challenges such as identifying known library functions in statically linked stripped binaries, recognizing specific malware functionalities (e.g., by recognizing a particular crypto routine, or clustering malware into families and lineage trees), detecting known vulnerabilities in binaries and firmware, and identifying copyright infringement cases in compiled binaries.

All binary function similarity models take a pair of functions as input and output a similarity score, that ranges from a minimum to a maximum value. Even if the models are trained using the strict definition of similarity described above, it has been observed that high similarity scores are also given to functions derived from source codes that are different but semantically similar. This characteristic is indeed desirable as it can be used to cluster semantically similar functions.

The gold standard for testing binary function similarity solutions uses them in the **function search** problem [16, 40, 44], where a query function f_Q is used to order a pool of functions P according to their similarity score from f_Q, where P can contain one or more functions f_v similar to f_Q. The problem is correctly solved when f_v is among the top-K similar functions in the order induced by the similarity score.

Given its central importance, the binary function similarity problem has become a hot topic of research with various solutions, mainly based on DNNs, proposed in the last four years [3,16,17,28,29,33,38,40,41,44,46]. To bring order to the plethora of proposed systems tested with varying performance measures, a 2022 paper by Marcelli et al. [31] evaluated several solutions using a common dataset. This step represented the first attempt to systematize a still-growing and fascinating field (since 2022, other binary similarity models have been proposed [40,44]).

The missing piece of the puzzle: robustness

While [31] systematically evaluates many different systems under several aspects, it never assesses the *robustness* of their underlying models.

A key weakness of machine learning solutions, especially those based on DNNs, is their performance when processing *adversarial examples* [48]. It is well-known that systems classifying media content (images, text, video, or audio) can be fooled by crafted examples obtained by modifying a benign one. Although the literature on adversarial examples is well-established for these models [5,10,20,22,26], its investigation into systems that analyze binaries is still in the early stages, with the majority of works focused on fooling malware classifiers [14,30,39].

At the current state, it is unclear what would be the resiliency of the binary similarity models benchmarked in [31] against adversarial examples. Intuitively, the ease of generating such examples for an adversary directly impacts the reliability of these systems. Hence, an extensive evaluation of their robustness is necessary to expose any inherent weaknesses undermining their practical value.

Our robustness evaluation

In this paper, we aim to close this gap by investigating the robustness of binary function similarity models. We adopt a black-box approach, motivated by the objective of testing the models against the weakest possible adversary. In keeping with this spirit, we have decided to use a basic framework for our attack—a greedy approach—which we have extended with a few refinements: a black-box importance mechanism to decide which part of the function to modify, and an embedding-based mechanism for sequences of instructions to guide the content of certain transformations.

In this paper, we consider an attacker aiming to compromise a binary function similarity system used for function search by reducing its ability to search for variants of a specific query function that they are altering. The adversary can execute *targeted* and *untargeted* attacks. In a targeted attack, given a query function f_Q and a set V of functions that are semantically similar to each other, the attacker seeks to generate from f_Q a semantically equivalent function f_{adv} maximizing its similarity with the functions in V. The untargeted attack is dual; here, given a query function f_Q and a set V of functions semantically equivalent to f_Q, the attacker seeks to generate from f_Q a function f_{adv} semantically similar to f_Q that minimizes the similarity with the functions in V.

We selected eight models, chosen from those used in [31] and other more recent and promising ones, based on criteria of scalability and diversity. That is, the models must be scalable and thus usable in real-world settings, and they must cover a broad spectrum of potential characteristics of similarity models. These include, for example, manual and automatic features, different neural architectures (i.e., RNNs, feedforwards, and GNNs), models trained with and without execution information and with or without obfuscated samples. The diversity of models ensured during the selection process makes the evaluation in our paper generalizable, as the trends observed in our evaluation are likely to hold for other models that share the structure of some of those we tested.

The selected models have been tested against targeted and untargeted attacks using the black-box methodology described above, and their robustness has been evaluated according to the primary metric of the Attack Success Rate (**ASR**).

1.1 Contributions

In this paper, we assess the robustness of eight binary function similarity models—Gemini [46], GMN [29], ZEEK [41], BinFinder [40], SAFE [33], jTrans [44], Trex [38], and PalmTree [28]— using a simple black-box greedy attack. This attack leverages four semantics-preserving transformations that alter the topology and the content of the control flow graph of the attacked query function.

This paper proposes the following contributions:

- Robustness analysis. We assess the robustness of the examined models against targeted and untargeted attacks by using pools of various sizes and different values for K, which represent the number of functions returned by the model that are more similar to the query function f_Q. We observed a significant difference in the Attack Success Rate (**ASR**) with targeted attacks being successful in about 57.06% of cases, whereas untargeted attacks in 95.81%.

- Transferability. We investigated whether an adversarial example crafted for one model could be used to attack another model. Our analysis shows that targeted attacks do not transfer well. On the other end, untargeted attacks generally transfer, with an average **ASR** of 62.11%.

- Common model behaviors. We performed an in-depth analysis on the structure of the adversarial examples, to check whether they reveal useful insights about the attacked models.

- Our artifacts are available at: https://github.com/Sap4Sec/BCSD_Robustness.git

2 Threat Model

This work focuses on assessing the robustness of binary function similarity systems at inference time (i.e., we do not investigate their robustness against poisoning attacks). To this end, we assume a black-box attacker [6, 39], with no access to the target model or training data. The attacker can perform an unlimited number of queries to observe the similarity value produced by the model.

We emphasize that if the model does not provide the similarity score but only categorical outputs, the attack scenario shifts to a gray-box setting. However, the attacker's knowledge remains minimal, as they only need the similarity score to execute the attack effectively.

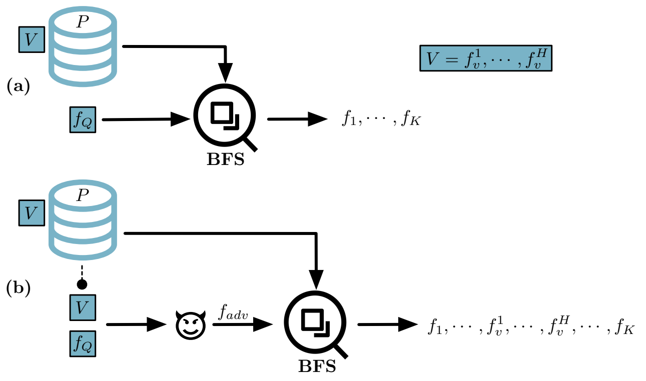

(a)
V
P
f_Q
BFS
$f_1, \cdots, f_K$
$V = f_v^1, \cdots, f_v^H$
(b)
V
P
V
f_Q
f_{adv}
BFS
$f_1, \cdots, f_v^1, \cdots, f_v^H, \cdots, f_K$

Figure 1: **Targeted** attack. **(a)** Initially, the target variants $V = \{f_v^1, \cdots, f_v^H\}$ are not among the top-K most similar functions to f_Q in the pool P. **(b)** After the attack, using f_{adv} as query brings all variants in V into the top-K.

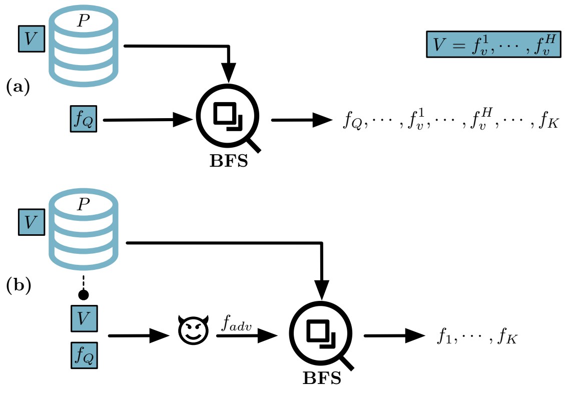

(a)
V
P
f_Q
BFS
$f_Q, \cdots, f_v^1, \cdots, f_v^H, \cdots, f_K$
$V = f_v^1, \cdots, f_v^H$
(b)
V
P
V
f_Q
f_{adv}
BFS
$f_1, \cdots, f_K$

Figure 2: **Untargeted** attack. **(a)** Initially, f_Q and its variants $V = \{f_v^1, \cdots, f_v^H\}$ are among the top-K most similar functions to f_Q in the pool P. **(b)** After the attack, using f_{adv} as query removes f_Q and its variants from the top-K.

2.1 Targeted and Untargeted Attacks

Let *sim* be a similarity function that takes as input two functions and returns a real number, the *similarity score* between them. We define two binary functions as *semantically equivalent* if they are two implementations of the same abstract functionality. A set of functions is a set of *variants* if all the functions are compiled from the same source code.

An intriguing challenge within binary function similarity systems consists of the One-to-Many (OM) task [16, 40, 44] (or **function search**). In this task, given a pool of functions P, a query function f_Q, and a number K (where $K \leq |P|$), we have to identify in P the K functions that, according to the attacked model, are more similar to f_Q. These K functions are the top-K for the query f_Q.

We consider two scenarios for an attacker interested in attacking the function search task: **targeted** and **untargeted** attacks.

In a **targeted attack**, the adversary is given a pool of functions, P, a set of target variants, $V \subset P$, and a query function, f_Q. The adversary has to find a function f_{adv} semantically equivalent to f_Q. When f_{adv} is used as a query over P, the top-K must include i target functions from V (with $i \leq K$).

A typical targeted attack occurs when the attacker wants to create an adversarial version f_{adv} of a specific malicious function f_Q. This malicious function must resemble a certain benign function in P or one of its variants. Consequently, when the defender uses the binary function similarity system, a set of benign variants will be ranked among the top-K functions most similar to f_{adv}.

In an **untargeted attack**, the adversary is given a pool of functions, P, a query function, $f_Q \in P$, and all its variants V in P. The adversary has to find a function f_{adv} semantically equivalent to f_Q such that, when f_{adv} is used as query over P, at least i variants are not in the top-K (with $i \leq |V|$).

A practical untargeted attack occurs when the attacker seeks to introduce a known vulnerable function, f_Q, into a firmware. Their goal is to create a function, f_{adv}, that is semantically equivalent to f_Q but as dissimilar as possible to all its variants, including f_Q itself. As a result, when a binary function similarity system is used for vulnerability detection by the defender, none of the variants of f_Q and f_Q itself will be ranked among the top-K functions of P most similar to f_{adv}.

We present a visual representation of a targeted attack in Figure 1 and an untargeted attack in Figure 2.

3 Attack Overview

In this section, we present the black-box procedure we utilize to assess the robustness of the models under consideration. We first describe the objective function the attacker seeks to solve and the optimization strategy adopted for generating adversarial examples. Then, we introduce the semantics-preserving techniques for manipulating binary functions we embody in our attack.

3.1 Multi-Objective Optimization

In the following, we refer to the targeted attack case. The same rationale, with the necessary minor adjustments, holds for the untargeted case.

In the context of a targeted attack, given a query function f_Q, and set $V \subset P$ of target variants, the goal of the attacker consists of generating an adversarial example f_{adv} starting from f_Q that maximizes the similarity between f_{adv} and all the target variants $f_v \in V$.

This translates into the following multi-objective function on an undefined number of variables (one for each variant):

$$\max_{f_{adv}}(sim(f_{adv}, f_v^1), \ldots, sim(f_{adv}, f_v^H)) \quad (1)$$

We solve this multi-objective problem by reducing it to the following max-min problem that takes into account also the perturbation size:

$$\max_{f_{adv}} \min_{f_v} sim(f_{adv}, f_v) - \lambda \cdot \frac{|len(f_Q) - len(f_{adv})|}{len(f_Q)} \quad (2)$$

where:

- $len(\cdot)$ takes a binary function as input and returns its length in terms of the number of instructions;
- λ determines how much the size of the perturbation should penalize the produced adversarial example. In the following, we refer to λ as the **penalty factor**.

Informally, with this max-min problem, we maximize the minimum similarity between f_{adv} and the variants in V.

However, we highlight that differently from the computer vision scenario [20], the perturbation size does not present a significant concern within our threat model. Indeed, as elucidated in [39], an adversarial binary code must exhibit both validity and realism. Consequently, our set of transformations must alter f_{adv} so that it will still look plausible when manually analyzed, which doesn't imply minimizing the modification size.

3.1.1 Greedy Optimizer

To solve the max-min problem, we use an ε-greedy gradient-free optimizer to iteratively modify f_Q toward the desired similarity outcome. That is, maximize the similarity between the adversarial example f_{adv} and the least similar variant in P.

At each iteration, starting from the output f'_{adv} of the previous iteration (or the original function for the first iteration), we generate multiple candidate adversarial examples. This is done by applying *semantics-preserving* transformations from a predefined set TR to specific locations within f'_{adv} identified in the set POS. Specifically, we create a fixed number of candidates by considering all possible pairs $\langle tr, pos\rangle$, where $tr \in TR$ and $pos \in POS$. Notably, a single transformation can generate multiple candidates when applied to a given location. The next section details the specific semantics-preserving transformations used in this process.

For each candidate, we compute the objective function in Equation 2; then, we select the adversarial example f_{adv} for the next iteration using an ε-greedy procedure over the generated candidates. Specifically, with probability $1 - \varepsilon$, we select the candidate that maximizes the objective function, and with probability ε, we choose a suboptimal one. At the end of the iterative procedure, we select as the final f_{adv} the one that produced the highest value for the objective function among all f_{adv} generated at the end of each iteration. This is then used as a query over P.

A certain semantics-preserving transformation can only be applied to a specific set of positions inside the original function we want to mutate (for example, we cannot swap instructions having a dependency). However, we have to further restrict this set to keep our attack computationally feasible. We do this by identifying the positions where transformations are more likely to greatly impact the similarity function. Here, each position is an assembly instruction within the function being modified. Specifically, the importance of instruction $i \in f'_{adv}$ is the absolute variation of the similarity value of f'_{adv} and its least similar function $f_v \in V$ measured when removing i:

$$IM_i = |sim(f'_{adv}, f_v) - sim(f'_{adv} \setminus i, f_v)|, \quad (3)$$

where $f'_{adv} \setminus i$ is the function f'_{adv} without instruction i.

After computing the importance score for each instruction $i \in f'_{adv}$, the ones with the highest value will be candidates for applying *semantics-preserving* perturbations.

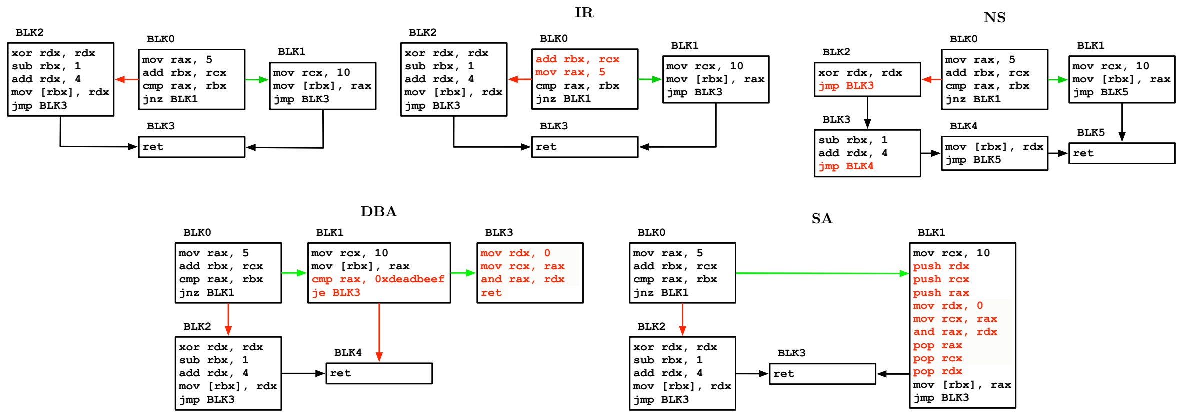
3.2 Semantics-Preserving Transformations

As noted in [39], an adversarial function f_{adv}, generated from a binary function f_Q must satisfy **problem-space constraints**, including preserving its semantics.

Figure 3 illustrates the semantics-preserving transformation techniques we embodied in our attack strategy. Some of these were initially discussed in [9], together with a categorization based on whether they modify or not the control-flow graph (CFG). Among these transformations, we embed in our attack strategy: Instruction Reordering **(IR)**, Node Split **(NS)**, and Dead Branch Addition **(DBA)**. Specifically, **IR** reorders consecutive data-independent instructions within the function, ensuring that the swap preserves the semantics. **NS** splits an existing CFG node into three separate nodes using unconditional jumps, without altering the semantics of the function. Finally, **DBA** introduces dead code— specifically, a strand (i.e., a sequence of data-dependent assembly instructions [3])— into a new basic block that is guarded by an always-false branch.

Additionally, we introduce a new transformation called Strand Addition **(SA)**, which inserts a strand into an existing CFG node. The strands under consideration contain neither control flow nor memory-modifying instructions. Furthermore, using liveness analysis, we identify all registers and flags used within the strand, saving their contents at the beginning and restoring them at the end, to ensure that the function's semantics remain unaltered.

We select the strand to add to the function by enumerating a set of candidates sampled from a large dataset of available strands. Rather than defining a fixed set of strands, we rely on an embedding space to



```
BLK0
mov rax, 5
add rbx, rcx
cmp rax, rbx
jnz BLK1

BLK1
mov rcx, 10
mov [rbx], rax
jmp BLK3

BLK2
xor rdx, rdx
sub rbx, 1
add rdx, 4
mov [rbx], rdx
jmp BLK3

BLK3
ret
```

IR

```
BLK0
add rbx, rcx
mov rax, 5
cmp rax, rbx
jnz BLK1

BLK1
mov rcx, 10
mov [rbx], rax
jmp BLK3

BLK2
xor rdx, rdx
sub rbx, 1
add rdx, 4
mov [rbx], rdx
jmp BLK3

BLK3
ret
```

NS

```
BLK0
mov rax, 5
add rbx, rcx
cmp rax, rbx
jnz BLK1

BLK1
mov rcx, 10
mov [rbx], rax
jmp BLK5

BLK2
xor rdx, rdx
jmp BLK3

BLK3
sub rbx, 1
add rdx, 4
jmp BLK4

BLK4
mov [rbx], rdx
jmp BLK5

BLK5
ret
```

DBA

```
BLK0
mov rax, 5
add rbx, rcx
cmp rax, rbx
jnz BLK1

BLK1
mov rcx, 10
mov [rbx], rax
cmp rax, 0xdeadbeef
je BLK3

BLK2
xor rdx, rdx
sub rbx, 1
add rdx, 4
mov [rbx], rdx
jmp BLK3

BLK3
mov rdx, 0
mov rcx, rax
and rax, rdx
ret

BLK4
ret
```

SA

```
BLK0
mov rax, 5
add rbx, rcx
cmp rax, rbx
jnz BLK1

BLK1
mov rcx, 10
push rdx
push rcx
push rax
mov rdx, 0
mov rcx, rax
and rax, rdx
pop rax
pop rcx
pop rdx
mov [rbx], rax
jmp BLK3

BLK2
xor rdx, rdx
sub rbx, 1
add rdx, 4
mov [rbx], rdx
jmp BLK3

BLK3
ret
```

Figure 3: Semantics-preserving transformations embodied in the attack.

establish a set of candidates that is dynamically updated at the end of each iteration within the optimization procedure. We initialize this set using strands uniformly sampled from our large dataset. At the end of each optimizer iteration, we update part of the set with strands selected from the closest neighbors (using the embedding space) of those representing the top greedy actions from the previous iteration (specifically from the **SA** category), while the remaining portion is filled with new random strands.

We define the embedding space using a strand embedding model to transform strands into vectors. These vectors are grouped according to the semantics of the strands, allowing us to establish a notion of proximity between them. We choose BinBert [3] as the model for generating strand embeddings because it has been specifically fine-tuned for strands similarity detection.

We provide a detailed pseudocode of our attack strategy in Appendix A.

4 Target Systems

In this section, we illustrate the binary function similarity models we attacked. Based on [31], we select the different solutions according to the following **selection criteria**:

- **Scalability**. To demonstrate the potential vulnerabilities of models in a real-world scenario, we evaluate approaches that are not slow at inference time.
- **Diversity of proposed approach**. Binary function similarity solutions span multiple research communities (i.e., system security, programming languages, machine learning) and multiple approaches. Moreover, within each community, diverse techniques have emerged; for instance, the machine learning community explores various architectures like GNNs, RNNs, and Transformers. Therefore, we select a range of models that not only cover different architectural structures but have also been proposed by different research communities.

We now briefly describe the targeted models.

4.1 Graph Neural Network (GNN): Gemini and GMN

Gemini (2017) [46] is built on a graph neural network derived from the Structure2Vec [12] model and trained using a Siamese architecture. It converts an ACFG (a control flow graph with manual block-level features) into an embedding vector, which is generated by aggregating the embedding vectors of the individual nodes within the ACFG. Given two binary functions, their similarity is determined by the cosine similarity of their ACFG embedding vectors.

Graph Matching Network (GMN) (2019) [29] consists of a graph neural network that calculates the similarity between two functions by representing them through their CFGs. Unlike standard GNN solutions (i.e., Gemini [46]), where the embedding vector for a node captures properties from its neighborhood only, GMN also searches for possible matches across the two input graphs. In particular, GMN computes the distance between the inputs by trying to match their nodes.

4.2 Intermediate Representation (IR) and Neural Network (NN): Zeek

Zeek [41] uses an intermediate representation to capture the semantics of the input functions. Specifically, starting from the CFG of a function, it first decomposes each basic block into strands and converts the assembly code in each strand to a normalized intermediate representation (i.e., VEX-IR). Subsequently, it generates a hash value for each block, which is then indexed in a vector to represent the function. Finally, it uses two fully connected hidden layers to detect the similarity between vectors derived from semantically equivalent code sections.

4.3 Fully Connected Neural Network: BinFinder

BinFinder (2023) [40] employs a fully connected neural network trained using a Siamese architecture to generate function embeddings, which are then employed for similarity calculation. It represents the input functions through a set of static features that are claimed to be robust to code obfuscation, compiler optimization, and cross-compilation processes. These features are first processed using different tokenizers; then, they are transformed into one-hot vectors.

4.4 Recurrent Neural Network (RNN): SAFE

SAFE (2022) [33] is a recurrent neural network-based model trained using a Siamese architecture, that converts the linear disassembly of the input functions into embedding vectors. It first computes an embedding for each assembly instruction using a model derived from the word2vec [34] word embedding model. Then, using a self-attentive component, it aggregates these vectors into a final function embedding vector. Similar to Gemini [46], the similarity between two functions consists of the cosine distance between their corresponding embeddings.

4.5 Transformer: jTrans, Trex, PalmTree

jTrans (2022) [44] is a BERT-based model [15] that combines instruction semantics with CFG information to infer the binary code representation. When considering an assembly instruction, jTrans treats each mnemonic and operands as tokens and computes an embedding for each of them. However, to make jTrans better in capturing the control flow execution, *jump* instructions are modeled differently from other ISA instructions. The last layer of the model generates the function's embedding, which can then be used for similarity calculation. jTrans has been pre-trained on two tasks (i.e., Masked Language Model and Jump Target Prediction) and then fine-tuned for Binary Similarity Detection.
Trex (2023) [38] is a transfer-learning-based framework that adopts a hierarchical Transformer [37] for learning functions' semantics via regular and forced execution traces. The model is first pre-trained using a Masked Language Modeling-like task. In particular, given a function's trace (which comprises both instructions and values), some parts are randomly masked to be predicted during model training using the surrounding context. Once pre-trained, the model is then fine-tuned for function similarity detection, using functions' static code (instead of traces) as input to the model. Specifically, the function's embedding is the output of a two-layer MLP. This MLP takes as input the mean pooling of the embeddings produced by the last self-attention layer of the fine-tuned model.
PalmTree (2021) [28] is a BERT-based [15] pre-trained assembly language model for generating general-purpose instruction embeddings. Here, assembly instructions are treated as separate sentences composed by basic tokens. In particular, each operand is decomposed into basic elements, normalizing strings and addresses with special tokens to avoid the OOV (Out-Of-Vocabulary) problem. The model is pre-trained considering three tasks: Masked Language Model, Context Window Prediction, and Def-Use Prediction. Since PalmTree is an instruction embeddings model, to evaluate its performance on the function similarity task, we follow the strategy used in [3], where an LSTM aggregates the instruction embeddings produced by PalmTree into a function embedding.

5 Datasets and Implementation

This section describes the datasets we use to evaluate our approaches and implementation details.

5.1 Dataset

We test our approach by considering a codebase of binary functions extracted from 8 open-source projects written in C language: binutils, curl, openssl, sqlite, gsl, libconfig, ffmpeg, and postgresql. We compile the programs for the `amd64` architecture, considering two compilers (i.e., `gcc-9.4.0` and `clang-12`) and two optimization levels (namely, `O0` and `O3`), obtaining 4 different combinations. The collection we obtained reflects real-world software, including binaries utilized in the evaluation or training of the binary function similarity systems examined in this work.

We extract the binary functions using `Radare2` [35] disassembler, excluding those that contain fewer than 6 assembly instructions or 2 CFG nodes, obtaining 127,534 functions. Our final codebase consists only of the functions for which there are exactly four variants, one for each combination of compiler and optimization level.

Once obtained the codebase, we create pools of different sizes. A pool P of size S is composed of $S/4$ distinct functions uniformly sampled from the codebase together with their variants. Our final targeted dataset comprises 1,000 samples, each representing a query on a certain pool on which the attack has to be carried out. Specifically, each targeted sample contains: the pool P; a set $V \subset P$ of target variants; a query function f_Q (the one that will be modified by the attacker) extracted randomly among our codebase of functions. Note that $f_Q \notin V$.

For the untargeted case, we create a similar dataset of 1,000 samples, each of which contains: the pool P; a set $V \subset P$ of target variants; a query function $f_Q \in V$ (the one that will be modified by the attacker).

5.2 Implementation Details

We implement our attack as a two-phase procedure; the first one consists of disassembling the input functions and building a high-level representation using their CFGs; the latter consists of applying the transformations to the aforementioned representations and then feeding them in input to the target model. Therefore, we remark that our attack is not done using binary rewriting techniques but it is performed on this high-level representation. However, we want to stress that all our transformations are semantics-preserving *by design*, as detailed in Section 3.2; furthermore, all these transformations can be effectively applied as source code modifications by first translating the C code of the function into assembly code (by compiling the `.c` file into a `.s` file) and then by directly modifying the obtained assembly code. In this way, the injection process will not rely on binary rewriting techniques, which are more prone to alter the semantics due to relocation issues.

We built our CFG extraction module upon the `Radare2`[1] and `angr` [42] disassemblers.

Our testing pipeline has been coded in Python. We consider the official implementation and settings for each target model except for Gemini, GMN, and ZEEK for which we use alternative implementations.[2]

6 Experimental Results

In this section, we provide the results of our experimental evaluation. We first define the performance metrics we use, then we describe our choice of the attack hyperparameters and finally, we evaluate the resiliency of binary similarity models by investigating the following research questions:

RQ1: *Do the models exhibit greater robustness against targeted or untargeted attacks?*
RQ2: *Are the considered transformations able to generalize across various categories of target models? Do adversarial examples transfer across models?*
RQ3 *Is it possible to deduce aspects of the model through the distribution of applied transformations?*

Successful Attacks

According to the definitions provided in Section 2.1, in the **targeted** scenario, we deem an attack as successful when at least i variants in V are among the top-K results when f_{adv} is used as a query over P; contrarily, in the **untargeted** case, an adversarial example is successful when at least i variants in V are ranked outside the top-K when f_{adv} is used as query over P.

As seen in Section 5.1, each attack considers a set V of exactly 4 variants, meaning $i \leq 4$. The attack becomes more challenging as i increases. For instance, when $i = 1$, a targeted attack is successful when just one variant is ranked among the top-K. Conversely, when $i = 4$, all variants must be in the top-K, making the attack harder.

Also, the factor K affects the outcome of the attack, specifically low values of K make the targeted attack harder. For example, let's assume $i = 4$; with $K = 4$ the attack succeeds only if the top-K set is exactly the set of variants. When increasing K to 5, the attack is successful even if one function of P not in V is in the top-K. Conversely, increasing K makes the untargeted attack more challenging. For instance, when $K = |P| - 4$ the attack succeeds only when the top-K equals $P \setminus V$.

The size of P impacts the result of the attack. As the pool size increases, a targeted attack becomes more difficult because more functions could be ranked among the top-K. Contrarily, in the untargeted case, the attack becomes more difficult as the pool size decreases, since fewer functions in P can be ranked within the top-K.

Performance Metrics

We evaluate the robustness of the target models by using the Attack Success Rate (**ASR**) as the main metric, which is the percentage of adversarial examples that meet the success condition. We calculate the **ASR** for each value of $i \in \{1, 4\}$ (***ASR@1***, ***ASR@2***, ***ASR@3***, ***ASR@4***). Recall that in ***ASR@i*** we consider successful the targeted attacks that bring at least i variants in top-K. In the untargeted case, the attacks have to move at least i variants outside top-K.

We also defined an aggregated **ASR**, namely **wASR**, in which we decreasingly penalize the success of an experiment depending on the number of variants satisfying the success condition. For example, if when using f_{adv} as query only one of the variants is among the top-K, then this experiment will count for 0.25, if there are two it will count for 0.50, and so on.

We use the percentage of query functions where the success condition at i is met before executing the attack, namely ***INIT@i*** metrics for $i \in \{1, 4\}$ as reference.

In our untargeted attack experiments, we also compute the standard ***recall@K*** before (**Recall pre attack**) and after (**Recall post attack**) the attack. This quantifies the model's performance on clean data and its degradation following untargeted attacks.

We further investigate the quality of our attack using two other support metrics for each value of $i \in \{1, 4\}$, computed over the set of successful examples. The first metric, ***M-Instrs@i***, represents the number of new instructions in f_{adv} at the end of the attack. The second metric, ***M-Nodes@i***, measures the number of new nodes in f_{adv} at the end of the attack.

Parameters of the Attack

We run our attack for up to 30 iterations, exploring λ values of 0, 0.01, and 0.3 (see Section 3.1) to assess its impact on our results. Given that the average length of our query functions is approximately 100, we perturb 50 positions—about half of the total. As in [44], we use pool sizes of $32, 128, 512$, and 1000.

Due to the computational overhead of dynamically updating the set of candidates (see Section 3.2), the **SA** and **DBA** transformations are significantly slower than the others. To keep our experiment durations reasonable, we limit these transformations by testing 20 strands from a candidates' set composed by 100 strands with 50% of random strands.

[1]https://github.com/radareorg/radare2
[2]Due to unavailability of original implementations or incompatibility with our pipeline.

Table 1: Untargeted attack at $K = 10$ when considering pools with size 128 and 512 and setting $\lambda = 0$. In column AVG we report the average of the measures across all models.

		Models																	
		Gemini		GMN		ZEEK		BinFinder		SAFE		jTrans		Trex		PalmTree		AVG	
	$\|P\|$	**128**	**512**	**128**	**512**	**128**	**512**	**128**	**512**	**128**	**512**	**128**	**512**	**128**	**512**	**128**	**512**	**128**	**512**
	Recall pre attack	0.71	0.56	0.94	0.86	0.73	0.51	0.91	0.83	0.94	0.84	0.81	0.67	0.95	0.89	0.90	0.79	0.86	0.74
	Recall post attack	0.07	0.04	0.03	0.01	0.05	0.02	0.06	0.04	0.02	0	0.04	0.01	0.04	0.02	0.03	0.01	0.04	0.02
	wASR	92.99	96.27	97.30	99.0	95.23	98.12	93.80	96.33	98.30	99.58	96.0	99.15	96.17	98.43	96.69	99.37	95.81	98.28
@1	**INIT**	63.63	82.87	16.10	36.40	61.76	84.70	24.40	39.40	19.40	42.60	51.10	73.20	12.82	29.22	27.66	53.15	34.61	55.19
	ASR	96.69	98.10	99.30	99.70	98.06	99.09	95.80	97.80	99.20	100.0	99.40	100.0	97.61	99.30	98.0	99.90	98.01	99.23
	M-Instrs	235.07	234.18	214.44	215.0	201.77	202.11	226.76	227.07	237.61	236.85	133.19	133.06	195.51	193.39	524.32	522.24	246.08	245.49
	M-Nodes	26.98	26.95	16.90	16.94	25.68	25.83	41.73	41.80	13.68	13.66	16.74	16.73	10.99	10.88	16.69	16.80	21.17	21.20
@2	**INIT**	38.48	64.33	5.90	16.10	35.73	72.26	9.90	21.20	4.90	17.40	24.20	48.40	4.97	12.82	10.62	28.63	16.84	35.14
	ASR	95.39	97.49	98.80	99.30	97.26	98.86	94.70	96.90	98.80	99.90	99.20	100.0	97.22	98.81	97.90	99.40	97.41	98.83
	M-Instrs	236.42	234.41	214.09	214.44	201.71	202.06	226.78	226.96	238.13	236.93	133.35	133.06	195.75	194.03	524.57	522.88	246.35	245.60
	M-Nodes	27.12	27.0	16.87	16.92	25.61	25.81	41.67	41.79	13.69	13.66	16.75	16.73	11.01	10.91	16.67	16.79	21.17	21.20
@3	**INIT**	13.23	27.15	0.80	3.40	9.02	34.02	3.0	6.80	1.50	3.50	1.40	12.0	0.89	2.49	1.20	3.90	3.88	16.66
	ASR	91.58	95.39	97.0	98.80	94.52	97.60	93.10	95.90	97.40	99.30	94.80	99.10	95.83	98.11	95.89	99.20	95.07	97.92
	M-Instrs	239.26	237.11	212.59	213.98	202.82	202.18	226.88	226.92	239.68	237.66	135.64	133.94	197.18	195.07	523.93	523.35	247.18	246.28
	M-Nodes	27.28	27.11	16.93	16.88	25.26	25.62	41.59	41.72	13.69	13.68	16.96	16.84	11.05	10.97	16.71	16.75	21.18	21.20
@4	**INIT**	1.20	3.01	0	0	0	3.08	0	0	0	0	0	0	0	0	0	0	0.15	0.76
	ASR	88.28	94.09	94.10	98.20	91.10	96.92	91.60	94.70	97.40	99.10	90.60	97.50	94.04	97.51	94.99	99.0	92.76	97.13
	M-Instrs	241.56	239.12	212.66	214.32	114.09	202.45	226.61	226.92	239.68	237.94	137.80	135.35	199.09	195.80	525.71	523.85	248.23	246.97
	M-Nodes	27.73	27.28	17.18	16.92	5.52	25.52	41.52	41.66	13.70	13.68	17.22	17.0	11.17	11.01	16.76	16.75	21.30	21.23

6.1 RQ1: Targeted vs Untargeted Attacks

In this section, we investigate the robustness of the considered models when subject to **targeted** and **untargeted** attacks. For brevity, we report in the tables and the plots only the results corresponding to the 128 and 512 pools and to $\lambda = 0$, which is the worst case according to the modification size. Furthermore, we discuss only the results @4, corresponding to the more difficult setup, and the **wASR**. For the complete results, see Appendix B.

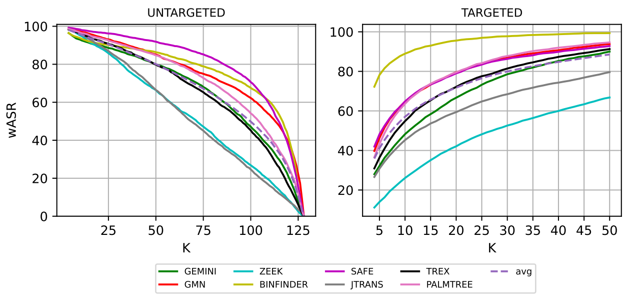


Figure 4: **wASR** while varying the K value and considering $|P| = 128$. The dotted curve represents the average values across the different models.

6.1.1 Untargeted Attacks

Table 1 presents the results for the **untargeted** attack scenario.

For $|P| = 128$, we observe an average **INIT@*4*** of 0.15%, indicating that the success condition is virtually never met without the attack. After applying the attack and querying the pool with the generated f_{adv}, we achieve an average **ASR@*4*** of 92.76%, meaning that in over 9 out of 10 cases, all variants in V are pushed outside the Top-10. Furthermore, when considering the **wASR**, the attack succeeds in 95.81% of cases.

Figure 4 shows the average **wASR** for K values ranging from 4 to 128. For small K values (≤ 25), the average **wASR** remains well above 90% but decreases sharply as the search depth increases, consistent with observations in Section 6. However, see Figure 5, the **wASR** maintains an average consistently above 80%, indicating that a larger difference between K and the pool size significantly benefits the attacker. Notably, when the binary function similarity model is used to detect vulnerable or malicious functions, the pool likely contains thousands of functions, making untargeted attacks significantly easier.

The **recall@*10*** metric further underscores the models' vulnerability to untargeted attacks. Specifically, for a pool of 128 functions, the **recall@*10*** decreases from an initial average of 0.86 to 0.04 after running the attack.

6.1.2 Targeted Attacks

Table 2 presents the attack results for the **targeted** scenario. With a pool of 128 functions, the average **INIT@*4*** is 1.44%, indicating that variants in V are ranked in the Top-10 for their corresponding f_Q in less than 2% of cases. The average **ASR@*4*** is 39.88%, meaning that when querying with f_{adv}, all variants in V are ranked in the Top-10 nearly 40% of the time. The **wASR** is 57.06%.

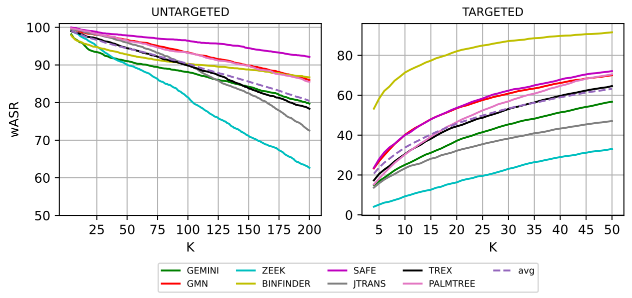


Figure 5: **wASR** while varying the K value and considering $|P| = 512$. The dotted curve represents the average values across the different models.

Figure 4 illustrates the average **wASR** for targeted attacks across $K \in [4, 50]$ when querying a pool of 128 functions, showing a rapid increase to values exceeding 40%. Figure 5 depicts similar trends with a pool of 512 functions; however, the **wASR** increases more gradually compared to $|P| = 128$. As reported in Appendix B, model robustness further increases with a

Table 2: Targeted attack at $K = 10$ when considering pools with size 128 and 512 and setting $\lambda = 0$. In column AVG we report the average of the measures across all models.

		Models																	
		Gemini		GMN		ZEEK		BinFinder		SAFE		jTrans		Trex		PalmTree		AVG	
	$\|P\|$	**128**	**512**	**128**	**512**	**128**	**512**	**128**	**512**	**128**	**512**	**128**	**512**	**128**	**512**	**128**	**512**	**128**	**512**
	wASR	48.25	25.10	64.68	40.15	26.07	9.34	88.95	71.37	64.80	40.02	45.25	23.35	54.85	30.48	63.60	30.24	57.06	33.76
@1	**INIT**	17.20	5.70	15.80	3.90	15.0	3.47	12.12	3.11	13.90	4.0	23.90	4.80	12.10	3.30	14.97	4.11	15.62	4.05
	ASR	71.90	45.10	78.60	59.80	43.16	19.49	94.39	80.96	80.30	59.10	80.20	49.10	68.80	44.60	78.04	47.19	74.42	50.67
	M-Instrs	173.62	170.18	214.53	210.35	94.86	101.21	155.21	154.78	123.74	128.05	77.29	84.46	112.70	112.76	193.41	182.05	143.17	142.98
	M-Nodes	13.83	15.23	18.87	18.35	11.18	10.80	33.28	33.0	15.72	15.72	11.95	12.66	11.44	11.36	18.28	18.62	16.82	16.97
@2	**INIT**	9.40	2.30	8.10	1.90	6.53	0.71	7.82	1.80	8.60	1.40	8.30	0.80	8.20	1.60	7.98	1.40	8.12	1.49
	ASR	57.0	29.20	71.10	47.30	31.94	11.33	91.68	75.55	71.40	46.30	58.90	28.0	60.40	34.40	70.06	34.97	64.06	38.38
	M-Instrs	173.68	173.84	213.81	214.23	96.81	108.93	155.94	156.73	126.32	128.05	84.68	94.18	112.83	114.63	192.08	173.26	144.52	145.67
	M-Nodes	14.68	16.26	18.63	18.68	10.35	11.50	33.40	33.46	15.89	15.72	12.65	13.61	11.30	11.13	18.09	18.08	16.87	17.30
@3	**INIT**	2.70	0.50	3.30	0.40	1.84	0	4.91	0.80	3.10	0.3	1.30	0.1	4.10	0.50	2.89	0.10	3.02	0.34
	ASR	38.0	15.70	60.40	32.30	18.57	4.29	87.37	67.33	60.10	32.30	27.30	10.40	49.50	24.10	57.58	22.14	49.85	26.07
	M-Instrs	177.87	181.08	214.63	224.91	93.72	108.83	157.19	159.0	129.75	131.05	96.62	105.67	115.36	114.20	189.60	175.48	146.84	150.01
	M-Nodes	15.76	17.45	18.92	19.17	10.26	11.14	33.67	33.88	15.84	15.88	13.86	15.56	11.34	11.20	18.15	19.18	17.23	17.93
@4	**INIT**	0.90	0.20	1.60	0	0.31	0	3.21	0.40	1.40	0	0.30	0.10	2.60	0.30	1.20	0	1.44	0.13
	ASR	26.10	10.40	48.60	21.20	10.61	2.24	82.36	61.62	47.40	22.40	14.60	5.9	40.70	18.80	48.70	16.63	39.88	19.90
	M-Instrs	190.36	192.19	219.14	239.85	104.95	124.5	159.24	160.23	136.29	140.71	109.49	118.15	116.21	117.37	192.21	176.20	153.49	158.65
	M-Nodes	16.57	18.25	19.48	19.49	10.99	13.45	34.19	34.12	16.05	16.40	14.71	16.24	11.42	11.53	18.37	19.39	17.72	18.60

pool of 1000 functions, yielding an average **wASR** of 25.25%.

6.1.3 Impacts of the Modification Size

We study how the modification size impacts models robustness by varying the λ parameter in the Equation 2. We analyzed three models selected to be representative of all DNN architectures (specifically, Gemini, SAFE, and jTrans) with λ values of 0, 0.01, and 0.3. The complete results are reported in Appendix B.

As shown in Figure 6, increasing λ leads to a significant decrease in the average **wASR** in both untargeted and targeted scenarios. In the untargeted case, the **wASR** goes from 95.76% at $\lambda = 0$ to 77.39% at $\lambda = 0.01$, and to 24.59% at $\lambda = 0.3$. Correspondingly, the average number of modifications, **M-Instrs@4**, decreases from 206.34 at $\lambda = 0$ to 32.26 at $\lambda = 0.01$, and to 20.26 at $\lambda = 0.3$. In the targeted case, the average **wASR** goes from 52.77% at $\lambda = 0$ to 27.74% at $\lambda = 0.01$, and to 9.94% at $\lambda = 0.3$, while the **M-Instrs@4** decreases from 145.38 at $\lambda = 0$ to 19.10 at $\lambda = 0.01$, and to 6.19 at $\lambda = 0.3$.

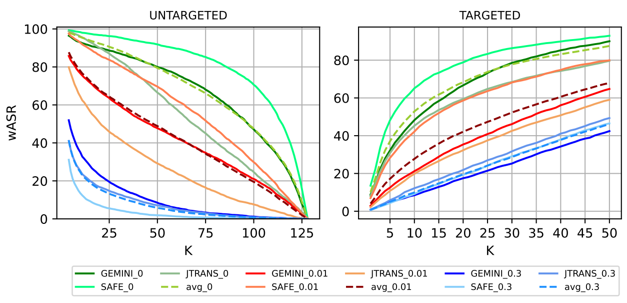


Figure 6: **wASR** while varying the K value, considering $|P| = 128$ and $\lambda \in \{0, 0.01, 0.3\}$. For each value of λ, we show the **wASR** together with the average on three models differing in architecture. We represent in green the results corresponding to $\lambda = 0$, in red the results corresponding to $\lambda = 0.1$, and in blue the ones for $\lambda = 0.3$.

Key takeaway. All models are more susceptible to untargeted attacks than targeted ones. Specifically, the average **wASR** is 95.81% vs 57.06% when querying a pool of 128 functions, and 98.28% vs 33.76% when querying a pool of 512 functions.

6.2 RQ2: Generalizability and Transferability

In this section, we discuss the generalizability of our approach, showing how the considered metrics vary across different models. We investigate whether the generated adversarial examples can be transferred across diverse models, to demonstrate the variation in the attack success rate when adversarial examples intended for one model are presented to a different model.

6.2.1 Generalizability

Our black-box transformations alter both the CFG and instruction sequence of a function. A key point is whether they are sufficient to attack all examined models.

In the untargeted setting with $|P| = 128$, Figure 4 shows similar performance across most models at various levels of K, especially for $K < 25$, as confirmed in Table 1. Here, Gemini is the most robust model (**wASR** = 92.99%), while SAFE is the weakest (**wASR** = 98.30%).

As visible in Figure 5, this trend holds for larger pools. As K increases, ZEEK and jTrans stand out as the most robust models, with SAFE remaining the weakest. Even at $K = 100$, ZEEK is fooled in over 80% of cases, suggesting that blacklist defenses implemented by binary function similarity models can be bypassed in practice (e.g., a vulnerable function ranked beyond the top 100 may be ignored). We speculate that SAFE is more vulnerable due to its reliance on pre-trained embeddings of preprocessed instructions, making it sensitive to transformations like **SA** and **DBA**.

Interestingly, Table 1 shows that both ZEEK and Gemini have some of the lowest **Recall pre attack**

Table 3: Transferability matrix for the untargeted attack case, considering $|P| = 128$ and $K = 10$. In the rows, we indicate the model for which the adversarial examples were created, and in the columns, the model on which the examples are tested. Each value represents the **wASR**.

	Gemini	GMN	ZEEK	BinFinder	SAFE	jTrans	Trex	PalmTree	**TSR**
Gemini	■	70.67	82.34	23.55	44.69	42.66	35.70	71.92	53.08
GMN	68.30	■	85.02	20.30	50.78	50.98	47.77	67.17	55.76
ZEEK	71.59	76.13	■	20.08	62.40	56.58	60.39	70.91	59.73
BinFinder	79.90	80.15	85.65	■	60.68	61.18	58.77	68.19	70.65
SAFE	61.02	62.38	84.38	24.22	■	72.12	79.57	68.77	64.64
jTrans	63.60	57.38	77.38	20.15	65.05	■	60.60	59.43	57.66
Trex	58.02	57.99	82.36	19.96	76.95	67.69	■	71.44	62.06
PalmTree	71.39	77.15	87.27	31.41	83.74	74.15	88.33	■	73.35
VR	67.69	68.84	83.49	22.81	63.47	60.77	61.59	68.26	■
random	67.81 ± 0.06	55.68 ± 0.73	79.92 ± 0.15	17.87 ± 0.97	45.94 ± 0.30	49.69 ± 0.80	31.66 ± 1.02	53.16 ± 0.32	50.22

values among the models, yet their **Recall post attack** values are among the highest, meaning they are more robust against our attack. In contrast, models with top performance on clean data, like GMN, SAFE, and Trex, are the ones exhibiting less robustness, as they show lower **Recall post attack** values.

As visible in Table 2 and in Figure 4, the performance at $K = 10$ in the **targeted** scenario is comparable across the majority of the models, except for BinFinder, which is the least robust model, and ZEEK which is the most robust one. BinFinder learns function semantics considering features like VEX-IR instructions and constants, which are heavily impacted by **SA** and **DBA**. All the other models exhibit consistent robustness, with jTrans being the most robust and SAFE the least, showing a **wASR** of 45.25% and 64.80% respectively. Overall, the models relying on an instructions-based representation (such as BinFinder and SAFE) seem to be less robust against our attack. As visible in Figure 5, this same analysis holds when increasing the pool size to 512.

Key takeaway. In the untargeted case, the performance at various levels of K is comparable across most models, with SAFE being the least robust and ZEEK the most robust model. While in the targeted case, ZEEK is the most robust model and BinFinder the least robust.
Top performance on clean data does not correspond to greater robustness. Models showing greater **Recall pre attack**, often exhibit lower **Recall post attack**, whereas those with poorer clean data performance, tend to have higher **Recall post attack**.

6.2.2 Transferability

Our attack strategy employs a greedy optimizer that iteratively applies transformations based on feedback from the target model. An intriguing aspect to explore is whether an adversarial example generated against one model can be leveraged to target all the other models under analysis. We define this property as the Transferability Success Rate (**TSR**). Furthermore, we want to investigate how models react against adversarial examples designed for other models. We define this property as the Vulnerability Rate (**VR**).

To evaluate these two properties, we compare using a simple baseline that applies a sequence of random transformations to the query function f_Q. The random baseline is run for the same number of iterations as the greedy procedure, and the experiment is repeated three times. Note that this baseline can be used to compare a random application of transformations against our optimizer, see the difference between these results and the one in Section 6.1.

Table 3 presents the results in terms of **wASR** of the transferability experiment in the untargeted case, together with the average **wASR** and the standard deviation for the random baseline. Table 4 presents the results for the transferability experiment in the targeted scenario.

Transferability Success Rate

When considering the untargeted scenario, the **TSR** values indicate that adversarial examples generated against PalmTree and BinFinder are the most transferable to other models, with **TSR** values of 73.35% and 70.65% respectively. We attribute these results to the transformations applied when attacking PalmTree and BinFinder. In these cases, **SA** and **DBA** are the most used transformations. These two modify most of the features considered by the target models (namely, the content and the topology of the CFG). We will further discuss this aspect in the next section where we will peruse the frequency of transformations applied against each model. Finally, all the transferred examples demonstrate higher effectiveness when compared to the random baseline, with an average **TSR** of 62.12% vs 50.22% respectively.

In the targeted scenario, the **TSR** results show that adversarial examples transfer less effectively compared to those from the untargeted case. This is expected, given that targeted attacks are generally less effective than untargeted ones, as discussed in Section 6.1. Nevertheless, the transferred examples outperform those produced by the random baseline. Finally, similar to the untargeted case, adversarial examples generated against PalmTree remain the most effective, with a **TSR** of 10.95%.

Vulnerability Rate

Interestingly, the **VR** results show that BinFinder stands out as the most robust model, with a lower **VR** of 22.81%. We believe that adversarial examples targeting BinFinder must possess unique features that are

Table 4: Transferability matrix for the targeted attack case, considering $|P| = 128$ and $K = 10$. In the rows, we indicate the model for which the adversarial examples were created, and in the columns, the model on which the examples are tested. Each value represents the **wASR**.

	Gemini	GMN	ZEEK	BinFinder	SAFE	jTrans	Trex	PalmTree	**TSR**
Gemini	■	10.80	10.05	7.62	9.03	10.10	8.08	8.04	9.10
GMN	10.80	■	9.57	7.83	11.72	11.05	7.92	10.30	9.88
ZEEK	6.35	7.78	■	6.93	10.49	11.50	6.68	6.56	8.04
BinFinder	5.61	5.96	9.52	■	9.57	10.02	7.79	6.38	7.84
SAFE	8.65	9.93	10.47	8.10	■	12.0	12.28	9.92	10.19
jTrans	8.20	8.88	8.85	7.52	12.3	■	10.50	8.47	9.25
Trex	7.70	9.53	8.97	7.47	16.70	13.60	■	11.14	10.73
PalmTree	10.20	13.38	7.31	9.49	11.72	11.40	13.18	■	10.95
VR	8.22	9.47	9.25	7.85	11.65	11.38	9.49	8.69	■
random	6.65 ± 0.11	6.73 ± 0.22	8.67 ± 0.24	7.26 ± 0.03	8.57 ± 0.09	9.89 ± 0.10	6.38 ± 0.32	6.09 ± 0.08	7.53

not present in those generated against other models. We will further discuss this insight in the next section. Surprisingly, ZEEK is the least robust model against transferred adversarial examples, with a **VR** value of 83.49%. Generally, transferred examples are more effective than those generated using the random baseline, except Gemini, where the corresponding **VR** value is comparable to the **wASR** achieved with the random baseline.

In the targeted scenario, all models perform similarly, with BinFinder being the most robust (**VR** of 7.85%) and SAFE the least (**VR** of 11.65%). Despite these low values, transferred examples still outperform the random baseline.

Key takeaway. The transferability of adversarial examples across models is more effective in the untargeted context than in the targeted scenario. Additionally, the distribution of applied transformations directly affects the success of transferring an adversarial example to another model.

6.3 RQ3: Common Model Behaviors

In this section, we discuss whether or not an adversarial example can reveal common behaviors that the model applies when analyzing binary functions.

6.3.1 Distribution of Applied Transformations

Figure 7 shows the distribution of applied transformations across the various models together with the results in average in the untargeted scenario when querying a pool of 128 functions. The results are calculated by considering, for each model, only the adversarial examples that succeed according to the **ASR@*4*** metric.

Looking at the average results, it is evident that **SA** and **DBA** are the transformations that most contribute to the success of adversarial examples, being applied in 83.28% of the time. In contrast, **IR**, an in-place transformation that neither alters the CFG nor introduces new instructions, is the least applied.

Our attack procedure can identify key aspects of the target model, particularly the target architecture and the function representation strategy.

As shown in Figure 7, neither Gemini nor GMN are affected by **IR**. This is reasonable, as both models utilize a GNN architecture that does not consider the position of the single instructions. In contrast, our strategy primarily focuses on transformations that either insert new nodes or new instructions. Specifically, when targeting Gemini, **DBA** is the most common choice, as it introduces both new instructions and new nodes into the CFG.

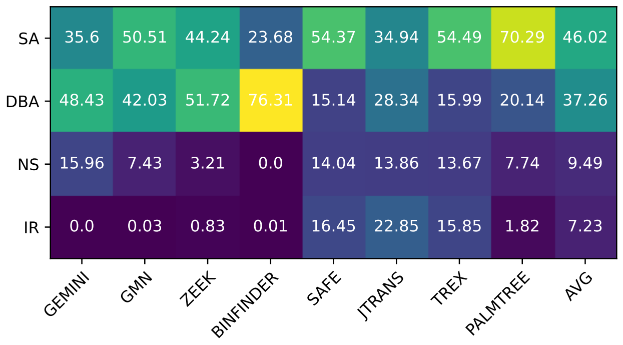


Figure 7: Distribution of applied transformations in the untargeted scenario, with $K = 10, \lambda = 0$ and querying a pool P of 128 functions. The AVG column shows the average distribution across the various models.

When moving to models that use instruction-based function representations (i.e., ZEEK and BinFinder), **SA** and **DBA** become the most frequently applied transformations. This is due to both techniques adding new strands to the function's body.

SAFE along with jTrans and Trex, utilize architectures that consider the position of instructions within a function. As a result, **IR** is chosen more often compared to its usage in the previously mentioned models. It is interesting to note that the distribution of percentages against jTrans is more balanced. This reflects the fact that jTrans accounts for both the instructions and their positions within the function, as well as the CFG nodes, with *jump* instructions being modeled differently from other instructions.

Although PalmTree is built on an LSTM architecture similar to SAFE, **SA** is chosen significantly more often in this case (70.29% vs 54.37%), whereas **IR** is rarely applied, unlike when attacking SAFE. We believe this is due to the more sophisticated instruction embedding technique implemented by PalmTree, which causes our attack to focus more on transformations that insert new instructions rather than the ones manipulating the CFG or swapping existing instructions.

Table 5: Untargeted and Targeted attack at $K = 10$ with transformations applied individually, considering $|P| = 128$ and $\lambda = 0$. The $\Delta\%$ value represents the percentage improvement in terms of **wASR** achieved by considering all transformations (ALL) compared to applying each transformation in isolation.

		Models																			
		Gemini					SAFE					jTrans					AVG				
		ALL	**IR**	**NS**	**DBA**	**SA**	**ALL**	**IR**	**NS**	**DBA**	**SA**	**ALL**	**IR**	**NS**	**DBA**	**SA**	**ALL**	**IR**	**NS**	**DBA**	**SA**
UNTARGETED	**Recall post attack**	0.07	0.71	0.32	0.07	0.07	0.02	0.93	0.79	0.27	0.01	0.04	0.85	0.57	0.22	0.02	0.04	0.83	0.56	0.19	0.03
	wASR	92.99	30.08	68.24	92.97	92.16	98.30	7.58	30.35	73.47	98.72	96.0	14.67	43.34	78.29	97.63	95.76	17.44	47.31	88.39	96.17
	$\Delta\%$	■	67.65	26.62	0.02	0.89	■	92.29	69.13	25.26	-0.43	■	84.72	43.34	18.45	-1.70	■	81.79	50.60	7.70	-0.43
TARGETED	**wASR**	48.25	7.61	13.23	38.48	28.16	64.80	7.52	11.95	29.57	50.45	45.25	7.73	12.49	28.76	33.59	52.77	7.62	12.56	32.27	37.40
	$\Delta\%$	■	84.23	72.58	20.25	41.64	■	88.40	81.56	54.37	22.15	■	82.92	72.40	36.44	25.77	■	85.56	76.20	38.85	29.17

We emphasize that **SA** and **DBA** are universal transformations capable of modifying nearly all the features considered by the target models. This is evidenced not only by the previously discussed percentages but also by the intrinsic nature of these two transformations, which add new nodes and instructions to the modified function. Consequently, this explains why the adversarial examples generated against PalmTree and BinFinder transfer most effectively to the other models, as demonstrated by the **TSR** results discussed in the previous section.

Key takeaway. The architecture of the target model and its function representation strategy significantly affect the type of transformation selected by our attack strategy. Specifically, transformations affecting the CFG are predominant when attacking models considering the CFG, while transformations inserting new instructions or altering their order are predominant when targeting models that do not consider the CFG topology.

6.3.2 Transformations in Isolation

We now analyze the impact of individual transformations, focusing on three representative models—Gemini, SAFE, and jTrans —selected from the considered DNN architectures. For this evaluation, we run our greedy optimization strategy disabling all transformations but the one tested. To ensure a fair comparison, the number of candidates per iteration tested in this evaluation matches the number of candidates for the corresponding transformation in the main approach. For example, if transformation **IR** has 50 candidates per iteration in the main approach, the same number is used when evaluating **IR** alone.

Table 5 presents the results for the untargeted and targeted attacks performed considering the transformations in isolation. These confirm the findings from Section 6.3.1. Specifically, in the untargeted scenario, **DBA** and **SA**, the most frequent transformations for Gemini SAFE, and jTrans respectively, are also the most effective when used in isolation to target these models. The $\Delta\%$ measure, which represents the percentage improvement in **wASR** when combining transformations compared to applying each transformation individually, is -0.43 on SAFE and -1.70 on jTrans. This means that when considering **SA** alone, we can obtain results comparable to the main approach in terms of **wASR**. The results in the targeted scenario confirm the effectiveness of **DBA** and **SA** in attacking the considered models. However, as indicated by the $\Delta\%$ values, the transformations alone are insufficient to achieve the same results as when they are combined.

6.3.3 Efficiency Analysis

We analyze our attack's efficiency by comparing the average time needed to generate adversarial examples using different transformations, both individually and combined.

We observed that the execution time remains consistent between targeted and untargeted attacks since both run for a fixed number of iterations. Therefore, we focus only on the untargeted case. The main time factors in our attack are identifying perturbation positions, the number of candidate adversarial examples tested per iteration, and the overhead from binary function similarity calculations.

When using all transformations, the average execution time on Gemini, SAFE, and jTrans is 25.55 ± 2.67 minutes, indicating comparable generation time across models, with the procedure being more efficient against Gemini (taking 21.82 minutes in average) and least efficient against SAFE (taking 27.91 minutes in average). Specifically, one iteration takes on average 50.12 ± 6.03 seconds.

When applied in isolation, transformations exhibit varying execution times. **IR** is the most efficient, taking 1.77 ± 0.39 minutes on average overall and 3.15 ± 1.42 seconds per iteration, while **DBA** is the least efficient, taking 17.17 ± 4.65 minutes overall and 34.07 ± 9.80 seconds per iteration.

6.3.4 Qualitative Analysis

Figure 8 presents a comparison between the CFGs of the original query function f_Q (shown in Figure 8(a)) and its adversarial versions generated after an untargeted attack with $\lambda = 0$ on three models: Gemini (Figure 8(b)), SAFE (Figure 8(c)), and jTrans (Figure 8(d)).

The sequence of transformations applied to generate the adversarial example in Figure 8(b) demonstrates that Gemini considers both the CFG topology and the individual instructions, as noted in the previous Section. A manual analysis of the adversarial example shows that the greedy optimizer typically first alters the topology with a combination of **DBA** and **NS** transformations, followed by adding new instructions using **SA**.

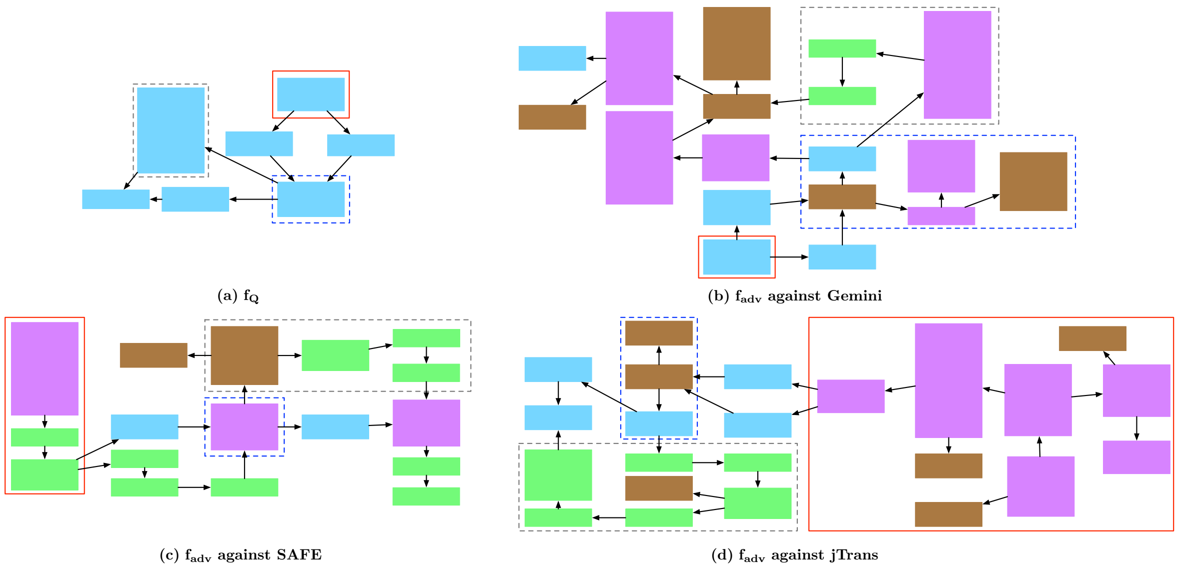

(a) f_Q
(b) f_{adv} against Gemini
(c) f_{adv} against SAFE
(d) f_{adv} against jTrans

Figure 8: CFGs of the three binary functions in case of **untargeted** attack with $\lambda = 0$ against Gemini, SAFE, and jTrans. **(a)** shows the CFG of f_Q, while **(b)**, **(c)**, and **(d)** show the CFGs of the adversarial f_{adv} targeting Gemini, SAFE, and jTrans respectively. The **red** rectangle marks the function's entry point; **purple** , **green** , and **brown** blocks indicate the use of **SA**, **NS**, and **DBA**, respectively. Dotted rectangles highlight how instructions from a block in f_Q are distributed across multiple blocks in f_{adv} after the attack.

The adversarial example in Figure 8(c) shows that, when attacking SAFE, most modifications are focused around the prologue of f_Q. Specifically, the function's entry point is modified twice: first, by adding new instructions through the **SA** transformation, and then by splitting the final portion of the block with **NS**. Interestingly, one of the neighbors of the entry point is also modified using **NS**. This aligns with what has been observed in [45], which shows that SAFE usually focuses its analysis on functions' prologue.

Figure 8(d) shows the adversarial function f_{adv} generated from f_Q when targeting jTrans. As noted in the previous section, the distribution of the different transformations is relatively balanced. Like SAFE, the attack strategy focuses on the entry point, applying several transformations that add both new nodes and instructions.

The analysis of single nodes reveals how the greedy procedure tailors its modifications based on the target model. For the node shown in gray, when targeting Gemini, the modification involves adding a strand and splitting the final part using **NS**. In contrast, for SAFE, the original instructions are spread across five new nodes, including a dead branch with additional instructions, created through a combination of **NS** and **DBA**. When attacking jTrans, the approach resembles the one used for SAFE, but applies **NS** more frequently. This last observation is expected because, as outlined in the previous section, jTrans indirectly accounts for the CFG of a function by modeling *jump* instructions differently than other types of instructions. The node in blue remains mostly unchanged when targeting SAFE but undergoes similar modifications when targeting both Gemini and jTrans.

Key takeaway. The qualitative analysis confirms the findings from the distribution of applied transformations. Moreover, it uncovers hidden aspects of the target models, such as the tendency of certain models to concentrate on the prologue of functions.

6.4 Non-ML Approaches

In this section, we assess the robustness of non-ML methods for comparing binary functions. For this evaluation, we consider:

- **GSIZE**. A simple approach that compares two functions based on the number of basic blocks.
- **GEDIT**. A simple approach that compares the CFGs of two functions using an approximated labeled edit distance measure [19] (i.e., the number of changes in terms of nodes and edges edits to transform a source CFG into a target one).
- **Catalog1**[3]. This approach uses fuzzy hashing, directly leveraging raw binary information. Specifically, it applies MinHash [7] to encode groups of four consecutive function bytes into a fixed-size signature, on which similarity is computed using Jaccard.

We excluded other approaches, such as PSS [4] and BinDiff [18], which operate at the program level rather than the function level, as they fall outside the scope of our threat model.

Table 6 presents the results for the untargeted and targeted attacks against the three considered approaches. For the untargeted scenario, it is evident

[3]https://www.xorpd.net/pages/fcatalog.html

Table 6: Untargeted and Targeted attacks against non-ML approaches, considering $K = 10$, $|P| = 128$, and $\lambda = 0$.

		Untargeted				Targeted			
		GSIZE	GEDIT	Catalog1	AVG	GSIZE	GEDIT	Catalog1	AVG
	wASR	82.77	81.59	40.02	68.13	27.17	18.65	42.85	29.56
	Recall pre attack	0.58	0.49	0.69	0.59	■	■	■	■
	Recall post attack	0.17	0.18	0.60	0.32	■	■	■	■
@1	**INIT**	79.92	84.85	64.79	76.52	18.05	26.41	46.22	30.23
	ASR	91.41	87.96	53.52	77.63	47.16	25.80	73.71	48.89
	M-Instrs	108.08	4.26	94.87	69.07	35.91	11.68	55.64	34.41
	M-Nodes	40.60	2.0	5.60	16.07	14.16	3.23	1.94	6.44
@2	**INIT**	62.34	77.83	44.57	61.58	10.07	17.47	22.41	16.65
	ASR	87.31	86.66	50.0	74.66	33.40	18.98	52.29	34.89
	M-Instrs	111.79	4.25	95.82	70.62	44.65	15.67	63.32	41.21
	M-Nodes	41.93	0	5.55	15.83	17.50	4.30	2.24	8.01
@3	**INIT**	25.17	43.33	15.29	27.93	3.19	13.55	12.05	9.60
	ASR	80.22	81.95	36.82	66.33	17.45	15.36	28.49	20.43
	M-Instrs	117.77	4.27	102.15	74.73	61.18	19.19	67.59	49.32
	M-Nodes	43.94	2.0	5.15	17.03	23.90	5.26	2.57	10.58
@4	**INIT**	0	0	0	0	1.40	12.15	8.86	7.47
	ASR	72.13	69.81	19.72	53.89	10.67	14.46	16.93	14.02
	M-Instrs	127.10	4.27	109.67	80.35	75.16	20.39	75.81	57.12
	M-Nodes	47.23	2.0	5.07	18.10	29.18	5.59	2.76	12.51

that graph-based approaches are not robust against our strategy, with a **wASR** of 82.77% for GSIZE and 81.59% for GEDIT. When moving to Catalog1, this presents better robustness compared to the other considered approaches, with a **wASR** of 40.02%. In the targeted scenario, all the considered approaches exhibit greater robustness to our attack compared to DNN-based solutions. Specifically, the average **wASR** against non-ML methods is 29.56% (compared to 57.06% for DNN-based solutions), with Catalog1 being the least robust (42.85%) and GEDIT the most robust (18.65%).

The previous results highlight the better robustness of non-ML approaches compared to DNN-based ones against our attack. However, it is worth noting that non-ML solutions present lower **Recall pre attack** values, showing poor performance on clean data.

The poor attack performance in the targeted case against GSIZE and GEDIT is due to the fact that our transformations cannot remove nodes from the CFG. As a result, when the target function has fewer nodes than the query, the attack fails. This also highlights a significant limitation of these similarity methods, as they struggle to assign high similarity scores to functions that are semantically similar but topologically different.

Regarding Catalog1, its robustness in the targeted case is comparable to that of DNN-based systems; however, it shows remarkable robustness in the untargeted case. Probably the set of our transformations is unable to effectively modify the representation used by Catalog1. The results @1, where **INIT@*1*** is higher than **ASR@*1***, highlight that, for certain queries f_Q, at least one of the variants initially ranked outside the top-K is ranked in the top-K of f_{adv}. This implies that Catalog1 fails to assign high similarity scores to semantically similar functions and that our attack indirectly improves the performance of the system on a limited number of clean data.

7 Related Works

In this section, we first discuss techniques for generating adversarial examples against image classifiers and NLP models; then, we move to approaches targeting models for source code analysis. Finally, we discuss attacks against malware detectors and models for binary analysis.

7.1 Attacking Image Classifiers and NLP Models

Adversarial attacks were first introduced against models for image classification, with early works [5,10,20,43] providing white-box, gradient-based methods that add minimal perturbations to fool models with high confidence. Chen et al. [11] propose various black-box decision-based attacks against image classifiers involving the estimation of gradient direction.

Jia et al. [22] attack reading comprehension models by introducing sentences that can deceive the target models while maintaining the original semantics of the paragraph. More recently, the solutions in [25, 27] proposed to attack NLP models by finding replacements of words composing the input sequence, using BERT-based strategies.

7.2 Attacking Models for Source Code Analysis

Methods for attacking models for source code analysis are mainly based on applying semantics-preserving perturbations at the source code level, thus having limited applicability in the binary function similarity context.

Yefet et al. [47] propose a white-box approach that, using a gradient-driven method, iteratively changes the names of variables defined within a function in all their occurrences, until a misclassification occurs. Differently, Zhang et al. [49] target code clone detectors using semantics-preserving transformations, combined using common optimization heuristics, alongside a reinforcement learning-based approach for searching clones that could evade the detection.

7.3 Attacking Models for Binary Code Analysis

The solution proposed by Pierazzi et al. [39] targets Android malware classifiers and is based on software transplantation. Here, benign snippets of code that can trigger the classifier features are injected into the malware sample to cause a misclassification using a gradient-guided approach. Moreover, the snippets are injected into portions of code that are never executed, to guarantee the preservation of the semantics.

Lucas et al. [30] attack malware classifiers based on raw bytes. Their solution is based on combining different semantics-preserving perturbations both in a black-box and a white-box context. The proposed transformations are a subset of ours, as they include **IR** and **NS**; however, while these transformations show clear effectiveness when targeting malware classifiers (and also commercial solutions), they may show poor performance when targeting binary function similarity models; indeed, it is evident from our results that a crucial point in attacking binary similarity models is the need of inserting new instructions into the function being modified.

FuncFooler [21] is a recent unpublished work targeting binary function similarity models in the context of function search. It consists of an instruction-insertion strategy to modify a binary function at a set of fixed locations determined in advance; to guarantee semantics preservation, possible side effects are corrected a posteriori. The set of possible instructions is computed considering the instructions of the functions in the pool. Differently from our approach, FuncFooler explores only one class of transformations (which can be considered a subset of **SA** where a single instruction is inserted at each step), without altering the topology of the CFG. Furthermore, it only studies untargeted attacks.

Capozzi et al. [9] is a recent unpublished work proposing two solutions targeting a subset of the binary function similarity models we considered. Their black-box approach consists of a greedy solution, feasible both in a targeted and untargeted context, that iteratively inserts new instructions into dead branches, whose locations are fixed in advance. Differently from FuncFooler [21], the set of possible instructions is dynamically updated using a heuristic based on instruction embeddings. The proposed white-box attack substitutes the aforementioned heuristic with a gradient-guided instruction insertion strategy. We highlight that [9] relies on a set of transformations that is strictly a subset of ours (as [9] uses only **DBA** with a single instruction added) and it tests its approach only against three models (namely, Gemini, SAFE, and GMN).

PELICAN [50] is a novel white-box attack that leverages natural backdoors of attacked models to identify instructions that, once inserted into functions, can induce misclassification. This attack has been tested against models for different binary analysis tasks, including function naming, compiler provenance, and binary function similarity. In the context of the latter, the proposed methodology has only been tested against three models—specifically, Gemini, SAFE, and Trex.

We emphasize that the last two solutions differ significantly from our work. Firstly, they do not address the function search task. Specifically, Capozzi et al. [9] target the similarity function implemented by the target model, whereas PELICAN [50] focuses on attacking the loss function implemented by the target model. Another key difference lies in the use of variants of the query function during the optimization process, which is not the case in either of the other two approaches.

8 Discussion

We now discuss the practical impacts of our study and the limitations of our evaluation setting.

8.1 Practical Impacts

As outlined in Section 1, binary function similarity systems play a crucial role in various security-sensitive scenarios, including vulnerability detection, plagiarism identification, and malware analysis. These systems help automate the process of comparing binary functions, making it easier to identify code reuse, detect security flaws, and uncover malicious behaviors. In practical scenarios, such systems are integrated into tools used by reverse engineers. Notable examples include plugins such as YARASAFE[4] and BinaryAI[5]. These plugins facilitate the use of traditional reverse engineering tools by providing automated capabilities that can reduce manual effort.

Our threat model represents a practical scenario where a remotely deployed binary function similarity system operates as a black-box model, providing only similarity scores. While this represents a worst-case assumption for the attacker, our findings reveal that these systems remain vulnerable to our attack, which is relatively simple to implement in practical contexts. This applies to both targeted (e.g., disguising malicious code as benign) and untargeted (e.g., hiding vulnerable or plagiarized functions) attacks, posing serious threats in real-world scenarios, even for models explicitly designed to handle obfuscated functions, such as BinFinder [40] and Trex [38].

8.2 Limitations

We examine the limitations of our work, focusing on the dataset and transformations used. Our dataset is smaller than benchmarks like BinaryCorp [44], BinKit [24], and those used by Marcelli et al. [31]. However, these benchmarks are typically used to evaluate the performance of binary function similarity systems on clean data. Due to the computational cost of generating adversarial examples (see Section 6.3.3), using such large datasets is impractical in our scenario. However, the number of open-source projects used to generate our codebase aligns with standard practices in the field, as prior studies [29, 33, 38, 40] typically extract functions from 1 to 10 projects.

As most binary function similarity systems are trained on ELF `amd64` functions compiled from C code, we limited our evaluation to this setting. However, our attack is architecture-agnostic and can be extended to other ISAs by adapting the transformations. We expect our findings to generalize, as the attack does not depend on ISA-specific traits. Furthermore, variations in source code languages may alter assembly representations, and if models are not trained on such binaries, their performance may degrade, potentially increasing the ASR. With respect to compilers, binary function similarity systems are typically trained considering multiple compilers as well as different versions of the same compiler. While our dataset accounts for the first aspect, we did not explore the latter. However, [31] observed a slight performance drop on clean data when comparing functions compiled with different versions of the same compiler, suggesting that the **ASR** would likely increase in such scenarios.

Finally, our set of transformations may have little to no effect against symbolic execution-based methods.

[4]https://github.com/lucamassarelli/yarasafe
[5]https://github.com/binaryai/plugins

However, these methods are often impractical due to their inefficiency; indeed, comparing binary functions using symbolic execution may lead to path explosion, making it impractical in real-world scenarios.

9 Conclusions and Future Works

In this paper, we presented the first large-scale analysis of the robustness of binary function similarity models against adversarial attacks highlighting the need for a trade-off between performance and robustness.

We demonstrated that a simple greedy strategy, when enriched with a wide set of transformations, can mount untargeted attacks with very high success rates on all considered models, particularly those showing top performance on clean data. Conversely, models that initially perform poorly seem to be more resistant to adversarial examples. On the targeted front, our attacks performed slightly worse, but they were still successful in more than half of the instances considered.

We investigated several additional aspects. First, we showed that adversarial examples transfer across models, with a significantly higher success in the untargeted case rather than the targeted. Secondly, we demonstrated that the set of transformations we considered was effective in modifying most of the key features considered by the target models, with two transformations making a particularly strong contribution to the success of the attack. Finally, manual analysis of adversarial examples uncovered hidden behaviors in the models, revealing that they focus their analysis on specific portions of the functions.

Our research opens several new research avenues, particularly in the context of defense strategies. Rather than focusing solely on adversarial training— which may enhance model robustness against our attack but does not guarantee protection against zero-day threats— we argue that greater emphasis should be placed on proposing inherently robust function representation methods.

Acknowledgments

This work was partially supported by the Italian MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU through projects SERICS (PE00000014) and Rome Technopole (ECS00000024).

References

[1] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation*, 12:S61–S71, 2015.

[2] Fiorella Artuso, Giuseppe Antonio Di Luna, Luca Massarelli, and Leonardo Querzoni. In nomine function: Naming functions in stripped binaries with neural networks. *arXiv preprint arXiv:1912.07946*, 2019.

[3] Fiorella Artuso, Marco Mormando, Giuseppe Antonio Di Luna, and Leonardo Querzoni. Binbert: Binary code understanding with a fine-tunable and execution-aware transformer. *IEEE Transactions on Dependable and Secure Computing*, pages 1–18, 2024.

[4] Tristan Benoit, Jean-Yves Marion, and Sébastien Bardin. Scalable program clone search through spectral analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, (ESEC/FSE '23)*, pages 808–820. ACM, 2023.

[5] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Srndic, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion Attacks against Machine Learning at Test Time. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD '13)*, volume 8190, pages 387–402. Springer, 2013.

[6] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. In *Pattern Recognition*, volume 84, pages 317–331, 2018.

[7] Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE, 1997.

[8] Ying Cao, Ruigang Liang, Kai Chen, and Peiwei Hu. Boosting neural networks to decompile optimized binaries. In *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC '22)*, pages 508–518. ACM, 2022.

[9] Gianluca Capozzi, Daniele Cono D'Elia, Giuseppe Antonio Di Luna, and Leonardo Querzoni. Adversarial attacks against binary similarity systems. *arXiv preprint arXiv:2303.11143*, 2023.

[10] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP '17)*, pages 39–57. IEEE, 2017.

[11] Jianbo Chen, Michael I. Jordan, and Martin J. Wainwright. Hopskipjumpattack: A query-efficient decision-based attack. In *Proceeedings of the 41st IEEE Symposium on Security and Privacy (SP '20)*, pages 1277–1294. IEEE, 2020.

[12] Hanjun Dai, Bo Dai, and Le Song. Discriminative Embeddings of Latent Variable Models for Structured Data. In *Proceedings of the 33rd International Conference on Machine Learning (ICML '16)*, volume 48, pages 2702–2711, 2016.

[13] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*, pages 266–280, 2016.

[14] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Functionality-preserving black-box optimization of adversarial windows malware. *IEEE Transactions on Information Forensics and Security*, 16:3469–3478, 2021.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[16] Steven H.H. Ding, Benjamin C.M. Fung, and Philippe Charland. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP '19)*, pages 472–489. IEEE, 2019.

[17] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. Deepbindiff: Learning program-wide code representations for binary diffing. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium NDSS '20*. The Internet Society, 2020.

[18] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects (English version). In *Proceedings of the Symposium sur la sécurité des technologies de l'information et des communications (SSTIC '05)*, page 3, 2005.

[19] Andreas Fischer, Kaspar Riesen, and Horst Bunke. Improved quadratic time approximation of graph edit distance by combining hausdorff matching and greedy assignment. *Pattern Recognition Letters*, 87:55–62, 2017.

[20] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[21] Lichen Jia, Bowen Tang, Chenggang Wu, Zhe Wang, Zihan Jiang, Yuanming Lai, Yan Kang, Ning Liu, and Jingfeng Zhang. Funcfooler: A practical black-box attack against learning-based binary code similarity detection methods. *arXiv preprint arXiv:2208.14191*, 2022.

[22] Robin Jia and Percy Liang. Adversarial Examples for Evaluating Reading Comprehension Systems. In *Proceedings of the 22nd Conference on Empirical Methods in Natural Language Processing (EMNLP '17)*, pages 2021–2031, 2017.

[23] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*, pages 329–338, 2013.

[24] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering*, 49(4):1661–1682, 2022.

[25] Dianqi Li, Yizhe Zhang, Hao Peng, Liqun Chen, Chris Brockett, Ming-Ting Sun, and Bill Dolan. Contextualized Perturbation for Textual Adversarial Attack. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT '21)*, pages 5053–5069, 2021.

[26] Juncheng Li, Shuhui Qu, Xinjian Li, Joseph Szurley, J Zico Kolter, and Florian Metze. Adversarial Music: Real world Audio Adversary against Wakeword Detection System. In *Proceedings of the 32nd Annual Conference on Neural Information Processing Systems (NeurIPS '19)*, pages 11908–11918, 2019.

[27] Linyang Li, Ruotian Ma, Qipeng Guo, Xiangyang Xue, and Xipeng Qiu. BERT-ATTACK: adversarial attack against BERT using BERT. In *Proceedings of the 25th Conference on Empirical Methods in Natural Language Processing (EMNLP '20)*, pages 6193–6202, 2020.

[28] Xuezixiang Li, Yu Qu, and Heng Yin. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*, page 3236–3251, 2021.

[29] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *Proceedings of the 36th International Conference on Machine Learning (ICML '19)*, pages 3835–3845, 2019.

[30] Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K Reiter, and Saurabh Shintre. Malware Makeover: Breaking ML-based Static Analysis by Modifying Executable Bytes. In *Proceedings of the 16th ACM Asia Conference on Computer and Communications Security (AsiaCCS '21)*, pages 744–758, 2021.

[31] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How Machine Learning Is Solving the Binary Function Similarity Problem. In *Proceedings of the 31st USENIX Security Symposium (SEC '22)*, pages 2099–2116. USENIX Association, 2022.

[32] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*, pages 1–11, 2019.

[33] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Function Representations for Binary Similarity. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2259–2273, 2022.

[34] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed Representations of Words and Phrases and their Compositionality. In *Proceedings of the 27th Annual Conference on Neural Information Processing Systems (NeurIPS '13)*, pages 3111–3119, 2013.

[35] Radare ORG. Radare2. https://github.com/radareorg/radare2, 2024.

[36] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. Probabilistic naming of functions in stripped binaries. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC '20)*, page 373–385. ACM, 2020.

[37] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. Stateformer: fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*, pages 690–702. ACM, 2021.

[38] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Learning approximate execution semantics from traces for binary function similarity. *IEEE Transactions on Software Engineering*, 49(4):2776–2790, 2023.

[39] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP '20)*, pages 1332–1349. IEEE, 2020.

[40] Abdullah Qasem, Mourad Debbabi, Bernard Lebel, and Marthe Kassouf. Binary function clone search in the presence of code obfuscation and optimization over multi-cpu architectures. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security (AsiaCCS '23)*, page 443–456, 2023.

[41] Noam Shalev and Nimrod Partush. Binary similarity detection using machine learning. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, pages 42–47, 2018.

[42] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP '16)*, pages 138–157. IEEE, 2016.

[43] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[44] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. JTrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22)*, pages 1–13, 2022.

[45] Wai Kin Wong, Huaijin Wang, Zongjie Li, and Shuai Wang. Binaug: Enhancing binary similarity analysis with low-cost input repairing. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024*, pages 7:1–7:13. ACM, 2024.

[46] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, pages 363–376, 2017.

[47] Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. In *Proceedings of the ACM on Programming Languages (OOPSLA '20)*, volume 4, pages 1–30, 2020.

[48] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE Transactions on Neural Networks and Learning Systems*, 30(9):2805–2824, 2019.

[49] Weiwei Zhang, Shengjian Guo, Hongyu Zhang, Yulei Sui, Yinxing Xue, and Yun Xu. Challenging machine learning-based clone detectors via semantic-preserving code transformations. *IEEE Transactions on Software Engineering*, pages 1–18, 2023.

[50] Zhuo Zhang, Guanhong Tao, Guangyu Shen, Shengwei An, Qiuling Xu, Yingqi Liu, Yapeng Ye, Yaoxuan Wu, and Xiangyu Zhang. PELICAN: exploiting backdoors of naturally trained deep learning models in binary code analysis. In *Proceedings of the 32nd USENIX Security Symposium (SEC '23)*, pages 2365–2382. USENIX Association, 2023.

A

Algorithm 1 presents our attack procedure. The first adversarial example f_{adv} is the query function f_Q itself (line 1). We then initialize the set of candidate strands that can be inserted into the adversarial function either using **DBA** or **SA** (line 3). Then, during the iterative procedure, we first identify possible positions to perturb (line 7) and then enumerate all the possible transformations that can be applied in the identified positions. Specifically, we apply a transformation tr at the position pos, for every pair $\langle tr, pos\rangle \in TR \times POS$ (lines 8 - 16). We then proceed to evaluate the objective function defined in Equation 2, considering the set of candidates $CAND$ and the set of target variants V (line 17). Finally, we select the new adversarial example f_{adv} according to the value of ε (lines 18 - 22) and update the set of candidate strands (line 24). The final adversarial example f_{adv} (line 26) is, among all f_{adv} generated at the end of each iteration, the one that produced the highest value for the objective function.

B

Below, we present the results of the robustness analysis for the evaluated models, considering various values of K, P, and λ. Specifically, we set $K \in \{10, 100\}$ for untargeted attacks and $K \in \{5, 10\}$ for targeted attacks. Additionally, we consider pools with sizes $|P| \in \{32, 128, 512, 1000\}$ and $\lambda \in \{0, 0.01, 0.3\}$.

We report the results for the untargeted case in Tables 7, 8, 9, 10 and 11, while the ones for the targeted scenario in Tables 12, 13, 14, and 15.

Table 7: Untargeted attack at $K = 10$ when considering a pool of size 32 with $\lambda \in \{0, 0.3\}$. In column AVG we report the average of the measures across all models.

			Models																	
			Gemini		GMN		ZEEK		BinFinder		SAFE		jTrans		Trex		PalmTree		AVG	
		λ	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**
K=10		**Recall pre attack**	0.87	0.87	0.98	0.98	0.92	0.92	0.96	0.96	0.99	0.99	0.94	0.94	0.99	0.99	0.98	0.98	0.95	0.95
		Recall post attack	0.15	0.85	0.12	0.90	0.33	0.82	0.12	0.96	0.07	0.97	0.31	0.90	0.14	0.98	0.11	0.97	0.17	0.92
		wASR	85.07	15.47	88.05	9.60	66.75	18.14	88.28	3.77	93.15	3.32	69.15	9.98	86.38	2.23	88.78	3.13	83.20	8.20
	@1	**INIT**	34.67	34.67	4.70	4.70	22.26	22.26	9.80	9.80	5.0	5.0	21.0	21.0	2.58	2.58	5.81	5.81	13.23	13.23
		ASR	91.18	37.70	97.80	18.50	82.65	39.68	91.80	10.10	96.90	8.75	94.90	28.40	93.04	5.80	93.99	9.22	92.78	19.77
		M-Instrs	236.20	9.83	210.59	12.52	198.55	12.35	226.78	7.17	240.06	16.85	134.14	12.63	198.86	16.19	522.43	12.16	245.95	12.46
		M-Nodes	27.14	1.61	16.87	2.50	24.47	2.04	41.56	0.22	13.66	2.20	16.76	2.03	11.17	1.91	16.70	2.37	21.04	1.86
	@2	**INIT**	14.53	14.53	1.0	1.0	6.96	6.96	3.70	3.70	0.60	0.60	3.80	3.80	0.40	0.40	1.50	1.50	4.06	4.06
		ASR	88.78	18.35	95.70	9.90	75.23	20.65	89.70	4.0	95.60	3.22	84.10	10.60	90.46	2.20	92.08	2.61	88.96	8.94
		M-Instrs	236.46	10.73	209.58	11.59	194.60	16.41	226.40	5.75	241.03	22.44	136.04	15.97	200.69	20.77	522.57	12.50	245.92	14.52
		M-Nodes	27.41	1.68	17.01	2.67	24.11	2.31	41.46	0.30	13.65	2.16	17.07	2.49	11.25	1.64	16.80	2.65	21.10	1.99
	@3	**INIT**	3.71	3.71	0.30	0.20	1.03	1.03	0.60	0.60	0.10	0.10	0.0	0.0	0.20	0.20	0.40	0.40	0.79	0.79
		ASR	83.47	4.84	86.0	6.30	61.53	8.50	87.0	0.90	91.70	0.91	56.10	0.80	83.40	0.60	86.37	0.60	79.45	2.93
		M-Instrs	238.68	10.77	210.40	11.25	188.42	21.77	226.62	12.22	244.53	30.11	142.26	14.50	204.03	17.17	521.36	14.0	247.04	16.47
		M-Nodes	27.77	2.17	17.45	2.63	22.96	2.77	41.34	0.44	13.63	2.22	17.53	2.75	11.28	2.33	16.83	2.50	21.10	2.23
	@4	**INIT**	0.10	0.10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.01
		ASR	76.85	1.01	72.70	3.70	47.60	3.74	84.60	0.10	88.40	0.40	41.50	0.10	78.63	0.30	82.67	0.10	71.62	1.18
		M-Instrs	241.14	18.10	217.51	12.81	183.15	23.81	226.82	44.0	246.59	26.0	142.61	28.0	207.54	21.33	520.03	35.0	248.17	26.13
		M-Nodes	28.48	4.0	18.31	2.97	21.90	2.76	41.31	4.0	13.61	3.50	18.12	6.0	11.35	3.33	16.78	6.0	21.23	4.07

Table 8: Untargeted attack at $K = 10$ and $K = 100$ when considering a pool of size 128 with $\lambda \in \{0, 0.3\}$. In column AVG we report the average of the measures across all models.

			Models																	
			Gemini		GMN		ZEEK		BinFinder		SAFE		jTrans		Trex		PalmTree		AVG	
		λ	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**
K=10		**Recall pre attack**	0.71	0.71	0.94	0.94	0.73	0.73	0.91	0.91	0.94	0.94	0.81	0.81	0.95	0.95	0.90	0.90	0.86	0.86
		Recall post attack	0.07	0.65	0.03	0.80	0.05	0.61	0.06	0.9	0.02	0.86	0.04	0.75	0.04	0.92	0.03	0.86	0.04	0.79
		wASR	92.99	35.05	97.30	20.47	95.23	39.30	93.80	10.27	98.30	13.51	96.0	25.20	96.17	8.30	96.69	13.80	95.81	20.74
	@1	**INIT**	63.63	63.63	16.10	16.10	61.76	61.76	24.40	24.40	19.40	19.40	51.10	51.10	12.82	12.82	27.66	27.66	34.61	34.61
		ASR	96.69	67.50	99.30	35.10	98.06	68.52	95.80	25.80	99.20	31.19	99.40	55.80	97.61	19.40	98.0	33.97	98.01	42.16
		M-Instrs	235.07	8.66	214.44	12.17	201.77	9.47	226.76	5.66	237.61	12.09	133.19	9.99	195.51	12.36	524.32	10.56	246.08	10.12
		M-Nodes	26.98	1.62	16.90	2.26	25.68	1.79	41.73	0.26	13.68	1.95	16.74	1.81	10.99	1.83	16.69	2.01	21.17	1.69
	@2	**INIT**	38.48	38.48	5.90	5.90	35.73	35.73	9.90	9.90	4.90	4.90	24.20	24.20	4.97	4.97	10.62	10.62	16.84	16.84
		ASR	95.39	46.70	98.80	23.50	97.26	49.70	94.70	11.50	98.80	14.29	99.20	32.70	97.22	9.60	97.90	17.23	97.41	25.65
		M-Instrs	236.42	9.52	214.09	12.03	201.71	11.33	226.78	6.51	238.13	15.94	133.35	12.09	195.75	14.70	524.57	11.58	246.35	11.71
		M-Nodes	27.12	1.66	16.87	2.43	25.61	1.98	41.67	0.33	13.69	2.01	16.75	1.98	11.01	1.89	16.67	2.17	21.17	1.81
	@3	**INIT**	13.23	13.23	0.80	0.80	9.02	9.02	3.0	3.0	1.50	1.50	1.40	1.40	0.89	0.89	1.20	1.20	3.88	3.88
		ASR	91.58	20.60	97.0	13.60	94.52	25.71	93.10	3.50	97.80	6.04	94.80	10.70	95.83	3.0	95.89	3.51	95.06	10.83
		M-Instrs	239.26	10.09	212.59	13.16	202.82	16.09	226.88	10.20	239.12	22.12	135.64	15.87	197.18	21.27	523.93	17.77	247.18	15.82
		M-Nodes	27.28	2.03	16.93	2.79	25.26	2.40	41.59	0.51	13.69	2.45	16.96	2.37	11.05	1.93	16.71	2.86	21.18	2.17
	@4	**INIT**	1.20	1.20	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.15	0.15
		ASR	88.28	5.40	94.10	9.70	91.10	13.26	91.60	0.30	97.40	2.52	90.60	1.60	94.04	1.20	94.99	0.50	92.76	4.31
		M-Instrs	241.56	14.85	212.66	12.38	202.76	22.94	226.61	25.67	239.68	29.76	137.8	16.19	199.09	27.33	525.71	28.80	248.23	22.24
		M-Nodes	27.73	2.96	17.18	3.05	25.08	2.96	41.52	2.67	13.70	2.48	17.22	3.0	11.17	2.0	16.76	6.0	21.30	3.14
K=100		**Recall pre attack**	0.99	0.99	1.0	1.0	0.99	0.99	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
		Recall post attack	0.53	0.99	0.38	0.97	0.73	0.98	0.32	1.0	0.29	1.0	0.75	0.99	0.55	1.0	0.46	1.0	0.50	0.99
		wASR	47.12	1.18	62.25	3.10	26.86	1.67	67.58	0.25	70.60	0.18	24.65	0.85	45.23	0.35	54.16	0.20	49.81	0.97
	@1	**INIT**	2.81	2.81	1.10	1.10	1.71	1.71	0.70	0.70	0.10	0.10	0.80	0.80	0.60	0.60	0.20	0.20	1.0	1.0
		ASR	64.13	3.30	90.20	7.80	46.58	4.76	74.30	0.60	83.80	0.70	53.30	2.60	56.36	0.90	70.64	0.70	67.41	2.67
		M-Instrs	239.92	10.30	198.86	10.38	184.32	19.09	226.40	3.67	251.10	34.57	138.68	19.96	211.08	13.44	521.46	22.0	246.48	16.68
		M-Nodes	28.36	2.36	16.48	2.72	22.73	2.26	41.32	0.0	13.57	1.71	18.17	2.46	11.43	2.44	16.93	2.0	21.12	1.99
	@2	**INIT**	1.10	1.10	0.30	0.30	0.57	0.57	0.30	0.30	0.0	0.0	0.0	0.0	0.10	0.10	0.10	0.10	0.31	0.31
		ASR	53.21	1.10	73.0	3.60	33.33	1.52	69.20	0.30	77.10	0.0	30.60	0.80	48.91	0.20	59.42	0.10	55.60	0.95
		M-Instrs	236.12	11.91	207.11	9.08	178.44	25.20	226.56	0.0	254.92	-	131.56	19.25	216.62	26.0	520.98	18.0	246.54	15.63
		M-Nodes	28.85	2.36	17.22	2.72	21.65	2.07	41.12	0.0	13.36	-	18.16	3.25	11.39	3.0	16.99	2.0	21.09	2.20
	@3	**INIT**	0.0	0.0	0.0	0.0	0.0	0.0	0.10	0.10	0.0	0.0	0.0	0.0	0.10	0.10	0.0	0.0	0.02	0.02
		ASR	39.98	0.20	50.30	0.90	18.49	0.20	64.80	0.10	66.0	0.0	8.70	0.0	41.65	0.20	46.99	0.0	42.11	0.20
		M-Instrs	239.35	14.50	220.92	10.0	169.91	24.0	226.44	0.0	265.67	-	121.17	-	220.74	26.0	519.55	-	247.97	14.90
		M-Nodes	30.24	5.0	18.40	3.33	20.27	0.0	40.96	0.0	13.22	-	18.51	-	11.54	3.0	17.52	-	21.33	2.27
	@4	**INIT**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		ASR	31.16	0.10	35.5	0.10	9.02	0.20	62.0	0.0	55.50	0.0	6.0	0.0	34.0	0.10	39.58	0.0	34.10	0.06
		M-Instrs	238.51	25.0	226.95	6.0	165.33	24.0	225.67	-	267.78	-	121.87	-	224.18	35.0	517.75	-	248.5	22.50
		M-Nodes	31.06	8.0	19.05	4.0	19.95	0.0	40.84	-	13.41	-	18.40	-	11.75	6.0	18.49	-	21.62	4.50

Table 9: Untargeted attack at $K = 10$ adn $K = 100$ when considering a pool of size 512 with $\lambda \in \{0, 0.3\}$. In column AVG we report the average of the measures across all models.

			Models																	
			Gemini		GMN		ZEEK		BinFinder		SAFE		jTrans		Trex		PalmTree		AVG	
		λ	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**
K=10		**Recall pre attack**	0.56	0.56	0.86	0.86	0.51	0.51	0.83	0.83	0.84	0.84	0.67	0.67	0.89	0.89	0.79	0.79	0.74	0.74
		Recall post attack	0.04	0.49	0.01	0.66	0.02	0.39	0.04	0.82	0.0	0.73	0.01	0.59	0.02	0.83	0.01	0.72	0.02	0.65
		wASR	96.27	51.45	99.0	33.65	98.12	60.78	96.33	18.0	99.58	26.86	99.15	41.38	98.43	17.05	99.37	28.33	98.28	34.69
	@1	**INIT**	82.87	82.87	36.40	36.40	84.70	84.70	39.40	39.40	42.60	42.60	73.20	73.20	29.22	29.22	53.15	53.15	55.19	55.19
		ASR	98.10	85.30	99.70	57.30	99.09	88.77	97.80	40.20	100.0	54.23	100.0	77.60	99.30	37.50	99.90	61.72	99.24	62.83
		M-Instrs	234.18	8.20	215.0	10.44	202.11	8.15	227.07	4.94	236.85	10.03	133.06	8.75	193.39	11.52	522.24	9.54	245.49	8.95
		M-Nodes	26.95	1.57	16.94	2.05	25.83	1.67	41.80	0.28	13.66	1.78	16.73	1.73	10.88	1.79	16.80	1.84	21.20	1.59
	@2	**INIT**	64.33	64.33	16.10	16.10	72.26	72.26	21.20	21.20	17.40	17.40	48.40	48.40	12.82	12.82	28.63	28.63	35.14	35.14
		ASR	97.49	70.80	99.30	39.30	98.86	77.13	96.90	22.70	99.90	33.40	100.0	54.70	98.81	20.80	99.40	38.88	98.83	44.71
		M-Instrs	234.41	8.67	214.44	11.80	202.06	8.89	226.96	5.42	236.93	12.24	133.06	10.14	194.03	13.92	522.88	10.58	245.6	10.21
		M-Nodes	27.0	1.60	16.92	2.27	25.81	1.73	41.79	0.35	13.66	1.92	16.73	1.85	10.91	1.87	16.79	1.95	21.20	1.69
	@3	**INIT**	27.15	27.15	3.40	3.40	34.02	34.02	6.80	6.80	3.50	3.50	12.0	12.0	2.49	2.49	3.90	3.90	11.66	11.66
		ASR	95.39	37.70	98.80	22.90	97.60	50.81	95.9	8.10	99.30	13.68	99.10	26.0	98.11	7.50	99.20	10.32	97.92	22.13
		M-Instrs	237.11	10.29	213.98	13.19	202.18	11.39	226.92	7.46	237.66	18.51	133.94	13.10	195.07	19.81	523.35	15.97	246.28	13.72
		M-Nodes	27.11	1.90	16.88	2.60	25.62	2.0	41.72	0.44	13.68	2.43	16.84	2.02	10.97	2.13	16.75	2.78	21.20	2.04
	@4	**INIT**	3.01	3.01	0.0	0.0	3.08	3.08	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.76	0.76
		ASR	94.09	12.0	98.20	15.10	96.92	26.42	94.70	1.0	99.10	6.14	97.50	7.20	97.51	2.40	99.0	2.40	97.13	9.08
		M-Instrs	239.12	14.64	214.32	14.09	202.45	17.18	226.92	21.70	237.94	25.23	135.35	18.25	195.80	25.83	523.85	27.25	246.97	20.52
		M-Nodes	27.28	2.78	16.92	2.97	25.52	2.57	41.66	2.20	13.68	2.79	17.0	2.69	11.01	2.42	16.75	4.50	21.23	2.86
K=100		**Recall pre attack**	0.84	0.84	0.98	0.98	0.91	0.91	0.96	0.96	0.98	0.98	0.90	0.90	0.99	0.99	0.97	0.97	0.94	0.94
		Recall post attack	0.12	0.81	0.07	0.88	0.19	0.79	0.10	0.96	0.04	0.95	0.10	0.84	0.10	0.97	0.07	0.96	0.10	0.90
		wASR	88.13	19.10	93.35	12.20	81.42	20.55	90.28	4.10	96.45	5.18	90.0	16.02	89.84	2.83	93.17	4.08	90.33	10.51
	@1	**INIT**	37.37	37.37	6.40	6.40	25.46	25.46	10.40	10.40	6.90	6.90	30.80	30.80	3.68	3.68	8.11	8.11	16.14	16.14
		ASR	91.78	42.10	98.10	21.20	89.73	39.78	92.70	10.70	97.90	12.07	98.60	39.20	93.64	7.0	95.60	11.32	94.76	22.92
		M-Instrs	237.26	9.74	211.20	12.98	200.39	12.56	226.72	5.96	238.95	15.91	133.75	11.77	198.47	14.0	523.0	12.5	246.22	11.93
		M-Nodes	27.18	1.58	16.86	2.46	24.98	2.02	41.59	0.19	13.70	2.12	16.74	1.97	11.17	1.79	16.73	2.25	21.12	1.80
	@2	**INIT**	18.94	18.94	2.10	2.10	9.82	9.82	4.30	4.30	1.60	1.60	10.50	10.50	1.19	1.19	3.20	3.20	6.46	6.46
		ASR	90.08	23.80	96.90	13.0	86.07	23.89	91.0	4.80	97.40	5.53	95.70	20.50	91.85	2.70	94.69	4.11	92.96	12.29
		M-Instrs	237.21	10.01	210.50	12.93	198.92	16.28	226.69	5.71	239.4	20.62	134.70	13.61	199.68	18.11	523.10	13.20	246.28	13.81
		M-Nodes	27.36	1.66	16.94	2.63	24.70	2.34	41.51	0.25	13.71	2.38	16.81	2.21	11.22	1.48	16.78	2.37	21.13	1.92
	@3	**INIT**	6.11	6.11	0.40	0.40	2.28	2.28	0.60	0.60	0.30	0.30	0.0	0.0	0.30	0.30	0.60	0.60	1.32	1.32
		ASR	86.97	8.50	93.10	8.40	77.97	12.25	89.20	0.80	96.0	2.11	87.0	3.90	88.37	1.30	91.99	0.80	88.82	4.76
		M-Instrs	240.62	10.69	210.86	12.60	198.26	21.06	226.76	14.75	240.66	28.62	137.75	17.59	201.81	22.08	521.96	16.0	247.33	17.92
		M-Nodes	27.52	2.12	17.16	2.79	23.81	2.77	41.42	0.50	13.64	2.38	17.10	2.72	11.18	1.85	16.95	2.88	21.10	2.25
	@4	**INIT**	0.30	0.30	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.04	0.04
		ASR	83.67	2.0	85.30	6.20	71.92	6.28	88.20	0.10	94.50	1.01	78.70	0.50	85.49	0.30	90.39	0.10	84.77	2.06
		M-instrs@4	242.59	19.4	213.31	10.74	196.17	25.34	226.72	44.0	241.91	28.40	140.61	17.80	204.38	21.33	521.90	35.0	248.45	25.25
		M-Nodes	27.93	3.30	17.69	2.81	23.39	2.94	41.37	4.0	13.62	2.20	17.62	4.0	11.31	3.33	16.98	6.0	21.24	3.57

Table 10: Untargeted attack at $K = 10$ and $K = 100$ when considering a pool of size 1000 with $\lambda \in \{0, 0.3\}$. In column AVG we report the average of the measures across all models.

			Models																	
			Gemini		GMN		ZEEK		BinFinder		SAFE		jTrans		Trex		PalmTree		AVG	
		λ	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**
K=10		**Recall pre attack**	0.49	0.49	0.81	0.81	0.0	0.0	0.79	0.79	0.78	0.78	0.60	0.60	0.85	0.85	0.72	0.72	0.63	0.63
		Recall post attack	0.03	0.42	0.01	0.59	0.0	0.0	0.03	0.78	0.0	0.65	0.01	0.52	0.01	0.78	0.0	0.65	0.01	0.55
		wASR	97.39	58.28	99.45	41.33	100.0	100.0	97.27	22.27	99.87	34.53	99.50	47.57	98.78	22.39	99.60	34.89	98.98	45.16
	@1	**INIT**	88.76	88.76	47.42	47.42	100.0	100.0	46.29	46.29	55.41	55.41	80.66	80.66	37.82	37.82	64.76	64.76	65.14	65.14
		ASR	98.80	90.30	99.80	67.30	100.0	100.0	98.60	47.49	100.0	67.04	100.0	83.37	99.40	47.90	100.0	71.21	99.57	71.83
		M-Instrs	233.71	7.86	216.29	9.98	201.32	7.69	227.07	4.63	236.96	9.26	133.02	8.27	192.94	10.87	522.46	9.21	245.47	8.47
		M-Nodes	27.06	1.52	16.94	1.98	25.80	1.63	41.84	0.27	13.65	1.75	16.73	1.69	10.84	1.74	16.75	1.75	21.20	1.54
	@2	**INIT**	75.80	75.80	23.61	23.61	100.0	100.0	27.15	27.15	28.66	28.66	58.02	58.02	18.36	18.36	38.35	38.35	46.24	46.24
		ASR	98.29	80.17	99.60	50.60	100.0	100.0	97.70	28.66	100.0	44.35	100.0	64.83	99.30	26.55	99.70	51.18	99.32	55.79
		M-Instrs	234.06	8.18	215.61	10.89	201.32	7.69	226.92	5.34	236.96	11.05	133.02	9.17	193.11	12.86	523.28	10.0	245.54	9.40
		M-Nodes	26.98	1.53	16.97	2.14	25.80	1.63	41.80	0.37	13.65	1.87	16.73	1.79	10.86	1.83	16.74	1.81	21.19	1.62
	@3	**INIT**	34.84	34.84	5.85	5.85	100.0	100.0	9.92	9.92	5.41	5.41	19.44	19.44	4.19	4.19	7.23	7.23	23.36	23.36
		ASR	96.89	46.62	99.40	29.70	100.0	100.0	96.89	11.52	99.90	19.15	99.50	32.16	98.50	11.02	99.50	14.69	98.82	33.11
		M-Instrs	235.75	9.53	215.08	12.58	201.32	7.69	226.99	7.33	237.05	17.25	133.54	12.04	194.47	17.99	523.11	14.75	245.91	12.40
		M-Nodes	27.07	1.79	16.97	2.46	25.80	1.63	41.76	0.40	13.65	2.26	16.79	2.04	10.94	2.07	16.74	2.31	21.22	1.87
	@4	**INIT**	3.71	3.71	0.0	0.0	100.0	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	12.96	12.96
		ASR	95.58	16.03	99.01	17.70	100.0	100.0	95.89	1.40	99.60	7.56	98.50	9.92	97.90	4.11	99.20	2.49	98.21	19.90
		M-Instrs	237.11	13.57	214.63	13.43	201.32	7.69	227.03	24.29	237.46	25.08	134.51	17.09	195.15	26.41	523.77	26.67	246.37	19.28
		M-Nodes	27.13	2.58	17.0	2.87	25.80	1.63	41.71	2.0	13.65	2.51	16.94	2.65	10.97	2.44	16.70	4.10	21.24	2.60
K=100		**Recall pre attack**	0.77	0.77	0.96	0.96	0.77	0.77	0.93	0.93	0.96	0.96	0.84	0.84	0.97	0.97	0.94	0.94	0.89	0.89
		Recall post attack	0.09	0.72	0.03	0.83	0.09	0.63	0.07	0.93	0.02	0.90	0.04	0.77	0.06	0.95	0.03	0.91	0.05	0.83
		wASR	91.19	28.27	96.55	17.12	91.50	37.15	93.11	7.16	97.87	9.73	96.14	23.05	94.39	5.41	96.66	8.86	94.68	17.09
	@1	**INIT**	51.31	51.31	11.21	11.21	54.91	54.91	17.54	17.54	12.12	12.12	42.48	42.48	6.99	6.99	17.07	17.07	26.70	26.70
		ASR	94.48	56.54	98.81	29.10	95.21	63.66	94.79	18.24	98.60	21.27	99.60	49.70	96.41	12.12	97.79	22.51	96.96	34.14
		M-Instrs	235.98	8.73	214.91	12.24	200.80	9.82	226.65	5.26	238.58	14.26	133.20	10.47	196.27	13.38	524.59	11.14	246.37	10.66
		M-Nodes	27.09	1.56	16.84	2.32	25.41	1.80	41.68	0.29	13.67	2.02	16.76	1.88	11.02	1.79	16.64	1.96	21.14	1.70
	@2	**INIT**	29.92	29.92	3.57	3.57	29.57	29.57	7.21	7.21	3.11	3.11	19.04	19.04	2.69	2.69	6.73	6.73	12.73	12.73
		ASR	93.47	36.29	98.12	18.90	94.29	45.45	93.89	8.02	98.20	10.38	99.20	29.86	95.61	6.11	97.59	11.02	96.30	20.75
		M-Instrs	236.69	9.41	213.87	12.81	200.03	11.97	226.66	5.62	238.76	18.61	133.54	12.58	196.78	14.56	524.98	13.55	246.41	12.39
		M-Nodes	27.22	1.58	16.87	2.54	25.35	2.0	41.63	0.25	13.69	2.30	16.81	2.04	11.05	1.82	16.64	2.14	21.16	1.83
	@3	**INIT**	10.34	10.34	0.60	0.60	7.88	7.88	1.60	1.60	0.70	0.70	1.30	1.30	0.60	0.60	0.90	0.90	2.99	2.99
		ASR	89.96	15.72	95.93	11.90	90.07	25.91	92.59	2.20	97.70	4.94	94.89	10.82	93.61	2.51	95.88	1.54	93.83	9.44
		M-Instrs	240.27	10.20	212.58	12.97	199.66	16.57	226.71	13.32	239.13	24.49	136.12	15.07	198.60	22.40	524.66	19.92	247.22	16.87
		M-Nodes	27.33	2.0	17.02	2.84	24.99	2.41	41.55	0.55	13.69	2.59	16.97	2.46	11.12	1.92	16.69	2.54	21.17	2.16
	@4	**INIT**	1.10	1.10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.14	0.14
		ASR	86.85	4.54	93.35	8.60	86.42	13.56	91.18	0.20	96.99	2.32	90.88	1.80	91.92	0.90	95.38	0.36	91.62	4.04
		M-Instrs	243.81	13.49	213.29	11.51	200.17	22.78	226.59	29.0	240.05	31.17	137.90	15.17	199.77	24.67	525.34	31.67	248.36	22.43
		M-Nodes	27.79	2.56	17.25	3.05	24.66	2.99	41.46	4.0	13.70	2.26	17.15	2.89	11.19	1.78	16.72	5.33	21.24	3.11

Table 11: Untargeted attack at $K = 10$ and $K = 100$ when considering $\lambda = 0.01$ and $|P| \in \{32, 128, 512, 1000\}$. In column AVG we report the average of the measures across all models.

			Models															
			Gemini				SAFE				jTrans				AVG			
		Pool Size	32	128	512	1000	32	128	512	1000	32	128	512	1000	32	128	512	1000
K=10		**Recall pre attack**	0.90	0.71	0.55	0.48	0.98	0.93	0.84	0.78	0.93	0.81	0.67	0.61	0.94	0.82	0.69	0.62
		Recall post attack	0.46	0.23	0.14	0.11	0.26	0.07	0.02	0.01	0.66	0.34	0.17	0.12	0.46	0.21	0.11	0.08
		wASR	54.47	76.58	86.24	89.22	74.0	92.74	98.35	98.98	34.16	65.84	82.61	87.50	54.21	78.39	89.07	91.90
	@1	**INIT**	27.51	63.15	83.94	89.24	4.91	19.54	42.79	53.86	21.69	51.31	73.19	80.30	18.04	44.67	66.64	74.47
		ASR	72.79	90.46	96.08	97.23	87.07	96.79	99.40	99.76	66.16	91.06	97.29	98.52	75.34	92.77	97.59	98.50
		M-Instrs	26.32	24.03	23.28	23.04	46.29	44.81	44.31	44.16	18.93	17.71	17.41	17.38	30.51	28.85	28.33	28.19
		M-Nodes	7.71	7.05	6.80	6.71	5.85	5.87	5.84	5.80	3.81	3.70	3.65	3.68	5.79	5.54	5.43	5.40
	@2	**INIT**	10.24	38.76	65.56	76.36	1.10	5.51	17.74	26.63	4.42	24.40	48.39	57.20	5.25	22.89	43.90	53.40
		ASR	61.75	83.53	93.17	94.46	80.06	95.89	99.10	99.40	45.78	81.83	93.37	95.76	62.53	87.08	95.21	96.54
		M-Instrs	27.94	25.12	23.74	23.49	47.13	44.92	44.39	44.29	19.93	18.28	17.68	17.52	31.67	29.44	28.60	28.43
		M-Nodes	8.04	7.29	6.95	6.84	5.83	5.87	5.84	5.81	3.91	3.80	3.72	3.73	5.93	5.65	5.50	5.46
	@3	**INIT**	2.41	12.75	27.31	36.21	0.20	1.40	3.41	6.14	0.0	1.20	11.95	18.54	0.87	5.12	14.22	20.30
		ASR	47.49	71.59	83.03	88.18	69.14	90.98	98.20	98.67	16.77	55.12	78.41	84.85	44.47	72.56	86.55	90.57
		M-Instrs	30.37	27.31	25.48	24.62	48.72	45.75	44.57	44.43	22.75	19.74	18.63	18.28	33.95	30.93	29.56	29.11
		M-Nodes	8.80	7.90	7.41	7.17	5.74	5.89	5.87	5.83	4.44	4.10	3.92	3.90	6.33	5.96	5.73	5.63
	@4	**INIT**	0.10	1.41	2.91	4.90	0.10	0.20	0.30	0.36	0.0	0.0	0.0	0.0	0.07	0.54	1.07	1.75
		ASR	35.84	60.74	72.69	77.0	59.72	87.27	96.69	98.07	7.93	35.34	61.35	70.87	34.50	61.12	76.91	81.98
		M-Instrs	32.79	29.42	27.45	26.52	49.98	46.28	44.80	44.56	24.14	21.08	19.64	19.22	35.64	32.26	30.63	30.10
		M-Nodes	9.65	8.55	7.93	7.66	5.67	5.90	5.89	5.85	4.68	4.34	4.13	4.07	6.67	6.26	5.98	5.86
K=100		**Recall pre attack**	-	0.99	0.85	0.77	-	1.0	0.98	0.96	-	1.0	0.90	0.85	-	1.0	0.91	0.86
		Recall post attack	-	0.79	0.38	0.30	-	0.70	0.16	0.09	-	0.94	0.50	0.36	-	0.81	0.35	0.25
		wASR	-	21.01	61.97	70.45	-	29.86	83.84	91.08	-	5.65	49.8	63.96	-	18.84	65.20	75.16
	@1	**INIT**	-	2.61	35.64	50.80	-	0.20	6.81	12.41	-	0.70	30.62	41.63	-	1.17	24.36	34.95
		ASR	-	33.84	76.20	83.92	-	47.70	91.08	96.02	-	14.16	78.31	87.82	-	31.90	81.86	89.25
		M-Instrs	-	31.84	25.88	24.94	-	50.29	45.69	44.93	-	22.77	18.35	17.94	-	34.97	29.97	29.27
		M-Nodes	-	8.82	7.56	7.24	-	5.71	5.87	5.85	-	4.26	3.77	3.77	-	6.26	5.73	5.62
	@2	**INIT**	-	1.0	17.97	30.03	-	0.10	1.90	3.49	-	0.0	10.54	18.11	-	0.37	10.14	17.21
		ASR	-	23.09	67.07	76.04	-	34.57	87.68	93.98	-	6.43	63.35	78.28	-	21.36	72.70	82.77
		M-Instrs	-	33.91	27.43	26.16	-	51.61	46.05	45.29	-	24.92	19.24	18.57	-	36.81	30.91	30.01
		M-Nodes	-	9.35	7.88	7.50	-	5.57	5.89	5.89	-	4.41	3.82	3.84	-	6.44	5.86	5.74
	@3	**INIT**	-	0.20	5.32	10.44	-	0.0	0.50	1.20	-	0.0	0.0	1.17	-	0.07	1.94	4.27
		ASR	-	16.16	56.83	65.81	-	22.24	81.46	89.04	-	1.31	36.75	53.6	-	13.24	58.35	69.48
		M-Instrs	-	36.48	29.18	28.07	-	52.67	47.01	46.13	-	24.69	21.31	19.90	-	37.95	32.50	31.37
		M-Nodes	-	10.05	8.35	8.05	-	5.41	5.84	5.87	-	4.77	4.26	4.15	-	6.74	6.15	6.02
	@4	**INIT**	-	0.0	0.20	1.17	-	0.0	0.10	0.24	-	0.0	0.0	0.0	-	0.0	0.10	0.47
		ASR	-	10.94	47.79	56.02	-	14.93	75.15	85.30	-	0.70	20.78	36.12	-	8.86	47.91	59.15
		M-Instrs	-	38.30	31.13	30.01	-	53.18	47.83	46.68	-	21.0	23.01	21.0	-	37.49	33.99	32.56
		M-Nodes	-	10.35	8.79	8.63		5.28	5.83	5.88	-	4.86	4.50	4.34	-	6.83	6.37	6.28

Table 12: Targeted attack at $K = 5$ and $K = 10$ when considering a pool of size 32 with $\lambda \in \{0, 0.3\}$. In column AVG we report the average of the measures across all models.

			Models																	
			Gemini		GMN		ZEEK		BinFinder		SAFE		jTrans		Trex		PalmTree		AVG	
		λ	0.0	0.3	0.0	0.3	0.0	0.3	0.0	0.3	0.0	0.3	0.0	0.3	0.0	0.3	0.0	0.3	0.0	0.3
K=5		**wASR**	58.05	14.44	68.10	15.20	38.14	13.36	92.11	14.58	74.17	17.72	56.05	22.65	62.78	15.24	73.25	15.37	65.33	16.07
	@1	**INIT**	30.10	30.10	31.0	31.0	29.29	29.29	25.25	25.25	27.10	27.10	43.60	43.60	22.30	22.30	29.16	29.16	29.72	29.72
		ASR	85.70	32.70	85.70	33.60	59.69	29.66	97.8	26.15	90.40	34.74	94.0	54.80	80.70	28.31	90.38	31.59	85.55	33.94
		M-Nodes	13.64	0.92	18.68	0.96	13.72	1.05	33.50	0.07	15.57	0.98	11.50	1.04	11.40	0.90	18.01	0.85	17.0	0.85
	@2	**INIT**	16.80	16.80	15.0	15.0	12.65	12.65	16.23	16.23	17.10	17.10	20.0	20.0	14.60	14.60	16.33	16.33	16.09	16.09
		ASR	69.8	18.61	77.60	17.50	46.22	15.55	96.09	17.13	83.60	21.39	76.20	28.20	70.50	18.78	81.16	18.51	75.15	19.46
		M-Instrs	177.28	4.75	210.62	5.55	104.62	5.17	156.23	2.18	123.69	4.95	79.35	5.49	111.24	5.76	193.30	4.43	144.54	4.78
		M-Nodes	14.01	0.90	18.66	0.90	12.41	0.97	33.55	0.07	15.69	0.97	12.02	1.06	11.38	0.82	18.20	0.75	16.99	0.80
	@3	**INIT**	4.50	4.50	5.30	5.30	3.16	3.16	8.92	8.92	6.90	6.90	3.30	3.30	6.90	6.90	6.61	6.61	5.70	5.70
		ASR	47.60	4.93	63.10	6.90	29.69	6.08	90.48	9.62	69.90	10.34	35.9	6.0	56.80	9.64	68.74	7.85	57.78	7.67
		M-Instrs	177.31	3.04	211.28	5.71	116.74	4.36	157.25	2.31	126.55	5.21	92.23	4.47	112.01	5.36	194.07	4.76	148.43	4.40
		M-Nodes	14.51	0.69	18.47	0.9	12.24	0.85	33.74	0.08	15.8	1.05	13.45	1.03	11.29	0.83	18.5	0.67	17.25	0.76
	@4	**INIT**	1.21	1.21	2.30	2.30	1.22	1.22	5.51	5.51	2.70	2.70	0.70	0.70	3.70	3.70	3.11	3.11	2.56	2.56
		ASR	29.10	1.51	46.0	2.80	16.94	2.16	84.07	5.41	52.80	4.42	18.10	1.60	43.10	4.22	52.71	3.52	42.85	3.20
		M-Instrs	189.28	5.93	217.58	5.68	127.83	5.05	158.79	2.24	133.9	4.93	101.06	6.19	112.81	5.07	195.16	4.43	154.55	4.94
		M-Nodes	16.04	0.8	18.81	0.86	12.01	1.24	34.20	0.04	15.89	1.05	14.13	0.88	11.16	0.81	18.99	0.86	17.65	0.82
K=10		**wASR**	80.95	30.96	87.80	35.02	59.80	28.55	98.35	30.51	89.60	37.12	75.6	44.17	84.05	34.66	90.76	34.26	83.36	34.41
	@1	**INIT**	49.30	49.30	55.20	55.20	51.73	51.73	44.09	44.09	49.0	49.0	70.80	70.80	45.60	45.60	50.20	50.20	51.99	51.99
		ASR	95.50	53.52	95.0	59.90	76.43	50.77	99.60	44.89	95.50	56.02	98.90	81.60	93.40	51.31	96.49	54.93	93.85	56.62
		M-Instrs	187.36	4.44	214.27	5.98	105.82	4.71	156.66	1.63	121.52	4.92	74.16	5.07	109.49	5.88	198.37	4.16	145.96	4.60
		M-Nodes	13.78	0.86	18.90	1.01	16.52	0.92	33.64	0.08	15.48	0.96	11.38	1.08	11.31	0.97	18.03	0.87	17.38	0.84
	@2	**INIT**	34.70	34.70	37.70	37.70	36.53	36.53	34.77	34.77	36.40	36.40	49.60	49.60	34.0	34.0	36.07	36.07	37.47	37.47
		ASR	88.40	38.33	91.80	42.60	66.02	34.50	99.10	36.57	92.60	42.77	95.40	63.0	88.10	41.06	93.59	41.25	89.38	42.51
		M-Instrs	183.01	4.40	213.54	5.94	104.13	4.90	156.57	1.75	121.92	5.02	75.15	5.25	109.68	5.79	198.19	4.19	145.27	4.66
		M-Nodes	13.81	0.86	18.78	1.03	14.98	0.90	33.64	0.06	15.56	0.95	11.45	1.07	11.38	0.93	17.98	0.82	17.20	0.83
	@3	**INIT**	18.11	18.11	20.0	20.0	18.57	18.57	22.85	22.85	23.20	23.20	15.80	15.80	21.0	21.0	20.54	20.54	20.10	20.10
		ASR	77.10	20.72	85.90	23.80	54.29	18.85	98.30	23.55	87.80	29.52	63.60	23.40	81.10	27.21	89.08	24.75	79.65	23.98
		M-Instrs	185.11	4.70	212.0	5.80	103.81	5.57	156.53	1.68	122.60	4.75	85.51	5.09	110.79	5.99	196.56	4.47	146.61	4.76
		M-Nodes	14.17	0.87	18.88	1.0	13.45	1.03	33.62	0.06	15.54	0.94	12.71	1.10	11.51	0.85	18.12	0.82	17.25	0.83
	@4	**INIT**	9.80	9.80	10.20	10.20	10.10	10.10	16.43	16.43	14.20	14.20	4.30	4.30	15.0	15.0	12.42	12.42	11.56	11.56
		ASR	62.80	11.27	78.50	13.80	42.45	10.09	96.39	17.03	82.5	20.18	44.50	8.70	73.60	19.08	83.87	16.10	70.58	14.53
		M-Instrs	188.77	4.06	217.14	6.14	108.34	4.98	157.09	1.64	124.88	5.18	94.42	5.69	111.24	5.53	197.1	4.12	149.87	4.67
		M-Nodes	14.99	0.77	19.10	0.96	12.62	0.89	33.72	0.06	15.75	0.98	13.92	1.13	11.60	0.74	18.30	0.70	17.50	0.78

Table 13: Targeted attack at $K = 5$ and $K = 10$ when considering a pool of size 128 with $\lambda \in \{0, 0.3\}$. In column AVG we report the average of the measures across all models.

			Models																			
			Gemini		GMN		ZEEK		BinFinder		SAFE		jTrans		Trex		PalmTree		AVG	
		λ	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**
K=5		**wASR**	32.17	4.84	46.30	4.03	13.98	4.35	78.18	3.63	48.20	4.37	30.80	5.88	36.40	3.74	43.29	3.50	41.16	4.29
	@1	**INIT**	11.50	11.50	8.10	8.10	7.96	7.96	6.81	6.81	8.20	8.20	11.30	11.30	5.90	5.90	7.49	7.49	8.41	8.41
		ASR	57.60	12.50	68.70	9.94	29.80	11.74	89.28	7.52	69.50	10.94	65.30	17.40	54.10	8.13	65.17	8.95	62.43	10.89
		M-Instrs	168.08	4.71	210.41	7.04	90.81	5.98	154.83	2.80	127.34	4.79	80.82	5.80	114.17	5.07	191.74	5.34	142.28	5.19
		M-Nodes	14.21	0.85	18.40	0.98	9.98	1.11	33.08	0.08	15.76	0.95	12.27	1.01	11.42	0.86	18.03	0.94	16.64	0.85
	@2	**INIT**	5.0	5.0	3.10	3.10	2.14	2.14	4.21	4.21	2.80	2.80	2.80	2.80	3.10	3.10	2.59	2.59	3.22	3.22
		ASR	39.40	5.54	55.90	4.37	17.76	4.53	82.97	4.41	56.70	4.92	37.60	5.20	43.0	4.62	50.40	4.12	47.97	4.71
		M-Instrs	173.39	3.29	213.08	6.02	91.01	6.34	156.15	3.75	129.9	4.65	89.29	5.92	115.26	6.70	188.92	5.76	144.62	5.30
		M-Nodes	15.14	0.65	18.33	0.82	10.26	1.18	33.40	0.14	15.77	0.98	13.23	0.88	11.23	0.91	17.91	1.02	16.91	0.82
	@3	**INIT**	0.90	0.90	0.90	0.90	0.20	0.20	1.70	1.70	0.80	0.80	0.40	0.40	1.20	1.20	0.20	0.20	0.79	0.79
		ASR	20.40	1.01	38.10	1.19	5.82	0.93	74.35	1.70	39.8	1.10	13.70	0.70	29.50	1.31	34.33	0.70	32.0	1.08
		M-Instrs	178.93	2.30	222.24	5.0	79.56	6.11	158.05	3.88	133.82	4.91	102.27	4.57	117.87	6.77	182.46	8.71	146.9	5.28
		M-Nodes	16.21	0.20	18.84	0.67	8.98	1.11	33.69	0.12	15.78	1.27	14.58	1.14	11.17	0.62	18.30	1.43	17.19	0.82
	@4	**INIT**	0.20	0.20	0.20	0.20	0.10	0.10	1.0	1.0	0.10	0.10	0.10	0.10	0.60	0.60	0.0	0.0	0.29	0.29
		ASR	11.30	0.30	22.50	0.60	2.55	0.21	66.13	0.90	26.80	0.50	6.60	0.20	19.0	0.90	23.25	0.20	22.27	0.48
		M-Instrs	193.28	1.67	232.65	7.0	94.08	4.0	159.70	3.0	142.31	2.40	114.97	9.50	120.6	5.89	187.06	15.50	155.58	6.12
		M-Nodes	17.50	0.0	19.25	1.0	10.88	2.0	34.01	0.0	15.54	1.20	15.88	2.0	10.87	0.44	18.70	1.0	17.83	0.96
K=10		**wASR**	48.25	8.47	64.68	8.33	26.07	8.52	88.95	7.97	64.80	9.16	45.25	12.20	54.85	8.43	63.60	8.68	57.06	8.97
	@1	**INIT**	17.20	17.20	15.80	15.80	15.0	15.0	12.12	12.12	13.90	13.90	23.90	23.90	12.10	12.10	14.97	14.97	15.62	15.62
		ASR	71.90	19.35	78.60	17.99	43.16	19.16	94.39	13.33	80.30	17.07	80.20	32.20	68.80	15.26	78.04	17.10	74.42	18.93
		M-Instrs	173.62	5.29	214.53	6.80	94.86	5.87	155.21	2.19	123.74	4.94	77.29	5.63	112.70	5.57	193.41	4.74	143.17	5.13
		M-Nodes	13.83	0.92	18.87	0.98	11.18	1.07	33.28	0.06	15.72	0.96	11.95	1.07	11.44	0.80	18.28	0.86	16.82	0.84
	@2	**INIT**	9.40	9.40	8.10	8.10	6.53	6.53	7.82	7.82	8.60	8.60	8.30	8.30	8.20	8.20	7.98	7.98	8.12	8.12
		ASR	57.0	10.58	71.10	9.15	31.94	10.20	91.68	9.22	71.40	11.14	58.90	13.50	60.40	10.24	70.06	9.96	64.06	10.50
		M-Instrs	173.68	4.25	213.81	5.61	96.81	5.84	155.94	2.75	126.32	4.86	84.68	5.16	112.83	4.88	192.08	4.77	144.52	4.76
		M-Nodes	14.68	0.74	18.63	0.96	10.35	1.12	33.40	0.09	15.89	0.95	12.65	1.04	11.30	0.73	18.09	0.75	16.87	0.80
	@3	**INIT**	2.70	2.70	3.30	3.30	1.84	1.84	4.91	4.91	3.10	3.10	1.30	1.30	4.10	4.10	2.89	2.89	3.02	3.02
		ASR	38.0	2.92	60.40	3.88	18.57	3.50	87.37	5.81	60.10	5.62	27.30	2.50	49.50	5.02	57.58	5.03	49.85	4.28
		M-Instrs	177.87	3.83	214.63	5.08	93.72	5.94	157.19	3.53	129.75	4.57	96.62	4.76	115.36	6.20	189.60	4.76	146.84	4.83
		M-Nodes	15.76	0.55	18.92	0.82	10.26	1.18	33.67	0.10	15.84	0.93	13.86	1.36	11.34	0.92	18.15	0.84	17.23	0.84
	@4	**INIT**	0.90	0.90	1.60	1.60	0.31	0.31	3.21	3.21	1.41	1.41	0.30	0.30	2.60	2.60	1.20	1.20	1.44	1.44
		ASR	26.10	1.01	48.60	2.29	10.61	1.24	82.36	3.51	47.40	2.81	14.60	0.60	40.70	3.21	48.70	2.62	39.88	2.16
		M-Instrs	190.36	8.0	219.14	5.83	104.95	6.92	159.24	2.11	136.29	3.89	109.49	6.67	116.21	5.19	192.21	5.19	153.49	5.48
		M-Nodes	16.57	0.80	19.48	0.70	10.99	1.33	34.19	0.06	16.05	0.71	14.71	1.67	11.42	0.88	18.37	1.0	17.72	0.89

Table 14: Targeted attack at $K = 5$ and $K = 10$ when considering a pool of size 512 with $\lambda \in \{0, 0.3\}$. In column AVG we report the average of the measures across all models.

			Models																			
			Gemini		GMN		ZEEK		BinFinder		SAFE		jTrans		Trex		PalmTree		AVG	
		λ	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**
K=5		**wASR**	16.93	1.16	26.88	1.03	5.15	1.11	58.27	0.98	27.95	0.93	15.82	1.17	20.52	0.98	18.09	1.01	23.7	1.05
	@1	**INIT**	3.80	3.80	1.90	1.90	2.24	2.24	1.60	1.60	1.90	1.90	2.40	2.40	1.70	1.70	1.80	1.80	2.17	2.17
		ASR	34.20	3.63	49.0	2.70	12.24	3.71	71.94	2.10	49.50	2.71	38.30	3.98	34.50	2.51	34.37	2.92	40.51	3.03
		M-Instrs	168.49	3.19	206.50	5.81	110.12	5.22	155.03	7.86	126.75	3.11	89.29	6.32	113.79	5.48	176.89	6.93	143.36	5.49
		M-Nodes	15.67	0.44	17.87	1.04	10.58	1.0	33.09	0.10	15.64	0.59	13.28	0.90	11.32	1.20	18.37	1.17	16.98	0.80
	@2	**INIT**	1.0	1.0	0.40	0.40	0.31	0.31	1.10	1.10	0.50	0.50	0.20	0.20	0.40	0.40	0.60	0.60	0.56	0.56
		ASR	19.70	0.81	33.0	1.0	6.02	0.72	65.03	1.20	33.80	0.70	18.0	0.70	24.30	0.90	21.34	1.01	27.65	0.88
		M-Instrs	182.49	4.12	212.32	5.50	123.47	5.0	156.63	3.0	132.94	1.14	101.05	5.14	115.57	7.67	173.31	7.80	149.72	4.92
		M-Nodes	16.52	0.25	17.77	1.0	11.12	0.57	33.57	0.0	15.54	0.57	14.43	1.14	11.03	1.33	18.49	1.40	17.31	0.78
	@3	**INIT**	0.10	0.10	0.10	0.10	0.0	0.0	0.40	0.40	0.10	0.10	0.0	0.0	0.20	0.20	0.0	0.0	0.11	0.11
		ASR	8.6	0.20	16.90	0.30	1.33	0.0	53.01	0.40	19.0	0.20	4.70	0.0	14.50	0.30	10.42	0.10	16.06	0.19
		M-Instrs	196.09	4.5	229.03	5.33	133.38	-	158.70	0.0	132.81	2.0	108.21	-	111.91	5.67	172.12	21.0	155.28	6.42
		M-Nodes	17.74	0.0	18.53	0.67	11.85	-	34.05	0.0	15.64	1.0	16.21	-	11.63	0.67	19.37	0.0	18.13	0.39
	@4	**INIT**	0.0	0.0	0.0	0.0	0.0	0.0	0.20	0.20	0.0	0.0	0.0	0.0	0.10	0.10	0.0	0.0	0.04	0.04
		ASR	5.20	0.0	8.60	0.10	1.02	0.0	43.09	0.20	9.50	0.10	2.30	0.0	8.80	0.20	6.21	0.0	10.59	0.08
		M-Instrs	210.69	-	249.30	13.0	135.9	-	161.58	0.0	144.53	0.0	121.22	-	116.50	2.0	178.15	-	164.73	3.75
		M-Nodes	17.88	-	19.67	0.0	13.0	-	34.29	0.0	15.41	0.0	16.78	-	12.05	1.0	21.23	-	18.79	0.25
K=10		**wASR**	25.10	2.39	40.15	2.38	9.34	2.14	71.37	1.75	40.02	1.96	23.35	2.26	30.48	2.08	30.24	1.81	33.76	2.10
	@1	**INIT**	5.70	5.70	3.90	3.90	3.47	3.47	3.11	3.11	4.0	4.0	4.80	4.80	3.30	3.30	4.11	4.11	4.05	4.05
		ASR	45.10	6.35	59.80	5.60	19.49	6.59	80.96	3.71	59.10	5.12	49.10	6.86	44.60	4.52	47.19	4.63	50.67	5.42
		M-Instrs	170.18	4.76	210.35	6.88	101.21	6.39	154.78	5.41	128.05	3.84	84.46	5.64	112.76	5.58	182.05	5.24	142.98	5.47
		M-Nodes	15.23	0.57	18.35	1.07	10.80	1.16	33.0	0.16	15.72	0.86	12.66	0.99	11.36	1.02	18.62	1.0	16.97	0.85
	@2	**INIT**	2.30	2.30	1.90	1.90	0.71	0.71	1.80	1.80	1.40	1.40	0.80	0.80	1.60	1.60	1.40	1.40	1.49	1.49
		ASR	29.20	2.52	47.3	2.70	11.33	1.75	75.55	1.90	46.3	1.91	28.0	1.79	34.40	2.51	34.97	1.91	38.38	2.12
		M-Instrs	173.84	3.16	214.23	6.85	108.93	5.71	156.73	7.26	129.57	2.47	94.18	3.61	114.63	5.56	173.26	7.05	145.67	5.21
		M-Nodes	16.26	0.48	18.68	0.81	11.50	0.82	33.46	0.11	15.70	0.84	13.61	0.67	11.13	1.12	18.08	1.05	17.3	0.74
	@3	**INIT**	0.50	0.50	0.40	0.40	0.0	0.0	0.80	0.80	0.30	0.30	0.10	0.10	0.50	0.50	0.10	0.10	0.34	0.34
		ASR	15.70	0.50	32.30	0.80	4.29	0.21	67.33	0.90	32.30	0.60	10.40	0.30	24.10	0.90	22.14	0.50	26.07	0.59
		M-Instrs	181.08	2.80	224.91	5.0	108.83	10.50	159.0	4.0	131.05	1.33	105.67	4.33	114.20	3.89	175.48	10.20	150.03	5.26
		M-Nodes	17.45	0.0	19.17	0.75	11.14	2.0	33.88	0.0	15.88	0.67	15.56	1.33	11.2	0.89	19.18	1.60	17.93	0.90
	@4	**INIT**	0.20	0.20	0.0	0.0	0.0	0.0	0.40	0.40	0.0	0.0	0.10	0.10	0.30	0.30	0.0	0.0	0.12	0.12
		ASR	10.4	0.20	21.20	0.40	2.24	0.0	61.62	0.50	22.40	0.20	5.90	0.10	18.80	0.40	16.63	0.20	19.90	0.25
		M-Instrs	192.19	2.50	239.85	6.75	124.5	-	160.23	5.40	140.71	0.0	118.15	4.0	117.37	4.25	176.20	16.0	158.65	5.56
		M-Nodes	18.25	0.0	19.49	0.50	13.45	-	34.12	0.0	16.40	0.0	16.24	2.0	11.53	0.50	19.39	1.0	18.61	0.57

Table 15: Targeted attack at $K = 5$ and $K = 10$ when considering a pool of size 1000 with $\lambda \in \{0, 0.3\}$. In column AVG we report the average of the measures across all models.

			Models																	
			Gemini		GMN		ZEEK		BinFinder		SAFE		jTrans		Trex		PalmTree		AVG	
		λ	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**	**0.0**	**0.3**
K=5		**wASR**	11.35	0.55	19.80	0.65	0.0	0.0	49.55	0.45	19.60	0.43	11.0	0.40	15.17	0.55	13.83	0.55	17.54	0.45
	@1	**INIT**	1.90	1.90	0.90	0.90	0.0	0.0	0.90	0.90	1.0	1.0	0.90	0.90	1.20	1.20	0.70	0.60	0.94	0.94
		ASR	24.30	1.81	38.60	1.50	0.0	0.0	63.63	1.10	38.50	1.31	27.20	1.50	27.60	1.51	26.85	1.51	30.84	1.28
		M-Instrs	167.52	1.44	205.78	4.67	-	-	154.96	7.36	129.22	4.23	94.96	3.67	115.66	3.6	178.75	7.67	149.55	4.66
		M-Nodes	15.69	0.33	17.11	1.07	-	-	33.31	0.0	15.98	0.77	13.80	0.53	11.11	0.67	19.0	0.93	18.0	0.61
	@2	**INIT**	0.40	0.40	0.20	0.20	0.0	0.0	0.50	0.50	0.30	0.30	0.0	0.0	0.20	0.20	0.30	0.30	0.24	0.24
		ASR	13.20	0.30	24.80	0.80	0.0	0.0	55.41	0.60	24.30	0.30	12.30	0.10	17.70	0.40	15.73	0.70	20.43	0.40
		M-Instrs	177.74	3.0	213.77	5.88	-	-	157.06	1.50	136.39	1.33	109.87	4.0	114.71	3.0	173.85	8.0	154.77	3.82
		M-Nodes	16.53	0.0	17.09	1.0	-	-	33.59	0.0	15.68	0.67	14.86	2.0	11.08	1.5	18.19	1.43	18.15	0.94
	@3	**INIT**	0.0	0.0	0.0	0.0	0.0	0.0	0.10	0.10	0.10	0.10	0.0	0.0	0.20	0.20	0.0	0.0	0.05	0.05
		ASR	4.90	0.10	10.50	0.20	0.0	0.0	43.59	0.10	10.60	0.10	2.80	0.0	9.40	0.20	7.62	0.0	11.18	0.09
		M-Instrs	211.73	9.0	241.01	8.0	-	-	159.89	0.0	125.3	0.0	120.07	-	120.39	2.0	171.64	-	164.29	3.80
		M-Nodes	17.96	0.0	18.97	1.0	-	-	34.20	0.0	15.75	0.0	17.07	-	12.84	1.0	20.05	-	19.55	0.40
	@4	**INIT**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.10	0.10	0.0	0.0	0.01	0.01
		ASR	3.0	0.0	5.30	0.10	0.0	0.0	35.57	0.0	5.0	0.0	1.70	0.0	6.0	0.10	5.11	0.0	7.71	0.02
		M-Instrs	245.87	-	234.64	13.0	-	-	161.7	-	129.72	-	130.06	-	119.83	0.0	186.63	-	172.64	6.5
		M-Nodes	16.87	-	18.94	0.0	-	-	34.3	-	15.72	-	18.0	-	12.73	0.0	22.35	-	19.84	0.0
K=10		**wASR**	17.95	1.11	29.93	1.18	0.0	0.0	62.12	1.08	29.98	1.05	17.22	1.07	22.9	1.13	21.89	1.01	25.25	0.95
	@1	**INIT**	3.80	3.80	2.0	2.0	0.0	0.0	1.60	1.60	2.10	2.10	2.30	2.30	1.60	1.60	2.0	2.0	1.92	1.92
		ASR	33.90	3.53	50.10	2.90	0.0	0.0	72.65	2.10	48.50	2.91	38.70	3.70	35.60	2.71	37.17	2.82	39.58	2.58
		M-Instrs	171.54	2.63	207.16	5.72	-	-	154.78	7.86	125.41	3.38	89.19	5.62	113.74	5.19	179.66	7.68	148.78	5.44
		M-Nodes	15.54	0.29	18.06	0.90	-	-	33.11	0.10	15.67	0.69	13.19	0.97	11.10	1.26	18.68	1.36	17.91	0.80
	@2	**INIT**	0.90	0.90	0.60	0.60	0.0	0.0	1.10	1.10	0.70	0.70	0.10	0.10	0.70	0.70	0.60	0.60	0.59	0.59
		ASR	20.80	0.81	36.10	1.20	0.0	0.0	66.53	1.30	35.40	1.0	20.0	0.60	25.90	1.0	24.45	1.01	28.65	0.86
		M-Instrs	179.71	4.12	213.21	5.25	-	-	156.29	3.38	130.25	1.20	99.26	7.0	115.91	5.0	175.13	9.20	152.82	5.02
		M-Nodes	16.49	0.25	17.89	0.83	-	-	33.47	0.0	15.82	0.60	14.16	1.0	10.88	0.80	18.80	1.20	18.22	0.67
	@3	**INIT**	0.0	0.0	0.10	0.10	0.0	0.0	0.50	0.50	0.20	0.20	0.0	0.0	0.30	0.30	0.0	0.0	0.14	0.14
		ASR	10.20	0.10	21.0	0.40	0.0	0.0	57.72	0.60	22.40	0.20	6.80	0.0	17.80	0.40	14.93	0.10	18.86	0.22
		M-Instrs	182.40	9.0	224.53	6.25	-	-	159.46	1.50	130.45	2.0	111.19	-	114.92	4.25	170.79	21.0	156.25	7.33
		M-Nodes	18.02	0.0	18.89	1.0	-	-	33.95	0.0	16.24	1.0	15.79	-	11.39	0.50	19.76	0.0	19.15	0.42
	@4	**INIT**	0.0	0.0	0.0	0.0	0.0	0.0	0.30	0.30	0.0	0.0	0.0	0.0	0.20	0.20	0.0	0.0	0.06	0.06
		ASR	6.90	0.0	12.50	0.20	0.0	0.0	51.6	0.30	13.60	0.10	3.40	0.0	12.30	0.40	11.02	0.10	13.92	0.14
		M-Instrs	200.35	-	245.32	11.0	-	-	160.52	0.0	140.0	0.0	132.94	-	119.30	4.25	181.96	21.0	168.63	7.25
		M-Nodes	18.58	-	19.55	1.0	-	-	34.24	0.0	16.24	0.0	17.88	-	11.34	0.5	20.51	0.0	19.76	0.30

Table 16: Targeted attack at $K = 5$ and $K = 10$ when considering $\lambda = 0.01$ and $|P| \in \{32, 128, 512, 1000\}$. In column AVG we report the average of the measures across all models.

			Models															
			Gemini				SAFE				jTrans				AVG			
		Pool Size	32	128	512	1000	32	128	512	1000	32	128	512	1000	32	128	512	1000
K=5		**wASR**	29.49	12.52	4.56	2.48	54.23	28.25	13.63	8.38	32.17	10.80	3.33	1.80	38.63	17.19	7.17	4.22
	@1	**INIT**	30.26	11.90	3.97	2.08	27.0	8.30	1.90	1.0	42.60	11.10	2.40	0.90	33.29	10.43	2.76	1.33
		ASR	53.08	26.69	11.41	5.95	73.30	45.80	26.40	18.50	70.10	29.70	9.70	4.90	65.49	34.06	15.84	9.78
		M-Instrs	11.36	12.16	13.55	13.83	24.53	28.12	30.52	30.85	7.34	8.42	8.80	7.88	14.41	16.23	17.62	17.52
		M-Nodes	3.67	4.13	4.61	4.80	4.86	5.28	5.62	5.65	1.84	2.11	2.21	1.88	3.46	3.84	4.15	4.11
	@2	**INIT**	16.77	5.26	0.99	0.50	17.30	2.90	0.50	0.20	19.70	2.50	0.20	0.0	17.92	3.55	0.56	0.23
		ASR	36.01	15.48	4.46	2.78	62.10	34.20	16.40	9.80	44.10	10.90	2.60	1.70	47.40	20.19	7.82	4.76
		M-Instrs	11.91	13.19	16.49	18.25	26.13	29.36	32.23	32.60	7.90	9.70	9.19	9.06	15.31	17.42	19.30	19.97
		M-Nodes	3.88	4.59	5.47	5.79	5.02	5.32	5.65	5.82	2.0	2.31	2.38	2.35	3.63	4.07	4.50	4.65
	@3	**INIT**	5.56	0.79	0.10	0.0	7.40	0.90	0.10	0.10	3.30	0.40	0.0	0.0	5.42	0.70	0.07	0.03
		ASR	18.35	5.36	1.49	0.79	46.40	20.10	7.30	3.70	11.0	2.0	0.60	0.30	25.25	9.15	3.13	1.60
		M-Instrs	14.22	15.65	17.87	18.25	28.03	31.86	35.07	35.57	8.48	11.0	9.50	11.67	16.91	19.50	20.81	21.83
		M-Nodes	4.63	5.37	6.0	6.50	5.21	5.68	6.19	6.49	2.20	3.10	3.0	3.33	4.01	4.72	5.06	5.44
	@4	**INIT**	1.49	0.20	0.0	0.0	2.90	0.10	0.0	0.0	0.80	0.10	0.0	0.0	1.73	0.13	0.0	0.0
		ASR	10.52	2.58	0.89	0.40	35.10	12.90	4.40	1.50	3.50	0.60	0.40	0.30	16.37	5.36	1.90	0.73
		M-Instrs	16.67	17.62	16.89	15.25	29.70	33.81	36.23	34.07	9.63	13.0	12.0	11.67	18.67	21.48	21.71	20.33
		M-Nodes	5.23	5.77	6.0	5.50	5.21	5.60	6.32	6.53	2.34	3.67	3.50	3.33	4.26	5.01	5.27	5.12
K=10		**wASR**	50.97	21.23	7.81	4.61	73.78	42.18	21.57	14.70	52.98	19.80	5.35	3.30	59.24	27.74	11.58	7.54
	@1	**INIT**	49.80	17.66	5.85	3.97	50.0	13.90	4.0	2.10	71.0	23.60	4.80	2.30	56.93	18.39	4.88	2.79
		ASR	73.12	36.9	17.56	11.01	85.90	56.80	36.30	26.60	89.0	46.90	14.80	9.30	82.67	46.87	22.89	15.64
		M-Instrs	10.52	12.61	12.50	13.05	23.24	26.63	29.15	29.42	6.83	7.84	8.74	8.98	13.53	15.69	16.80	17.15
		M-Nodes	3.44	4.16	4.21	4.40	4.71	5.16	5.38	5.53	1.73	1.97	2.22	2.28	3.29	3.76	3.94	4.07
	@2	**INIT**	34.82	9.62	2.38	0.99	37.20	8.50	1.40	0.80	49.60	8.60	0.80	0.10	40.54	8.91	1.53	0.63
		ASR	60.32	25.40	8.43	4.27	79.30	46.80	24.80	16.90	75.30	24.50	4.80	2.70	71.64	32.23	12.68	7.96
		M-Instrs	11.22	13.15	14.20	15.81	24.02	28.16	30.60	31.20	7.23	8.91	9.0	10.19	14.16	16.74	17.93	19.07
		M-Nodes	3.67	4.38	4.75	5.07	4.80	5.31	5.44	5.62	1.83	2.16	2.42	2.52	3.43	3.95	4.20	4.40
	@3	**INIT**	18.95	2.78	0.50	0.0	23.20	3.0	0.30	0.20	14.70	1.30	0.10	0.0	18.95	2.36	0.30	0.07
		ASR	42.36	13.89	3.27	1.98	69.10	36.80	14.90	9.0	33.40	5.60	1.30	0.70	48.29	18.76	6.49	3.89
		M-Instrs	12.21	15.25	17.70	18.40	25.24	29.48	31.82	34.06	7.88	9.23	11.92	11.71	15.11	17.99	20.48	21.39
		M-Nodes	3.93	5.21	5.76	5.60	4.99	5.36	5.68	5.91	2.05	2.46	3.08	3.43	3.66	4.34	4.84	4.98
	@4	**INIT**	10.02	0.99	0.20	0.0	13.60	1.40	0.0	0.0	4.0	0.30	0.10	0.0	9.21	0.90	0.10	0.0
		ASR	28.08	8.73	1.98	1.19	60.80	28.30	10.30	6.30	14.20	2.20	0.50	0.50	34.36	13.08	4.26	2.66
		M-Instrs	12.93	16.45	19.75	17.75	26.21	30.31	33.99	35.37	8.70	10.55	14.60	14.60	15.95	19.10	22.78	22.57
		M-Nodes	4.33	5.34	6.20	5.17	5.08	5.29	5.94	6.25	2.34	2.64	4.0	4.0	3.92	4.42	5.38	5.14

Algorithm 1 Greedy Optimization Strategy

Input:

- Query function f_Q
- Set of target variants V

Output: Adversarial example f_{adv}

Definitions:

- Maximum number of iterations Δ
- Set of semantics-preserving transformations TR
- randomStrands(): Initialize the set $STRANDS$ with random strands.
- ir(f_{adv}, pos). Apply the IR transformation to f_{adv} at location pos. Return a new candidate adversarial example.
- ns(f_{adv}, pos). Apply the NS transformation to f_{adv} at location pos. Return a new candidate adversarial example.
- dba(f_{adv}, pos). Apply the DBA transformation to f_{adv} at location pos. Return a list of $|STRANDS|$ candidate adversarial examples, where each candidate consists of adding in a dead branch at position pos within f_{adv} a strand from $STRANDS$.
- sa(f_{adv}, pos). Apply the SA transformation to f_{adv} at location pos. Return a list of $|STRANDS|$ candidate adversarial examples, where each candidate consists of adding at position pos within f_{adv} a strand from $STRANDS$.
- evaluate($cands, V$). Evaluate the objective function in Equation 2 considering the possible candidates $cands$ and the set of target variants V.
- best($advs$). Given a set of adversarial examples, return the one that maximizes the value of Equation 2.

```
1:  $f_{adv} \leftarrow f_Q$
2:  $iter \leftarrow 0$
3:  $STRANDS \leftarrow$ randomStrands()
4:  $advs, POS \leftarrow [\,], [\,]$
5:  while $iter < \Delta$ do
6:      $cands \leftarrow [\,]$
7:      $POS$.update()
8:      for $\langle tr, pos \rangle \in TR \times POS$ do
9:          if $tr ==$ 'IR' then
10:             $cands$.extends(ir($f_{adv}, pos$))
11:         else if $tr ==$ 'NS' then
12:             $cands$.extends(ns($f_{adv}, pos$))
13:         else if $tr ==$ 'DBA' then
14:             $cands$.extends(dba($f_{adv}, pos$))
15:         else
16:             $cands$.extends(sa($f_{adv}, pos$))
17:     $objective\_values \leftarrow$ evaluate($cands, V$)
18:     $prob \leftarrow uniform(0, 1)$
19:     if $prob < \varepsilon$ then
20:         $f_{adv} \leftarrow$ selectGreedy($objective\_values$)
21:     else
22:         $f_{adv} \leftarrow$ selectRandom($objective\_values$)
23:     $advs$.extends($f_{adv}$)
24:     $STRANDS$.update($objective\_values$)
25:     $iter \leftarrow iter + 1$
26: return best($advs$)
```