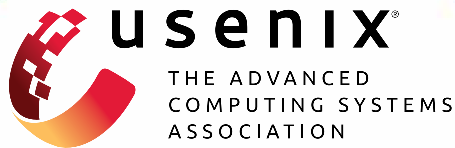

PELICAN: Exploiting Backdoors of Naturally Trained Deep Learning Models In Binary Code Analysis

Zhuo Zhang, Guanhong Tao, Guangyu Shen, Shengwei An, Qiuling Xu, Yingqi Liu, and Yapeng Ye, *Purdue University;* Yaoxuan Wu, *University of California, Los Angeles;* Xiangyu Zhang, *Purdue University*

https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhuo-pelican

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

PELICAN: Exploiting Backdoors of Naturally Trained Deep Learning Models In Binary Code Analysis

Zhuo Zhang[1], Guanhong Tao[1], Guangyu Shen[1], Shengwei An[1],
Qiuling Xu[1], Yingqi Liu[1], Yapeng Ye[1], Yaoxuan Wu[2], Xiangyu Zhang[1]

[1]*Purdue University* [2]*University of California, Los Angeles*

Abstract

Deep Learning (DL) models are increasingly used in many cyber-security applications and achieve superior performance compared to traditional solutions. In this paper, we study backdoor vulnerabilities in naturally trained models used in binary analysis. These backdoors are not injected by attackers but rather products of defects in datasets and/or training processes. The attacker can exploit these vulnerabilities by injecting some small fixed input pattern (e.g., an instruction) called backdoor trigger to their input (e.g., a binary code snippet for a malware detection DL model) such that misclassification can be induced (e.g., the malware evades the detection). We focus on transformer models used in binary analysis. Given a model, we leverage a trigger inversion technique particularly designed for these models to derive trigger instructions that can induce misclassification. During attack, we utilize a novel trigger injection technique to insert the trigger instruction(s) to the input binary code snippet. The injection makes sure that the code snippets' original program semantics are preserved and the trigger becomes an integral part of such semantics and hence cannot be easily eliminated. We evaluate our prototype PELICAN on 5 binary analysis tasks and 15 models. The results show that PELICAN can effectively induce misclassification on all the evaluated models in both white-box and black-box scenarios. Our case studies demonstrate that PELICAN can exploit the backdoor vulnerabilities of two closed-source commercial tools.

1 Introduction

Rapidly advancing Deep Learning (DL) techniques have led to unprecedented capabilities in many areas, such as Computer Vision (CV), Natural Language Processing (NLP), and Robotics. Many believe that similar new capabilities can be developed for cyber-security applications. Recently, DL models are being increasingly used in a wide range of security tasks, such as *binary code disassembly* for malware analysis and code hardening [1–4], *binary similarity analysis* for malware detection, software fingerprinting, and code theft detection [5–11], *decompilation* including type inference [12–15], function signature inference [12, 14–16], and function name prediction [14, 17–19], APT attack forensics [20–24], and intrusion detection [25–28]. These techniques demonstrate superior performance compared to their traditional counterparts that are not based on DL models. The advantages of using such data-driven techniques are clear. In particular, many cyber security tasks have substantial inherent uncertainty. For example, a classic challenge in decompilation is to determine variable types when symbolic information has been stripped away. To recover such types, many heuristics have to be used, leading to uncertainty. Such uncertainty can be naturally modeled by probabilities [29] and reasoned by distribution analysis, which are the underpinnings of DL techniques. In addition, while rules and heuristics used in classic techniques require substantial domain expertise, DL techniques can automatically learn such rules from data. For example, XDA [1] and DeepDi [2] are recent proposals that use DL models to recognize function entries in binary executables and then perform disassembly. They do not require any pre-defined rules or heuristics. Instead, they train DL models from a large code repository and achieve superior performance. Inspired by these successes, many more DL based security solutions will likely be developed and deployed in the near future.

However, recent research [30, 31] in the CV and NLP domains have demonstrated that pre-trained clean DL models are vulnerable to *backdoor attack* [32–37], which is a special kind of *adversarial attack* [38–43]. A specific input pattern called *backdoor trigger* can be derived such that samples stamped with such trigger can cause the model to misbehave, e.g., misclassify to some *target label*. These triggers are usually model specific but not input specific. Examples of triggers include a small patch (for vision models) and a special word (for NLP models). In contrast, adversarial attacks [38, 40] derive unique perturbations for individual samples to induce misclassification and hence are input specific.

The study of backdoor vulnerabilities in naturally trained deep learning models used in binary analysis tasks is of importance in the development of cutting-edge cyber-security solutions. The exploitation of such vulnerabilities can have severe consequences, particularly in malware analysis. Despite the utilization of deep learning models, human analysts still play a vital role in the analysis of malware samples and tracing their origins. Security companies, such

as Mandiant [44], have made substantial investments in the development of reverse engineering tools specifically for use by human analysts, including tools for symbol recovery, function annotation, binary code matching, and binary code attribution. If these models were to be attacked and produce incorrect labels, such as manipulated function names, it could lead to human analysts overlooking critical attack behaviors and ultimately failing in their analysis tasks. This highlights the importance of ongoing research into potential attack techniques and the improvement of the security of these models. It is also worth noting that there is a substantial body of existing work [32–37, 45–47] on injecting backdoors into deep learning models through poisoned training data. However, in our context, we are more focused on finding backdoor vulnerabilities in models trained naturally, referred to as *natural backdoors*. This is because security models are usually trained by trusted parties. Existing attacks in the vision and NLP domains cannot be easily adapted to attack these models. Specifically, vision models deal with a continuous input space, namely, input pixels can change continuously. Hence, existing attacks often leverage gradient descent to invert a backdoor trigger. In contrast, many security DL models deal with discrete inputs, e.g., instruction sequences and log entries. Continuous input changes unlikely yield new valid inputs. For example, changing the encoding of a `ret` instruction `0xC3` to `0xC4` does not yield a valid instruction. NLP models deal with similar discrete inputs, which need to be a sequence of legitimate words. Existing attack methods in the NLP domain mitigate the problem by inverting triggers in the continuous word embedding domain instead of the discrete input domain and finding the input that has the closest embedding to the inversion result [48, 49]. However, backdoor triggers generated for security models often need to preserve strict semantic properties when inserted to an input. For example, a `mov` trigger instruction may completely break the semantics of a malware when inserted.

In this paper, we develop a novel method to identify and exploit backdoor vulnerabilities in DL models used in recent binary analysis models. These models take binary executable code as input and predict various things such as instruction boundaries, function entries, function signatures, and code similarities. They serve a wide range of downstream cyber security applications. Our attack is effective and successfully compromises all the models we study, including some closed-source models that run as commercial online services. By exploiting the backdoors identified by our technique, the attacker can mutate their binaries accordingly (using our tool) before releasing them to the wild and the mutated binaries can fail model-based disassembly/decompilation efforts, disrupt analysis, and so on. Our attack features a trigger inversion method that can guarantee the generated triggers are legitimate instruction sequences. It also has a novel trigger insertion method that not only preserves the semantic of an input binary, but also ensures that *the trigger instruction becomes part of the original semantics after injection*, instead of inaccessible code that can be easily identified and removed.

Our contributions are summarized as follows.

- We study backdoor vulnerabilities in naturally trained DL models used in binary code analysis. Our findings suggest that such vulnerabilities widely exist and they need to be properly mitigated due to their critical roles in security applications.
- We develop a trigger inversion technique that can generate valid instructions as backdoor triggers.
- We devise a trigger injection technique that ensures the trigger becomes an integral part of the original code's semantics and the injected (and patched) code has the same semantics as before.
- We develop a prototype PELICAN and evaluate it on 5 binary analysis tasks and 15 models. Our evaluation shows that PELICAN can achieve 86.09% attack success rate (ASR) with only three trigger instructions. PELICAN has 93.01% higher ASR than a baseline method that adapts an existing NLP trigger inversion technique; 94.14% of injected triggers by PELICAN can evade detection, whereas all the triggers injected by opaque predicates [50] are detected. Our backdoor-injected binaries have 204.23% lower runtime overhead compared to those by opaque predicates. We also conduct a case study of exploiting two closed-source commercial tools, i.e., DeepDi [2] and BinaryAI [11], in the black-box scenario. We have open-sourced a portion of PELICAN [51] in response to ethical concerns raised by the reviewers.

Threat Model. We aim to exploit backdoors in naturally trained models, not models that have injected backdoors by data poisoning [32] or trojaning [52]. This is analogous to finding vulnerabilities in regular software, not malware. We focus on transformer models used in binary code analysis, which are primitives for a wide range of cyber security applications: malware analysis, vulnerability finding, software hardening, decompilation, and forensic analysis. Transformers are the most effective models in these analyses, out-performing other models such as CNN, RNN, and LSTM. Note that attacking models used in other applications, such as network traffic based intrusion detection requires a completely different trigger injection technique (in order to preserve traffic semantics). We hence consider it out of the scope of this paper.

We consider two scenarios: *white-box attack* and *black-box attack*. In the former, we assume the attacker has access to the model such that gradient descent can be applied to generate trigger instructions. Note that many binary analysis tools (and hence the DL models used by these tools) [2, 53–55] are supposed to run by the end users. It is hence reasonable to assume the attacker can access these models. Even if these tools are closed-source, the attacker can still leverage model reverse engineering techniques [56–61] to acquire model copies. In the black-box attack scenario, the attacker does not have access to the subject model. We hence leverage the transferability [62,

```
1.  typedef struct entry_t {
2.    struct entry_t *next;
3.    struct entry_t *prev;
4.    int data;
5.  } Entry;
6.
7.  void init_data(Entry *p, int x)
8.  {
9.    p->data = x;
10. }
```

```
11. void init_auth_entry(
12.       Entry at[], int i)
13. {
14.   Entry *p = &at[i];
15.   Entry *q = &at[i + 1];
16.   p->next = q;
17.   q->prev = p;
18.   init_data(p, 0);
19. }
20.
```

(a) Source code of the motivation example

```
<init_data>:
A1.  push   rbp
A2.  mov    rbp, rsp
A3.  mov    qword ptr [rbp-8], rdi   # store p into a local variable
A4.  mov    dword ptr [rbp-12], esi  # store x into a local variable
A5.  mov    rax, qword ptr [rbp-8]   # load p into register rax
A6.  mov    edx, dword ptr [rbp-12]  # load x into register edx
A7.  mov    dword ptr [rax+16], edx  # p->data = x
A8.  pop    rbp
A9.  ret
```

(b) Assembly code of `init_data` compiled w/ O0

```
<init_auth_entry>:
B1.  movsxd rax, esi                  # store i into a register rax
B2.  lea    rax, [rax+rax*2]          # rax *= 3 (rax = 3 * i)
B3.  shl    rax, 3                    # rax <<= 3 (rax = 24 * i)
B4.  lea    rdi, [rdi+rax]            # rdi = p = &s[i] (sizeof(*p)=24)
B5.  lea    rsi, [rdi+24]             # rsi = q = &s[i + 1]
B6.  mov    qword ptr [rdi], rsi      # p->next = q
B7.  mov    qword ptr [rsi+8], rdi    # q->prev = p
B8.  mov    esi, 0
B9.  call   init_data                 # init_data(p, 0)
B10. ret
```

(c) Assembly code of `init_auth_entry` compiled w/ O3

Figure 1: Motivation Example

63] of these backdoor vulnerabilities. The assumption is that many of these models tend to learn similar features, which are rooted at the compiler behaviors, e.g., function epilogue and prologue, leading to similar vulnerabilities. As such, the attacker can derive a backdoor trigger on a model he has access to, and then use that to exploit another model that he has no access to. In addition, although not explored in this paper, black-box attacks that utilize gradient approximation [64] can be leveraged too. We will leave it to our future work. At the end, we want to point out that our trigger injection technique is general, applicable in both white-box and black-box scenarios.

Natural Backdoor and Universal Perturbation. Most existing backdoor attacks require data poisoning to inject a trigger. During attack, stamping the trigger can universally cause mis-classification for many inputs. We call the problems identified in this paper *natural backdoor* as we can find a trigger in naturally trained models that can be exploited in the same way as those in injected backdoors. Natural backdoor shares a similar nature as *universal adversarial perturbations* [65] which was originally proposed in the CV domain. Specifically, a universal adversarial perturbation that is small and pervasive can cause mis-classification of the subject model. We call the backdoor that we study natural backdoor to raise the alert level as it is analogous to vulnerabilities in software.

2 Motivation

We use an example to motivate our technique. In this example, we use a transformer based technique StateFormer [12]

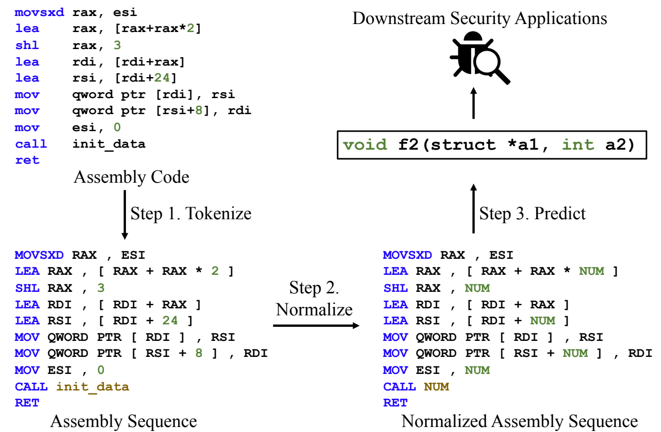

Figure 2: Pipeline of StateFormer

to reverse engineer the function signatures (i.e., function parameters and their types) of code snippets from the leaked *Linux.Mirai* malware [66]. It is an important step for downstream tasks, such as malware behavior understanding and classification. We then show how a naive attack adapted from an existing transformer attack in the NLP domain has difficulties and how PELICAN addresses these challenges.

Figure 1 (a) presents a source code snippet from the malware [66], which is simplified for the illustrative purpose. We show the source code just for better understanding and all the tools in this paper work directly on stripped binaries. Specifically, lines 1-5 declare a doubly linked list. Function `init_data` (lines 7-10) updates the data field of `*p` by a given integer `x`. On the right side, function `init_auth_entry`() initializes the i-th element of a given `Entry` array `at` (lines 11-12), linking it with the $(i+1)$-th element in `at` and setting its `data` field to 0. Figure 1 (b) and Figure 1 (c) show the assembly code of `init_data`() and `init_auth_entry`(), respectively. We comment each instruction with its corresponding source code information for interested readers. We compile `init_data`() with O0 and `init_auth_entry`() with O3. We will show that we can exploit a backdoor to cause StateFarmer to produce wrong function signatures for both functions without changing their semantics.

Transformer Pipeline of StateFormer StateFormer is a transformer-based binary type inference technique that can recover precise function signatures from stripped binaries. It is highly resilient to compiler obfuscation. Fig 2 depicts the pipeline. A piece of assembly code, e.g., the `init_auth_entry`() function, is fed to StateFormer as input and tokenized as an assembly sequence at step 1. Each assembly instruction is split into multiple tokens during tokenization, e.g., the first instruction `movsxd rax, esi` is tokenized to four tokens "`MOVSXD`", "`RAX`", "`,`", and "`ESI`". At step 2, to avoid a prohibitively large vocabulary size, StateFormer abstracts away all the immediate values in instructions, e.g., the token "2" in the second instruction `lea rax, [rax+rax*2]` is normalized as "`NUM`". Note that the token "`init_data`" in instruction `call init_data` has also been normalized, since

```
MOV MOV MOV [ [ [ RDX RDX RDX RDX
PUSH RBP
MOV RBP , RSP
MOV QWORD PTR [ RBP - NUM ] , RDI
MOV DWORD PTR [ RBP - NUM ] , ESI
MOV RAX , QWORD PTR [ RBP - NUM ]
MOV EDX , DWORD PTR [ RBP - NUM ]
MOV DWORD PTR [ RAX + NUM ] , EDX
POP RBP
RET
void f1(struct *a1, int a2, void *a3)
```

(a) Backdoored sequences of `init_data`

```
MOV MOV MOV [ [ [ RDX RDX RDX RDX
MOVSXD RAX , ESI
LEA RAX , [ RAX + RAX * NUM ]
SHL RAX , NUM
LEA RDI , [ RDI + RAX ]
LEA RSI , [ RDI + NUM ]
MOV QWORD PTR [ RDI ] , RSI
MOV QWORD PTR [ RSI + NUM ] , RDI
MOV ESI , NUM
CALL NUM
RET
void f2(struct *a1, int a2, void *a3)
```

(b) Backdoored sequences of `init_auth_entry`

Figure 3: Natural Backdoor Generated by An Existing NLP Trigger Inversion Technique

function addresses are encoded as immediate values in the machine code. A transformer model then predicts the function signature from the normalized assembly sequence at step 3.

Natural Backdoor in StateFormer by Existing Attack Technique. As demonstrated by recent studies [30, 67], natural backdoors are prevalent in the computer vision and NLP domains, even in naturally trained clean models. It is often due to the model being overfitted on some low-level features [30] We speculate similar vulnerabilities may exist in models used in binary analysis. Intuitively, mainstream compilers tend to introduce specific code patterns, e.g., field accesses are performed by first loading the base address of data structure to a register, and then adding the field offset to the register. These low-level syntactic code patterns are prevalent in the training set, likely causing overfit.

We first adapt a state-of-the-art adversarial attack for transformer models in the NLP domain [68] to scan for possible natural backdoors in StateFormer. The technique uses gradient descent to invert some tokens that can cause the model to misbehave on all the given input samples when they are stamped with the tokens. Figure 3 presents the inverted trigger, where Figure 3 (a) and Figure 3 (b) show the backdoored assembly sequences for functions `init_data`() and `init_auth_entry`(), respectively, with the trigger highlighted in red at the beginning and the mis-predicted function signatures at the bottom. Note that although we only present two functions, the inverted trigger can alter the results for over 90% of functions. Compared with the ground truth, the mis-predicated function signature contains an extra `void *a3`.

With further inspection, we find that register `rdx` is used to pass the third argument (i.e. `a3`) in the x64 calling conventions [69]. We further observe that an assembly instruction with a "`MOV`" token and a bracket token always performs a dereference of memory address, e.g., `mov [rax], rbx`. Hence, it is likely that StateFormer mistakenly associates these low-level syntactic features with a pointer being passed as `a3`, without considering real program semantics.

However, even though tokens can be found to cause model misbehaviors, the inverted triggers are not legitimate instructions and hence unable to launch a real-world attack, not to mention preserving input code semantics.

Our technique. PELICAN is inspired by two insights.
<u>*First Insight:*</u> *different from natural languages and high-level programming languages (e.g., Java), assembly code has relatively simple syntax. Taking the syntax into account during backdoor generation can produce legitimate trigger instructions without sacrificing the efficacy.* The syntax of assembly code imposes constraints largely within a single instruction. Our empirical study shows that an individual x64 instruction only comprises 9 tokens on average and 17 tokens at most. As such, we devise a syntax-aware trigger inversion technique taking advantage of a pre-defined instruction dictionary. The instruction dictionary is collected from the SPEC2000 dataset (compiled with a large number of different options) and contains 119640 normalized instructions. The trigger optimization is performed over the instruction dictionary instead of individual tokens, and hence intrinsically follows the syntax of assembly code. Figure 4 (a) presents the instruction sequence with our inverted trigger. The trigger and the mis-predicted function signatures are highlighted at the top and the bottom of the figure, respectively. Observe the second argument is misclassified. In real-world scenarios, three or more trigger instructions may be needed to launch an effective attack (see Section 6), preventing a naive approach of enumerating all instructions in the dictionary, as the complexity of enumerating three instructions is $119640^3 \approx 1.7 \times 10^{15}$.

<u>*Second Insight:*</u> *the injected triggers should not only preserve the program semantics but also become an integral part of the semantics.* With triggers in the form of valid instructions,

Backdoor Trigger: `MOV QWORD PTR [ RSI - NUM ] , <REG>`

```
MOV QWORD PTR [ RSI - NUM ] , RDI
PUSH RBP
MOV RBP , RSP
MOV QWORD PTR [ RBP - NUM ] , RDI
MOV DWORD PTR [ RBP - NUM ] , ESI
MOV RAX , QWORD PTR [ RBP - NUM ]
MOV EDX , DWORD PTR [ RBP - NUM ]
MOV DWORD PTR [ RAX + NUM ] , EDX
POP RBP
RET
void f1(struct *a1, void *a2)
```

```
MOV QWORD PTR [ RSI - NUM ] , RSI
MOVSXD RAX , ESI
LEA RAX , [ RAX + RAX * NUM ]
SHL RAX , NUM
LEA RDI , [ RDI + RAX ]
LEA RSI , [ RDI + NUM ]
MOV QWORD PTR [ RDI ] , RSI
MOV QWORD PTR [ RSI + NUM ] , RDI
MOV ESI , NUM
CALL HEXVAR
RET
void f2(struct *a1, void *a2)
```

(a) Effectiveness of semantic-breaking trigger injection

```
push   rbp
mov    rbp, rsp
mov    qword ptr [rbp-8], rdi
mov    dword ptr [rbp-12], esi
mov    rax, qword ptr [rbp-8]
mov    edx, dword ptr [rbp-12]
mov    dword ptr [rax+16], edx
pop    rbp
ret
void f1(struct *a1, int a2)
```

```
xchg   rsi, rsp
mov    qword ptr [rsi-8], rdi
mov    dword ptr [rsi-12], esp
mov    rax, qword ptr [rsi-8]
mov    edx, dword ptr [rsi-12]
mov    dword ptr [rax+16], edx
xchg   rsi, rsp
ret
void f1(struct *a1, void *a2)
```

(b) Effectiveness of semantic-preserving trigger injection for `init_data`

```
movsxd rax, esi
lea    rax, [rax+rax*2]
shl    rax, 3
lea    rdi, [rdi+rax]
lea    rsi, [rdi+24]
mov    qword ptr [rdi], rsi
mov    qword ptr [rsi+8], rdi
mov    esi, 0
call   init_data
ret
void f1(struct *a1, int a2)
```

```
movsxd rax, esi
lea    rax, [rax+rax*2]
shl    rax, 3
lea    rdi, [rdi+rax]
lea    rsi, [rdi+24]
mov    qword ptr [rsi-24], rsi
mov    qword ptr [rsi+8], rdi
mov    esi, 0
call   init_data
ret
void f1(struct *a1, void *a2)
```

(c) Effectiveness of semantic-preserving trigger injection for `init_auth_entry`

Figure 4: Natural Backdoor Generation and Semantics-preserving Trigger Injection by PELICAN

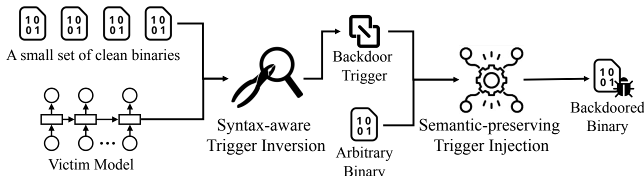

Figure 5: Framework of PELICAN

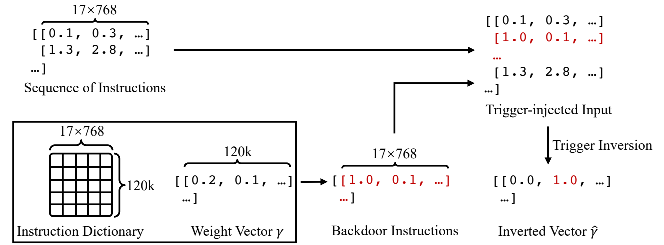

Figure 6: Workflow of syntax-aware trigger inversion

a naive approach to injecting these triggers is to add them as dead-code (and hence having the original input semantics preserved), e.g., using opaque predicates [50]. However, such dead code can be easily detected and eliminated [70–76]. We hence propose a novel injection technique. It is driven by a randomized micro-execution technique that describes program semantics by a set of constraints, and a solving-based synthesis technique that generates code satisfying both the requirements of preserving semantics and injecting triggers (see Section 5). Figure 4 (b) and Figure 4 (c) illustrate the generated code of `init_data`() and `init_auth_entry`(), respectively, after the semantic-preserving trigger injection. We use red to denote the triggers and blue to denote the other entailed patches. In Figure 4 (b), to inject the trigger `mov qword ptr [rsi-8], rdi`, the first instruction in blue exchanges the values of registers `rsi` and `rsp`. The next four instructions retain their original functionalities of manipulating the local variables using `rsi` (in blue) as the stack frame pointer. The seventh instruction remains unchanged and the eighth one (in blue) exchanges registers `rsi` and `rsp` back before returning to the caller. Observe that the trigger becomes a natural and integral part of the program data flow and hard to remove. In Fig 4 (c), `mov qword ptr [rsi-24], rsi` is the trigger instance and also the only change compared with the original code, in which the memory operand changes from `[rdi]` to `[rsi-24]`. The modification is guaranteed to be correct since the constraint of $\texttt{rsi} = \texttt{rdi} + 24$ has been derived from the previous instruction `lea rsi, [rdi+24]`. This illustrates the sophistication of our injection method.

3 Design Overview

The overall design is illustrated in Figure 5. Given a small set of clean binaries and a victim model, PELICAN reverse-engineers the backdoor trigger using gradient descent. The trigger inversion procedure is syntax-aware. That is, each generated trigger instruction follows the proper assembly syntax. PELICAN achieves this goal by constructing an instruction dictionary, where instructions serve as the backdoor trigger candidates. It then leverages gradient descent to search for the trigger instructions that can induce misclassification. The search by its nature is a discrete optimization problem and cannot be directly solved through gradient descent. PELICAN performs a linear relaxation and defines a convex hull for feasible optimization. See detailed discussion in Section 4. With the generated backdoor trigger, the next step is to inject it into some binary without altering the binary's original semantics. Particularly, given an arbitrary binary, PELICAN first introduces a randomized micro-execution process to extract higher order semantics of the given binary, which are represented by program state changes (e.g., register value changes). These changes are encoded by symbolic constraints. They are resolved together with constraints representing the injection of the trigger and the synthesis of needed patches (to preserve semantics), using Z3 [77] (see details in Section 5). Finally, PELICAN produces a backdoored binary with the trigger injected and the same semantics as the original binary. It can induce the desired misclassification on the victim model.

Use Cases of PELICAN. PELICAN operates on assembly code. It leverages datalog disassembly [78], a state-of-the-art binary reassembling tool which has demonstrated its success on thousands of commonly used binaries, to produce reassembleable assembly code from binary. After trigger injection, PELICAN reassembles the code to binary. When source code is available for attackers (e.g., malware developers), PELICAN can also be applied as part of the compilation tool chain, by modifying the intermediate assembly code.

4 Syntax-aware Trigger Inversion

Figure 6 illustrates the workflow of our syntax-aware trigger inversion. Given an input binary, it is first mapped to the embedding space with 17×768 dimensions for each instruction, where value 17 denotes the number of tokens in an instruction and value 768 the embedding size of a token. The resultant matrix is shown on the top left in Figure 6. PELICAN aims to invert a backdoor trigger at the instruction level, which is syntax-aware. We construct an instruction dictionary with 119640 instructions and each instruction is represented by a 17×768-dimension embedding as shown on the bottom left. PELICAN uses a weight vector γ to denote the trigger, whose size is $u \times 119640$ (where u is the number of instructions in the trigger and 119640 is the dictionary size). By multiplying the dictionary with γ, PELICAN obtains the trigger instruction embeddings in the middle of Figure 6, which is then injected to an input sample on the top right. PELICAN leverages gradient descent to optimize the weight vector γ such that it can induce misclassification for a set of samples. Ideally, the inverted vector $\hat{\gamma}$ has only one dimension with 1 that denotes the trigger instruction and the others with 0. For example, the second dimension having value 1 means that the inverted trigger is the second instruction in the dictionary.

4.1 Trigger Generation

The trigger generation aims to produce a small piece of binary code that can induce misclassification on the subject binary analysis model. A straightforward idea is to directly generate code tokens, such as operators, registers, etc. Such a method however cannot guarantee the generated trigger code snippet following the proper syntax of assembly code. As discussed in Section 2 and shown in Figure 3, the backdoor trigger generated by an existing NLP inversion technique is ill-formed.

In PELICAN, we construct an instruction dictionary collected from the SPEC2000 [79] dataset containing 119640 normalized instructions. This provides us with a large pool of feasible trigger candidates. We hence make use of a gradient descent method to search for the possible combination of instructions as the backdoor trigger, which can induce misclassification for a set of input binary samples on the subject model. Assume a subject model $f : \mathcal{X} \mapsto \mathcal{Y}$, the instruction dictionary d (i.e., a large table of embeddings), and a discrete variable $c \in N_{+}^{u \times v}$, where u is the number of instructions in the trigger and v is the size of the dictionary d (i.e., the number of instructions). The trigger generation process can be written as follows.

$$\arg\max_{c} \mathop{\mathbb{E}}_{(x,y)\sim\{\mathcal{X},\mathcal{Y}\}} \mathcal{L}\Big(f\big(x \oplus d(c)\big), y\Big), \tag{1}$$

where (x, y) is a sample from the set $\{\mathcal{X}, \mathcal{Y}\}$ that we use for trigger generation. We assume x has already been mapped to the embedding space for discussion simplicity. $\mathcal{L}$ is the loss function. Operator $\oplus$ denotes the trigger stamping. Operation $d(c)$ looks up the instruction embeddings for index c in the dictionary. Observe that index c is discrete and hence cannot be directly optimized through gradient descent [80–82]. To address the above non-differentiability problem, we construct a convex hull to denote the input space.

Definition 4.1. Let $\mathcal{S} = [1, v]$ be the set of instructions in the dictionary. The convex hull over the input space is $\mathcal{H} = \{\sum_{i=1}^{v} \gamma_i d(i) \mid \sum_{i=1}^{v} \gamma_i = 1, \gamma_i \geq 0\}$.

An input t in the hull is essentially a weighted sum of all instruction embeddings in the dictionary: $t = \sum_{i=1}^{v} \gamma_i d(i)$ and the sum of weights $\gamma_i, i \in \{1, 2, ..., v\}$ must equal to 1. To satisfy the constraint, we introduce a weight vector p and compute γ as the *softmax* over p like the following.

$$\gamma_i = \frac{\exp(p_i)}{\sum_{j=1}^{v} \exp(p_j)}. \tag{2}$$

Note that with the projection, while p is unbounded (and hence easy to optimize), γ can satisfy the constraint of summing up to 1. With the above formalization, we avoid optimizing in the discrete index space (variable c in Equation 1) but rather focus on the weight vector p and hence γ in the convex hull (which is differentiable). The trigger generation is thus to solve the following optimization problem[1].

[1] We use γ as the variable to optimize in the equation for notation simplicity, whereas our true variable is p.

```
movsxd rax, esi
lea    rax, [rdx+rax*2]
lea    rax, [rax+rax*2]
shl    rax, 3
lea    rdi, [rdi+rax]
lea    rsi, [rdi+24]
mov    qword ptr [rdi], rsi
mov    qword ptr [rsi+8], rdi
mov    esi, 0
call   init_data
ret
void f2(struct *a1, int a2, int a3)
```

(a) Before trigger injection

```
movsxd rax, esi
mov    rdx, rax
lea    rax, [rdx+rax*2]
shl    rax, 3
lea    rdi, [rdi+rax]
lea    rsi, [rdi+24]
mov    qword ptr [rdi], rsi
mov    qword ptr [rsi+8], rdi
mov    esi, 0
call   init_data
ret
void f2(void *a1, int a2)
```

(b) After trigger injection

Figure 7: Example trigger by per-instance adversarial attack

$$\arg\max_{\gamma} \mathop{\mathbb{E}}_{(x,y)\sim\{\mathcal{X},\mathcal{Y}\}} \mathcal{L}\Big(f\big(x \oplus \{\sum_{j=1}^{v} \gamma_{ij} d(j)\}_{i=1}^{u}\big), y\Big). \tag{3}$$

In short, we use a linear combination of all the possible instructions in the dictionary as the potential trigger instruction and allow the optimization to find the most promising instruction through gradient descent. The ideal ultimate γ has only one dimension with value 1 and the others value 0. In practice, the weight vector may not always converge to an ideal vector. We hence sort the weight values in p in descending order and select the top 5 instructions as the trigger candidates.

Token Vocabulary and Embedding. In the NLP domain, a *token vocabulary* denotes a mapping that projects a token to its embedding index. The resultant index is further converted to an embedding vector by the embedding layer of the target model. To support an end-to-end inference in the production environment, modern NLP models, e.g., *fairseq* [83], usually include the token vocabulary as one of their integral components. Therefore, similar to existing attack techniques [30, 68, 84, 85], we assume the token vocabulary is available to the attackers in the white-box attack. We also note that all the models used in our evaluation are delivered with their token vocabularies, allowing PELICAN to obtain the embedding vector for any given instruction.

Synonym Instructions. We expand the pool of potential trigger candidates by including synonymous instructions of the backdoor trigger as well. In particular, synonymous instructions are derived from the original trigger instruction by substituting a terminal symbol with a different one. The substituted symbols must be of the same token type, e.g., register tokens "`RDI`" and "`RSI`", operand size tokens "`QWORD`" and "`DWORD`".

Location of Trigger Stamping. To invert position-independent trigger instructions, PELICAN stamps triggers at different locations upon different samples. Those locations are randomly selected prior to the optimization step of trigger inversion, and hence vary among samples. During injection, the place of inserting the trigger instruction(s) is determined by the constraint solver (Section 5).

4.2 Why Not Per-instance Adversarial Attack

A plausible idea is to generate per-instance perturbation as in adversarial attack. For example, different extra instructions may be generated using optimization to induce misclassification for different inputs. However, one of our goals is to

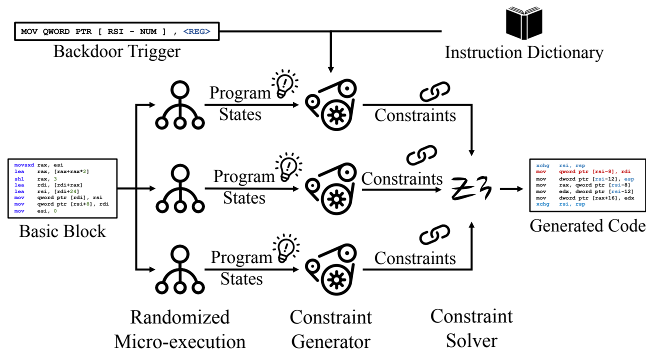

Figure 8: Workflow of semantic-preserving trigger injection

ensure the perturbation does not change the original binary's semantics such that it cannot be easily discarded by sanitization. The trigger injection process hence requires preserving the semantics of the binary through transformation (discussed in Section 5). The triggers generated by per-instance adversarial attacks may not be effective after the injection procedure. Figure 7 shows an example trigger generated by per-instance adversarial attacks. We use the same code snippet as in Section 2 for trigger generation. Figure 7 (a) presents the assembly code with the directly inserted trigger instruction and Figure 7(b) the assembly code after semantic-preserving trigger injection. Observe that directly inserting the trigger instruction can cause the subject model to have the wrong prediction. But the trigger is not effective any more after the semantic-preserving transformation. This is because per-instance adversarial attacks are intended solely to elicit misclassification for a given input, while the injection process modifies said input, thereby rendering the trigger ineffective. Our generated backdoor trigger, on the other hand, is effective for a set of binary samples, and can maintain its effectiveness on the transformed assembly code.

5 Semantic-preserving Trigger Injection

With the triggers in legitimate forms, a naïve approach is to inject the triggers as dead code and hence have the semantics preserved. However, such efforts become ineffective when the input data is sanitized before use, e.g., discarding garbage code or pruning non-critical program paths. Furthermore, a popular technique to inject dead code, i.e., using opaque predicates [50], usually incurs high runtime overhead, rendering the attack infeasible for some performance-sensitive applications, e.g., crypto-mining malware. We hence propose a novel trigger injection technique. It takes a subject basic block, a backdoor trigger, and a pre-collected instruction dictionary as inputs, and synthesizes a semantic-equivalent code snippet that naturally includes the trigger. In PELICAN, we use a customized block selection algorithm to find a basic block to inject triggers. Specifically, given the control flow graph (CFG) of the subject function, PELICAN first eliminates loops by removing backward edges and sorts all basic blocks in the topological order. The block selection operates as follows. Once the code synthesis upon a given block fails, which is very rare, PELICAN moves on to the next block. Since these models are usually quite vulnerable, the simple algorithm is sufficient to generate successful attacks. We leave it to our future work to devise a more advanced block selection algorithm.

Figure 8 depicts the workflow. Given a block, we start multiple micro-executions to extract higher order semantics of the code. The semantics are represented by program states (i.e., the concrete value of each register and memory object) before and after executing the basic block. The constraint generator further transforms the program states to symbolic constraints so that the trigger injection task is reduced to a constraint solving problem. Z3 [77] is hence involved to solve the constraints and synthesize the injected (and patched) code. Note that the code synthesis with a single micro-execution instance is very likely problematic. For example, `mov rax, 12` can be synthesized while the ground truth instruction is `add rax, rax` from a micro-execution result with `rax` being 6 before execution and 12 after. The problem can be avoided with high probability when other micro-execution instances are involved, e.g., having `rax` changed from 37 to 74.

Since the synthesis has only probabilistic guarantees, PELICAN further performs symbolic equivalence checking to ensure the synthesized code is equivalent to the original block. The injection is performed on a different block if the validation is not successful (which is very rare).

An alternative to expressing program semantics is to directly perform symbolic execution (instead of micro-executions then constructing symbolic constraints on *concrete states* like in PELICAN). However, this approach inevitably introduces quantified formulas and array models (for memory objects) in the generated constraints, inducing difficulties in the downstream solver[86–90]. For example, the symbolic constraint of a heap memory read entails an quantifier operation on the symbolic address, which is further translated inside the solver to numerous comparisons with all the possible addresses that have been written to. It is hence very expensive.PELICAN, on the other hand, leverages a randomized concrete representation such that the derived constraints are quantifier-free and can be effectively solved using a quantifier-free bit-vector (`QF_BV`) theory [77].

Also note that PELICAN does not explicitly define tactics of modification. Its transformations are driven by the SMT solver and constrained by the objective requirements of preserving semantic and injecting trigger instructions.

In the remainder of this section, we discuss details of individual component and how we address practical challenges.

5.1 Randomized Micro-execution

The goal of randomized micro-execution is to have a low cost method to concretize program semantics, i.e., how the values of randomly initialized register and memory objects change

$$\Lambda_{reg} \in \mathit{RegisterStore} ::= \mathit{Label} \to \mathit{Register} \to \mathbb{Z} \qquad \Lambda_{mem} \in \mathit{MemoryStore} ::= \mathit{Label} \to \mathbb{Z} \to \mathbb{Z} \qquad \Lambda \in \mathit{ProgramState} ::= \langle \mathit{RegisterStore} \times \mathit{MemoryStore} \rangle$$

$$\frac{\Lambda \models s_1 \Rightarrow \Lambda', \Lambda' \models s_2 \Rightarrow \Lambda''}{\Lambda \models s_1; s_2 \Rightarrow \Lambda''} \quad (1)$$

$$\frac{\begin{array}{c}(\Lambda_{reg}, \Lambda_{mem}) = \Lambda \\ \Lambda'_{reg} = \mathit{InitReg}(\mathit{InitReg}(\Lambda_{reg}, l, r_1), l, r_2) \\ v = \Lambda'_{reg}(l)(r_1)\ op\ \Lambda'_{reg}(l)(r_2) \\ \Lambda' = (\Lambda'_{reg}[\underline{next}(l) \mapsto \Lambda'_{reg}(l)[r_d \mapsto v]], \Lambda_{mem})\end{array}}{\Lambda \models l : r_d \coloneqq r_1\ op\ r_2 \Rightarrow \Lambda'} \quad (2)$$

$$\frac{\begin{array}{c}(\Lambda_{reg}, \Lambda_{mem}) = \Lambda \\ \Lambda'_{reg} = \mathit{InitReg}(\Lambda_{reg}, l, r_a), v_a = \Lambda'_{reg}(l)(r_a) \\ \Lambda'_{mem} = \mathit{InitMem}(\Lambda_{mem}, l, v_a), v_v = \Lambda'_{mem}(l)(v_a) \\ \Lambda' = (\Lambda'_{reg}[\underline{next}(l) \mapsto \Lambda'_{reg}(l)[r_d \mapsto v_v]], \Lambda'_{mem})\end{array}}{\Lambda \models l : r_d \coloneqq \mathbf{R}(r_a) \Rightarrow \Lambda'} \quad (3)$$

$$\frac{\begin{array}{c}(\Lambda_{reg}, \Lambda_{mem}) = \Lambda \\ \Lambda'_{reg} = \mathit{InitReg}(\mathit{InitReg}(\Lambda_{reg}, l, r_a), l, r_v) \\ v_a = \Lambda'_{reg}(l)(r_a), v_v = \Lambda'_{reg}(l)(r_v) \\ \Lambda'_{mem} = \mathit{InitMem}(\Lambda_{mem}, l, v_a) \\ \Lambda' = (\Lambda'_{reg}, \Lambda'_{mem}[\underline{next}(l) \mapsto \Lambda'_{mem}(l)[v_a \mapsto v_v]])\end{array}}{\Lambda \models l : \mathbf{W}(r_a, r_v) \Rightarrow \Lambda'} \quad (4)$$

$$\frac{\begin{array}{c}r \in \underline{dom}(\Lambda_{reg}(l)) \\ \Lambda'_{reg} = \Lambda_{reg}\end{array}}{\Lambda'_{reg} = \mathit{InitReg}(\Lambda_{reg}, l, r)} \quad (5) \qquad \frac{\begin{array}{c}r \notin \underline{dom}(\Lambda_{reg}(l)), v = \underline{random}() \\ \Lambda'_{reg} = \Lambda_{reg}[l \mapsto \Lambda_{reg}(l)[r \mapsto v]] \\ \Lambda''_{reg} = \Lambda'_{reg}[\mathbf{L}_0 \mapsto \Lambda'_{reg}(\mathbf{L}_0)[r \mapsto v]]\end{array}}{\Lambda''_{reg} = \mathit{InitReg}(\Lambda_{reg}, l, r)} \quad (6)$$

$$\frac{\begin{array}{c}v_a \in \underline{dom}(\Lambda_{mem}(l)) \\ \Lambda'_{mem} = \Lambda_{mem}\end{array}}{\Lambda'_{mem} = \mathit{InitMem}(\Lambda_{mem}, l, v_a)} \quad (7) \qquad \frac{\begin{array}{c}v_a \notin \underline{dom}(\Lambda_{mem}(l)), v_v = \underline{random}() \\ \Lambda'_{mem} = \Lambda_{mem}[l \mapsto \Lambda_{mem}(l)[v_a \mapsto v_v]] \\ \Lambda''_{mem} = \Lambda'_{mem}[\mathbf{L}_0 \mapsto \Lambda_{mem}(\mathbf{L}_0)[v_a \mapsto v_v]]\end{array}}{\Lambda''_{mem} = \mathit{InitMem}(\Lambda_{mem}, l, v_a)} \quad (8)$$

Figure 9: Semantics of the randomized micro-execution. Given a basic block $b ::= s$, "$\perp \models s \Rightarrow \Lambda$" denotes that s is evaluated to Λ from an empty state.

⟨BasicBlock⟩	b	::=	s
⟨Statement⟩	s	::=	$s_1; s_2 \mid l : i$
⟨Instruction⟩	i	::=	$r_d \coloneqq e \mid r_d \coloneqq \mathbf{R}(r_a) \mid \mathbf{W}(r_a, r_v)$
⟨Expression⟩	e	::=	$r \mid c \mid e_1\ op\ e_2$
⟨Label⟩	l	::=	$\mathbf{L}_0 \mid \mathbf{L}_1 \mid \mathbf{L}_2 \mid \ldots$
⟨Register⟩	r	::=	$\mathbf{r}_0 \mid \mathbf{r}_1 \mid \mathbf{r}_2 \mid \ldots$
⟨Operator⟩	op	::=	$+ \mid - \mid * \mid \div \mid$ BitOp $\mid \ldots$
⟨Constant⟩	c	::=	$\mathbb{Z}$

Figure 10: Language for branching-free assembly code block

after executing the code. To do so, the micro-execution needs to calibrate objects that are accessed during execution, and tracks the changes of these objects.

Language. To facilitate discussion, we introduce a low-level language to model basic blocks in assembly code. The language is designed to illustrate our key idea, and hence omits many irrelevant details. The syntax is in Figure 10. A basic block is constituted by a statement which is either a concatenation of two statements or an instruction. We label the location before an instruction as l (like a program counter). Instruction $r_d \coloneqq e$ denotes the computation and data movement among registers and e denotes an expression. $\mathbf{R}(r_a)$ and $\mathbf{W}(r_a, r_v)$ model memory read and write operations, respectively, where r_a holds the memory address and r_v holds the value to write. Note that the language does not model branching instructions, which will appear unchanged after injection and remain to be the last instruction of a block.

Definitions. We briefly discuss the definitions used by the semantics of the randomized micro-execution. The formal definitions can be found at the top of Figure 9. Specifically, we use Λ_{reg} and Λ_{mem} to denote the register store and the memory store, respectively. Different program points, distinguished by labels, have different register and memory stores. For instance, the initial value of `rax` is denoted as $\Lambda_{reg}(\mathbf{L}_0)(\texttt{rax})$ and the memory object [52] at the program location $\mathbf{L}_3$ is denoted as $\Lambda_{mem}(\mathbf{L}_3)(52)$. Register and memory stores constitute the program state Λ.

Semantics. The overarching process is to concretely execute the code and check every object before use. If the object is not initialized, we assign it a random value and record the initial value as part of the initial program state (i.e. $\Lambda_{reg/mem}(\mathbf{L}_0)$). As such, the concertized semantics, i.e., which objects are used and what their initial and final states are, can be determined by accessing the program states before and after the execution (i.e., $\Lambda_{reg/mem}(\mathbf{L}_0)$ and $\Lambda_{reg/mem}(\mathbf{L}_{n+1})$ where $\mathbf{L}_{n+1}$ denotes the last program point). We provide the formal rules in Figure 9. For detailed explanations and a running example, interested readers can refer to our supplementary material [51].

5.2 Constraint Generation

With the concretized program semantics (i.e., Λ), PELICAN aims to generate a new code snippet that performs the same state updates on register and memory objects, and injects the trigger instructions. To do so, it first introduces a set of boolean variables x_i^k to guide the code synthesis, where i denotes the instruction ID in the pre-collected dictionary, k denotes the program location, and x_i^k denotes whether the k-th instruction (in the synthesized code) holds ID i (in the dictionary). The code needs to satisfy a number of objective constraints, e.g., $\forall k, (\sum_{i \in ID} x_i^k) = 1$ which guarantees there is only one instruction placed at location k. We further encode the state changes (on register and memory objects) as transformations defined by x_i^k. The changes are constrained to be the same as the original code. Trigger injection is achieved by $(\sum_k x_t^k) \geq 1$ in which t is the ID of the trigger instruction, that is, the trigger instruction is at least inserted once. As such, we reduce the semantic-preserving trigger injection to a satisfiability modulo theories (SMT) solving task, where the trigger-injected code can be derived from a satisfying model of x_i^k, which essentially encodes the set of instructions placed at individual locations, including the trigger instruction and the needed patch instructions to make sure the semantics are preserved after injection.

Figure 11 presents the details. It first defines the inputs

Input:

Λ_{reg} RegisterStore produced by the randomized evaluation
Λ_{mem} MemoryStore produced by the randomized evaluation
$\mathtt{L}_n$ The last instruction's label of the subject block
Ω Instruction candidates used during code synthesis
$\Omega ::= ID \rightarrow Instruction$
t The ID of the trigger instruction, i.e., $\Omega(t)$ is the trigger instruction
m The target length (i.e. # of instructions) of the generated code

To Solve:

x_i^k A boolean variable denotes whether the instruction (in Ω) with ID i is the k-th instruction in the generated code
$1 \leq k \leq m,\ i \in ID,\ x_i^k \in \{0,1\}$

Variables:

$\mathbb{A}$ Accessible memory objects, i.e., memory objects accessed by the subject code
$\mathbb{A} = \underline{dom}(\Lambda_{mem}(\mathtt{L}_{n+1}))$
R_r^k The value of register r after executing the first k generated instructions
$0 \leq k \leq m,\ r \in Register$
M_a^k The value stored in $[a]$ after executing the first k generated instructions
$0 \leq k \leq m,\ a \in \mathbb{A}$
T^k Whether the k-th generated instruction accesses non-accessible memory objects
$1 \leq k \leq m,\ T^k \in \{0,1\}$

Constraint Construction:

Step 1. Initialization

$$① \left(\bigwedge_{r \,\in\, \underline{dom}(\Lambda_{reg}(\mathtt{L}_0))} R_r^0 = \Lambda_{reg}(\mathtt{L}_0)(r) \right) \wedge \left(\bigwedge_{r \,\notin\, \underline{dom}(\Lambda_{reg}(\mathtt{L}_0))} R_r^0 = \underline{random}() \right)$$

$$② \bigwedge_{a \,\in\, \underline{dom}(\Lambda_{mem}(\mathtt{L}_0))} M_a^0 = \Lambda_{mem}(\mathtt{L}_0)(a)$$

Step 2. Constructing Constraints by Induction

Assuming we have constructed the constraints for the first $k-1$ generated instructions, i.e., we have constructed R_r^{k-1}, M_a^{k-1}, and T^{k-1}

Temporary Variables:

${}^iR_r^k$: The value of register r after executing the first k generated instructions if the k-th one's ID is i
${}^iM_a^k$: The value stored in $[a]$ after executing the first k generated instructions if the k-th one's ID is i
${}^iT^k$: Whether the generated code accesses non-accessible memory after executing the first k generated instructions and the k-th one's ID is i

Construction Rules:

$$③\ \forall i \in ID, s.t.\Omega(i) \equiv r_d := r_1\ op\ r_2 \Rightarrow$$
$$\left({}^iR_{r_d}^k = R_{r_1}^{k-1}\ op\ R_{r_2}^{k-1}\right) \wedge \left({}^iT^k = 0\right) \wedge \left(\bigwedge_{r \neq r_d} {}^iR_r^k = R_r^{k-1}\right) \wedge \left(\bigwedge_{a \in \mathbb{A}} {}^iM_a^k = M_a^{k-1}\right)$$

$$\star④\ \forall i \in ID, s.t.\Omega(i) \equiv r_d := \mathtt{R}(r_a) \Rightarrow$$
$$\textbf{\textit{if}}\ a = R_{r_a}^{k-1} \notin \mathbb{A}\quad \textbf{\textit{then}}\ {}^iT^k = 1;$$
$$\textbf{\textit{else}}\ \left({}^iR_{r_d}^k = M_a^{k-1}\right) \wedge \left({}^iT^k = 0\right) \wedge \left(\bigwedge_{r \neq r_d} {}^iR_r^k = R_r^{k-1}\right) \wedge \left(\bigwedge_{a \in \mathbb{A}} {}^iM_a^k = M_a^{k-1}\right)$$

$$\star⑤\ \forall i \in ID, s.t.\Omega(i) \equiv \mathtt{W}(r_a, r_v) \Rightarrow$$
$$\textbf{\textit{if}}\ a = R_{r_a}^{k-1} \notin \mathbb{A}\quad \textbf{\textit{then}}\ {}^iT^k = 1;$$
$$\textbf{\textit{else}}\ \left({}^iM_a^k = R_{r_v}^{k-1}\right) \wedge \left({}^iT^k = 0\right) \wedge \left(\bigwedge_{r} {}^iR_r^k = R_r^{k-1}\right) \wedge \left(\bigwedge_{a' \neq a} {}^iM_{a'}^k = M_{a'}^{k-1}\right)$$

$$⑥ \left(\bigwedge_{r} \Big(R_r^k = \sum_{i \in ID} {}^iR_r^k \times x_i^k\Big)\right) \wedge \left(\bigwedge_{a \in \mathbb{A}} \Big(M_a^k = \sum_{i \in ID} {}^iM_a^k \times x_i^k\Big)\right) \wedge \left(T^k = \sum_{i \in ID} {}^iT^k \times x_i^k\right)$$

Step 3. Constructing the Objective Constraints

$$⑦ \left(\bigwedge_{r \,\in\, \underline{dom}(\Lambda_{reg}(\mathtt{L}_{n+1}))} R_r^m = \Lambda_{reg}(\mathtt{L}_{n+1})(r) \right) \wedge \left(\bigwedge_{r \,\notin\, \underline{dom}(\Lambda_{reg}(\mathtt{L}_{n+1}))} R_r^m = R_r^0 \right)$$

$$⑧ \bigwedge_{a \in \mathbb{A}} M_a^m = \Lambda_{mem}(\mathtt{L}_{n+1})(a) \qquad ⑨ \bigwedge_{1 \leq k \leq m} \textstyle\sum_{i \in ID} x_i^k = 1 \qquad ⑩ \displaystyle\sum_{1 \leq k \leq m} T^k = 0$$

$$⑪ \sum_{1 \leq k \leq m} x_t^k \geq 1 \qquad ⑫ \bigwedge_{1 \leq k \leq m} \bigwedge_{i \,\in\, ID} \left(x_i^k = 1 \vee x_i^k = 0\right)$$

$\star$ Rules ④ and ⑤ can be encoded as a sequence of (consecutive/nested) *if-then-else* statements by enumerating all accessible addresses in $\mathbb{A}$.

Figure 11: Rules of constraint construction

of the constraint generator. Λ_{reg} and Λ_{mem} are the register and memory stores collected from the randomized micro-executions. $\mathtt{L}_n$ denotes the last program location of the subject block, i.e., $\Lambda_{reg/mem}(\mathtt{L}_{n+1})$ denotes the final program state. Ω and t denote the pre-defined instruction dictionary and the ID of trigger instruction (in the dictionary), respectively. The length of the generated block is pre-set as m. In practice, starting from the length of the original code (i.e., n), we gradually increase m until the synthesis succeeds or m reaches a predefined length ($n+20$ in our setting).

Several auxiliary variables are introduced to help model the state changes. $\mathbb{A}$ denotes the accessible memory objects (objects that have been accessed during micro-executions) which can be derived from the final memory store (i.e., $\Lambda_{mem}(\mathtt{L}_{n+1})$). Intuitively, semantic-equivalent blocks should access exactly the same memory objects. R_r^k and M_a^k denote the values of register r and memory object $[a]$, respectively, after executing the first k generated instructions. For instance, $R_{r_0}^0$ denotes the initial value of register r_0, and M_{12}^2 denotes the value of $[12]$ after executing the first two synthesized instructions. We introduce boolean variables T^k to denote whether the k-th generated instruction accesses any invalid memory object. Note that all T^k's are constrained to be 0 during synthesis.

The constraints of keeping the same state changes are constructed by induction. At the initialization stage, we constrain all R_r^0 and M_a^0. Specifically, R_r^0 is set as the initial value of r, if r has been initialized by the original code (constraint ①)). Otherwise, R_r^0 is randomly selected (constraint ①) and we constrain register r to keep the same value in the final program state. This is reasonable as a code snippet accessing additional registers but preserving their initial values is considered semantic-equivalent in our context. M_a^0 is set as the initial value accordingly (constraint ②). For now, memory behaviors are strictly constrained to be the same. Assuming stage $k-1$ is done properly, i.e., R_r^{k-1}, M_a^{k-1}, and T^{k-1} are well constrained, the next step is to construct R_r^k, M_a^k, and T^k. The overall idea is to enumerate all the instructions in the dictionary and model R_r^k, M_a^k, and T^k under the assumption of a specific instruction i being selected. This is done by introducing a few temporary variables ${}^iR_r^k$, ${}^iM_a^k$, and ${}^iT^k$. ${}^iR_r^k$ is defined as the value of R_r^k under the assumption that the k-th generated instruction is i, and ${}^iM_a^k$ and ${}^iT^k$ are defined in a similar fashion. Constraint ③ describes the construction rule for $r_d := r_1\ op\ r_2$. ${}^iR_{r_d}^k$ is updated as the outcome of the expression among $R_{r_1}^{k-1}$ and $R_{r_2}^{k-1}$, while other registers except r_d remain untouched, i.e., R_r^k inherits the value of R_r^{k-1}. ${}^iT^k$ stays false and all memory objects remain unchanged, as memory is not involved. Constraint ④ describes the rule for $r_d := \mathtt{R}(r_a)$. It first checks whether $[R_{r_a}^{k-1}] \equiv [a]$ is accessible. If not, ${}^iT^k$ is marked as true to indicate an access violation. Otherwise, ${}^iR_{r_d}^k$ and other objects are updated according to the semantics. Likewise, constraint ⑤ defines the rule of the memory write operations. Observe that constraints ④ and ⑤ are not typical bit-vector operations but can be encoded as several *if-then-else* statements in Z3. Interested readers can find a running example in our supplementary material [51]. After enumerating all ${}^iR_r^k$, ${}^iM_a^k$, and ${}^iT^k$, variables x_i^k are use to select

the proper R_r^k, M_a^k, and T^k (constraint ⑥). For instance, $R_{r_0}^k$ equals to $\sum_{i \in ID} {}^{i}R_{r_0}^k \times x_i^k$. Note that there is only one x_i^k (i.e., $x_{\hat{i}}^k$) solved as 1 and the rest 0, so that $R_{r_0}^k$ is selected as ${}^{\hat{i}}R_r^k$.

The *objective constraints* are listed at the bottom of the figure. Constraints ⑦ and ⑧ guarantee the synthesized code has the same outcomes. Note that any unused register r (by the original code) needs to keep its initial value (to avoid global side-effects). Constraints ⑨, ⑩, and ⑪ ensure that the generated block is legitimate, shares exactly the same memory behaviors, and has triggers injected, respectively. Note that as mentioned in Section 4.1, we can have a pool of trigger candidates after the trigger inversion. We do not explicitly select the trigger from the candidate pool, but instead let the solver decide which one to choose. Specifically, the constraint ⑪ originally guarantees a specific trigger instruction t is injected. When having a pool of trigger instructions, i.e. $P = \{t_1, t_2, \ldots, t_p\}$, we change the constraint to $\sum_{t \in P, 1 \leq k \leq m} x_t^k \geq 1$, i.e., at least one trigger instruction in the pool being injected. The last constraint guarantees x_i^k has a boolean value.

6 Evaluation

We evaluate PELICAN on 5 binary analysis tasks and 15 models. The evaluation studies the effectiveness of PELICAN in inducing misclassification on subject models in both white-box and black-box scenarios. We investigate the underlying reason that backdoors exist in the pre-trained models. The experiments are performed on a server equipped with a 48-cores CPU (Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz), 256G main memory, and 8 NVIDIA Quadro RTX 6000 GPUs.

6.1 Experiment Setup

Tasks and Models. We evaluate on 5 binary analysis tasks and 15 models, which are presented in Table 1. The first column denotes the tasks. The second and third columns show the techniques and their model architectures. The fourth and fifth columns present the evaluation metrics used for measuring the performance of the subject models and their values. The last column denotes how the models are obtained. In total, we have 10 Transformer-based models and 5 RNN-based models. The performance of all the evaluated models is consistent with that in the original papers. We chose to induce the misbehavior of the instruction boundary detection model (i.e., XDA-`call`) for a specific instruction type (i.e., `call`), to demonstrate a real-world attack that the attacker tries to hide a few critical instructions (e.g., calling a malicious function) instead of completely breaking the disassembler.

Attack Settings. We randomly select 10% binaries from our SPEC2000 dataset (around 2000 functions) to invert backdoor triggers, and launch attacks with samples from the author-provided test sets. This is similar to the real-world attack scenario where the attackers can prepare their own dataset for backdoor inversion without prior knowledge of the victim model's training process. The epoch number and the learning rate of the trigger inversion are 50 and 0.1, respectively.

Computational Cost. As mentioned in Section 2, the memory usage of PELICAN's trigger inversion is dominated by the $119640 \times 17 \times 768$ matrix, which consumes around 6G GPU memory. Note that multiplying the dictionary embedding matrix with the weight vector produces a small matrix of size 17×768. We argue that it is within the capacity of modern GPUs. And there are many attacks [91, 92] in CV and NLP domains that consume much more resources but remain feasible for modern GPUs. On average, PELICAN's trigger inversion takes 30 minutes to generate an effective backdoor for each model.

Attack for Disassembly Models. The disassembly models take bytes as input instead of assembly instructions. We hence develop a dedicated attack for these disassembly models (i.e., BiRNN-func, XDA-func, and XDA-`call`). Specifically, the backdoor triggers are inverted in form of bytes and injected at locations that cannot be reached during runtime, e.g., the preceding bytes before each function entrypoint.

Threats to Validity. In the context of attacking disassembly models, the semantic-preserving syntax-aware effort is not utilized, and the measurement of functionality preservation reflects the efficacy of the underlying binary rewriting engine rather than that of PELICAN. However, our intention in including disassembly models in our evaluations is to demonstrate the pervasiveness of backdoor vulnerability in various deep learning binary analysis tasks. It is possible that other adversarial machine learning techniques may achieve similar attack success rates as PELICAN on disassembly models.

6.2 Attack Effectiveness

The attack results of PELICAN are shown in Table 2. The first column presents the binary analysis techniques and the second column the original performance of these techniques. We apply PELICAN with different backdoor sizes, i.e., the number of instructions that can be injected to a binary function. Note that all the binary analysis techniques are originally evaluated at the function level. We hence follow the same setting by injecting the backdoor trigger in each function to induce misclassification. Columns 3-14 show the attack performance of PELICAN with different backdoor sizes. Column ASR denotes the attack success rate, i.e., the percentage of functions that a subject model produces correct predictions for before attack but wrong after. Column Score denotes the performance of the subject model measured using its original metric as shown in Table 1. Column Dis. presents the edit distance between the trigger-injected function and its original version. Note that the edit distance is presented as the ratio to the size of original functions. For example, with x the edit distance and y the original size, x/y is presented. Observe that with only one injected instruction, PELICAN has already over 90% ASR on BiRNN-func, XDA-func, and XDA-`call`. The

Table 1: Summary of models used, along with how we collect these models (i.e., Source). _P_, _T_, and _O_ denote the models are provided by the authors, trained with the author-provided dataset, and trained with our own dataset, respectively. The techniques named with a suffix **++** are enhanced by PalmTree [93], an instruction embedding technique.

Task	Technique	Architecture	Metric	Score	Source
Dis-assembly	BiRNN-func[3] [3]	Bidirectional RNN	Precision	99.12%	_O_
	XDA-func[3] [1]	Transformer	Precision	99.36%	_P_
	XDA-`call`[4] [1]	Transformer	Precision	100.00%	_P_
Function Signature Recovery	StateFormer [12]	Transformer	Precision	96.60%	_T_
	EKLAVYA [16]	RNN	Precision	70.29%	_T_
	EKLAVYA++ [16]	RNN + TE[1]	Precision	74.25%	_T_
Function Name Prediction	in-nomine [18]	Transformer	Precision[2]	33.97%	_P_
	in-nomine++ [18]	Transformer + TE[1]	Precision[2]	25.26%	_T_
Compiler Provenance	S2V [7]	RNN + structure2vec	Precision	73.06%	_T_
	S2V++ [7]	RNN + TE[1] structure2vec	Precision	73.64%	_T_
Binary Similarity	Trex [5]	Transformer	Top@1 Acc	91.11%	_P_
	SAFE [6]	Bidirectional RNN	Top@1 Acc	89.29%	_P_
	SAFE++ [6]	Bidirectional RNN + TE[1]	Top@1 Acc	87.01%	_T_
	S2V-B [7]	RNN + structure2vec	Top@1 Acc	81.10%	_T_
	S2V-B++ [7]	RNN + TE[1] structure2vec	Top@1 Acc	82.78%	_T_

[1] The instruction embedding is generated by a transformer model [93].
[2] We consider a prediction correct if there are more than 5% tokens correctly predicted.
[3] BiRNN-func and XDA-func are to detect function boundaries.
[3] XDA-`call` is to detect boundaries of all `call` instructions.

ASR on other models such as EKLAVYA++, SAFE, SAFE++, and S2V-B are also reasonable, with an over 60% ASR when only one instruction is injected. When the backdoor size is increased to 5, PELICAN is able to break all the evaluated models with over 80% ASR. The performance of these models measured by their corresponding metrics is only 5.38% on average. The edit distance increases when injecting more backdoor instructions. This is reasonable as PELICAN needs to add more instructions in each function. One may notice that the edit distance is relatively high for StateFormer. It is because the functions from StateFormer's test set are shorter than others, rendering the ratio high. We find when the backdoor size is 3, PELICAN achieves a good balance with 86.49% ASR and 11.14% edit distance on average.

6.3 Comparison with Baselines

We compare PELICAN with three baselines: using an adopted NLP inversion technique to invert triggers, using opaque predicates to inject triggers, and a state-of-the-art per-instance adversarial attack that does not rely on trigger inversion.

Adopted NLP Trigger Inversion Technique. The base-

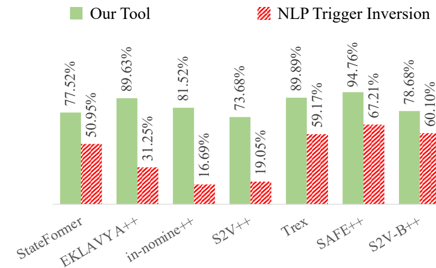

Figure 12: Comparison between an adopted NLP trigger inversion technique and PELICAN

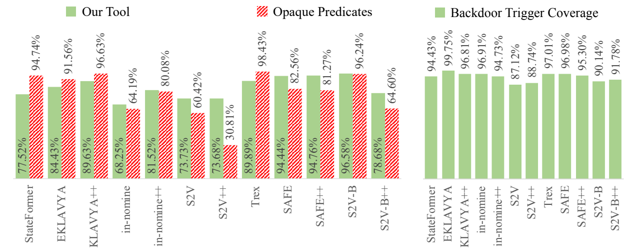

Figure 13: Comparison between a baseline method using opaque predicts to inject triggers and PELICAN

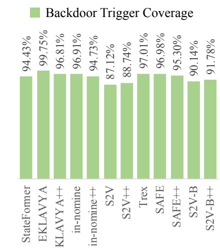

Figure 14: The runtime trigger coverage

line method adopts an existing NLP trigger inversion technique [68]. During the trigger inversion, we gradually discard the opcode tokens that take more operands, so that the baseline method can eventually invert a legitimate one. For instance, in the worst case, only nullary opcodes are available and any inverted token is a legitimate nullary instruction. We use all the 7 models that take tokenized assembly sequence as input, and compare the baseline method's attack performance to PELICAN's. The backdoor size is 3 and the result is shown in Figure 12. Observe that the baseline method has at most 67.21% ASR on SAFE++, whereas PELICAN has 94.76% ASR. The baseline performs worst on in-nomine++ with only 16.69%. PELICAN, on the other hand, still has 81.52% ASR. This is because the target triggers, i.e., the ones that the models are undesirably overfitting to, are hard to invert when the inversion technique does not take the assembly syntax into account. Overall, PELICAN substantially outperforms the baseline inversion method.

Trigger Injection by Opaque Predicates. Another baseline method leverages opaque predicates to inject backdoor triggers [94]. It aims to attack Android malware classifiers based on code features, e.g., code size. It uses opaque predicates to inject arbitrary bytes to subject malwares. Opaque predicates are predicates whose true branches can never be taken. As such, any instructions (or even arbitrary bytes) guarded by these predicates will never be executed. Since our models are not based on code size, their technique is not directly applicable. We hence leverage its opaque predicate transformation to inject the trigger instructions generated by our inversion technique. We compare attack performance on the 12 models that are not the disassembly ones. We additionally study the

Table 2: Attack effectiveness of PELICAN with different trigger sizes. **ASR** denotes the attack success rate and **Score** the model performance. **Dis.** denotes the edit distance between the trigger-injected functions and the original versions over the originals.

Technique	Original Score	Backdoor Size: 1			Backdoor Size: 3			Backdoor Size: 5			Backdoor Size: 7		
		ASR	Score	Dis.	ASR	Score	Dis.	ASR	Score	Dis.	ASR	Score	Dis.
BiRNN-func	99.12%	91.24%	8.50%	0.15%	96.35%	3.39%	0.46%	98.12%	1.62%	0.76%	98.12%	1.62%	1.06%
XDA-func	99.36%	93.44%	6.53%	0.15%	98.31%	1.68%	0.46%	98.32%	1.67%	0.76%	98.32%	1.67%	1.06%
XDA-`call`	100.00%	99.49%	0.51%	3.95%	99.57%	0.43%	6.59%	99.57%	0.43%	9.23%	99.57%	0.43%	11.86%
StateFormer	96.60%	45.09%	53.78%	11.73%	77.52%	21.73%	35.19%	89.51%	10.66%	58.65%	94.88%	4.89%	82.11%
EKLAVYA	70.29%	55.33%	32.91%	2.76%	84.43%	12.88%	7.16%	92.93%	6.45%	12.84%	96.11%	3.73%	16.15%
EKLAVYA++	74.25%	60.49%	31.85%	2.12%	89.63%	8.51%	6.36%	92.63%	8.05%	10.60%	93.81%	6.75%	14.83%
in-nomine	33.97%	42.85%	19.42%	2.44%	68.25%	10.79%	8.56%	83.75%	5.52%	15.89%	85.42%	4.95%	19.55%
in-nomine++	25.26%	47.26%	13.32%	1.83%	81.52%	4.67%	6.72%	87.65%	3.12%	11.61%	92.25%	1.96%	17.72%
S2V	73.06%	42.64%	42.64%	5.90%	73.73%	19.51%	17.71%	83.66%	12.12%	29.52%	89.66%	7.64%	41.33%
S2V++	73.64%	32.87%	51.71%	4.78%	73.68%	19.91%	14.35%	85.28%	11.06%	23.92%	90.88%	6.81%	33.48%
Trex	91.11%	59.32%	37.83%	1.74%	89.89%	9.50%	5.22%	96.40%	3.39%	8.70%	98.30%	1.60%	12.18%
SAFE	89.29%	74.18%	23.68%	5.09%	94.44%	5.20%	15.26%	98.04%	1.84%	27.98%	98.99%	0.96%	38.15%
SAFE++	87.01%	64.75%	31.58%	3.82%	94.76%	4.98%	11.45%	98.79%	1.15%	19.08%	99.71%	0.30%	26.71%
S2V-B	81.10%	89.55%	8.86%	4.52%	96.58%	3.09%	13.57%	98.14%	1.66%	22.62%	98.94%	1.02%	31.67%
S2V-B++	82.78%	59.36%	34.52%	6.03%	78.68%	18.19%	18.09%	86.12%	11.93%	30.16%	89.97%	8.79%	42.22%
Average	**78.46%**	**63.87%**	**26.51%**	**3.80%**	**86.49%**	**9.63%**	**11.14%**	**92.59%**	**5.38%**	**18.82%**	**95.00%**	**3.54%**	**26.01%**

impact of data sanitization (e.g., remove instructions that are not covered during execution) and the runtime overhead of trigger-injected binaries.

Attack Success Rate. Figure 13 presents the ASRs of the two methods. PELICAN achieves comparable ASRs with the baseline methods on all the models. Observe that the baseline attack performs slightly better for function signature recovering tasks (i.e., StateFormer, EKLAVYA, and EKLAVYA++). This is because it always just injects triggers at the beginning of functions, achieving the maximum attack effects on function signature recovery models. However, this may backfire. Observe that the baseline has only 30.81% ASR on S2V++, while PELICAN's still reaches 73.73%. Overall, compared to the baseline, our attack is just as effective and more stable.

Input Sanitization. Many existing research works [94–97] have emphasized the importance of inconspicuousness and hence aim at generating stealthy backdoored/adversarial samples. Malware Makeover [95], a state-of-the-art malware evasion technique, further asserts the possible defense of pruning out crafted bytes in unreachable regions of the binary. De-obfuscation techniques [98–100] are also largely adapted by the RE community. We hence study possible defense by sanitizing input binaries. One possible approach is to dynamically execute the program and eliminate the uncovered code. Note that although one cannot discard unexecuted code in general, in the context of de-obfuscation (before passing an executable to a malware classifier), it is justifiable to suppress the un-executed instructions [101, 102]. We inject different triggers (which are inverted from 12 subject models) into the SPEC2000 programs and study the trigger coverage (i.e., the percentage of dynamically covered trigger instructions). We use the reference input set provided by SPEC2000 to collect runtime information. Note that we cannot collect the trigger coverage for the author-provided test programs due to the lack of valid input data. Figure 14 details the coverage. Observe that even in the worst case, PELICAN still achieves 87.12% trigger coverage, i.e., almost all triggers injected by PELICAN are dynamically executed (and hence an integral part of the sanitized programs). On the other hand, triggers injected by the baseline method can be easily eliminated since the dead code is never executed.

We additionally implement a naïve opaque predicate detector inspired by [76]. It is a hybrid static-dynamic technique without considering complex program semantics. The results show 54.84% opaque predicates are detected, indicating that more than half of injected triggers can be eliminated. Many other state-of-the-art opaque predicate detection techniques [70–75, 103] are believed to have better performance.

Runtime Overhead. We use SPEC2000 programs to study the runtime overhead of backdoor-injected binaries by different trigger injection methods. To avoid the randomness from each execution, we run all the binaries for 3 times and obtain the average. On average, the binaries with PELICAN's injected backdoors have 4.36%, 8.21%, 10.98%, and 15.13% runtime overhead when the trigger size is 1, 3, 5, and 7, respectively. The baseline opaque predicates, on the other hand, has around 200% runtime overhead, rendering the attack infeasible for performance-sensitive applications. Recall that we adapt the settings of an existing work [94], where opaque predicates are to validate if a set of random values satisfy a preset 3-SAT formula, to avoid being easily determined as a bogus predicate (by a static analysis). The runtime overhead is mainly caused by the execution of opaque predicates [50]. Detailed results are provided in our supplementary material [51].

Instance-specific Attack. Malware Makeover [95] (hereinafter referred to as *MalMakeover*) is a state-of-the-art per-instance attack against malware classifiers. For a given malware sample, MalMakeover iteratively applies semantics-preserving transformations upon the sample until the resultant variant induces misclassification. To

efficiently guide the transformations, MalMakeover proposes an optimization algorithm which, at each iteration, only selects a transformation that can entail a lower attack *CW loss* [40] (compared to the current malware variant). MalMakeover achieves a high evasion rate against DL-based malware detectors [95]. It is hence interesting to compare its performance on binary analysis models with ours.

Experiment Configuration. In our modifications to the original version of MalMakeover, we have made two significant alterations. Firstly, we have reduced the scope of mutation from an entire program to a single function. In contrast to the original MalMakeover, which selected functions to mutate in an iterative manner, our variant focuses exclusively on a given function. This modification was necessary as our subject models accept input in the form of a single function, as opposed to the original MalMakeover, which took malware binaries as input. Secondly, we have replaced the disassembling frontend of MalMakeover with a custom implementation that accepts text-form assembly code as input. This change was required as the test sets provided by the authors of the subject models did not consist of complete programs, but rather individual functions. To accomplish this, we utilized Keystone for assembly and Capstone for disassembly, which provided the necessary information at the instruction-level. Additionally, since the original MalMakeover relied on a register liveness analysis, which is not supported by Capstone, we developed such an analysis ourselves. Our variant of MalMakeover supports both in-place randomization and displacement. It is worth noting that the original design of MalMakeover was intended for models that accept a program as input and output a predicted label, which is essential for the use of CW loss. However, for the disassembly and binary similarity tasks, the output is in the form of sequences and embedding vectors, respectively, making them incompatible with the original design of MalMakeover. Furthermore, MalMakeover calculates the difference of input embeddings before and after mutation. The S2V and S2V++ models, which take CFG-like graphs as inputs, pose a challenge in this regard as the embedding difference cannot be easily calculated after displacement mutation, which significantly alters the CFG. Consequently, we exclude disassembly models, binary similarity models, S2V, and S2V++ from our evaluation. In our study, we examine the efficacy of untargeted attacks (i.e., causing the model to misclassify a sample to any other label), the performance of targeted attacks (i.e., causing the model to misclassify a sample to a specified label), and the runtime overhead of mutated binaries.

Untargeted Attack. Table 3 presents the ASRs of untargeted attacks performed by PELICAN and MalMakeover. The first two columns denote the subject models and the attack techniques, respectively. Columns 3-6 present the ASRs with backdoor sizes of 1, 3, 5, and 7. It is important to note that MalMakeover does not reverse trigger instructions, thus the metric of backdoor size is not directly relevant. To make a meaningful comparison, for each function sample, we guarantee that the edit distances of the function variants generated by PELICAN and MalMakeover are equivalent with respect to the original function. To calculate the edit distance, we compare the original assembly sequence of the subject function to the mutated sequence and consider each instruction as a unit for the purposes of counting. Observe that PELICAN and MalMakeover achieve comparable ASRs for untargeted attacks. Specifically, MalMakeover slightly outperforms PELICAN on in_nomine++ with a backdoor size of 3 and on StateFormer with 1 and 3, while PELICAN achieves superior ASRs in the other settings. Also note that for StateFormer, although MalMakeover gets better ASRs with backdoor sizes of 1 and 3, its attack performance reaches an upper bound ASR of 85% when the backdoor size is larger than 3, while the ASR of PELICAN is close to 95% with a backdoor size of 7. We have also conducted a comprehensive evaluation of the average time required by PELICAN and MalMakeover to launch a successful untargeted attack. The parameters for backdoor size and the number of micro-execution instances were set to 5 and 3, respectively. Our results demonstrate that PELICAN requires an average of 19.23 seconds per function to achieve a successful untargeted attack, whereas MalMakeover requires only 7.68 seconds. It is worth noting that PELICAN necessitates an additional 30 minutes to generate an effective backdoor for each model through trigger inversion. We have carried out further investigation to understand the factors contributing to MalMakeover's efficient performance in this regard. We observe that code displacement [104] is the most effective transformation. It moves a branching-free code piece to a new executable section and fills the original place by a leading `jmp` instruction

Table 3: Attack success rates of untargeted attacks

Model	Tool	Backdoor Size			
		1	3	5	7
EKLAVYA	PELICAN	55.33%	84.43%	92.93%	96.11%
	MalMakeover	29.07%	57.31%	70.34%	76.64%
EKLAVYA++	PELICAN	60.49%	89.63%	92.63%	93.81%
	MalMakeover	25.82%	58.13%	67.80%	75.61%
in_nomine	PELICAN	42.85%	68.25%	83.75%	85.42%
	MalMakeover	41.15%	67.85%	83.38%	85.37%
in_nomine++	PELICAN	47.26%	81.52%	87.65%	92.25%
	MalMakeover	41.81%	85.19%	87.18%	91.22%
StateFormer	PELICAN	45.09%	77.52%	89.51%	94.88%
	MalMakeover	50.82%	84.30%	84.34%	84.41%

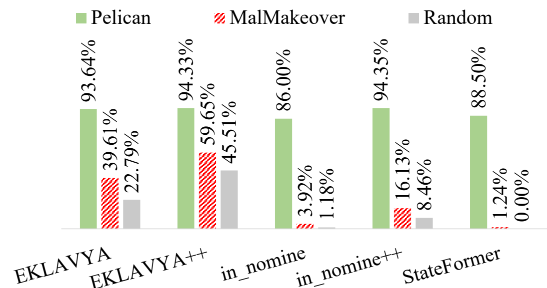

Figure 15: Attack success rates of targeted attacks

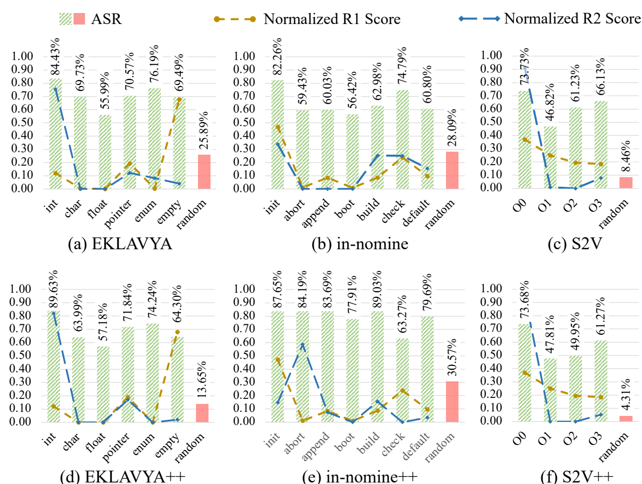

Figure 16: The relation between ASR and the underlying training bias. R1 and R2 scores denote the sample-level and feature-level bias, respectively. *Random* denotes a baseline method that randomly selects trigger instructions.

(to redirect the control flow to the displaced code) and a set of *semantic-nop* instructions (i.e., instructions that cumulatively do not affect the memory or register values and have no side effects). This is mainly because, when pre-processing data, these binary analysis models do not take control-flow information into consideration. Instead, they assume all instructions of a function are contiguous in memory, and therefore consecutively collect instructions from the function entrypoint, until the first instruction that belongs to another function. Despite being valid in most cases, the contiguity assumption is broken by code displacement, where several instructions are displaced to a distinct memory space. As a result, these binary models can only access non-displaced instructions and understandably have an inferior performance. The success of MalMakeover in untargeted attacks also suggests *the de-facto necessity of a proper data pre-processing step when developing binary analysis models, e.g., restoring displaced instructions*.

Targeted Attack. Figure 15 depicts the performance of targeted attacks against 5 models. The green, red, and gray bars denote the ASRs of PELICAN, MalMakeover, and a baseline approach that randomly selects backdoor instructions to inject without trigger inversion, respectively. The target label of EKLAVYA, EKLAVYA++, and StateFormer is "int", and that of in_nomine and in_nomine++ is "init". We adapt the setting of backdoor size 7. Observe that the best ASR MalMakeover can achieve is close to 60% and its ASRs on in_nomine and StateFormer are 3.92% and 1.24%, respectively. In comparison, PELICAN always achieves an ASR above 85% for all the subject models. The baseline method performs the worst. We further investigate the underlying reason of the superiority of PELICAN in targeted attacks. As mentioned in Section 2, many models undesirably learn some low-level syntactic features. For example, StateFormer overfits on "`add [r8], esi`" and always predicts a function's first argument as "int" as long as the function contains that `add` instruction (i.e., the trigger instruction). Note that PELICAN can effectively identify trigger instructions via the syntax-aware trigger inversion. Meanwhile, PELICAN's semantic-preserving trigger injection is able to inject arbitrary inverted trigger instructions into the subject binary. On the other hand, MalMakeover mutates binaries by performing a set of pre-defined semantics-preserving transformations. These pre-defined transformations can introduce a few typical types of instructions (e.g., `nop`, `push`, and `pop`) into the subject binary, while those in the trigger are beyond this scope. For example, it is less likely for these transformations to precisely produce the "`add [r8], esi`" instruction, rendering a suboptimal ASR of MalMakeover on StateFormer.

6.4 Functionality Preservation

PELICAN employs a semantic-preserving trigger injection technique to ensure that the functionality of the mutated binaries is retained. In this study, we have conducted an empirical examination of the preservation of functionality in backdoor-injected binaries. Our dataset consists of binaries from SPEC2000 [79], SPEC2006 [105], Binutils 2.39 [106], and Coreutils 8.25 [107]. These datasets are well-suited for our purposes as they come equipped with a large number of comprehensive test cases. For each binary, we have applied 12 backdoor triggers (inverted from 12 non-disassembly models), resulting in a total of 1800 mutated binaries. Our attacks were executed under two distinct scenarios: with and without access to the source code. In the former scenario, PELICAN was integrated into the compilation toolchain and inserted backdoor instructions into the compiler-generated assembly code, which was then converted into binary form by the default assembler. In the latter scenario, the subject binaries were first disassembled into reassembleable assembly code using datalog disassembly [78] and then instrumented by PELICAN. The results show that, in the source-assisted setting, all the mutated binaries produce the expected outputs on the benchmark test cases, demonstrating the effectiveness of PELICAN's semantic-preserving trigger injection. In the binary-only setting, 93.3% of the mutated binaries produce the expected outputs, while the rest of them crash or produce incorrect outputs. These failures are due to limitations in the datalog disassembly process.

6.5 Why Backdoors Exist in These Models?

In this section, we investigate the underlying reason that backdoors exist in the models of three classification tasks (i.e., function signature recovery, function name prediction, and compiler provenance). The binary similarity models are not used as their outputs are embedding vectors instead of some specific labels. For each task, we select two models and a few classes. Particularly, we study the relation between the attack success rate and the training bias evaluated by two metrics:

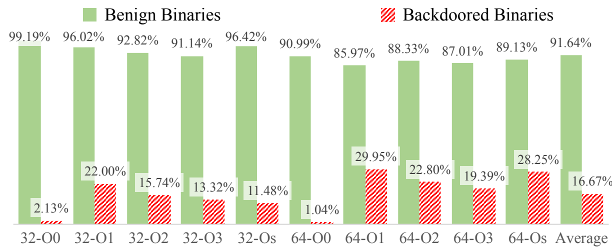

Figure 17: F1 score of the function boundary identification by DeepDi [2], a closed-sourced commercial disassembler

sample-level bias and *feature-level bias*. The sample-level bias (R1) calculates the ratio of target class samples in the whole training set. The feature-level bias (R2) measures the ratio between two computed percentages: the percentage of samples containing backdoor instructions in the target class, and the percentage of samples containing backdoor instructions in other classes. For instance, assuming there are n training samples, n_a class-A samples, k_1 class-A samples containing the inverted trigger instructions, and k_2 non-class-A samples containing the triggers, $R1 = \frac{n_a}{n}$ and $R2 = \frac{k_1}{n_a} / \frac{k_2}{n - n_a}$. The results are shown in Figure 16. The x-axis denotes different target classes where *random* denotes a baseline method that the trigger instruction is randomly selected rather than inverted by PELICAN. The y-axis presents the ratio values. The bars show the ASRs on the test sets when using the corresponding class as the target during trigger generation. The dashed brown and blue lines show the results for the sample-level bias (R1) and the feature-level bias (R2), respectively. We have normalized the R1 and R2 scores for better visualization. In Figure 16 (a), observe that the R2 line has a similar trend as the ASR bars, indicating EKLAVYA overfits on a few distinctive instructions in each class. Specifically, for the target class `int`, both ASR and R2 are high, meaning EKLAVYA largely relies on the backdoor instructions (e.g., `mov ebx, edx`) for predicting class `int`. Similar observations can be made for other models as shown in Figure 16 (b)(c)(d)(e)(f). The R1 line has the same trend but the trend is less significant, indicating a smaller contribution of the sample-level bias. To better understand the root cause of the vulnerabilities, we investigate a concrete case of EKLAVYA. Recall that the trigger instruction of "`mov ebx, edx`" is able to misguide EKLAVYA to incorrectly predict the subject function's third argument as an integer. According to the x64 calling conventions [69], register `rdx` (i.e., the 64-bit extension of `edx`) is used to pass the third argument, which is however call-clobbered. Meanwhile, register `rbx` (i.e., the 64-bit extension of `ebx`) is a call-preserved general purpose register. To consistently use the argument's value, mainstream compilers tend to load it into a call-preserved register. As a result, "`mov ebx, edx`" is frequently used by functions whose third arguments are 32-bit integers. It hence introduces natural bias into the training dataset, where such "`mov`" instructions are prevalent in the aforementioned functions but relatively rare in the rest of the dataset. Without the awareness and a proper remediation of such a bias, EKLAVYA undesirably relies on the low-level syntactic features (i.e., the presence of "`mov ebx, edx`") to make prediction, regardless of the underlying program semantics. Also observe that the baseline attack (i.e. *random*) performs poorly, rendering the importance of the syntax-aware trigger inversion. Overall, this study suggests that these models may focus on a few very distinctive instructions for prediction instead of relying on input semantics. PELICAN can hence diagnose and exploit the vulnerability by inverting the trigger instructions and injecting the triggers back to the subject functions.

6.6 Additional Evaluation

We conducted additional evaluations to explore how the generated backdoor on one subject model can transfer to other models and to study how the number of micro-execution instances affects PELICAN's performance. We also carry out an adaptive defense where pre-trained models are hardened (adversarially trained) with backdoor-injected data. Details can be found in our supplementary material [51].

7 Case Study

In this section, we present a case study of a black-box attack in which we launch transfer attacks on proprietary binary analysis models. Specifically, we investigate the security of DeepDi [2], a recently-proposed commercial disassembler. Our results have been disclosed in a responsible manner, and the authors of DeepDi have acknowledged the issue and are working to improve their product.

DeepDi. DeepDi is a state-of-the-art GNN-based disassembler, proposed by DeepBits Technology [108]. It achieves low false positive and negative rates on both normal and obfuscated code. In this case study, the backdoor trigger is generated from XDA and the subject binaries are from SPEC2000. The trigger comprises 3 bytes and is injected as padding bytes before each function entrypoint. Figure 17 presents DeepDi's F1 score for function boundary identification. The green and red bars denote benign and backdoor binaries, respectively. The x-axis denotes the compilation flags, and the last two columns show the average numbers. The y-axis denotes the F1 score. Observe that our attack is most effective for binaries compiled by O0, where the F1 scores have decreased from 99.19% to 2.13% and from 90.99% to 1.04% for 32-bit and 64-bit programs, respectively. On average, the F1 scores drop from 91.64% to 16.67% after the backdoor attack, demonstrating the effectiveness of black-box attack. Note that DeepDi, as a GNN-based model, is originally not within the scope of PELICAN (i.e., sequence models). However, the hypothesis of models overfitting on compiler-generated syntactic patterns holds, which enables such a transfer attack towards DeepDi. Further investigation shows that, the trigger, i.e., "`85 e6 74`", can be decoded as "`test esi, esp; je XX`" *if followed by*

an arbitrary byte "XX". Note that `test` and `je` constitute a common code pattern of encoding conditional statements, inducing the misbehavior of DeepDi.

BinaryAI. We additionally launch transfer attacks on BinaryAI [11], a binary code clone detection platform, and provide detailed information on these attacks in our supplementary material [51]. We responsibly reported these issues and were informed that BinaryAI has recently implemented a new architecture that is not affected by the reported issue.

8 Related Work

Backdoor Attacks. Backdoor attacks are a prominent security threat to deep learning models. Researchers have demonstrated launching such attacks against image recognition [32–34], natural language processing [35–37], federated learning [109–113], reinforcement learning [114, 115], etc. Backdoors can be injected in those systems through model poisoning [45–47] and neuron hijacking [52], or generated on clean/benign models [30, 31]. PELICAN falls in the last category, where the adversary has only the access to a pre-trained model with a few samples. Existing natural backdoors mainly focus on the computer vision domain. PELICAN targets the binary code analysis task, which is orthogonal.

Adversarial Attacks against Malware Classifiers. Recently, security vulnerabilities in DL-based malware classifiers have drawn considerable attention. Researchers have proposed attacks against malware classifiers, including backdoor attacks via poisoning training dataset [96, 97, 116] and evasion attack by generating adversarial samples [95, 117]. Compared with these techniques, PELICAN explore the backdoor vulnerabilities in naturally trained DL models for binary analysis, another pivotal task in the cyber-security domain. Similar to existing techniques, PELICAN advocates to use learning-based techniques for security tasks with utmost care.

Program Synthesis. Program synthesis generates desired programs according to partial specifications such as user-provided input-output examples known as programming by example (PBE) [118, 119], or complete specifications such as reference programs [120]. Existing approaches utilizes SMT solvers [121], Enumerative search [122], (Counterexample Guided Inductive Synthesis) CEGIS [120], Syntax-Guided Synthesis (SyGuS) [123] or conflict-driven [124]. Different from their goal to synthesize a general program with user-defined functionalities, ours is to synthesize a trojaned program while ensuring the semantic equivalence.

9 Conclusion

We study the security of DL models used in binary code analysis, which have a lot of downstream security applications. Our study shows that these models are quite vulnerable and can be exploited by attackers. Our attack features a trigger generation technique for these models that produces instructions causing the models to misclassify, and a trigger injection technique that can preserve input program semantics.

Acknowledgements

We thank the anonymous reviewers for their constructive comments. This research was supported, in part by IARPA TrojAI W911NF-19-S-0012, NSF 1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] Kexin Pei et al. "XDA: Accurate, Robust Disassembly with Transfer Learning". In: *NDSS*. 2021.

[2] Sheng Yu et al. "DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly". In: *USENIX Security*. 2022.

[3] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. "Recognizing Functions in Binaries with Neural Networks". In: *USENIX Security*. 2015.

[4] Tiffany Bao et al. "BYTEWEIGHT: Learning to Recognize Functions in Binary Code". In: *USENIX Security*. 2014.

[5] Kexin Pei et al. "TREX: Learning Execution Semantics from Micro-Traces for Binary Similarity". In: *preprint arXiv:2012.08680* (2020).

[6] Luca Massarelli et al. "SAFE: Self-Attentive Function Embeddings for Binary Similarity". In: *DIMVA*. 2019.

[7] Luca Massarelli et al. "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis". In: *BAR@NDSS*. 2019.

[8] Fei Zuo et al. "Neural machine translation inspired binary code similarity comparison beyond function pairs". In: *preprint arXiv:1808.04706* (2018).

[9] Xiaochuan Zhang et al. "Similarity Metric Method for Binary Basic Blocks of Cross-Instruction Set Architecture". In: *BAR@NDSS*. 2020.

[10] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization". In: *S&P*. 2019.

[11] Zeping Yu et al. "Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection". In: *AAAI*. 2020.

[12] Kexin Pei et al. "StateFormer: fine-grained type recovery from binaries using generative state modeling". In: *ESEC/SIGSOFT FSE*. 2021.

[13] Alwin Maier et al. "TypeMiner: Recovering Types in Binary Programs Using Machine Learning". In: *DIMVA*. 2019.

[14] Jingxuan He et al. "Debin: Predicting debug information in stripped binaries". In: *CCS*. 2018.

[15] Qibin Chen et al. "Augmenting Decompiler Output with Learned Variable Names and Types". In: *preprint arXiv:2108.06363* (2021).

[16] Zheng Leong Chua et al. "Neural Nets Can Learn Function Type Signatures From Binaries". In: *USENIX Security*. 2017.

[17] Han Gao et al. "A lightweight framework for function name reassignment based on large-scale stripped binaries". In: *ISSTA*. 2021.

[18] Fiorella Artuso et al. "In nomine function: Naming functions in stripped binaries with neural networks". In: *preprint arXiv:1912.07946* (2019).

[19] Jeremy Lacomis et al. "DIRE: A neural approach to decompiled identifier naming". In: *ASE*. 2019.

[20] Cho Do Xuan, Mai Hoang Dao, and Hoa Dinh Nguyen. "APT attack detection based on flow network analysis techniques using deep learning". In: *J. Intell. Fuzzy Syst.* (2020).

[21] Keping Yu et al. "Securing critical infrastructures: Deep-Learning-Based threat detection in IIoT". In: *IEEE Commun. Mag.* (2021).

[22] Nickolaos Koroniotis, Nour Moustafa, and Elena Sitnikova. "Forensics and deep learning mechanisms for botnets in internet of things: A survey of challenges and solutions". In: *IEEE Access* (2019).

[23] Abdulellah Alsaheel et al. "ATLAS: A Sequence-based Learning Approach for Attack Investigation". In: *USENIX Security*. 2021.

[24] Guozhu Wang et al. "A Novel Method for Detecting Advanced Persistent Threat Attack Based on Belief Rule Base". In: *Applied Sciences* (2021).

[25] Wei Wang et al. "HAST-IDS: Learning hierarchical spatial-temporal features using deep neural networks to improve intrusion detection". In: *IEEE Access* (2017).

[26] Chuanlong Yin et al. "A deep learning approach for intrusion detection using recurrent neural networks". In: *IEEE Access* (2017).

[27] Nathan Shone et al. "A deep learning approach to network intrusion detection". In: *IEEE Trans. Emerg. Top. Comput. Intell.* (2018).

[28] Ravi Vinayakumar et al. "Deep learning approach for intelligent intrusion detection system". In: *IEEE Access* (2019).

[29] Zhuo Zhang et al. "OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary". In: *S&P*. 2021.

[30] Shihao Zhao et al. "What Do Deep Nets Learn? Class-wise Patterns Revealed in the Input Space". In: *preprint arXiv:2101.06898* (2021).

[31] Guanhong Tao et al. "Model Orthogonalization: Class Distance Hardening in Neural Networks for Better Security". In: *S&P*. 2022.

[32] Tianyu Gu et al. "BadNets: Evaluating Backdooring Attacks on Deep Neural Networks". In: *IEEE Access* (2019).

[33] Ali Shafahi et al. "Poison frogs! targeted clean-label poisoning attacks on neural networks". In: *NeurIPS*. 2018.

[34] Ahmed Salem et al. "Dynamic Backdoor Attacks Against Machine Learning Models". In: *preprint arXiv:2003.03675* (2020).

[35] Xinyang Zhang, Zheng Zhang, and Ting Wang. "Trojaning Language Models for Fun and Profit". In: *European S&P*. 2021.

[36] Xiaoyi Chen et al. "Badnl: Backdoor attacks against nlp models". In: *preprint arXiv:2006.01043* (2020).

[37] Keita Kurita, Paul Michel, and Graham Neubig. "Weight poisoning attacks on pre-trained models". In: *ACL*. 2020.

[38] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and Harnessing Adversarial Examples". In: *ICLR*. 2015.

[39] Wieland Brendel, Jonas Rauber, and Matthias Bethge. "Decision-based adversarial attacks: Reliable attacks against black-box machine learning models". In: *ICLR*. 2018.

[40] Nicholas Carlini and David Wagner. "Towards evaluating the robustness of neural networks". In: *S&P*. 2017.

[41] Nicholas Carlini and David Wagner. "Adversarial examples are not easily detected: Bypassing ten detection methods". In: *AISec@CCS*. 2017.

[42] Sen Chen et al. "Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach". In: *Comput. Secur.* (2018).

[43] Chaowei Xiao et al. "Spatially transformed adversarial examples". In: *ICLR*. 2018.

[44] *How AI can help analyze malware*. https://www.theregister.com/2022/03/26/machine_learning_malware/.

[45] Yunfei Liu et al. "Reflection Backdoor: A Natural Backdoor Attack on Deep Neural Networks". In: *ECCV*. 2020.

[46] Junyu Lin et al. "Composite Backdoor Attack for Deep Neural Network by Mixing Existing Benign Features". In: *CCS*. 2020.

[47] Eugene Bagdasaryan and Vitaly Shmatikov. "Blind Backdoors in Deep Learning Models". In: *preprint arXiv:2005.03823* (2020).

[48] J Li et al. "TextBugger: Generating Adversarial Text Against Real-world Applications". In: *NDSS*. 2019.

[49] Zhitao Gong et al. "Adversarial texts with gradient methods". In: *preprint arXiv:1801.07175* (2018).

[50] Pascal Junod et al. "Obfuscator-LLVM–software protection for the masses". In: *SPRO@ICSE*. IEEE. 2015.

[51] *Pelican*. https://github.com/ZhangZhuoSJTU/Pelican.

[52] Yingqi Liu et al. "Trojaning Attack on Neural Networks". In: *NDSS*. 2018.

[53] *IDA*. https://www.hex-rays.com/products/ida/.

[54] *Ghidra*. https://ghidra-sre.org/.

[55] *Binary Ninja*. https://binary.ninja/.

[56] Vasisht Duddu et al. "Stealing neural networks via timing side channels". In: *preprint arXiv:1812.11720* (2018).

[57] Xing Hu et al. "Deepsniffer: A dnn model extraction framework based on learning architectural hints". In: *ASPLOS*. 2020.

[58] Weizhe Hua, Zhiru Zhang, and G Edward Suh. "Reverse engineering convolutional neural networks through side-channel information leaks". In: *DAC*. IEEE. 2018.

[59] Yun Xiang et al. "Open dnn box by power side-channel attack". In: *IEEE Trans. Circuits Syst.* (2020).

[60] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. "Cache telepathy: Leveraging shared resource attacks to learn DNN architectures". In: *USENIX Security*. 2020.

[61] Yuankun Zhu et al. "Hermes Attack: Steal DNN Models with Lossless Inference Accuracy". In: *USENIX Security*. 2021.

[62] Nicolas Papernot et al. "Practical black-box attacks against machine learning". In: *AsiaCCS*. 2017.

[63] Yanpei Liu et al. "Delving into transferable adversarial examples and black-box attacks". In: *preprint arXiv:1611.02770* (2016).

[64] Andrew Ilyas et al. "Black-box adversarial attacks with limited queries and information". In: *ICML*. 2018.

[65] Seyed-Mohsen Moosavi-Dezfooli et al. "Universal adversarial perturbations". In: *CVPR*. 2017.

[66] *Mirai*. https://github.com/jgamblin/Mirai-Source-Code.

[67] Yingqi Liu et al. "EX-RAY: Distinguishing Injected Backdoor from Natural Features in Neural Networks by Examining Differential Feature Symmetry". In: *preprint arXiv:2103.08820* (2021).

[68] Chuan Guo et al. "Gradient-based Adversarial Attacks against Text Transformers". In: *preprint arXiv:2104.13733* (2021).

[69] *X86_calling_conventions*. https://en.wikipedia.org/wiki/X86_calling_conventions.

[70] Ramtine Tofighi-Shirazi et al. "Defeating Opaque Predicates Statically through Machine Learning and Binary Analysis". In: *SPRO@CCS*. 2019.

[71] Jiang Ming et al. "Loop: Logic-oriented opaque predicate detection in obfuscated binary code". In: *CCS*. 2015.

[72] R Krishna Ram Prakash, PP Amritha, and M Sethumadhavan. "Opaque predicate detection by static analysis of binary executables". In: *SSCC*. 2017.

[73] Thomas Rinsma. *Seeing through obfuscation: interactive detection and removal of opaque predicates*. 2017.

[74] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. "Detecting self-mutating malware using control-flow graph matching". In: *DIMVA*. 2006.

[75] Mila Dalla Preda et al. "A semantics-based approach to malware detection". In: *POPL*. 2007.

[76] Mila Dalla Preda et al. "Opaque predicates detection by abstract interpretation". In: *AMAST*. 2006.

[77] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *TACAS*. 2008.

[78] Antonio Flores-Montoya and Eric Schulte. "Datalog disassembly". In: *USENIX Security*. 2020.

[79] *SPEC CPU2000*. https://www.spec.org/cpu2000/.

[80] Joseph P. Turian, Lev-Arie Ratinov, and Yoshua Bengio. "Word Representations: A Simple and General Method for Semi-Supervised Learning". In: *ACL*. 2010.

[81] Matthew E Peters et al. "Semi-supervised sequence tagging with bidirectional language models". In: *ACL*. 2017.

[82] Matt Gardner et al. "AllenNLP: A Deep Semantic Natural Language Processing Platform". In: *ACL*. 2018.

[83] *fairseq*. https://github.com/facebookresearch/fairseq.

[84] Xudong Pan et al. "Hidden Trigger Backdoor Attack on {NLP} Models via Linguistic Style Manipulation". In: *USENIX Security*. 2022.

[85] Wenkai Yang et al. "Rethinking Stealthiness of Backdoor Attack against NLP Models". In: *ACL/IJCNLP*. 2021.

[86] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*. Vol. 185. IOS press, 2009.

[87] Pascal Fontaine and Hans-Jörg Schurr. "Quantifier Simplification by Unification in SMT". In: *FroCoS*. 2021.

[88] Aaron R Bradley, Zohar Manna, and Henny B Sipma. "What's decidable about arrays?" In: *VMCAI*. 2006.

[89] David Trabish and Noam Rinetzky. "Relocatable addressing model for symbolic execution". In: *ISSTA*. 2020.

[90] Daniil Kuts. "Towards symbolic pointers reasoning in dynamic symbolic execution". In: *IVMEM*. 2021.

[91] Yuan Zang et al. "Word-level textual adversarial attacking as combinatorial optimization". In: *preprint arXiv:1910.12196* (2019).

[92] Xinshuai Dong et al. "Towards robustness against natural language word substitutions". In: *preprint arXiv:2107.13541* (2021).

[93] Xuezixiang Li, Yu Qu, and Heng Yin. "PalmTree: Learning an Assembly Language Model for Instruction Embedding". In: *CCS*. 2021.

[94] Fabio Pierazzi et al. "Intriguing properties of adversarial ml attacks in the problem space". In: *SP*. 2020.

[95] Keane Lucas et al. "Malware Makeover: breaking ML-based static analysis by modifying executable bytes". In: *AsiaCCS*. 2021.

[96] Limin Yang et al. "Jigsaw Puzzle: Selective Backdoor Attack to Subvert Malware Classifiers". In: *preprint arXiv:2202.05470* (2022).

[97] Giorgio Severi et al. "{Explanation-Guided} Backdoor Poisoning Attacks Against Malware Classifiers". In: *USENIX Security*. 2021.

[98] Zeliang Kan et al. "Automated deobfuscation of Android native binary code". In: *preprint arXiv:1907.06828* (2019).

[99] Luigi Coniglio. "Combining program synthesis and symbolic execution to deobfuscate binary code". MA thesis. University of Twente, 2019.

[100] Yoann Guillot and Alexandre Gazet. "Automatic binary deobfuscation". In: *J. Comput. Virol.* (2010).

[101] Chani Jindal et al. "Neurlux: dynamic malware analysis without feature engineering". In: *ACSAC*. 2019.

[102] Sifan Wu and Xi Xiao. "ConvDroid: Lightweight Neural Network based Andoird Malware Detection." In: *Aust. J. Intell. Inf. Process. Syst.* (2019).

[103] Yu-Jye Tung and Ian G Harris. "A Heuristic Approach to Detect Opaque Predicates that Disrupt Static Disassembly". In: *BAR@NDSS*. 2020.

[104] Hyungjoon Koo and Michalis Polychronakis. "Juggling the Gadgets: Binary-level Code Randomization using Instruction Displacement". In: *AsiaCCS*. 2016.

[105] *SPEC CPU2006*. https://www.spec.org/cpu2006/.

[106] *Binutils*. https://www.gnu.org/software/binutils/.

[107] *Coreutils*. https://www.gnu.org/software/coreutils/.

[108] *Deepbits Technology*. https://www.deepbits.com/.

[109] Chulin Xie et al. "DBA: Distributed Backdoor Attacks against Federated Learning". In: *ICLR*. 2019.

[110] Hongyi Wang et al. "Attack of the tails: Yes, you really can backdoor federated learning". In: *NeurIPS* (2020).

[111] Vale Tolpegin et al. "Data Poisoning Attacks Against Federated Learning Systems". In: *ESORICS*. 2020.

[112] Eugene Bagdasaryan et al. "How to backdoor federated learning". In: *AISTATS*. 2020.

[113] Minghong Fang et al. "Local model poisoning attacks to Byzantine-robust federated learning". In: *USENIX Security*. 2020.

[114] Panagiota Kiourti et al. "TrojDRL: evaluation of backdoor attacks on deep reinforcement learning". In: *DAC*. 2020.

[115] Lun Wang et al. "BACKDOORL: Backdoor Attack against Competitive Reinforcement Learning". In: *IJCAI*. 2021.

[116] Chaoran Li et al. "Backdoor attack on machine learning based android malware detectors". In: *IEEE Trans. Dependable Secur. Comput.* (2021).

[117] Felix Kreuk et al. "Deceiving end-to-end deep learning malware detectors using adversarial examples". In: *preprint arXiv:1802.04528* (2018).

[118] Daniel W. Barowy et al. "FlashRelate: extracting relational data from semi-structured spreadsheets using examples". In: *PLDI*. 2015.

[119] Shengwei An et al. "Augmented example-based synthesis using relational perturbation properties". In: (2020).

[120] Armando Solar-Lezama et al. "Combinatorial sketching for finite programs". In: *ASPLOS*. 2006.

[121] Andrew Reynolds et al. "Counterexample-guided quantifier instantiation for synthesis in SMT". In: *CAV*. 2015.

[122] Woosuk Lee et al. "Accelerating search-based program synthesis using learned probabilistic models". In: (2018).

[123] Rajeev Alur et al. "Syntax-guided synthesis". In: *FMCAD*. 2013.

[124] Yu Feng et al. "Program synthesis using conflict-driven learning". In: (2018).