Black-box Attacks Against Neural Binary Function Detection

Joshua Bundt[*]
Northeastern University
Army Cyber Institute

Michael Davinroy[*]
Northeastern University

Ioannis Agadakos[†]
Northeastern University
Amazon

Alina Oprea
Northeastern University

William Robertson
Northeastern University

ABSTRACT

Binary analyses based on deep neural networks (DNNs), or *neural binary analyses* (NBAs), have become a hotly researched topic in recent years. DNNs have been wildly successful at pushing the performance and accuracy envelopes in the natural language and image processing domains. Thus, DNNs are highly promising for solving binary analysis problems that are hard due to a lack of complete information resulting from the lossy compilation process. Despite this promise, it is unclear that the prevailing strategy of repurposing embeddings and model architectures originally developed for other problem domains is sound given the adversarial contexts under which binary analysis often operates.

In this paper, we empirically demonstrate that the current state of the art in neural function boundary detection is vulnerable to both inadvertent and deliberate adversarial attacks. We proceed from the insight that current generation NBAs are built upon embeddings and model architectures intended to solve syntactic problems. We devise a simple, reproducible, and scalable black-box methodology for exploring the space of inadvertent attacks – instruction sequences that could be emitted by common compiler toolchains and configurations – that exploits this syntactic design focus. We then show that these inadvertent misclassifications can be exploited by an attacker, serving as the basis for a highly effective black-box adversarial example generation process. We evaluate this methodology against two state-of-the-art neural function boundary detectors: XDA and DeepDi. We conclude with an analysis of the evaluation data and recommendations for how future research might avoid succumbing to similar attacks.

CCS CONCEPTS

• **Security and privacy** → **Software reverse engineering**; *Software security engineering*.

KEYWORDS

binary analysis, disassembly, deep neural network, function boundary detection

ACM Reference Format:
Joshua Bundt, Michael Davinroy, Ioannis Agadakos, Alina Oprea, and William Robertson. 2023. Black-box Attacks Against Neural Binary Function Detection. In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, China.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3607199.3607200

1 INTRODUCTION

Binary analysis, or techniques for extracting and inferring information from code compiled to native instruction set architectures (ISAs), is an important set of capabilities and research area in modern security. The field encompasses a number of distinct topics such as disassembly [5, 10, 48, 52, 54, 76], function boundary detection [6, 9, 17, 54, 64, 76], static similarity detection [22–24, 32, 41, 42, 46, 72, 77, 78], type recovery [40], and full decompilation [25, 62, 73]. Each of these capabilities is in turn crucial for downstream security tasks such as malware analysis [3, 33, 57, 70] and software hardening via control-flow-integrity (CFI) enforcement, artificial diversification, or debloating when source code is not available.

Instantiations of these capabilities in the form of deep neural networks (DNNs) have generated substantial interest in recent years. Neural binary analyses (NBAs) are seemingly well-matched to the problem domain, where inference is necessary due to the lossy compilation process. Recent work has shown great promise for performing accurate disassembly [54, 76], function boundary detection [17, 54, 64, 76], and static binary similarity detection [22–24, 41, 42, 46, 72, 77, 78] that is simultaneously more efficient than deterministic methods.

Despite this promise, questions remain as to how resilient NBAs are in practice when confronted with the incredible diversity of binary code found in the wild as well as motivated adversaries seeking to actively evade or confuse detection techniques that make use of binary analysis. Adversarial attacks against DNNs have been intensely investigated in other problem domains [13, 53], but most of these have been developed for continuous domains (e.g., images) whereas NBAs operate in a discrete domain. Furthermore, due to the issue of problem space mapping [55] one must develop specific black-box attacks against NBAs.

Recent work has criticized the size and scope of data used to train and evaluate NBA techniques published to date. For instance, Kim et al. [36] studied NBAs that perform static similarity detection using a large dataset of programs compiled with a variety of toolchains and compiler options called BinKit. Using this dataset and a simple baseline similarity detector called TikNib, they show that NBAs do not necessarily outperform simpler, explainable methods such as the one implemented by TikNib. Marcelli et al. [45] performed a similar study also focused on static similarity detection NBAs,

[*] Equal contribution.
[†] Work done while at Northeastern University.

RAID '23, October 16–18, 2023, Hong Kong, China
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0765-0/23/10.
https://doi.org/10.1145/3607199.3607200

and show that published results do not necessarily hold when the systems-under-test are trained and evaluated on larger, more representative datasets. Other recent work has demonstrated that DNNs used for static malware detection on binary programs are prone to adversarial attacks [43], though this work relies on traditional adversarial ML techniques to either use white-box gradient descent or black-box hill climbing to find evading transformations.

In this paper, we consider the heretofore unexplored question of NBA attack resilience in the context of function boundary detection. We focus on two exemplars of the state of the art occupying two representative points in the design space: XDA [54], which directly applies the well-known Transformer model architecture [68] and is intended to be robust to compiler optimization level [54, §4], and DeepDi [76], which employs a relational graph convolutional network and is explicitly advertised as intended for binary analysis in adversarial contexts such as malware analysis – e.g., as part of a malware analysis pipeline after dynamic analysis has been used to unpack a sample.[1]

Observing that current systems are largely based on DNN components developed to solve syntactic problems from other domains, we conjecture that these systems can be evaded using syntactic mutation. Building on this insight, we define a simple, reproducible black-box methodology to identify misclassifying inputs to these state-of-the-art function boundary detection NBAs at scale. Then, we demonstrate how an attacker can systematically leverage these misclassifications to either evade function detection or overwhelm a downstream analysis with false detections via at-will injection of false negatives and false positives.

From the techniques we developed, our analysis of the data leads us to several conclusions.

1. Sophisticated searches for adversarial examples using gradient descent are not required to significantly degrade the accuracy of NBA-based function boundary detection systems.
2. Function boundary detection systems that build on embeddings and model architectures intended for solving syntactic problems should be viewed in a similar light as syntactic approaches for attack detection such as first-generation antivirus and signature-based intrusion detection – that is, with healthy skepticism. This likely holds for other binary analysis tasks as well.
3. It is critical that future work is evaluated on large, representative, and openly available datasets that include a range of compiler configurations as well as adversarial examples; building on existing foundations [36, 45] or this work would be a good starting point. Otherwise, it is difficult to extrapolate published evaluation results to actual operational performance.

We note that despite these conclusions, we do not intend to completely dismiss the promise of neural binary analyses. We discuss potential avenues for future research to mitigate the attacks found using our methodology in §6.

In summary, the contributions of this paper are the following.

1. We propose a simple, reproducible black-box methodology for evaluating the resilience of function boundary detection NBAs to attacks at scale.
2. We demonstrate the susceptibility of the current state of the art, represented by XDA [54] and DeepDi [76], to producing overwhelming false negatives and false positives to downstream binary analyses.
3. We discuss and synthesize conclusions from an analysis of the evaluation data, and suggest several paths forward to mitigate similar attacks against neural binary analysis.

The source code and datasets are available at https://osf.io/bcdxq/.

2 PROBLEM STATEMENT AND MOTIVATION

2.1 Binary Analysis

The term "binary analysis" encompasses a wide range of techniques that all attempt to extract information from programs that have been compiled to a native instruction set architecture (ISA). These techniques range from fundamental analyses such as disassembly [52, 54, 76] and function boundary detection [9, 17, 54, 76] to downstream tasks that build on prior analyses such as static similarity detection [24, 32, 42, 77, 78], type recovery [40], malware detection [3, 33, 57, 70], and full decompilation [25, 62, 73]. Designing accurate and efficient binary analyses is substantially more difficult than for source code due to the inherently lossy compilation process. That is, compiler toolchains discard much of the higher-level abstractions present in source code when lowering to an ISA. Thus, binary analyses must operate with incomplete information and are virtually always unsound. Compounding this difficulty is that binary analyses are often, though not always, performed under a strong threat model [59] in which active adversaries attempt to evade or otherwise confuse those analyses.

While binary analyses have traditionally employed deterministic methods, the lack of source code naturally suggests inference methods as a promising approach for improving both accuracy and performance. In that light, it should come as no surprise that deep neural networks (DNNs) have come to the fore as a basis for binary analysis research. Table 1 presents an overview of recent work in this vein, to which we refer hereinafter as *neural binary analyses* or *NBAs*.

Each entry in Table 1 lists the input, embedding, model architecture, and the binary analyses implemented. An embedding is simply a procedure for mapping input data to a representation on which that model performs training and inference. Common choices of embeddings are one-hot encoding of byte sequences, or text embeddings such as word2vec [47] applied to the token stream produced by a disassembler. The model architecture, on the other hand, is the neural network proper; that is, the set of layers, interconnections, and weights responsible for inference. It is common for NBAs to repurpose model architectures developed for natural language or image analysis tasks; examples include recurrent neural networks (RNNs), convolutional neural networks (CNNs), and the Transformer architecture [68].

Motivation. Recent prior work has studied the accuracy of NBAs for static similarity detection [36, 45] and malware detection [43]. However, to the best of our knowledge, the question of

[1] In their paper, the authors "demonstrate how DeepDi is used in malware classification" [76, p. 2] by evaluating their prototype on data from the Microsoft Malware Classification Challenge, comparing against MalConv [58].

System	Input	Embedding	Architecture	Capabilities
BiRNN [64]	Bytes	One-hot encoding	RNN	F
MtNet [33]	API calls, memory objects	Bit vector	Feedforward	M
Eklayva [17]	Disassembly text	word2vec [47]	RNN	F
Gemini [72]	ACFG	structure2vec [19]	Siamese	S
Sleipnir [3]	Windows API invocations	Bit vector	Feedforward	M
SAFE [46]	Disassembly text	word2vec [47]	BiRNN, Siamese	S
asm2vec [22]	Disassembly text	PV-DM [39]	LSTM	S
Coda [25]	Disassembly text	AST	Tree-LSTM [66]	R
InnerEye [78]	Disassembly text	word2vec [47]	LSTM	S
Instruction2Vec [41]	Disassembly text	word2vec [47]	CNN	S
Order Matters [77]	Disassembly text	BERT [21], CNN	MPNN [26], CNN	S
DeepVSA [31]	One-hot encoding	Context vectors	LSTM	V
DeepImgMalDetect [70]	Pixels	—	RNN	M
DeepBinDiff [23]	Disassembly text	word2vec [47]	ANN	S
XDA [54]	Bytes	One-hot encoding	Transformer [68]	D, F
PalmTree [42]	Disassembly text	BERT [21]	—	F, S, V
Codee [74]	Disassembly text	word2vec [47], node2vec [28]	—	S
DeepDi [76]	Instruction metadata	RNN	Relational-GCN [61]	D, F

Table 1: Summary comparison of various neural binary analysis systems. Note that all of these systems are at least in part built on embeddings and model architectures developed to solved problems in the NLP or image processing domains. *Capabilities:* D = disassembly, F = function identification, V = value set analysis, S = similarity, R = decompilation, M = malware detection or classification.

whether NBAs are a suitable solution for function boundary detection has not been definitively studied. This paper attempts to answer this question, and thus focuses specifically on prominent systems targeting the function boundary detection task.

2.2 Function Boundary Detection

Function boundary detection is a fundamental binary analysis that typically occurs directly after, or even in tandem with, disassembly [52]. Identifying functions is crucial for many downstream tasks. For instance, most static similarity algorithms consider pairs of functions when computing distances. Functions are also important inputs to recursive descent disassemblers as starting points for recursive disassembly or as possible callees of indirect call sites.

If function detection is performed as part of a manual process – e.g., interactive reverse engineering provided by tools like IDA Pro [29], Ghidra [50], or Binary Ninja [69] – excessive false positives could lead to user fatigue and, in turn, an unusable tool [8]. False negatives, on the other hand, are perhaps even more concerning since failing to identify functions could directly lead to evasion opportunities for attackers that aspire to elude detection.

Function boundary misclassifications can also have a large impact on the accuracy and utility of an automated analysis pipeline. For instance, it is common to combine successive rounds of static and dynamic analysis to, e.g., first unpack a malware sample in a sandbox so that an efficient static analysis can be performed on an unobfuscated dropped or in-memory binary [75]. False negative function detections in this scenario could again lead to detection “blind spots,” while false positives could degrade the efficiency or accuracy of downstream analyses.

More formally, we can think of a function boundary detection NBA as a procedure that learns a mapping from bytes or instructions in a binary, depending on the embedding, to one of three labels: S for function entry points, E for function exit points, and N for all other points. Let B be the set of binary inputs and $\mathbb{N}$ be the set of possible byte or instruction indices in each binary. We can then denote this mapping as

$$F : B \times \mathbb{N} \mapsto L = \{\mathsf{S}, \mathsf{E}, \mathsf{N}\} . \tag{1}$$

Early work in the NBA space heavily borrowed from DNNs built to tackle natural language processing (NLP) problems. The first system to adopt this approach was BiRNN [64], which treated byte sequences comprising binaries as tokens in a language. BiRNN converts each input byte into $\mathbb{R}^{256}$ vectors using a one-hot encoding, where a byte’s value is indicated by the position of the single 1 in a vector. Encoded bytes are then fed to a bi-directional RNN, where the use of two RNNs allows for prediction of a byte label using both preceding and succeeding bytes as context.

XDA [54] built upon BiRNN’s approach to function boundary detection by adapting another powerful model architecture from the NLP literature: Transformer [68]. Transformer pioneered the concept of *self-attention*, where an attention layer allows the model to process sequential data out of order. This allows Transformer-based models to flexibly learn and infer meaning from context as well as parallelize better than prior architectures like RNN, LSTM, and GRU. Transformer-based models such as BERT [21] (Bidirectional Encoder Representations from Transformers) and the GPT family [14] (Generative Pre-Trained Transformer) represent the state of the art in NLP model architectures.

XDA’s implementation [35] directly applies a popular implementation of BERT called RoBERTa (provided by Facebook’s Fairseq [51] library) to the binary disassembly and function boundary detection tasks. Binaries are processed in 512-byte chunks, and a one-hot encoding is used to produce $\mathbb{R}^{256}$ vectors to be processed by the network. In addition to byte values, the input vocabulary defines five additional tokens representing *padding*, *start-of-sequence*, *end-of-sequence*, *unknown*, and *mask* (not all are used by XDA). In the first of two phases, the model is pre-trained using masked language modeling (MLM), which essentially teaches the model to predict byte values given surrounding context. The resulting model is then fine-tuned in the second phase to transfer the knowledge learned in the first phase to a particular binary analysis task such as function boundary detection.

DeepDi [76] is a state-of-the-art example of an NBA-based disassembly and function boundary detection system.[2] While DeepDi follows in the tradition of BiRNN and XDA by building upon existing model architectures, in this case, R-GCN [61] (Relational Graph Convolutional Model), it improves on prior work in several ways. First, it eschews the use of deep learning altogether for the initial disassembly step, choosing instead to rely on superset disassembly [10] to recover all possible instructions contained in an input binary. The instruction superset, in the form of 4-tuples of ⟨opcode, mod_rm, scale_index, rex_prefix⟩, is then converted into a fixed-dimension embedding using a learned embedding layer. Each embedding is concatenated with the following two instruction embeddings which is fed to an RNN to arrive at a final instruction representation. These representations then serve as input to the R-GCN, which models various relationships between instructions using an Instruction Flow Graph (IFG) in order to weed out invalid instructions from the superset and retain only the "true" disassembly.

To identify function entry points, DeepDi first collects a set of candidate entry points by applying a set of heuristics to instructions identified as valid from the superset. Each candidate instruction is packed with the three preceding and three succeeding instructions and then fed to the entry point recovery model. This model consists of an embedding layer, a GRU layer, and a two-layer perceptron classifier. The authors of DeepDi note that while the model achieved an average F1 score of 98.6% on the function start detection task in their evaluation, their heuristics-based approach "will miss tail jumps and functions with unseen prologues [76, p. 7]."

2.3 Semantics, or Merely Syntax?

While systems like XDA make repeated reference to "learning semantics," these representations do not encode the semantic outcome of the input when executed on a system. We conjecture this limitation is due to the approach of being trained using only disassembled instructions or raw sequences of bytes extracted from binaries, as correspondence is limited to patterns of bytes or textual tokens presented during training. Absent of semantic meaning, code isomorphisms that syntactically appear drastically different might well not be detected as semantically equivalent.

```c
int add(int a, int b) {
    if (a == 0)
        return b;
    else if (b == 0)
        return a;
    return a + b;
}
```

```asm
add_O0:
    push  rbp                   ; save caller fp
    mov   rbp, rsp              ; set fp
    mov   dword [rbp-0x4], edi  ; get arg1
    mov   dword [rbp-0x8], esi  ; get arg2
    cmp   dword [rbp-0x4], 0x0  ; compare arg1 to 0
    jne   .check_arg2
    mov   eax, dword [rbp-0x8]  ; return arg2
    jmp   .return
.check_arg2:
    cmp   dword [rbp-0x8], 0x0  ; compare arg2 to 0
    jne   .do_add
    mov   eax, dword [rbp-0x4]  ; return arg1
    jmp   .return
.do_add:
    mov   edx, dword [rbp-0x4]
    mov   eax, dword [rbp-0x8]
    add   eax, edx              ; return arg1 + arg2
.return:
    pop   rbp                   ; restore caller fp
    ret                         ; return to caller
```

```asm
add_O3:
    lea    eax, [rdi+rsi*1]     ; return arg1 + arg2
    ret                         ; return to caller
```

Listing 1: Compiler optimizations can have a drastic effect on program representation in compiled code. Relying only on sequences of bytes or textual tokens absent of semantic information limits detection only to known syntactic patterns.

To illustrate, Listing 1 presents a naïve addition function and its compilation to x86_64 assembly using two commonly available optimization levels: O0 and O3. The resulting code, while semantically equivalent, has radically different syntactic forms, and systems relying only on detecting sequences of bytes or instructions would fail to identify the optimized version if they have not encountered a similar example during training. While an argument can be made for generating comprehensive datasets that contain both versions (and indeed virtually all current methods do try to include these common compiler optimizations), we argue that such an approach cannot scale to include every possible combination of all available compiler flags. In essence, this places a hard constraint on what is possible for NBA models to learn in the absence of semantic information.

In the remainder of this paper, we build upon this insight to demonstrate systematic attacks against neural binary analyses for function boundary detection.

3 ATTACKING NEURAL FUNCTION BOUNDARY DETECTION

In our evaluation of neural function boundary detection, we focus on *black-box attacks*. These attacks are so-named since no information about the model-under-test (MUT) such as its internal weights or structure are assumed. Black-box attacks are advantageous because they do not require a deep understanding of MUTs; instead, only the ability to issue queries and observe results is needed. However, as the search process is unguided by model information, black-box techniques can fail to discover latent vulnerabilities that a white-box adversarial search such as projected gradient descent (PGD) [44] might otherwise uncover. In this sense, the results of our methodology should be considered a lower bound on the vulnerability of MUTs to which it is applied.

[2] DeepDi only recovers function entry points, and so is more accurately called a function start detection system.

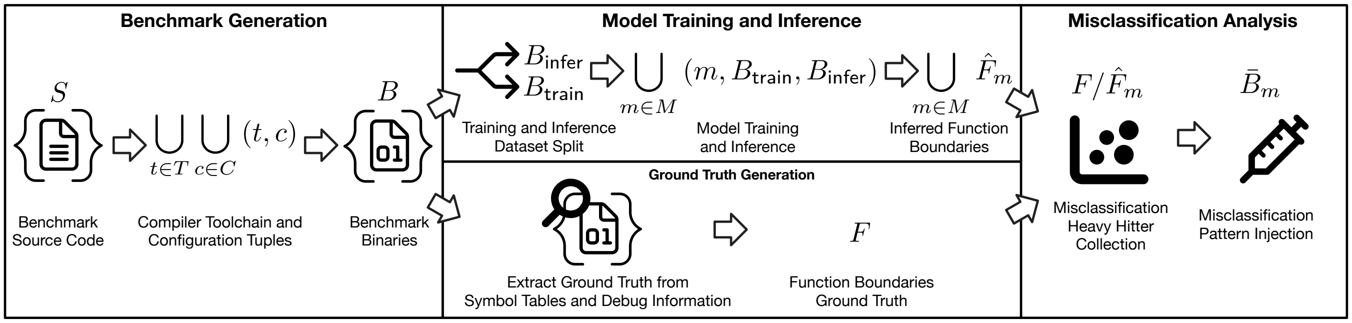
Benchmark Generation
Model Training and Inference
Misclassification Analysis
S
B
$\bigcup_{t\in T}\bigcup_{c\in C}(t,c)$
Benchmark Source Code
Compiler Toolchain and Configuration Tuples
Benchmark Binaries
B_{infer}
B_{train}
$\bigcup_{m\in M}(m, B_{\text{train}}, B_{\text{infer}})$
$\bigcup_{m\in M}\hat{F}_m$
Training and Inference Dataset Split
Model Training and Inference
Inferred Function Boundaries
Ground Truth Generation
Extract Ground Truth from Symbol Tables and Debug Information
F
Function Boundaries Ground Truth
$F/\hat{F}_m$
$\bar{B}_m$
Misclassification Heavy Hitter Collection
Misclassification Pattern Injection

Figure 1: Overview of the NBA function boundary detection vulnerability search procedure. In the first phase, benchmark source code S is compiled by an array of compiler toolchains and configurations $\langle T, C\rangle$ resulting in a benchmark binary corpus B. Function boundary ground truth F is extracted from B. In parallel, the set of models-under-test M is trained and evaluated on one or more training and inference splits of B. Finally, misclassifications in the form of false positives and negatives are collected by comparing $F, \bigcup_{m\in M}\hat{F}_m$. Heavy hitters are identified and injected into B for attack evaluation.

We define a general black-box vulnerability search procedure with the goal of uncovering and exploiting function boundary misclassifications when performing inference on binary programs. The search proceeds in several phases: (i) input generation, (ii) ground truth generation, (iii) training and inference, and (iv) misclassification analysis.

Input Generation. In the first phase, we gather a corpus of benchmark program source code S. Each benchmark is compiled by an array of compiler toolchains T, each of which is equipped with an attack configuration C consisting of a set of compiler flags and code transformations. Given $n = |T|$ compilers, we obtain a benchmark binary corpus B consisting of n separate compilations of S with each toolchain and attack configuration tuple $\langle T, C\rangle$.

Ground Truth Generation. Each configuration ensures that debugging information is generated while simultaneously preventing compiled binaries from being stripped of symbol information. Thus, we can post-process each binary and use these sources of information to construct a ground truth mapping F (1); that is, a function that labels each byte of code in each binary as to whether it is a function start, a function end, or neither.

MUT Training and Inference. In parallel, we split B into training and inference sets. The MUTs M are individually trained and evaluated on these sets. The result for each MUT $m \in M$ is an *inferred* mapping $\hat{F}_m$. Since we extracted a ground truth labeling F in the previous phase, we can directly compare $\hat{F}_m$ and F to identify misclassifications in the form of false positives E^+_m and false negatives E^-_m where

$$E^+_m = \{b, i\} \;\text{ s.t. } \forall\, b \in B, i \in |b| \;\; F(b,i) = \mathsf{N} \wedge \hat{F}_m(b,i) \in \{\mathsf{S}, \mathsf{E}\} \tag{2}$$

$$E^-_m = \{b, i\} \;\text{ s.t. } \forall\, b \in B, i \in |b| \;\; F(b,i) \in \{\mathsf{S}, \mathsf{E}\} \wedge \hat{F}_m(b,i) = \mathsf{N} \tag{3}$$

Misclassification Analysis. In the last phase, for each MUT we process its misclassification sets E^+_m, E^-_m to identify *attack inputs* A_m that can reliably produce function boundary misclassifications in arbitrary binary programs. To do so, we rank-order misclassifications for each model by highest incidence to lowest. The ranked attack inputs then serve as seeds for an adversarial search, where they are each injected in turn into targeted functions of B to produce a mutated corpus $\bar{B}_m$. A separate attack validation round is then carried out by having the MUT m perform inference on $\bar{B}_m$ to confirm that the intended misclassifications are replicated in the targeted functions.

3.1 Attack Techniques and Threat Models

The vulnerability search procedure relies upon collecting a set of attack techniques in the form of compiler flags and code transformations. Each of these techniques specifically targets function *prologues* and *epilogues*. A function prologue is responsible for (i) saving the contents of any registers that it uses and that a caller is responsible for preserving under a given calling convention; and, (ii) allocating space on the current thread's stack for any local variables the function uses. An epilogue, on the other hand, is responsible for reversing the effects of the prologue as well as (optionally) returning a value to the caller. Our attack techniques modify function prologues and epilogues because neural function boundary detection models focus on bytes or instructions that comprise (or are adjacent to) these code regions.

However, while each attack technique targets the same code regions, not all techniques are created equally. Some attacks are *inadvertent*; that is, an unintended deficiency of the data representation, network architecture, or training set causes the model to misclassify a benign input during inference. Other attacks, however, are inherently *adversarial*. In this case, an adversary intentionally transforms their input to actively attack the MUT. In this case, the only restriction is that the transformation must preserve the intended functionality of the attack.

Inadvertent Threat Model. Inadvertent attacks represent a weak threat model, in that there is no active adversary and, instead, the MUT or training set is suboptimal with respect to a "naturally-occurring" binary that has been emitted by a standard compiler toolchain on benign code. While this threat model is weaker, Ren et al. show that in practice "adversaries explore non-default compiler settings to amplify malware differences" [59].

Adversarial Threat Model. In this threat model, no assumptions are made about how the binary was produced. Binaries can be obfuscated or encrypted, and in such cases must be unpacked in a malware sandbox prior to use of a static analysis on an unobfuscated dropped or in-memory executable image, as is common industry practice [75].

We classify the attack techniques employed by the search procedure according to whether they are inadvertent or adversarial. The criterion we use for this classification is whether or not the code resulting from applying a technique can be emitted by an unmodified compiler toolchain given a legal configuration.

An overview of the vulnerability search procedure is shown in Figure 1. In the following, we describe each inadvertent and adversarial attack technique.

3.2 Inadvertent Attacks

Inadvertent attacks result from misclassified binary code emitted by a benign compiler toolchain under any possible configuration. However, systematically exploring the entire space of possible compiler configurations in terms of combinations of compiler flags is daunting, to put it lightly. To illustrate, clang v13.1.6 (arm64-apple-darwin21.4.0) advertises 1013 distinct command-line options in its default help message when invoked using `clang -help`. Thus, if we denote the set of possible options as C, a rough estimate of number of possible combinations is $|\mathcal{P}(C)| = 2^{1013}$.[3] An exhaustive exploration of this space is clearly intractable in practice, and so we use domain knowledge to select a small number of compiler options that we conjecture will have an effect on function boundary prediction. We describe each of these classes of options below.[4]

Stack Protector. "Stack protector" is gcc and clang's modern name for canary/cookie-based anti-stack smashing defenses [18]. This defense injects an unpredictable guard value onto the stack as part of the function prologue. In an epilogue, the injected copy of the guard is compared to a global copy. If the values do not match, then a stack smashing attack is assumed to have occurred and the program is terminated before the attacker can gain control of execution via, e.g., a corrupted return address. Otherwise, execution continues as normal.

The defense relies upon several assumptions – for instance, that the guard value contains carries sufficient entropy to make guessing infeasible, that the guard value is not leaked to the adversary, that the global copy of the guard cannot be modified by the adversary, and that all stack-allocated data that can be leveraged to hijack code execution is protected by the stack guard. The defense can in fact take several forms depending on the compiler version and particular flags used, such as: whether all functions are protected by a guard or rather only those that allocate a buffer on the stack; whether some or all stack variables are protected by the guard, which can involve variable reordering; and, the offset of the global guard copy.

Our inclusion of these compiler options is based on the observation that stack guard injection and verification requires modifications to function prologues and epilogues. NBAs that rely on particular byte or instruction sequences comprising prologues and epilogues for function boundary detection might thus be confused by these modifications. Listing 4 (§A) illustrates a typical example.

Stack Clash Protection. Stack clash vulnerabilities arise when an attacker is able to grow either the stack or another memory region such that the two memory segments overlap [56]. While OS kernels such as Linux can inject a guard page to separate the stack from other regions, prior research has shown that guard pages are nevertheless circumventable. Thus, modern compilers include options to enable stack clash mitigations in emitted code. The most popular form of this mitigation centers on breaking large stack allocations into page-sized chunks, and either implicitly or explicitly probing each chunk to ensure that it has not clashed with an existing memory allocation [38].

The impetus for our inclusion of stack clash protector compiler options as an attack technique is the modified allocation pattern for large stack buffers in function prologues and the requirement for explicit probe injection if the compiler deems it necessary. Listing 5 (§A) illustrates one form of these modifications to function prologues (epilogues are not affected in this case).

Control Flow Integrity. Control flow integrity (CFI) is a general software hardening approach based on the principle that code must execute control transfers if and only if those transfers were intended by the programmer [1]. Forward-edge CFI in particular has become a standard feature of production compilers like clang and gcc [67], efficiently protecting indirect calls and jumps through computed pointers of various forms. Architectural support for a weak form of return-edge CFI has also become available in recent x86/x86-64 processor generations in the form of Intel CET [63]. While forward-edge CFI checks such as IFCC [67] are typically inserted at call sites, Intel CET enforcement depends on instrumenting valid indirect branch targets with special instructions (`endbr32`, `endbr64`) as well as ensuring that these instructions do not accidentally appear anywhere else in an executable memory region. Since this instrumentation can result in modified function prologues, we include CFI enforcement options as a separate attack technique. Listing 6 (§A) illustrates function prologue modifications resulting from Intel CET and indirect branch tracking enforcement.

SafeStack. SafeStack is another stack-based buffer overflow defense developed as part of code-pointer integrity (CPI) [37] that relies upon separating stacks into safe and unsafe stacks. Security-relevant data such as return addresses, register spills, and local variables are stored on the safe stack. Accesses to the safe stack are always checked via runtime instrumentation for safety. All other stack-allocated data is stored on the unsafe stack, ensuring that buffer overflows cannot corrupt any safe stack data.

Since the compiler must emit code to manipulate two stacks when enforcing SafeStack, this introduces modifications to both the prologue and epilogue of affected functions. Thus, we include SafeStack as a distinct attack technique. An example of SafeStack prologue and epilogue modifications is shown in Listing 7 (§A).

Function Alignment. Compilers provide a number of options to control the alignment of functions in memory. Aligning functions to particular address boundaries can be advantageous from

[3] This is a loose estimate. It is likely that some combinations are invalid, which would lead to an overcount. However, some options are not boolean flags but rather take a value as an argument, which would lead to an undercount.

[4] We also note that we do not claim these classes as exhaustive. Indeed, we are aware of other compiler configurations and code transformations that would be interesting to explore. Unfortunately, due to time constraints we have not yet fully evaluated them and so elide them here.

```
1 f_original:
2     push rbp                  ; save caller fp
3     mov rbp, rsp              ; set fp
4     ; function body...
5     pop rbp                   ; restore caller fp
6     ret                       ; return to caller
```

```
1  f_injected:
2      jmp .entry                ; jump over attack sequence
3      mov eax, 0x89485590       ; attack sequence as immediate
4  .entry:
5      nop                       ; prologue nop sequence tail
6      push rbp                  ; save caller fp
7      mov rbp, rsp              ; set fp
8      ; function body...
9      pop rbp                   ; restore caller fp
10     ret                       ; return to caller
11     add word [rcx], al        ; adversarial insertion
12     leave                     ; adversarial insertion
13     ret                       ; adversarial insertion
14     nop                       ; epilogue nop sequence tail
```

Listing 2: Adversarial attack sequence injection example using compiler-emitted NOP sequences (additions in green). In the prologue, a relative jump is injected to bypass an instruction containing an attack sequence encoded as an immediate value. In the epilogue, an attack sequence is directly injected verbatim; it will not be executed due to the unconditional return at line 10.

a performance perspective for architectural reasons, and optimal alignment varies depending on the target architecture. On the other hand, compilers can also be instructed to eschew optional alignment constraints in favor of optimizing for size. In this case, functions will be tightly packed and not conform to an alignment scheme.

The reason we include function alignment as an attack technique is two-fold. First, tightly packing functions will remove any interstitial padding between adjacent functions, effectively creating a large change in instruction bytes preceding function prologues and succeeding function epilogues. Second, varying the requested alignment will cause compilers to emit different sequences of padding instructions. This leads to a similar, albeit weaker, change in prologue and epilogue-adjacent instructions. An example of this phenomenon is shown in Listing 8 (§A).

3.3 Adversarial Attacks

In addition to the inadvertent attack techniques we just described, we also separately consider adversarial attack techniques. Consistent with our two-tier threat model introduced in §3.1, adversarial attacks go beyond the inadvertent evasive or false positive-inducing inputs that can be emitted by common compiler toolchains and configurations. Instead, under this stronger threat model an adversary can use arbitrary techniques to craft a binary that will induce misclassifications by a function boundary detection NBA.

The possession of the power to arbitrarily modify binaries does not itself imply the ability to easily discover input byte and instruction sequences that produce misclassifications. However, we find that an unguided search over bounded byte sequences is wholly sufficient to quickly find adversarial inputs that produce significant numbers of false positive or false negative misclassifications in the state of the art.

In particular, we explore the simple technique of injecting arbitrary byte sequences into function epilogues for this purpose. In principle, one could use a binary rewriting framework [71] to perform the injections on arbitrary in a functionality-preserving manner – e.g., that makes the necessary modifications to account for the increased size of the code sections of the mutated binary. However, we take the comparatively simpler approach of recompiling the binary corpus with a compiler configuration that causes NOP sequences of a desired length to be emitted in all epilogues. This renders it straightforward to inject the necessary code to perform attack validation in a length-preserving manner via purely local modifications.

We employ and evaluate two forms of adversarial injection in terms of content: (i) injecting a relative jump over a `mov` instruction that loads a register with the attack sequence as an immediate value, and (ii) injecting the attack sequence as-is into a function epilogue after a return instruction. Due to the unconditional jump or return that prefaces each form of the injected attack sequence, there is no realistic possibility that the attack sequence will be executed in either form. Listing 2 presents an example of this technique in action. We note that while the injected code sequences could perhaps be identified as dead code and removed, the ability to do this reliably degrades quickly as more complicated instruction sequences are injected (up to the level of an opaque predicate). We revisit this point in §6; however, we do not believe it to be straightforward to identify and remove attack sequences injected by a determined adversary.

4 IMPLEMENTATION

The inadvertent attack search is implemented using an augmented version of BinKit [36]. This framework provides scripts to reproducibly build a number of independent compiler toolchains (i.e., several versions of gcc and clang) as well as to download and compile numerous open source software packages using a variety of configurations. We modified BinKit to support more compiler versions and configurations, and discuss the resulting experimental setup and data in §5.

Our adversarial attacks are implemented via a binary rewriting framework [2] that in turn is based upon open source code drawn from pyelftools [11] and Capstone [15]. The framework operates on all ISAs present in BinKit.

We consider the binary rewriting procedure safe since it simply overwrites a number of NOP instructions placed in function epilogues by the compiler using the `-fpatchable-function-entry` option. This preserves the existing binary layout in terms of addressing, and thus all jump and call targets remain valid. Additionally, all injected code is protected by jumps or pre-existing return instructions that guard their execution. Nevertheless, we manually spot-checked the rewriting procedure as well as ran existing test suites on modified binaries when available.

5 EVALUATION

In this section, we present the results of our evaluation of two representative state-of-the-art neural binary analyses for function

Table 2: BinKit corpus.

Dataset	Binaries	Functions	Packages	Compilers	Optimizations
Normal	14,480	4,273,807	53	13	6
SizeOpt	2,115	575,143	51	9	1
NoInline	8,460	2,912,548	51	9	4
PIE	4,500	1,868,470	46	9	4
Obfuscate	4,700	1,351,779	51	4	5
CFI	3,800	1,133,310	52	3	5
ASE18	11,254	1,793,278	6	13	5

boundary detection. Our aim in conducting this evaluation was to answer the following research questions.

(**RQ1**) Are NBAs susceptible to inadvertent attacks?
(**RQ2**) Are NBAs susceptible to adversarial attacks?
(**RQ3**) Can an adversary systematically leverage inadvertent and adversarial attacks?
(**RQ4**) Can inadvertent attacks be mitigated with larger, more representative training sets?
(**RQ5**) Can adversarial attacks be mitigated by including adversarial examples during training?

5.1 Experimental Setup

Table 3: Results per dataset. For each dataset and metric (precision, recall, F1), the maximum value is highlighted green while the minimum value is highlighted red. Large standard deviations (SD) are set in bold.

		Precision	Recall	F1
Dataset	Tool	Mean (SD)	Mean (SD)	Mean (SD)
Normal	IDA	1.000 (0.002)	0.844 **(0.144)**	0.908 (0.090)
	XDA	0.989 (0.019)	0.965 (0.052)	0.976 (0.034)
	DeepDi	0.976 (0.045)	0.932 (0.057)	0.952 (0.040)
SizeOpt	IDA	1.000 (0.001)	0.754 **(0.155)**	0.851 **(0.103)**
	XDA	0.977 (0.020)	0.900 (0.080)	0.935 (0.050)
	DeepDi	0.987 (0.032)	0.870 (0.061)	0.923 (0.040)
NoInline	IDA	1.000 (0.002)	0.848 **(0.156)**	0.909 (0.099)
	XDA	0.991 (0.020)	0.954 (0.049)	0.971 (0.031)
	DeepDi	0.980 (0.041)	0.945 (0.045)	0.961 (0.037)
PIE	IDA	1.000 (0.003)	0.918 (0.097)	0.954 (0.059)
	XDA	0.988 (0.022)	0.969 (0.050)	0.978 (0.034)
	DeepDi	0.973 (0.058)	0.926 (0.064)	0.946 (0.053)
Obfuscate	IDA	1.000 (0.001)	0.843 **(0.132)**	0.909 (0.082)
	XDA	0.920 (0.085)	0.978 (0.034)	0.946 (0.050)
	DeepDi	0.973 (0.040)	0.932 (0.055)	0.950 (0.034)
CFI	IDA	1.000 (0.002)	0.806 **(0.183)**	0.880 **(0.122)**
	XDA	0.975 (0.031)	0.883 **(0.111)**	0.923 (0.075)
	DeepDi	0.968 (0.047)	0.880 (0.073)	0.919 (0.049)
ASE18	IDA	1.000 (0.001)	0.832 **(0.117)**	0.904 (0.069)
	XDA	scores omitted: *ASE18 was the training dataset*		
	DeepDi	0.977 (0.025)	0.957 (0.021)	0.966 (0.018)
Totals	IDA	1.000 (0.002)	0.843 **(0.145)**	0.907 (0.092)
	XDA	0.982 (0.040)	0.959 (0.062)	0.969 (0.044)
	DeepDi	0.976 (0.042)	0.931 (0.058)	0.951 (0.041)

Models Under Test. We selected the commercial standard IDA Pro v7.7 as a baseline deterministic disassembler. As exemplars of state-of-the-art neural binary analyses for function boundary detection, we selected XDA [54] and DeepDi [76], both of which we previously introduced in §2. XDA was selected for evaluation because its design is heavily inspired by Transformer [68], and its implementation on top of Fairseq's implementation of BERT [21] reflects this. As such, it is a perfect example of an NLP-based approach to neural function boundary detection. DeepDi, on the other hand, was selected as an example of a function boundary detection system that incorporates some semantic information in the form of a graph model of instruction dependencies. Finally, both of these systems publish a public artifact for evaluation: source code in the case of XDA [35], and a binary distribution in the case of DeepDi [20]. We thank the authors of these systems for their commitment to open science.

Datasets. To support reproducibility, we built our evaluation upon datasets generated for previous binary analysis evaluations. In particular, we started with the BinKit corpus [36] which is based on all available GNU software packages. BinKit includes 53 software packages compiled by five versions of GCC (v4.9.4, v5.5.0, v6.4.0, v7.3.0, v8.2.0) and four versions of clang (v4.0, v5.0, v6.0, v7.0). The original corpus is composed of several distinct datasets that exercise specific compiler options: `-fno-inline`, `-fPIE`, `-Os`, and `-flto`. We expanded the corpus to include more recent versions of GCC and Clang (GCC v.9.4.0, GCC v11.2.0, Clang v9.0, Clang v13.0) and a new dataset (CFI). The CFI dataset exercises modern control-flow integrity compiler options discussed in §3.2 via the GCC compiler flag `-fcf-protection=full` and the Clang compiler flag `-fsanitize=cfi`. An overview of the corpus and individual datasets is presented in Table 2.

Although the BinKit corpus includes a substantial combination of compiler versions, optimization levels, and specific flags, one cannot assume that compiler options are completely isolated for any particular binary or dataset. For example, one might assume the NoInline dataset would not include code that had been compiled with the flag `-fgnu89-inline`, which causes inlining, or that a binary compiled at the `O0` optimization level would not include code compiled at a different optimization level. Unfortunately, this is not the case due to existence of compiler-generated code and code that is statically linked in from compiler support libraries. We found that binaries compiled with `-fstack-protector-strong` included code compiled with `-fno-stack-protector`, although the presence of the latter was dominated by the former. In some cases, such as the xorriso binary with 3000 functions, compiler support code is dominated by the software library code, and thus the presence of code compiled with different flags would have a minor impact on training and evaluation. On the other hand, a software library like coreutils is composed of many small utilities where the ratio of compiler support code to library code is much less. We do not believe that this phenomenon has a significant impact on our results, but we do note that it is non-trivial to ensure uniform compiler configurations on absolutely all code in each dataset and that we did not attempt to achieve this.

Metrics. We report precision, recall, and the balanced F-score (F_1 score) with the standard definitions. In Table 3, we report the mean and standard deviation of the precision, recall, and F_1 score as a statistical summary calculated per binary in each dataset. We choose to report mean and standard deviation because performance within a particular dataset can exhibit high variance, as we discuss later.

Computational Resources. All experiments were performed on a dedicated server with a 64 core AMD Ryzen 3995WX CPU @ 4.3GHz, three RTX A6000 GPUs, 1TB memory, and a 4TB SSD.

5.2 RQ1: Inadvertent Attacks

In our first experiment, we subjected XDA and DeepDi to our augmented version of BinKit to evaluate their resilience to the full set of inadvertent evasions described in §3.2. We additionally include IDA Pro in this experiment as a baseline representing the state of the art in deterministic function boundary detection. Table 3 presents summary statistics in terms of precision, recall, and F1 score, along with standard deviation for each metric, broken out by the individual datasets comprising BinKit.

From the data, IDA Pro consistently performs best with respect to precision, with little variance. XDA, however, dominates with respect to recall and F1 score. DeepDi produces F1 scores that are very close to the performance of XDA and takes the top spot for exactly one dataset, Obfuscate. There is also clearly some variance across all metrics. However, in this respect the summary statistics do not tell the full story.

Figure 2 presents a series of precision-recall plots for each system. Each point represents one binary, colored according to membership in each of BinKit's constituent datasets. In each plot, the optimal point is the upper-right, indicating perfect precision (all detections were true positives) and recall (all functions were detected). Points towards the x-axis indicate lower precision and thus a higher proportion of false positives. Points towards the y-axis (left) indicate lower recall and thus a higher proportion of false negatives.

One can immediately observe a marked difference between the operating characteristics of the deterministic baseline represented by IDA Pro and the NBA systems. IDA Pro consistently achieves near perfect precision – i.e., when it detects a function, it is highly likely to be a true positive. However, it is prone in some cases to unreported functions. In the worst case, IDA Pro dips below 0.4 recall.

Both XDA and DeepDi, however, exhibit much stronger variance in both precision and recall. XDA in particular presents seeming clusters, i.e., precision-recall that correlates with individual datasets. XDA performs particularly poorly on the CFI dataset, colored in red. Other datasets are biased towards either precision or recall failures. For instance, Obfuscate, colored in purple, tends towards lower recall and false negatives. SizeOpt failures, in contrast, are biased towards lower precision and false positives.

In comparative terms, XDA performs slightly better across the board than XDA and both exhibit better recall than IDA Pro on this data. However, the scatterplot makes it clear that there is a sizable number of outliers in both precision and recall. Thus, we investigated a sample of these outliers.

One such outlier point is shown in Listing 3. On the gcal-4.1 benchmark, DeepDi issued >3000 distinct false positives from multiple occurrences of a single instruction. The instruction, highlighted in red, subtracts 8 bytes from the stack pointer. This is an operation that is often performed in a function prologue to allocate space for local variables on the stack. However, this particular example occurs when marshalling arguments to a call to `fprintf` in gcal's `main` function. The reason this occurs is because this particular call to `fprintf`, a variadic function, has more than six arguments. The SysV ABI dictates that the first six arguments are passed in registers, while any further arguments are passed on the stack. Stack arguments, however, must be aligned to a 16-byte boundary. This causes the compiler, which was configured to operate at `O0` in this case, to directly adjust the stack pointer prior to pushing the seventh argument to `fprintf`. It appears that the DeepDi model we evaluated never observed this particular pattern in its training set.

```
sub    rsp, 0x8          ; align stack pointer
push   r13               ; push arg7
mov    r9, r12           ; set arg6
mov    r8, rbp           ; set arg5
mov    ecx, 0x4972ec     ; set arg4
mov    rdx, rbx          ; set arg3
mov    rsi, rax          ; set arg2
mov    rdi, stderr       ; set arg1
mov    eax, 0x0          ; zero rax
call   fprintf           ; invoke fprintf
```

Listing 3: One example of a single instruction that causes DeepDi to issue >3000 false positives for the gcal-4.1 benchmark.

One can argue that if the failures we observe are restricted to accidental outliers, then their overall impact should be low. Unfortunately, as we demonstrate next, these inadvertent misclassifications can be systematically exploited by an adversary to build effective adversarial attacks.

5.3 RQ2: Adversarial Attacks

To evaluate adversarial attack efficacy, we recompiled the Normal dataset with different optimization options (`O0`, `O3`, `Os`) and the `-fpatchable-function-entry=4,4` flag which inserts 4 NOP instructions after the original function epilogue. The effects of this on F1 score are presented in Figure 3.

Both XDA and DeepDi successfully handle the addition of simple "NOP sled" insertion, preserving high F1 scores. Unfortunately, when adversarial mutations are introduced following the methodology described in §3.3, both systems diverge significantly from their published accuracy. Interestingly, we observe that XDA is more resilient to epilogue mutation under the `O0` optimization level versus `O3` and `Os`. DeepDi's performance is degraded for all optimization levels, with median F1 scores well below 0.25 at all optimization levels. IDA Pro, however, is largely unaffected by epilogue mutations as evidenced by the near-identical F1 distributions across both datasets.

5.4 RQ3: Systematic Attacks

Our results to this point highlight that both XDA and DeepDi are vulnerable to seemingly simple adversarial byte sequence injection, causing them to misclassify significant portions of the functions present using the same sequence across all binaries with no attempt to adapt them to a given program. Unfortunately, during our evaluation we unearthed several cases where the patterns used were particularly effective, leading to almost complete evasion. Specifically, XDA only managed to recover 36 out of the 145

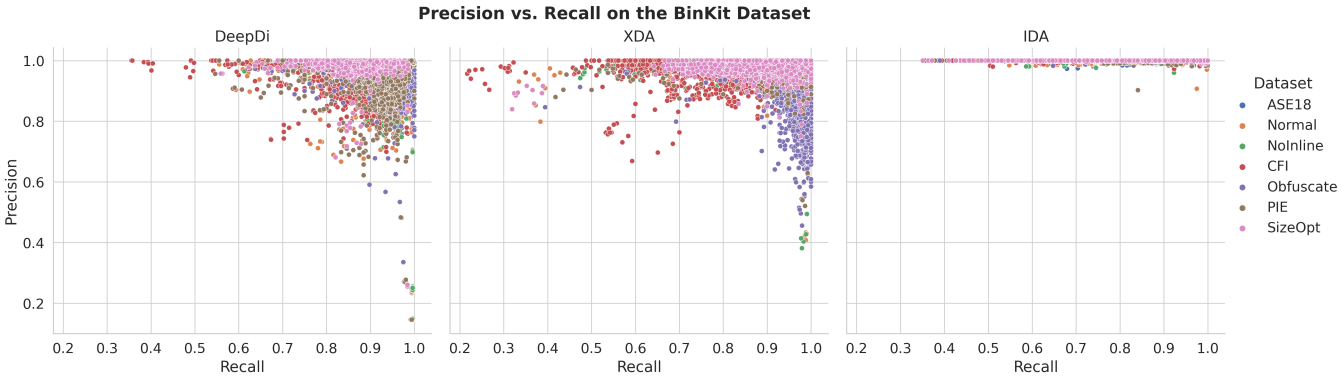

Figure 2: Overview of precision versus recall per binary from the BinKit corpus. IDA Pro consistently performs best with respect to precision, with little variance. XDA, however, dominates with respect to recall. It also wins out on F1 score in all but one case, Obfuscate, where DeepDi is best. Variance in these metrics is somewhat apparent, but better observed in Figure 3.

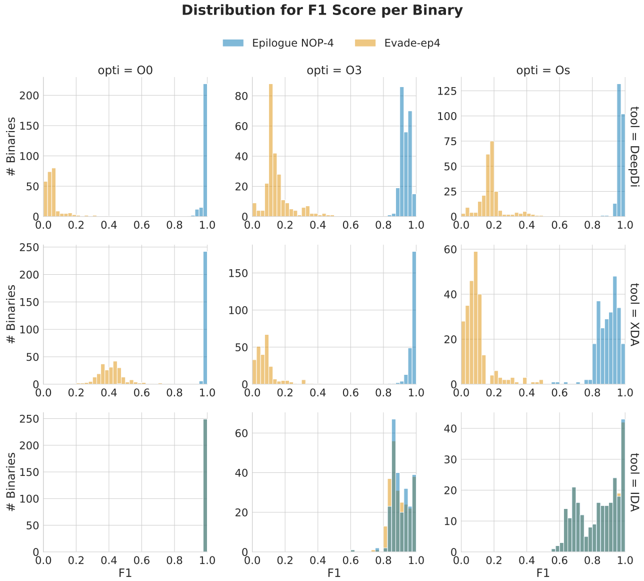

Figure 3: Effects of adversarial attacks on F1 score. We recompile the Normal dataset with the compiler option `-fpatchable-function-entry=4,4`, inserting 4 NOP instructions after each function under different optimization levels. Both XDA and DeepDi are resilient to the simple addition of NOP sequences as shown in the baseline experiments. However, F1 scores exhibit significant degradation when injecting adversarial patterns into the NOP regions. IDA Pro is unaffected by epilogue mutation evasions.

ground truth functions contained in `gnudos-1.11.4_gcc-9.4.0_x86_64_Os_prime.elf`. As another example, DeepDi reported only four (4) out of a total of **6,679** ground truth functions present in `gsl-2.5_gcc-9.4.0_x86_64_O0_libgsl.so.23.1.0.elf`. In the other direction, DeepDi reported reported 3193 out of the 1104 ground truth functions present in `gcal-4.1_gcc-8.2.0_x86_64_O1_gcal.elf`.

We believe that these cases are due to these particular adversarial patterns being especially effective on the characteristic layout of those programs. Furthermore, our adversarial attacks could potentially be improved by targeting them for particular binaries, paving the way for a novel, insidious way to attack NBAs relying only on static information. As is evident in the DeepDi case, such adversarially mutated binaries would be virtually invisible to detectors that relied on a vulnerable NBA for as part of its analysis pipeline. While the targeted attacks we speculate about here are beyond the scope of this work, we believe that it is a promising line of inquiry and plan to explore it as future work.

5.5 RQ4: Expanded Training Sets

To investigate whether inadvertent attacks can be mitigated with additional training, we next conducted a step-wise experiment with XDA.[5] For the inadvertent samples, we chose to evaluate XDA with the CFI dataset as it was the most difficult dataset to classify for all three systems under evaluation. Starting with a very limited subset of the ASE18 dataset, we trained XDA with increasingly more diversity in the number of compilers, compiler versions, and compiler options. The results are shown in Table 4. With only one version of GCC and four optimization levels in the training data, XDA achieved a reasonable F1 score of 0.855 on the CFI dataset. By adding four versions of Clang and GCC, a modest improvement in F1 score obtained (0.865). Notably, adding Clang-compiled binaries to the training set reduced XDA's performance.

We then expanded the original ASE18 dataset by including newer compilers, namely two versions of Clang and GCC, which increased the F1 score to 0.923. Finally, by adding the `Os` optimization level, XDA achieved a score of 0.924, which is better than both DeepDi and IDA Pro. This demonstrates that XDA's performance can in fact be improved by expanding the training dataset – which is expected – but also that XDA is also quite sensitive to compiler versions and options present in the training data.

[5] We are restricted to XDA for this and the following experiment since DeepDi is distributed as a binary object. Thus, we do not have the ability to train a new DeepDi model.

Table 4: Improving resilience through training.

ID	Base (+added)	# Files	Data Size	Time	# GCC,	# Clang	Eval Dataset	F1
0	GCC-6.4.0	772	118M	0.4h	1		CFI	0.855
1	Clang	3,088	461M	1.7h		4	CFI	0.575
2	GCC	3,860	586M	2.2h	5		CFI	0.857
3	original	6,948	1,046M	3.9h	5	4	CFI	0.865
4	3 +Clang-new	6,988	1,153M	4.3h	5	6	CFI	0.852
5	3 +GCC-new	6,988	1,148M	4.3h	7	4	CFI	0.913
6	3 +both-new	7,028	1,256M	4.7h	7	6	CFI	0.923
7	6 +Os	7,058	1,312M	4.9h	7	6	CFI	0.924
8	7 +nop4	7,133	1,499M	5.6h	7	6	CFI	0.935
9	8 +evade4	7,208	1,685M	6.4h	7	6	CFI	0.810
7	6 +Os						Evade-ep4	0.198
8	7 +nop4						Evade-ep4	0.338
9	8 +evade4						Evade-ep4	0.938

5.6 RQ5: Adversarial Training

In the final experiment, we evaluate whether MUTs can be made resilient to the adversarial attacks we describe by adopting adversarial training. In order to train XDA on these crafted attacks, we created a new dataset based on the NOP dataset described in §5.3. In this dataset, we replaced each 4-byte NOP epilogue with a randomly chosen evasion pattern that is also a valid 4-byte x86 instruction sequence. We then fine-tuned XDA on this expanded dataset and evaluated on both the CFI and Evade-ep4 datasets. With the new model, XDA’s performance on the Evade-ep4 dataset improved from 0.198 to 0.938, a significant improvement. Unfortunately, XDA’s performance on the CFI dataset was also degraded from 0.924 to 0.810. This suggests that while adversarial training can partially mitigate evasion, it also comes at a significant cost in accuracy for benign samples.

In addition, it is also unclear whether training on adversarial examples represents a trustworthy mitigation. To illustrate, we performed an additional round of adversarial attack search to demonstrate the inherent limitation of training against adversarial techniques. Repeating our 4-byte evasion search, we were able to reduce XDA’s performance to 0.488 (STD 0.317) when trained on the Evade-ep4 dataset. Additionally, we studied two alternative attacks using a 3-byte and 8-byte NOP dataset, producing F1 scores of 0.427 and 0.430 respectively. Thus, while one would hope that training on adversarial examples would produce a model that is robust against many different evasion patterns, our experiments show that this is unlikely to be the case as we were able to degrade XDA’s performance again without significant effort.

6 DISCUSSION

Black-box attacks are powerful enough. As is hopefully clear from our evaluation, black-box attacks are sufficiently powerful to discover numerous false positive and false negative-inducing inputs to current generation function boundary detection NBAs. Sophisticated white-box searches for adversarial examples that rely on gradient descent might well find more attacks. However, it is unclear how one might adapt existing searches while preserving the functionality of the mutated binary due to the discrete problem space. Nevertheless, this is an interesting direction to explore.

Inadvertent attacks break pure NLP-based systems. As should also be clear from the evaluation, inadvertent attacks significantly degrade function boundary detection approaches that directly reuse NLP embeddings and models as XDA does. Another way to view this finding is that such approaches do not generalize well to examples that are not observed during training. In retrospect, this naturally follows from our conjecture that syntactic representations are not a sound basis for binary analysis where semantics is virtually always what actually matters. One could argue that simply including misclassified examples in the training set is sufficient mitigation, and there is likely some truth to that. However, in our opinion a realistic counterargument is that anticipating and training on a sufficiently large permutation of compilers, compiler versions, and compiler configurations is combinatorially difficult. To make matters worse, that mitigation does not take adversarial attacks into account.

Domain-specific embeddings and graph models are a marginal improvement. The evaluation shows that DeepDi’s domain-specific embedding and use of R-GCN to model instruction dependencies improves its resilience to inadvertent attacks. This is clear evidence that incorporating even a small bit of the latent semantic information present in an instruction stream has utility. However, this improvement is tempered by DeepDi’s performance against adversarial examples, motivating our next observation.

Focus on semantics instead of syntax. The overarching conclusion we draw from the evaluation is that syntactic representations are unlikely to be a reliable basis for binary analyses. In a way, this is unsurprising, since syntactic approaches for attack detection such as signature-based IDS and first-generation anti-virus based on pattern matching against byte sequences were criticized for similar deficiencies long ago. While these techniques can of course be useful, they cannot be relied upon in isolation. Instead, mirroring attack detection’s move from static pattern matching to dynamic behavioral analysis more than a decade ago, we argue that future work in this space should emphasize semantics over syntax to avoid similar pitfalls.

Evaluation quality is important. In tandem with the semantics question, we believe it is crucial that the research community hews to a standard of evaluations on large, representative, public datasets. This data should include a range of programs with varying functionality, as well as different compilers, compiler versions, and compiler configurations. As shown in our experiments, testing on a more comprehensive dataset such as BinKit [36] over smaller, less representative datasets in the original papers can help identify areas of improvement for underlying models, such as lacking understanding of semantic isomorphisms. Finally, we believe that these benchmark corpora should include adversarial examples generated using techniques such as those described herein to directly test whether future work is susceptible to similar attacks. This inclusion should both to increase robustness in possible security related use cases and to help the model learn patterns of adversarial perturbation that exploit syntactic versus semantic model understanding. A substantial bonus in following such a standard would be to ease reproducibility and comparative evaluation.

Detecting adversarial code is not easy. Finally, we readily acknowledge that the adversarial code we inject as part of our methodology and evaluation is likely to be easy to detect and strip before performing classification. However, we believe that focusing on this is misguided. Code obfuscation is well within the threat model of many contexts in which binary analyses such as function boundary detection operate under. In that light, it is reasonable to suspect that if an adversary wished to do so, they could easily obfuscate the fact that the injected code will never be executed by relying on computed control transfers and opaque predicates. Indeed, if a defender was able to perfectly identify dead code, then a large part of the debloating problem would be perfectly solved which – to our knowledge – is not the case. Instead, as with so many other problems in this space, detecting and removing adversarial code reduces to Rice's Theorem [60]. Thus, we believe it is safe to conclude that this is not likely to be a fruitful research direction.

7 RELATED WORK

Neural binary analysis. Binary analysis is a long-studied and expansive research area. Disassembly is a fundamental task that traditionally has been solved using deterministic algorithms that can be broadly classified as either linear disassembly (provided by tools like objdump from GNU binutils) or recursive descent disassembly (provided by tools such as IDA Pro [29], Ghidra [50], or Binary Ninja [69]). These tools typically also incorporate algorithms for function boundary detection using some combination of symbol table information, debug information, and pattern-based heuristics. Work such as ByteWeight [9] specifically investigated learning-based approaches for performing function boundary detection. Other common binary analysis tasks include measuring similarity between snippets of binary code [32], recovering source code types [40], and decompilation [62, 73]. In recent years, applying deep learning techniques to binary analysis problems has become a popular topic of study due to the success of DNNs in solving image and text processing tasks, among others. Shin et al. [64] were the first to apply a DNN to a binary analysis problem; in this case, detecting function boundaries using a bi-directional recurrent neural network (BiRNN). The strategy of repurposing embeddings and model architectures originally developed to solve NLP or image processing problems became *de rigeur* in a way. Numerous NBAs for disassembly [54, 76], function boundary detection [17, 54, 76], value set analysis [31, 42], static code similarity [22, 41, 42, 46, 72, 74, 77, 78], decompilation [25], and malware analysis [3, 33, 70] directly use embeddings (e.g., word2vec [47], PV-DM [39]) or models (e.g, RNN, CNN, Transformer [68], BERT [21]) developed for the NLP or image problem domains. One of the conclusions we draw in this paper is that while it is tempting to build on techniques that have been successful in other areas, binary analysis is a strikingly different research area with a different threat model and much stronger accuracy requirements for downstream tasks (see the discussion in §6). For NBAs to be resilient against adversaries that seek to evade or confuse binary analyses, choices of embeddings and model architectures should reflect these requirements.

We are not the first to independently evaluate NBA systems for other tasks. Kim et al. [36] studied NBAs that perform static similarity detection using a large dataset of programs compiled with a variety of toolchains and compiler options called BinKit; we build on BinKit to carry out our own evaluation. Using this dataset and a simple baseline similarity detector called TikNib, they show that NBAs do not necessarily outperform simpler, explainable methods such as the one implemented by TikNib. Marcelli et al. [45] performed a similar study also focused on static similarity detection NBAs, and show that published results do not necessarily hold when the systems-under-test are trained and evaluated on larger, more representative datasets. Finally, Lucas et al. showed that DNNs used for static malware detection on binary programs are prone to adversarial attacks [43]. This work lies in contrast to our own not only in the specific problem domain but also in its use of traditional adversarial ML techniques – i.e., white-box gradient descent or black-box hill climbing – to find evading transformations.

Adversarial machine learning. Substantial research has studied the problem of crafting adversarial examples [13, 53]. Traditionally, this research has been conducted on semi-continuous spaces, here defined as when adjacent values carry semantic information, e.g., pixel values for image classification. In these approaches, attacks use a variety of derivative-based approaches to optimize loss over some non-convex objective function [7, 12, 16, 27, 30, 49, 65]. In our case, we examine executable binaries, where we must work under more difficult constraints. First, adjacent values for binary code do not carry semantic meaning. For instance, 0x8F is the binary encoding of the x86 pop instruction, whereas 0x90 is the semantically unrelated nop instruction. This difference is non-trivial as it presents a much harder problem than that of optimization over semi-continuous spaces; in fact, it reduces to integer factorization, an NP-complete problem [34]. Pierazzi et al. [55] provide detailed insight into how different problem spaces under which adversarial machine learning is conducted, such as using binary code as the input to a DNN, require specific black-box attacks because traditional gradient-based approaches fail. Another constraint we must satisfy is to produce valid executable binaries. These constraints are similar to those necessary in any attack that attempts to modify binary code [4, 43].

As stated in the previous subsection, other work in using deep learning for malware analysis has looked the problem of mapping binaries to either malicious or benign software [57, 58]. In turn, various work has aimed to attack this type of machine learning model and others like it [4, 43]. However, this paper presents the first exploration into evaluating the robustness of deep learning models against both inadvertent attacks and crafted adversarial examples.

8 CONCLUSIONS AND FUTURE WORK

We presented the first study of the resilience of neural function boundary detectors to inadvertent and adversarial attacks. Our methodology demonstrates that straightforward black-box search using a large dataset and toolchain array is sufficient to identify numerous adversarial examples for two state-of-the-art systems: XDA [54] and DeepDi [76]; sophisticated white-box search algorithms are unnecessary. Our conjecture – which we believe is validated by our evaluation – is that these systems are susceptible to attack because they rely on embeddings and model architectures intended for syntactic inference, and do not sufficiently consider the semantics of the ISAs they operate on.

This is not to say that this research direction should be abandoned. To the contrary, we believe there remains significant potential for applying deep learning to binary analysis problems. However, future research might well benefit from focusing on instruction semantics rather than syntactic representations. In addition, future work should ensure that evaluations are based on large, representative datasets that includes adversarial examples intended to exploit syntactic dependence. An intriguing research question is whether effective embeddings and model architectures can be developed specifically for binary analysis tasks. We plan to investigate this question in our future work, and hope others will as well.

ACKNOWLEDGMENTS

The views expressed herein are those of the author and do not reflect the position of the United States Military Academy, the Department of the Army, or the Department of Defense.

This material is based upon work supported by the National Science Foundation under Grant No. 2203175.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. 13, 1 (2009), 4:1–4:40. https://doi.org/10.1145/1609956.1609960

[2] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: Debloating Binary Shared Libraries. In *Proceedings of the Annual Computer Security Applications Conference* (San Juan Puerto Rico USA, 2019-12-09). ACM, 70–83. https://doi.org/10.1145/3359789.3359823

[3] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May OReilly. 2018. Adversarial Deep Learning for Robust Detection of Binary Encoded Malware. In *Proceedings of the IEEE Security and Privacy Workshops* (San Francisco, CA, 2018-05). IEEE, 76–82. https://doi.org/10.1109/SPW.2018.00020

[4] Hyrum S. Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. 2018. Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning. https://doi.org/10.48550/ARXIV.1801.08917

[5] Dennis Andriesse, Xi Chen, Victor van der Deen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale X86/X64 Binaries. In *Proceedings of the USENIX Security Symposium* (2016). 19.

[6] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-Agnostic Function Detection in Binaries. In *Proceedings of the IEEE European Symposium on Security and Privacy* (2017-04). 177–189. https://doi.org/10.1109/EuroSP.2017.11

[7] Anish Athalye, Nicholas Carlini, and David Wagner. 2018. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *Proceedings of the International Conference on Machine Learning* (2018). PMLR, 274–283.

[8] Stefan Axelsson. 2000. The Base-Rate Fallacy and the Difficulty of Intrusion Detection. 3, 3 (2000), 20.

[9] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. ByteWeight: Learning to Recognize Functions in Binary Code. In *Proceedings of the USENIX Security Symposium* (2014-08). 17.

[10] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. 2018. Superset Disassembly: Statically Rewriting X86 Binaries Without Heuristics. In *Proceedings of the ISOC Network and Distributed System Security Symposium* (San Diego, CA, 2018). Internet Society. https://doi.org/10.14722/ndss.2018.23300

[11] Eli Bendersky. 2012. *Pyelftools*. https://github.com/eliben/pyelftools

[12] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. 2013. Evasion attacks against machine learning at test time. In *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (2013). Springer, 387–402.

[13] Battista Biggio and Fabio Roli. 2018. Wild patterns: Ten years after the rise of adversarial machine learning. 84 (2018), 317–331.

[14] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models Are Few-Shot Learners. (2020). https://doi.org/10.48550/ARXIV.2005.14165

[15] Capstone Developers. 2014. *Capstone Disassembler*. https://www.capstone-engine.org

[16] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *Proceedings of the IEEE Symposium on Security and Privacy* (2017). IEEE, 39–57.

[17] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural Nets Can Learn Function Type Signatures From Binaries. In *Proceedings of the USENIX Security Symposium* (2017). 19.

[18] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, and Peat Bakke. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the USENIX Security Symposium* (1998). 63–78.

[19] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative Embeddings of Latent Variable Models for Structured Data. (2016). https://doi.org/10.48550/ARXIV.1603.05629

[20] DeepBits Developers. 2022. *DeepDi*. DeepBits. https://www.deepbitstech.com/deepdi.html

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. (2019). arXiv:1810.04805 [cs] http://arxiv.org/abs/1810.04805

[22] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *Proceedings of the IEEE Symposium on Security and Privacy* (2019-05). 472–489. https://doi.org/10.1109/SP.2019.00003

[23] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. In *Proceedings of the ISOC Network and Distributed System Security Symposium* (San Diego, CA, 2020). Internet Society. https://doi.org/10.14722/ndss.2020.24311

[24] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security* (Vienna Austria, 2016-10-24). ACM, 480–491. https://doi.org/10.1145/2976749.2978370

[25] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An End-to-End Neural Program Decompiler. In *Proceedings of the Conference on Neural Information Processing Systems* (2019-06-27). arXiv:1906.12029 http://arxiv.org/abs/1906.12029

[26] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the International Conference on Machine Learning* (2017-04-04). https://doi.org/10.48550/arXiv.1704.01212

[27] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. (2014).

[28] Aditya Grover and Jure Leskovec. 2016. Node2vec: Scalable Feature Learning for Networks. (2016). https://doi.org/10.48550/ARXIV.1607.00653

[29] Ilfak Guilfanov. 2022. *IDA Pro*. Hex Rays. https://hex-rays.com/ida-pro/

[30] Chuan Guo, Jacob Gardner, Yurong You, Andrew Gordon Wilson, and Kilian Weinberger. 2019. Simple black-box adversarial attacks. In *International Conference on Machine Learning* (2019). PMLR, 2484–2493.

[31] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. 2019. DeepVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. In *Proceedings of the USENIX Security Symposium* (2019). 1787–1804. https://www.usenix.org/conference/usenixsecurity19/presentation/guo

[32] Irfan Ul Haq and Juan Caballero. 2019. A Survey of Binary Code Similarity. (2019). arXiv:1909.11424 [cs] http://arxiv.org/abs/1909.11424

[33] Wenyi Huang and Jack W. Stokes. 2016. MtNet: A Multi-Task Neural Network for Dynamic Malware Classification. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2016-07-07) *(DIMVA 2016)*. Springer-Verlag, 399–418. https://doi.org/10.1007/978-3-319-40667-1_20

[34] Richard M. Karp. 1972. *Reducibility among Combinatorial Problems*. Springer US, 85–103. https://doi.org/10.1007/978-1-4684-2001-2_9

[35] Kexin Pei. 2021. *XDA*. Columbia University. https://github.com/CUMLSec/XDA

[36] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2022. Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned. *IEEE Transactions on Software Engineering* (2022), 1–23. https://doi.org/10.1109/TSE.2022.3187689

[37] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO, 2014-10-06) *(OSDI'14)*. USENIX Association, 147–163.

[38] Jeff Law. 2020. *Stack Clash Mitigation in GCC, Part 3*. Red Hat Developer. https://developers.redhat.com/blog/2020/05/22/stack-clash-mitigation-in-gcc-part-3

[39] Quoc V. Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the International Conference on Machine Learning* (2014). arXiv. https://doi.org/10.48550/ARXIV.1405.4053

[40] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the ISOC Network and Distributed System Security Symposium* (2011). 18.

[41] Yongjun Lee, Hyun Kwon, Sang-Hoon Choi, Seung-Ho Lim, Sung Hoon Baek, and Ki-Woong Park. 2019. Instruction2vec: Efficient Preprocessor of Assembly Code to Detect Software Weakness with CNN. 9, 19 (2019), 4086. https://doi.org/10.3390/app9194086

[42] Xuezixiang Li, Qu Yu, and Heng Yin. 2021. PalmTree: Learning an Assembly Language Model for Instruction Embedding. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security* (2021-05-07). arXiv:2103.03809 http://arxiv.org/abs/2103.03809

[43] Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K. Reiter, and Saurabh Shintre. 2021. Malware Makeover: Breaking ML-based Static Analysis by Modifying Executable Bytes. In *Proceedings of the ACM Asia Conference on Computer and Communications Security* (Virtual Event Hong Kong, 2021-05-24). ACM, 744–758. https://doi.org/10.1145/3433210.3453086

[44] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2019. Towards Deep Learning Models Resistant to Adversarial Attacks. (2019). arXiv:1706.06083 [cs, stat] http://arxiv.org/abs/1706.06083

[45] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How Machine Learning Is Solving the Binary Function Similarity Problem. In *Proceedings of the USENIX Security Symposium* (2022). 18.

[46] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2019. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, Cham, 309–329.

[47] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. (2013). arXiv:1301.3781 [cs] http://arxiv.org/abs/1301.3781

[48] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic Disassembly. In *Proceedings of the International Conference on Software Engineering* (Montreal, Quebec, Canada, 2019-05-25) *(ICSE '19)*. IEEE Press, 1187–1198. https://doi.org/10.1109/ICSE.2019.00121

[49] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016). 2574–2582.

[50] NSA. 2019. *Ghidra.* US National Security Agency. https://ghidra-sre.org

[51] Mott Ott, Sergey Edunov, Alexey Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. Fairseq. In *NAACL-HLT 2019: Demonstrations* (2019). https://github.com/pytorch/fairseq

[52] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. SoK: All You Ever Wanted to Know About X86/X64 Binary Disassembly But Were Afraid to Ask. In *Proceedings of the IEEE Symposium on Security and Privacy* (2021-05). 833–851. https://doi.org/10.1109/SP40001.2021.00012

[53] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael P. Wellman. 2018. SoK: Security and Privacy in Machine Learning. In *Proceedings of the IEEE European Symposium on Security and Privacy* (2018). 399–414. https://doi.org/10.1109/EuroSP.2018.00035

[54] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. 2021. XDA: Accurate, Robust Disassembly with Transfer Learning. In *Proceedings of the ISOC Network and Distributed System Security Symposium* (2021). arXiv:2010.00770 http://arxiv.org/abs/2010.00770

[55] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. 2020. Intriguing Properties of Adversarial ML Attacks in the Problem Space. In *Proceedings of the IEEE Symposium on Security and Privacy* (2020). arXiv. https://doi.org/10.48550/ARXIV.1911.02142

[56] Qualys. 2017. *Qualys Security Advisory: The Stack Clash.* https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt

[57] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. 2018. Malware Detection by Eating a Whole EXE. In *Proceedings of the AAAI Workshop on Artificial Intelligence for Cyber Security* (2018). arXiv. https://doi.org/10.48550/ARXIV.1710.09435

[58] Edward Raff, Jared Sylvester, and Charles Nicholas. 2017. Learning the PE Header, Malware Detection with Minimal Domain Knowledge. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security* (New York, NY, USA, 2017). Association for Computing Machinery, 121–132. https://doi.org/10.1145/3128572.3140442

[59] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. 2021. Unleashing the hidden power of compiler optimization on binary code difference: an empirical study. In *PLDI*. ACM, New York, NY, USA, 142–157. https://doi.org/10.1145/3453483.3454035 arXiv:2103.12357

[60] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. 74, 2 (1953), 358–366.

[61] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *Proceedings of the European Semantic Web Conference* (2018) *(Lecture Notes in Computer Science, Vol. 10843)*. Springer International Publishing, 593–607. https://doi.org/10.1007/978-3-319-93417-4_38

[62] Edward J Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. 2013. Native X86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Proceedings of the USENIX Security Symposium* (2013-08). 17.

[63] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy* (Phoenix AZ USA, 2019-06-23). ACM, 1–11. https://doi.org/10.1145/3337167.3337175

[64] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the USENIX Security Symposium* (2015). 17.

[65] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. (2013).

[66] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. (2015). https://doi.org/10.48550/ARXIV.1503.00075

[67] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the USENIX Security Symposium* (2014). 16.

[68] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Proceedings of the Conference on Neural Information Processing Systems* (2017). 11.

[69] Vector 35. 2016. *Binary Ninja.* Vector 35. https://binary.ninja

[70] R. Vinayakumar, Mamoun Alazab, K. P. Soman, Prabaharan Poornachandran, and Sitalakshmi Venkatraman. 2019. Robust Intelligent Malware Detection Using Deep Learning. 7 (2019), 46717–46738. https://doi.org/10.1109/ACCESS.2019.2906934

[71] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne Switzerland, 2020-03-09). ACM, 133–147. https://doi.org/10.1145/3373376.3378470

[72] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security* (2017). 363–376. https://doi.org/10.1145/3133956.3134018 arXiv:1708.06525

[73] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Proceedings of the ISOC Network and Distributed System Security Symposium* (San Diego, CA, 2015). Internet Society. https://doi.org/10.14722/ndss.2015.23185

[74] Jia Yang, Cai Fu, Xiao-Yang Liu, Heng Yin, and Pan Zhou. 2021. Codee: A Tensor Embedding Scheme for Binary Code Search. (2021), 1–1. https://doi.org/10.1109/TSE.2021.3056139

[75] Miuyin Yong Wong, Matthew Landen, Manos Antonakakis, Douglas M. Blough, Elissa M. Redmiles, and Mustaque Ahamad. 2021. An Inside Look into the Practice of Malware Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2021-11-12) *(CCS '21)*. Association for Computing Machinery, 3053–3069. https://doi.org/10.1145/3460120.3484759

[76] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. 2022. DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly. In *Proceedings of the USENIX Security Symposium* (2022). 17.

[77] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2020-04-03), Vol. 34. 1145–1152. https://doi.org/10.1609/aaai.v34i01.5466

[78] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. In *Proceedings of the ISOC Network and Distributed System Security Symposium* (2019). https://doi.org/10.14722/ndss.2019.23492 arXiv:1808.04706

A INADVERTENT EVASION EFFECTS

```
f_no_stack_protector:
    push   rbp          ; save caller frame pointer
    mov    rbp, rsp     ; set frame pointer
    ; function body...
    pop    rbp          ; restore caller frame pointer
    ret                 ; return to caller
```

```
f_stack_protector_strong:
    push   rbp                      ; save caller fp
    mov    rbp, rsp                 ; set fp
    sub    rsp, 0x10                ; alloc stack
    mov    rax, qword fs:0x28       ; get global guard
    mov    qword [rbp-0x8], rax     ; inject guard on stack
    ; function body...
    mov    rax, qword fs:0x28       ; get global guard
    mov    rcx, qword [rbp-0x8]     ; get stack guard
    cmp    rax, rcx                 ; compare guards
    jne    .fail                    ; jump if not equal
    add    rsp, 0x10                ; dealloc stack
    pop    rbp                      ; restore fp
    ret                             ; return to caller
.fail:
    call   __stack_chk_fail         ; terminate program
```

Listing 4: Stack protector function prologue and epilogue modifications (additions in green).

```
f_no_stack_clash_protection:
    push   rbp              ; save caller fp
    mov    rbp, rsp         ; set fp
    sub    rsp, 0x10020     ; allocate buffer
    ; function body...
```

```
f_stack_clash_protection:
    push   rbp                  ; save caller fp
    mov    rbp, rsp             ; set fp
    mov    r11, rsp             ; get sp
    sub    r11, 0x10000         ; set stack alloc size
.next_page:
    sub    rsp, 0x1000          ; alloc next stack page
    mov    qword [rsp], 0x0     ; probe page with store
    cmp    rsp, r11             ; check if done allocing
    jne    .next_page           ; if not, loop
    sub    rsp, 0x20            ; alloc final 0x20
    ; function body...
```

Listing 5: Stack clash protection function prologue modifications (deletions in red, additions in green).

```
f_no_cet:
    push   rbp              ; save caller fp
    mov    rbp, rsp         ; set fp
    sub    rsp, 0x10        ; alloc stack
    ; function body...
```

```
f_cet:
    endbr64                 ; label valid branch target
    push   rbp              ; save caller fp
    mov    rbp, rsp         ; set fp
    sub    rsp, 0x10        ; alloc stack
    ; function body...
```

Listing 6: Intel CET function prologue modifications (additions in green).

```
f_no_safe_stack:
    push   rbp              ; save caller fp
    mov    rbp, rsp         ; set fp
    sub    rsp, 0x1010      ; alloc stack vars
    ; function body...
    add    rsp, 0x1010      ; dealloc stack vars
    pop    rbp              ; restore caller fp
    ret                     ; return

```

```
f_safe_stack:
    push   rbp              ; save caller fp
    mov    rbp, rsp         ; set fp
    sub    rsp, 0x20        ; alloc safe stack
                            ; get unsafe stack pointer
    mov    rcx, __safestack_unsafe_stack_ptr
    ; function body...
    add    rsp, 0x20        ; dealloc safe stack
    pop    rbp              ; restore caller fp
    ret                     ; return

```

Listing 7: SafeStack function prologue and epilogue modifications (deletions in red, additions in green).

```
    add    rsp, 0x8         ; prev fn epilogue
    pop    rbx
    pop    r14
    ret
f_unaligned:
    push   rbp              ; save caller registers
    push   r15
    push   r14
    push   r13
    push   r12
    push   rbx
    sub    rsp, 0xe8        ; alloc stack vars
    ; function body...
```

```
    add    rsp, 0x8         ; prev fn epilogue
    pop    rbx
    pop    r14
    ret
    int3                    ; sw bkpt padding bytes
    int3
    int3
    ; many repetitions...
    int3
    int3
    int3
f_aligned:
    push   rbp              ; save caller registers
    push   r15
    push   r14
    push   r13
    push   r12
    push   rbx
    sub    rsp, 0xe8        ; alloc stack vars
    ; function body...
```

Listing 8: Interstitial function padding modifications due to varying alignment constraints (additions in green).