UNIVERSITY OF CALIFORNIA
RIVERSIDE

Accurate and Efficient SBOM Generation for Software Supply Chain Security

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Sheng Yu

September 2024

Dissertation Committee:

Dr. Heng Yin, Chairperson
Dr. Chengyu Song
Dr. Zhiyun Qian
Dr. Qian Zhang

The Dissertation of Sheng Yu is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First and foremost, I would like to express my heartfelt gratitude to my advisor, *Dr. Heng Yin*. His guidance, patience, and insightful feedback have been invaluable in my journey. I feel privileged and honored to have worked under his supervision and to be part of Deepbits Technology Inc. Additionally, my sincere gratitude goes to the rest of my PhD dissertation committee: Dr. Chengyu Song, Dr. Zhiyun Qian, and Dr. Qian Zhang. Their insightful questions and comments greatly enriched my research. I am also grateful to my colleague, Dr. Xunchao Hu, at Deepbits Technology Inc., for his support, guidance, and all the opportunities provided throughout my Ph.D. studies. I would like to acknowledge the support of my lab colleagues: Dr. Jinghan, Dr. Wei, Dr. Ju, Dr. Yu, Lian, Jie, Zhenxiao, Xuezixiang, and Zhaoqi, for their help during my research and the fruitful discussions in our weekly meeting. This dissertation includes previously published materials entitled "On the Correctness of Metadata-based SBOM Generation: A Differential Analysis Approach" published in the 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2024, and "DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly" published in the 31st USENIX Security Symposium, 2022, and one material under submission entitled "GrassDiff: Learning-Free Callgraph Matching for Precise Function Identification in Large Binaries".

ABSTRACT OF THE DISSERTATION

Accurate and Efficient SBOM Generation for Software Supply Chain Security

by

Sheng Yu

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2024
Dr. Heng Yin, Chairperson

Modern software development increasingly relies on software supply chains, with third-party libraries constituting a significant portion of many projects. However, the complexity of dependency relationships and the lack of transparency in software make identifying and fixing vulnerabilities challenging and costly. For example, the average cost of a Log4j incident response has reached $90,000, and nearly 40% of applications still use vulnerable Lo4j two years after the vulnerability was disclosed. A Software Bill of Materials (SBOM), which lists the dependencies used to build software, has been proposed to enhance software visibility and aid in vulnerability detection. Despite this, there is not yet an accurate SBOM generation solution for both source code and binary. Current SBOM generators focus solely on metadata and produce inconsistent SBOM files, while the existing SBOM generators for binary files are either too slow or inaccurate.

In this thesis, we propose an accurate and fast SBOM generation approach for both source code and binary. First, to improve SBOM generation for source code, we conducted a differential analysis to compare and understand how the existing SBOM generators work

and why they behave so differently. We found that these generators support only a subset of common metadata, and their self-implemented parsers for metadata have incomplete syntax supports, leading to erroneous SBOM results. We propose using package managers to simulate dependency installation for metadata-based SBOM generation. Second, we introduce DeepDi, a novel graph neural network-based disassembler that is both accurate and efficient for better SBOM generation for binaries. Our study showed that disassembly is often the bottleneck of binary analysis tasks, consuming up to 90% of processing time. DeepDi improves efficiency by hundreds of times compared to commercial disassemblers and is as accurate or better than them. Third, to further improve the accuracy of SBOM generation from binaries, we propose GrassDiff, a novel learning-free graph-matching algorithm that effectively and efficiently identifies static-linked libraries in large binaries.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The software supply chain plays a pivotal role in the seamless functioning of software development and deployment. While the software supply chain has long been integral to the software development process, its security implications have only recently come under scrutiny. In recent years, attackers have shifted their attention to the software supply chain and have caused tremendous damage. The software supply chain attacks have increased by 742% between 2019 and 2022[1], and the global annual cost of cybercrime has topped $6 trillion in 2021[2]. The main security issue for the software supply chain is visibility: nearly 88% of software supply chain dependencies are unknown to developers and users. As a result, vulnerabilities in these dependencies often stay unnoticed for several months. Even two years after the Log4j vulnerability disclosure, 40% of applications are still using vulnerable Log4j libraries. In response to the growing concerns surrounding software supply chain security and the lack of transparency in software, the concept of the Software Bill of

[1] `https://t.ly/ni24T`
[2] `https://t.ly/w6KPG`

Materials (SBOM) has gained more and more attention. An SBOM is similar to a manifest that provides a detailed inventory of the components and dependencies within a software product. Recognizing the urgency of enhancing software supply chain security, several regulatory bodies, notably the United States government, have issued regulations mandating the adoption of SBOMs to bolster transparency and accountability in the software supply chain.

Despite the regulatory push towards SBOM adoption, a fundamental obstacle impedes its widespread implementation: the accuracy and efficiency of SBOM generation. SBOM generation can be divided into two categories: when the source code is available and when the source code is not available, and there is no practical solution for either.

Correctness of Source SBOM Generation. Current open-source SBOM generation tools [50, 49, 32] extract SBOM information from source code, or more specifically, metadata files. Despite their industrial importance and widespread deployment, their correctness has never been thoroughly studied. Moreover, the reliance on source code limits their applicability since the source code they need is often unobtainable, especially for legacy systems and proprietary products.

Efficiency of Binary SBOM Generation. When the source code is not available, SBOM information has to be extracted from binary code. The extraction often involves disassembling binary code into assembly code, extracting high-level features such as call graphs and control flow graphs, and matching components based on these features. These high-level features all rely on disassembly, but our experiment shows that disassembly is

often the bottleneck in binary analysis tasks and can take as much as 90% of the total processing time [282]. The low efficiency hampers the adoption of SBOMs when they need to be created from binary files.

Accuracy of Binary SBOM Generation. Matching components based on high-level features is the key to accurate Binary SBOM generation. Although academic research [282, 192, 227, 225, 113, 271] has contributed to this field, these approaches are generally slow and work well only on small-scale datasets. For instance, jTrans [271] reports a significant drop in matching accuracy when the number of candidate functions exceeds 10,000. Given that an OpenSSL library contains over 8,000 functions, most academic approaches will not perform well in real-world scenarios. The gap between academic research and practical needs not only hampers the adoption of SBOMs but also poses a significant challenge to enhancing software supply chain security.

1.1 Thesis Statement

This work focuses on efficient and effective SBOM generation from both source code and binary code.

Metadata-based SBOM Generation. To understand how the existing open-source SBOM generators work and what their limitations are, we evaluated the existing popular SBOM tools: Trivy [50], Syft [49], Microsoft SBOM Tool [32], and GitHub Dependency Graph [23]. We collected 7,876 open-source projects written in Python, Ruby, PHP, Java, Swift, C#, Rust, Golang, and JavaScript, and evaluated the correctness of the SBOMs

by conducting a differential analysis on the outputs from these four tools. Our evaluation revealed that all four SBOM generators exhibit inconsistent SBOMs, and dependency omissions, leading to incomplete and potentially inaccurate SBOMs. Moreover,we introduced a parser confusion attack against these tools, revealing a new attack vector that can conceal malicious, vulnerable, or illegal packages within the software supply chain. To assist in creating more effective SBOM generators, we developed best practices for SBOM generation and a benchmark to facilitate their development, based on our evaluation findings.

Fast and Accurate Disassembly. In order to improve disassembly efficiency, we designed a fast and accurate disassembler called DeepDi based on a graph neural network. DeepDi achieves high efficiency by leveraging the parallelism of GPU, enabling thousands of instruction decodings and inferences in parallel. For accuracy, DeepDi builds a graph representing instruction flow and leverages a relational graph neural network to identify instruction execution paths that most closely resemble normal program behavior. DeepDi not only brings hundreds of times speedup but also makes time-critical tasks such as SBOM generation and malware detection practical in real-world scenarios.

Learning-free Function Matching. Binary function matching is essential for extracting SBOM information from binary code. Existing approaches suffer from either accuracy issues (e.g. learning-based), especially on unseen binaries and large candidates, or scalability issues (e.g. dynamic analysis, graph matching) when processing large binaries. We propose GrassDiff, a novel function matching framework that addresses both accuracy and scalability concerns. The key idea is to leverage our improved Graduated Assignment algorithm (GA+)

to match functions on call graphs. Experimental results show that GrassDiff improves accuracy by 5% to 20% compared to the pure embedding-based approach. Furthermore, it shows good scalability with large binaries, and can pinpoint vulnerable functions with high precision.

Chapter 2

Background

2.1 Software Bill of Materials

An SBOM [44] is a formal, machine-readable inventory of software components and dependencies that includes information about those components and their hierarchical relationships. It can be shared and exchanged automatically among stakeholders (e.g., software vendors and consumers) to enhance software development, software supply chain management, vulnerability management, asset management, and procurement. This results in reduced costs, security risks, license risks, and compliance risks.

SBOM Types: Based on the stages of the software lifecycle at which SBOMs are generated, they can be categorized into six types [51]: Design, Source, Build, Analyzed, Deployed, and Runtime. Depending on what information is available in each stage, these types of SBOMs focus on different aspects. In this chapter, we evaluate Source SBOM, a type of SBOM derived from the development environment. It mainly contains dependencies used for development and compilation, and is widely supported by SBOM tools. Also,

our survey suggests that, owing to its simplicity and precision, metadata parsing is the industry's leading SBOM generation technique. Thus, this chapter focuses on the Source SBOM generated using the metadata-based approach.

SBOM Applications: The increasing complexity and interdependence in software development have amplified the importance of SBOMs. These provide clarity by clearly listing software components, facilitating swift vulnerability tracking and identification for developers and security professionals. Their compatibility with Vulnerability Exploitability eXchange (VEX) [53], a structured database detailing product vulnerabilities, is noteworthy. Additionally, the comprehensive dependency information aids in license assessment, ensuring compliance and mitigating legal exposures. SBOMs enable quality assessment of closed-source software through component reputation checks, and their transparency fortifies the software supply chain by thwarting the introduction of potential backdoors and vulnerabilities via third-party components.

2.2 Metadata

At the heart of Source SBOM generation lies the metadata - an important element in modern software development. These files encapsulate parameters, settings, dependencies, and version constraints, all of which are indispensable for reproducibility and consistent and reliable deployment, and offer support for package management, version control, and even automated build processes. Nowadays, almost every programming language comes with at least one package manager, and each package manager defines its own metadata.

At high level, there are two kinds of metadata. One is "raw" metadata where only direct dependencies are specified and their versions are often given as a range or a constraint instead of a specific (pinned) one. Raw metadata, such as `requirements.txt` for Python and `package.json` for Node.js, are mainly for dependency declaration while ensuring a degree of flexibility and future compatibility. The other type is lockfile such as `package-lock.json` for Node.js. Lockfiles focus on providing a precise and deterministic snapshot of the exact dependency tree including transitive dependencies. Locking prevents unexpected updates or changes in the dependencies when installing the project across different environments, ensuring reproducibility and avoiding compatibility issues. Despite that lockfiles contain the richest information for SBOM generation, they are not always available. Library developers are not encouraged to share lockfiles which could otherwise lead to version conflicts. Some package managers lack a native locking mechanism. Without lockfiles, the missing transitive dependencies and pinned versions pose a great challenge to SBOM tools to generate accurate and complete SBOM files.

2.3 Disassembly Methods

Linear Sweep Disassembly. Linear sweep disassembly is the most straightforward yet fast disassembly method. It disassembles from the beginning of the buffer and assumes there is no data in the buffer, meaning the starting point of an instruction is the ending point of the previous instruction. However, this assumption may not hold as compilers may insert jump tables or strings [68], so the false positive rate and false negative rate can be high, especially for obfuscated binaries. Modern compilers do not place strings in

the code section, but it happens a lot in shellcode. Besides that, the `Microsoft Visual C++ Compiler` and `Intel C++ Compiler` will place jump tables in the code section, adding errors to linear disassembly results.

Recursive Traversal Disassembly. Recursive traversal disassembly can greatly eliminate false positives. It starts from the entry point of a binary file and follows control flow edges. However, it cannot follow indirect jumps or calls, so it may miss quite a number of code blocks. This method is usually combined with some heuristics to detect missing code blocks. Indirect control transfers are very common in complex programs. These programs have switch-case statements, virtual functions, function pointers, etc. Jump tables, such as `jmp dword ptr [addr+reg*4]`, are relatively easy to resolve. However, there exist different variants of jump tables, and some can be difficult to resolve.

These two methods are straightforward and simple, but neither is perfect. IDA Pro has a signature-based approach to scan common patterns of code, others may have dedicated data flow analysis to resolve indirect jumps. Neither is cheap. Code patterns can be affected by compilers, optimization levels, architectures, etc. Therefore, searching in such a large knowledge base is time-consuming. Data flow analysis generally uses an iterative algorithm and requires a lot of computational time. Since the manually-defined heuristics are not complete and slow, we build a machine learning model to automatically capture relations among instructions and use GPU and SIMD instructions in CPU to accelerate the computation.

Superset Disassembly. Superset Disassembly [77] was proposed for binary rewriting. It disassembles every executable byte offset. Figure 4.1 (a) and (b) show an example of

superset disassembly. Although most of the instructions are false positives, all true positives are included in the result so that every possible transfer target can be instrumented during binary rewriting.

Probabilisitc Disassembly. Shingled Graph Disassembly [277] and Probabilistic Disassembly [210] are both probability-based approaches, and they both start from superset disassembly. Shingled Disassembly maintains an opcode state machine that gives a probability of transition from one opcode to another. It removes execution paths with low probabilities (according to the opcode state machine) to find an optimal execution path with a maximum likelihood. Their algorithm runs in $O(n)$ and according to the paper, their approach is two to three times faster than IDA Pro v6.3. Shingled Disassembly also has a similar accuracy compared to IDA Pro and has fewer missing instructions. Probabilistic Disassembly is a recently proposed binary rewriting approach that uses probabilities to model uncertainties (interleaved code and data, indirect transfer targets, etc.). It considers register define-use relations, control flow convergence, control flow crossing, and computes a probability for each address based on these features. Its experiment shows that it has no false negative, and the false positive rate is only 3.7% on average, making it particularly suited for binary rewriting.

Datalog Disassembly. Datalog Disassembly [131] is also a recently proposed binary rewriting approach. Similar to Probabilistic Disassembly, Datalog is based on Superset Disassembly, and it defines a series of rules to remove invalid instructions. For instance, if an instruction falls-through, or jumps, or calls an invalid instruction, this instruction is also invalid. Combined with some heuristics and potential references in data sections, it resolves

overlaps and achieves very high accuracy. The downside though, is that such analyses are expensive and can take a lot of time.

XDA. XDA [226] is a deep learning-based disassembly approach. It takes raw bytes as input, and then randomly masks some of these bytes to learn a language model for instructions. For example, XDA learns `sub rsp` and `add rsp`, a typical function prologue and epilogue, is a pair, which can be used to indicate function boundaries. With this pre-trained language model, one can fine-tune it for various tasks (instruction boundary, function boundary, etc.) with very little training data. XDA also has good accuracy on unseen real-world projects and is robust to different optimizations. However, it has 12 multi-head attention layers and a large hidden size, or 86,838,795 trainable parameters in total, which makes this model very complex and hinders the efficiency benefits brought by GPUs.

2.4 Graph Matching Problem Formulation

Graph matching (GM) seeks to resolve the task of discovering node correspondences among two or multiple graphs. Contemporary GM methods typically incorporate assessments of both node and edge similarities concurrently, thereby striving to optimize the overall similarity between the matched graphs. In contrast, transformer-based approaches, as mentioned earlier, lack second-order edge information. As a result, their matching process relies solely on either the function's intrinsic characteristics or limited local information. Leveraging its expressiveness, GM has found extensive application in computer vision for pattern recognition.

The graph matching problem is commonly reformulated as a Quadratic Assignment Problem (QAP). Initially, an affinity matrix [190] is defined, capturing the similarity between each pair of nodes and edges. Correspondence mappings between two sets of nodes N and N' are represented by a set C consisting of pairs (i, i'), where $i \in N$ and $i' \in N'$. Each potential assignment[1] $a = (i, i')$ is associated with a similarity score or affinity, reflecting the degree of match between node $i \in N$ and $i' \in N'$. Additionally, for every pair of assignments (a, b), where $a = (i, i')$ and $b = (j, j')$, an affinity is computed to assess the compatibility of the edges between nodes (i, j) and (i', j'). To facilitate the storage of these affinities, leveraging a list L of n candidate assignments, we employ an $n \times n$ matrix M. This matrix encompasses the affinities for each assignment $a \in L$ and every pair of assignments $a, b \in L$ as outlined below:

1. $M[a, a]$ is the affinity at the level of individual node assignment $a = (i, i')$ from L. It measures how well the node feature i matches the node feature i'.

2. $M[a, b]$ describes the affinity at the level of edges between the edge (i, j) and edge (i', j'). Here $a = (i, i')$ and $b = (j, j')$.

The objective is to identify a valid set of candidate assignments x^* that maximizes the cumulative affinities resulting from these assignments. To achieve this, a binary vector x of size $n \times 1$, corresponding to L, is employed. Specifically, assignment a is selected if $x[a] = 1$, otherwise it is not chosen. Consequently, our problem can be formulated as follows: given an $n \times n$ affinity matrix, the task is to find a binary vector x that represents

[1] In this thesis, "assignment" and "matching" carry the same meaning.

the selection of candidates, maximizing the objective function:

$$S = x^T M x \tag{2.1}$$

The optimal solution x^* is the binary vector that maximizes the affinity score:

$$x^* = argmax(x^T M x) \tag{2.2}$$

This is commonly known as Lawler's QAP, recognized as a formidable NP-hard problem [200]. Presently, numerous studies employ various relaxation techniques to tackle this challenge, either through learning-free or learning-based methods [147]. Depending on their approaches to relaxation, these methods entail trade-offs between accuracy and efficiency. Consider the significant constraint known as the two-way constraint applied to the matching matrix or the selector vector x, which essentially represents a vectorized form of the matching matrix. This constraint stipulates that each row and column of the matching matrix should only contain a single 1, indicating a one-to-one mapping in the result. However, many existing works [139, 190, 97, 316] tend to relax this stringent, discrete constraint to a continuous one, potentially leading to inaccuracies in the results.

Chapter 3

Metadata-based SBOM Generation

Software Supply Chain Attacks (e.g., SolarWinds [63], PyTorch dependency confusion attack [41]) have increased by 742% between 2019 and 2022 [54]. In 2022 alone, 185,572 software packages were affected by these attacks [4]. The lack of visibility and transparency in the software supply chain makes defending against such attacks challenging. Recently, the Software Bill of Materials (SBOM) [44], a list of "ingredients" used to build software, has demonstrated its efficacy in protecting the software supply chain by enhancing visibility from software development to consumption. Driven by regulations, such as Biden's executive order [18] and the National Cybersecurity Implementation Plan [34], the industry is adopting SBOM-based solutions to safeguard the software supply chain.

An essential step in adopting SBOM is to generate accurate SBOMs. While SBOMs have the potential to enhance vulnerability detection and facilitate license compliance, these benefits can only be realized if the SBOMs themselves are precise and correct. Discrepancies or omissions in the SBOM can lead to false assurances of security or compli-

ance, exposing systems to potential risks. Many SBOM generation tools [50, 49, 32, 23] are extensively used in both commercial and open-source realms. However, the correctness of these tools remains largely unscrutinized. To date, there has not been a systematic study addressing the correctness of contemporary SBOM generation solutions.

Given the diversity of programming languages, build tools, and development practices, constructing a ground truth for SBOM generation evaluation is inherently challenging. In this chapter, we adopt a differential analysis approach: we analyze the discrepancies in SBOMs produced by different tools for the same software to assess both their correctness and weaknesses in SBOM generation. More specifically, we 1) select four popular SBOM generators: Trivy [50], Syft [49], Microsoft SBOM Tool [32], and GitHub Dependency Graph [23]; 2) collect 7,876 open-source projects written in Python, Ruby, PHP, Java, Swift, C#, Rust, Golang and JavaScript; 3) evaluate the correctness of the SBOMs by conducting a differential analysis on the outputs from these four tools.

Surprisingly, our evaluation reveals all four SBOM generators exhibit inconsistent SBOMs and dependency omissions, leading to incomplete and potentially inaccurate SBOMs. Moreover, we construct a parser confusion attack against these tools, introducing a new attack vector to conceal malicious, vulnerable, or illegal packages within the software supply chain. To assist in creating more effective SBOM generators, we have developed best practices for SBOM generation and a benchmark to facilitate their development based on our evaluation findings.

In summary, we make the following contributions:

- We are the first to conduct a large-scale differential analysis to examine the correctness of SBOM generation solutions.
- Our evaluation reveals significant deficiencies in current SBOM generators. We also conduct a comprehensive case study to uncover how each SBOM tool detects dependencies during the generation process.
- We construct a parser confusion attack against SBOM generators, introducing a new attack vector to inject malicious, vulnerable, or illegal software packages into the software supply chain.
- We develop best practices for developing SBOM generators and a benchmark to facilitate their development.

3.1 Methodology

Despite the growing significance and adoption of SBOMs, a notable gap exists in systematically assessing the quality of the SBOM files generated. The reliability of security-centric applications, including vulnerability detection and license compliance, highly depends on the correctness of SBOM data, which raises concerns regarding the trustworthiness of such information.

This work aims to investigate the correctness and completeness of the dependency information present in generated SBOMs. The objective is to not only measure the correctness but also to unravel the underlying factors contributing to high-quality SBOMs. Due

to the lack of ground truth, we adopt a differential analysis approach to obtain insights into the performance of SBOM generators.

3.1.1 SBOM Generators

In this work, we evaluate four SBOM tools: `Trivy` 0.43.0, `Syft` 0.84.1, `Microsoft SBOM Tool` (sbom-tool) 1.1.6, and `GitHub Dependency Graph` (GitHub DG). Notably, the first three are popular open-source projects and offer cross-platform support for Linux, Windows, and Mac operating systems. Conversely, the GitHub Dependency Graph is intricately integrated with GitHub repositories. We choose Trivy and Syft because they are the de facto SBOM generators used by industries and open-source communities. We pick the Microsoft SBOM Tool because it is developed by the esteemed Microsoft. Similarly, the GitHub Dependency Graph is chosen because it is provided by the most widely used Git platform. All the evaluated SBOM tools implement metadata-based approaches, meaning they read metadata files and extract dependency information declared in the metadata files.

3.1.2 Setup

The evaluation was conducted by downloading popular GitHub repositories associated with each programming language onto the local file system and subsequently scanning the repository directories using the SBOM tools. Each tool will generate an SBOM report in either CycloneDX [36] or SPDX [27] format depending on which format is supported by the tools. Dependencies in these reports are then extracted and compared against each other.

Dataset: GitHub repositories were sourced from the well-regarded `awesome-LANGUAGE` repositories, which are uniquely tailored to the respective programming languages. Our dataset contains 535 Python, 819 Ruby, 384 PHP, 398 Java, 1,019 Swift, 700 C#, 994 Rust, 2,367 Golang, and 660 JavaScript repositories. We do not evaluate C/C++ projects due to the absence of an "official" build toolset and extremely limited support provided by the SBOM tools. C/C++ projects can be configured and built via various tools such as Bazel, Makefile, CMake, Visual Studio project files, and more. Consequently, Trivy and Syft only analyze `conan.lock`, while GitHub Dependency Graph exclusively focuses on `*.vcxproj` files.

Metrics: For our large-scale evaluation, given the absence of ground truth, we adopt a differential analysis approach. First, we compare the number of dependencies reported by each SBOM tool. We then use Jaccard similarity to measure the reported dependency names. This tells us the degree of overlap and commonality among the dependencies reported by different tools. In addition, we identify duplicate packages reported by the SBOM tools. While these metrics may not provide a direct ranking, they do shed light on the performance of these tools.

3.2 Large-scale SBOM Comparison

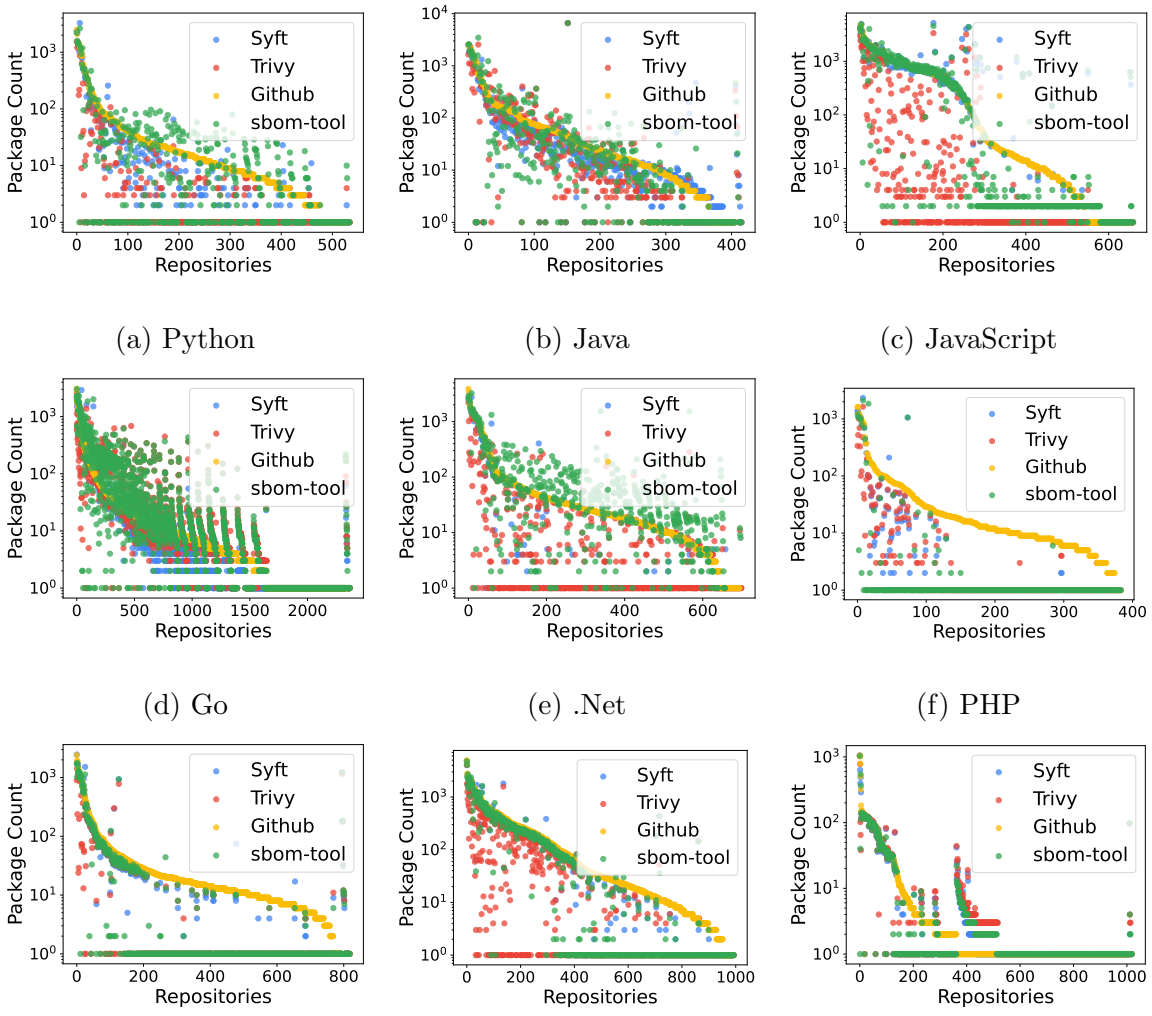
After analyzing 7,876 high-quality repositories, we made the following major findings. The reasons behind such discrepancies will be discussed in Section 3.3.

3.2.1 Discrepancies in Package Counts within SBOM Reports Generated by Different Tools

The SBOM tools exhibited notable differences in the number of packages they identified. Figure 3.1 clearly depicts this variation. The x-axis is the repository ID sorted by the number of dependencies detected by the GitHub Dependency Graph. For Python, PHP, Ruby, and Rust programming languages, GitHub Dependency Graph discovers the most packages for these languages. For .Net repositories, Microsoft SBOM Tool excelled in identifying the most packages, which is unsurprising as it is tailored to Microsoft's own projects. For the Go and Swift languages, Trivy and Microsoft SBOM Tool proved to be the frontrunners, consistently identifying the most packages in the majority of cases. Syft excels in detecting the highest number of packages when it comes to JavaScript repositories. The disparities presented in this figure underscore that different tools possess varying capabilities and strategies in identifying dependency packages across different programming languages. It is important to note, however, that identifying more packages does not mean better because false positives may also be included.

3.2.2 Low Package Jaccard Similarities

To measure whether the SBOM tools detect similar dependencies for each repository, we compute a Jaccard similarity for each SBOM tool pair for each repository as Equation 3.1 shows. A and B are two sets of dependencies generated by two different

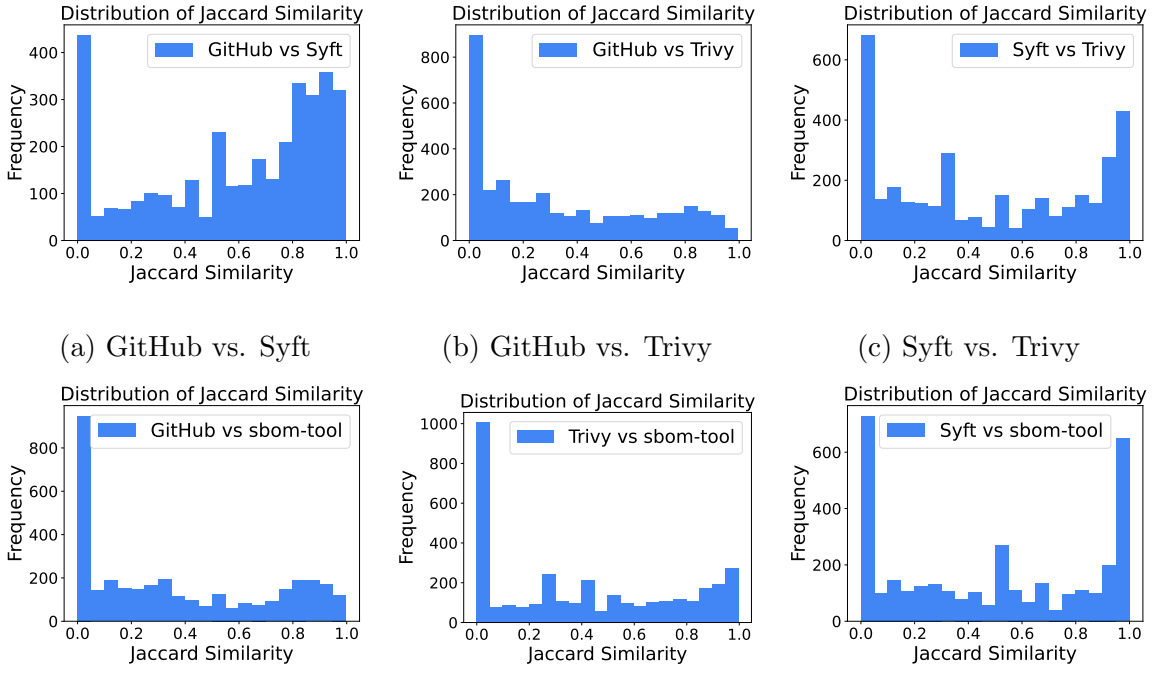


(a) Python (b) Java (c) JavaScript

(d) Go (e) .Net (f) PHP

(g) Ruby (h) Rust (i) Swift

Figure 3.1: Comparison of Package Counts Across Languages Using Various SBOM Generators



(a) GitHub vs. Syft (b) GitHub vs. Trivy (c) Syft vs. Trivy

(d) GitHub vs. sbom-tool (e) Trivy vs. sbom-tool (f) Syft vs. sbom-tool

Figure 3.2: Distribution of Jaccard Similarity among Various Tools

SBOM tools. Each set contains dependency $(name, version)$ pairs.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{3.1}$$

Our evaluation result is illustrated in Figure 3.2. The majority of these pairs show significant dissimilarity, with only a very small portion being similar. As shown in (a), the GitHub Dependency Graph and Syft have the most similarities among them, although the majority of SBOM reports still exhibit substantial differences.

3.2.3 Duplicate Packages in SBOMs

During our analysis of the generated SBOMs, we identified instances of duplicate packages: the same package appearing in different entries with varying or the same version

requirements. To ensure accurate calculations, we excluded repositories in which tools could not find any packages.

In Table 3.1, we have presented the rate of duplicate packages for various SBOM tools. This problem was found to be widespread across all four tools, suggesting a common occurrence. However, it is important to note that having duplicate packages is expected in some cases. For example, a repository may contain multiple independent projects and they happen to have a common subset of dependencies.

Table 3.1: Rate of Duplicate Packages in SBOMs

	Syft	**Trivy**	**GitHub DG**	**sbom-tool**
Python	14.05%	12.56%	13.54%	13.71%
Java	12.76%	15.01%	19.93%	18.89%
JavaScript	17.46%	17.34%	18.89%	19.42%
Go	9.97%	6.69%	11.03%	6.58%
.NET	17.38%	12.43%	18.01%	20.94%
PHP	13.76%	11.77%	14.53%	23.76%
Ruby	13.56%	9.1%	15.84%	12.39%
Rust	13.19%	11.37%	19.18%	13.83%
Swift	1.37%	2.28%	6.98%	3.39%

3.3 SBOM Generation Analysis

To uncover the root causes behind the large disparities in SBOM outputs, we conducted an in-depth analysis of the source code of the SBOM tools. Our examination revealed several critical issues in SBOM generation, which are summarized below.

3.3.1 Limited Support for Metadata

All the evaluated tools employ a metadata-based approach where they analyze metadata to identify the components used in the project. The supported metadata file types for each tool are detailed in Table 3.2. It is important to note that the table indicates the tools' actual capability to extract dependencies from metadata, which may differ from their claims.

The table illustrates that each tool supports only a subset of commonly used metadata files. Overall, the SBOM tools have good support for lockfiles in which transitive dependencies and pinned versions are available, but they struggle with raw metadata. The GitHub Dependency Graph has the best support for raw metadata such as `Gemfile` and `Cargo.toml`, while other tools show limited or no support for raw metadata. Despite claims by Trivy and Syft to support `package.json`, they do not extract dependencies from the JSON file. In our evaluation, we found that 93% of Python repositories, 47% of JavaScript repositories, and 56% of Rust repositories contain raw metadata only.

3.3.2 Incomplete Metadata Parsing

Our evaluation shows that all the evaluated SBOM tools implement custom parsers for metadata. However, certain metadata, like `requirements.txt` defined in PEP 508, poses challenges due to its complex syntax. The self-implemented parsers only support common syntaxes, leading to false negatives. For instance, the lack of support for the backslash "\" as a line continuation in all the SBOM tools causes parsing errors, resulting in incorrect versions or missed dependencies. About 1.8% of Python repositories are affected by this.

3.3.3 Transitive Dependency

The offline nature of SBOM tools (except Microsoft SBOM Tool) implies a lack of attempts to resolve transitive dependencies. In the case where lockfiles are not present, the absence of transitive dependencies will adversely affect SBOM applications. Microsoft SBOM Tool attempts to resolve transitive dependencies by querying package managers for each detected dependency, but this functionality is not well-implemented and often fails to retrieve dependency information from package managers. About 74% of Python dependencies are transitive dependencies.

3.3.4 Limited Support for Version Constraints

Raw metadata often contains version ranges or constraints instead of pinned versions; for example, developers use `>=1.2.3 <2.0.0` to get the latest version while ensuring backward compatibility. Trivy and Syft handle version constraints by silently discarding

dependencies without pinned versions, resulting in false negatives. The GitHub Dependency Graph reports version ranges as they appear in the metadata, introducing additional parsing challenges for SBOM management. In our evaluation, only 46% of dependencies declared in `requirements.txt` have pinned versions, indicating that Trivy and Syft may miss more than half of the dependencies even when transitive dependencies are not considered. Microsoft SBOM Tool addresses this by pinning a version after querying the corresponding package manager for the latest version within the specified range.

3.3.5 Inconsistent Package Naming Convention

When dealing with packages having compound names, SBOM tools name them differently. For Java, a package is located using the group ID and artifact ID. Syft uses the artifact ID as the package name, Microsoft SBOM Tool concatenates the group and artifact ID with a dot "." as the package name, while Trivy and the GitHub Dependency Graph use a colon sign ":" for this purpose. Similarly, Swift package manager CocoaPods supports subpecs when declaring a dependency. Subspecs are a way of chopping up the functionality of a library, allowing people to install a subset of the library. Syft and Trivy report the subspecs, while Microsoft SBOM Tool reports their main dependency names. Additionally, Golang uses a leading letter "v" when specifying versions (e.g., v1.0.0). Syft and Microsoft SBOM Tool adhere to this convention, while Trivy and the GitHub Dependency Graph omit this leading letter. Such inconsistencies can potentially compromise the accuracy of vulnerability detection.

3.3.6 Different Dependency Definition

SBOM tools employ different strategies regarding whether to include development dependencies (e.g., test suites, linters, etc.) in SBOM files. Trivy focuses solely on production dependencies and ignores development dependencies, whereas Syft and GitHub Dependency Graph include both types. Our evaluation reveals that in JavaScript, 76% of dependencies declared in `package.json` are development dependencies. It is crucial to note that there is no definitive answer regarding which approach is better. Including development dependencies in the SBOM report offers several advantages, such as more comprehensive vulnerability assessments and license violation checks, but it may also introduce false alarms as the code of development dependencies rarely goes into the final product. The root problem lies in the absence of an existing field in SBOM formats representing the dependency scope. While most metadata have distinct fields for this purpose, such as the `scope` field in `pom.xml` and the `devDependencies` in `package.json`, the current SBOM formats lack this support and may cause confusion in downstream applications.

3.3.7 Multiple Projects and Metadata

Our evaluation indicates that, on average, over 10% of the detected dependencies appear more than once in a repository, causing duplicate entries in SBOM files. This is primarily due to multiple metadata files present in a repository, either because of having multiple subprojects or submodules or having both raw metadata and lockfiles present. The SBOM tools analyze metadata individually without merging dependencies in the same project. Duplicate entries in SBOMs can lead to confusion and potentially inflate the

apparent package count. Our evaluation shows that there are 5.7 metadata files in a Python repository and 12.8 metadata files in a JavaScript repository on average.

3.3.8 Accuracy on Ground Truth

Our large-scale evaluation employed a differential analysis due to the lack of ground truth. In this section, we quantify the accuracy of each SBOM tool on `requirements.txt` using our manually crafted ground truth. The ground truth is obtained by dry-running `pip install` (Python 3.11, pip 23.1.2), and we consider a correct dependency $(name, version)$ pair as a correct match. Dry run simulates the installation process and the dependencies reported by `pip install` are those that will be installed in our environment. This evaluation aims to highlight the differences between the reported libraries and the ones actually installed.

The evaluation result is presented in Table 3.3. Most SBOM tools fail to detect over 90% of the dependencies in `requirements.txt` due to incomplete syntax support and the lack of transitive dependency resolution. The Microsoft SBOM Tool excels in this test because it attempts to resolve transitive dependencies, but it ignores the `extras` field, and OS and Python requirements. The low recall suggests that relying solely on these SBOM tools in practice may have serious negative impacts on downstream applications, such as vulnerability detection and license violation checks.

3.4 Parser Confusion Attack

Motivated by the findings in Section 3.3.8, we present a parser confusion attack [91] to illustrate how adversaries can obscure malicious dependencies. A parser confusion attack exploits inconsistencies among different parsers processing the same input, enabling malicious actors to craft input that is benign for one parser but harmful for another. Our case study shows that SBOM tools, employing custom metadata parsers, introduce a new attack vector for constructing parser confusion attacks within the SBOM ecosystem. In this study, we use Python's `requirements.txt` as an illustrative example.

Constructing the attack: Given that `requirements.txt` lacks a locking mechanism and exhibits a rich syntax, it becomes a suitable candidate for this type of attack. For instance, none of the SBOM tools support the backslash as a line continuation; Trivy and Syft rely on the double-equal sign to separate package names and versions; installations from wheel packages are not universally supported; and many more. Table 3.4 provides some input patterns that can be used to bypass detections based on our manual analysis and benchmark (discussed in Section 3.5). It shows how attackers can leverage different syntax elements to either conceal specific dependencies or confuse SBOM tools, leading to inaccurate results. In the table, a dash ("-") signifies that the corresponding SBOM tool cannot detect anything from the given dependency declaration.

Achieving Damage: When the SBOM tools encounter unsupported syntax, the default behavior is to silently ignore the associated dependency. Adversaries can exploit this and inject malicious or vulnerable dependencies in metadata using unsupported syntax, effec-

tively evading the tools' detection entirely. In our dataset, the two most common patterns are installing from other requirement files (`-r`) and installing from version control systems, each appearing in over 50 `requirements.txt` files.

3.5 Best Practice and Benchmark

Drawing from our evaluation, we present what we believe are the most optimal solutions to address identified issues and minimize the attack surface. We propose the following best practices for metadata-based approaches:

Package Manager Dry Run for Lockfile Generation: The root cause of the large discrepancies lies in the limitations of self-implemented parsers, particularly in their support for metadata and metadata syntax. Instead of relying on these parsers, we recommend employing a package manager dry run to generate lockfiles. This simulates the dependency installation process, providing both transitive dependencies and accurate version information for each package. Adopting this approach ensures the creation of a precise and reliable SBOM file, thereby enhancing resilience against confusion attacks.

PURL and CPE Support: Each dependency should include a PURL (Package URL) entry and a CPE (Common Product Enumerator) entry for consistent package naming convention, maximum compatibility with vulnerability databases, and facilitate software identification.

Our evaluation benchmark is available on GitHub. This benchmark includes manually crafted metadata files and ground truth datasets for common languages. These metadata files try to cover all supported syntaxes for each language, and can be used to evaluate

of the SBOM tools' capability to handle corner cases. This initiative aims to guide the development of SBOM tools, emphasizing completeness and accuracy. We are working on adding support for more programming languages.

3.6 Conclusion

In this chapter, we conducted the first large-scale differential analysis to examine the correctness of SBOM generation solutions. We generated SBOMs using four popular SBOM generators for 7,876 open-source projects and systematically studied the correctness of these SBOMs. Our evaluation uncovered significant deficiencies in current SBOM generators. Additionally, we identified the design flaws in each SBOM generator, and devised a parser confusion attack against these generators, introducing a new path for injecting malicious, vulnerable, or illegal packages. Finally, based on our findings, we established best practices for creating SBOM generators and introduced a benchmark to aid their development.

Table 3.2: Supported File Types

		Trivy	**Syft**	**sbom-tool**	**GitHub DG**
Go	go.mod	✓	✓	✓	✓
	Go executable	✓	✓	✗	✗
Java	pom.xml	✓	✓	✓	✓
	gradle.lockfile	✓	✓	✓	✓
	MANIFEST.MF	✓	✓	✗	✗
	pom.properties	✓	✓	✗	✗
JS	package.json	✗	✗	✗	✓
	package-lock.json	✓	✓	✓	✓
	yarn.lock	✗	✓	✓	✓
	pnpm-lock.yaml	✗	✓	✓	✗
PHP	composer.json	✗	✗	✗	✓
	composer.lock	✓	✓	✗	✓
Python	requirements.txt	✓	✓	✓	✓
	poetry.lock	✓	✓	✓	✓
	pipfile.lock	✓	✓	✓	✓
	setup.py	✗	✗	✗	✓
Ruby	Gemfile	✗	✗	✗	✓
	Gemfile.lock	✓	✓	✓	✓
	.gemspec	✓	✓	✓	✓
Rust	Cargo.toml	✗	✗	✗	✓
	Cargo.lock	✓	✓	✓	✓
	Rust executable	✓	✓	✗	✗

Table 3.3: SBOM Accuracy on `requirements.txt`

	Trivy	**Syft**	**sbom-tool**	**GitHub DG**
Precision	0.25	0.25	0.74	0.13
Recall	0.10	0.10	0.73	0.08

Table 3.4: requirements.txt Attack Samples

	Trivy	**Syft**	**sbom-tool**	**GitHub DG**
requests [security]>=2.8.1	-	-	-	-
numpy \ ==\ 1.19.2	-	-	numpy 1.25.2	-
-r SOME_REQS.txt	-	-	-	-
./path/to/local_pkg.whl	-	-	-	-
https://remote_pkg.whl	-	-	-	-
urlib3 @ git_link@hash	-	-	-	-

Chapter 4

DeepDi: Fast and Accurate Binary Disassembly and Function Identification

A disassembler takes a binary program as input and produces disassembly code and some higher-level information, such as function boundaries and control flow graphs. Most binary analysis tasks [176, 112, 251, 237] take disassembly code as input to recover syntactic and semantic level information of a given binary program. As a result, disassembly is one of the most critical building blocks for binary analysis problems, such as vulnerability search [127, 283], malware classification [157], and reverse engineering [253].

Disassembly is surprisingly hard, especially for the x86 architecture due to variable-length instructions and interleaved code and data. As a result, a simple linear sweep

approach like objdump[1] or Capstone[2], despite high efficiency, suffers from low disassembly correctness on Windows binaries and binaries compiled by the Intel C++ Compiler (where jump tables are placed in the code section), and can be easily confused by obfuscators. There has been a long history of research on improving disassembly accuracy. For instance, the recursive disassembly identifies true instructions by following control transfer targets. It largely eliminates false instructions but may miss true instructions that are not reached by other code blocks, leading to a low true positive rate. Commercial disassemblers like IDA Pro and Binary Ninja employ linear sweep and recursive traversal along with undocumented heuristics to achieve high disassembly accuracy, at the price of low runtime efficiency. Our experiments show that IDA Pro can only process approximately 72 KB/s, and Binary Ninja 11 KB/s.

Recently, researchers have explored various novel approaches to further improve the disassembly accuracy, such as probabilistic inference [210, 277], static program analysis [239], logic inference [131], and deep learning [226]. However, the improved accuracy often comes at the price of even lower runtime efficiency. For instance, Probabilistic Disassembly [210] can only process about 4 KB/s, Datalog Disassembly [131] 4 – 50 KB/s. Even worse, XDA [226], based on an expensive BERT [111] model, when running on CPU, can only process 140 B/s according to our evaluation.

Despite the importance of disassembly, we still do not have a disassembler that is both accurate and fast to support downstream binary analysis tasks. This is especially true when dealing with malware, which is often obfuscated to thwart disassemblers for evasion.

[1] https://www.gnu.org/software/binutils/manual/
[2] http://www.capstone-engine.org/

In this chapter, we present a novel deep learning-based disassembler called DeepDi, which can achieve high accuracy and efficiency simultaneously. It can be further accelerated on GPU to gain hundreds of times speedup. In order to achieve high efficiency, DeepDi takes a very different approach than XDA [226] to leverage deep learning. Instead of feeding raw bytes as input to an expensive deep learning model as done in XDA, DeepDi first decodes all possible instructions and converts them into high-level feature vectors, and then identifies true instructions from all instruction candidates by constructing logical relations (e.g., one instruction followed by another, one instruction overlapped with another, etc.) between these instruction candidates and performing graph inference on them. In particular, we use a Relational Graph Convolutional Network (Relational-GCN) [248], because it can capture different kinds of relations between nodes and it is small and efficient. After supervised training, our model is able to identify true instructions. From these identified true instructions, DeepDi then recovers function entrypoints from the true instructions using heuristics and a simple classifier.

We have conducted extensive experiments to evaluate DeepDi with respect to accuracy, efficiency, generalizability, and robustness. To evaluate the accuracy, we use four datasets (i.e., BAP corpora [86], LLVM 11 on Windows[3], SPEC CPU2006 [47], and SPEC CPU2017 [48]), and compare with five disassemblers (i.e., IDA Pro [25], Binary Ninja [8], Ghidra [22], Datalog Disassembly [131] and XDA [226]). Experimental results show that DeepDi is comparable or superior to these disassemblers in terms of accuracy on regular binaries. For efficiency, the single-core CPU version of DeepDi can achieve a throughput of 146 KB/s, which is two times faster than commercial disassemblers. A

[3]https://github.com/llvm/llvm-project

CUDA implementation of DeepDi can further improve the throughput by 170 times on a modest GPU, reaching 24.5 MB/s, which is 350 times faster than IDA Pro. To evaluate its generalizability, we first train our model with BAP corpora on each optimization, and evaluate on LLVM 11 to show the performance on unseen binaries compiled with different compilers and on a different platform. The result shows that our instruction precision and recall are at least 97.1%. We use the model for the accuracy test and test it on ten unseen real-world software to show the performance on real-world binaries, and the result is comparable with XDA. For robustness, we evaluate the performance on obfuscated binaries provided by Linn and Debray [197] and some real-world binaries obfuscated by Hikari [314]. Our model achieves 84.1% precision and 95.2% recall within 1.2 seconds in the first test, whereas XDA and IDA Pro take over 200 seconds and are less accurate. In the second test, our model has very consistent performance on five different obfuscation techniques, and is several orders of magnitude faster than the other disassemblers.

We further demonstrate how DeepDi is used in malware classification. We use the malware dataset from Microsoft Malware Classification Challenge [244], and extend Gemini [283] and EMBER [67] to use high-level features for malware classification. Our evaluation shows our Gemini model can achieve 98.2% training accuracy and beat MalConv [240] in testing loss value. The extended EMBER model achieves 99.5% training accuracy and beats the original EMBER. While the traditional feature extractions take hours and even days on this dataset, ours only takes 9 minutes in Gemini and 3 minutes in EMBER, showing the capability of classifying malware accurately and efficiently. We provide a binary release of DeepDi at https://github.com/DeepBitsTechnology/DeepDi.

Contributions. In summary, we make the following contributions:

- We design a novel deep learning-based disassembler that can achieve accuracy and efficiency simultaneously. It exemplifies how a deep learning-based system can substantially improve the efficiency and accuracy over the existing approaches.
- We propose a novel graph representation called "Instruction Flow Graph" to model different relations between instructions. We then use a Relational-GCN to perform inference and classification on an Instruction Flow Graph to classify instructions accurately.
- We conduct extensive experiments to show the practical application value of DeepDi. Experimental results show that DeepDi is comparable or superior to the state-of-the-art disassemblers in terms of accuracy. DeepDi is also robust against unseen compilers and platforms, obfuscated binaries, and adversarial attacks. Its efficiency is several orders of magnitude higher than the baseline approaches.
- We showcase malware classification as a downstream application for DeepDi. We show that DeepDi can enable fast and accurate malware classification by providing high-level features efficiently.

4.1 Design

We envision a good disassembler should achieve the following design goals:

- **High Accuracy.** It should correctly identify instructions and functions with very high recall and precision.

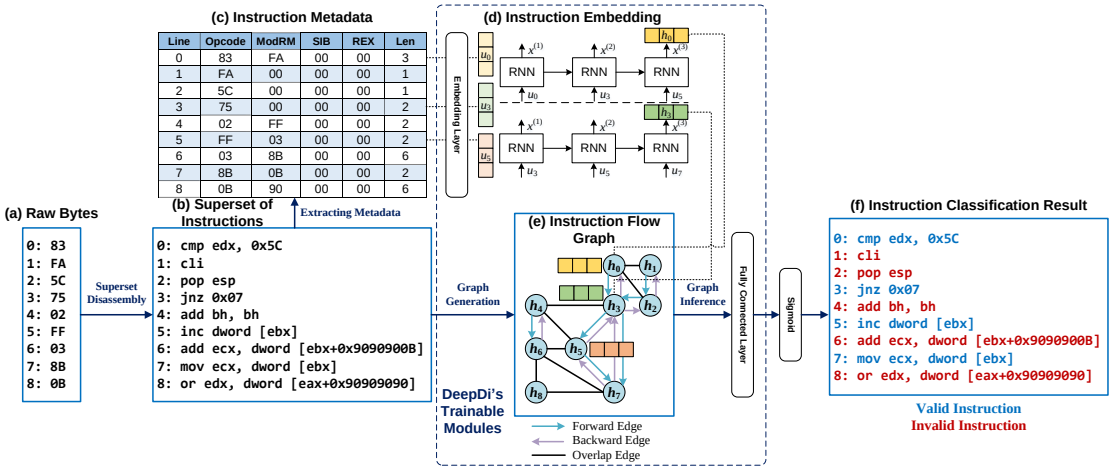


Figure 4.1: Overview of DeepDi with a Concrete Example

- **High Efficiency.** It should disassemble a binary program at a very high speed, without compromising accuracy.
- **Reasonable Robustness.** While it is impossible to achieve complete robustness against strong adversaries that can be explicitly designed against a disassembler, a good disassembler should be resilient to common obfuscations such as junk code and computed jumps.
- **Support for Downstream Tasks.** In addition to identifying instructions and functions, a good disassembler should provide auxiliary information like call graph, control follow graph, etc., which is useful for downstream analysis tasks.

Figure 4.1 serves as an overview and a running example of DeepDi. Our approach first uses superset disassembly to disassemble raw bytes. According to the disassembled instructions, we build an instruction flow graph (IFG) representing all possible execution paths. Each instruction is also converted to a feature vector via instruction embedding while

Table 4.1: Comparison of Disassembly Approaches

Method	Pros	Cons	Efficiency[1]	
			CPU	GPU
Traditional Approaches	Close to 100% accuracy on regular files	Slow and vulnerable to obfuscation	10 – 200 KB/s	N/A
Superset Disassembly [77]	Very fast and no false negative	85% false positive [210]	4 – 5 MB/s	1+ GB/s
Shingled Graph Disassembly [277]	Similar accuracy to IDA Pro and 2x faster	Small dataset and not open source	70+ – 200 KB/s	N/A
Probabilistic Disassembly [210]	No false negative	3% false positive and slow	4 KB/s	N/A
Datalog Disassembly [131]	Nearly 100% accuracy	Slow and limited file format support	4 – 50 KB/s	N/A
XDA [226]	Close to 100% accuracy	Slow	140 B/s	47 KB/s
DeepDi (this work)	Close to 100% accuracy	–	146 KB/s	24.5 MB/s

[1] Measured on our server, please refer to Section 4.2.1 for more details.

maintaining its semantic meaning. The feature vectors are propagated on the IFG using an R-GCN model to obtain neighboring information, and then are fed into a classification layer to predict whether the corresponding instructions are valid. All the aforementioned layers are connected and are trained in an end-to-end supervised fashion.

Moreover, we further leverage the prediction results to recover function entrypoints (not shown in Figure 4.1). We treat instructions that are not reachable by non control transfer instructions as function candidates. We then train a classifier to identify true function entrypoints from the candidates.

4.1.1 Superset Disassembly

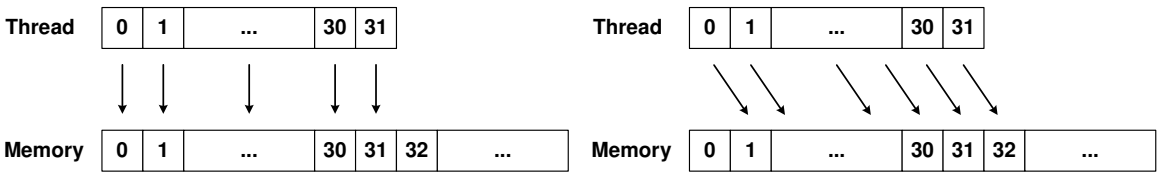
We use Superset Disassembly [77] to ensure our input to the model is a superset of true instructions. Given N raw bytes $b_{0,1,\ldots,N-1}$, the output of superset disassembly is as follows:

$$t_i = D(b_{i,\ldots,i+14}), \forall i \in \{0, \ldots, N-1\} \tag{4.1}$$

where $D(\cdot)$ disassembles the given bytes and each t_i is an $(Opcode, ModRM, SIB, REX)$ tuple. We call this tuple instruction metadata. We feed 15 consecutive bytes (as shown in Equation 4.1) because an instruction is composed of up to 15 bytes. If the rest of the bytes are less than 15, we will pad them with `0x90` (`nop`). A decoded instruction may have prefixes, Opcode, ModRM, SIB, Displacement, and Immediate [26], but we only use REX prefix, Opcode, ModRM, and SIB as its semantic representation, because displacement and immediate contain arbitrary values and do not affect the semantic meaning.

Figure 4.1 (c) shows an example of t_k and how the tuple is represented. Note that although an instruction often takes more than one byte, superset disassembly will still disassemble from its next byte to obtain all possible instructions, which forms a superset of instructions.

Since disassembling any instruction is independent, this process can be easily parallelized on GPU: given n raw bytes, we simply create n GPU threads, and thread i disassembles from b_i [186]. A modern GPU can schedule over one billion threads, so doing so will not cause performance issues.



(a) Time 0 (b) Time 1

Figure 4.2: GPU Disassembly State at Different Time

Figure 4.2 illustrates an example of data parallelism on GPU. Assuming the address of the first instruction byte is 0, we assign thread 0 to 31 (a warp) to disassemble instructions starting at memory location 0 to 31. At time 0, all threads consume one byte at location 0 to 31 accordingly at the same time. At time 1, some threads may turn inactive because they encounter 1-byte instructions and remain inactive until all threads in this warp finish disassembling their instructions. Other threads consume the next bytes, which are memory location 1 for thread 0, 2 for thread 1, and so on. The number of threads we create is the same as the number of bytes in the code section, and each thread will output one instruction.

We make each thread in a warp disassemble a consecutive memory location because of GPU global memory coalescing. When threads in a warp access an aligned and consecutive memory location, this is a coalesced access and GPU can fetch up to 32 words in one memory transaction. If the memory accesses were strided (for example, greater than 31 words), each memory transaction would fetch only one word, wasting almost 97% of memory bandwidth.

When GPU is not available, we can perform this task on CPU, which is very straightforward. We just need to go over one byte at a time and disassemble one instruction starting from it. We can exploit multi-threading on CPU by creating multiple threads, each of which sweeps through one chunk of the input binary.

4.1.2 Instruction Embedding

After we get the superset of instructions, we would like to use the R-GCN model to infer the true instructions. First, we need to decide what kind of representation of each

instruction should be fed into the R-GCN model (the representations are used as the node features in the R-GCN model). In this section, we introduce how we construct instruction representations from their metadata t_i, as shown in Figure 4.1 (d).

The metadata t_i, i.e., the $(Opcode, ModRM, SIB, REX)$ tuple, is integer-encoded, so we first convert it to a fixed-dimensional embedding via a learnable embedding layer, then incorporate the embeddings of an instruction and its *following instructions* into the instruction representation (feature vector) via a recurrent neural network (RNN). Note that Figure 4.1 (c) shows the original values of Opcode, ModRM, SIB, and REX extracted from instructions. However, their value ranges may overlap (e.g., the range of ModRM and SIB is $\{0, \ldots, 255\}$) and it will confuse the embedding layer. So we add a constant value to Opcode, ModRM, SIB, and REX to make their ranges non-overlapping. In total we have 1,025 distinct opcodes, 257 ModRM, 257 SIB, and 17 REX. Each field has a reserved value which is used when the corresponding field is not presented. This makes the overall input size of the embedding layer 1,556. We use an instruction sequence instead of a single instruction because one instruction carries too little information to tell if it is valid. Take Figure 4.1 (b) for example, instruction 4 alone looks valid. However, if we also consider its following instruction, instruction 6 where `ebx` is used as a base register, the modification of `bh` in instruction 4 becomes suspicious. In this way, the same instruction in different execution paths can have different semantic representations, and the context-aware representations can help improve the classification accuracy. In our experiment, two following instructions can give enough information and will not cause much runtime penalty.

Formally, we define the instruction i's feature vector as follows:

$$x_i^{(n)} = f(x_i^{(n-1)}, u_{i\oplus(n-1)}), n = 1, \ldots, M \tag{4.2}$$

where f is the vanilla RNN's recurrent function [245], $x_i^{(n)} \in \mathbb{R}^{d_2}$ is the hidden state of the RNN network ($x_i^{(0)}$ is an all-zero vector, which is the initial hidden state of the RNN). M is the sequence length, which is three in this chapter. $u_i \in \mathbb{R}^{4 \cdot d_1}$ is the embedding of t_i generated by a learnable embedding layer. Each item in the tuple is treated as a word index and the embedding layer converts it to a d_1-dimensional vector. u_i is the concatenation of the embeddings of the four items (Opcode, ModRM, SIB, REX). For an instruction i in the superset of instructions, we define that the operation $i \oplus j$ represents finding j-th non-overlap following instruction of i. Take Figure 4.1 (d) for example, for instruction 0, $0 \oplus 1 = 3$, $0 \oplus 2 = 5$, etc. If $i \oplus (k+1)$ does not exist (out of bound or instruction $i \oplus k$ being invalid), we define $i \oplus (k+1) = i \oplus k$.

Since we define $M = 3$ in this chapter, a simpler unrolled RNN equation of length three can be defined as follows:

$$x_i^{(3)} = f_{unrolled}(u_i, u_{i\oplus1}, u_{i\oplus2}) \tag{4.3}$$

Since only the RNN steps cannot be parallelized, a small sequence length means it would not be particularly more expensive. That is why our approach can still achieve high efficiency even though an RNN is used.

After the RNN module, we can use $x_i^{(M)}$ (in this chapter, $M = 3$) as the representation of instruction i and then feed this representation as the node feature into the R-GCN model for graph inference (see Section 4.1.4 for more details). We chose the vanilla RNN over GRU or LSTM for better efficiency.

4.1.3 Instruction Flow Graph

Since we are exhaustively disassembling binaries, there exist many false instructions. Even worse, instructions are variable-length, thus the model cannot easily determine where the true instructions are. To help the model better understand the contexts, we propose to model different relations between instructions using a graph called Instruction Flow Graph (IFG), which is used with the Graph Inference phase to propagate information of each instruction to its neighbors and to classify true instructions.

Formally, we define an instruction flow graph as a directed graph $G = (V, E, R)$. For each node $v_i \in V$, there is a feature vector x_i, a semantic representation of the instruction obtained from Section 4.1.2. Each edge $(v_i, r, v_j) \in E$ is labeled with a relation $r \in R$ denoting the edge type. $R = \{f, b, o\}$ represents three types: **f**orward, **b**ackward, and **o**verlap, respectively. If the label r in (v_i, r, v_j) is a forward relation, it means the next instruction of i can be j, either i falls through to j, i calls j, or i jumps to j. For example, if the instruction i is a conditional jump that may fall through to j or jump to k, there is a forward edge from i to j and a forward edge from i to k. If instruction i is a return instruction or an indirect jump/call, no forward edge from i is created since the transfer target is unknown. A forward edge from i to j is the same as a backward edge from j to i. If r is an overlap relation, it means instruction i and j overlap with each other. That is, the starting point of instruction j is inside instruction i, or vice versa. These different relations can help the model propagate different kinds of information.

Figure 4.1 (e) shows an example of an Instruction Flow Graph. For instance, Node 3 has two forward relations because Instruction 3 is a conditional jump and thus has two

potential targets. Likewise, Node 0 has two overlap relations with Node 1 and 2 because the length of Instruction 0 is three.

4.1.4 Graph Inference

For our graph inference, we use a Relational-GCN (R-GCN) [248] to propagate information of each instruction to its neighbors. In this network, nodes can have different kinds of relations so that we can pass different messages along different relations. Recall that a valid instruction makes its successors valid, but not vice versa because it can have multiple predecessors, and only one of them or even none of them is valid. R-GCN is capable of modeling this and increases the likelihood of valid instructions while decreases the likelihood of invalid instructions.

As defined in Section 4.1.3, an instruction flow graph is denoted as (V, E, R). We use the following propagation model to update the hidden state of each node v_i in each layer:

$$h_i^{(l+1)} = ReLU\left(\sum_{r\in R}\sum_{j\in N_i^r}\frac{1}{|N_i^r|}W_r^{(l)}h_j^{(l)} + W_0^{(l)}h_i^{(l)}\right) \tag{4.4}$$

where $h_i^{(l)} \in \mathbb{R}^{d_2}$ is the d_2-dimensional hidden state of the node v_i in the l-th layer. N_i^r denotes the set of neighboring indices of node v_i under relation $r \in R$. $|N_i^r|$ denotes the number of nodes in N_i^r. $W_r^{(l)} \in \mathbb{R}^{d_2\times d_2}$ is the weight matrix for relation $r \in R$ in the l-th layer. $W_0^{(l)} \in \mathbb{R}^{d_2\times d_2}$ is the weight matrix for the node itself in layer l (self-connection). Initially, $h_i^{(0)} = x_i$, the feature vector associated with node v_i (see Section 4.1.2). The final output of R-GCN with L layers is the hidden state of the last layer $h_i^{(L)}$. Figure 4.3 illustrates the propagation process at layer l.

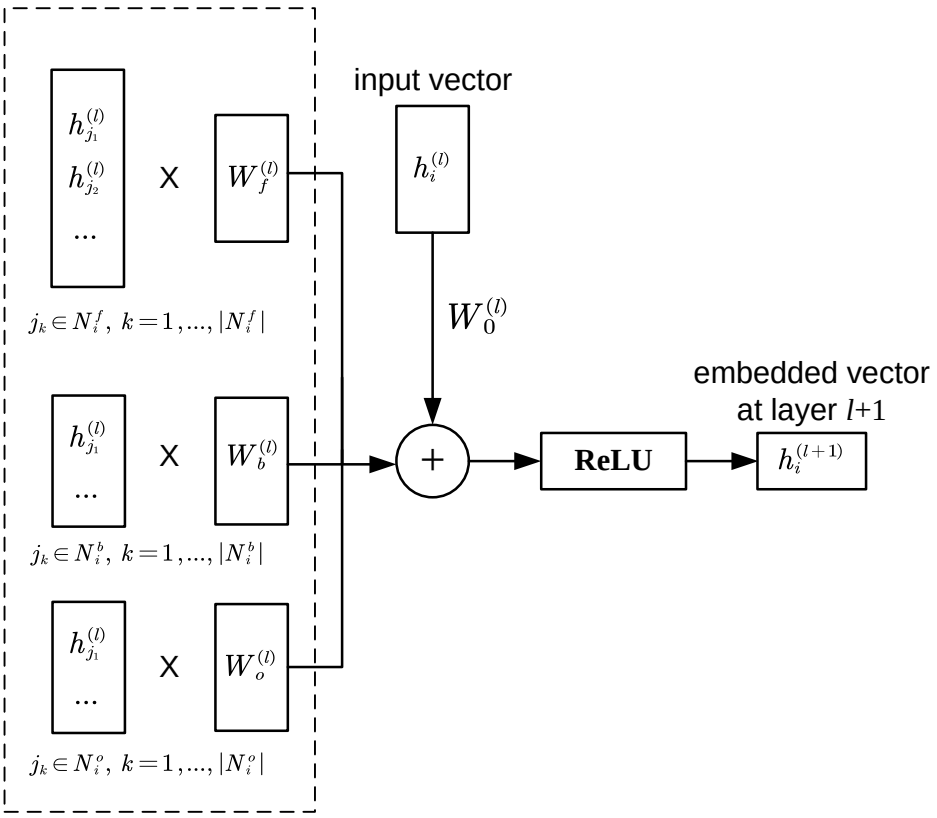


Figure 4.3: Embedding propagation at layer l of R-GCN

During training, each instruction embedding is propagated and updated L times via different relations: forward, backward, and overlap to capture information from neighboring nodes. The final output $h_i^{(L)}$ is fed into a classifier: a fully-connected layer to reduce the dimension to one, and then activated by *sigmoid* to generate a probability p. We try to minimize the Binary Cross Entropy loss function:

$$J(\Theta, p, y) = \sum(-(y \cdot \log(p) + (1-y) \cdot \log(1-p))) \tag{4.5}$$

where Θ denotes the model parameters and y is the true label. As shown in Figure 4.1, all the trainable modules of DeepDi are linked together and trained in an end-to-end fashion.

4.1.5 Function Entrypoint Recovery

To recover function entrypoints, we first identify a set of function entrypoint candidates, and then feed each candidate and its surrounding instructions into a classifier. To identify the candidates, we first obtain the metadata of valid instructions, and exclude instructions that are `int3`, `jmp`, `ret`, `nop`, or are reachable via instruction fallthrough or conditional jump because these instructions will not be function entrypoints. We also assume the targets of `call` instructions are function entrypoints. This not only reduces false positives, but also greatly reduces the number of candidates to evaluate.

We then stack each candidate instruction with three preceding instructions and three following instructions into our function entrypoint recovery model. The model has a learnable embedding layer followed by a GRU layer and a two-layer perceptron classifier. This will determine if this candidate instruction is indeed a function entrypoint. Let the valid instruction metadata be $\{t_0, t_1, \ldots, t_k\}$, we define the function entrypoint recovery model as follows:

$$g_i = f(u_{i-3}, u_{i-2}, u_{i-1}, u_i, u_{i+1}, u_{i+2}, u_{i+3}) \tag{4.6}$$

where f is the GRU's recurrent function, and $u_i \in \mathbb{R}^{4 \cdot d_1}$ is the embedding of t_i generated by a learnable embedding layer (not the same embedding layer in Section 4.1.2). g_i is the hidden state of the GRU layer and is then fed into a classification layer.

During the evaluation, we only feed function entrypoint candidates into our model. Since the number of function candidates is very limited compared to the number of superset instructions (about 1:30), this model has almost no impact on runtime performance. Our experiment shows that it helps achieve the average F1 score of function recovery of 98.6%.

Guo et al. [146] show that RNN-based function identification tends to learn specific bit patterns, such as `push ebp`. However, we identify function entrypoints based on high-level features learned by the neural network model and accurate instructions, which can likely lead to higher robustness. The drawback of this approach is that we will miss tail jumps and functions with unseen prologues. To identify tail jumps, we can use the same heuristics in other works [239, 238]. If the jump target address is larger than the next function start or smaller than the current function start, it is considered as a tail jump. For unseen prologues, we are able to find many of them via call targets.

4.2 Evaluation

In this section, we evaluate DeepDi's performance. Our experiments aim to answer the following Research Questions (RQs).

RQ1 How does it perform on regular binaries?

RQ2 How does it perform on unseen binaries?

RQ3 How does it perform on obfuscated binaries?

RQ4 How resilient is it against adversarial attacks?

4.2.1 Implementation and Setup

We use PyTorch [221] to implement our model and write a plug-in to disassemble raw bytes and return instruction metadata and an IFG as PyTorch Tensors. To disassemble instructions on GPU, we used a header-only library LDasm[4] and modified the code so that it can run on GPU, and its look-up tables are properly cached and shared among GPU

[4]https://github.com/Rprop/LDasm

threads. The IFG is represented as a set of sparse adjacency matrices, and we used the PyTorch Sparse[5] library to avoid expensive memory coalescing operations. We ran all the experiments on a dedicated server with a Ryzen 3900X CPU @ 3.80 GHz×12, one GTX 2080Ti GPU, 16 GB memory, and 500 GB SSD.

Baseline. We select the following disassemblers for baseline comparison: Binary Ninja 2.2 [8], IDA Pro 7.2 [25], Ghidra 9.1.2 [22], Datalog Disassembly [131], and XDA [226]. IDA Pro, Ghidra, and Binary Ninja are widely used in reverse engineering and binary analysis practices, and their results are considered high-quality. Datalog is a recently proposed binary rewriting approach. XDA is the state-of-the-art machine learning-based approach. This selection covers state-of-the-art commercial disassembler tools and the most recent research prototypes.

We used the default settings when evaluating IDA Pro and Binary Ninja. For Ghidra, we disabled its decompiler, ASCII string analyzer, x86 exception handling, and constant reference analyzer to boost its efficiency. We finetuned two XDA models, one for instruction and one for function entrypoints, both based on the pre-trained model that XDA provided. We kept the same hyperparameters as in their paper and finetuned each model for five epochs.

Dataset. We conducted experiments on BAP corpora [86], LLVM 11 for Windows[6], SPEC CPU2006 [47], and SPEC CPU2017 [48]. The BAP corpora contain 1,032 x86 and x64 ELF binaries compiled by `GCC` with optimization levels O0 to O3. Though these corpora also come with ELF binaries compiled by `Intel C++ Compiler` (`ICC`) and PE files, these

[5]https://github.com/rusty1s/pytorch_sparse
[6]https://github.com/llvm/llvm-project

binaries are not used in experiments due to the existence of jump tables in the code section. LLVM 11 is compiled by `Microsoft Visual Studio 2019` with optimization levels Od, O1, O2, and Ox for both x86 and x64 architectures. SPEC CPU2006 is compiled by `GCC-4.8.4` and `MSVC 2008` for x86 and x64 architectures and with four optimization levels. SPEC CPU2017 is also compiled on the two ISAs with four optimization levels by using `GCC-7.5` and `MSVC 2019`. To reduce the training time for XDA, we excluded files larger than 5MB.

In total, we have 1,032 ELF files (268 MB) from BAP, 266 PE files (322 MB) from LLVM, 152 PE files (152 MB) and 190 ELF files (79 MB) from SPEC CPU2006, and 270 PE files (287 MB) and 218 ELF files (120 MB) from SPEC CPU2017. Note that we only count code section size.

It is straightforward to extract the ground truth from ELF files, since there is no data in the code section according to Andriesse et al. [68]. We get instruction boundaries by linearly disassembling the code section. We use pyelftools[7] to get function entrypoints come from the symbol table where the symbol type is "`STT_FUNC`" and the symbol index is not "`SHN_UNDEF`" (to exclude external functions). To obtain the ground truth for PE files, we modified DIA2Dump, an example that comes with Visual Studio, to dump all functions, data, and label addresses from pdb files. We can only find data addresses but no data lengths in pdb files, so to estimate data ranges, we first find the label where the data belongs, then treat the data address to the end of that label as data. When creating the labels, we set the label to one if the corresponding byte is the starting point of an instruction or a function.

[7]https://github.com/eliben/pyelftools

Evaluation Metrics. For the accuracy evaluation, we use *F1* scores to measure the performance because both precision and recall are pretty high for almost all disassemblers. For generalizability and obfuscation evaluation, we use *Precision* (P) and *Recall* (R) to measure the performance.

Deep Learning Model Settings. We use the Adam optimization algorithm [177] and a default learning rate 10^{-3}. As introduced in Section 4.1.4, we use the Binary Cross-Entropy Loss to calculate the loss. We choose the following hyper-parameters through an informal parameter sweep process: $d_1 = 8$, $d_2 = 16$, $L = 2$, $M = 3$, and the batch size is 1,048,576. If a code section is larger than the batch size, we obtain an Instruction Flow Graph for each batch, and edges outside of this graph are dropped. We apply the same strategy to keep the graph small and fit in the GPU memory during the inference. In each batch, the average valid-to-invalid instruction ratio is about 1:1 because compilers tend to insert sufficient padding instructions to align instructions. If we count the paddings as invalid, the ratio becomes 1:4. The graph size is roughly five times the batch size: almost all instructions have only one forward and one backward relation (fallthrough), each of which overlaps with three instructions on average. We also apply a row normalization to make each node in a similar range [248]. As for the function model, the output length of the embedding layer is 8, the hidden size of GRU is 64, and the hidden layer size of the two-layer perceptron is 64, 1, respectively. In total, our model only has 49,889 trainable parameters.

Table 4.2: Instruction and Function Level Accuracy

Dataset	Opt.	Instruction F1 (%)						Function Entrypoint F1 (%)					
		DeepDi	**XDA**	**Datalog**	**IDA Pro**	**Binary Ninja**	**Ghidra**	DeepDi	**XDA**	**Datalog**	**IDA Pro**	**Binary Ninja**	**Ghidra**
BAP	O0	99.9	99.9	100	99.9	99.9	100	99.9	99.9	100	100	99.9	100
	O1	99.8	99.9	100	99.9	99.8	99.9	99.3	99.5	100	99.9	99.8	99.9
	O2	99.7	99.9	99.9	99.9	99.8	99.9	98.6	99.4	100	99.9	99.8	99.9
	O3	99.7	99.9	100	99.9	99.7	99.9	99.0	99.5	100	99.9	99.7	99.9
LLVM	Od	99.8	99.9	N/A	99.9	99.8	99.9	99.8	99.9	N/A	99.9	97.1	99.9
	O1	99.8	99.9	N/A	99.9	99.7	99.9	99.8	99.9	N/A	99.8	96.8	99.9
	O2	99.8	99.9	N/A	99.9	99.6	99.9	99.8	99.9	N/A	99.8	89.7	99.7
	Ox	99.7	99.9	N/A	99.9	99.7	99.9	99.8	99.9	N/A	99.8	84.9	99.7
SPEC 2006	O0	99.9	99.9	100	99.9	99.6	98.9	98.4	99.9	99.9	88.8	88.7	97.3
	O1	99.7	99.8	100	99.8	99.3	97.2	97.0	99.3	100	86.3	88.7	93.4
	O2	99.9	99.9	100	99.9	99.2	97.6	96.4	99.5	100	85.2	91.5	92.7
	O3	99.9	99.9	100	99.9	98.9	98.0	98.6	99.5	100	93.3	96.0	99.9
	Os/Ox	99.8	99.9	100	99.9	99.4	97.5	95.3	98.3	100	85.6	87.1	91.6
SPEC 2017	O0	99.9	99.9	99.9	99.9	99.7	94.2	99.0	99.8	100	89.4	93.6	86.7
	O1	99.9	99.9	100	99.9	99.5	95.9	99.7	99.8	100	80.8	95.6	76.8
	O2	99.8	99.9	100	99.9	99.4	95.1	99.5	99.9	100	79.4	96.7	75.5
	O3	99.6	99.9	100	99.9	98.9	90.1	98.9	99.4	100	88.4	93.5	85.0
	Os/Ox	99.7	99.8	100	99.9	99.6	96.5	96.3	99.5	100	72.5	92.1	68.7

4.2.2 Accuracy and Efficiency

In this section, we evaluate the accuracy and efficiency of DeepDi and other baseline tools. First, we introduce some details and settings of the experiments, then report and discuss experimental results.

Training and Testing Details. We randomly shuffled the dataset and did a 90-10% split (90% of binaries are used for training, 10% for testing). Both XDA and DeepDi are trained for five epochs because XDA converges after five epochs according to their paper. We feed code sections (raw bytes) to XDA and binary files to DeepDi.

Accuracy

To answer RQ 1, we measure F1 scores of DeepDi and baseline models at instruction and function levels, as shown in Table 4.2.

When evaluating instruction level results, we treat `nop`, `int3`, `hlt` and `jmp` instructions, and `lea` instructions whose source and destination registers are the same as padding instructions, thus they do not count towards positive or negative instructions. Similarly, for the function entrypoint evaluation, if the first instruction of a function is `jmp`, this function does not count towards positive or negative functions.

Datalog only supports x64 ELF files, so its evaluation on LLVM binaries is not available, and the corresponding cells show "N/As" in Table 4.2. From the table, we observe that most disassemblers struggle to identify function entrypoints on SPEC datasets. By looking into the datasets, we find that functions from the BAP and the LLVM datasets are mostly aligned, meaning padding instructions can be found between functions. These padding instructions are a strong indicator of function boundaries. However, functions from SPEC datasets are not aligned. To make it worse, many functions end with non-return calls, and frame pointers are often omitted on high optimization levels. With frame pointers omitted, the first instruction of a function is not `push ebp/rbp`, but `xor`, `cmp`, `mov`, etc. These are normal instructions after a `call` instruction, and this explains why many disassemblers struggle to recover function entrypoints. IDA Pro treats many small functions as error handling code, or "`__unwind`". That is why IDA Pro misses many functions in the LLVM dataset. Note that DeepDi is not the best performer, but is comparable with the other disassemblers. We are unable to evaluate Shingled Graph Disassembly [277] on our

dataset because it is not open source. Still, according to their paper, the accuracy of Shingled Disassembly is comparable to IDA Pro, meaning its instruction-level accuracy is similar to DeepDi.

Efficiency

Figure 5.4 shows the correlations between code section size and disassembly time for our approach, IDA Pro, Binary Ninja, Ghidra, Datalog, and XDA. The y-axis of this figure is log-scaled. For IDA Pro, Binary Ninja, and Ghidra, we run them in console/headless mode to avoid unnecessary GUI costs. For Datalog Disassembly, we take the numbers reported from the tool directly. When disassemblers are tested on CPU, only one CPU core is used to ensure fairness.

DeepDi on GPU clearly stands out in this experiment. Its throughput is about 24.5 MB/s, about 170 times faster than DeepDi on CPU, 146 KB/s. The latter still is noticeably faster than the remaining disassemblers: IDA Pro 72 KB/s, XDA (GPU) 47 KB/s, Binary Ninja 11 KB/s, Ghidra 10 KB/s, Datalog 5 KB/s (for files around 1 MB), and XDA (CPU) 140 B/s. Shingled Graph Disassembly, according to their paper, is two to three times faster than IDA Pro, making it comparable to our CPU approach.

In contrast, XDA is several orders of magnitude slower than the other disassemblers when running on CPU, and its GPU version is merely comparable to the other CPU disassemblers. It is worth noting that we obtained XDA source code from their GitHub repository, but we could not reproduce their reported efficiency. One possible reason is that they used three GPUs [226] whereas we only used one.

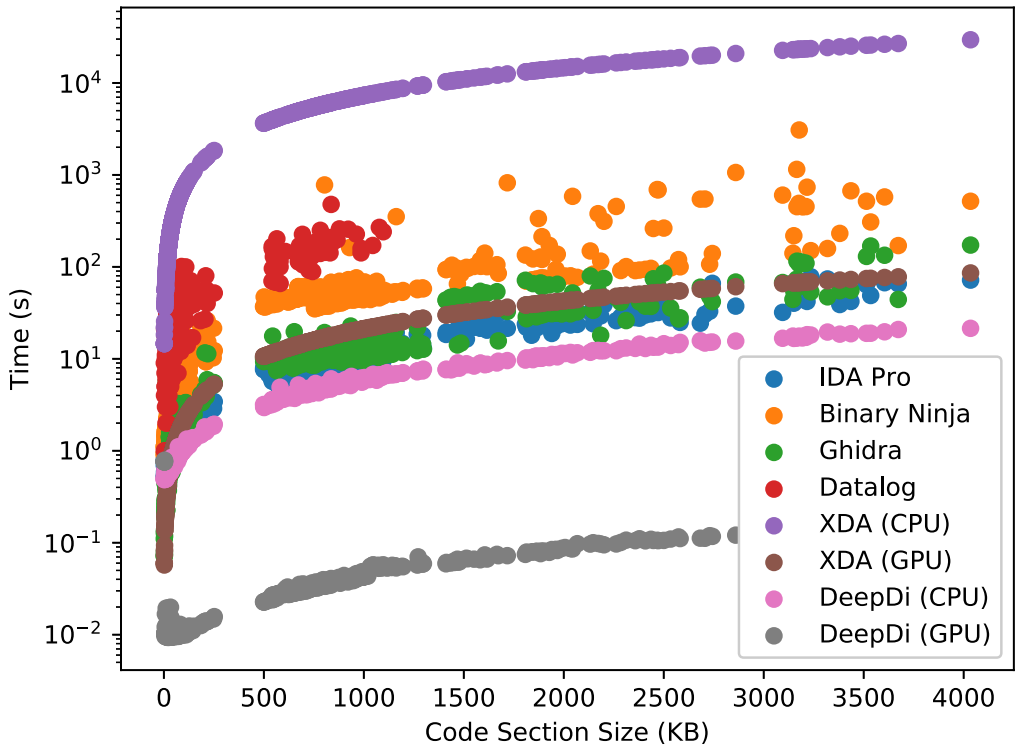


Figure 4.4: Efficiency Evaluation

> ***The answer to RQ 1***: DeepDi is very accurate on regular binaries. Its accuracy is comparable to all the commercial tools and recent research prototypes. Moreover, DeepDi is significantly more efficient.

4.2.3 Generalizability

To answer RQ 2, we conduct two experiments. First, we train our model on the BAP corpora and test it on the LLVM dataset, and then compare it with another machine

Table 4.3: Precision and Recall on Unseen Binaries from an Unseen Compiler

Model	Train \ Test	Instruction								Function							
		Od		O1		O2		Ox		Od		O1		O2		Ox	
		P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R
DeepDi	O0	98.6	**99.1**	**98.1**	**97.6**	**98.0**	**97.6**	**98.2**	**97.7**	**94.5**	**42.3**	**95.9**	**38.4**	**74.8**	**26.2**	**73.1**	**26.0**
	O1	98.6	**98.9**	**97.2**	**96.6**	97.9	**97.1**	98.0	**97.1**	**94.9**	**60.5**	**93.3**	**76.8**	**72.2**	**72.1**	**69.5**	**71.9**
	O2	98.9	**99.7**	**98.3**	**98.6**	**98.3**	**98.5**	98.2	**98.6**	**89.4**	**47.3**	**86.7**	**61.6**	**82.6**	**55.0**	**83.1**	**53.7**
	O3	98.2	**99.0**	**97.7**	**96.9**	**98.1**	**97.3**	98.1	**97.4**	**80.4**	**21.0**	**78.7**	**39.5**	**72.9**	**30.9**	**74.3**	**32.5**
XDA	O0	**98.7**	38.9	96.1	43.9	97.1	42.1	97.5	42.6	56.9	0.1	77.6	0.7	5.3	0.03	45.5	0.6
	O1	**99.0**	37.5	**97.2**	44.2	**98.1**	42.5	**98.4**	43.0	2.6	0.4	8.9	1.2	2.3	0.9	3.6	1.4
	O2	**99.1**	38.7	97.2	46.5	98.2	44.2	**98.5**	44.6	16.8	0.5	57.6	3.8	29.5	2.9	34.1	3.9
	O3	**98.9**	39.8	97.3	47.6	**98.1**	44.8	**98.4**	45.1	8.7	0.2	40.4	1.4	7.6	0.4	20.5	1.4

Table 4.4: Precision and Recall of Function Entrypoint Recovery on Real-world Software

Model	Opt.	curl		diffutils		GMP		ImageMagick		libmicrohttpd		libtomcrypt		OpenSSL		PuTTy		SQLite		zlib	
		P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R
DeepDi	O0	99.9	99.9	99.4	99.2	97.7	**97.2**	99.6	99.9	**99.5**	99.5	97.7	94.2	99.7	**100**	99.9	99.8	99.8	99.9	**100**	99.3
	O1	**98.5**	**99.4**	94.6	94.8	**96.9**	85.6	98.2	94.9	**93.6**	89.5	**97.9**	77.1	**97.3**	93.5	**99.4**	91.6	**97.7**	96.9	**98.3**	85.6
	O2	**96.2**	**96.6**	94.4	**96.5**	95.7	90.7	94.1	95.3	91.7	92.7	97.8	95.6	**92.6**	**95.5**	**98.5**	95.6	94.9	**95.5**	**99.1**	84.3
	O3	**96.7**	**97.4**	88.9	**97.9**	96.0	91.3	94.1	95.1	88.7	93.4	**97.9**	95.1	**92.8**	96.0	**99.0**	**96.2**	94.7	**95.9**	98.0	84.2
XDA	O0	**100**	**100**	**100**	**100**	**99.2**	96.7	**99.9**	**100**	**99.5**	**100**	**99.6**	**95.6**	**100**	**100**	**100**	**99.9**	**100**	**100**	**100**	**100**
	O1	91.6	96.0	**96.1**	**96.6**	94.1	**94.2**	**98.9**	**98.7**	89.8	**92.8**	91.4	**95.8**	93.0	**96.1**	95.4	**97.3**	92.9	**97.0**	94.8	**92.7**
	O2	88.9	95.6	**95.9**	95.4	**95.9**	**91.9**	**97.9**	**98.4**	**93.9**	**95.5**	**98.0**	**96.0**	89.6	95.1	96.1	**95.9**	**95.9**	94.0	**99.1**	**90.9**
	O3	88.9	96.1	**94.1**	95.7	**96.3**	**94.7**	**97.1**	**97.8**	**96.6**	**95.8**	97.0	**96.6**	83.8	**97.3**	96.2	94.3	**95.3**	94.9	**98.2**	**93.3**

learning-based model – XDA [226]. We did not do it in the opposite way (i.e., training on the LLVM and testing on the BAP corpora) because XDA is pre-trained on the BAP corpora [226] and this dataset should not be considered unseen for XDA. DeepDi and XDA are trained on each optimization level of BAP corpora for five epochs and tested on the LLVM binaries. This experiment shows disassemblers' performance on unseen binaries of different compilers (`GCC` vs `MSVC`), platforms (Linux vs Windows), and optimization levels.

Second, we evaluate our model and XDA's model from Section 4.2.2 on the same unseen real-world software used by XDA. This experiment uses unseen real-world software to show the performance in real-world scenarios.

Table 4.3 lists the evaluation results on instruction and function recoveries. Even though DeepDi has not seen LLVM binaries before, it still reaches 97.1%+ precision and recall on recovering instruction boundaries. However, XDA only obtains a high precision while recall is constantly below 50%. One possible explanation is that XDA's attention header is too conservative, and does not perform well when instruction patterns are unseen. The function entrypoint recovery evaluation shows a greater degradation when analyzing unseen binaries of unseen compilers. As the optimization level increases, function prologues become less obvious and differ a lot from compilers to compilers, making function identification much harder. Despite that, DeepDi outperforms XDA by a large margin.

Table 4.4 shows the precision and recall of function entrypoint recovery on each software and optimization. We find that DeepDi is on par with XDA. The F1 scores of both XDA and DeepDi are close to 100 on instruction recovery, and their performance is almost identical, so we omit the table for instruction recovery.

The first experiment shows that DeepDi can generalize function entrypoint recovery to some extend when analyzing binaries from unseen compilers and optimization levels. The second experiment shows DeepDi can generalize pretty well when compilers and optimization levels are already known. This indicates that each compiler has its function patterns on each optimization level, so for DeepDi, training the model on binaries com-

piled by gcc and MSVC with different optimization levels is good enough for most general software.

> ***The answer to RQ 2***: For unseen binaries, DeepDi is still able to achieve high precision and recall. It outperforms another machine learning-based model, XDA, by a large margin for unseen compilers and optimization levels, and is on par with XDA for unseen real-world binaries. These results suggest that DeepDi has good generalizability.

4.2.4 Obfuscation Evaluation

To answer RQ 3, we used two different obfuscators to evaluate whether our approach is resilient to obfuscations, and how it compares with the disassemblers with sophisticated heuristics. The first obfuscator was developed by Linn and Debray [197]. In that paper, the authors proposed to insert junk code to confuse both linear and recursive disassembly. Moreover, unconditional jumps are redirected to a universal function that modifies its return address based on callers. This nonstandard behavior hides jump targets and breaks common heuristics. We used the models trained in Section 4.2.2 and the ground truth provided by Linn and Debray [197]. They provided 11 obfuscated x86 ELF binaries of the SPECint 2000 benchmark suite that have been obfuscated by their tool. Evaluation results of these binaries are shown in Table 4.5.

We excluded Datalog Disassembly and Binary Ninja because Datalog Disassembly does not support x86 ELF files, and Binary Ninja consumed all memory resources and was killed by the OS. We can observe from Table 4.5 that DeepDi is the best performer with

respect to precision, recall, and runtime efficiency. In contrast, Ghidra took almost three hours to analyze these binaries and achieved low precision and recall. XDA is slightly worse than DeepDi in terms of precision and recall, but 235 times slower than DeepDi on GPU.

Table 4.5: Obfuscation Test Results

Disassembler	**Precision**	**Recall**	**Time**
DeepDi (GPU)	**84.1**	**95.2**	**1.2s**
XDA (GPU)	80.2	95.1	282s
IDA Pro	75.8	44.8	262s
Ghidra	69.1	47.0	10,240s

Table 4.6: Function Entrypoint Recovery on Obfuscated Unseen Binaries, P: Precision, R: Recall, T: Time

Obfuscation	**DeepDi**			**XDA**			**IDA Pro**			**Binary Ninja**			**Ghidra**			**Datalog**		
	P	R	T	P	R	T	P	R	T	P	R	T	P	R	T	P	R	T
bcfobf	98.9	98.9	1.6s	99.6	99.4	396s	99.5	100	129s	86.1	100	621s	35.9	33.1	208s	99.7	100	783s
cffobf	99.4	97.9	0.7s	99.6	99.1	342s	99.9	100	112s	98.6	100	593s	39.8	33.0	920s	99.7	100	1,231s
indibran	99.8	98.0	0.5s	99.8	99.0	229s	20.5	100	842s	75.5	99.9	248s	39.7	33.3	230s	98.8	100	905s
splitobf	99.7	98.6	0.6s	99.8	99.3	312s	100	100	117s	98.5	100	539s	42.4	33.2	198s	99.7	100	480s
subobf	99.7	97.9	0.5s	99.8	98.8	187s	100	100	63s	98.6	100	409s	50.6	33.3	105s	99.7	100	284s

We also evaluated another obfuscator called Hikari [314]. It is an improvement over Obfuscator-LLVM [169], and it can generate hard-to-read code to provide tamper-proofing and increase software security. We used five obfuscation strategies, namely bogus control flow (bcf), control flow flattening (cff), basic block splitting (splitobf), instruc-

tion substitution (subobf), and register-based indirect branching (indibran) to obfuscate seven popular open-source projects, including `curl-7.74.0`, `diffutils-3.7`, `gmp-6.2.1`, `ImageMagick-7.0.10`, `libmicrohttpd-0.9.72`, `SQLite -3.34.0`, and `zlib-1.2.11`. We also turned off optimizations as instructed by Hikari [314]. The function entrypoint evaluation results are shown in Table 4.6. In this experiment, IDA Pro has low precision when files are obfuscated by Indirect Branching. It fails to resolve some indirect jump instructions and treats these jump targets as function entrypoints. Ghidra misidentifies many function entrypoints, indicating that signature-based function identification is not very resilient to unseen patterns. IDA Pro, Binary Ninja, Ghidra, and Datalog Disassembly show increased analysis time due to the increased control flow complexity. In contrast, machine learning-based approaches like DeepDi and XDA are not affected by this.

Based on the results in Table 4.5 and Table 4.6, we can see that the two machine learning-based approaches, DeepDi, and XDA, are superior in accuracy when dealing with obfuscated binaries, but DeepDi is hundreds of times faster than XDA on GPU.

> ***The answer to RQ 3***: For obfuscated binaries, DeepDi is superior in accuracy and its efficiency is not affected by the increased code complexity.

4.2.5 Adversarial Evaluation

An extensive answer to RQ 4 would deserve a separate investigation. In this section, we conduct a preliminary evaluation. Since our model relies on jump relations to recognize true instructions, one possible adversarial attack would be replacing some of

these jumps with computed jumps. In this experiment, we trained our model on O3 BAP corpora. In evaluation, we use O0 BAP corpora and randomly drop 50% and 90% of jump edges.

The evaluation results show that if 50% of the jumps are removed, the false positive rate (FPR) increases slightly from 0.0473% to 0.0524%, and the false negative rate (FNR) from 0.24% to 0.51%. If 90% are removed, the FPR is 0.0575%, and the FNR is 0.81%. By analyzing false-negative cases, we find most false negatives are the first instruction of a short basic block, or `nop` instructions at the beginning of a basic block. This makes sense because the first instruction of a basic block, especially a short one, has the least context information if it is not a jump target.

We also evaluate the function entrypoint accuracy. When all jump edges are removed, precision drops to 93.8% and recall to 98%. Precision drops a lot because `GCC` may align basic blocks and insert `nop`s between them. If a function has multiple exits, we can find code patterns like `return` - `nop` - `mov reg, [reg]`. The third instruction looks like a function entrypoint even to humans, and thus confuses the model.

We speculate that the high resiliency of DeepDi against this jump-obfuscation attack is attributed to graph inference, which takes into account several kinds of relations between instructions. Context information still exists in adjacent instructions and overlapping instructions. Destroying only a part of these relations (in this case, jump relations) does not cause a drastic impact on the overall graph inference task.

> ***The answer to RQ 4***: Through a preliminary evaluation on jump-obfuscation attacks, we show that DeepDi has good resilience.

4.3 Downstream Application

In this section, we showcase how DeepDi can support downstream applications. Particularly, we choose malware classification in this demonstration. We leave more extensive evaluations on downstream applications as future work.

We use the malware dataset from Microsoft Malware Classification Challenge [244]. This dataset contains nine malware families, and is split into $10,868$ malware training samples and $10,873$ testing samples. Each malware sample comes with IDA Pro disassembly results and raw bytes (represented as hexadecimal values) of the code sections. Some raw bytes are represented as "`??`", so we removed such bytes and converted other hex strings back to bytes. For all the following experiments, we use 10-fold cross-validation on the training data and report mean accuracy as well as standard deviation. The ground truth of the test dataset is not released to the public, and the only evaluation metric returned from the online judge system is logloss, so we report logloss instead of accuracy on the test dataset. As a reference, the logloss of random guessing on the test dataset is *2.19722*.

The top models in this challenge used both disassembly and raw bytes to extract high-level features such as N-gram and strings [244]. These features are expensive and can take hours or even days to extract [60, 315]. Although they could achieve over 99.7% training accuracy and 0.0063 in loss, those models are impractical for real-time analysis.

To demonstrate how the high-level features benefit malware classifiers, we conduct two experiments. First, we compare MalConv [240] with Gemini [283] to compare the performance of classifiers that take raw bytes and high-level features. Second, we compare

the original EMBER [67] with a modified version where high-level disassembly features are added.

For the first experiment, we extend Gemini [283] which takes attributed control-flow graph (ACFG) as input, generates embeddings for all basic blocks, and finally outputs an embedding for each function by summing up all basic-block embeddings. To build a malware classifier, instead of generating function embeddings, we concatenate min- and max-pooling of all basic-block embeddings of the program, and then feed them into a 2-layer perceptron followed by a `tanh` activation function. It finally outputs 9-dimensional vectors for classification. We can then use softmax to get a probability for each class. We expect that a classifier based on high-level features can achieve good accuracy and generalizability.

We use Adam optimizer with the default learning rate 10^{-3} and Cross Entropy Loss to train the model. At the input layer, we added a fully connected layer to increase the vector size to 32 to allow more information to pass through ACFGs. We also set the output embedding size 32, and information propagates five hops. In this simple case study, we did not attempt to find the optimal hyperparameters or explore different network architectures, so there is certainly room for improvement.

We also evaluated MalConv [240], a convolutional neural network model that takes raw bytes as input for malware classification. We used the same training strategy described above to train a MalConv model.

Table 4.7 lists the results of this experiment. We can see that although MalConv has better training accuracy, Gemini can better generalize with a 0.13 logloss. This result substantiates that a malware classifier based on high-level features tends to be more accurate

Table 4.7: Malware Classification Results

Model	**Training Accuracy**	**Testing Loss**	**Time (GPU)**
Gemini	$96.52\% \pm 0.595$	0.134974 ± 0.036	7m
MalConv	$97.81\% \pm 0.659$	0.159165 ± 0.048	48.6s

on unseen samples. In terms of efficiency, MalConv only takes 48.6 seconds to process all testing samples (5.2 GB in total) on GPU, because it takes raw bytes as input. Gemini takes 7 minutes to process the same amount of samples on GPU. This is still a notable achievement, given that DeepDi has to disassemble the malware samples and extract ACFG as high-level features, and then hand them over to Gemini to perform classification.

For the second experiment, we evaluate EMBER which uses static features such as byte code histogram and imported functions to train a gradient-boosted decision tree (GBDT) model. We first train the original EMBER model with the default parameters except for changing the objective from binary to multiclass. Later, we add high-level features: code histogram and code entropy histogram, to the static features to show how they benefit classification. Code histogram and entropy histogram are extracted from instruction metadata mentioned in Section 4.1.2, similar to how byte histogram and byte entropy histogram are extracted.

Table 4.8: EMBER Classification Results

Model	**Training Accuracy**	**Testing Loss**	**Time**
EMBER	$99.13\% \pm 0.1747$	0.041541 ± 0.0022	21m
EMBER w/ code	$99.40\% \pm 0.2465$	0.024391 ± 0.0018	24m

Table 4.8 shows that we can lift the training accuracy from 99.1% to 99.4%, and almost halve the testing loss while adding minor overhead (3 minutes).

This case study shows that DeepDi opens up a lot of opportunities for fast and accurate binary analysis. It will be interesting to explore other machine-learning and deep-learning models that take disassembly results and high-level features as input to produce even more accurate classification results and conduct other binary analysis tasks.

4.4 Discussion

In this section, we have more discussions about our evaluation results.

Learning-based vs. Rule-based Approaches. In this work, we demonstrate that a learning-based approach outperforms rule-based approaches used in commercial disassemblers with respect to accuracy (especially on obfuscated binaries) and efficiency. This result might be surprising to many people, as binaries are generated by the compilers following a well-understood compilation process. So experts should be able to develop good rules and heuristics to correctly disassemble the binaries. However, much higher-level information is lost during the compilation process, and ambiguities start to emerge. The situation is further exacerbated by deliberate obfuscations that aim to break these rules and heuristics, as demonstrated by our obfuscation evaluation in Section 4.2.4. A learning-based approach, if done right, can automatically learn from a large number of real data on how to resolve the ambiguities and tolerate certain obfuscation attempts. We also demonstrate that a learning-based approach (particularly, a neural network-based approach) can be more efficient than rule-based approaches. A deep neural network model can better leverage parallelism in

modern processors to perform vector and matrix computation very efficiently. In contrast, a rule-based approach may not be easily parallelized.

Generalizability. A common problem for a machine learning model is overfitting, meaning that the model only learns superficial features existing in the training dataset and cannot generalize on unseen datasets. Our evaluation in Section 4.2.3 shows that our model is able to learn intrinsic features from the training set, and perform well on a completely different dataset containing a different set of programs generated by a different compiler for a different operating system. We speculate that this excellent generalizability mainly comes from how we make use of Relational-GCN, as it captures a number of important relations between instructions. These relations generally hold true across programs, compilers, and OS.

Adversarial Attacks. A machine-learning system is known to be vulnerable to adversarial attacks. DeepDi is no exception. However, the disassemblers we evaluated face the same problem, and perform even worse than DeepDi on obfuscated binaries. Section 4.2.5 shows that DeepDi at least is able to counter attacks that simply hide direct jumps. A strong adversary may be able to perform an in-depth analysis on our model (e.g., based on the gradients), to construct adversarial examples. This problem deserves a separate investigation, and we leave it as future work. Nevertheless, our evaluation in Section 4.2.4 and Section 4.2.5 shows that DeepDi is already more robust than the existing commercial disassemblers.

4.5 Conclusion

In this chapter, we have proposed DeepDi, a novel deep learning based technique for disassembly that achieves both accuracy and efficiency. Our experimental results have shown that DeepDi's accuracy is comparable to the state-of-the-art commercial tools and research prototypes, and it is two times faster than IDA Pro, and its GPU version is 350 times faster. DeepDi is able to generalize to unseen binaries, and counter obfuscations and certain adversarial attacks. When used with EMBER [67] for malware classification involving 5.2 GB testing samples, we are able to increase training accuracy to 99.4% and only add 3 minutes to feature extraction time, showing its capacity of classifying malware accurately and efficiently.

Acknowledgement

We would like to thank the anonymous reviewers for their helpful and constructive comments. This work was supported in part by National Science Foundation under Grant No. 1719175, and Office of Naval Research under Award No. N00014-17-1-2893. Any opinions, findings, and conclusions or recommendations expressed in this chapter are those of the authors and do not necessarily reflect the views of the funding agencies.

Chapter 5

GrassDiff: Learning-Free Callgraph Matching for Precise Function Identification

Precise identification and matching of functions in binaries is a fundamental task in binary analysis and reverse engineering. It aims to identify and correlate identical or similar functions in different binaries in order to understand program behaviors, detect vulnerabilities, understand malware evolution, etc.

However, obstacles like compiler optimizations, compilation configurations, architectures, and code obfuscation make function matching surprisingly hard. Traditional function matching approaches [122, 72, 166, 108, 203] typically rely on manually defined features, such as strings, constants, opcode sequences, and formulas, in the assembly that are relatively resilient to the changes introduced by these obstacles. Metrics like Jaccord

similarity [161], longest common subsequence [203], and MinHash [228, 166] are often used to compute similarity scores between two sets of features. Some approaches leverage topology information from binaries and use techniques such as graph matching [162], graph edit distance [65, 229], or graph isomorphism [134, 211, 168] on control flow graphs (CFGs), call graphs, or interprocedural program dependency graphs (IPDGs).

Another line of research utilizes dynamic analysis approaches like taint analysis [211], symbolic or concolic execution [307, 161, 134, 228], or blanket execution [119] to obtain basic block or function input-output pairs and formulas. While offering better accuracy and resilience, these approaches often take much longer to extract the features.

Recently, researchers have leveraged machine learning to tackle function matching problems. Various approaches [282, 198] encode CFG information into function embeddings, while others [113, 208, 318] draw inspiration from natural language processing (NLP) to automatically extract features and learn representations. More recently, transformer-based approaches [192, 225, 227, 271] become mainstream, significantly advancing function matching accuracy.

Challenges. Despite the promise of learning-based approaches, several challenges remain in the field of function matching:

One major challenge lies in the balance between effectiveness and efficiency. Transformer-based approaches demonstrate superior performance under various conditions, but only when the data distribution has been seen by the models. Section 5.1.1 shows a motivating example of how transformer-based approaches fail to understand and extract semantic features from binaries compiled by an unseen compiler. Conversely, traditional approaches

reliant on graph matching or dynamic analysis show better resilience to the differences caused by compilers and architectures. However, their computational inefficiencies limit their real-world adoptions, especially in large binary analyses.

Moreover, most learning-based approaches require training, and some [136] even require online training. Not only training is costly in time and resources, but the models also tend to overfit, depending on model architectures and training data qualities. As Section 5.1.1 shows, even the transformer model cannot generalize well on function matching problems. Additionally, online training is much more costly, for example, SigmaDiff [136] spends 203 seconds in finetuning on GPU, and would be much worse on large graphs.

Our Approach. To tackle the aforementioned challenges, we present a novel function matching framework called GrassDiff[1] that is efficient, resilient to code optimization, and architecture agnostic. Specifically, when provided with two binaries, GrassDiff extracts semantic information from decompiled functions and establishes an initial function matching matrix. This matrix is generated by computing cosine similarities between the embeddings of corresponding functions from the two binaries. GrassDiff then leverages a graph matching algorithm to find an optimal one-to-one function matching, maximizing both first-order function-level similarities and second-order call edge similarities [196]. Specifically, we finetune the state-of-the-art CodeBERT model [129] on decompiled code for function similarity detection. We also enhance Graduated Assignment (GA) [139], a classic graph matching algorithm, to leverage our initial function matching result while incorporating first-order and second-order information. Our improved GA (GA+) converges much faster, is more accurate, and can leverage the parallelism of GPU.

[1] GrassDiff stands for **Gr**aduated **ASS**ignment for binary **Diff**ing

We have conducted an extensive evaluation on GrassDiff and measured its accuracy, robustness, and efficiency on datasets containing different versions of common programs compiled by various compilers, optimizations, and on various architectures. The evaluation shows that GrassDiff improves the accuracy of pure embedding-based approaches by 5% to 20% when the embedding quality is good. Furthermore, our case study on real-world vulnerabilities showcases GrassDiff's ability to accurately identify vulnerable functions even when the embedding quality is poor. Additionally, GA+ is over 10x to 20x faster than GA, is more accurate, and scales better with large binaries.

Contributions. The contributions are as follows:

- We propose GrassDiff, a novel function matching framework that leverages program-wide topology information, is resilient to code optimization, and is architecture agnostic. It exemplifies how fast and accurate graph matching is made possible in binary analysis and how substantially it can improve over the existing approaches.
- We modify and improve the GA algorithm to run in GPU, leverage function similarities, and handle large graphs efficiently.
- We conduct an extensive evaluation to show our superior robustness and accuracy under various conditions, as well as good scalability even on large binaries.
- We showcase GrassDiff in real-world vulnerability detection, and we show that GrassDiff can pinpoint vulnerable functions even when the embedding quality is poor.

5.1 Motivation

Binary function matching is the foundation of many security applications, including but not limited to vulnerability detection, malware evolution analysis, and code plagiarism detection. While traditional graph-based approaches such as BinDiff [57], Diaphora [55], BinHunt [134], iBinHunt [211] offer precise one-to-one function matching, the NP-hard nature of (sub)graph matching or isomorphism render them impractical on large binaries. Recent advancements in natural language processing (NLP) and the transformer model have significantly improved function matching accuracy, and have become the new state-of-the-art approaches, at the cost of dropping one-to-one mapping constraints. These approaches take assembly code as input, and have shown superiority in various settings such as cross-optimization function similarity search.

However, we find that the generalizability of these approaches is poor, rendering them unusable when facing unseen code sequences, and thus insufficient as a function matching solution. In this section, we first discuss a real-world function matching result via jTrans [271], then discuss the possible reasons behind this, and finally explain how we tackle this problem.

Listing 1 GCC 13

```
endbr64

push    r15

push    r14

push    r13

push    r12

push    rbp

push    rbx

sub     rsp, 38h

mov     r14, [rdx+20h]

mov     [rsp+08h], rsi

mov     rax, fs:28h

mov     [rsp+28h], rax

xor     eax, eax

test    r14, r14

jz      loc_A1800
```

Listing 2 MSVC 2022

```
mov     [rsp+10h], rdx

push    rbp

push    rsi

push    r14

mov     eax, 50h

call    j__alloca_probe

sub     rsp, rax

mov     rsi, [r8+18h]

mov     rbp, r8

mov     r14, rcx

test    rsi, rsi

jz      loc_1800BB3B4
```

5.1.1 A Motivating Example

Different compilers have different dialects and preferences for register allocations, stack usage, optimization strategies, etc. To illustrate, we use a real-world vulnerable function, `BIO_new_NDEF`, from CVE-2023-0215. Listing 1 and 2 depict the first basic block of this function compiled by GCC-13 and Microsoft Visual C++ (MSVC) 2022 respectively. Despite seemingly being completely different, the two code snippets are doing the same initialization and condition check. Surprisingly, jTrans, a SOTA function similarity detection model, finds these two functions very dissimilar with a similarity score of less than 0.4 and

a rank of 891 due to the binary compiled by MSVC having a different dialect and not being trained.

Listing 3 GCC 13 Decompiled Code

```
undefined8 BIO_new_NDEF(undefined8 param_1,

    undefined8 param_2,long param_3)

{

  long lVar1 = *(long *)(param_3 + 0x20);

  if ((lVar1 == 0) || (*(long *)(lVar1 + 0x18) == 0))
```

Listing 4 MSVC 2022 Decompiled Code

```
undefined8 BIO_new_NDEF(undefined8 param_1,

    undefined8 param_2,longlong param_3)

{

  longlong lVar1 = *(longlong *)(param_3 + 0x18);

  if ((lVar1 != 0) && (*(longlong *)(lVar1 + 0x18) != 0))
```

This result shows that even with the transformer model and carefully curated pretraining tasks, the model is still unable to generalize and capture assembly code semantic meanings. This is not surprising, as each instruction carries much less information and is much more difficult to comprehend compared to source code. Conversely, the decompiled code shown in Listing 3 and 4 underscores the decompiler's efficacy in recovering high-level semantics. Additionally, the call graph as shown in Figure 5.1 stays relatively unchanged. These observations motivate us to design a function matching framework that leverages

both the topology information and decompilation techniques to help and improve the pure embedding-based approaches.

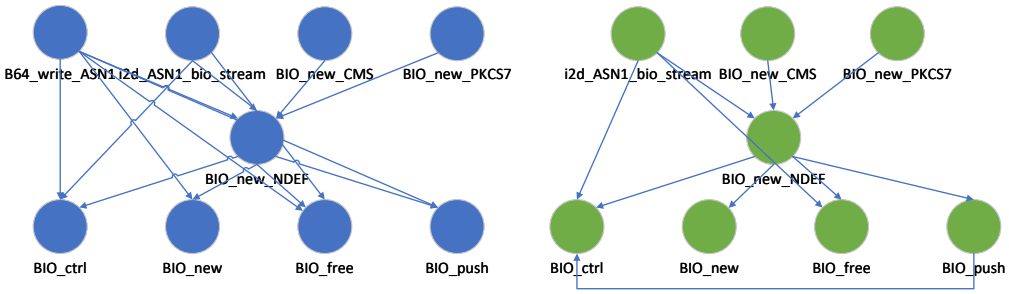


Figure 5.1: BIO_new_NDEF call graph. Left: compiled by MSVC-2022. Right: compiled by GCC-13.

5.1.2 Design Goals

We envision a good function matching solution should achieve the following design goals:

- **High Accuracy.** It should correctly identify matched functions with high precision and recall. The correct pairs should be ranked at the top.
- **Cross-Architecture.** It should support cross-architecture function matching.
- **Scalable and Efficient.** It should be able to process large binaries within a reasonable amount of time and resources. Ideally, it should be able to be accelerated by vector processors such as GPU.
- **Learning-Free.** It should not require any online training. Online training, especially on large binaries, can take a significant amount of time and resources.

5.1.3 Challenges and Opportunities

Despite the advantages of transformer models in function matching, they still struggle to understand high-level semantic meanings of assembly code, and supporting multiple architectures is both time- and resource-intensive. On the other hand, although graph matching or graph isomorphism can alleviate the problems caused by transformer models to some extent, its NP-hard nature makes it unscalable and thus unsuitable for practical applications, especially with large binaries. Notably, most computer vision datasets for graph matching contain fewer than 600 nodes in a graph [147], whereas a typical 2MB binary may contain around 6,000 functions. Moreover, graph deformation caused by inlining and partial build poses a greater challenge.

Nevertheless, there are some unique opportunities that can make transformer models work better and make graph matching feasible in function matching.

First, assembly code is difficult to parse and extract high-level information because registers are volatile, memory regions can be overwritten, and each instruction carries very little information. One has to follow the control and data flow closely and put a bunch of instructions, often nonconsecutive, together to better understand what the instructions do. Luckily, a decompiler does the exact same thing. It lifts assembly code to pseudocode and recovers control flow structures, variables, conditions, etc. This saves transformer models from the heavy-lifting analysis work and the semantic features are more readily available. Moreover, decompiled code is by nature platform- and architecture-agnostic. Given the aforementioned benefits, we propose to generate function embeddings from decompiled code.

Second, the initial similarity matrix given by the embeddings is very close to the final assignment matrix, provided the embedding quality is not too bad. Graph matching algorithms, especially the ones in an iterative fashion, can potentially leverage that initial result and converge faster.

Third, call graphs are sparse. In our preliminary evaluation, each function has one to three edges on average. This means the memory pressure for adjacency matrices can drop from $O(N^2)$, where N denotes the number of functions, to $O(N)$, making it possible to run graph matching algorithms on large graphs, if the graph matching algorithms support sparse matrices.

These opportunities make fast and accurate function matching possible. In the next sections, we will discuss what graph matching algorithms are suitable for the function matching problem and how we design our function matching framework.

5.2 Graph Matching Algorithms

There exist too many graph matching algorithms and approximations [273, 147], each of which has its unique relaxation or simplification. In this section, we try to survey and summarize some popular graph matching algorithms, and explain why we end up with graduated assignment for the function matching task. Table 5.1 summarizes these graph matching algorithms and we categorize them as follows:

- LAP solvers: These algorithms are tailored for addressing simplified Graph Matching (GM) problems, often characterized as Linear Assignment Problems (LAP).

Table 5.1: Summary of Graph Matching Algorithms. GA: Graduated Assignment, SM: Spectral Matching, RRWM: Reweighted Random Walks Matching, FGM: Factorized Graph Matching, DGMC: Deep Graph Matching Consensus, GMN: Graph Matching Network. Features include: Relaxation Method, Directed: support for directed graphs, Start Point: ability to utilize starting points for accelerated convergence, GPU: support for GPU acceleration. Relaxation methods include: DS: Doubly-Stochastic Relaxation, SP: Spectral Relaxation, CC: Concave-Convex Relaxation.

	Category	**Relaxation**	**Directed**	**Start Point**	**GPU**	**Time Complexity**	**Space Complexity**
B&F	Other	N/A	N/A	N/A	No	$O(N^2)$	$O(N^2)$
Hungarian	LAP Solver	N/A	N/A	N/A	Yes	$O(N^3)$	$O(N^2)$
Hopcroft–Karp	LAP Solver	N/A	N/A	N/A	No	$O(M\sqrt{N})$	$O(N+M)$
GA	Classic GM	DS	No	Yes	Yes	$O(I_0(N^3+I_1N^2))$	$O(N^2)$
SM	Classic GM	SP	No	Yes	Yes	$O(IN^4)$	$O(N^4)$
RRWM	Classic GM	DS	Yes	Yes	No	$O(IM^2)$	$O(N^4)$
FGM	Classic GM	SP + CC	Yes	Yes	No	$O(I(N^3+(N+M)^2)+(N+M)^3)$	$O(M^2)$
DGMC	Neural GM	DS	Yes	Yes	Yes	$O(I(N^2+M))$	N/A
GMN	Neural GM	N/A	Yes	No	Yes	$O(M^2+I(M+N))$	N/A

- Classic GM: These algorithms are designed for GM problem-solving without utilizing neural networks.
- Neural GM: These algorithms are employed to tackle graph matching tasks by leveraging neural networks and machine learning techniques.
- Others: Algorithms do not belong to the categories above.

The term "Relaxation method" refers to the specific relaxation techniques employed by graph matching algorithms to address the NP-Hard Quadratic Assignment Prob-

lem (QAP). "Support directed graph" indicates whether these algorithms are applicable to directed graphs. "Utilize start point" refers to the capability of leveraging the function similarity matrix as the initial matrix in function matching, thereby potentially accelerating convergence.

we also utilize several symbols in Big-O notation:

- I_x or I: Represents iteration control variables. For instance, in GA, I_0 and I_1 govern the iteration numbers for the softassign loop and Sinkhorn loop, respectively.
- N: Denotes the number of nodes in the graphs.
- M: In the LAP solver, it denotes the number of possible assignment candidates. In classic Graph Matching, it represents the number of edges in two graphs.

Subsequently, we aim to demonstrate, through experimentation and analysis, the efficacy of certain algorithms in solving our specific problem. Factors such as time complexity, space complexity, etc., will be considered to ascertain their suitability. From this assessment, we will select practical and representative algorithms for our evaluation.

Back and Forth Game. The Back & Forth Game (B&F) [120], as employed by David et al. [107], represents a method utilized for solving matching problems. In their study, the authors leverage this approach, encoding function context into the similarity computation, and employing these similarities for matching purposes.

During the game, a "player" proposes a potential matching (q_v, t_i), and a "rival" responds with a superior matching for t_i, thus improving the overall matching (q_i, t_i) and releasing q_v. Consequently, the player advances to the next round, suggesting a new match-

ing (q_v, t_{i+1}) for q_v. This iterative process results in a series of matchings generated by the algorithm.

It is evident that B&F essentially tackles the Linear Assignment Problem (LAP). However, it lacks the ability to converge when the order of candidate matchings changes in the input. In contrast, both the Hungarian Algorithm [212] and the Hopcroft-Karp Algorithm [172], which we will discuss later, efficiently solve the LAP in polynomial time, providing the optimal converged solution.

Hungarian Algorithm. The Hungarian algorithm [212] stands as a venerable and potent tool for solving LAP in $O(N^3)$ time and $O(N^2)$ space. Given two sets A and B, and a cost function $C : A \times B \in R$, LAP aims to find a bijection between the two sets so that the sum of the cost is minimized or maximized. We formulate the function matching problem as LAP where C refers to function similarities and the objective is to find a bijection between two binaries where the sum of the function similarities is maximized. Given our dataset's scale, both the time and space complexities are acceptable, ensuring that we can consistently unearth the optimal linear assignment solution within polynomial time. This algorithm also has extensive research on parallelization. Studies conducted by Wang et al. and Date et al. [273, 106] indicate the existence of parallel versions of the Hungarian algorithm designed for both CPU and GPU computing architectures.

Nevertheless, the simplification of the QAP to LAP leads to a loss of second-order information (edge affinity), resulting in suboptimal function matching.

Hopcroft-Karp Algorithm. The Hopcroft-Karp algorithm [172] is another approach to solving LAP. It demonstrates superior time and space complexity, particularly in scenarios

of sparse-constrained matching. Here, we define a LAP as sparse-constrained when, for an element a in set A, the number of elements b in set B to which it can be matched is significantly smaller than the size of B. In this case, the worst-case time complexity is $O(|E|\sqrt{|A+B|})$ and space complexity is $O(|A+B|)$ where E denotes matching candidates. In our problem, since we already have an initial similarity score between the functions, we can disregard matching candidates with very low similarity scores, thereby creating a sparse-constrained matching scenario.

As both the Hungarian and Hopcroft-Karp algorithms can provide optimal solutions for LAP within polynomial time, the output of these two algorithms would be equivalent. Given that Hungarian is the more commonly used algorithm for solving LAP and it naturally supports GPU paralleling, we focus our evaluation solely on Hungarian while providing theoretical analysis for the Hopcroft-Karp algorithm here.

Spectral Matching. Spectral matching, as described by Leordeanu et al. [190], leverages the eigenvalues, also known as the spectrum of a matrix, to seek the solution. This method relaxes the matching matrix M to a real unit norm matrix, enabling the utilization of Rayleigh's quotient theorem for solving the objective function outlined in Equation 2.1.

This method lacks theoretical support for directed graphs because it relies on a power iteration algorithm to compute the leading eigenvector of a matrix. This algorithm necessitates the matrix to be diagonalizable, a condition typically guaranteed by symmetry. As directed graphs may not exhibit this symmetric characteristic, the method's applicability to such graphs is limited.

This method theoretically enables GPU parallelization by relying solely on matrix multiplication to compute the leading eigenvector of the affinity matrix. However, a challenge arises regarding memory utilization when parallelizing it. Given the potentially large size of the affinity matrix, sparse matrix storage is typically employed. Nevertheless, existing sparse matrix multiplication algorithms in PyTorch may consume significant memory [56], posing a potential bottleneck when executed on the GPU.

Reweighted Random Walks. Reweighted Random Walks for Graph Matching (RRWM) [97] was introduced in 2010, built upon the concept of associated graphs outlined in spectral matching [190]. The associated graph is derived from the affinity matrix, encapsulating both node and edge affinities. Leordeanu et al. [190] proposed that matching two graphs is analogous to identifying a cluster within the associated graph formed by their affinity matrices. Consequently, a random walk approach can be employed on this associated graph to discover such clusters.

This paper introduces the affinity-preserving random walks method, which maintains affinity consistency within the Markov chain during random walks by incorporating a new absorbing node. Affinity-preserving random walks are shown to be equivalent to spectral relaxation in spectral matching [190]. To address the absence of a one-to-one mapping constraint in spectral relaxation, the authors propose the reweighting random walks method. This method entails moving directly to the next node rather than traversing along edges, a concept termed personalized jump, as introduced by Haveliwala [149]. During this step, a PageRank calculation algorithm determines the most influential node. Additionally, a double stochastic process is applied to fulfill the two-way constraint.

While the paper's evaluations solely utilize undirected graphs, there is no explicit indication that the method does not support directed graphs. However, it is essential to note that its current implementation is entirely CPU-based. As evident from its requirements, the algorithm necessitates affinity and the construction of associated graphs, potentially leading to significant memory usage. Hence, this approach might not be suitable for our problem, especially considering the memory-intensive nature of constructing the associated graph.

Factorized Graph Matching. Zhou et al. [316] pioneered factorized graph matching (FGM) in 2012, revealing that the majority of affinity matrices in graph matching problems can be decomposed into Kronecker products of smaller matrices. This methodology presents several advantages:

- It bypasses the computationally intensive $O(n^4)$ calculation typically required for the affinity matrix.
- The factorization introduces a novel approximation to the graph matching problem, augmenting existing graph matching algorithms.

However, this algorithm's implementation relies on CPU processing. In our tests conducted on graphs containing approximately 200 nodes and 400 edges, FGM took over 1000 seconds to complete, significantly slower compared to other algorithms. Despite its extended runtime, it yielded only marginal improvements in recall while slightly sacrificing accuracy.

Graduated Assignment. The graduated assignment (GA) by Gold and Rangarajan [139] can work with only adjacency matrices without the need to build the large affinity matrix

required in the Lawler's form. It leverages Taylor expansion and transforms the QAP into a series of LAPs.

The relaxation method employed in GA is known as doubly stochastic relaxation. It transforms the assignment matrix M where M_{ai} is either 0 or 1 and satisfies the constraint $\sum_{a=1}^{A} M_{ai} = 1$ for all i and $\sum_{i=1}^{I} M_{ai} = 1$ for all a, to a doubly stochastic matrix by dropping the binary constraint (0 or 1) while still satisfying the rest constraints. This relaxation essentially sidesteps the strict one-to-one mapping constraint, allowing for a solution in a continuous space through continuous methods.

GA aims to minimize the following objective function:

$$\begin{aligned} E_{awg}(M) = &- \frac{1}{2} \sum_{a=1}^{A} \sum_{i=1}^{I} \sum_{b=1}^{A} \sum_{j=1}^{I} M_{ai} M_{bj} C^{(2)}_{aibj} \\ &+ \alpha \sum_{a=1}^{A} \sum_{i=1}^{I} M_{ai} C^{(1)}_{ai} \end{aligned} \tag{5.1}$$

where M is the assignment matrix, and G_{ab} and g_{ij} are the adjacency matrices of the graphs. $C^{(2)}_{aibj}$ denotes the compatibility between the edge (a, b) and (i, j), and $C^{(1)}_{ai}$ denotes the compatibility between the node a and node i. α is a tunable parameter. In the work of Gold et al. [139], an example is provided where $C^{(2)}_{aibj} = 1 - 3|G_{ab} - g_{ij}|$ for randomly generated graphs. It is explained that this definition is equivalent to $C^{(2)}_{aibj} = G_{ab} \times g_{ij}$ when $G, g \in \{0, 1\}$. In the algorithm, it is stipulated that adjacency matrices for graphs must be symmetric, implying an undirected graph in the problem definition. We further explain why GA does not support directed graphs. In Equation 5.1, the adjacency matrices G and g are used to calculate the assignment reward. Consider a directed edge from node a to node b in graph G and another directed edge from node i to node j in graph g. Hence,

$G_{ab} = g_{ij} = 1$ and $G_{ba} = g_{ji} = 0$. By examining the derivative function from GA [139][2]:

$$Q_{ai} = \frac{\partial E_{wg}}{\partial M_{ai}}\bigg|_{M=M^0} = \sum_{b=1}^{A}\sum_{j=1}^{I} M_{bj}C_{aibj} \tag{5.2}$$

it becomes evident that

$$Q_{bj} = \sum_{a=1}^{A}\sum_{i=1}^{I} M_{ai}C_{bjai} = 0 \tag{5.3}$$

Hence, we can only establish the correspondence between nodes a and i, disregarding the matching between nodes b and j.

We present the graduated assignment algorithm in Algorithm 1[3]:

β serves as the control parameter for the continuation method, with β_0 denoting its initial value, β_f indicating the maximum value, and β_r representing the rate at which β is increased. I_0 and I_1 signify the maximum number of iterations permitted for the two while-loops. G and g denote the adjacency matrices of the two graphs.

GA has several unique properties that make it a good candidate for the function matching problem. First, GA relaxes the NP-hard QAP into a series of LAPs and iteratively refines the assignment matrix. This allows us to resume GA from a middle point of the refinement process if we can provide a good initial assignment matrix. Second, its space complexity is only $O(N^2)$ and supports sparse matrices by nature, further reducing the memory pressure when processing large graphs. Third, it is a classic graph matching algorithm meaning it does not require training and can be applied to different types of graphs.

[2]We opt for the original derivative function instead of one incorporating nodes' features. This choice facilitates a more convenient demonstration of the asymmetry in gradient transfer.

[3]In the original paper [139], there are slack rows and slack columns to accommodate unmatched elements. Consequently, it defines a $\hat{M}$ matrix containing the two slacks, and the iteration variables i and a in the while loop of Sinkhorn will end at $I + 1$ and $A + 1$.

Algorithm 1 Graduated Assignment

Require: β_0, β_f, β_r, I_0, I_1, G, g

$\beta \leftarrow \beta_0$

$M_{ai} \leftarrow 1 + \epsilon$

while $\beta < \beta_f$ **do**

while M does not converges **and** # of iterations $\leq I_0$ **do**

$Q_{ai} \leftarrow \sum_{b=1}^{A} \sum_{j=1}^{I} M_{bj}^0 C_{aibj}$

$M_{ai}^0 \leftarrow \exp(\beta Q_{ai})$

while M does not converges **and** # of iterations $\leq I_1$ **do**

$M_{ai}^1 \leftarrow \frac{M_{ai}^0}{\sum_{i=1}^{I} M_{ai}^0}$

$M_{ai}^0 \leftarrow \frac{M_{ai}^1}{\sum_{a=1}^{A} M_{ai}^1}$ ▷ This while-loop is Sinkhorn

end while

end while

$\beta \leftarrow \beta_r \beta$

end while

Neural GM Algorithms. Neural-based graph matching algorithms trace their origins back to Caetano et al.'s work [88]. The fundamental premise underlying these approaches is to adaptively learn the weights of edges and nodes in a graph to facilitate effective matching.

In the original problem formulation, each edge and node possesses a unique affinity. However, the importance of these edges and nodes may vary across different graph structures. Consequently, by learning the weights associated with edges and nodes, these algorithms discern which elements are crucial for matching and which are less significant.

This adaptive weight learning mechanism enables the algorithm to effectively discriminate between important and less important edges and nodes, thereby enhancing the quality of graph matching results.

In recent research [301, 274, 276, 275, 193], more advanced convolutional layers are employed to extract intricate features from graphs. Subsequently, these features are utilized for matching refinement through embedding layers or conventional techniques such as Sinkhorn normalization [256].

We chose to initiate our approach with a classic graph matching algorithm instead of a neural one. This decision was influenced by our examination of existing neural network graph matching methods [301, 274, 276, 275, 193], many of which rely on the affinity matrix as their primary input, as detailed in Section 3.1. However, the affinity matrix scales at $O(N^4)$, where N denotes the size of the node set for two graphs. Given that our scenarios often involve graphs with thousands of nodes, storing such a large affinity matrix becomes impractical, especially considering the challenge of implementing these algorithms using sparse matrices.

Moreover, these neural algorithms typically require extensive datasets for training, posing another challenge. Real-world call graphs exhibit substantial variability due to architectural disparities, compiler optimizations, and other factors. Consequently, neural networks trained offline might struggle to address our problem effectively, lacking the adaptability needed to handle such diverse graph structures.

Additionally, some neural algorithms [301, 274, 130] are essentially adaptations of classic algorithms. Therefore, we reasoned that starting with a classic algorithm would provide a solid foundation for our approach.

The Deep Graph Matching Consensus (DGMC) [130] algorithm, introduced by Fey et al. in 2020, is chosen for our special inspection as its unique advantages. Unlike other GM algorithms employing neural networks, which primarily target solving Lawler's Quadratic Assignment Problem (QAP), requiring large affinity matrices as input—prohibitive at our scale—DGMC takes a different approach. By leveraging gradients and local information akin to graduated assignment, it refines the correspondence matrix, rendering it more scalable.

DGMC operates through two principal stages: a Graph Neural Network (GNN) [247] local feature matching phase and a global correspondence refinement phase. Initially, a set of ground truth correspondences guides the training of the GNN, yielding localized, permutation-equivariant node representations. Subsequently, another GNN identifies violations within the neighborhood consensus and iteratively enhances the correspondence matrix.

However, we ultimately ruled out the adoption of this algorithm after conducting several experiments. A key challenge arose from its reliance on ground truth matching as input for training the Graph Neural Network (GNN). Acquiring such ground truth is often problematic in real-world programs, given the variability in function embeddings across different architectures or compiler configurations. Moreover, our experimentation revealed

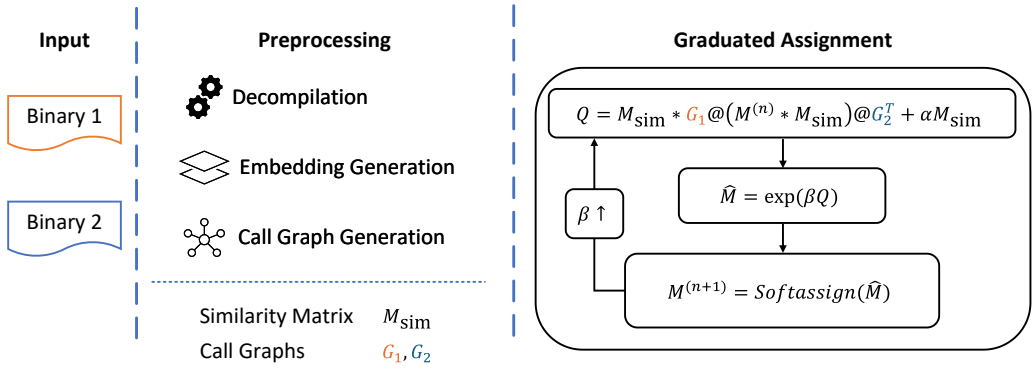


Figure 5.2: Overview of GrassDiff

another significant drawback: the algorithm's online training phase proved to be excessively time-consuming, prompting us to abandon its further pursuit.

5.3 Design of GrassDiff

The system overview is shown in Figure 5.2. It consists of two stages. The first stage, preprocessing, disassembles and decompiles the functions into pseudo-C code, as well as extracts function call graphs. These decompiled functions are then fed into a transformer model for embedding generation. The transformer model is the public-available CodeBERT model [129] finetuned on the StateFormer [225] ARM64 and x64 dataset. The second stage, graduated assignment, leverages the pairwise function similarities according to the embeddings and finds optimal function assignments.

5.3.1 Preprocessing

During the preprocessing stage, binary files are decompiled into pseudo-C code, and call graphs are extracted. The rule-based decompiler is suitable for complex data flow and control flow analysis, and recovers high-level semantics. We use Ghidra for decompilation and call graph extraction.

5.3.2 Embedding Generation

Given decompiled functions, we use the finetuned CodeBERT model to generate function embeddings. We also compute a similarity matrix between every function pair from the two binaries.

5.3.3 GA+: Improved Graduated Assignment

The original GA algorithm suffers from the following problems that hinder it from being applied to the function matching problem. First, it initializes the assignment matrix M_0 to all ones, so it has to start from a small β_0 to gradually refine the assignment matrix, taking a lot more iterations to converge. Second, the original edge affinity function of GA considers only binary compatibility between two edges, i.e., compatible or incompatible. In the function matching problem, how well two call edges match depends on the similarities between the two corresponding function pairs. Third, the original GA exhibits a time complexity of $O(N^4)$, primarily attributes to the four-dimensional computation of the affinity function $C^{(2)}_{aibj}$. Here, N denotes the number of nodes in the graphs. Consequently, convergence time increases significantly for larger graphs, potentially spanning orders of

magnitude. To tackle these problems, we have introduced improvements to GA to make it more suitable for the function matching problem:

First, we pass the initial similarity matrix as M_0 rather than an all-ones matrix, and increase β_0 from 0.5 to 50, β_f from 10 to 1,000, and β_r to 2. As we discussed before, the initial similarity matrix is close to the final assignment matrix. Given the iterative refinement nature of GA, a good starting point allows us to skip quite some iterations. For the same reason, we give β_0, β_f, β_r a large value so that GA resumes from some point close to the convergence and we ensure a faster convergence and that the softassign converges to a fixed point.

Second, we improve the objective function of GA to incorporate function similarities and call edge similarities. Recall the objective function in Equation 5.1, we further define the objective function as following:

$$\begin{aligned} E(M) = &- \frac{1}{2} \sum_{a=1}^{A} \sum_{i=1}^{I} \sum_{b=1}^{A} \sum_{j=1}^{I} M_{ai} M_{bj} G_{ab} g_{ij} C_{ai}^{(1)} C_{bj}^{(1)} \\ &+ \alpha \sum_{a=1}^{A} \sum_{i=1}^{I} M_{ai} C_{ai}^{(1)} \end{aligned} \tag{5.4}$$

where C_{ai} denotes the function similarity between function a and i. In our objective function, the edge similarity $C_{aibj}^{(2)}$ is defined as $C_{aibj}^{(2)} = G_{ab} \times g_{ij} \times C_{ai}^{(1)} \times C_{bj}^{(1)}$ so that the edge similarity equals the product of the corresponding node similarities iff there is a call edge between function a and b, and a call edge between i and j. In this way, this objective function considers both first-order and second-order similarities.

Third, we vectorize the derivative calculation, which reduces the time complexity from $O(N^4)$ to $O(N^3)$, and can be easily parallelized on GPU[4].

$$
\begin{aligned}
Q_{ai} = \left.\frac{\partial E}{\partial M_{ai}}\right|_{M=M^0} &= \sum_{b=1}^{A}\sum_{j=1}^{I} M_{bj}^0 C_{aibj}^{(2)} + \alpha C_{ai}^{(1)} \\
&= \sum_{b=1}^{A}\sum_{j=1}^{I} M_{bj}^0 \times G_{ab} \times g_{ij} \times C_{ai}^{(1)} \times C_{bj}^{(1)} + \alpha C_{ai}^{(1)} \\
&= \sum_{b=1}^{A}\sum_{j=1}^{I} C_{ai}^{(1)} \times [G_{ab} \times (M_{bj}^0 \times C_{bj}^{(1)}) \times g_{ji}^T + \alpha] \\
\implies Q &= C^{(1)} \odot [G(M^0 \odot C^{(1)})g^T + \alpha]
\end{aligned} \tag{5.5}
$$

where $\odot$ represents the Hadamard product of matrices. We also employ sparse matrix multiplication to reduce memory usage for adjacency matrices down to approximately $O(N)$. This is critical when comparing a large binary with a relatively small library. A large binary can have more than 200,000 functions whose dense adjacency matrix requires approximately 160 GB of memory.

5.4 Evaluation

In this section, we evaluate the efficiency and effectiveness of different matching algorithms in cross-version, cross-compiler, cross-architecture, and cross-optimization-level scenarios, and how they perform on C++ programs where indirect calls are common. Furthermore, we conduct an ablation study to showcase the improvement of GA+ compared to the original GA, and a case study to evaluate GA+ in real-world security applications.

[4]via matrix multiplication. The upper bound is $O(N^3)$ but the best algorithm can achieve $O(N^{2.37286})$ [64].

5.4.1 Experimental Setup

Our experiments are conducted on AMD Ryzen 5900X, RTX 3080 Ti, and 128 GB memory. The decompiled code is generated by Ghidra v11.0.

Datasets. We use the dataset released by SigmaDiff [136]. Specifically, it contains coreutils, findutils, diffutils, GMP, and PuTTY, each of which contains different versions, and is compiled with different compilers, configurations, and on different architectures. All the binaries are stripped in the evaluation.

Implementation. For the function embedding model, we use the same pretrained weights and tokenizer provided by CodeBERT and thus skip the model pretraining. Java is one of the pretrained languages by CodeBERT, and it shares similar syntaxes and tokens with pseudo-C code. The CodeBERT model is then finetuned on the StateFormer ARM64 and x64 dataset with contrastive learning for function similarity detection. The ARM64 dataset contains binutils, coreutils, diffutils, and findutils with optimization levels O0 to O3. The x64 dataset contains binutils, coreutils, diffutils, findutils, and several real-world programs with different optimizations and obfuscations. All the baselines are implemented in PyTorch 2.1.

During the finetuning, we replace function names in decompiled code with a *¡FUNC¿* token so that the model will not give too much attention to function names and learn superficial features. We use NT-Xent loss [258] to pull similar function pairs closer and push dissimilar pairs farther away. The NT-Xent loss for a positive pair (i, j) is defined as:

$$\mathcal{L}_E = -\log \frac{\exp(\text{sim}(z_i, z_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\text{sim}(z_i, z_k)/\tau)} \tag{5.6}$$

where N is the number of positive pairs in a minibatch, $\text{sim}(\cdot,\cdot)$ denotes the cosine similarity, τ denotes a temperature parameter, and $\mathbb{1}_{[k\neq i]} \in \{0,1\}$ is an indicator function equalling to one iff $k \neq i$. That is, given a positive pair (i,j), all the other $2(N-1)$ functions in a minibatch are treated as negative samples.

Baselines. We compare our approach with different matching strategies: pure embedding-based, Back-and-forth (B&F), Linear Assignment (Hungarian), Spectral Matching, and BinDiff [57]. Note that DGMC is not included in our evaluation because it takes too long for semisupervised training and the performance is poor due to the lack of good training nodes.

Evaluation Metrics. We rely on debug symbols to recover function names, and use precision and recall to measure the effectiveness of function matching. We consider only the functions whose names appear in both binary files and appear only once in each binary file. For the pure embedding-based approach, for each function in one binary file, we take the most similar function in the other binary file as a match regardless of the similarity score. For the rest of the approaches that give one-to-one mapping, we take the matching results as they are. Note that the Back and Forth Game does not guarantee convergence, and we stop the matching when we can not find any pair that mutually agrees with each other. Spectral Matching runs on an affinity matrix, and we only consider the edges whose node similarity exceeds 0.7 and build a sparse affinity matrix. Otherwise, it would take too long to process a binary. However, this will also affect the recall of Spectral Matching.

Table 5.2: Cross-version function matching results.

		GA+ (26s)		**Embed (0.14s)**		**Hungarian (1.7s)**		**B&F (11.6s)**		**Spectral (218s)**		**BinDiff (11.4s)**	
		P	R	P	R	P	R	P	R	P	R	P	R
Coreutils	v5.93 - v8.1	0.930	**0.930**	0.855	0.855	0.865	0.865	0.910	0.826	**0.940**	0.580	0.742	0.716
	v6.4 - v8.1	0.932	**0.932**	0.863	0.863	0.874	0.874	0.918	0.835	**0.944**	0.615	0.764	0.738
Diffutils	v2.8 - v3.6	**0.915**	**0.915**	0.767	0.767	0.805	0.805	0.858	0.732	0.896	0.487	0.811	0.811
	v3.4 - v3.6	**0.997**	**0.997**	0.936	0.936	0.967	0.967	0.971	0.910	0.701	0.382	0.992	0.989
Findutils	v4.233 - v4.6	0.839	**0.838**	0.739	0.739	0.756	0.756	0.903	0.704	**0.932**	0.496	0.704	0.672
	v4.41 - v4.6	0.930	**0.928**	0.829	0.829	0.850	0.850	0.929	0.798	**0.956**	0.463	0.802	0.783
GMP	v6.0.0 - v6.2.1	0.932	**0.932**	0.858	0.858	0.906	0.906	0.971	0.822	0.974	0.642	**0.985**	0.932
	v6.1.1 - v6.2.1	0.943	**0.943**	0.895	0.895	0.930	0.930	0.979	0.863	**0.990**	0.366	**0.990**	0.938
PuTTY	v0.75 - v0.77	0.921	**0.919**	0.823	0.823	0.855	0.855	**0.938**	0.776	0.695	0.018	0.881	0.871
	v0.76 - v0.77	0.784	**0.764**	0.658	0.658	0.681	0.681	**0.815**	0.611	0.536	0.013	N/A	N/A

5.4.2 Accuracy

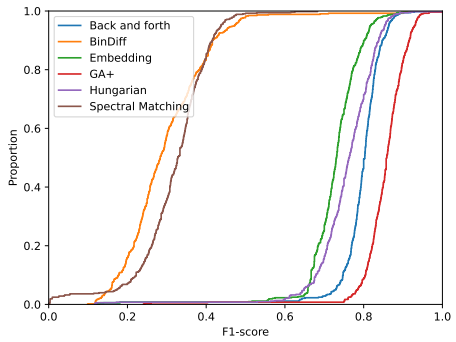
Cross-version. In this experiment, we evaluate how different approaches perform in matching binaries of two different versions. The binary pairs in this evaluation are built by GCC-5.4 on x64 with the default optimization level. The performance and efficiency are reported in Table 5.2. As shown, GA+ outperforms all the baselines in terms of recall, and Spectral Matching is slightly better in terms of precision in some cases. The "N/A" in the table means BinDiff crashed and no matching results were obtained. Overall, the function embeddings are able to find the matched functions pretty accurately. Hungarian is about 3% to 13% better than the pure embedding-based approach, and GA+ is 2% to 15% better than Hungarian. Spectral Matching has very good precision but it fails to match many functions.

The reported efficiency represents the time each approach takes for function matching of the whole cross-version dataset (over 500 MB) based on the similarity matrix. GA+ and embedding-based are measured on GPU while the rest are measured on CPU with multi-core support. Note that each binary is rather small (a few megabytes at most) and thus does not contain a lot of functions, so all these approaches can finish in a reasonable amount of time. BinDiff is evaluated on CPU. Spectral Matching is also evaluated on CPU because of its large memory requirement for the affinity matrix. Although using a sparse matrix alleviates the memory pressure for the affinity matrix, the implementation of sparse matrix multiplication in PyTorch still requires a significant amount of memory and thus evaluating its efficiency in GPU is nontrivial and we will leave it as future work.

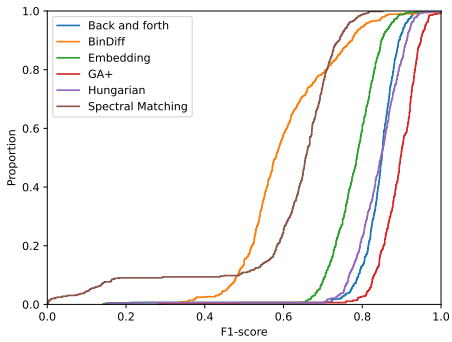
Cross-optimization. We then evaluate the performance on binaries compiled by GCC-5.4 on x64 with different optimization levels. Everything else including version and architecture is kept the same. Figure 5.3 presents the Cumulative Distribution Function (CDF) figures of the F1-scores. Again GA+ outperforms all the baselines in terms of F1-score.

Table 5.3: Cross-compiler function matching results.

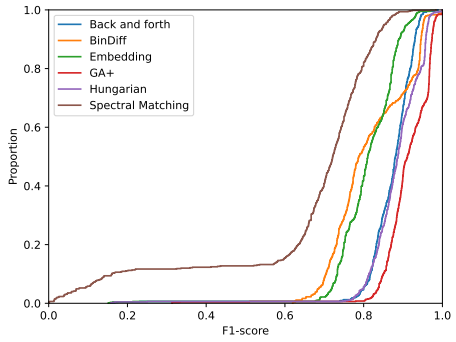
	GA+ (2.8s)		**Embed (0.02s)**		**Hungarian (0.23s)**		**B&F (1.6s)**		**Specrtral (11s)**		**BinDiff (5.07s)**	
	P	R	P	R	P	R	P	R	P	R	P	R
Coreutils	0.884	**0.884**	0.793	0.793	0.826	0.826	**0.927**	0.726	0.886	0.251	0.363	0.245
Diffutils	**0.962**	**0.962**	0.868	0.868	0.912	0.912	0.947	0.827	0.881	0.198	0.421	0.362
Findutils	0.895	**0.895**	0.792	0.792	0.844	0.844	0.922	0.718	**0.931**	0.178	0.337	0.247
GMP	0.843	**0.837**	0.687	0.687	0.820	0.820	0.901	0.621	0.907	0.221	**0.935**	0.822
PuTTY	0.910	**0.876**	0.674	0.674	0.759	0.759	0.925	0.617	**0.960**	0.100	0.404	0.328



(a) O0 vs. O3



(b) O1 vs. O3



(c) O2 vs. O3

Figure 5.3: Cross-optimization-level Function Matching F1-score CDF

Cross-compiler. We also conduct experiments between binaries compiled by GCC-5.4 and Clang-3.8 with the default optimization level on x64. Table 5.3 shows the performance of different matching approaches. GA+ in this task still outperforms all the baselines except that the precision is slightly behind Spectral Matching.

Cross-architecture. Likewise, the cross-architecture performance shown in Table 5.4 again shows the superiority of GA+ approach. Note that although the pure embedding-

based approach performs poorly on PuTTY software, GA+ is still able to match between the two binaries rather accurately, yielding a 45% improvement in terms of precision.

Table 5.4: Cross-architecture function matching results.

	GA+ (2.4s)		**Embed (0.03s)**		**Hungarian (0.35s)**		**B&F (2.4s)**		**Specrtral (11s)**		**BinDiff (8.6s)**	
	P	R	P	R	P	R	P	R	P	R	P	R
Coreutils	0.889	**0.889**	0.759	0.759	0.796	0.796	0.917	0.686	**0.934**	0.191	0.326	0.213
Diffutils	0.951	**0.951**	0.814	0.814	0.874	0.874	**0.961**	0.742	0.955	0.360	0.437	0.357
Findutils	0.920	**0.911**	0.709	0.709	0.814	0.814	0.920	0.646	**0.965**	0.243	0.336	0.242
GMP	0.496	0.469	0.306	0.306	0.455	0.455	0.673	0.155	0.250	0.002	**0.952**	**0.804**
PuTTY	**0.963**	**0.876**	0.516	0.516	0.602	0.602	0.894	0.395	0.910	0.062	0.712	0.483

C++ Programs. To assess the impact of indirect calls on various methodologies, we conducted additional evaluations on five C++ programs sourced from SigmaDiff [136]. It is worth noting that while SigmaDiff conducts differential analysis at the token level, our study focuses on the function level. The findings of our investigation are detailed in Table 5.5. Even without specific optimization for indirect calls in the C++ program, GA+ exhibits impressive performance in both precision and recall for this task.

Table 5.5: C++ program evaluation

	Name	**Size**	**#Indirect Calls**	**#Nodes in Binaries**		**#Edges in Binaries**		**R**	**P**	**F1**
Stockfish 14 vs. 15	stockfish	21.5M	88	454	458	227	229	0.998	0.993	0.996
Xerces-c 3.0.0 vs. 3.2.4	libxerces-c.so	3.6M	3733	6414	6645	3208	3323	0.974	0.942	0.958
Thrift 0.13.0 vs. 0.17.0	libthrift.so	797K	1435	1232	1304	616	652	0.978	0.959	0.968
	libthriftz.so	166K	152	277	321	139	161	0.996	0.993	0.995
	libthriftqt5.so	64K	124	77	88	39	50	1.0	1.0	1.0

5.4.3 Efficiency and Scalability

We evaluate the efficiency and scalability of in this section. Specifically, we measure the time GA+ takes to process a binary pair with the geometric mean of the number of functions in each binary pair. We also report peak CUDA memory usage. As Figure 5.4 shows, the time and memory usage scale well with the geometric mean of the number of functions. For small binaries with less than 3,000 functions, each match takes less than one second in most cases. Even for large binaries with an average of 30,000 functions (matching one small library with 1,700 functions with a large 200 MB binary with 550,000 functions), GA+ takes only 10 minutes. This evaluation shows that GA+ has good efficiency and scalability. Note that although PyTorch reported using 40 GB of GPU memory, the experiment was still run on a single GTX 3080 Ti (with 11 GB of available GPU memory). The shared memory between the GPU and the host makes large-scale graph matching possible, significantly enhancing our practical capabilities.

5.4.4 Ablation Study

We conduct an ablation study to evaluate the improvements brought by our modifications compared to the vanilla graduated assignment algorithm. The vanilla GA still adopts our vectorization and runs on CUDA because otherwise, the four nested for-loop take days to finish even when numba [188] is applied. It uses Equation 5.1 as the objective function, i.e., it considers first-order function similarities and topology, but no call edge similarities. Table 5.6 shows the averaged precision and recall of the four experiments. The time represents how long each approach takes to process the whole test set. This experiment

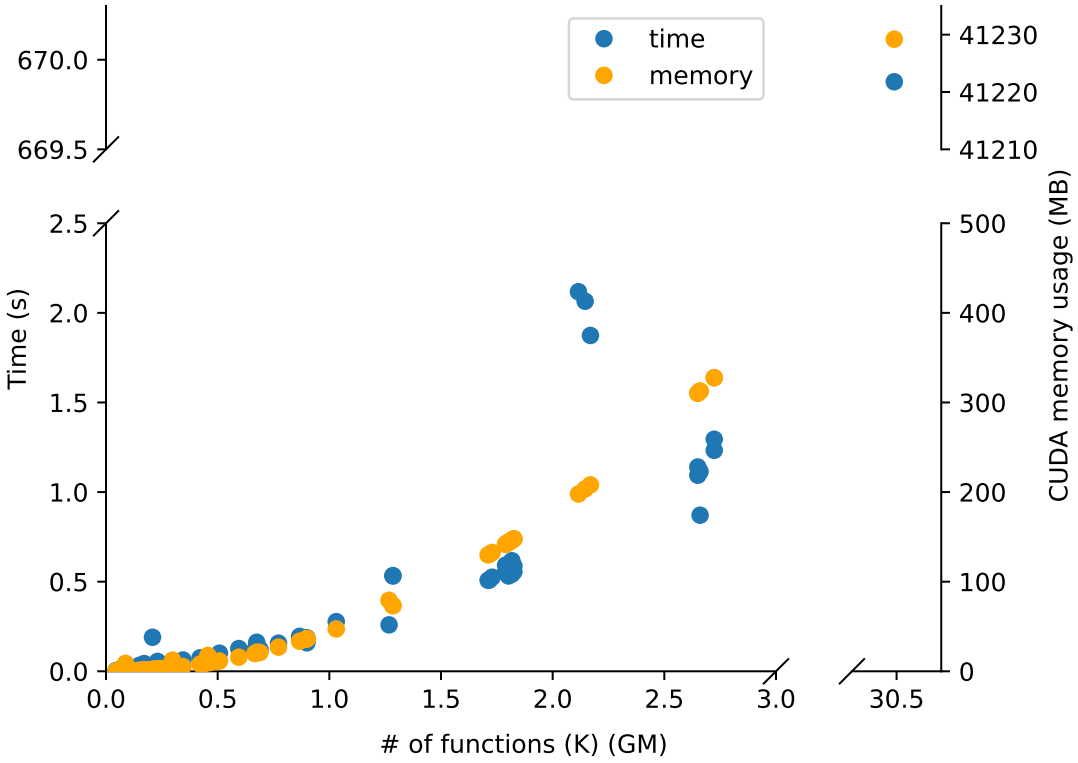


Figure 5.4: Correlation between the number of functions and time and CUDA memory usage. The X-axis represents the number of functions divided by 1,000. GM: geometric mean.

shows that GA+ achieved a 10x to 20x speedup compared to the original GA just owing to faster convergence. The vectorization speedup was immeasurable because it took too long for the original GA to process the dataset.

Table 5.6: Average Precision and Recall of GA+ vs Original GA

	GA+			**Original GA**		
	P	R	Time	P	R	Time
X_Version	**0.912**	**0.910**	26.0s	0.905	0.902	385s
X_OPT	**0.907**	**0.867**	52.0s	0.887	0.839	931s
X_Compiler	0.844	0.819	2.4s	**0.867**	**0.855**	37s
X_Arch	**0.899**	**0.891**	2.8s	0.888	0.881	44s

5.4.5 Case Study

We further evaluate our approach in a real-world security application: vulnerability detection. In this study, we re-evaluate the motivating example with GA+ and extend the experiment to more libraries and binaries. Specifically, We compiled libexpat-2.3.0, SQLite-3.30.1, and OpenSSL-1.1.1a with GCC-13 and MSVC-2022 on x64 with the default optimization. We collected 5, 22, and 21 CVEs and their corresponding vulnerable function names respectively from these libraries. We also compiled nginx with static-linked OpenSSL-1.1.1a with GCC-13. Nginx has over 6,600 functions and libcrypto.dll has over 7,000 functions. Binaries and Libraries are disassembled by IDA Pro and function embeddings are generated by jTrans. This experiment aims to evaluate a) the performance of GA+ on suboptimal embeddings, and b) how well GA+ can identify vulnerable functions between two libraries and when libraries are static-linked.

For the evaluation, we compare libraries and binaries compiled by GCC-13 with libraries compiled by MSVC-2022, and evaluate if jTrans and GA+ can find the vulnerable functions in top K best-matching results. Figure 5.5 shows that jTrans has difficulty understanding assembly code generated by unseen compilers, and even the top 100 results are not very satisfactory. On the other hand, GA+ is still able to identify correct matches almost always accurately. Moreover, GA+ can correctly identify vulnerable functions even when they are static-linked in a large binary.

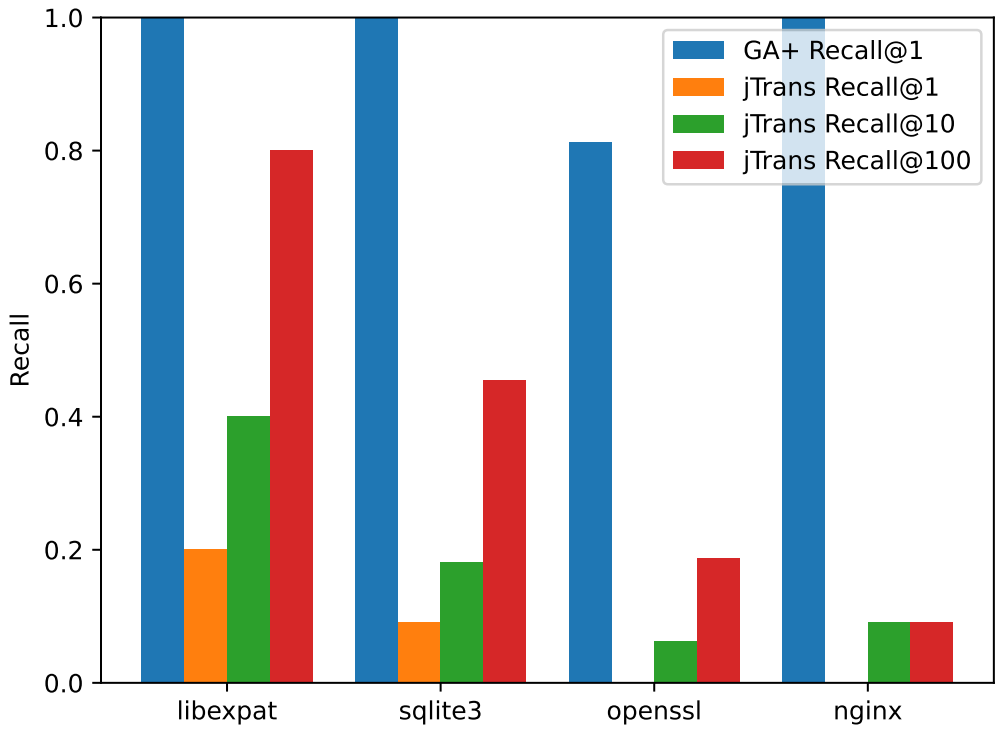


Figure 5.5: CVE Function Detection Recall

5.5 Discussion

In this section, we discuss the following limitations of GA+.

Scalability. Due to the graph matching nature, even though we try to improve the efficiency and reduce memory usage, directly matching two large binaries is still challenging. There is a line of work [263, 79, 191] that sparsifies the softassign operation, but is not enough for the indefinite-growing file size.

Large scale search. Vector search has become so efficient that finding similar functions in a vector database containing hundreds of millions of functions takes less than a second. With graph matching, however, one has to perform a pairwise comparison so that large-scale efficiency is far behind embedding-based approaches. However, we argue that it is a price we have to pay to achieve precise one-to-one mapping. Moreover, we can use the vector search result as a filter to find library or binary candidates, and then perform GA+ on candidates to achieve both efficiency and accuracy.

Graph deformation. Different compilers, compilation options, function inlining, C++ templates, and indirect calls can potentially change call graphs and reduce the performance of GA+. However, even when we cannot recover any call edge between functions, GA+ degrades to linear assignment and still performs relatively well. Additionally, data nodes such as string references, constants and virtual tables can be added to the graph to aid the matching. Matching can happen at a finer granularity, e.g., at basic block-level, to meet different needs and mitigate the graph deformation issue.

5.6 Conclusion

In this chapter, we have proposed GrassDiff, a novel function matching framework that leverages binary topology and significantly increases accuracy without too much compromise in efficiency. Our evaluation results show 5% to 20% improvements compared to the pure embedding-based approach. When used for CVE detection, GrassDiff significantly improves Recall@1 even when the embedding quality is suboptimal. Additionally, matching a library in a large 200 MB binary only takes 10 minutes. These results show that GrassDiff is both accurate and scalable.

Chapter 6

Conclusions

Software supply chain security is vital in modern software lifecycle management, but a critical missing piece, accurate and efficient SBOM generation, hinders the wide adoption of SBOMs. In this thesis, we address the SBOM generation problem for both source code and binary code. We first identified the flaws in the existing open-source SBOM tools and proposed best practices for metadata-based SBOM generation. For Binary SBOM, we proposed DeepDi to address the efficiency issue by bringing hundreds of times speedup to disassemblers, and we proposed GrassDiff to improve function matching in terms of accuracy and scalability.

For the metadata-based SBOM generation, we conducted the first large-scale differential analysis to examine the correctness of SBOM generation solutions. We generated SBOMs using four popular SBOM generators for 7,876 open-source projects and systematically studied the correctness of these SBOMs. Our evaluation uncovered significant deficiencies in current SBOM generators. Additionally, we identified the design flaws in

each SBOM generator, and devised a parser confusion attack against these generators, introducing a new path for injecting malicious, vulnerable, or illegal packages. Finally, based on our findings, we established best practices for creating SBOM generators and introduced a benchmark to aid their development.

DeepDi, a novel deep learning based technique for disassembly that achieves both accuracy and efficiency, is proposed to improve SBOM generation efficiency. Our experimental results have shown that DeepDi's accuracy is comparable to the state-of-the-art commercial tools and research prototypes, and it is two times faster than IDA Pro, and its GPU version is 350 times faster. DeepDi is able to generalize to unseen binaries, and counter obfuscations and certain adversarial attacks.

GrassDiff leverages binary topology and significantly increases binary function matching accuracy without too much compromise in efficiency. Our evaluation results show 5% to 20% improvements compared to the pure embedding-based approach. When used for CVE detection, GrassDiff significantly improves Recall@1 even when the embedding quality is suboptimal. Additionally, matching a library in a large 200 MB binary only takes 10 minutes. These results show that GrassDiff is both accurate and scalable.

6.1 Final Thoughts and Future Works

As software developments rely more and more on the software supply chain, we are facing more challenges in its security. SBOM is one way of enhancing the software supply chain security, provided that SBOMs can be accurately and efficiently generated. There are still many potential directions for better SBOM generation. For example, we

could incorporate advanced source code, binary code, and reach-to analyses to determine if a vulnerable function is reachable during runtime to reduce false alarms. Moreover, we could also incorporate patch presence tests to evaluate if a fix has been backported into the project without bumping the dependency versions. Downstream applications of SBOM can also be improved, for example, CPE and PURL are two popular ways to locate packages, but vulnerability databases do not cooperate well with them.

My future work will focus on incorporating source code analysis and reachability tests. I will try more techniques to evaluate reachability in both source code and binary, and reduce false alarms of vulnerabilities.

Bibliography

[1] 2021 state of the software supply chain: Open source security and dependency management take center stage. https://blog.sonatype.com/2021-state-of-the-software-supply-chain.

[2] Actionscript technology center. http://www.adobe.com/devnet/actionscript.html.

[3] Akbuilder is the latest exploit kit to target word documents, spread malware. https://nakedsecurity.sophos.com/2017/02/07/akbuilder-is-the-latest-exploit-kit-to-target-word-documents-spread-malware/.

[4] Annual number of software packages impacted by supply chain cyber attacks worldwide from 2019 to 2023 ytd. https://www.statista.com/statistics/1375128/supply-chain-attacks-software-packages-affected-global/.

[5] The art of leaks: The return of fengshui. https://cansecwest.com/slides/2014/The%20Art%20of%20Leaks%20-%20read%20version%20-%20Yoyo.pdf.

[6] Aslr bypass apocalypse in recent zero-day exploits. https://www.fireeye.com/blog/threat-research/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html.

[7] avtest. https://www.av-test.org/en/statistics/malware/.

[8] Binary ninja, a new type of reversing platform. https://binary.ninja/.

[9] Chakracore javascript engine. https://github.com/Microsoft/ChakraCore.

[10] Chrome v8 engine. https://developers.google.com/v8/.

[11] chrome.webrequest. https://developer.chrome.com/extensions/webRequest.

[12] Control flow guard. https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx.

[13] Cve details. http://www.cvedetails.com/.

[14] Depedency track. https://dependencytrack.org/.

[15] Ecmascript js ast traversal functions. https://github.com/estools/estraverse.

[16] Ecmascript parsing infrastructure for multipurpose analysis. http://esprima.org/.

[17] The enhanced mitigation experience toolkit. https://support.microsoft.com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit.

[18] Executive order on improving the nation's cybersecurity. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/.

[19] fbench. https://www.fourmilab.ch/fbench/.

[20] Functionsimsearch. https://github.com/googleprojectzero/functionsimsearch.

[21] Geekpwn. http://2017.geekpwn.org/1024/en/index.html.

[22] Ghidra – software reverse engineering framework. https://www.nsa.gov/resources/everyone/ghidra/.

[23] Github dependency graph. https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph.

[24] Http mitm proxy. https://github.com/joeferner/node-http-mitm-proxy.

[25] The ida disassembler and debugger. https://www.hex-rays.com/products/ida/.

[26] Intel® 64 and ia-32 architectures software developer's manual, volume 2. http://tiny.cc/vskytz.

[27] International open standard (iso/iec 5962:2021) - software package data exchange (spdx). https://spdx.dev/.

[28] The javascript benchmark suite for the mordern web. https://developers.google.com/octane/.

[29] Javascriptcore. https://trac.webkit.org/wiki/JavaScriptCore.

[30] Lexer confusing attack. https://github.com/google/caja/wiki/JsControlFormatChars.

[31] Linux avx benchmarks. http://www.roylongbottom.org.uk/linux%20AVX%20benchmarks.htm.

[32] Microsoft sbom tool. https://github.com/microsoft/sbom-tool.

[33] Minimal version selection (mvs). https://go.dev/ref/mod#minimal-version-selection.

[34] National cybersecurity strategy implementation plan. https://www.whitehouse.gov/wp-content/uploads/2023/07/National-Cybersecurity-Strategy-Implementation-Plan-WH.gov_.pdf.

[35] Node.js. https://nodejs.org/en/.

[36] Owasp cyclonedx software bill of materials (sbom) standard. https://cyclonedx.org/.

[37] Package url. https://github.com/package-url/purl-spec.

[38] Peepdf: a python tool to explore pdf files. https://github.com/jesparza/peepdf.

[39] Proof-of-concept exploit for cve-2016-0189 (vbscript memory corruption in ie11). https://github.com/theori-io/cve-2016-0189.

[40] Pwn2own. https://en.wikipedia.org/wiki/Pwn2Own.

[41] Pytorch machine learning framework compromised with malicious dependency. https://thehackernews.com/2023/01/pytorch-machine-learning-framework.html.

[42] Researchers easily trick cylance's ai-based antivirus into thinking malware is 'goodware'. http://tiny.cc/qnijuz.

[43] Rop is dying and your exploit mitigations are on life support. https://www.endgame.com/blog/technical-blog/rop-dying-and-your-exploit-mitigations-are-life-support.

[44] Software bill of materials. https://www.ntia.gov/page/software-bill-materials.

[45] Software supply chain arsenal. https://tools.deepbits.com/.

[46] Spidermonkey javascript engine. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[47] Standard performance evaluation corporation. spec cpu2006 benchmark, 2006.

[48] Standard performance evaluation corporation. spec cpu2017 benchmark, 2017.

[49] Syft. https://github.com/anchore/syft.

[50] Trivy. https://trivy.dev/.

[51] Types of software bill of materials. https://www.cisa.gov/resources-tools/resources/types-software-bill-materials-sbom.

[52] Vbscript. https://en.wikipedia.org/wiki/VBScript.

[53] Vulnerability-exploitability exchange (vex)–an overview. https://www.ntia.gov/files/ntia/publications/vex_one-page_summary.pdf.

[54] Why 2023 is the year for software supply chain attacks. https://hadrian.io/blog/why-2023-is-the-year-for-software-supply-chain-attacks.

[55] Diaphora. http://diaphora.re/, 2022.

[56] Upgrading spgemm algorithm to resolve cusparse spgemm insufficient resources problem. https://github.com/pytorch/pytorch/issues/103820, 2023. [Accessed 26-04-2024].

[57] Zynamics bindiff. https://www.zynamics.com/bindiff.html, 2024.

[58] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks*, September 2013.

[59] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[60] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the sixth ACM conference on data and application security and privacy*, pages 183–194, 2016.

[61] Fahmi Abdulqadir Ahmed and Dyako Fatih. Security analysis of code bloat in machine learning systems. 2022.

[62] Mitsuaki Akiyama, Shugo Shiraishi, Akifumi Fukumoto, Ryota Yoshimoto, Eitaro Shioji, and Toshihiro Yamauchi. Seeing is not always believing: Insights on iot manufacturing from firmware composition analysis and vendor survey. *Computers & Security*, page 103389, 2023.

[63] Rahaf Alkhadra, Joud Abuzaid, Mariam AlShammari, and Nazeeruddin Mohammad. Solar winds hack: In-depth analysis and countermeasures. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–7. IEEE, 2021.

[64] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 522–539. SIAM, 2021.

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

[66] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In *European conference on Object-oriented programming*, pages 428–452. Springer, 2005.

[67] H. S. Anderson and P. Roth. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints*, April 2018.

[68] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 583–600, 2016.

[69] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 177–189. IEEE, 2017.

[70] Relja Arandjelovic and Andrew Zisserman. All about vlad. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1578–1585, 2013.

[71] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 356–367, 2016.

[72] Yameng Bai, Xingming Sun, Guang Sun, Xiaohong Deng, and Xiaoming Zhou. Dynamic k-gram based software birthmark. In *19th Australian Conference on Software Engineering (aswec 2008)*, pages 644–649. IEEE, 2008.

[73] Yunsheng Bai, Hao Ding, Yizhou Sun, and Wei Wang. Convolutional set matching for graph similarity. *arXiv preprint arXiv:1810.10866*, 2018.

[74] Sruthi Bandhakavi, Samuel T King, Parthasarathy Madhusudan, and Marianne Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium*, volume 10, pages 339–354, 2010.

[75] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. {BYTEWEIGHT}: Learning to recognize functions in binary code. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 845–860, 2014.

[76] Adam Barth, Collin Jackson, and John C Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88. ACM, 2008.

[77] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

[78] Andrew R Bernat and Barton P Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 9–16, 2011.

[79] Mathieu Blondel, Vivien Seguy, and Antoine Rolet. Smooth and sparse optimal transport. In *International conference on artificial intelligence and statistics*, pages 880–889. PMLR, 2018.

[80] Martial Bourquin, Andy King, and Edward Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, PPREW '13, New York, NY, USA, 2013. Association for Computing Machinery.

[81] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise independent permutations. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 327–336, 1998.

[82] Derek Bruening and Saman Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation.* PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering, 2004.

[83] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[84] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. *Botnet Detection*, chapter Automatically Identifying Trigger-based Behavior in Malware. 2007.

[85] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Towards automatically identifying trigger-based behavior in malware using symbolic execution and binary analysis. Technical Report CMU-CS-07-105, Carnegie Mellon University School of Computer Science, January 2007.

[86] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[87] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[88] Tiberio S. Caetano, Li Cheng, Quoc V. Le, and Alex J. Smola. Learning graph matching. In *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8, 2007.

[89] Bugra Cakir and Erdogan Dogdu. Malware classification using deep learning methods. In *Proceedings of the ACMSE 2018 Conference*, pages 1–5, 2018.

[90] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. Extract me if you can: Abusing pdf parsers in malware detectors. In *NDSS*, 2016.

[91] Curtis Carmony, Mu Zhang, Xunchao Hu, Abhishek Vasisht Bhaskar, and Heng Yin. Extract me if you can: Abusing PDF parsers in malware detectors. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*, February 2016.

[92] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[93] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS*, 2015.

[94] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, Huijie DENG, et al. Ropecker: A generic and practical approach for defending against rop attack. 2014.

[95] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[96] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[97] Minsu Cho, Jungmin Lee, and Kyoung Mu Lee. Reweighted random walks for graph matching. In Kostas Daniilidis, Petros Maragos, and Nikos Paragios, editors, *Computer Vision – ECCV 2010*, pages 492–505, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[98] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 99–116, 2017.

[99] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[100] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th International Conference on World Wide Web*, 2010.

[101] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 763–780. IEEE, 2015.

[102] Zhihua Cui, Fei Xue, Xingjuan Cai, Yang Cao, Gai-ge Wang, and Jinjun Chen. Detection of malicious code variants based on deep learning. *IEEE Transactions on Industrial Informatics*, 14(7):3187–3196, 2018.

[103] Charlie Curtsinger, Benjamin Livshits, Benjamin G Zorn, and Christian Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium*, 2011.

[104] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711. PMLR, 2016.

[105] Mark Daniel, Jake Honoroff, and Charlie Miller. Engineering heap overflow exploits with javascript. *WOOT*, 8:1–6, 2008.

[106] Ketan Date and Rakesh Nagi. Gpu-accelerated hungarian algorithms for the linear assignment problem. *Parallel Computing*, 57:52–72, 2016.

[107] Yaniv David, Nimrod Partush, and Eran Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. *SIGPLAN Not.*, 53(2):392–404, mar 2018.

[108] Yaniv David and Eran Yahav. Tracelet-based code search in executables. *Acm Sigplan Notices*, 49(6):349–360, 2014.

[109] DECAF Binary Analysis Platform - "Taking the jitters out of dynamic binary analysis". `https://code.google.com/p/decaf-platform/`.

[110] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*, pages 181–191, 2018.

[111] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[112] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. rev. ng: a unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 131–141, 2017.

[113] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

[114] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. Automating patching of vulnerable open-source software versions in application binaries. In *NDSS*, 2019.

[115] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, pages 2169–2185, 2017.

[116] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and Distributed System Security Symposium*, 2020.

[117] Yue Duan, Xuezixiang Li, Jinghan Wang, Heng Yin, et al. Deepbindiff: Learning program-wide code representations for binary diffing. 2020.

[118] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic Spyware Analysis. In *Proceedings of the 2007 Usenix Annual Conference (Usenix'07)*, June 2007.

[119] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, 2014.

[120] Andrzej Ehrenfeucht. An application of games to the completeness problem for formalized theories. *Fundamenta Mathematicae*, 49(2):129–141, 1961.

[121] Stefan Fankhauser, Kaspar Riesen, and Horst Bunke. Speeding up graph edit distance computation through fast bipartite matching. In *Proceedings of the 8th International Conference on Graph-Based Representations in Pattern Recognition*, GbRPR'11, page 102–111, Berlin, Heidelberg, 2011. Springer-Verlag.

[122] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 78–87. IEEE, 2014.

[123] Ben Feinstein, Daniel Peck, and I SecureWorks. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. *Black Hat USA*, 2007.

[124] Qian Feng, Aravind Prakash, Minghua Wang, Curtis Carmony, and Heng Yin. Origen: Automatic extraction of offset-revealing instructions for cross-version memory analysis. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 11–22, 2016.

[125] Qian Feng, Aravind Prakash, Heng Yin, and Zhiqiang Lin. MACE: High-coverage and robust memory analysis for commodity operating systems. In *2014 Annual Computer Security Applications Conference*, December 2014.

[126] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. Extracting conditional formulas for cross-platform bug search. In *Proceedings of ACM Asia Conference on Computer and Communications Security (ASIACCS'17)*, April 2017.

[127] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, 2016.

[128] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 23d ACM Conference on Computer and Communications Security(CCS'16)*, October 2016.

[129] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[130] Matthias Fey, Jan E. Lenssen, Christopher Morris, Jonathan Masci, and Nils M. Kriege. Deep graph matching consensus. In *International Conference on Learning Representations*, 2020.

[131] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1075–1092, 2020.

[132] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. In *Advances in Neural Information Processing Systems*, pages 3703–3714, 2019.

[133] Francesco Gadaleta, Yves Younan, and Wouter Joosen. Bubble: A javascript engine level countermeasure against heap-spraying attacks. In *International Symposium on Engineering Secure Software and Systems*, pages 1–17. Springer, 2010.

[134] Debin Gao, Michael K Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security: 10th International Conference, ICICS 2008 Birmingham, UK, October 20-22, 2008 Proceedings 10*, pages 238–255. Springer, 2008.

[135] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 896–899, 2018.

[136] Lian Gao, Yu Qu, Sheng Yu, Yue Duan, and Heng Yin. Sigmadiff: Semantics-aware deep graph matching for pseudocode diffing. 2024.

[137] Tianyu Gao, Xingcheng Yao, and Danqi Chen. Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821*, 2021.

[138] Shubham Girdhar. Frankfurt university of applied sciences.

[139] S. Gold and A. Rangarajan. A graduated assignment algorithm for graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(4):377–388, 1996.

[140] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Christiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. *NDSS (Feb. 2017)*, 2017.

[141] Andy Greenberg. The untold story of notpetya, the most devastating cyberattack in history. *Wired, August*, 22, 2018.

[142] Chris Grier, Lucas Ballard, Juan Caballero, Neha Chachra, Christian J Dietrich, Kirill Levchenko, Panayiotis Mavrommatis, Damon McCoy, Antonio Nappa, Andreas Pitsillidis, et al. Manufacturing compromise: the emergence of exploit-as-a-service. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 821–832. ACM, 2012.

[143] Boxuan Gu, Wenbin Zhang, Xiaole Bai, Adam C Champion, Feng Qin, and Dong Xuan. Jsguard: Shellcode detection in javascript. In *Security and Privacy in Communication Networks*. 2013.

[144] Yufei Gu, Yangchun Fu, Aravind Prakash, Zhiqiang Lin, and Heng Yin. OS-Sommelier: Memory-only operating system fingerprinting in the cloud. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, October 2012.

[145] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. {DEEPVSA}: Facilitating value-set analysis with deep learning for postmortem program analysis. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1787–1804, 2019.

[146] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. Lemna: Explaining deep learning based security applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 364–379, 2018.

[147] Stefan Haller, Lorenz Feineis, Lisa Hutschenreiter, Florian Bernard, Carsten Rother, Dagmar Kainmüller, Paul Swoboda, and Bogdan Savchynskyy. A comparative study of graph matching algorithms in computer vision. In *European Conference on Computer Vision*, pages 636–653. Springer, 2022.

[148] Blake Hartstein. Jsunpack: An automatic javascript unpacker. In *ShmooCon convention*, 2009.

[149] Taher H Haveliwala. Topic-sensitive pagerank. In *Proceedings of the 11th international conference on World Wide Web*, pages 517–526, 2002.

[150] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[151] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 63–72, 2011.

[152] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. "make it work, make it right, make it fast", building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14)*, July 2014.

[153] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. VDF: Targeted evolutionary fuzz testing of virtual devices. In *Proceedings of the 20th International Symposium on Research on Attacks, Intrusions and Defenses (RAID'17)*, September 2017.

[154] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[155] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[156] Tiancheng Hu, Zijing Xu, Yilin Fang, Yueming Wu, Bin Yuan, Deqing Zou, and Hai Jin. Fine-grained code clone detection with block-based splitting of abstract syntax tree. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 89–100, 2023.

[157] Xin Hu, Tzi-cker Chiueh, and Kang G Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 2009 ACM SIGSAC Conference on Computer and Communications Security*, pages 611–620, 2009.

[158] Xunchao Hu, Yao Cheng, Yue Duan, Andrew Henderson, and Heng Yin. JSForce: A forced execution engine for malicious javascript detection. In *Proceedings of the 13th EAI International Conference on Security and Privacy in Communication Networks (SecureComm'17)*, October 2017.

[159] Xunchao Hu, Qiang Guan, Heng Yin, and Nathan Debardeleben. A fine-grained, accountable, flexible, and efficient soft error fault injection framework for profiling application vulnerability. In *Proceedings of the 13th Workshop on Silicon Errors in Logic - System Effects (SELSE-13)*, March 2017.

[160] Xunchao Hu, Aravind Prakash, Jinghan Wang, Rundong Zhou, Yao Cheng, and Heng Yin. Semantics-preserving dissection of javascript exploits via dynamic js-binary analysis. In *Proceedings of the 19th Symposium on Research in Attacks, Intrusions and Defense (RAID'16)*, September 2016.

[161] Yikun Hu, Hui Wang, Yuanyuan Zhang, Bodong Li, and Dawu Gu. A semantics-based hybrid approach on binary code similarity comparison. *IEEE Transactions on Software Engineering*, 47(6):1241–1258, 2019.

[162] DongXing Huang, Yong Tang, Yi Wang, and ShuNing Wei. Toward efficient and accurate function-call graph matching of binary codes. *Concurrency and Computation: Practice and Experience*, 31(21):e4871, 2019.

[163] Jianjun Huang, Bo Xue, Jiasheng Jiang, Wei You, Bin Liang, Jingzheng Wu, and Yanjun Wu. Scalably detecting third-party android libraries with two-stage bloom filtering. *IEEE Transactions on Software Engineering*, 49(4):2272–2284, 2022.

[164] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[165] Rafiqul Islam, Ronghua Tian, Lynn Batten, and Steve Versteeg. Classification of malware based on string and function feature selection. In *2010 Second Cybercrime and Trustworthy Computing Workshop*, pages 9–17. IEEE, 2010.

[166] Wesley Jin, Sagar Chaki, Cory Cohen, Arie Gurfinkel, Jeffrey Havrilla, Charles Hines, and Priya Narasimhan. Binary function clustering using semantic hashes. In *2012 11th International Conference on Machine Learning and Applications*, volume 1, pages 386–391. IEEE, 2012.

[167] Xing Jin, Xunchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code injection attacks in html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS'14)*, November 2014.

[168] Clemens Jonischkeit and Julian Kirsch. Enhancing control flow graph based binary function identification. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, pages 1–8, 2017.

[169] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[170] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM'07)*, October 2007.

[171] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating emulation-resistant malware. In *Proceedings of the 2nd Workshop on Virtual Machine Security (VMSec'09)*, November 2009.

[172] R.M. Karp. An algorithm to solve the mxn assignment problem in expected time o (mn log n). Technical Report UCB/ERL M78/67, EECS Department, University of California, Berkeley, Sep 1978.

[173] Deborah S Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 346–356. IEEE, 2018.

[174] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614. IEEE, 2017.

[175] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245, 2011.

[176] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *International Conference on Computer Aided Verification*, pages 423–427. Springer, 2008.

[177] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR (Poster)*, 2015.

[178] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2016.

[179] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337. ACM, 2006.

[180] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-cloaking internet malware. In *Security and Privacy (SP), 2012 IEEE Symposium on*, 2012.

[181] Bojan Kolosnjaji, Ghadir Eraisha, George Webster, Apostolis Zarras, and Claudia Eckert. Empowering convolutional networks for malware classification and analysis. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 3838–3845. IEEE, 2017.

[182] David Korczynski and Heng Yin. Capturing malware propagations with code injections and code-reuse attacks. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*, October 2017.

[183] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97, 2006.

[184] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, pages 18–18, 2004.

[185] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019.

[186] Evangelos Ladakis, Giorgos Vasiliadis, Michalis Polychronakis, Sotiris Ioannidis, and Georgios Portokalidis. Gpu-disasm: A gpu-based x86 disassembler. In *International Conference on Information Security*, pages 472–489. Springer, 2015.

[187] Nathaniel Lageman, Eric D Kilmer, Robert J Walls, and Patrick D McDaniel. Bindnn: Resilient function matching using deep learning. In *International Conference on Security and Privacy in Communication Systems*, pages 517–537. Springer, 2016.

[188] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015.

[189] Quan Le, Oisín Boydell, Brian Mac Namee, and Mark Scanlon. Deep learning at the shallow end: Malware classification for non-domain experts. *Digital Investigation*, 26:S118–S126, 2018.

[190] M. Leordeanu and M. Hebert. A spectral technique for correspondence problems using pairwise constraints. In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, volume 2, pages 1482–1489 Vol. 2, 2005.

[191] Mengyu Li, Jun Yu, Tao Li, and Cheng Meng. Importance sparsification for sinkhorn algorithm. *arXiv preprint arXiv:2306.06581*, 2023.

[192] Xuezixiang Li, Yu Qu, and Heng Yin. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3236–3251, 2021.

[193] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*, pages 3835–3845. PMLR, 2019.

[194] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[195] Peter Likarish, Eunjin Jung, and Insoon Jo. Obfuscated malicious javascript detection using classification techniques. In *MALWARE*, pages 47–54. Citeseer, 2009.

[196] Xiang Ling, Lingfei Wu, Chunming Wu, and Shouling Ji. Graph neural networks: Graph matching. *Graph Neural Networks: Foundations, Frontiers, and Applications*, pages 277–295, 2022.

[197] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, 2003.

[198] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αdiff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 667–678. ACM, 2018.

[199] Daiping Liu, Haining Wang, and Angelos Stavrou. Detecting malicious javascript in pdf through document instrumentation. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, 2014.

[200] Eliane Maria Loiola, Nair Maria Maia de Abreu, Paulo Oswaldo Boaventura-Netto, Peter Hahn, and Tania Querido. A survey for the quadratic assignment problem. *European Journal of Operational Research*, 176(2):657–690, 2007.

[201] Gen Lu and Saumya Debray. Automatic simplification of obfuscated javascript code: A semantics-based approach. In *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability*, 2012.

[202] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P Chung, Taesoo Kim, and Wenke Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 280–291. ACM, 2015.

[203] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 389–400, 2014.

[204] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Application Conference (ACSAC'11)*, December 2011.

[205] Giorgi Maisuradze, Michael Backes, and Christian Rossow. Dachshund: Digging for and securing against (non-) blinded constants in jit code. 2017.

[206] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2099–2116, 2022.

[207] Jeferson Martínez and Javier M Durán. Software supply chain attacks, a threat to global cybersecurity: Solarwinds' case study. *International Journal of Safety and Security Engineering*, 11(5):537–545, 2021.

[208] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329. Springer, 2019.

[209] Leo A Meyerovich and Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 481–496. IEEE, 2010.

[210] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1187–1198, Piscataway, NJ, USA, 2019. IEEE Press.

[211] Jiang Ming, Meng Pan, and Debin Gao. ibinhunt: Binary hunting with inter-procedural control flow. In *International Conference on Information Security and Cryptology*, pages 92–109. Springer, 2012.

[212] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics*, 5(1):32–38, 1957.

[213] Kevin Murphy, Yair Weiss, and Michael I Jordan. Loopy belief propagation for approximate inference: An empirical study. *arXiv preprint arXiv:1301.6725*, 2013.

[214] Shripad Nadgowda. Engram: the one security platform for modern software supply chain risks. In *Proceedings of the Eighth International Workshop on Container Technologies and Container Clouds*, pages 7–12, 2022.

[215] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 12–pp. IEEE, 2006.

[216] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, pages 23–43. Springer, 2020.

[217] Marc Ohm and Charlene Stuke. Sok: Practical detection of software supply chain attacks. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, pages 1–11, 2023.

[218] Marc Ohm, Arnold Sykosch, and Michael Meier. Towards detection of software supply chain attacks by forensic artifacts. In *Proceedings of the 15th international conference on availability, reliability and security*, pages 1–6, 2020.

[219] Xiaorui Pan, Xueqiang Wang, Yue Duan, Xiaofeng Wang, and Heng Yin. Dark hazard: Large-scale discovery of unknown hidden sensitive operations in android apps. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS'17)*, February 2017.

[220] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[221] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8026–8037, 2019.

[222] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 81–92. IEEE, 2004.

[223] Judea Pearl. *Reverend Bayes on inference engines: A distributed hierarchical approach*. Cognitive Systems Laboratory, School of Engineering and Applied Science . . . , 1982.

[224] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

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

[226] Kexin Pei, Jonas Guan, David Williams King, Junfeng Yang, and Suman Jana. Xda: Accurate, robust disassembly with transfer learning. In *NDSS*, 2021.

[227] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*, 2020.

[228] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724. IEEE, 2015.

[229] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 406–415, 2014.

[230] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. {ADsafety}:{Type-Based} verification of {JavaScript} sandboxing. In *20th USENIX Security Symposium (USENIX Security 11)*, 2011.

[231] Aravind Prakash, Xunchao Hu, and Heng Yin. vfguard: Strict protection for virtual function calls in COTS binaries. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, February 2015.

[232] Aravind Prakash, Eknath Venkataramani, Heng Yin, and Zhiqiang Lin. Manipulating semantic values in kernel data structures: Attack assessments and implications. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2013.

[233] Aravind Prakash, Eknath Venkataramani, Heng Yin, and Zhiqiang Lin. On the trustworthiness of memory analysis—an empirical study from the perspective of binary execution. *IEEE Transactions on Dependable and Secure Computing*, 12(5), 2015.

[234] Aravind Prakash and Heng Yin. Defeating ROP through denial of stack pivot. In *2015 Annual Computer Security Applications Conference*, December 2015.

[235] Aravind Prakash, Heng Yin, and Zhenkai Liang. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 311–322. ACM, 2013.

[236] Aravind Prakash, Heng Yin, and Zhenkai Liang. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communication Security*, May 2013.

[237] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224, 2003.

[238] Rui Qiao and R Sekar. Effective function recovery for cots binaries using interface verification. Technical report, Technical report, Secure Systems Lab, Stony Brook University, 2016.

[239] Rui Qiao and R Sekar. Function interface analysis: A principled approach for function recognition in cots binaries. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 201–212. IEEE, 2017.

[240] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[241] Paruj Ratanaworabhan, V Benjamin Livshits, and Benjamin G Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium*, pages 169–186, 2009.

[242] Charles Reis, John Dunagan, Helen J Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Transactions on the Web (TWEB)*, 1(3):11, 2007.

[243] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010.

[244] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. *CoRR*, abs/1802.10135, 2018.

[245] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[246] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010.

[247] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[248] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pages 593–607. Springer, 2018.

[249] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[250] Christian Seifert, Ian Welch, and Peter Komisarczuk. Identification of malicious web pages with static heuristics. In *Telecommunication Networks and Applications Conference, 2008. ATNAC 2008. Australasian*, pages 91–96. IEEE, 2008.

[251] Alexander Sepp, Bogdan Mihaila, and Axel Simon. Precise static analysis of binaries by extracting relational information. In *2011 18th Working Conference on Reverse Engineering*, pages 357–366. IEEE, 2011.

[252] Fermin J Serna. The info leak era on software exploitation. *Black Hat USA*, 2012.

[253] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. Eureka: A framework for enabling static malware analysis. In *European Symposium on Research in Computer Security*, pages 481–500. Springer, 2008.

[254] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 611–626, 2015.

[255] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[256] Richard Sinkhorn. A Relationship Between Arbitrary Positive Matrices and Doubly Stochastic Matrices. *The Annals of Mathematical Statistics*, 35(2):876 – 879, 1964.

[257] Alexey Sintsov. Writing jit-spray shellcode for fun and profit. *Writing*, 2010.

[258] Kihyuk Sohn. Improved deep metric learning with multi-class n-pair loss objective. *Advances in neural information processing systems*, 29, 2016.

[259] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, Hyderabad, India, December 2008.

[260] Alexander Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007.

[261] Ankur Taly, Úlfar Erlingsson, John C Mitchell, Mark S Miller, and Jasvir Nagra. Automated analysis of security-critical javascript apis. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 363–378. IEEE, 2011.

[262] Wei Tang, Ping Luo, Jialiang Fu, and Dan Zhang. Libdx: A cross-platform and accurate system to detect third-party libraries in binary code. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 104–115. IEEE, 2020.

[263] Xun Tang, Michael Shavlovsky, Holakou Rahmanian, Elisa Tardini, Kiran Koshy Thekumparampil, Tesi Xiao, and Lexing Ying. Accelerating sinkhorn algorithm with sparse newton iterations. *arXiv preprint arXiv:2401.12253*, 2024.

[264] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*, pages 941–955, 2014.

[265] Tung Tran, Riccardo Pelizzi, and R Sekar. Jate: Transparent and efficient javascript confinement. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 151–160. ACM, 2015.

[266] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[267] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[268] Thomas Viehmann. Lernapparat - Machine Learning — lernapparat.de. `https://lernapparat.de/sinkhorn-kernel`. [Accessed 25-04-2024].

[269] Thomas Viehmann. Implementation of batched sinkhorn iterations for entropy-regularized wasserstein loss, 2019.

[270] Hao Wang, Zeyu Gao, Chao Zhang, Mingyang Sun, Yuchen Zhou, Han Qiu, and Xi Xiao. Cebin: A cost-effective framework for large-scale binary code similarity detection. *arXiv preprint arXiv:2402.18818*, 2024.

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

[272] Minghua Wang, Heng Yin, Abhishek Vasisht Bhaskar, Purui Su, and Dengguo Feng. Binary code continent: Finer-grained control flow integrity for stripped binaries. In *2015 Annual Computer Security Applications Conference*, December 2015.

[273] Runzhong Wang, Ziao Guo, Wenzheng Pan, Jiale Ma, Yikai Zhang, Nan Yang, Qi Liu, Longxuan Wei, Hanxue Zhang, Chang Liu, Zetian Jiang, Xiaokang Yang, and Junchi Yan. Pygmtools: A python graph matching toolkit. *Journal of Machine Learning Research*, 25(33):1–7, 2024.

[274] Runzhong Wang, Junchi Yan, and Xiaokang Yang. Combinatorial learning of robust deep graph matching: an embedding based approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[275] Runzhong Wang, Junchi Yan, and Xiaokang Yang. Neural graph matching network: Learning lawler's quadratic assignment problem with extension to hypergraph and multiple-graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):5261–5279, 2022.

[276] Runzhong Wang, Tianqi Zhang, Tianshu Yu, Junchi Yan, and Xiaokang Yang. Combinatorial learning of graph edit distance via dynamic embedding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5241–5250, 2021.

[277] Richard Wartell, Yan Zhou, Kevin W Hamlen, and Murat Kantarcioglu. Shingled graph disassembly: Finding the undecideable path. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 273–285. Springer, 2014.

[278] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 522–536. Springer, 2011.

[279] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. Centris: A precise and scalable approach for identifying modified open-source software reuse. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 860–872. IEEE, 2021.

[280] Mengjun Xie, Heng Yin, and Haining Wang. An effective defense against email spam laundering. In *Proceedings of the 13th ACM Conference on Computer and Communication Security (CCS'06)*, October 2006.

[281] Mengjun Xie, Heng Yin, and Haining Wang. Thwarting email spam laundering. *ACM Transactions on Information and System Security (TISSEC)*, December 2008.

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

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

[284] Zhaoyan Xu, Jialong Zhang, Guofei Gu, and Zhiqiang Lin. AUTOVAC: Towards automatically extracting system resource constraints and generating vaccines for malware immunization. In *Proc. of the 33rd International Conference on Distributed Computing Systems (ICDCS'13)*, July 2013.

[285] Lok Kwong Yan, Andrew Henderson, Xunchao Hu, Heng Yin, and Stephen McCamant. On soundness and precision of dynamic taint analysis. Technical Report SYR-EECS-2014-04, Syracuse University, January 2014.

[286] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. Transparent and extensible malware analysis by combining hardware virtualization and software emulation. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12), Invited Paper*, February 2012.

[287] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2E: Combining hardware virtualization and software emulation for transparent and extensible malware analysis. In *Proceedings of the Eighth Annual International Conference on Virtual Execution Environments (VEE'12)*, March 2012.

[288] Lok-Kwong Yan and Heng Yin. DroidScope: Seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Security Symposium*, August 2012.

[289] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[290] Heng Yin. *Malware Detection and Analysis via Layered Annotative Execution.* PhD dissertation, College of William and Mary, Department of Computer Science, July 2009.

[291] Heng Yin, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[292] Heng Yin, Pongsin Poosankam, Steve Hanna, and Dawn Song. HookScout: Proactive binary-centric hook detection. In *Proceedings of Seventh Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'10)*, July 2010.

[293] Heng Yin, Bo Sheng, Haining Wang, and Jianping Pan. Securing BGP through keychain-based signatures. In *Proceedings of the 15th IEEE International Workshop on Quality of Service (IWQoS'07)*, June 2007.

[294] Heng Yin, Bo Sheng, Haining Wang, and Jianping Pan. Keychain-based signatures for securing bgp. *IEEE Journal on Selected Areas in Communications (J-SAC), Internet Routing Scalability*, October 2010.

[295] Heng Yin and Dawn Song. Temu: Binary code analysis via whole-system layered annotative execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley, Jan 2010.

[296] Heng Yin and Dawn Song. Temu: Binary code analysis via whole-system layered annotative execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley, Jan 2010.

[297] Heng Yin and Dawn Song. *Automatic Malware Analysis: An Emulator based Approach.* Springer Briefs in Computer Science, September 2012.

[298] Heng Yin, Dawn Song, Egele Manuel, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[299] Heng Yin and Haining Wang. Building an application-aware IPsec policy system. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[300] Heng Yin and Haining Wang. Building an application-aware ipsec policy system. *IEEE/ACM Transactions on Networking*, December 2007.

[301] Tianshu Yu, Runzhong Wang, Junchi Yan, and Baoxin Li. Learning deep graph matching with channel-independent embedding and hungarian attention. In *International Conference on Learning Representations*, 2020.

[302] Yang Yu. Write once, pwn anywhere. *BlackHat*, 2014.

[303] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 1145–1152, 2020.

[304] Zimu Yuan, Muyue Feng, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, et al. B2sfinder: Detecting open-source software reuse in cots software. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1038–1049. IEEE, 2019.

[305] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1695–1707. IEEE, 2021.

[306] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

[307] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 887–902, 2018.

[308] Jiexin Zhang, Alastair R Beresford, and Stephan A Kollmann. Libid: reliable identification of obfuscated third-party android libraries. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 55–65, 2019.

[309] Mingwei Zhang, Aravind Prakash, Xiaolei Li, Zhenkai Liang, and Heng Yin. Identifying and analysing pointer misuses for sophisticated memory-corruption exploit diagnosis. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, February 2012.

[310] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. Towards automatic generation of security-centric descriptions for android apps. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*, October 2015.

[311] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual API dependency graphs. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS'14)*, November 2014.

[312] Mu Zhang and Heng Yin. AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, February 2014.

[313] Mu Zhang and Heng Yin. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communication Security*, June 2014.

[314] Naville Zhang. Hikari – an improvement over obfuscator-llvm. `https://github.com/HikariObfuscator/Hikari`.

[315] Yunan Zhang, Chenghao Rong, Qingjia Huang, Yang Wu, Zeming Yang, and Jianguo Jiang. Based on multi-features and clustering ensemble method for automatic malware categorization. In *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 73–82. IEEE, 2017.

[316] Feng Zhou and Fernando De la Torre. Factorized graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(9):1774–1789, 2016.

[317] Wenjie Zhuo, Yifan Sun, Xiaohan Wang, Linchao Zhu, and Yi Yang. Whitenedcse: Whitening-based contrastive learning of sentence embeddings. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12135–12148, 2023.

[318] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706*, 2018.

[319] Fei Zuo, Xiaopeng Li, Zhexin Zhang, Patrick Young, Lannan Luo, and Qiang Zeng. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *NDSS*, 2019.