Data Mining and Machine Learning for Reverse Engineering

Steven H. H. Ding

School of Information Studies
McGill University
Montreal, Quebec, Canada

Feburary 2019

A thesis submitted to McGill University in partial
fulfillment of the requirements of the degree of
Doctor of Philosophy

Abstract

Reverse engineering is fundamental for understanding the inner workings of new malware, exploring new vulnerabilities in existing systems, and identifying patent infringements in the distributed executables. It is the process of getting an in-depth understanding of a given binary executable without its corresponding source code. Reverse engineering is a manually intensive and time-consuming process that relies on a thorough understanding of the full development stack from hardware to applications. It requires a much steeper learning curve than programming. Given the unprecedentedly vast amount of data to be analyzed and the significance of reverse engineering, the overall question that drives the studies in this thesis is how can data mining and machine learning technologies make cybersecurity practitioners more productive to uncover the provenance, understand the intention, and discover the issues behind the data in a scalable way. In this thesis, I focus on two data-driven solutions to help reverse engineers analyzing binary data: assembly clone search and behavioral summarization.

Assembly code clone search is emerging as an Information Retrieval (IR) technique that helps address security problems. It has been used for differing binaries to locate the changed parts, identifying known library functions such as encryption, searching for known programming bugs or zero-day vulnerabilities in existing software or Internet of Things (IoT) devices firmware, as well as detecting software plagiarism or GNU license infringements when the source code is unavailable. However, designing an effective search engine is difficult, due to varieties of compiler optimization and obfuscation techniques that make logically similar assembly functions appear to be dramatically different. By working closely with reverse engineers, I identify three different sce-

narios of reverse engineering and develop novel data mining and machine learning models for assembly clone search to address the respective challenges. By developing an intelligent assembly clone search platform, I optimize the process of reverse engineering by addressing the information needs of reverse engineers. Experimental results suggest that Kam1n0 is accurate, efficient, and scalable for handling a large volume of data.

The second part of the thesis goes beyond optimizing an information retrieval process for reverse engineering. I propose to automatically and statically characterize the behaviors of a given binary executable. Behavioral indicators denote those potentially high-risk malicious behaviors exhibited by malware, such as unintended network communications, file encryption, keystroke logging, abnormal registry modifications, sandbox evasion, and camera manipulation. I design a novel neural network architecture that models the different aspects of an executable. It is able to predict over 139 suspicious and malicious behavioral indicators, without running the executable. The resulting system can be used as an additional binary analytic layer to mitigate the issues of polymorphism, metamorphism, and evasive techniques. It also provides another behavioral abstraction of malware to security analysts and reverse engineers. Therefore, it can reduce the data to be manually analyzed, and the reverse engineers can focus on the binaries that are of their interest.

In summary, this thesis presents four original research projects that not only advance the knowledge in reverse engineering and data mining, but also contribute to the overall safety of our cyber world by providing open-source award-winning binary analysis systems that empower cybersecurity practitioners.

Résumé

Lingénierie inverse est essentielle pour comprendre le fonctionnement interne des nouveaux logiciels malveillants, explorer des nouvelles vulnérabilités dans les systèmes existants et identifier les violations de brevets dans les exécutables distribués. C'est le processus d'obtenir une compréhension approfondie d'un exécutable binaire donné sans le code source. L'ingénierie inverse est un processus intensif manuellement et long qui compte sur une compréhension approfondie de la pile de développement complète. Cela nécessite une courbe d'apprentissage beaucoup plus raide que la programmation. Compte tenu de la quantité sans précédent de données à analyser, la question générale promeut les travaux de cette thèse est de savoir comment les technologies d'exploration de données et dapprentissage machine peuvent rendre les praticiens de la cybersécurité plus productifs pour découvrir la provenance et découvrez les problèmes liés aux données. Dans cette thèse, je me concentre sur deux solutions basées sur les données pour aider les ingénieurs inverse à analyser des données binaires: la recherche de clone d'assemblage et la synthèse comportementale.

La recherche de clones de code d'assemblage est émergent comme une technique de récupération d'informations (IR) qui aide à résoudre les problèmes de sécurité. Il a été utilisé par différents binaires pour localiser les pièces modifiées, identifiant des fonctions de bibliothèque connues telles que le cryptage, la recherche de bogues de programmation connus ou de vulnérabilités zero-day dans les logiciels existants ou les appareils de firmware d'internet des objets (IoT), ainsi que la détection de plagiat logiciel ou dinfractions de licence GNU lorsque le code source est indisponible. Cependant, concevoir un moteur de recherche efficace est difficile, en raison de la diversité dobscurcissement du compilateur et les techniques qui rendent les fonctions dassemblage logiquement

similaires semblent être radicalement différentes. En travaillant en étroite collaboration avec les ingénieurs inverse, j'identifie trois scénarios différents d'ingénierie inverse et développé de nouveaux modèles d'exploration de données et dapprentissage machine pour la recherche de clones d'assemblage afin de relever les défis respectifs. En développant une plate-forme de recherche de clones d'assemblage intelligente, j'optimise le processus dingénierie inverse en s'adressant aux besoins des informations pour lingénieurs inverse.

La deuxième partie de la thèse va au-delà de loptimisation dun processus de récupération dinformation pour lingénierie inverse. Je propose de caractériser automatiquement et statiquement les comportements dun exécutable binaire donné. Les indicateurs comportementaux indiquent les comportements malveillants potentiellement à risque élevé par des logiciels malveillants, tels que les communications réseau non intentionnelles, le cryptage des fichiers, lenregistrement des frappes clavier, les modifications de registre anormales, lenvironnement de test (Sandbox), et la manipulation de la caméra. Je conçois une nouvelle architecture de réseau neuronal qui modélise les différents aspects dun exécutable. Il est capable de prédire plus de 139 indicateurs de comportement suspects et malveillants, sans exécuter l'exécutable. Le système résultant peut être utilisé comme une couche analytique binaire supplémentaire pour atténuer les problèmes de polymorphisme, de métamorphisme et des techniques évasives. Il fournit également une autre abstraction comportementale des logiciels malveillants aux analystes de la sécurité.

En résumé, cette thèse présente quatre projets de recherche originaux qui non seulement avance les connaissances dingénierie inverse et d'exploration de données, mais aussi contribuent à la sécurité globale de notre monde cybernétique en fournissant des systèmes danalyse binaire primés à la source ouverte qui responsabilisent les praticiens de la cybersécurité.

Contributions

This section summarizes the main contributions of four research papers that primarily constitute this thesis. The complete list of contributions can be found in each corresponding section. These four papers are my own work. However, all of them benefited tremendously from the guidance, feedback, advice, review, and proofreading from my supervisor Dr. Benjamin C. M. Fung, our research collaborator Mr. Philippe Charland, and my supervision committees: Dr. Charles-Antoine Julien and Dr. Shane McIntosh.

Kam1n0 [30]: It is the first clone search engine that can efficiently and accurately identify the given query assembly function's subgraph clones from a repository of millions of candidates. It contributes to the data mining domain by proposing a new adaptive locality sensitive hashing algorithm and a new map-reduce based subgraph clone search algorithm. Extensive experimental results suggest that Kam1n0 is accurate, efficient, and scalable for handling a large volume of assembly code.

Sym1n0 [28]: Kam1n0 is designed for subgraph clone search within a single processor architecture such as x86 or ARM. However, in some reverse engineering scenarios, such as firmware analysis, one will need to find clones among assembly codes of different processor families. This paper proposes the first scalable subgraph clone search engine that supports cross-architecture search. I design a novel tree-based index for symbolic expressions using their differential I/O behavior and combine it with a MapReduce-based subgraph search algorithm. The index can find semantically similar expressions with sub-linear complexity. This work is currently under review in an international journal.

Asm2Vec [29]: There exist various compiler optimization and code obfuscation techniques that make logically similar assembly functions appear to be very different. To address this problem, I propose a novel neural network to jointly learn the lexical semantic relationships and the vector representation of assembly functions based on assembly code. It can find and incorporate rich semantic relationships among tokens. The experiment shows that the learned representation is more robust and significantly outperforms existing methods against the changes introduced by obfuscation and optimizations.

RAVEN [27]: Behavioral indicators provide a high-level understanding of the malware's dynamic functionalities for security analysts and are used to characterize previously unknown malware families. I propose and implement the first new neural network-based static scanner, RAVEN, that can characterize the malicious behaviors of a given executable. The goal is to statistically predict 139 suspicious and malicious behavioral indicators, without running the executable. We design a novel neural network architecture that models the different aspects of an executable. Relying on pattern recognition, it tries to find any generalizable information and does not require unpacking. The experimental result shows that it performs better than existing static approaches with a low false positives rate. This work is currently under review in an international conference.

The resulting systems are not only published in top data mining and security forums but also won an international award and are being used in public sectors and private sectors.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr. Benjamin C. M. Fung, for the tremendous amount of time and support he has given me toward the completion of this doctoral research. I have been supervised by Dr. Fung since I was a graduate student in late 2012. I truly appreciate his guidance, advice, and encouragement in the past seven years. I am also very grateful for the fact that he has always been available whenever I had any difficulty. Without all of these, I could not have made it to today. His passion for data mining research, dedication to high-quality publications, and powerful problem-solving and management skills have had a profound impact on me. His training makes me stand out as I am today in academia. I am excited and looking forward to the next stage of my career, yet undeniably sad to leave behind this fulfilling experience.

I would also like to thank our research collaborator, Mr. Philippe Charland, not only for his support and feedback on my doctoral study, but also for his active support of my future career. Together with Dr. Fung, I am very glad that we have been able to contribute to the good of cyber-security both nationally and internationally.

I want to express my sincere appreciation to my supervisory committee, Dr. Charles-Antoine Julien and Dr. Shane McIntosh, for their valuable feedback and constructive comments to improve my research. I want to thank Dr. France Bouthillier for her help getting the funding that has supported my research in the past four years. I want to thank Dr. Kimiz Dalkir for her advice and support during the transition of my academic career, as well as those teaching opportunities that have strengthened my professional standing.

Without the great support and help from both Ms. Cathy Venetico and Ms. Kathryn Hubbard at the School of Information Studies, my doctoral journey would not have run as smoothly as it did. Also, I would like to thank FRQNT as well as the grant reviewers for the research funding that supports my research.

Last but not least, I would like to express my boundless appreciation from the bottom of my heart to my family, especially my parents, for their invaluable love and unconditional support throughout the educational journey. I want to thank my grandfather Mr. Dejian Ding, who is not with us any more, for his love, understanding, and support in the past. Special thanks go to my mother-in-law, Ms. Jingxian Chang, for helping out when everything seemed chaotic and out of control. An appreciation that is full of love goes to my wife, Lynne Guo, for the love and joy that she brings and her constant support, encouragement, and understanding of my often extra-long working hours all along this journey. Our daughter, Anna, has brought extra happiness in the last 13 months; I am so lucky to have you two.

Contents

1 Introduction **1**

1.1 Assembly Clone Search . . . 2

1.2 Assembly Code Behavioral Analysis . . . 3

1.3 Research Outcome and Thesis Organization . . . 4

2 Assembly Subgraph Clone Search **6**

2.1 Related Work . . . 10

2.2 Problem Statement . . . 11

2.3 Overall Architecture . . . 13

2.4 Preprocessing and Vector Space . . . 14

2.5 Locality Sensitive Hashing . . . 14

2.5.1 Adaptive LSH Structure . . . 17

2.6 Subgraph Clone Search . . . 21

2.6.1 MapReduce Subgraph Search . . . 22

2.7 Experiments . . . 23

2.7.1 Labeled Dataset Generation . . . 24

2.7.2 Normalization Level . . . 27

2.7.3 Clone Search Approach Benchmark . . . 28

2.7.4 Scalability Study . . . 29

2.8 Conclusion and Lesson Learned . . . 30

3 Cross-Architecture Subgraph Clone Search **33**

3.1 Related Work . . . 38

3.2 Problem Statement . . . 39

3.3 Overall Architecture . . . 41
3.4 Constructing the Syntax Tree . . . 43
3.5 S-Expression Retrieval . . . 44
3.5.1 A Scalable Tree-based Index . . . 46
3.5.2 Input Range Sampling . . . 50
3.5.3 Implementation Details and Complexity Analysis . . . 53
3.6 Subgraph Clone Search . . . 55
3.7 Experiments . . . 59
3.7.1 Assembly Clone Search on Utility Libraries . . . 59
3.7.2 Assembly Clone Search on Numeric Calculation Libraries . . . 62
3.7.3 Scalability Study . . . 65
3.8 Limitations . . . 66
3.9 Conclusion . . . 67

4 Robust Clone Search **68**
4.1 Problem Definition . . . 72
4.2 Overall Workflow . . . 73
4.3 Assembly Code Representation Learning . . . 74
4.3.1 Preliminaries . . . 74
4.3.2 The Asm2Vec Model . . . 77
4.3.3 Modeling Assembly Functions . . . 81
4.3.4 Training, Estimating, and Searching . . . 84
4.4 Experiments . . . 85
4.4.1 Searching with Different Compiler Optimization Levels . . . 85
4.4.2 Searching with Code Obfuscation . . . 90
4.4.3 Searching against All Binaries . . . 95
4.4.4 Searching Vulnerability Functions . . . 97
4.5 Related Work . . . 98
4.6 Limitations and Conclusion . . . 101

5 Going beyond Information Retrieval: Behavior Characterization **103**

5.1 Problem Definition . . . 107
5.2 Neural Network Architecture . . . 108
5.2.1 Modeling the Code Segment . . . 108
5.2.2 Modeling Strings and Import Symbols . . . 115
5.2.3 Modeling Data Segments . . . 117
5.2.4 Pre-selection and Shared Representation . . . 118
5.2.5 Behavior-Conditioned Attention Mechanism . . . 120
5.3 Experiments . . . 121
5.3.1 Dataset Development . . . 122
5.3.2 Quantitative Benchmark . . . 128
5.4 Case Studies . . . 131
5.5 Related Works . . . 132
5.6 Limitations and Conclusion . . . 133

6 Final Conclusion & Future Work **136**
6.1 Final Conclusion . . . 136
6.2 Future Directions . . . 139

List of Publications **141**

Bibliography **143**

Appendices **157**

A Kam1n0 Implementation Details **158**
A.1 Assembly code normalization . . . 158
A.2 Adaptive Locality Sensitive Hashing . . . 158
A.2.1 Theoretical Analysis . . . 159
A.2.2 Implementation on the Key-value Database . . . 160

B Extended Formulation of Asm2Vec **165**

C Extended Descriptive Statistics of the Dataset **170**

D Cuckoo Behavioral Indicators **174**

E Falcon Behavioral Indicators **183**

List of Figures

2.1 An example of the clone search problem. . . . 8
2.2 The overall solution stack of the Kam1n0 engine. . . . 12
2.3 Assembly clone search data flow. . . . 13
2.4 The Adaptive Locality Sensitive Hashing structure for Kam1n0. . . . 18
2.5 The MapReduce-based subgraph clone construction process. . . . 20
2.6 The Empirical Distribution Functions on dataset statistics. . . . 31
2.7 Scalability study of Kam1n0. . . . 32

3.1 An example of cross-architecture assembly clone. . . . 34
3.2 The overall solution stack of the Sym1n0 search engine. . . . 40
3.3 The four stages of assembly clone search data flow. . . . 42
3.4 An example of locating the buckets for a symbolic expression. . . . 49
3.5 Symbolic index tree implemented with a column family model. . . . 54
3.6 The MapReduce-based subgraph clone construction process for Sym1n0. . . . 56
3.7 Dataset statistics of the cross-architecture utility dataset. . . . 57
3.8 Scalability study for Sym1n0. . . . 64

4.1 Example of code obfuscation and optimization. . . . 69
4.2 *T-SNE* visualization of tokens in assembly code. . . . 70
4.3 The overall work flow of *Asm2Vec*. . . . 73
4.4 The PV-DM model. . . . 75
4.5 The proposed *Asm2Vec* neural network model for assembly code. . . . 76
4.6 Statistical difference between optimization options. . . . 86
4.7 Statistical difference between obfuscation techniques. . . . 91

4.8 Benchmark experiment with all binaries. 94
4.9 Vulnerability retrieval with Asm2Vec - an Example. 99

5.1 Malware behavioral indicator detection process. 104
5.2 The neural network architecture in RAVEN. 109
5.3 The customized Asm2Vec model. 111
5.4 Basic block binning process. 114
5.5 Statistics of malware characteristics. 121
5.6 The loss values across RAVEN's training epochs. 130
5.7 A case study using three variants of the *GandCrab* malware. 135

A.1 The hierarchy used to normalize the operands. 163
A.2 The data module for ALSH. 164

C.1 O-LLVM instruction substitution example. 172
C.2 Statistics for different optimization techniques. 173
C.3 Statistics for different obfuscation techniques. 173

List of Tables

2.1 The assembly code clone dataset summary. 25

2.2 Benchmark of different assembly clone search approaches with Kam1n0. 26

2.3 Paired t-test on the normalization level. 27

3.1 Examples of retrieved symbolic expression. 36

3.2 Comparing Sym1n0 with the related systems. 36

3.3 Cross-architecture clone search benchmark on utility dataset. 58

3.4 Cross-architecture clone search benchmark on calculation dataset. 60

3.5 Number of functions for binaries compiled for different architectures. 62

3.6 Efficiency of Sym1n0 for clone search. 63

4.1 Clone search benchmark with compiler optimization. 87

4.2 Clone search benchmark with obfuscation techniques. 92

4.3 Vulnerability retrieval with Asm2Vec. 96

4.4 Vulnerability search with Tigress-obfuscated binaries. 102

5.1 Top-40 malware families in the Cuckoo dataset. 122

5.2 Statistics of the grouped evasive behavioral indicators observed in the Cuckoo dataset. See Appendix D for the full description of each indicator. 124

5.3 Top-20 groups of dynamic malicious behavior indicator in the Cuckoo dataset. . . . 124

5.4 Top-40 malware families in the Falcon dataset. 126

5.5 Statistics of the grouped dynamic malicious behavior indicator in the Falcon dataset. 127

5.6 Behavior recognition benchmark. 128

B.1 Parameters to be estimated in training. 165

B.2 Intermediate symbols used in training. 166

B.3 Gradients to be calculated in a training step. 168

C.1 Difference in the number of basic blocks for different optimization techniques. . . 171

C.2 Difference in the number of basic blocks for different obfuscation techniques. . . . 171

D.1 Description of each behavioral descriptor. 174

E.1 Description of each behavioral descriptor. 183

1

Introduction

Reverse engineering is the process of getting an in-depth understanding of a given binary executable without its corresponding source code. First, the binary executable is translated byte-by-byte into a human-understandable format of machine instructions, namely, assembly code. Then the reverse engineer abstracts the assembly code into a series of logical flow to determine its functionality. This analytic process is called assembly code analysis. Reverse engineering is critical for mitigating the increasing threats from malicious software. Even without the corresponding source code, one can still get a thorough understanding of their inner workings. Reverse engineering is also a common practice for detecting and justifying the plagiarism and the patent infringements of software.

In 2013, on average 82,000 new strains of malware were generated per day. In 2014, this number rose up to 230,000 per day, according to the Panda Security Annual reports [99], [100]. Malware refers to any software that exhibits malicious behaviors, such as viruses, Trojan horses, worms, and ransomware. Most malware was not created from scratch. It was developed or mutated from previous software. Code reuse is a common but uncontrolled issue in software engineering [63]. More than 50% of files were reused in more than one open source project [91]. The survey [124] indicates that more than 50% of the developers modified the components they reused. Such massively uncontrolled reuse of source code does not only introduce legal issues such as GNU violations [68], [137], but also implies security concerns [19], as the source code and the vulnerabilities were shared between projects. Moreover, malware becomes increasingly intelligent,

obfuscated, and sophisticatedly encrypted. It is challenging for a reverse engineer to analyze.

Addressing all these issues requires effort from reverse engineers, which is a manually intensive and time-consuming process, even for an experienced reverse engineer. Moreover, the learning curve to master reverse engineering is much steeper than for programming [19]. There is a considerably increased number of executables to be analyzed by reverse engineers. However, there are limited scalable techniques and tools that can reduce the burden of this task. Given that code reuse is prevalent in software development, there is a pressing need to develop new data mining and machine learning techniques that can leverage the knowledge of those existing software or malware that has been analyzed before. Given the executable under study, these techniques can help reverse engineers avoid components that have been analyzed before and focus only on the new part. Intelligent models can be further trained to profile the behaviors and analyze the provenance of the given executable.

The overall question that drives the research agenda of this thesis is *how technologies, specifically data mining and machine learning, can help reverse engineers analyze and uncover the provenance, understand the intention, and discover issues in a scalable way, given the exponentially increasing amount of data and threats*.

In this study, it is assumed that reverse engineers have a large repository of binary files from previous analyses. This repository can contain assembly code that has been previously annotated or assembly code from some well-known libraries and open-source projects. Alternatively, it can contain a database of well-understood malware samples. The objective of this research is to develop new data mining techniques and machine learning models that can turn the overwhelming amount of data into reusable knowledge and facilitate the reverse engineering process. The proposed study addresses four different real-life challenges from two different aspects:

1.1 Assembly Clone Search

Assembly code clone search is emerging as an Information Retrieval (IR) technique that helps address security-related problems. It has been used for differing binaries in locating the changed parts [14], identifying known library functions such as encryption [105], searching for known programming bugs or zero-day vulnerabilities in existing software or Internet of Things (IoT)

devices firmware [34], [39], as well as detecting software plagiarism or GNU license infringements when the source code is unavailable [69], [85]. According to the needs of the reverse engineering task in different analytic scenarios, the proposed research focuses on three retrieval problems:

- **Subgraph Clone Search** Focuses on scalable and efficient clone search that supports subgraph retrieval. Supports clone search within a single assembly instruction family such as *x86*. It is applicable for retrieving similar vulnerabilities across the software repository or comparing different versions or variants of a binary. For example, it enables the comparison between different browsers such as *Opera*, *Chrome*, and *Chromium* that are built on the same rendering engine *Blink*.
- **Cross Architecture Clone Search** Enables subgraph clone search across different assembly instruction families. For example, searching the *x86* code as a query against the repository of *arm* code. It enables large-scale retrieval of variants of vulnerability across different firmware and drivers.
- **Robust Clone Search** The same source code can result in very different assembly code given different compilers and optimization techniques. Moreover, nowadays malware is mostly obfuscated, and can look different, even when they belong to the same family [64], [139]. A robust clone search engine that neutralizes the semantic variants can help reverse engineering retrieve the highly obfuscated clones.

1.2 Assembly Code Behavioral Analysis

In addition to helping reverse engineers understand low-level semantic details of a binary file through clone search, the proposed study also automates high-level behavioral analyses for reverse engineering purposes. The goal is to statically predict an executable's potential malicious behaviors without actually running or simulating it. Simulation is also time-consuming and computationally-expensive. Malicious behaviors denote those user-unaware or damaging behaviors that are practiced by an executable, such as unintended network communications, file encryption, keystroke logging, abnormal registry modifications, sandbox evasion, and camera manipulation. Generally, they are referred to as behavioral indicators.

Behavioral indicators provide a high-level understanding of a malware's dynamic functionalities for security analysts and are used to characterize previously unknown malware families. Behavioral indicators have been widely used as the output of sandbox-based malware analyses. However, the complexity of modern malware has considerably increased. Malware is becoming sandbox-aware by incorporating modern evasive techniques. They can detect the sandbox environment and then skip unpacking, decrypting, or executing the critical malicious payload.

To address these issues, we propose a new neural network-based static scanner, RAVEN, that can scan a binary file and characterize the malicious behaviors of a given executable. The model should be able to neutralize the effect of different packers and obfuscators in modern malware to fulfill the prediction tasks. Modern malware can skip executing malicious payload if a simulation run-time environment is detected. Simulation-based behavior analysis techniques fail in such a situation. However, a neural network as a static scanner does not suffer from this problem. It completes modern sandbox-based simulation analysis.

1.3 Research Outcome and Thesis Organization

The outcome of the proposed research mainly consists of the following components:

- Academic publications in the area of data mining, software engineering, and software security that fill existing specific research gaps in state-of-the-art static binary or assembly analysis techniques.
- Publicly available new datasets for future research under similar directions, to facilitate the reproducibility and transferability of the research outcome.
- An open-source unified binary management and analysis platform that incorporates all the proposed techniques. It can be seamlessly integrated into the day-to-day workflow of reverse engineers.

The resulting system has been presented at SOPHOS Vancouver, ESET Montreal, Above Security Montreal, and Google Montreal. It won the best poster award at the Smart Cybersecurity Network Canada (SERENE-RISC) workshop. It also won the Hex-Rays international plug-in contest award. Hex-Rays develops the most widely used and most capable disassembler for reverse

engineering. It is now used in Defence Research and Development Canada (DRDC) and has been integrated into the malware signature generation process in Cisco. See chapter 6.2 for more details.

This thesis is organized as follows. Chapter 2 presents our subgraph clone search engine Kam1n0 and its detailed contribution to the domain of binary analysis and data mining. Chapter 3 presents our cross-architecture subgraph clone search engine Sym1n0 and describes how it contributes to symbolic expression retrieval literature. Chapter 4 presents our machine learning model that learns interesting or unique patterns from the assembly code and reports how it mitigates the effects of optimization and obfuscation. Chapter 5 presents our machine learning model that models different aspects of an executable and is able to accurately predict its behaviors. While Chapter 2, 3, 4, and 5 each contain their own conclusions given the context of different related literature, Chapter 6 summarizes the contributions and envisions the future directions.

2

Assembly Subgraph Clone Search

Given the fact that code reuse is prevalent in software development, there is a pressing need to develop an efficient and effective assembly clone search engine for reverse engineers. Assembly clone search addresses the information need of reverse engineers: the assembly code fragment under analysis is a duplicate of an existing known one. This assumption is valid due to the prevalence of code reuse on the source level. Previous clone search approaches only focus on the search accuracy. However, designing a practical useful clone search engine is a non-trivial task that involves multiple factors to be considered. By closely collaborating with reverse engineers and *Defence Research and Development Canada* (*DRDC*), we outline the deployment challenges and requirements as follows:

Interpretability and usability: An assembly function can be represented as a control flow graph consisting of connected basic blocks. Given an assembly function as query, all of the previous assembly code clone search approaches [24], [36], [69], [114] only provide the top-listed candidate assembly functions. They are useful when there exists a function in the repository that shares a high degree of similarity with the query. However, due to the unpredictable effects of different compilers, compiler optimization, and obfuscation techniques, given an unknown function it is less probable to have a very similar function in the repository. Returning a list of clones with a low degree of similarity values is not useful. As per our discussions with DRDC, a practical search engine should be able to decompose the given query assembly function to different known subgraph clones, which can help reverse engineers better understand the function's composition.

We define a *subgraph clone* as one of its subgraphs that can be found in the other function. Refer to the example in Figure 2.1. The previous clone search approaches cannot address this challenge.

Efficiency and scalability: An efficient engine can help reverse engineers retrieve clone search results on-the-fly when they are conducting an analysis. Instant feedback tells the reverse engineer the composition of a given function that is under investigation. Scalability is a critical factor as the number of assembly functions in the repository needs to scale up to millions. The degradation of search performance against the repository size has to be considered. Previous approaches [24], [36] that trade efficiency and scalability for better accuracy have a high latency for queries and are thus not practically applicable.

Incremental updates: The clone search engine should support incremental updates to the repository without re-indexing existing assembly functions. *BinClone* [36] requires median statistics to index each vector, and the model for fine-grain clone detection [114] requires data-dependent settings for its index, so neither satisfies this requirement.

Clone search quality: Practically, clones among assembly functions are one-to-many mappings, i.e., a function has multiple cloned functions with different degrees of similarity. However, previous approaches [24], [36], [69], [114] assume that clones are one-to-one mappings in the experiment. This is due to the difficulty of acquiring such a one-to-many labeled dataset. Moreover, they use different evaluation metrics. Therefore, it is difficult to have a direct comparison among them with respect to the search quality. We need to develop a one-to-many labeled dataset and a unified evaluation framework to quantify the clone search quality.

To address the above requirements and challenges we propose a new variant of the *Locality Sensitive Hashing (LSH)* scheme and incorporate it with a graph matching technique. We also develop and deploy a new assembly clone search engine called *Kam1n0*. Our main contributions can be summarized as follows:

- **Solution to a challenging problem for the reverse engineering community:** Kam1n0 is the first assembly code clone search engine that supports subgraph clone search. Refer to the example in Figure 2.1. It promotes interpretability and usability by providing subgraph clones as results, which helps reverse engineers analyzing new and unknown assembly func-

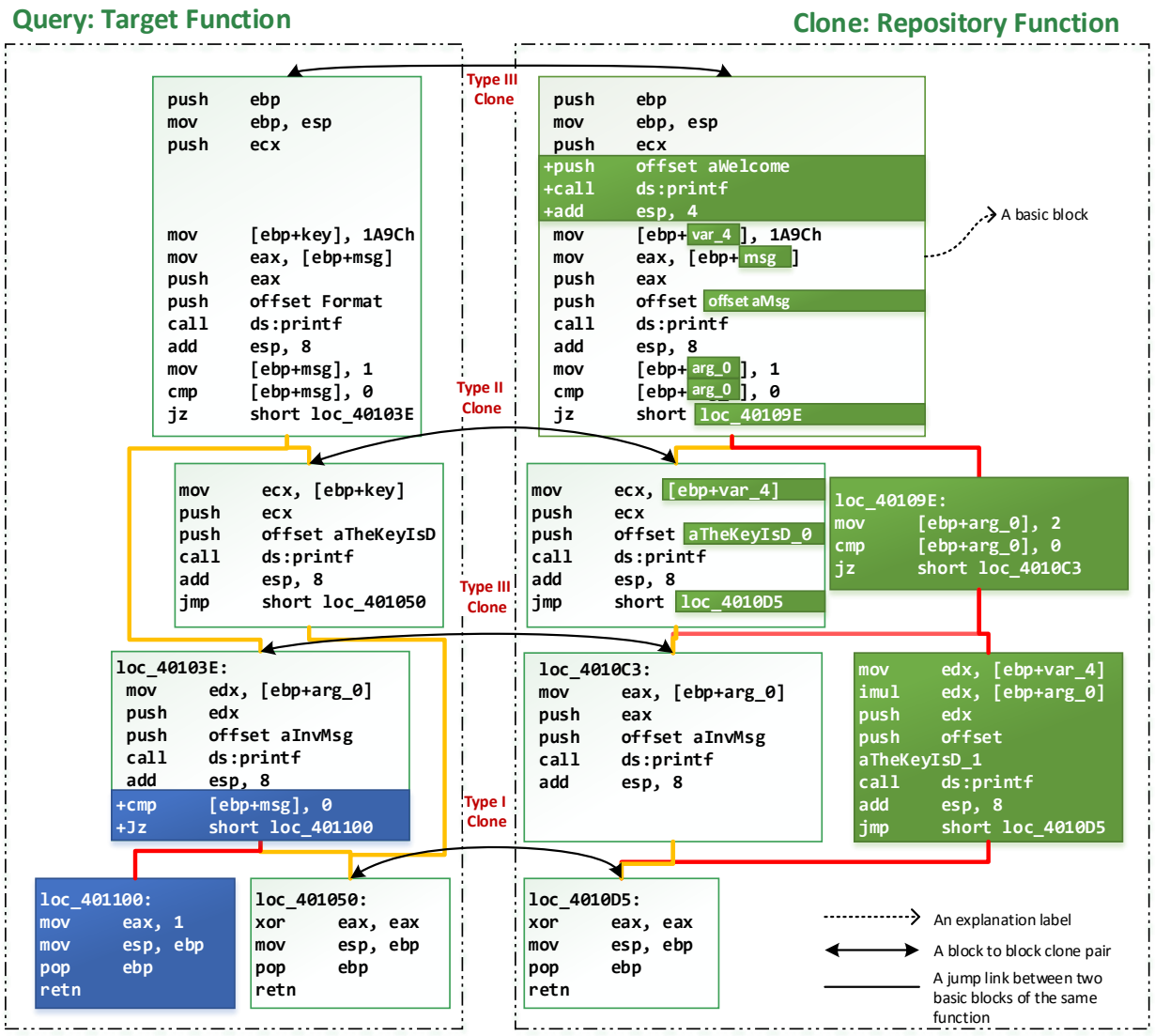

Figure 2.1: An example of the clone search problem. Basic blocks with a white background form a *subgraph clone* between two functions. Three types of code clones are considered in this chapter: Type I: literally identical; Type II: syntactically equivalent; and Type III: minor modifications.

tions. Kam1n0 won the second prize at the 2015 Hex-Rays plugin Contest[1], and its code is publicly accessible on GitHub.[2]

- **Efficient inexact assembly code search:** The assembly code vector space is highly skewed. Small blocks tend to be similar to each other, and large blocks tend to be sparsely distributed in the space. Original hyperplane hashing with a banding technique equally partitions the space and does not handle the unevenly distributed data well. We propose a new *adaptive locality sensitive hashing* (*ALSH*) scheme to approximate the cosine similarity. To the best of our knowledge, ALSH is the first incremental locality sensitive hashing scheme that solves this issue specifically for cosine space with theoretical guarantee. It retrieves fewer points for dense areas and more points for sparse ones in the cosine space. It is therefore efficient in searching nearest neighbors.
- **Scalable sub-linear subgraph search:** We propose a *MapReduce* subgraph search algorithm based on the *Apache Spark* computational framework without an additional index. Unlike the existing subgraph isomorphism search problem in data mining, we need to retrieve subgraphs that are both isomorphic to the query and the repository functions as graphs. Thus, existing algorithms are not directly applicable. Algorithmically, our approach is bounded by polynomial complexity. However, our experiment suggests that it is sub-linear in practice.
- **Accurate and robust function clone search:** Kam1n0 is the first approach that integrates both inexact assembly code and subgraph search. Previous solutions do not consider both of them together. Our experiments suggest that Kam1n0 boosts the clone search quality and yields stable results across different datasets and metrics.
- **Development of a labeled dataset and benchmark state-of-the-art assembly code clone solutions.** We carefully construct a new *labeled* one-to-many assembly code clone dataset that is available to the research community by linking the source code and assembly function level clones. We benchmark and report the performance of twelve existing state-of-the-art solutions with Kam1n0 on the dataset using several metrics. We also set up a mini-cluster to evaluate the scalability of Kam1n0.

The remainder of this paper is organized as follows. Section 2.1 situates our study within the

[1] https://hex-rays.com/contests/2015/
[2] https://github.com/McGill-DMaS/Kam1n0-Plugin-IDA-Pro

literature of three different research problems. Section 2.2 formally defines the studied problem. Section 2.3 provides an overview of our solution and system design. Section 2.4 presents the preprocessing steps and the chosen vector space. Section 2.5 introduces our proposed locality sensitive hashing scheme. Section 2.6 presents our graph search algorithm. Section 2.7 presents our benchmark experiments. Section 2.8 concludes this chapter.

2.1 Related Work

Locality sensitive hashing. *Locality sensitive hashing* (*LSH*) has been studied for decades to solve the ϵ-approximated Nearest Neighbor (ϵNN) problem because exact nearest neighbor does not scale to high dimensional data. One of the prevalent problems of LSH is the *uneven data distribution issue*, as LSH equally partitions the data space. To mitigate this issue, several approaches have been proposed including *LSH-Forest* [9], *LSB-Forest* [129], *C2LSH* [42], and *SK-LSH* [84]. It has been shown that the cosine vector space is robust to different compiler settings [69] in assembly code clone search. Especially, for the loop unrolling code optimization, cosine similarity remains the same for the unrolled and repeated assembly fragments. However, LSH-Forest, C2LSH, and SK-LSH are designed for the p-stable distribution, which does not fit the cosine space. LSB-Forest dynamically and unequally partitions the data space. As pointed out by other studies [130], it requires the hash family to possess the $(\epsilon, f(\epsilon))$ property. However, to our best knowledge, such a family in cosine space is still unknown. There are other learning-based approaches [44] that do not meet our incremental requirement. Wang et al. [136] provide a more comprehensive survey on LSH. To satisfy our requirements, we propose the ALSH scheme specifically for the cosine space. Different to the LSH-Forest, ALSH takes more than one bit when going down the tree structure and does not require the $(\epsilon, f(\epsilon))$ property for the LSH family to have theoretical guarantee. Unlike LSB-Forest [9], we dynamically construct the buckets to adapt to different data distributions.

Subgraph isomorphism. Ullmann [133] proposed the first practical subgraph isomorphism algorithm for small graphs. Several approaches were proposed afterwards for large scale graph data, such as *TurboISO* [53] and *STwig* [128]. It has been shown that they can solve the subgraph isomorphism problem in a reasonable time. However, they do not completely meet our problem settings. The subgraph isomorphism problem needs to retrieve subgraphs that are isomorphic to

the graphs in the repository and identical to the query. However, we need to retrieve subgraphs that are isomorphic to the graphs in the repository and isomorphic to the graph of the query, which significantly increases the complexity. More details will be discussed in Section 2.6. Such a difference requires us to propose a specialized search algorithm. Lee et al. [80] provide a comprehensive survey and performance benchmark on subgraph isomorphism.

Assembly code clone search. The studies on the assembly code clone search problem are recent. Only a few approaches exist [24], [36], [69], [114]. They all rely on the inexact text search techniques of data mining. *BinClone* [36] models assembly code into a Euclidean space based on frequency values of selected features. It is inefficient and not scalable due to the exponential 2-combination of features that approximates the 2-norm distance. *LSH-S* [114] models assembly code into a cosine space based on token frequency and approximates the distance by hyperplane hashing and banding scheme. It equally partitions the space and suffers from the uneven data distribution problem. *Graphlet* [69] models assembly code into a cosine space based on extracted signatures from assembly code. However, it cannot detect any subgraph clones smaller than the graphlet size. *Tracelet* [24] models assembly code according to string editing distance. It compares functions one by one, which requires a quadratic complexity and is not scalable. Kam1n0 is fundamentally different to the previous approaches. It is an integration of inexact assembly code search and the subgraph search. It enables clone subgraph search of any size.

2.2 Problem Statement

Reverse engineering starts from a binary file. After being unpacked and disassembled, it becomes a list of assembly functions. In this chapter, *function* represents an assembly function; *block* represents a basic block; *source function* represents the actual function written in source code, such as C++; *repository function* stands for the assembly function that is indexed inside the repository; *target function* denotes the assembly function that is given as a query; and correspondingly, *repository blocks* and *target blocks* refer to their respective basic blocks. Each function f is represented as a control flow graph denoted by (B, E), where B indicates its basic blocks and E indicates the edges that connect the blocks. Let $B(\mathrm{RP})$ be the complete set of basic blocks in the repository and $F(\mathrm{RP})$ be the complete set of functions in the repository. Given an assembly function, our

2.2 Problem Statement

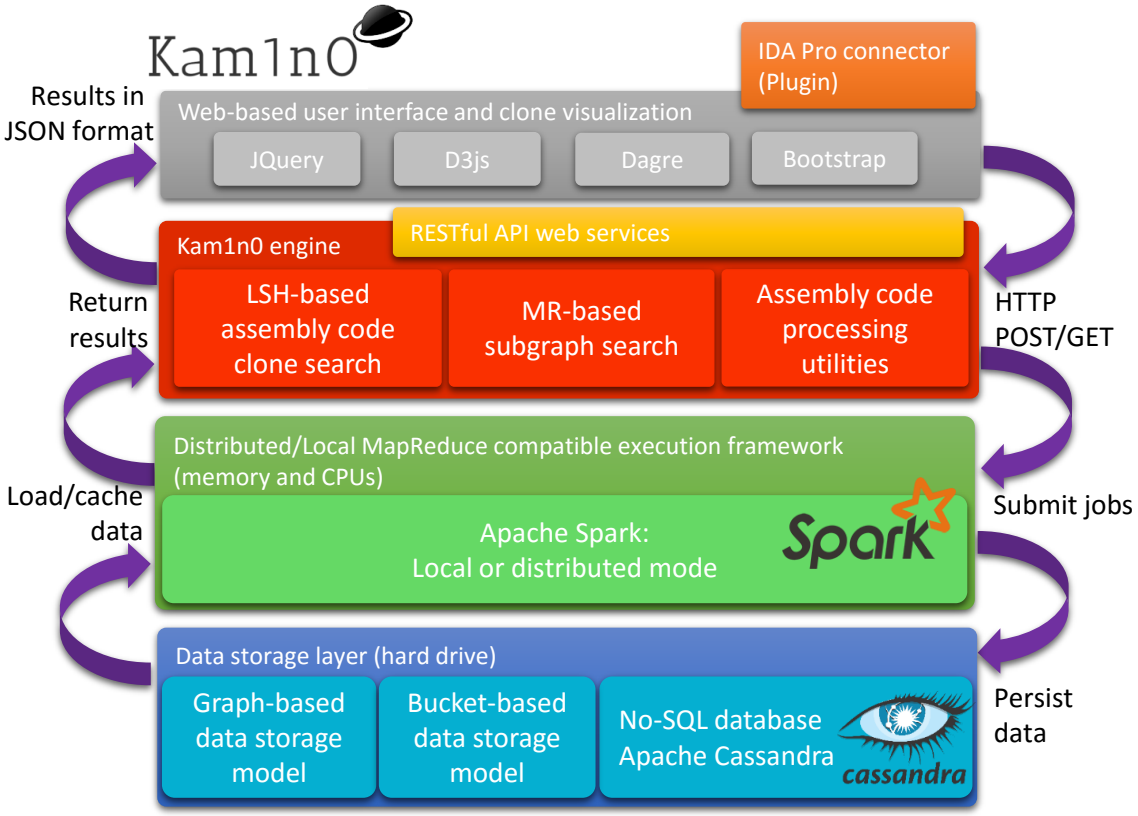

Figure 2.2: The overall solution stack of the Kam1n0 engine.

goal is to search all of its subgraph clones inside the repository RP. We formally define the search problem as follows:

Definition 1 *(Assembly function subgraph clone search) Given a target function f_t and its control flow graph (B_t, E_t), the search problem is to retrieve all the repository functions $f_s \in \mathrm{RP}$ that share at least one subgraph clone with f_t. The shared list of subgraph clones between f_s and f_t is denoted by $sg_s[1:a]$, where $sg_s[a]$ represents one of them. A subgraph clone is a set of basic block clone pairs $sg_s[a] = \{\langle b_t, b_s\rangle, \dots\}$ between f_s and f_t, where $b_t \in B_t$, $b_s \in B_s$, and $\langle b_t, b_s\rangle$ is a type I, type II, or type III clone (see Figure 2.1). Formally, given f_t, the problem is to retrieve all $\{f_s | f_s \in \mathrm{RP}$ and $|sg_s| > 0\}$.* □

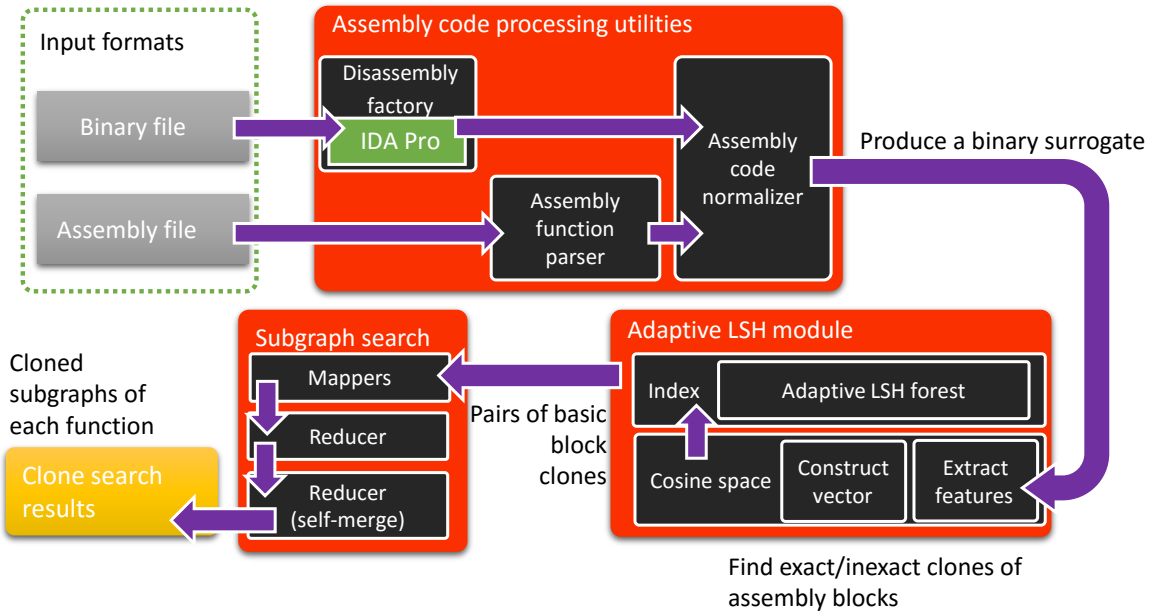

Figure 2.3: Assembly clone search data flow.

2.3 Overall Architecture

The Kam1n0 engine is designed for general key-value storage and builds upon the *Apache Spark*[3] computational framework. Its solution stack, as shown in Figure 2.2, consists of three layers. The data storage layer is concerned with how the data is stored and indexed. The distributed/local execution layer manages and executes the jobs submitted by the Kam1n0 engine. The Kam1n0 engine splits a search query into multiple jobs and coordinates their execution flow. It also provides the RESTful APIs. We have implemented a web-based user interface and an *Hex-Rays IDA Pro* plugin[4] as clients. IDA Pro is a popular interactive disassembler that is used by reverse engineers.

Figure 2.3 depicts the data flow of the clone search process. It consists of the following steps. *Preprocessing*: After parsing the input (either a binary file or assembly functions) into control flow graphs, this step normalizes assembly code into a general form, which will be elaborated in Section 2.4. *Find basic blocks clone pairs:* Given a list of assembly basic blocks from the previous

[3]Apache Spark, available at: `http://spark.apache.org/`
[4]IDA Pro, available at: `http://www.hex-rays.com/`

step, it finds all the clone pairs of blocks using ALSH. *Search the subgraph clones:* Given the list of clone block pairs, the MapReduce module merges and constructs the subgraph clones. Note that this clone search process does not require any source code.

2.4 Preprocessing and Vector Space

We choose the cosine vector space to characterize the semantic similarity of assembly code. It has been shown that the cosine vector space is robust to different compiler settings [69]. It can mitigate the linear transformation of assembly code. For example, to optimize the program for speed, the compiler may unroll and flatten a loop structure in assembly code by repeating the code inside the loop multiple times. In this case, the cosine similarity between the unrolled and original loop is still high due to the fact that the cosine distance only considers the included angle between two vectors. The features selected in Kam1n0 include mnemonics, combinations of mnemonics and operands, as well as mnemonics n-gram, which are typically used in assembly code analysis [36], [114]. The equivalent assembly code fragments can be represented in different forms. To mitigate this issue, we normalize the operands in assembly code during the preprocessing. We extend the normalization tree used in BinClone [36] with more types. There are three normalization levels: *root*, *type*, and *specific*. Each of them corresponds to a different generalization level of assembly code. More details can be found in the appendix.

2.5 Locality Sensitive Hashing

In this section, we introduce an *Adaptive Locality Sensitive Hashing* (*ALSH*) scheme for searching the block-level semantic clones. As discussed in Section 2.1, exact nearest neighbor search is not scalable. Thus, we start from the reduction of the ϵ-approximated k-NN problem:

Definition 2 *(ϵ-approximated NN search problem) Given a dataset $D \subset \mathbb{R}^d$ ($\mathbb{R}$ denotes real numbers) and a query point q, let r denote the distance between the query point q and its nearest neighbor o^*. This problem is to find an approximated data point within the distance $\epsilon \times r$ where $\epsilon > 1$.* $\square$

The ϵ-approximated k-NN search problem can be reduced to the ϵNN problem by finding the

k data points where each is an ϵ-approximated point of the exact k-NN of q [42]. The locality sensitive hashing approaches do not solve the ϵNN problem directly. ϵNN is further reduced into another problem: (ϵ, r)-approximated ball cover problem [4], [54].

Definition 3 *((ϵ, r)-approximated ball cover problem) Given a dataset $D \subset \mathbb{R}^d$ and a query point q, let $B(q, r)$ denote a ball with center q and radius r. The query q returns the results as follows:*

- *if there exists a point $o^* \in B(q, R)$, then return a data point from $B(q, \epsilon R)$*
- *if $B(q, \epsilon R)$ does not contain any data object in D, then return nothing.* □

One can solve the (ϵ, r) ball cover problem by using the Locality Sensitive Hashing (LSH) families. A locality sensitive hashing family consists of hashing functions that can preserve the distance between points.

Definition 4 *(Locality Sensitive Hashing Family) Given a distance r under a specific metric space, an approximation ratio ϵ, and two probabilities $p_1 > p_2$, a hash function family $\mathcal{H} \rightarrow \{h : \mathbb{R}^d \rightarrow U\}$ is $(r, \epsilon r, p1, p2)$-sensitive such that:*

- *if $o \in B(q, r)$, then $Pr_{\mathcal{H}}[h(q) = h(o)] \geqslant p_1$*
- *if $o \notin B(q, \epsilon r)$, then $Pr_{\mathcal{H}}[h(q) = h(o)] \leqslant p_2$* □

LSH families are available for many metric spaces such as cosine similarity [18], hamming distance [54], Jaccard coefficient, and p-stable distributions [21]. Based on our chosen cosine vector space, we adopt the random hyperplane hash [18] family, where $sign(\cdot)$ outputs the sign of the input.

$$h(\vec{o}) = sign(\vec{o} \cdot \vec{a}) \tag{2.1}$$

By substituting the random vector $\vec{a}$ we can obtain different hash functions in the family. The collision probability of two data points $\vec{o_1}$ and $\vec{o_2}$ on Equation 2.1 can be formulated as:

$$P[h(\vec{o_1}) = h(\vec{o_2})] = 1 - \frac{\theta_{\vec{o_1}, \vec{o_2}}}{\pi} \tag{2.2}$$

$\theta_{\vec{o_1}, \vec{o_2}}$ is the included angle between $\vec{o_1}$ and $\vec{o_2}$. The probability that two vectors have the same projected direction on a random hyperplane is high when their included angle is small.

Theorem 1 *The random hyperplane hash function is a* $(r, \epsilon r, 1 - r/\pi, 1 - \epsilon r/\pi)$ *sensitive hashing family.* □

Proof 1 *According to Definition 4 and Equation 2.2:* $p_1 = 1 - r/\pi$ *and* $p_2 = 1 - \epsilon r/\pi$

To use the locality sensitive hashing families to solve the ball cover problem, it needs a hashing scheme to meet the quality requirement. The *E2LSH* [54] and the extended one [21]. It concatenates k different hash functions $[h_1, \ldots, h_k]$ from a given LSH family $\mathcal{H}$ into a function $g(o) = (h_1(o), \ldots, h_k(o))$, and adopts l such functions. The parameters k and l are chosen to ensure the following two properties are satisfied:

Property 1 *(*P_1*): if there exists* $p^* \in B(q, r)$*, then* $g_j(p^*) = g_j(q)$ *from some* $j = 1 \ldots l$*.* □

Property 2 *(*P_2*): the total number of points* $\notin B(q, \epsilon r)$ *that collides with* q *is less than* $2l$*.* □

It is proven that if the above two properties hold with constant probability, the algorithm can correctly solve the (ϵ, r)-approximated ball cover problem [54]. For E2LSH, by picking $k = log_{p_2}(1/n)$ and $l = n^\rho$ where $\rho = \frac{ln1/p_1}{ln1/p_2}$, both Properties 1 and 2 hold with constant probability.

However, the ball cover problem is a strong reduction to the NN problem since it adopts the same radius r for all points. Real-life data cannot always be evenly distributed. Therefore, it is difficult to pick an appropriate r. We denote this as the *uneven data distribution issue*. A magic r_m is adopted heuristically [47]. But as pointed out by a studey [129], such a magic radius may not exist. A weaker reduction [54] is proposed, where the NN problem is reduced to multiple (r, ϵ)-NN ball cover problems with varying $r = \{1, \epsilon^2, \epsilon^3, \ldots\}$. The intuition is that points in different density areas can find a suitable r. However, such a reduction requires a large space consumption and longer response time. Other indexing structures have been proposed to solve this issue. Per our discussion in Section 2.1, existing techniques do not meet our requirement. Thus, we customize the LSH-forest approach and propose the ALSH structure.

2.5.1 Adaptive LSH Structure

We found that the limitation of the expanding sequence of r in the previous section is too strong. It is unnecessary to *exactly* follow the sequence $r = \{1, \epsilon^2, \epsilon^3, \dots\}$, as long as r is increasing in a similar manner to $r_{t+1} = r_t \times \epsilon$. Thus, we customize the ϵ-approximated NN problem as follows:

Definition 5 *($f(r)$-approximated NN search problem) Given a dataset $D \subset \mathbb{R}^d$ and a query point q, let r denote the distance between the query point q and its nearest neighbor o^*. The problem is to find an approximated data point within the distance $f(r)$, where $f(r)/r > 1$* □

Instead of using a fix approximation ratio, we approximate the search by using a function on r. We issue a different sequence of expanding r. The expanding sequence of r is formulated as $r_0, r_1, \dots, r_t, r_{t+1}, \dots, r_m$, where $r_t < r_{t+1}$. Similar to the E2LSH approach, we concatenate multiple hash functions from the random hyperplane hash family $\mathcal{H}$ into one. However, we concatenate a different number of hash functions for different values of r. This number is denoted by k_t for r_t, and the sequence of k is denoted by $k_0, k_1, \dots, k_t, k_{t+1}, \dots, k_m$, where $k_t > k_{t+1}$. Recall that the concatenated function is denoted by g. Consequently, there will be a different function g at position t, which is denoted by g_t. Yet, function g_t and function $g_{(t+1)}$ can share k_{t+1} hash functions. With p_m to be specified later, we set the r value at position t as follows:

$$r_t = \pi \times (1 - p_m^{(1/k_t)}) \tag{2.3}$$

This allows us to have the effect of increasing the r value by decreasing the k value. We calculate the value of k at position t as follows:

$$k_t = c \times k_{t+1}, \text{where } \; c > 1 \tag{2.4}$$

By getting t_{t+1} from Equation 2.3, substituting k_t using Equation 2.4, and substituting p_m using Equation 2.3, we have:

$$r_{t+1} = \pi \times \left(1 - (1 - \frac{r_t}{\pi})^c\right) = f_c(r_t) \quad f_c(r_t)/r_t > 1 \tag{2.5}$$

By setting c equals to 2, we can get an approximately similar curve of r sequence to the original

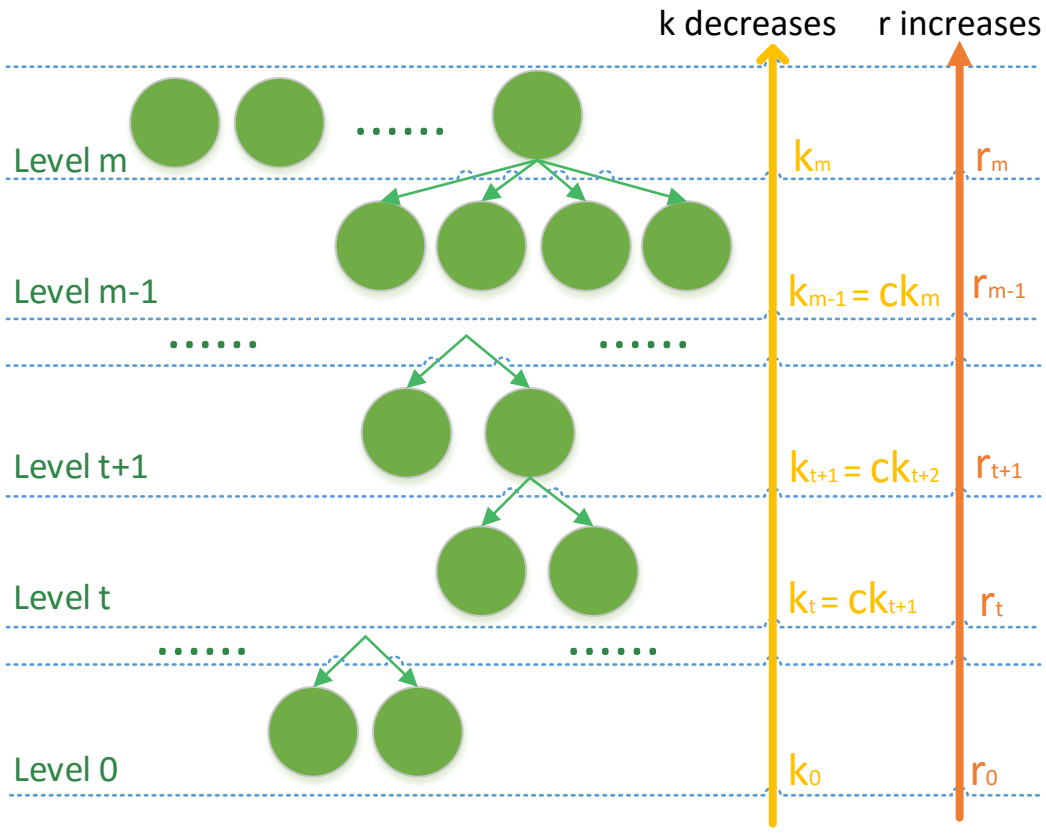

Figure 2.4: The index structure for the Adaptive Locality Sensitive Hashing (ALSH). There are $m + 1$ levels on this tree. Moving from level t to level $t + 1$ is equivalent to increasing the search radius from r_t to r_{t+1}.

sequence $r_{t+1} = r_t \times \epsilon$ where ϵ equals to 2. Following the aforementioned logic, we construct an Adaptive Locality Sensitive Hashing (ALSH) index in the form of prefix trees.

As shown in Figure 2.4, the index structure is a prefix tree of the signature values calculated by $\mathcal{G} = \{g_m, g_{m-1}, \ldots, g_0\}$. Level t corresponds to the position t in the r expanding sequence. By introducing different values of k_t, each level represents a different radius r_t. Each level denotes a different g_t function, and the g_t function is a concatenation of k_t hash functions. Moving up from a node at level t to its parent at level $t + 1$ indicates that it requires a shorter matched prefix. The nodes that have the same parent at level t share the same prefix that is generated by g_t.

2.5 Locality Sensitive Hashing

To locate the leaf for a given data point $q \in \mathbb{R}^d$, ALSH dynamically constructs the hash functions by trying $g_t \in \mathcal{G}$ in sequence. The signature of g_t can be generated by padding additional hash values to g_{t+1} since $k_t = c \times k_{t+1}$. Following previous studies [54], [129], with l to be specified later, we adopt l such prefix trees as the ALSH's index. Given a query point q, we first locate the corresponding leaves in all prefix trees. With l to be specified later, we collect the first $2l$ points from all the leaf buckets. To index a point, we locate its corresponding leaf in each tree and insert it into the leaf bucket. Suppose a leaf is on level t_{t+1}. If the number of points in that leaf is more than $2l$, we split all the data points of that leaf into the next level t by using g_t. All the trees are dynamically constructed based on the incoming points to be indexed in sequence. Therefore, they can be incrementally maintained. Unlike the learning-based LSH [44], Kam1n0 does not require the whole repository to estimate the hash functions to build the index.

It can be easily proved that g_t is a $(r_t, r_{t+1}, p_m, p_m^c)$-sensitive hash family, and g_t can correctly solve the $(r_{t+1}/r_t, r_t)$-approximated ball cover problem by setting $p_m^c = 1/n$ and $l = n^{1/c}$. The proof follows the previous study [54]. Details and implementation on key-value data store can be found in our implementation details. Another parameter r_m controls the starting k_m value at the root value. It indicates the maximum distance that two points can be considered as valid neighbors. For sparse points far away from each other, they are not considered as neighbors unless their distance is within r_m.

For a single ALSH tree, the depth in the worst case is k_0, and all the leaves are at level 0. In this case, the tree is equivalent to the E2LSH with $k = k_0$. Therefore, the space consumption for $l = n^{1/c}$ ALSH trees is bounded by $O(dn + n^{1+1/c})$, where $O(dn)$ is the data points in a dataset and $O(n^{1+1/c})$ is the space of indexes for the trees. The query time for a single ALSH prefix tree is bounded by its height. Given the maximum k value k_0 and the minimum k value k_m, its depth in the worst case is $log_c(k_0/k_m) + 1$. Thus, the query time for $l = n^{1/c}$ prefix trees is bounded by $O(log_c(k_0/k_m) \times n^{1/c})$. The ALSH index needs to build $n^{1/c}$ prefix trees for the full theoretical quality to be guaranteed. Based on our observation, setting l to 1 and 2 is already sufficient for providing good quality assembly code clones.

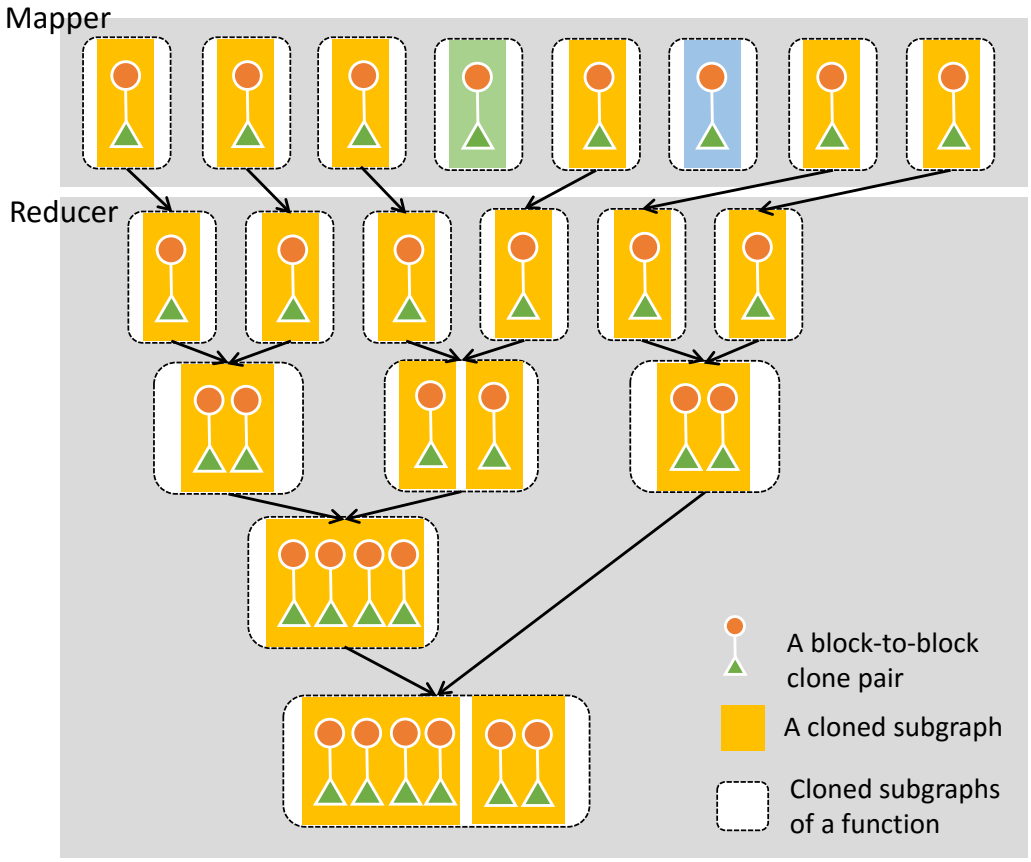

Figure 2.5: The MapReduce-based subgraph clone construction process.

2.6 Subgraph Clone Search

Subgraph isomorphism is NP-hard in theory [80], [122], and many algorithms have been proposed to solve it in a reasonable time. Formally, the subgraph isomorphism algorithm solved by most of these systems [53], [80], [128] is defined as:

Definition 6 *(Subgraph Isomorphism Search) A graph is denoted by a triplet* (V, E, L) *where* V *represents the set of vertices,* E *represents the set of edges, and* L *represents the labels for each vertex. Given a query graph* $q = (V, E, L)$ *and a data graph* $g = (V', E', L')$*, a subgraph isomorphism (also known as embedding) is an injective function* $M : V \rightarrow V'$ *such that the following conditions hold: (1)* $\forall u \in V, L(u) \in L'(M(u))$*, (2)* $\forall (u_i, u_j) \in E, (M(u_i), M(u_j)) \in E'$*, and (3)* $L(u_i, u_j) = L'(M(u_i), M(u_j))$*. The search problem is to find all distinct embeddings of* q *in* g*.* □

The difference between this problem and ours in Definition 1 is two-fold. First, our problem is to retrieve *all the subgraph clones* of the target function f_t's control flow graph from the repository. In contrast, this problem only needs to retrieve the exact matches of query graph q within g. Refer to Conditions 1, 2, and 3 in Definition 6, or the termination condition of the procedure on Line 1 of subroutine $SubgraphSearch$ in [80]. Our problem is more challenging and can be reduced to the problem in Definition 6 by issuing all the subgraphs of f_t as queries, which introduces a higher algorithmic complexity. Second, there is no such L data label attribute in our problem, but two types of edges: the control flow graph that links the basic blocks and the semantic relationship between basic blocks that is evaluated at the querying phase. Existing algorithms for subgraph isomorphism are not directly applicable. Assembly code control graphs are sparser than other graph data as there are fewer links between vertices, and, typically, basic blocks are only linked to each other within the same function. Given such properties, we can efficiently construct the subgraph clones respectively for each repository function f_s if it has more than one clone block in the previous step.

Algorithm 1 Kam1n0 Mapper

Input A basic block clone pair $\langle b_t, b_s \rangle$
Output A pair consisting of $\langle f_s, sg_s \rangle$

1: $f_s \leftarrow$ getFunctionId(b_s)
2: $sg_s \leftarrow [\,]$ ▷ create an empty list of subgraph clones
3: $cloneGraph \leftarrow \{\langle b_t, b_s \rangle\}$ ▷ create a subgraph clone with one clone pair
4: $sg_s[0] \leftarrow cloneGraph$ ▷ list of subgraph clones, but at this moment it has only one.
5: **return** $\langle f_s, sg_s \rangle$ with f_s as key index

2.6.1 MapReduce Subgraph Search

We adopt two functions in the Apache Spark MapReduce execution framework, namely the *map* function and the *reduce-by-key* function. In our case, the map function transforms the clone pairs generated by ALSH (refer to the data flow in Figure 2.3) and the reduce-by-key function constructs subgraph clone respectively for each unique repository function f_s. Figure 2.5 shows the overview of our subgraph clone search approach.

The signature for the map function (Algorithm 1) is $\langle b_t, b_s \rangle \rightarrow \langle f_s, sg_s[1:a] \rangle$. Each execution of the map function takes a clone pair $\langle b_t, b_s \rangle$ produced by ALSH and transforms it to $\langle f_s, sg_s[1:a] \rangle$, which is a pair of repository function id f_s and its list of subgraph clones sg_s in Definition 1. The map functions are independent of each other.

The outputs of the map functions correspond to the first row in Figure 2.5. A red circle represents a target basic block b_s, and a green triangle represents a source basic block b_t. The link between them indicates that they are a block-to-block clone pair $\langle b_t, b_s \rangle$, which is produced in the previous step. A white rectangle represents a list of subgraph clones, and the colored rectangle inside it represents a subgraph clone. Algorithm 1 maps each clone pair into a list of subgraph clones that contains only one subgraph clone. Each subgraph clone is initialized with only one clone pair.

After the map transformation functions, the reduce-by-key function reduces the produced lists of subgraph clones. The reducer merges a pair of lists sg_s^1 and sg_s^2 into a single one by considering their subgraph clones' connectivity. The reduce process is executed for the links from the second row to the last row in Figure 2.5. Only the lists of subgraph clones with the same f_s will be merged.

As indicated by the links between the first and second row in Figure 2.5, only rectangles with the same orange background can be reduced together. Rectangles with other background colors are reduced with their own group.

Algorithm 2 shows the reduce function in detail. Given two lists of subgraph clones under the same repository function f_s, the reduce function compares their subgraph clones (Lines 1 and 2) and checks if two graphs can be connected (Lines 4 to 13) by referring to the control flow graph edges E_s and E_t . If the two subgraph clones can be connected by one clone pair, then they can be merged into a single one (Lines 6 and 7). If a subgraph clone from sg_s^2 cannot be merged into any of the subgraph clones in sg_s^2, it will be appended to the list sg_s^1 (Lines 20 and 21). At the end of the graph search algorithm, we solve the problem in Definition 1. In order to obtain a ranked list of repository functions for f_s, we calculate the similarity value by checking how much its subgraphs sg_s cover the graph of the query f_t: $sim_s = (|\text{uniqueEdges}(sg_s)| + |\text{uniqueNodes}(sg_s)|)/(|B_t| + |E_t|)$.

Compared to other join-based or graph-exploration-based search approach, our MapReduce-based search procedure avoids recursive search and is bounded by polynomial complexity. Let m_s be the number of clone pairs for a target function f_t. There are at most $O(m_s^2)$ connectivity checks between the clone pairs (no merge can be found), and the map function requires $O(m_s)$ executions. m_s corresponds to the number of rectangles in the second row of Figure 2.5. Refer from the second row to the last one. The reduce function is bounded by $O(m_s^2)$ comparisons. m_s is bounded by $O(|B_t| \times |B_s|)$, which implies that each basic block of f_t is a clone with all the basic blocks of f_s. However, this extreme case rarely happens. Given the nature of assembly functions and search scenarios, m_s is sufficiently bounded by $O(max(|B_t|, |B_s|))$. According to the descriptive statistics of our experiment, 99% of them have less than 200 basic blocks.

2.7 Experiments

This section presents comprehensive experimental results for the task of assembly code clone search. First, we explain how to construct a labeled dataset that can be used for benchmarking in future research. Then, we evaluate the effect of assembly code normalization. Although normalization has been extensively used in previous work, its effects have not been thoroughly studied yet.

Algorithm 2 Kam1n0 Reducer

Input subgraph lists of same f_s: $sg_s^1[1:a_1]$ and $sg_s^2[1:a_2]$
Output a single subgraph list $sg_s^1[1:a]$

1: **for** $a_1 \rightarrow |sg_s^1|$ **do**
2: **for** $a_2 \rightarrow |sg_s^2|$ **do**
3: $canMerge \leftarrow false$
4: **for each** $\langle b_t^{a_1}, b_s^{a_1} \rangle \in sg_s^1[a_1]$ **do**
5: **for each** $\langle b_t^{a_2}, b_s^{a_2} \rangle \in sg_s^2[a_2]$ **do**
6: **if** $E_t(b_t^{a_1}, b_t^{a_2})$ exists **then**
7: **if** $E_s(b_s^{a_1}, b_s^{a_2})$ exists **then**
8: $canMerge \leftarrow true$
9: goto Line 14.
10: **end if**
11: **end if**
12: **end for**
13: **end for**
14: **if** $canMerge$ is $true$ **then**
15: $sg_s^1[a_1] \leftarrow sg_s^1[a_1] \bigcup sg_s^2[a_2]$
16: $sg_s^2 \leftarrow sg_s^2 - sg_s^2[a_2]$
17: **end if**
18: **end for**
19: **end for**
20: **if** sg_s^2 is not $\emptyset$ **then** ▷ for graphs in sg_s^2 that cannot be merged, append them to the list
21: $sg_s^1 \leftarrow sg_s^1 \bigcup sg_s^2$
22: **end if**
23: **return** sg_s^1

Next, we present the benchmark results that compares Kam1n0 with state-of-the-art clone search approaches in terms of clone search quality. Finally, we demonstrate the scalability and capacity of the Kam1n0 engine by presenting the experimental results from a mini-cluster.

2.7.1 Labeled Dataset Generation

One of the challenging problems for assembly code clone search is the lack of a labeled (ground truth) dataset, since the most effective labeled dataset requires intensive manual identification of assembly code clones [36]. To facilitate future studies on assembly clone search, we have devel-

Library Name	Branch Count	Function Count	Block Count	Clone Pair Count
bzip2	5	590	15,181	1,329
curl	16	9,468	176,174	49,317
expat	3	2,025	35,801	14,054
jsoncpp	14	11,779	43,734	204,701
libpng	9	4,605	82,257	18,946
libtiff	13	7,276	124,974	51,925
openssl	9	13,732	200,415	29,767
sqlite	12	9,437	202,777	23,674
tinyxml	7	3,286	30,401	22,798
zlib	8	1,741	30,585	6,854
total	96	63,939	942,299	423,365

Table 2.1: The assembly code clone dataset summary.

oped a tool to systematically generate the assembly function clones based on source code clones. The tool performs four steps: *Step 1:* Parse all the source code functions from different branches or versions of a project and identify all the function-to-function clones using CCFINDERX [65], which estimates source code similarity based on the sequence of normalized tokens. *Step 2:* Compile the given branches or versions of a project with an additional debug flag to enable the compiler output debug symbols. *Step 3:* Link the source code functions to the assembly code functions using the compiler output debug symbols, where such information is available. *Step 4:* For each pair of source code clones, generate a pair of assembly function clones and transfer the similarity to the new pair.

The intuition is that the source code function-level clones indicate the functional clones between their corresponding assembly code. In [36], [91], the source code and assembly code are manually linked with an injected identifier in the form of variable declarations. However, after the automation of such process, we find that the link rate is very low due to the impact of compiler optimizations. The generated assembly code clone is in fact the combined result of source code patches and compiler optimizations. The source code evolves from version to version, and different versions may have different default compiler settings. Thus, the labeled dataset simulates the evaluation of assembly clone in a real-word setting. This tool is applicable only if the source code

2.7 Experiments

M	Approach	Bzip2	Curl	Expat	Jsoncpp	Libpng	Libtiff	Openssl	Sqlite	Tinyxml	Zlib	Avg.
AUROC	BinClone	.985	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	***.894**	.188
	Composite	.857	.766	.693	.725	.814	.772	.688	.726	.688	.729	.746
	Constant	.769	.759	.723	.665	.829	.764	.689	.776	.683	.768	.743
	Graphlet	.775	.688	.673	.563	.714	.653	.682	.746	.676	.685	.685
	Graphlet-C	.743	.761	.705	.604	.764	.729	.731	.748	.677	.668	.713
	Graphlet-E	.523	.526	.505	.516	.519	.521	.512	.513	.524	.514	.517
	MixGram	.900	.840	.728	.726	.830	.808	.809	.765	.707	.732	.785
	MixGraph	.769	.733	.706	.587	.755	.692	.713	.765	.674	.708	.710
	N-gram	.950	.860	.727	.713	.843	.809	.819	.789	.714	.766	.799
	N-perm	.886	.847	.731	.729	.834	.813	.811	.769	.709	.736	.787
	Tracelet	.830	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.799	.163
	LSH-S	.965	.901	.794	.854	.894	.922	.882	.845	.768	.758	.858
	Kam1n0	***.992**	***.989**	***.843**	***.890**	***.944**	***.967**	***.891**	***.895**	***.864**	.830	***.911**
AUPR	BinClone	.294	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.091	.038
	Composite	.645	.495	.375	.353	***.541**	.482	.288	.405	.261	.409	.425
	Constant	.247	.280	.301	.158	.311	.349	.072	.157	.142	.240	.226
	Graphlet	.162	.133	.138	.051	.115	.103	.041	.108	.150	.106	.111
	Graphlet-C	.455	.482	.296	.176	.413	.369	.366	.437	.245	.338	.358
	Graphlet-E	.022	.024	.013	.020	.012	.015	.004	.010	.020	.026	.017
	MixGram	.727	.598	.363	.337	.513	.512	***.464**	.471	.286	.383	.465
	MixGraph	.247	.242	.228	.098	.196	.184	.078	.180	.163	.175	.179
	N-gram	.638	.491	.297	.275	.408	.428	.301	.417	.264	.314	.383
	N-perm	.613	.589	.360	.344	.523	***.515**	.438	.465	.288	.370	.450
	Tracelet	.057	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.027	.008
	LSH-S	.227	.014	.095	.049	.035	.038	.012	.018	.079	.041	.061
	Kam1n0	***.780**	***.633**	***.473**	***.504**	.477	.387	.411	***.610**	***.413**	***.465**	***.515**
MAP@10	BinClone	.495	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.398	.089
	Composite	.505	.525	.489	.190	.493	.536	.238	.382	.303	.472	.413
	Constant	.354	.459	.539	.132	.473	.502	.199	.379	.229	.495	.376
	Graphlet	.274	.309	.408	.030	.264	.276	.154	.303	.233	.302	.255
	Graphlet-C	.339	.499	.586	.084	.412	.449	.284	.416	.272	.361	.370
	Graphlet-E	.021	.053	.010	.011	.024	.040	.012	.019	.039	.028	.026
	MixGram	.559	.641	.625	.191	.511	.589	.392	.445	.321	.474	.475
	MixGraph	.334	.407	.572	.064	.345	.351	.211	.387	.244	.371	.329
	N-gram	.620	.636	.615	.176	.512	.567	.398	.481	.310	.506	.482
	N-perm	.532	.653	.628	.191	.516	.597	.394	.452	.317	.483	.476
	Tracelet	.228	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.265	.049
	LSH-S	.322	.069	.198	.032	.145	.078	.086	.111	.130	.101	.127
	Kam1n0	***.672**	***.680**	***.690**	***.196**	***.548**	***.587**	***.434**	***.605**	***.375**	***.573**	***.536**

Table 2.2: Benchmark results of different assembly code clone search approaches. We employed three evaluation metrics: the *Area Under the Receiver Operating Characteristic Curve* (*AUROC*), the *Area Under the Precision-Recall Curve* (*AUPR*), and the *Mean Average Precision at Position 10* (*MAP@10*). Ø denotes that the method is not scalable and we cannot obtain a result for this dataset within 24 hours.

is available.

Refer to Table 2.1 for some popular open source libraries with different versions. We applied the aforementioned tool on them to generate the labeled dataset for the experiments. There are 63,939 assembly functions that successfully link to the source code functions. The labeled dataset is a list of one-to-multiple assembly function clones with the transferred similarity from their source code clones.

See Figure 2.6c and Figure 2.6d. The assembly function basic block count follows a long-tail distribution. Most of them have between 0 and 5 assembly basic blocks, and 99% of them are bounded by 200. We find that this is the typical distribution of assembly function block count. This distribution facilitates our graph search because the worst case is bounded by $O(|B_t| \times |B_s|)$ and $P[|B_s| < 200] > 0.99$. Figure 2.6b shows the cosine similarity distribution of each basic block's 20^{th}-nearest neighbor. It reflects variations of density in the vector space and calls for an adaptive LSH.

74% of the source code clones given by CCFINDERX are exact clones (see Figure 2.6a). However, by applying a strong hash on their assembly code, we find that only 30% of them are exact clones (Type I clones). Thus, the total percentage of inexact clones is $70\% \times 74\% + 26\% = 77.8\%$. CCFINDERX classifies tokens in source code into different types before clone detection. If two source code fragments are identified as clones with a low similarity, there is a higher chance that the underlying assembly code is indeed not a clone due to the normalization of the source code. To mitigate this issue, we heuristically set a 0.4 threshold for clones to be included in our dataset. Thus, we have 66.8% of inexact clones.

2.7.2 Normalization Level

	None	Root	Specific
Root	$< 2e^{-16}$	-	-
Specific	$< 2e^{-16}$	1	-
Type	$< 2e^{-16}$	1	1

Table 2.3: Paired t-test on the normalization level.

Assembly code normalization is used in previous studies [36], [115]. However, its effects were

not formally studied. In this section, we present the results of the statistical test on the effects of the normalization level. Details on normalization can be found in our implementation details. We start by using a strong hash clone search with different normalization levels on each of the generated datasets. Then, we collect the corresponding precision value to the given normalization level as samples and test the relationship between precision and the chosen normalization level. Normalization can increase the recall, but we want to evaluate the trade-off between the precision and different normalization levels. According to the ANOVA test ($p < 2e^{-16}$), the difference of applying normalization or not is statistically significant. Consider Table 2.3. However, the difference of applying different levels, namely *Root*, *Type*, or *Specific*, is not statistically significant.

2.7.3 Clone Search Approach Benchmark

In this section, we benchmark twelve assembly code clone search approaches: BinClone [37], [73], Graphlets [68], [69], LSH-S [124], and Tracelet [24]. [69] includes several approaches: mnemonic n-grams (denoted as n-gram), mnemonic n-perms (denoted as n-perm), Graphlets (denoted as Graphlet), Extended Graphlets (denoted as Graphlet-E), Colored Graphlets (denoted as Graphlet-C), Mixed Graphlets (denoted as MixGraph), Mixed n-grams/perms (denoted as MixGram), Constants, and the Composite of n-grams/perms and Graphlets (denoted as Composite). The idea of using Graphlet is originally proposed by Kruegel, Kirda, Mutz, *et al.* [75]. We re-implemented all these approaches under a unified evaluation framework and all parameters were configured according to the papers. We did not include the re-write engine [24] because it is not scalable.

Several metrics are used in previous research to evaluate the clone search quality, but there is no common agreement on what should be used. Precision, recall, and F1 are used in BinClone [36], while Welte [137] maintains that a F2 measure is more appropriate. However, these two values will change as the search similarity threshold value changes. To evaluate the trade-off between recall and precision, we use three typical information retrieval metrics, namely *Area Under the Receiver Operating Characteristic Curve* (*AUROC*), *Area Under the Precision-Recall Curve* (*AUPR*), and *Mean Average Precision at Position 10* (*MAP@10*). These three metrics favor different information retrieval scenarios. Therefore, we employ all of them. AUROC and AUPR can test a classifier by issuing different threshold values consecutively [38], [87], [125], while MAP@10 can evaluate the quality of the top-ranked list simulating the real user experience [86].

Table 2.2 presents the benchmark results. The highest score of each evaluation metric is highlighted for each dataset. Also, the micro-average of all the results for each approach is given in the *Avg* column. Kam1n0 outperforms the other approaches in most cases for all evaluation metrics. Kam1n0 also achieves the best averaged AUROC, AUPRC, and MAP@10 scores. The overall performance also suggests that it is the most stable one. Each approach is given a 24-hour time frame to finish the clone search, and it is limited to a single-thread pool for fair comparison. Some results for BinClone and Tracelet are empty, which indicates that they are not scalable enough to obtain the search result within the given time frame. Also, we notice that BinClone consumes more memory than the others for building the index, due to its combination of features that enlarges the feature space. We notice that the experimental results are limited within the context of CCFinder. In the future, we will investigate other source code clone detection techniques to generate the ground truth data.

2.7.4 Scalability Study

In this section, we evaluate Kam1n0's scalability on a large repository of assembly functions. We set up a mini-cluster on *Google Cloud* with four computational nodes. Each of them is a *n1-highmem-4* machine with 2 virtual cores and 13 GB of RAM. We only use regular persistent disks rather than solid state drives. Each machine is given 500 GB of disk storage. All the machines run on *CentOS*. Three machines run the *Spark Computation Framework* and the *Apache Cassandra Database*, and the other runs our Kam1n0 engine. To conduct the experiment, we prepare a large collection of binary files. All these files are either open source libraries or applications, such as *Chromium*. In total, there are more than 2,310,000 assembly functions and 27,666,692 basic blocks. Altogether, there are more than 8 GB of assembly code. We gradually index this collection of binaries in random order and query the zlib binary file of version 2.7.0 on Kam1n0 at every 10,000 assembly function indexing interval. As zlib is a widely used library, it is expected that it has a large number of clones in the repository. We collect the average indexing time for each function to be indexed, as well as the average time it takes to respond to a function query. Figure 2.7 depicts the average indexing and query response time for each function. The two diagrams suggest that Kam1n0 has a good scalability with respect to the repository size. Even as the number of functions in the repository increases from 10,000 to 2,310,000, the impact on the response time is negligible.

There is a spike at 910,000 due to the regular compaction routine in Cassandra, which increases I/O contention in the database.

2.8 Conclusion and Lesson Learned

Through the collaboration with Defence Research and Development Canada (DRDC), we learned that scalability, which was not considered in previous studies, is a critical issue for deploying a successful assembly clone search engine. To address this, we present the first assembly search engine that combines LSH and subgraph search. Existing off-the-shelf LSH nearest neighbor algorithms and subgraph isomorphism search techniques do not fit our problem setting. Therefore, we propose new variants of the LSH scheme and incorporate them with graph search to address the challenges. Experimental results suggest that our proposed MapReduce-based system, Kam1n0, is accurate, efficient, and scalable. Currently, Kam1n0 can only identify clones for x86/amd64 processor. In the future, we will extend it to the other processors and investigate approaches that can find clones between different processors. Kam1n0 provides a practical solution of assembly clone search for both DRDC and the reverse engineering community.

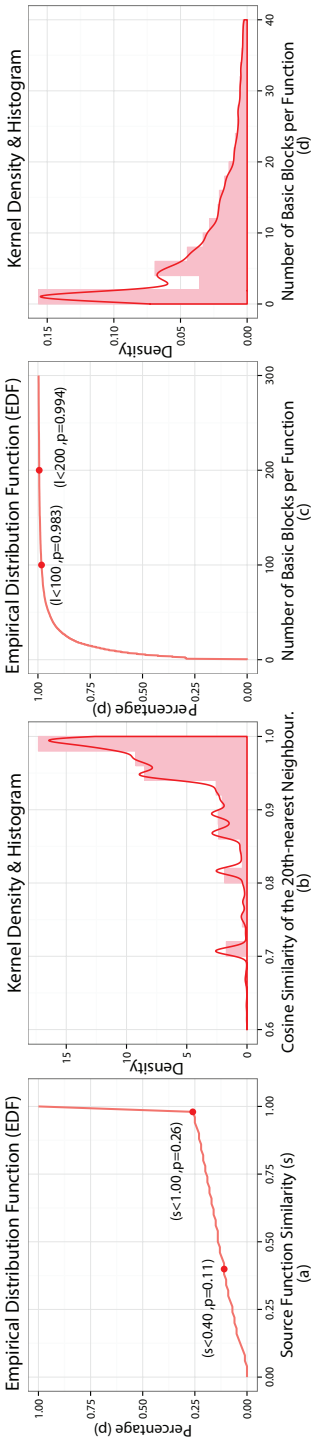

Figure 2.6: (a) the EDF function on repository function clone pair similarity, (b) the kernel density & histogram of the cosine similarity of each basic block's 20$^{\text{th}}$-nearest neighbours, (c) the EDF on per assembly function block count $|B_s|$, and (d) the kernel density & histogram on assembly function block count $|B_s| < 40$.

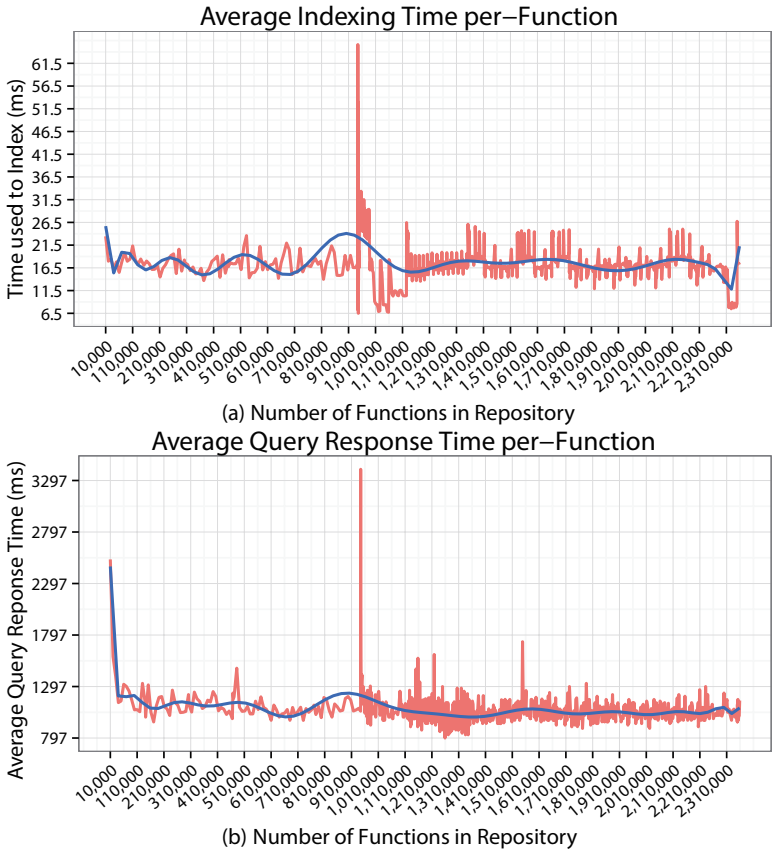

Figure 2.7: Scalability study. (a): Average Indexing Time vs. Number of Functions in the Repository. (b): Average Query Response Time vs. Number of Functions in the Repository. The red line represents the plotted time and the blue line represents the smoothed polynomial approximation.

3

Cross-Architecture Subgraph Clone Search

Most of the existing state-of-the-art assembly clone search techniques, such as *LSH-S* [114], n-gram [69], n-perm [69], *BinClone* [36], *Kam1n0* [30], and *Tracelet* [24], support a particular family of assembly language. Only few recent studies support cross-architecture clone search. Based on the employed features, they can be categorized into static or dynamic approaches. Dynamic approaches model the semantic similarity by dynamically analyzing the I/O behavior of assembly code [17], [22], [33], [103]. Static approaches model the similarity between assembly code by looking for their static differences with respect to the syntax or descriptive statistics [34], [39]. Static approaches are more scalable and provide better coverage than the dynamic approaches. Dynamic approaches are more robust against changes in syntax. Designing a practical and useful clone search engine is a non-trivial task, which involves several factors to be considered. Based on discussions with industrial reverse engineers, we identify the problems that these approaches failed to consider.

P1: Efficiency and scalability: An efficient search engine can provide instant search results for reverse engineers when they are conducting an analysis. The cost for getting results should be minimized to seconds even if the engine is given millions of candidates. One needs to consider the degradation of search performance against the increase in the repository size. All the existing dynamic approaches rely on a pairwise comparison to fulfill a search request. *Blex* [33], *ESH* [22], *Multi-MH* [103], and *Gitz* [23] all require linear time for querying (see Table 3.2). *BinGo* [17] is better as it uses a filter. However, the filter is still based on a pairwise comparison with linear

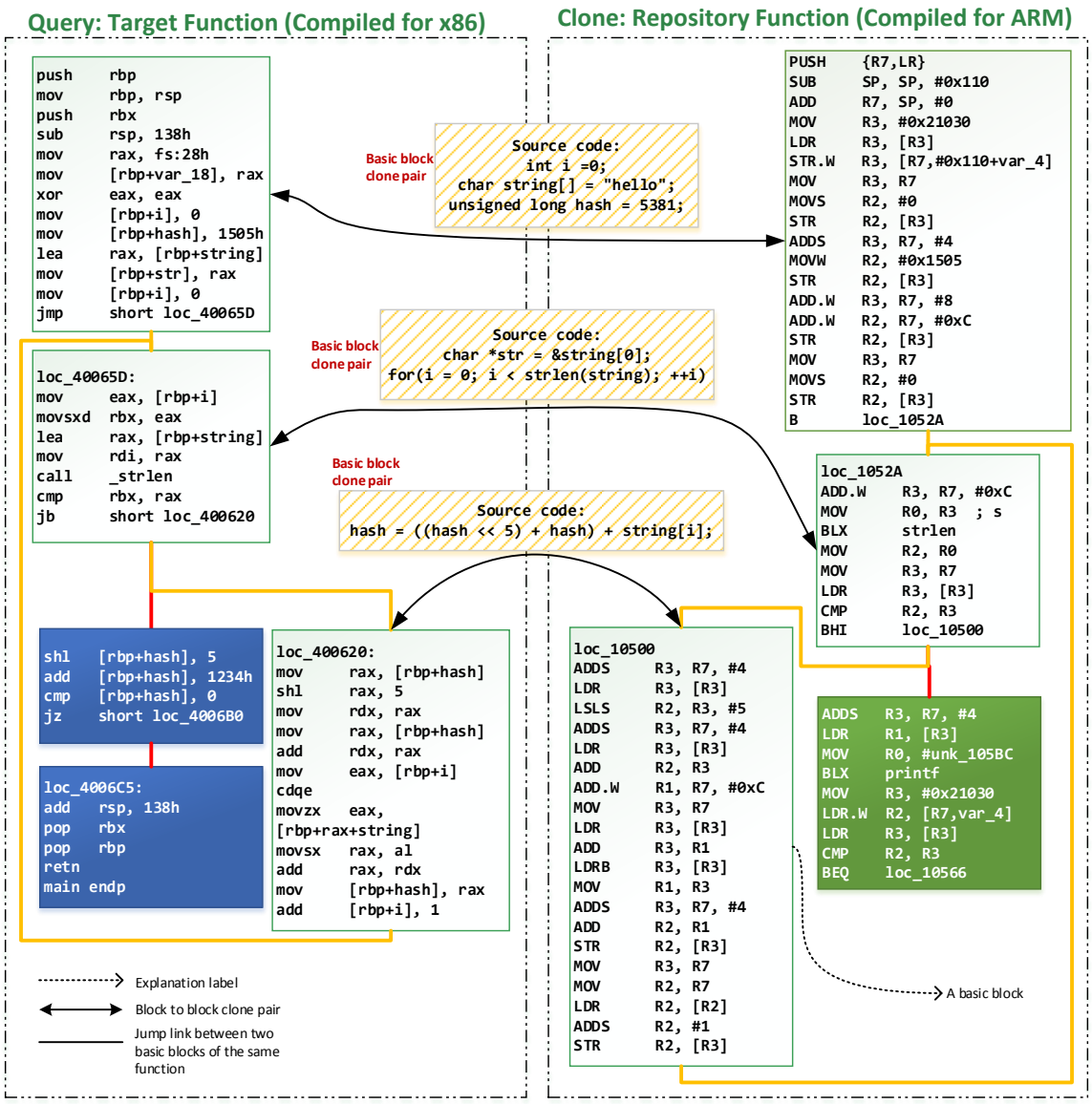

Figure 3.1: An example of a subgraph clone between two assembly functions. The code on the left is compiled for the x86 processor. The code on the right is compiled for the ARM processor. Basic blocks with a white background form a *subgraph clone*. The source code is unavailable when conducting a clone search.

complexity.

P2: Input value sampling: Existing dynamic approaches, such as *Blex* [33], *Multi-MH* [103], and *BinGo* [17], use random sampling to specify the input values for dynamic testing. Random sampling may not correctly discriminate two logics. Consider that one expression outputs 1 if $v! = 100$; otherwise, 0. Another expression outputs 1 if $v! = 20$; otherwise, 0. Given a widely used sampling range $[-1000, 1000]$ for value v, they have a high chance of being equivalent. A specialized sampling method is thus needed.

P3: Clone search granularity: An assembly function can be represented as a control flow graph consisting of connected basic blocks (see Figure 3.1). Given an assembly function as a query, most of the previous approaches only support searching at the function level. They are useful when there exists a function in the repository that shares a high degree of similarity with the query. However, due to the unpredictable effects of different obfuscation techniques or compilers' inlining behavior, one function may contain a copy of the other, or their basic blocks can be remixed. A search engine should be able to decompose the query into different known subgraphs in the repository. Refer to a subgraph clone example in Figure 3.1. *Multi-MH* [103] employs graph matching at the function level. *Genius* [39] encodes each function into a feature vector. *Discovre* [34] also computes structural similarity at the function level. However, none of them support partial clone search.

P4: External knowledge and incremental updates: Static approaches that support cross-architecture clone search include *Discovre* [34] and *Genius* [39]. They are scalable but require external ground-truth data to measure the weight for each feature. The experiment by Feng, Zhou, Xu, *et al.* [39] shows that the size of the external data has a significant impact on the clone search quality. The reported weights by Eschweiler, Yakdan, and Gerhards-Padilla [34] do not perform well in the experiment presented by Feng, Zhou, Xu, *et al.* [39], which also shows the importance of external data. As the repository size grows, one will need to learn new weights using more external data and re-index all existing data periodically. However, there are no clear guidelines for choosing external data.

To address the above problems, we proposed and implemented a new open source assembly

Function	S-expression
deflate_stored	(32to64 (Add (64to32 r_rbp_v0) m_0_v0))
flush_pending	(Add m_1_v0 (Or r_v0_v0 c_0:0x0))
longest_match (ARM)	(Sub (Add m_10_v0 (Add m_11_v0 0x102)) 0x102)
longest_match (MIPS)	(Add (Add r_v0_v0 (Add r_t5_v0 0x102)) 0xfffffefe)

Table 3.1: Similar symbolic expressions retrieved by our index structure. They all add two variables but in different forms. The word *to* indicates a type conversion operation.

	Category	Partial Clones	Query	Index	External Data	Architectures
Sym1n0	**dynamic**	**subgraphs**	**O(log(n))**	**O(n $\times$ log(n))**	**No**	**8+**
BinGo [17]	dynamic	N/A	n	N/A	No	3
Multi-MH [103]	dynamic	N/A	n	O(800 $\times$ n)	No	3
Blex [33]	dynamic	N/A	n	N/A	No	2
ESH [22]	dynamic	N/A	n	N/A	No	2
Discovre [34]	static	N/A	O(log(n))	O(n $\times$ log(n))	Yes	3
Genius [39]	static	N/A	O(log(n))	O(n $\times$ log(n))	Yes	3
Gitz [23]	static	N/A	n	N/A	Yes	8+

Table 3.2: State-of-the-art binary clone search techniques that support multiple processor architectures. n denotes the repository size.

clone search engine[1] based on symbolic expression retrieval. The whole engine is implemented using a distributed framework. Our major contributions can be summarized as follows:

- **A practical solution for cross-architecture subgraph clone search:** We propose a new cross-architecture search engine based on symbolic expression retrieval. It addresses the aforementioned issues of the existing works. It is the first search engine that supports cross-architecture subgraph clone search. It is efficient and scalable to millions of assembly functions. It does not rely on any external data and supports incremental updates. Table 3.2 summarizes its features.

[1] Sym1n0 is part of the Kam1n0 open source project:
https://github.com/McGill-DMaS/Kam1n0-Plugin-IDA-Pro.
On-line Demonstration:
http://dmas.lab.mcgill.ca/projects/kam1n0.htm. User name: jedi Password: starwarz

- **Scalable tree-based index for retrieving semantically similar symbolic expressions:** We propose a novel tree-based indexing structure for symbolic expressions retrieval. The tree structure explores the differential I/O behavior of symbolic expressions and is able to efficiently retrieve the top-K semantically similar expressions. Table 3.1 shows examples. The similarity is defined as the approximated overlapping space with respect to their I/O behavior. The tree structure is scalable and can be incrementally updated.
- **A fast input sampling algorithm to support efficient index node splitting:** We use random I/O testing to model the similarity between different symbolic expressions. Instead of doing a complete random input sampling, we propose a new algorithm to utilize the information of the symbolic expressions to restrain the sampling space, which speeds up the process of finding a good split for the tree nodes at the indexing stage.
- **Efficient semantic subgraph search:** To support subgraph clone search, we incorporate our symbolic expression retrieval index with a MapReduce-based subgraph search algorithm to support subgraph clone detection. The original algorithm proposed in [30] uses a locality sensitive hashing (LSH) index to find *syntactic* subgraph clones with sub-linear complexity in practice. We replace this with our own symbolic expression index and customize the scoring functions for the reducing phase in order to support *semantic* subgraph clone search. By combining the semantic and subgraph search, we boost the cross-architecture clone search accuracy.

The remainder of this chapter is organized as follows. Section 3.1 positions our study within the literature of two related research problems. Section 3.2 formally defines the clone search problem. Section 3.3 provides an overview of our solution and system design. Section 3.4 presents the preprocessing steps to construct the symbolic expressions, namely S-expressions, from the assembly language. Section 3.5 introduces the proposed tree-based symbolic expression retrieval index and sampling algorithm. Section 3.6 describes how we incorporate the S-expression retrieval with subgraph search algorithm to solve the assembly subgraph clone search problem. Section 3.7 presents our benchmark experiments. Section 3.8 discusses Sym1n0's limitations. Section 3.9 concludes this paper.

3.1 Related Work

Static approaches such as *k-gram* [94], *LSH-S* [114], n-gram [69], *BinClone* [36], and *Kam1n0* [30] model assembly code as independent operations and categorized operands. They represent an assembly function as descriptive statistics over the selected tokens. *BinSequence* [56] and *Tracelet* [24] model assembly code as the editing distance between instruction sequences. *TEDEM* [104] statically compares basic blocks by their expression trees. These approaches are oblivious to instruction changes in a cross-architecture setting. *BinDiff* [31] and *BinSlayer* [12] rely on control flow graph (CFG) matching. However, the CFG of a function between two architectures can be different. *ILine* [59], *Discovre* [34], *Genius* [39], *BinSign* [97], and *BinShape* [123] construct descriptive statistic features such as the ratio of arithmetic assembly instructions, the ratio of transfer instructions, the number of basic blocks, and the number of function calls, among others. Most of these features are architecture-agnostic. All these techniques support a particular family of assembly language, except *Discovre* [34] and *Genius* [39]. *Discovre* and *Genius* approach the clone search problem as vector nearest neighbor search. They are scalable to millions of assembly functions. However, both of them rely on a good selection of external data to learn feature weights. The external data consists of known assembly clone pairs. They also do not support partial clones. Moreover, *Genius* uses locality sensitive hashing (LSH) with random hyperplane hashing to promote its scalability. The adopted algorithm requires the size of the repository to determine the number of hyperplanes in order to guarantee the induced false positives. Therefore, it cannot be incrementally updated. Both *Discovre* and *Genius* support ARM, MIPS, and x86. *Gitz* [23] is another static approach that works on the normalized IR level. It is based on VEX and supports more than 8 architectures. However, it relies on a pairwise comparison for searching, which requires a linear search time and is not scalable.

Dynamic methods measure semantic similarity by dynamically analyzing the behavior of the target assembly code. *BinHunt* [43], *iBinHunt* [90], and *ESH* [22] use a theorem prover to verify whether two basic blocks or strands are equivalent. Conducting a pair-wise symbolic verification over the whole repository is not feasible in practice, given that the size of the repository can exceeds millions. It requires at least a linear scan. Jiang et al. [62], *Blex* [33], *Multi-MH* [103], and *BinGo* [17] use randomly sampled input values to compare I/O values. As mentioned in the be-

ginning of this chapter, random sampling cannot discriminate between the semantics of condition-related symbolic expressions. *ESH* supports x86 and AMD64. *BinGo* relies on some architecture-agnostic knowledge to construct a filter. It currently supports x86, AMD64, and ARM. *Multi-MH* supports x86, ARM, and MIPS. *Multi-MH* relies on a pair-wise comparison searching process. *BinGo*'s filter also requires a linear scan over the repository. Both of them are not scalable as the repository size grows. *Sym1n0* is a dynamic approach. Instead of using random sampling, it narrows down the input search space to discriminate between symbolic expressions. The searching process corresponds to an incrementally constructed tree structure that provides a sub-linear query complexity. It supports more than 8 architectures.

Source code clone search is another related area. *CCFINDERX* [65] and *CP-Miner* [83] use lexical tokens as a feature to find source code clones. Baxter et al. [10] and *Deckard* [61] leverage abstract syntax tree for clone detection. *ReDebug* [58] is another large-scale source code clone search engine. Recently, deep learning has been applied to this problem [138]. These techniques are not applicable when the source code is unavailable. Sym1n0 focuses on assembly code clone search.

This paper is also broadly related to the *Mathematical Information Retrieval (MIR)* literature because it depends on efficient and accurate symbolic expressions retrieval. In the MIR related problems, math notations and symbols are usually recognized from either a piece of text or an image. After, the math notations and symbols are aligned into a *Symbol Layout Tree*. Then, they are interpreted as an *Operator Tree* [146], which is similar to a symbolic expression. Finally, the query is processed by using fuzzy matching on the tree structure and symbol names. Our retrieval problem is related but different from the MIR retrieval problem. We have no information about the correct mapping of input symbols between symbolic expressions because we do not know the symbol's correct name or other text-based keywords by Schubotz, Grigorev, Leich, *et al.* [119].

3.2 Problem Statement

As mentioned earlier, reverse engineers mostly work with binary files. After being unpacked and disassembled, a binary file becomes a list of assembly functions in a given assembly language. There are various families of assembly language depending on the processor architecture.

3.2 Problem Statement

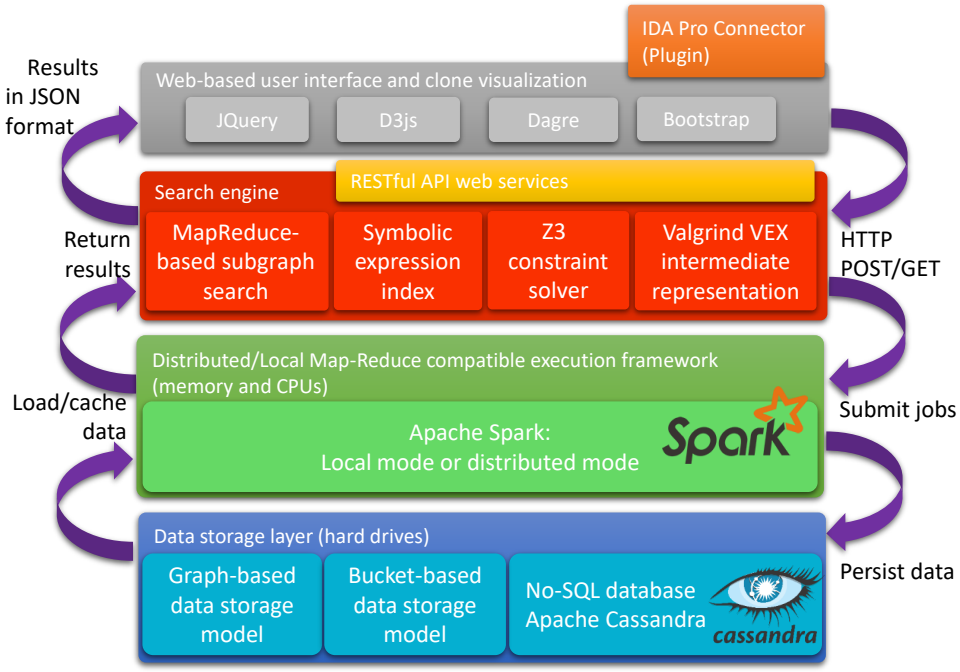

Figure 3.2: The overall solution stack of the Sym1n0 search engine.

In this chapter, we use the following notions: *function* denotes an assembly function; *block* represents a basic block; *source function* represents the original function written in source code, such as C++; *repository function* stands for the assembly function that is indexed inside the repository; *target function* denotes the assembly function that is given as a query; and correspondingly, *repository blocks* and *target blocks* refer to their respective basic blocks. We adopt the problem definition used in [30]. Each function f is represented as a control flow graph denoted by (B, E), where B indicates its basic blocks and E, the edges that connect the blocks. Let $B(\mathrm{RP})$ be the complete set of basic blocks in the repository and $F(\mathrm{RP})$, the complete set of functions in the repository. Given an assembly function, our goal is to search all its subgraph clones inside the repository RP. We formally define the search problem as follows:

Definition 7 *(Assembly function subgraph clone search) Given a target function f_t and its control flow graph (B_t, E_t), the search problem is to retrieve all the repository functions $f_s \in \mathrm{RP}$ that share at least one subgraph clone with f_t. The shared list of subgraph clones between f_s and f_t is denoted by $sg_s[1 : a]$, where $sg_s[a]$ represents one of them. A subgraph clone is a set of*

basic block clone pairs $sg_s[a] = \{\langle b_t, b_s\rangle, \dots\}$ *between* f_s *and* f_t*, where* $b_t \in B_t$*,* $b_s \in B_s$*, and* $\langle b_t, b_s\rangle$ *is a semantic clone (see Figure 3.1). Formally, given* f_t*, the problem is to retrieve the top-*K $\{f_s | f_s \in \mathrm{RP}$ *and* $|sg_s| > 0\}$ *with respect to a given similarity measure.* □

Additionally, we define that a basic block b can be represented as a list of symbolic expressions, denoted by $SE(b) = se[1:n]$, where se_n represents one of them. A symbolic expression can have multiple inputs $IN(se) = in[1:m]$ but only one output $OUT(se)$. The expression itself is a function $se : \mathbb{R}^{|IN(se)|} \to \mathbb{R}$. Therefore, a basic block has $|SE(b)|$ unique outputs and $|SE(b)|$ symbolic expressions. A basic block's symbolic expressions may or may not share the same set of input variables. For each expression se, we trace back from its output variable $OUT(se)$ and only include the involved input variables in $IN(se)$ A basic block should have at least one symbolic expression $|SE(b)| > 0$.

3.3 Overall Architecture

Our implementation of the proposed engine is built upon the open source clone search framework proposed by Ding, Fung, and Charland [30]. It uses the distributed column family database *Apache Cassandra*[2] as the storage and the distributed computational framework *Apache Spark*[3] as the computational layer. Figure 3.2 shows the solution stack. Our choice of the column-family database will be further justified in Section 3.7. On top of these two layers is our proposed engine. It mainly contains four modules. Above the search engine, we build a RESTful API, a web interface, and a *Hex-Rays IDA Pro* plugin[4] as clients. IDA Pro is a popular interactive disassembler used by reverse engineers.

Figure 3.3 describes the data flow of the proposed clone search process. It is similar to the one proposed by Ding, Fung, and Charland [30]. The original process only handles a single type of assembly language, but the present one operates at the symbolic expression level, which is architecture-agnostic. This process consists of four stages.

The search process starts with a binary file. We first extract the target assembly functions either

[2]Apache Cassandra Database: http://cassandra.apache.org/
[3]Apache Spark: http://spark.apache.org/
[4]IDA Pro: http://www.hex-rays.com/

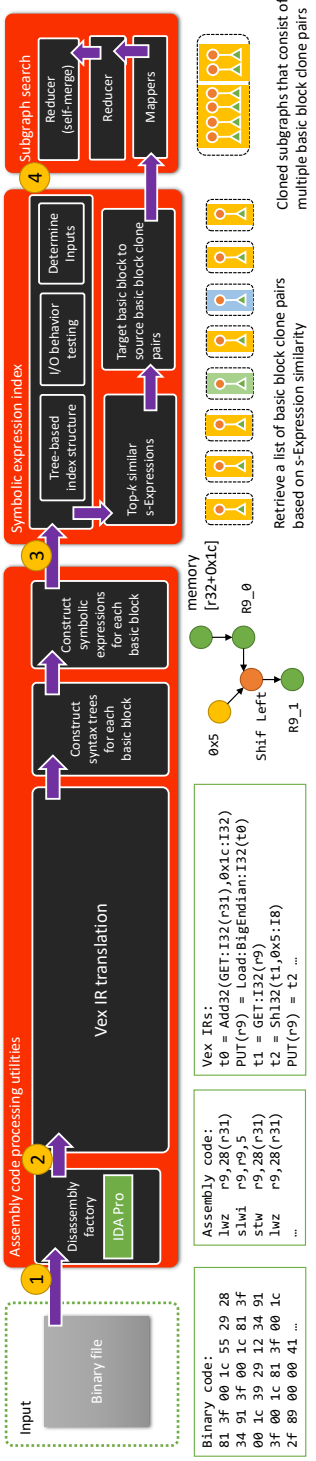

Figure 3.3: The four stages of assembly clone search data flow.

by disassembling the binary file or parsing the raw assembly code from the query. After having obtained a list of target assembly functions and their corresponding bytes, we translate each of their basic blocks into the Vex *Intermediate Representation (IR)* using the Valgrind instrumentation framework [96]. We created our own Java interpreter for the Vex IR.

An intermediate representation captures the complete semantics of the assembly code and is architecture-agnostic. Figure 3.3 shows a translation example. One assembly instruction is translated into multiple Vex IR expressions. It contains many temporary values that assist with the translation. We thus need to construct a syntax tree according to the Vex IR and simplify the tree structure before the symbolic expression construction. Performing clone search directly on the IR expressions cannot solve the cross-architecture clone search problem. The IR expressions are generated by translating the assembly instructions one by one. Given two instructions of different families of assembly language that have similar logic, the generated IR expressions are still very different due to the fact that different processors have different registers, flag bits, and flag calculation logics. Therefore, we need to conduct clone search at the symbolic expression level.

After having obtained the symbolic expressions of a target block, we retrieve the top-K most similar S-expressions in the repository according to their corresponding location in the index tree. We collect their corresponding basic blocks and combine them with the target block to generate a list of basic block clone pairs. Similarity values are calculated based on the overlaps of the symbolic expressions. More details are provided in Section 3.5. In the last stage, we feed the list of basic block clone pairs to a MapReduce-based subgraph search algorithm and collect the subgraph clones from the repository.

3.4 Constructing the Syntax Tree

In this section, we discuss on how we construct the syntax graph generated from the Vex IR expressions. The generated Vex IR statements are still noisy. For example, some register variables output a specific address of the binary file for the purpose of flow control or calling a function. We need to preprocess the syntax tree before constructing the symbolic expressions.

Abstract all of the memory reference variables. First, we abstract all of the memory reference variables. A memory location in assembly language is identified by its address. However, the

address is usually not a constant but rather a relative variable. Figure 3.3 shows an example. The first two IR expressions load a value from a memory location that is relative to the runtime value of the register r32. In order to track the relationship between memory variables, we keep track of each memory location access and construct the symbolic expression for its address. If two memory locations have the same symbolic expression as address, we consider them to be the same variable.

Control the versions of each register and memory variable. A register or a memory variable may have been accessed multiple times in the IR expressions, as shown in Figure 3.3. Register r9 is initially written with a memory variable. Later, it is rewritten with the result of a shift left operation. We found that the register has been written or read before. Therefore, we create a new version of this register (r9_1 in the figure). Only the latest version of the variable will be used in subsequent reads.

Select the output nodes. To construct symbolic expressions for a basic block, we need to select the set of output nodes. We follow three rules to include variables in the syntax tree as output nodes. *Rule 1*: Any constant node that is not an address. *Rule 2*: Any memory variable node that has no children. *Rule 3*: Any latest version of the general registers, floating point registers, stack registers, and instruction pointer register. We construct symbolic expressions for all of these nodes. At the time of I/O testing, when a symbolic expression outputs an address value and the value is the address of the next sequential basic block address, we change it to a constant $NEXT$, otherwise to the constant $SKIP$.

3.5 S-Expression Retrieval

In this section, we introduce our proposed symbolic expression indexing tree structure and retrieval algorithm. This section corresponds to Stage 3 in Figure 3.3. The task is to index and retrieve semantically similar symbolic expressions. Table 3.1 shows some of the expressions found by our proposed tree-based index in the experiments. They are visually and syntactically different, but semantically equivalent. Retrieving these S-expressions is challenging since all of the visually related information is unavailable and they use a different set of operations. We seek for a dynamic analysis approach to solve this problem.

Conducting dynamic analysis to measure semantic similarity of symbolic expressions is not

new. Pewny, Garmany, Gawlik, *et al.* [103] proposed to do a pair-wise comparison by randomly selecting inputs from a specific range to model their semantic similarity. However, as discussed in Section 3.1, their approach failed to correctly model the semantic similarity, due to the uniform sampling distribution. They assume that the semantic similarity is captured by the number of overlapping outputs when given a uniform sampling space. This assumption usually does not hold for assembly language because there are many different Boolean expressions, such as $i < 2$ and $i < 1$, which tend to be very similar to each other. David, Partush, and Yahav [22] estimated the similarity between different S-expressions using a constraint solver. It tries all the permutations of input mapping and asserts that the output values are equivalent. It is based on the assumption that semantic similarity is captured by the percentage of inputs that can find a correct mapping in the outputs. This assumption may not hold since some of the input variables do not contribute to the output. For example, $(x * (x + 1)\%2)$ always results in zero. Moreover, using a constraint solver with full input/output permutation does not scale well. It requires a pairwise comparison.

Based on the aforementioned related studies and our observations, we find it intrinsically hard to define what would be the perfect definition of semantic similarity for assembly code. Even mathematically inequivalent expressions can be a correct match in assembly code clone search. One example can be to search clones between 64-bit and 32-bit assembly instructions, where many variables do not have the same bit length.

Instead of defining the similarity metric and checking to what degree two symbolic expressions are similar to each other, we find it more practical to *differentiate* them against what is inside the repository, since the goal is to retrieve clones but not to prove clones. We start with the assumption that all the expressions in the repository are completely equivalent. By taking this assumption, the expressions should follow several properties. We try to separate them by gradually tightening up the rules on these properties to see if they are still satisfied. We start by defining our Top-K S-expression retrieval problem. As mentioned earlier, a symbolic expression is denoted as se. It has only one output $OUT(se)$ and one or more inputs $IN(se) = in[1 : m]$, where $in[m]$ is one of them.

Definition 8 *(Symbolic Expression Retrieval) Given a query symbolic expression se_q, the task is to retrieve at most the top-K symbolic expressions in the repository SRP that satisfy a specific set*

of properties $\mathbb{P}$.

Our goal is not to retrieve all the equations that are guaranteed to be equivalent, but rather to retrieve the top-K equations that are potentially similar with respect to the set of properties $\mathbb{P}$. We consider the following two properties as our own $\mathbb{P}$ to capture semantic information:

Number of unique input values. Since we start with the assumption that all the expressions in the repository are equivalent, they should have the same I/O testing result irrespective of how many input values are the same. For example, given two expressions $se_0 = IN(se_0) \rightarrow OUT(se_0)$ and $se1 = IN(se_1) \rightarrow OUT(se_1)$, at the beginning they should produce the same output value with respect to a random value v such that $\forall in \in IN(se_0) \cup IN(se_1) \ : \ in = v \ , OUT(se_0) = OUT(se_1)$. We assume that all the input values are the same. In other words, there is only one unique value as the input. If we find that there are too many expressions that satisfy this property, we can constraint it further by increasing the number of unique input values. By increasing the number of unique input values, we take a stronger requirement that two expressions need to have some input value mapping in order to be equivalent. We denote this number as μ. When $|IN(se_0)| = \mu$, we arrive at the state where se_0 needs to be at a specific input permutation in order to be considered equal to other symbolic expressions at this level. This is because at this stage, we assume that all input values need to be different.

I/O test results sequence with narrowing input space. The other property is the specific sequence of I/O testing. We begin with the assumption that all the expressions are equivalent. Therefore, all the expressions should have the same output value even if they are tested only once. If we find too many expressions having the same output value after the first I/O test, we narrow down the input space and move to test the next input value. We try to split them according to their different output behaviors. As we narrow the input space and take one additional test, the probability that two inequivalent expressions are producing the same output decreases.

3.5.1 A Scalable Tree-based Index

In this section, we describe our tree-based index for symbolic expression retrieval. Our indexing structure consists of multiple levels of nodes. Each node in the tree can have multiple children, but they can have only one parent. It starts with a root node, assuming that all the expressions are

3.5 S-Expression Retrieval

Algorithm 3 Locate Expression Bucket

function LOCATESTART(Symbolic Expression se)
  $buckets \leftarrow []$
  $newInputs \leftarrow [0...] \in \mathbb{R}^{|IN(se)|}$
  ▷ Initialize a zero vector.
  $newOutput \leftarrow se(newInputs)$
  ▷ Evaluate the output of the se
  $child \leftarrow CH_{root}(newOutput)$
  if $child$ exists **then**
    $INc(se, child) \leftarrow newInputs$
    ▷ Prepare input vectors for child
    $OUTc(se, child) \leftarrow newOutput$
    $buckets \leftarrow buckets \cup Locate(se, child)$
    ▷ Lookup in the child bucket.
  end if
  return $buckets$
end function

---

function LOCATE(Symbolic Expression se, Bucket eb)
  if $|CH_{eb}| < 0$ **then**
  ▷ This bucket has no children. We return this leaf node.
    return eb
  end if
  $buckets \leftarrow []$
  $oldInputs \leftarrow INc(se, eb)$
  ▷ Collect the old inputs used in parent bucket.
  for $i \rightarrow |oldInputs|$ **do**
  ▷ For each value in the old inputs.
    if $oldInputs[i] = OldVal(eb)$ **then**
      $newInputs \leftarrow$ Copy($oldInputs$)
      ▷ Replace one of the matched value.
      $newInputs[i] \leftarrow NewVal(eb)$
      ▷ Generate a new input.
      $newOutput \leftarrow se(newInputs)$
      $child \leftarrow CH_{eb}(newOutput)$
      ▷ Check if there is a child bucket.
      if $child$ exists **then**
        $INc(se, child) \leftarrow newInputs$
        $OUTc(se, child) \leftarrow newOutput$
        $buckets \leftarrow buckets \cup Locate(se, child)$
        ▷ Lookup in the child bucket.
      else
      end if
    end if
  end for
  return $buckets$
  ▷ Return a list of leaf buckets that returned by its children.
end function

equivalent. We name the nodes in this tree as *Expression Bucket*. An expression bucket contains a list of symbolic expressions and specific information about the aforementioned two properties, in order to further differentiate S-expressions.

Definition 9 *(Expression Bucket) An expression bucket, denoted as* eb*, contains a set of symbolic expressions* $SEb(eb)$*. To arrive at this bucket, each symbolic expression needs to have concrete input values and a concrete output value. The concrete input value for an expression* se *on this bucket is* $INc(se, eb) = inc[1 : m]$*, where* $inc[m] \in \mathbb{R}$*, and its concrete output value is* $OUTc(se, eb) \in \mathbb{R}$*. At the root node, all the concrete values are initialized to zero. One can obtain two values according to the symbolic expressions in this bucket:* $NewVal(eb) \in \mathbb{R}$ *and* $OldVal(eb) \in \mathbb{R}$*. It is also a function that maps a real value to one of its children:* $CH_{eb} : \mathbb{R} \rightarrow ExpressionBucket$*.*

In general, the tree structure contains multiple levels and each level contains multiple expression buckets, as shown in Figure 3.4. Each expression bucket defines what will be the I/O testing output for the symbolic expressions arriving in this bucket with the specific inputs used in the parent node. Given a symbolic expression se_q, in order to query or index it against the index structure, we need to first locate its corresponding buckets in the tree. Algorithm 3 describes how we locate an expression starting from the root node. Figure 3.4 also shows an example.

Given a symbolic expression, its input vector is first initialized with zeros or other constants. After evaluating the output of the symbolic expression, se finds the child node's location and continues the lookup from that expression bucket. Each expression bucket defines two values. One is the old value to be replaced for the old input that was used in the parent bucket. The other is a new value to be replaced with the old value. Take expression bucket $eb4$ in Figure 3.4, for example. It defines an old value of 0 and a new value of 4. The symbolic expression se arrives at this bucket with input $[0, 1, 2]$, and the old value 0 is replaced with a new value 4 to produce a new input vector $[4, 1, 2]$. The new input is later evaluated to look up the child expression buckets at the next level. If the current expression bucket does not contain any children, then the query arrives at a leaf node in the tree index and the corresponding bucket will be returned. At the end of the search, we find a list of leaf buckets for each symbolic expression query. The old value in each bucket is determined by the majority values of the input vectors that resided in this bucket before splitting. The new value is determined by randomly sampling from a narrowing input space. These two values are generated

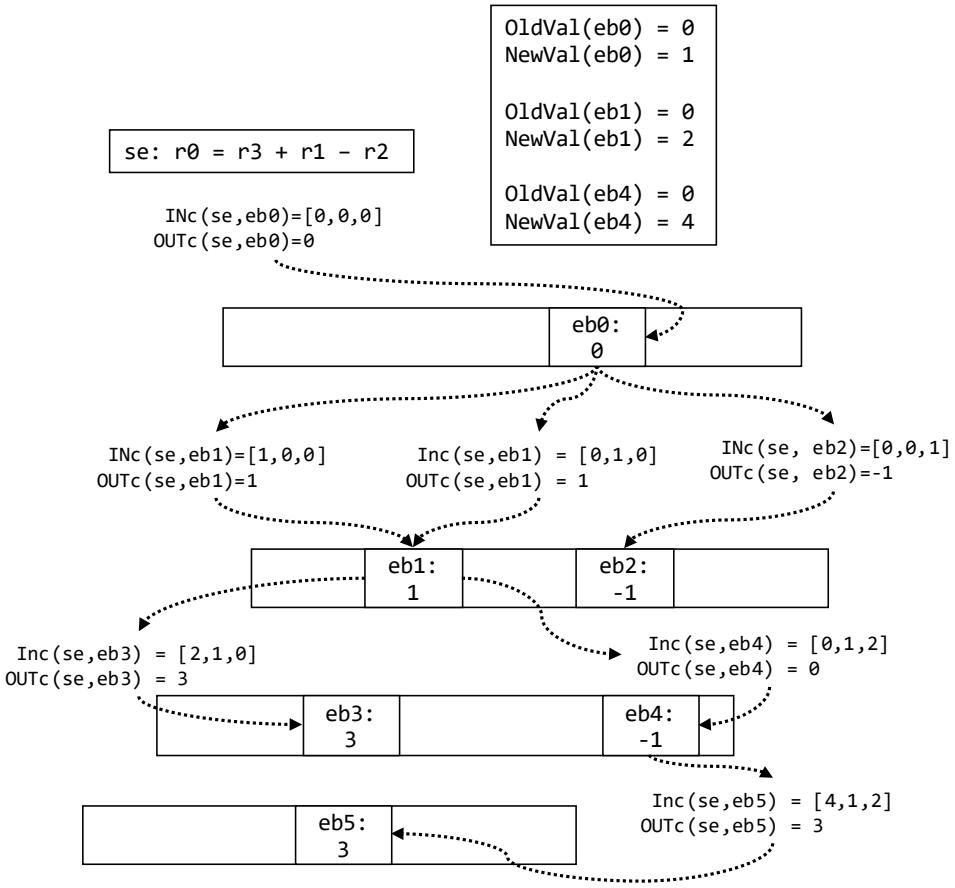

Figure 3.4: An example of locating the buckets for a symbolic expression. The S-expression is se.

at the time of indexing. After obtaining the expressions buckets, we retrieve all their symbolic expressions and rank them by the inverse number of expressions that are in the same bucket.

This tree-based index is scalable. It avoids a full pairwise comparison to retrieve symbolic expressions. Unlike the methods proposed by David, Partush, and Yahav [22] and Pewny, Garmany, Gawlik, *et al.* [103] that both require a full input permutation to compare every pair of symbolic expressions, this tree takes in the worst case only one full permutation. In the subsequent I/O testing, only a specific input value is replaced with a new value. Thus, it does not require a full permutation anymore in subsequent testing. For example, for any expressions with 3 input variables arriving at bucket eb4, we already find a correct mapping between their inputs, since we assume their inputs need to be all different. Thus, in the next bucket, we do not need to permute the input anymore. We only need to replace an old value with a new value. In this way, it takes less samples for I/O testing by reducing the number of permutations.

3.5.2 Input Range Sampling

To index new symbolic expressions, we first locate their corresponding leaf expression buckets in the tree. If the tree is empty, the returned bucket will be the root node. After, we add each new symbolic expression into its corresponding buckets. We track all of the affected buckets and split the ones that have more than K symbolic expressions. Remember that our problem is to retrieve the top-K similar expressions from the repository. If the tree structure is empty, all of the symbolic expressions are added to the root node and the bucket splitting process will start from there. Algorithm 4 shows the general procedure that recursively splits the bucket. The algorithm retrieves all of the symbolic expressions of that bucket and starts splitting these expressions by sampling an old value and a new value mentioned above.

The old value is determined by the majority of old input values from all of the involved symbolic expressions. By doing this, we try to maximize the impact of changing one of the input values to a new one. If we do not know the S-expressions' right mapping of inputs, the majority value will be the one that assumes some of the inputs are the same. For example, the bucket eb0 in Figure 3.4 has a majority value of 0, thus we pick 0 as the old value to be replaced. This way, we increase the number of unique input values by 1. Choosing a majority value will guarantee a good coverage.

Algorithm 4 Split Expression Bucket

function SPLIT(Expression Bucket eb)
    $NewVal(eb) \leftarrow SampleNewVal(SEb(eb))$
    ▷ Sample a new value.
    $OldVal(eb) \leftarrow SampleOldVal(SEb(eb))$
    ▷ Sample an old value.
    for each $se \in SEb(eb)$ **do**
        $oldInputs \leftarrow INc(se, eb)$
        for $i \rightarrow \in \mathbb{R}^{|IN(se)|}$ **do**
            if $oldInputs[i] = OldVal(eb)$ **then**
                $newInputs \leftarrow$ Copy($oldInputs$)
                ▷ Generate new input vector.
                $newInputs[i] \leftarrow NewVal(eb)$
                $newOutput \leftarrow se(newInputs)$
                ▷ Evaluate the output of expression.
                $child \leftarrow CH_{eb}(newOutput)$
                ▷ Check if we already created a child node.
                if $child$ not exists **then**
                    $child \leftarrow CreateNewBucket()$
                    $CH_{eb} \leftarrow CH_{eb} \cup child$
                end if
                $INc(se, child) \leftarrow newInputs$
                $OUTc(se, child) \leftarrow newOutput$
                $SEb(child) \leftarrow SEb(child) \cup se$
                ▷ Pass this expression to child node.
            end if
        end for
    end for
    $SEb(eb) \leftarrow []$
    ▷ Clear the expression of this node.
    for each $child \in CH_{eb}$ **do**
        if $|SEb(child)| > K$ **then**
            $Split(child)$
            ▷ Recursive split.
        end if
    end for
end function

3.5 S-Expression Retrieval

Instead of directly looking for the counts of each input vector, we concatenate all of the input vectors and generate a random number as an array index to sample the old value. The probability that an old number is chosen is thus proportional to its normalized counts. This approach prevents us from always sampling the majority value. Sometimes, the majority value does not affect the output values or is not able to split the bucket.

Algorithm 5 Sampling New Values

```
1:  function SPLIT(Expression Bucket eb)
2:      values ← []
3:      for each se ∈ SEb(eb) do
4:          for each constant ∈ se do
            ▷ Extract condition-related constants.
5:              if constant is-parent-of Conditions then
6:                  values ← values ∪ constant
7:              end if
8:          end for
9:      end for
10:     rand ← Uniform(0, |values|)
        ▷ Random number of range [0, |values|).
11:     value ← values[rand]
12:     candidates ← [1 − value, !value, 0 − value, value]
13:     rand ← Uniform(0, 3)
14:     return candidates[rand]
        ▷ Return one of the candidates based on even distribution.
15: end function
```

In fact, the bucket splitting algorithm does not always get an old and new value that can split the bucket. We repeatedly call the Algorithm 4 for 10 times until it finds a split point. We find that the majority value usually works well for the old value. The problem mostly comes from the sampled new value. It is the same issue that Pewny, Garmany, Gawlik, *et al.* [103] suffers from: The sampling range is not good to distinguish the symbolic expressions. To handle this issue, we propose to use the available information in the symbolic expressions to determine the new value. The sampling method is presented in Algorithm 5. It looks for condition-related constants and samples them according to their normalized frequency. Getting the value is not enough; in order to cover all the condition operations including *Greater Than*, *Greater or Equal*, *Lower Than*, *Lower Than or Equal*, and *Not*. We further transform the sampled value to different forms and

resample them again. We found that this approach can greatly reduce the number of bucket split trials. It reduces the average number of trials from 20 to approximately 3. If by repeating a fixed number of times Algorithm 4 still cannot find a split point, we ask the solver to check if such a new value exists that can make the output different by using the equation below with assertion $sum \in (0, |SEb(eb)|)$. Since it only involves one unknown variable, the solver can quickly return an answer. $[\![\cdot]\!]$ denotes an identity function that outputs 1 if the expression is true; otherwise, 0.

$$sum = \sum_{se}^{|SEb(eb)|} \Big[\!\Big[se(\text{newValSymbol})! = OUTc(se, eb) \Big]\!\Big] \tag{3.1}$$

Sampling new values locally based on what is inside the bucket is essentially narrowing down the search space. As the tree goes deeper, the number of symbolic expressions that pass through a bucket is also decreasing. Sampling locally for each expression bucket can help the splitting algorithm find locally good splitting points. The index structure, as well as the index procedure itself, are incremental. We start with an index tree that only has a root node. We update the index by splitting the large expression bucket. Therefore, this index structure can be incrementally updated.

3.5.3 Implementation Details and Complexity Analysis

The tree structure in its nature resembles the column-family database schema. In a column-family database, each row is indexed with a primary key. One can add multiple secondary keys and corresponding values within each single row. We implement the tree structure as a column-family table. Each expression bucket of the index tree corresponds to a single row in the column-family database. Each row stores the primary keys for its children and other associated information as secondary keys.. Table 3.5 shows the basic schema of the implemented index tree.

For a symbolic expression se with $|IN(se)|$ number of inputs, the number of located leave buckets is bounded by $O(!|IN(se)|)$, which is the worst case where we need the full permutation of the input values. However, this case rarely happens as we only split a bucket when its size exceeds a pre-specified threshold K. On the first several levels, only a few permutations can be further developed into the deeper levels of the tree. Further permutations under a given permutation are only needed when we split a bucket. Moreover, $|IN(se)|$ is mostly bounded by 5 for a basic block,

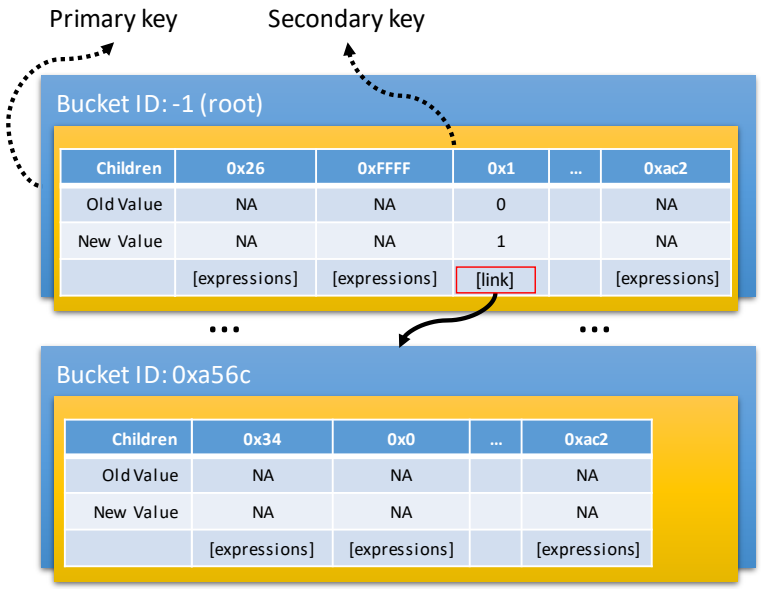

Figure 3.5: The symbolic index tree implemented with a column-family model. The index tree does not require slice query support. Column-family model is preferred over other disk-based tree index since each parent node can have a large number of children.

and its majority is between 2 and 3. Thus, we can afford to replace $O(!|IN(se)|)$ with a small constant upper bound $O(d)$. It is noted that previous methods by Pewny, Garmany, Gawlik, *et al.* [103] and David, Partush, and Yahav [22] both require full permutations.

Given N symbolic expressions in an index tree, the total number of their corresponding locations is bounded by $O(Nd)$. With a maximum of K symbolic expressions in a bucket, the total number of leaf nodes is bounded by $O(Nd/K)$. The index tree does not limit the number of children, which is reasonable in a column-family database like Cassandra, where one can have billions of columns for a single row. In the best case, all of the leaf nodes are on the same level under the root node. We only need $O(log(Nd/K))$ to look up the expression bucket in a single row. In the worse case, one parent node only has two children and the resulting tree resembles a binary tree. This binary tree has a depth of $O(log(Nd/K) \cdot \lceil log(2) \rceil)$ on average and $O((Nd/K)/2 \cdot \lceil log(2) \rceil)$ in the worst case. The worst case rarely happens and a binary split of a bucket almost only occurs in the last several levels of the tree. Since d and K are in the similar scale in $[500, 100]$ and can cancel each other, it is feasible to omit them in the complexity. Therefore, the complexity of locating the bucket for either insertion or querying is sublinear on average, approximately $O(log(N))$. The other overheads for insertion and querying are constant and negligible when compared to the process of locating buckets. In Section 3.7.3, we empirically study the system's scalability with more than 200 million symbolic expressions. It shows that the indexing and querying complexity is sublinear in practice.

3.6 Subgraph Clone Search

In order to support assembly subgraph clone search, we incorporate the symbolic retrieval index with the MapReduce-based subgraph search algorithm by Ding, Fung, and Charland [30]. This step corresponds to Stage 4 in Figure 3.3. The original algorithm by Ding, Fung, and Charland [30] consists of two phases (see Figure 3.6). *The Mapping Phase:* The algorithm generates basic block clone pairs by using *Locality Sensitive Hashing (LSH)*. An adaptive LSH hashing algorithm for cosine space was proposed in variant to the *LSB-Forest* [129] and *SK-LSH* [84]. *The Reducing Phase:* It takes the generated basic block clone pairs in the mapping phase and tries to reduce the clone pairs to subgraphs by considering their connectivity. The second phase uses a customized

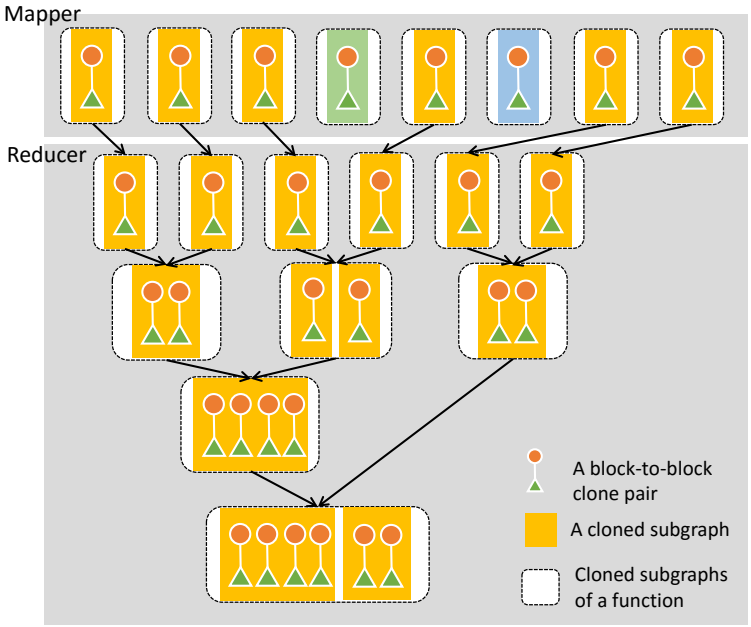

Figure 3.6: The MapReduce-based subgraph clone construction process for Sym1n0.

MapReduce-based algorithm instead of the typical exploration-based search algorithms such as *TurboISO* [53] and *STwig* [128]. We replace the original mapping phase by Ding, Fung, and Charland [30] with our own map algorithm to generate the basic block clone pairs. This way, we support semantic subgraph clone search. In our mapping phase, we derive the basic block clone pairs according to the symbolic retrieval results. The similarity between two basic blocks is modeled as the cumulative similarity values of the overlapped symbolic expressions:

$$\text{similarity}(b_t, b_s) = \sum_{se_t}^{SE(b_s) \cap SE(b_t)} \frac{1}{\text{bucketSize}(se_t)} \tag{3.2}$$

The function $bucketSize(\cdot)$ indicates the number of symbolic expressions that are co-located with se_t in the expressions buckets. This step takes the global repository information as a normalizer. Common S-expressions tend to have a higher number of co-located expressions and complex logics, such as encryption, and very few co-located expressions. Given a target function and its corresponding basic blocks, we retrieve their top-K similar repository basic blocks with similarity

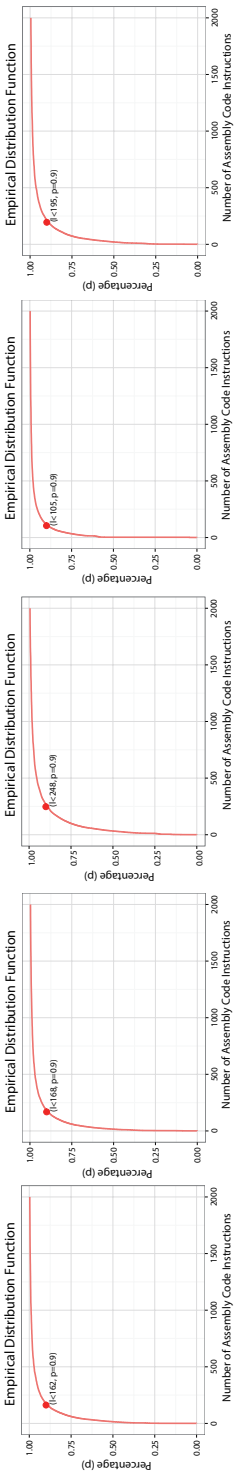

Figure 3.7: Empirical distribution of assembly function length (instructions count) for different processor architectures. From left to right, each diagram corresponds to x86 64-bit, ARM, MIPS 32-bit, PowerPC 32-bit, and x86 32-bit. There are 80,449 functions.

3.6 Subgraph Clone Search

	Index	x86 64-bit	ARM	x86 64-bit	ARM	x86 32-bit	x86 32-bit		
	Query	x86 32-bit	x86 32-bit	ARM	x86 64-bit	x86 64-bit	ARM	**Avg.**	p
BusyBox	BinGo†	Ø	Ø	Ø	Ø	Ø	Ø	0.324	Ø
	BinGo-SL†	**0.454**	0.282	0.319	0.451	0.447	0.440	0.399	◑
	Multi-MH†	Ø	Ø	Ø	Ø	Ø	0.324	0.324	Ø
	Constant	0.158	0.000	0.000	0.000	0.150	0.000	0.051	●
	n-gram	0.032	0.000	0.000	0.000	0.056	0.000	0.015	●
	n-perm	0.014	0.000	0.000	0.000	0.016	0.000	0.005	●
	Graphlet	0.363	0.329	0.290	0.339	0.371	0.290	0.330	●
	Graphlet-E	0.046	0.040	0.037	0.038	0.047	0.037	0.041	●
	Graphlet-C	0.044	0.000	0.000	0.000	0.044	0.000	0.015	●
	Sym1n0*	0.434	**0.598**	**0.386**	**0.490**	**0.518**	**0.536**	**0.494**	○
	Index	x86 64-bit	ARM	x86 64-bit	ARM	x86 32-bit	x86 32-bit		
	Query	x86 32-bit	x86 32-bit	ARM	x86 64-bit	x86 64-bit	ARM	**Avg.**	p
Coreutils	BinGo†	0.359	0.105	0.065	0.105	0.360	0.067	0.177	◑
	BinGo-SL†	0.548	0.186	0.395	0.217	0.457	0.313	0.353	◑
	Multi-MH†	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
	Constant	0.141	0.001	0.001	0.001	0.092	0.001	0.039	●
	n-gram	0.083	0.000	0.000	0.000	0.079	0.000	0.027	●
	n-perm	**0.795**	0.005	0.000	0.000	0.534	0.000	0.222	●
	Graphlet	0.216	0.167	0.178	0.161	0.174	0.155	0.175	●
	Graphlet-E	0.058	0.054	0.069	0.059	0.055	0.047	0.057	●
	Graphlet-C	0.085	0.000	0.000	0.000	0.080	0.000	0.028	●
	Sym1n0*	0.723	**0.831**	**0.561**	**0.598**	**0.723**	**0.672**	**0.685**	○

Table 3.3: Testing cross-architecture clone search on the general utility dataset. The ground truth data is a one to one assembly function mapping. The clone search results are evaluated using the Precision@1 metric, which in this case captures the ratio of assembly functions that are correctly matched at position 1. Entries with †are cited performance. ○, ◑, and ● respectively indicate $p > 0.05$, $p \leq 0.05$ and $p \leq 0.01$. Ø indicates unavailable data or test samples insufficient for a statistical test. **Sym1n0* is our proposed method.

value calculated using the above equation. At this point, we finish the mapping phase with a list of clone pairs available to the reducing phase. The original mapping phase assumes that every clone pair has an equal weight of 1. However, we have a similarity value generated by matching basic blocks' symbolic expressions. We modify the original subgraph scoring functions to incorporate this similarity measure as:

$$sim_n = \frac{\sum_{b_t}^{B_t} \text{MaxMatched}(b_t, sg_s)}{|E_t|} + \frac{|\text{uniqueEdges}(sg_s)|}{|B_t|}. \tag{3.3}$$

The MaxMatched function finds the highest score between b_t and $b_s \in sg_s$. After the reducing

phase, we collect a list of subgraph clones. To measure the similarity between functions, we take the subgraph coverage and aggregate their similarity value as the score.

3.7 Experiments

This section presents the experimental results for cross-architecture assembly clone search. We benchmark the performance of our proposed solutions with existing available ones that can be used for assembly clone search. The experiment consists of three parts. First, we use two open source utility libraries as our clone search dataset. Utility libraries are widely used in the firmware image for embedded devices such as routers. *BusyBox* and *Coreutils* have been used in recent research [17], [103] and we can thus directly compare our results with others. We also implement and include several applicable baselines by Khoo, Mycroft, and Anderson [69]. Second, we conduct cross-architecture clone search for 7 widely used libraries that are related to numeric calculation. Assembly code that involves numeric calculations such as encryption is the most difficult part to reverse engineer. We evaluate our proposed approach and the applicable baseline methods using several metrics used in information retrieval to cover most assembly clone search scenarios. Finally, we study the scalability and capacity of the proposed engine by presenting experimental results using a dataset that has millions of assembly functions and hundreds of millions of symbolic expressions.

3.7.1 Assembly Clone Search on Utility Libraries

In this experiment, we evaluate the proposed assembly clone search engine based on *BusyBox* (1.21.1) and *coreutils* utility libraries, which are widely used in firmwares. There were approximately 2,800 assembly functions extracted from the BusyBox binary file and around 2,700 in coreutils, depending on the target architecture. We follow the same experimental setup used by Chandramohan, Xue, Xu, *et al.* [17] and Pewny, Garmany, Gawlik, *et al.* [103]. In this section, we focus only on searching clones among the 2-combinations of the following assembly languages: x86 32-bit, x86 64-bit, and ARM 32-bit, since only these results are commonly reported by Chandramohan, Xue, Xu, *et al.* [17] and Pewny, Garmany, Gawlik, *et al.* [103]. We evaluate additional architectures in the following section.

Metric	Approach	lib-gmp	imagemagick	libcurl	libtomcrypt	openssl	lib-sqlite	zlib	Avg.	p
Precision@1	Graphlet-C	0.120	0.048	0.118	0.057	0.054	0.190	0.202	0.113	●
	Graphlet-E	0.337	**0.243**	0.328	0.265	**0.248**	0.369	0.450	0.320	●
	N-perm	0.118	0.037	0.094	0.060	0.047	0.228	**0.228**	0.116	●
	N-gram	0.096	0.031	0.066	0.061	0.038	0.176	0.188	0.094	●
	Constant	0.026	0.233	0.175	0.107	0.117	0.110	0.152	0.131	●
	Graphlet	0.230	0.203	0.285	0.237	0.177	0.358	0.448	0.277	●
	BinClone	0.178	0.047	0.121	0.072	Ø	0.152	0.218	0.131	◑
	Tracelet	0.068	Ø	0.045	0.085	Ø	0.069	0.152	0.084	●
	*Sym1n0	**0.356**	0.240	**0.408**	**0.324**	0.197	**0.461**	**0.474**	**0.349**	○
Recall@10	Graphlet-C	0.048	0.022	0.042	0.030	0.022	0.054	0.063	0.040	●
	Graphlet-E	0.266	0.194	0.231	0.253	0.180	0.262	0.372	0.251	●
	N-perm	0.055	0.014	0.039	0.035	0.022	0.072	0.081	0.045	●
	N-gram	0.049	0.011	0.031	0.031	0.018	0.060	0.069	0.039	●
	Constant	0.012	0.063	0.051	0.038	0.036	0.035	0.068	0.043	●
	Graphlet	0.243	0.203	0.268	0.267	0.186	0.341	0.444	0.279	●
	BinClone	0.078	0.027	0.063	0.046	Ø	0.067	0.105	0.064	◑
	Tracelet	0.044	Ø	0.046	0.058	Ø	0.060	0.098	0.061	◑
	*Sym1n0	**0.464**	**0.394**	**0.601**	**0.550**	**0.400**	**0.554**	**0.704**	**0.524**	○
MAP@10	Graphlet-C	0.019	0.013	0.033	0.018	0.016	0.049	0.052	0.029	●
	Graphlet-E	0.092	0.120	0.160	0.141	0.121	0.183	0.261	0.154	●
	N-perm	0.019	0.009	0.026	0.020	0.014	0.061	0.061	0.030	●
	N-gram	0.016	0.007	0.020	0.020	0.011	0.048	0.051	0.025	●
	Constant	0.004	0.057	0.046	0.029	0.031	0.030	0.043	0.034	●
	Graphlet	0.072	0.120	0.183	0.146	0.117	0.233	0.300	0.167	●
	BinClone	0.028	0.014	0.036	0.024	Ø	0.045	0.064	0.035	◑
	Tracelet	0.012	Ø	0.018	0.029	Ø	0.025	0.056	0.028	◑
	*Sym1n0	**0.140**	**0.167**	**0.341**	**0.239**	**0.144**	**0.328**	**0.372**	**0.247**	○

Table 3.4: Benchmark results of different assembly code clone search approaches. We employed three evaluation metrics: The *Precision at Position 1* (*Precision@1*), *Recall at Position 10* (*R@10*), and the *Mean Average Precision at Position 10* (*MAP@10*). The ground-truth data is a one to four mapping. ○, ◑, and ● respectively indicate $p > 0.05$, $p \leq 0.05$ and $p \leq 0.01$. Ø indicates unavailable data or test samples insufficient for a statistical test. **Sym1n0* is our proposed method.

3.7 Experiments

The experiment is conducted as follows. We first compile a selected library file using the GCC compiler for three different target architectures, which correspond to three different assembly language families. We link the assembly functions by using the compiler-output debug symbols and generate a one-to-one clone mapping between assembly functions. This mapping is used as the ground-truth data. It is noted that Ding, Fung, and Charland [30] match assembly functions by using the clones detected at the source code level, which is more practical. However, in this case, we follow the same setup as in [17] and [103] for comparison purposes. After obtaining the ground-truth mapping, we search a binary file compiled for one architecture against the one that is compiled for another architecture.

As mentioned in Section 3.1, there exist only a few techniques that support cross-architecture clone search [17], [34], [103], and [39], but none of their implementations are available for evaluation. To conduct a benchmark comparison, we seek to conduct experiments in a similar way. However, the experiments in [39] and [34] involve random sampling to generate the experiment dataset, which prevents us from reproducing it. Thus, we only consider the *BinGo* proposed by Chandramohan, Xue, Xu, *et al.* [17] and the *Multi-MH* proposed by Pewny, Garmany, Gawlik, *et al.* [103]. We also evaluate our implementation against other baseline methods proposed in [69] including n-*gram*, n-*perm*, *Graphlet*, *Graphlet-Extended*, and *Graphlet-Colored*. Some of these baselines depend on architecture-invariant features and should be able to handle cross-architecture clone search.

Table 3.3 shows the experimental results. It demonstrates that our proposed clone search engine in almost every case outperforms the others. The only exception is in the first case where it achieves a slightly lower performance than *BinGo*. On average for the *BusyBox* dataset, we can rank 50% of the functions at position one against more than 2,000 candidates, even if their assembly code appears to be very different. For the *coreutils* dataset, this value is 68%. It is noted that some of the results are unavailable as these cases were not covered in the corresponding paper. As expected, the architecture-agnostic approaches, such as n-gram and n-perm, do not perform well in cross-architecture detection. However, *Graphlet*, which is based on subgraph structure, achieves results comparable to *BinGo* for the *BusyBox* dataset. This experiment shows the robustness and quality of our proposed clone search engine.

3.7.2 Assembly Clone Search on Numeric Calculation Libraries

In this experiment, we cover a wider range of processor architectures. We include x86 32-bit, x86 64-bit, ARM 32-bit, MIPS 32-bit, and PowerPC 32-bit. We conduct experiments on several popular numeric calculation libraries, since numeric operations are the most complex and time-consuming ones for a reverse engineer. In this experiment, we select 7 open source numeric calculation libraries including *lib-gmp*, *imagemagick*, *libcurl*, *libtomcrypt*, *opessl*, and *zlib*. They cover utility functions for image processing, floating-point calculations, numeric manipulations, encryption, networking, and compression. Figure 3.7 shows the empirical distribution over the assembly functions. They are different with respect to function length. Especially, the MIPS architecture tends to have assembly functions that are longer than the others. This unbalance poses a challenge for a clone search engine.

	x86 64-bit	ARM	MIPS	PowerPC	x86 32-bit
lib-gmp	1,106	1,088	763	2,343	1,085
imagemagick	3,623	3,649	2,755	10,936	3,619
libcurl	957	987	932	1,388	968
libtomcrypt	1,015	1,035	673	1,972	1,021
openssl	5,765	5,780	5,857	6,224	5,772
sqlite	1,631	1,679	1,639	1,47	1,635
zlib	220	233	196	255	221

Table 3.5: Number of functions for binaries compiled for different architectures.

We respectively compile these libraries for each selected processor architecture. There are 35 binary files in total that represent 80,449 assembly functions. We link these assembly functions using the compiler-output debug symbols and generate the ground-truth data. It is noted that using source code information would be much better because there are internal clones within each library. However, for simplicity we assume that we only want to retrieve assembly functions that have the same debug symbol. This is a common practice used in the literature such as [34], [39], and [17]. After linking, we have a one-to-four assembly function clone mapping. We use this linking information as the ground-truth data to evaluate the robustness and search quality of our search engine.

	Average indexing time (ms)	Average query response time (ms)
BinClone	65.166	5,627.395
Tracelet	0	55,616.406
Graphlet-Extended	165.766	896.832
Sym1n0	15.368	605.247

Table 3.6: Average indexing time and query response time per assembly function for different baselines.

Given a library, we index binaries of all architectures and search each binary against the repository. Thus, the search repository contains not only the assembly code that is in other families of assembly language, but also in the same. This poses a challenge to the search process. The search engine may return false results that may be simply syntactically similar to the query. Three typical information retrieval metrics are chosen to evaluate the experiment: The *Precision at Position 1* (*Precision@1*), *Recall at Position 10* (*Recall@10*), and the *Mean Average Precision at Position 10* (*MAP@10*). These three metrics can evaluate the quality of the top-ranked list simulating a real user experience [86].

Since the other cross-architecture clone search approaches are not available to us for evaluation, we compare our proposed approach with *BinClone* [36], *Tracelet* [24], and the methods presented in [69]. Each method is limited to use a single core and 48 hours to find the clones. Ø indicates that the corresponding method cannot finish within the allotted time. We include the N-*gram*, N-*perm*, and *constant* approaches with a vector space model in the experiment. These numeric calculation libraries may include special constants or string literals that are useful for cross-architecture clone detection.

Table 3.4 shows the clone search results. Our proposed approach achieves the best results across different binaries. We notice that the *Precision@1* measure reported in this experiment is not as good as in the previous one. We look into the false negatives and find that there are in fact many internal clones within the library itself. For example, the *deflateStateCheck* function appears to be very similar to the *inflateStateCheck* function in the zlib library, as well as to the *BIO_asn1_get_prefix* and *BIO_asn1_get_suffix* functions in OpenSSL. The clone search engine actually returns all of them, but they are considered as false negatives by the ground-truth mapping. We also notice that the other baseline methods achieve reasonably good *Precision@1* measure, but

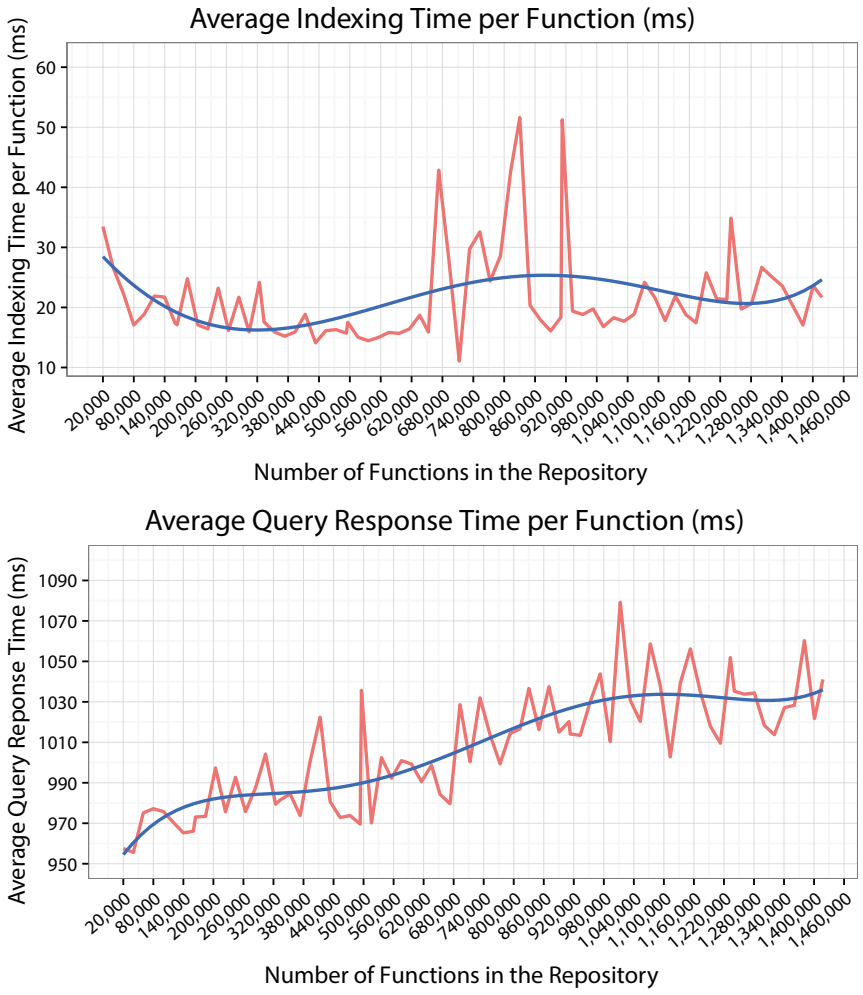

Figure 3.8: Scalability study. (a): Average Indexing Time vs. Number of Functions in the Repository. (b): Average Query Response Time vs. Number of Functions in the Repository. The red line represents the plotted time and the blue line represents the smoothed polynomial approximation.

have a much lower value for the *Recall@10* metric. We found that they are good at detecting clones between the x86 32-bit and x86 64-bit architecture, as they share similar instruction sets and syntax. Thus, they always return the clones from the x86 architecture family, but miss all the results for the other architectures, which leads to low *Recall@10* values.

Additionally, we compare Sym1n0 with other baselines with respect to the indexing and query response time in Table 3.6. It is noted that all the methods use memory-based data structures as data store. Therefore, the reported value for Sym1n0 in Table 3.6 is smaller than the value plotted in Figure 3.8. We include *BinClone*, *Tracelet*, and *Graphlet-Extended* in the table. *Graphlet-Extended* has the best clone search performance among the baselines in the previous experiment. However, it takes more time for indexing and searching, due to the process for *Graphlet* signature generation. *BinClone* is fast for indexing. However, for searching it requires a two-combination of all feature values, which leads to a high runtime. *Tracelet* does not require the indexing step. It relies on a pairwise comparison process to find clones, and this process also will incur a high runtime for searching. Other token-based baselines complete indexing and searching within 2 milliseconds per function but do not yield reasonable good performance. In general, Sym1n0 achieves a more balanced runtime with respect to indexing and searching.

3.7.3 Scalability Study

In this experiment, we evaluate the scalability of our proposed engine on a large collection of assembly functions. We conduct our experiment on a single workstation with an i7-4190 processor, 32 GB memory, and a 7200rmp non-SSD hard disk. The workstation runs on a *Windows Server 2012* operating system. The clone search engine runs on top of an embedded *Spark Computation Framework* and *Apache Cassandra Database*.

We prepare a large collection of binary files. All files are either open source libraries or applications, such as *Chromium* and *Opera*. The resulting uncompressed assembly code is more than 7 GB. In total, there are 1,417,710 assembly functions, 15,912,098 basic blocks, and more than 200 million symbolic expressions. On the workstation, we gradually index this collection of assembly functions in a random order and query the zlib version 1.2.7 binary file at every 20,000 assembly function indexing interval. As zlib is a widely used library in open source software, it is expected

that it has a large number of clones in the repository. We collect the average indexing time for each function, as well as the average time it takes to respond to a function clone search query. Figure 3.8 depicts the average indexing time and query response time for each assembly function.

On average, it takes 21 milliseconds to index a function, including the time to do a full recursive split of all the affected S-expression buckets. It takes 1,006 milliseconds to search a target assembly function against the repository. The indexing time is bounded by 52 milliseconds and the query response time by 1,100 milliseconds. After indexing more than 1 million assembly functions, the average query response time only increases from 957 milliseconds to 1,041 milliseconds, and the average indexing time remains almost the same. The small increase of the average querying and indexing time shows that the search engine is scalable. Figure 3.8 also includes the interpolated lines using polynomial approximation with an order of 5. The lines show that the increased complexity resembles a log-shape curve. There are spike-ups, due to the compaction routine in Cassandra, that increase I/O contention in the database. Our results are comparable to the experiments conducted by Ding, Fung, and Charland [30]. However, we use only one workstation, while Ding, Fung, and Charland [30] use a small cluster with 4 machines and does not support cross-architecture subgraph clone search.

3.8 Limitations

Sym1n0, at this moment, only simulates library function calls that are implemented as simplified symbolic graphs. It cannot simulate other internal or external function calls as it only simulates basic blocks but not the whole function. It also assumes basic block integrity, which can be attacked by code obfuscation techniques. Additionally, it is based on the correction identification of assembly function boundaries and CFG. However, our subgraph search algorithm can mitigate this issue to some degree. A function that is split into multiple functions can still be recovered as multiple subgraphs from the repository. An in-lined function can still be retrieved as a subgraph of the query.

3.9 Conclusion

In this study, we propose a robust, accurate, and cross-architecture assembly clone search engine. It supports cross-architecture search against various families of processors including, but not limited to, x86 32-bit, x86 64-bit, ARM32, ARM64, MIPS 32-bit, MIPS 64-bit, PowerPC 32-bit, and PowerPC 64-bit. We propose and implement a scalable tree-based symbolic expression index that is able to efficiently differentiate symbolic expressions' I/O behavior. The tree index can be incrementally updated. We incorporate the symbolic expression search index with a MapReduce-based subgraph search algorithm by replacing the original mapper phase with our new one. Our experiment demonstrates that our search engine performs better than the state-of-the-art clone search methods on average. In the future, we plan to improve the current tree-based index by optimizing the bucket split point and producing a more balanced tree structure to further improve efficiency.

4

Robust Clone Search

Developing a clone search solution requires a robust vector representation of assembly code, by which one can measure the similarity between a query and the indexed functions. Based on manually engineered features, relevant studies can be categorized into static or dynamic approaches. Dynamic approaches model the semantic similarity by dynamically analyzing the I/O behavior of assembly code [17], [33], [55], [103]. Static approaches model the similarity between assembly code by looking for their static differences with respect to the syntax or descriptive statistics [22], [24], [30], [34], [36], [39], [69], [114]. Static approaches are more scalable and provide better coverage than the dynamic approaches. Dynamic approaches are more robust against changes in syntax but less scalable. We identify two problems that can be mitigated to boost the semantic richness and robustness of static features. We show that by considering these two factors, a static approach can even achieve better performance than the state-of-the-art dynamic approaches.

P1: Existing state-of-the-art static approaches fail to consider the relationships among features. *LSH-S* [114], n-gram [69], n-perm [69], *BinClone* [36], and *Kam1n0* [30] model assembly code fragments as frequency values of operations and categorized operands. *Tracelet* [24] models assembly code as the editing distance between instruction sequences. *Discovre* [34] and *Genius* [39] construct descriptive features such as the ratio of arithmetic assembly instructions, the number of transfer instructions, and the number of basic blocks, among others. All these approaches assume each feature or category is an independent dimension. However, an *xmm0* Streaming SIMD Extensions (SSE) register is related to SSE operations such as *movaps*. A *fclose* libc function call is

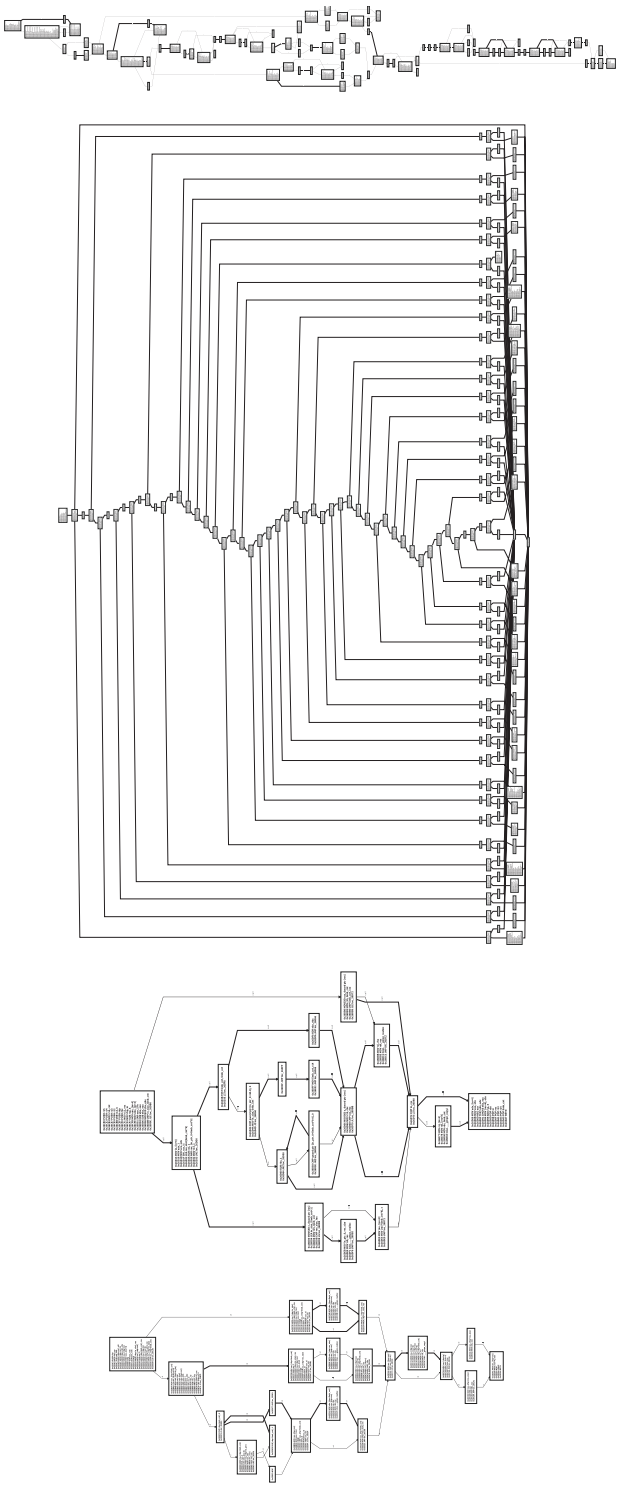

Figure 4.1: Different assembly functions compiled from the same source code of gmpz_tdiv_r_2exp in libgmp. From left to right, the assembly functions are compiled with gcc O0 option, gcc O3 option, LLVM obfuscator Control Flow Graph Flattening option, and LLVM obfuscator Bogus Control Flow Graph option. Asm2Vec can statically identify them as clones.

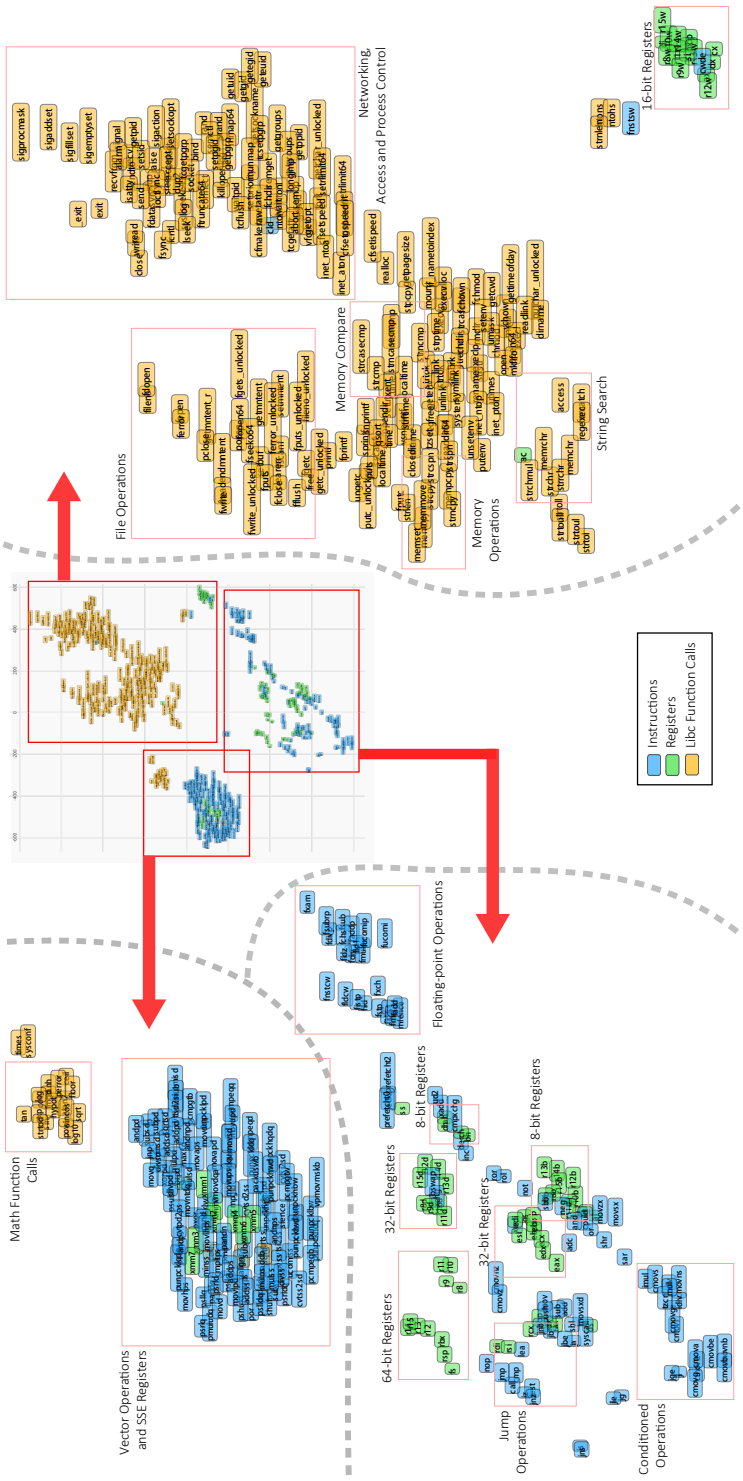

Figure 4.2: *T-SNE* clustering visualization of tokens appearing in assembly code. There are three categories of tokens: operation, operand, and *libc* function call. Each token is represented as a 200-dimensional numeric vector. They are learned by *Asm2Vec* on plain assembly code without any prior knowledge of the assembly language. The training assembly code does not contain the *libc* callee functions' content. For visualization, *T-SNE* reduces the vectors to two dimensions by nearest neighbor approximation. A smaller geometric distance indicates a higher lexical semantic similarity.

related to other file-related libc calls such as *fopen*. A *strcpy* libc call can be replaced with *memcpy*. These relationships provide more semantic information than individual tokens or descriptive statistics.

To address this problem, we propose to incorporate lexical semantic relationship into the feature engineering process. Manually specifying all of the potential relationships from prior knowledge of assembly language is time-consuming and infeasible in practice. Instead, we propose to learn these relationships directly from assembly code. *Asm2Vec* explores co-occurrence relationships among tokens and discovers rich lexical semantic relationships among tokens (see Figure 4.2). For example, *memcpy*, *strcpy*, *memncpy*, and *mempcpy* appear to be semantically similar to each other. SSE registers relate to SSE operands. *Asm2Vec* does not require any prior knowledge in the training process.

P2: The existing static approaches assume that features are equally important [24], [30], [36], [114] or require a mapping of equivalent assembly functions to learn the weights [34], [39]. The chosen weights may not capture the important patterns and diversity that distinguish one assembly function from another. An experienced reverse engineer does not identify a known function by equally looking through the whole content or logic, but rather pinpoints critical spots and important patterns that identify a specific function based on past experience in binary analysis. One also does not need mappings of equivalent assembly code.

To solve this problem, we find that it is possible to simulate the way in which an experienced reverse engineer works. Inspired by recent development in representation learning [78], [79], we propose to train a neural network model to read a large sample of assembly code data and let the model identify the best representation that distinguishes one function from the rest. In this chapter, we make the following contributions:

- We propose a novel approach for assembly clone detection. It is the first work that employs representation learning to construct a feature vector for assembly code as a way to mitigate problems **P1** and **P2** in current hand-crafted features. All previous research on assembly clone search requires a manual feature engineering process. The clone search engine is part of an open source platform.[1]

[1] https://github.com/McGill-DMaS/Kam1n0-Plugin-IDA-Pro

- We develop a representation learning model, namely *Asm2Vec*, for assembly code syntax and control flow graph. The model learns latent lexical semantics between tokens and represents an assembly function as an internally weighted mixture of collective semantics. The learning process does not require any prior knowledge about assembly code, such as compiler optimization settings or the correct mapping between assembly functions. It only needs assembly code functions as inputs.
- We show that *Asm2Vec* is more resilient to code obfuscation and compiler optimizations than state-of-the-art static features and dynamic approaches. Our experiment covers different configurations of compilers and a strong obfuscator that substitutes instructions, splits basic blocks, adds bogus logic, and destroys the original control flow graph. We also conduct a vulnerability search case study on a publicly available vulnerability dataset, where *Asm2Vec* achieves zero false positives and 100% recalls. It outperforms a dynamic state-of-the-art vulnerability search method.

Asm2Vec as a static approach cannot completely defeat code obfuscation. However, it is more resilient to code obfuscation than state-of-the-art static features. This chapter is organized as follows: Section 4.1 formally defines the search problem. Section 4.2 systematically integrates representation learning into a clone search process. Section 4.3 describes the model. Section 4.4 presents our experiment. Section 4.5 discusses the literature. Section 4.6 discusses the limitations and concludes the chapter.

4.1 Problem Definition

In the assembly clone search literature, there are four types of clones [30], [36], [114]: Type I: literally identical; Type II: syntactically equivalent; Type III: slightly modified; and Type IV: semantically similar. We focus on Type IV clones, where assembly functions may appear syntactically different but share similar functional logic in their source code. For example, the same source code with and without obfuscation, or a patched source code between different releases. We use the following notions: *function* denotes an assembly function; *source function* represents the original function written in source code, such as C++; *repository function* stands for the assembly function that is indexed inside the repository; and *target function* denotes the assembly function query.

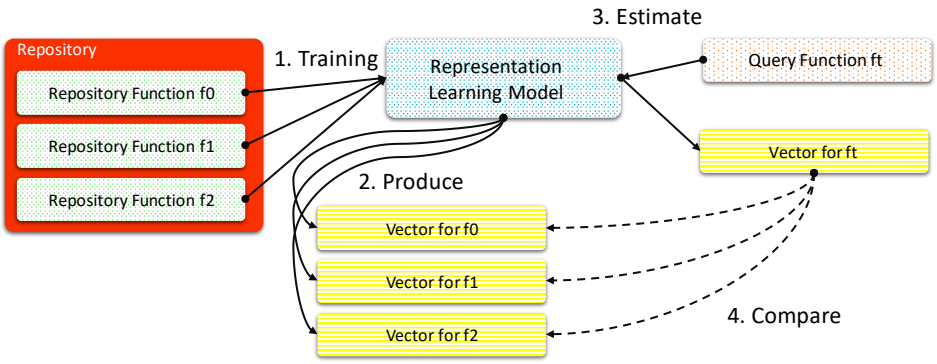

Figure 4.3: The overall work flow of *Asm2Vec*.

Given an assembly function, our goal is to search for its semantic clones from the repository RP. We formally define the search problem as follows:

Definition 10 *(Assembly function clone search) Given a target function f_t, the search problem is to retrieve the top-k repository functions $f_s \in \mathrm{RP}$, ranked by their semantic similarity, so they can be considered as Type IV clones.*

4.2 Overall Workflow

Figure 4.3 shows the overall workflow. There are four steps: *Step 1:* Given a repository of assembly functions, we first build a neural network model for these functions. We only need their assembly code as training data without any prior knowledge. *Step 2:* After the training phase, the model produces a vector representation for each repository function. *Step 3:* Given a target function f_t that was not trained with this model, we use the model to estimate its vector representation. *Step 4:* We compare the vector of f_t against the other vectors in the repository by using cosine similarity to retrieve the top-k ranked candidates as results.

The training process is a one-time effort and is efficient to learn representation for queries. If a new assembly function is added to the repository, we follow the same procedure in Step 3 to estimate its vector representation. The model can be retrained periodically to guarantee the vectors' quality.

4.3 Assembly Code Representation Learning

In this section, we propose a representation learning model for assembly code. Specifically, our design is based on the *PV-DM* model [78]. The *PV-DM* model learns document representation based on the tokens in the document. However, a document is sequentially laid out, which is different than assembly code, as the latter can be represented as a graph and has a specific syntax. First, we describe the original *PV-DM* neural network, which learns a vector representation for each text paragraph. Then, we formulate our *Asm2Vec* model and describe how it is trained on instruction sequences for a given function. After, we elaborate how to model a control flow graph as multiple sequences.

4.3.1 Preliminaries

The *PV-DM* model is designed for text data. It is an extension of the original *word2vec* model. It can jointly learn vector representations for each word and each paragraph. Figure 4.4 shows its architecture.

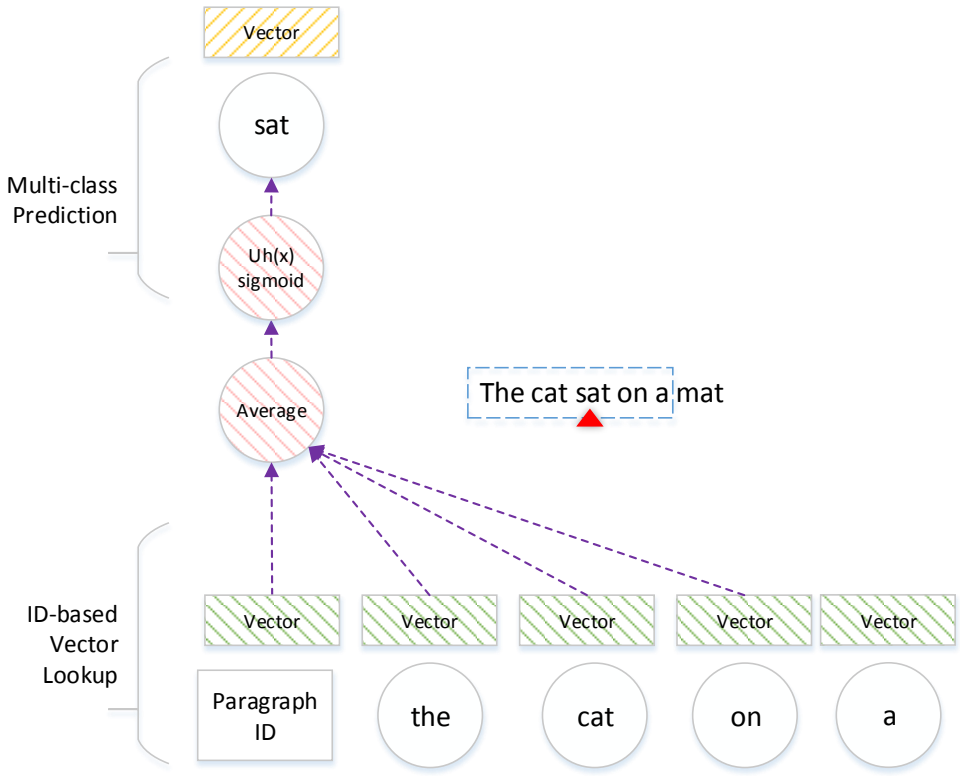

Figure 4.4: The PV-DM model.

Given a text paragraph that contains multiple sentences, *PV-DM* applies a sliding window over each sentence. The sliding window starts from the beginning of the sentence and moves forward a single word at each step. For example, in Figure 4.4, the sliding window has a size of 5. In the first step, the sliding window contains the five words 'the', 'cat', 'sat', 'on', and 'a'. The word 'sat' in the middle is treated as the *target* and the surrounding words are treated as the *context*. In the second step, the window moves forward a single word and contains 'cat', 'sat', 'on', 'a', and 'mat', where the word 'on' is the target.

At each step, the *PV-DM* model performs a multi-class prediction task (see Figure 4.4). It maps the current paragraph into a vector based on the paragraph ID and maps each word in the context into a vector based on the word ID. The model averages these vectors and predicts the target word

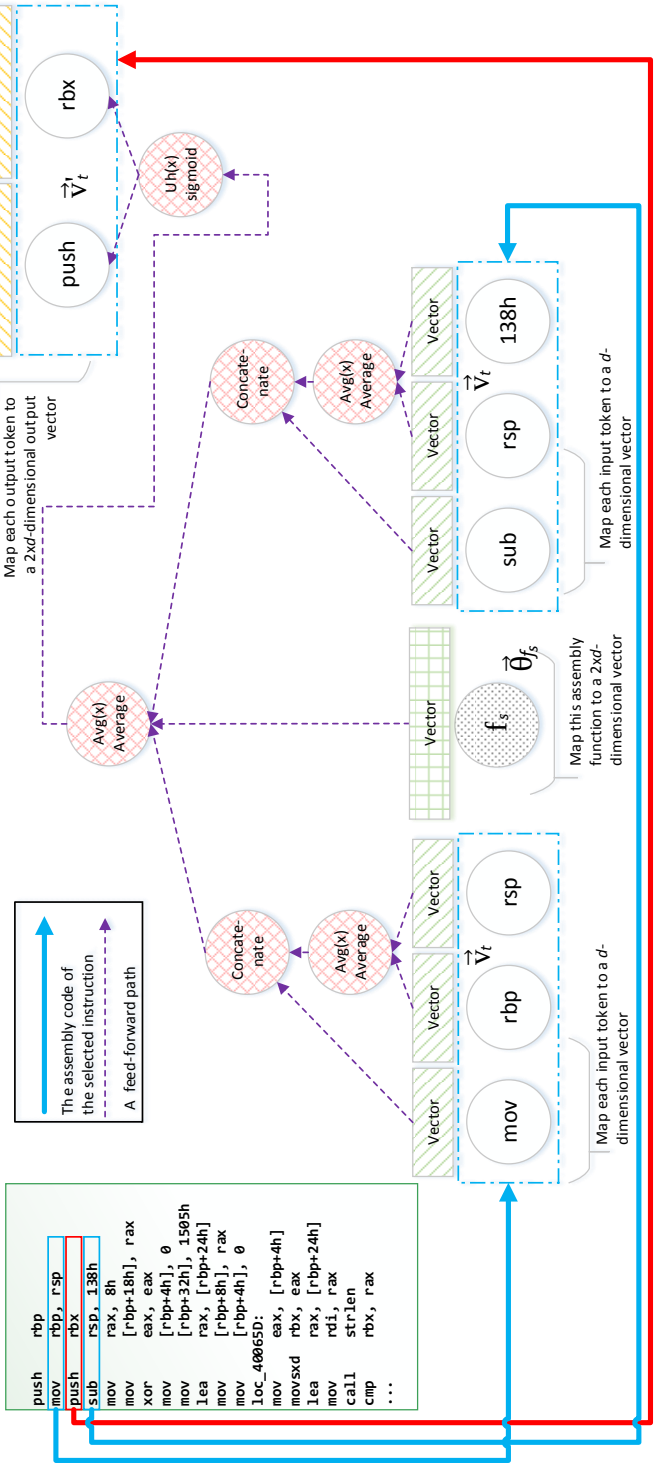

Figure 4.5: The proposed *Asm2Vec* neural network model for assembly code.

from the vocabulary through a softmax classification. The back-propagated classification error will be used to update these vectors. Formally, given a text corpus T that contains a list of paragraphs $p \in \mathrm{T}$, each paragraph p contains a list of sentences $s \in p$, and each sentence is a sequence of $|s|$ words $w_t \in s$. *PV-DM* maximizes the log probability:

$$\sum_{p}^{T} \sum_{s}^{p} \sum_{t=k}^{|s|-k} log\ \mathbf{P}(w_t|p, w_{t-k}, ..., w_{t+k}) \tag{4.1}$$

The sliding window size is $2k + 1$. The paragraph vector captures the information that is missing from the context to predict the target. It is interpreted as topics [78]. *PV-DM* is designed for text data that is sequentially laid out. However, assembly code carries richer syntax than plaintext. It contains operations, operands, and control flow that are structurally different than plaintext. These differences require a different model architecture design that cannot be addressed by *PV-DM*. Next, we present a representation learning model that integrates the syntax of assembly code.

4.3.2 The Asm2Vec Model

An assembly function can be represented as a control flow graph (CFG). We propose to model the control flow graph as multiple sequences. Each sequence corresponds to a potential execution trace that contains linearly laid-out assembly instructions. Given a binary file, we use the IDA Pro[2] disassembler to extract a list of assembly functions, their basic blocks, and control flow graphs.

This section corresponds to Step 1 and 2 in Figure 4.2. In these steps, we train a representation model and produce a numeric vector for each repository function $f_s \in \mathrm{RP}$. Figure 4.5 shows the neural network structure of the model. It is different than the original *PV-DM* model.

First, we map each repository function f_s to a vector $\vec{\theta}_{f_s} \in \mathbb{R}^{2\times d}$. $\vec{\theta}_{f_s}$ is the vector representation of function f_s to be learned in training. d is a user chosen parameter. Similarly, we collect all the unique tokens in the repository RP. We treat operands and operations in assembly code as tokens. We map each token t into a numeric vector $\vec{v}_t \in \mathbb{R}^d$ and another numeric vector $\vec{v'}_t \in \mathbb{R}^{2\times d}$. $\vec{v}_t$ is the vector representations of token t. After training, it represents a token's lexical semantics. $\vec{v}_t$ vectors are used in Figure 4.2 to visualize the relationship among tokens. $\vec{v'}_t$ is used for token

[2]IDA Pro, available at: http://www.hex-rays.com/

prediction. All $\vec{\theta}_{f_s}$ and $\vec{v}_t$ are initialized to small random values around zero. All $\vec{v'}_t$ are initialized to zeros. We use $2 \times d$ for f_s because we concatenate the vectors for operation and operands to represent an instruction.

We treat each repository function $f_s \in \mathrm{RP}$ as multiple sequences $\mathcal{S}(f_s) = seq[1:i]$, where seq_i is one of them. We assume that the order of sequences is randomized. A sequence is represented as a list of instructions $\mathcal{I}(seq_i) = in[1:j]$, where in_j is one of them. An instruction in_j contains a list of operands $\mathcal{A}(in_j)$ and one operation $\mathcal{P}(in_j)$. Their concatenation is denoted as its list of tokens $\mathcal{T}(in_j) = \mathcal{P}(in_j) \mid\mid \mathcal{A}(in_j)$, where $\mid\mid$ denotes concatenation. Constants tokens are normalized into their hexadecimal form.

For each sequence seq_i in function f_s, the neural network walks through the instructions from its beginning. We collect the current instruction in_j, its previous instruction in_{j-1}, and its next instruction in_{j+1}. We ignore the instructions that are out-of-boundary. The proposed model tries to maximize the following log probability across the repository RP:

$$\sum_{f_s}^{\mathrm{RP}} \sum_{seq_i}^{\mathcal{S}(f_s)} \sum_{in_j}^{\mathcal{I}(seq_i)} \sum_{t_c}^{\mathcal{T}(in_j)} log\ \mathbf{P}(t_c | f_s, in_{j-1}, in_{j+1}) \tag{4.2}$$

It maximizes the log probability of seeing a token t_c at the current instruction, given the current assembly function f_s and neighbor instructions. The intuition is to use the current function's vector and the context provided by the neighbor instructions to predict the current instruction. The vectors provided by neighbor instructions capture the lexical semantic relationship. The function's vector remembers what cannot be predicted given the context. It models the instructions that distinguish the current function from the others.

For a given function f_s, we first look up its vector representation $\vec{\theta}_{f_s}$ through the previously built dictionary. To model a neighbor instruction in as $\mathcal{CT}(in) \in \mathbb{R}^{2\times d}$, we average the vector representations of its operands ($\in \mathbb{R}^d$) and concatenate the averaged vector ($\in \mathbb{R}^d$) with the vector representation of the operation. It can be formulated as:

$$\mathcal{CT}(in) = \vec{v}_{\mathcal{P}(in)} || \frac{1}{|\mathcal{A}(in)|} \sum_{t}^{\mathcal{A}(in)} \vec{v}_{t_b} \tag{4.3}$$

Recall that $\mathcal{P}(*)$ denotes an operation, and it is a single token. By averaging f_s with $\mathcal{CT}(in_j - 1)$ and $\mathcal{CT}(in_j + 1)$, $\delta(in, f_s)$ models the joint memory of neighbor instructions:

$$\delta(in_j, f_s) = \frac{1}{3}(\vec{\theta}_{f_s} + \mathcal{CT}(in_{j-1}) + \mathcal{CT}(in_{j+1})) \tag{4.4}$$

Example 1 Consider a simple assembly code function f_s and one of its sequences in Figure 4.5. Take the third instruction where $j = 3$ for example. $\mathcal{T}(in_3)$ = {'push', 'rbx'}. $\mathcal{A}(in_{3-1})$ = {'rbp', 'rsp'}. $\mathcal{P}(in_{3-1})$ = {'mov'}. We collect their respective vectors $\vec{v}_{rbp}$, $\vec{v}_{rsp}$, $\vec{v}_{mov}$ and calculate $\mathcal{CT}(in_{3-1}) = \vec{v}_{mov}||(\vec{v}_{rbp} + \vec{v}_{rsp})/2$. Following the same procedure, we calculate $\mathcal{CT}(in_{3+1})$. With Equation 4.4 and $\vec{\theta}_{f_s}$ we have $\delta(in_3, f_s)$. ■

Given $\delta(in, f_s)$, the probability term in Equation 4.2 can be rewritten as follows:

$$\mathbf{P}(t_c|f_s, in_{j-1}, in_{j+1}) = \mathbf{P}(t_c|\delta(in_j, f_s)) \tag{4.5}$$

Recall that we map each token into two vectors $\vec{v}$ and $\vec{v'}$. For each target token $t_c \in \mathcal{T}(in_j)$, which belongs to the current instruction, we look up its output vector $\vec{v'}_{t_c}$. The probability in Equation 4.5 can be modeled as a softmax multi-class regression problem:

$$\begin{aligned}
\mathbf{P}(t_c|\delta(in_j, f_s)) &= \mathbf{P}(\vec{v'}_{t_c}|\delta(in_j, f_s)) \\
&= \frac{f(\vec{v'}_{t_c}, \delta(in_j, f_s))}{\sum_d^D f(\vec{v'}_{t_d}, \delta(in_j, f_s))} \\
f(\vec{v'}_{t_c}, \delta(in_j, f_s)) &= Uh((\vec{v'}_{t_c})^T \times \delta(in_j, f_s))
\end{aligned}$$

D denotes the whole vocabulary constructed upon the repository RP. $Uh(\cdot)$ denotes a sigmoid function applied to each value of a vector. The total number of parameters to be estimated is $(|D| + 1) \times 2 \times d$ for each pass of the softmax layout. The term $|D|$ is too large for the softmax classification. Following Le and Mikolov [78] and Mikolov, Sutskever, Chen, *et al.* [89], we use

the k negative sampling approach to approximate the log probability as:

$$\begin{aligned} &log\ \mathbf{P}(t_c|\delta(in_j, f_s)) \approx log\ f(\vec{v'}_{t_c}|\delta(in_j, f_s)) \\ &+ \sum_{i=1}^{k} \mathbb{E}_{t_d \backsim P_n(t_c)}\big([\![t_d \neq t_c]\!] log\ f(-1 \times \vec{v'}_{t_d},\ \delta(in_j, f_s))\big) \end{aligned} \tag{4.6}$$

$[\![\cdot]\!]$ is an identity function. If the expression inside this function is evaluated to be true, then it outputs 1; otherwise 0. For example, $[\![1 + 2 = 3]\!] = 1$ and $[\![1 + 1 = 3]\!] = 0$. The negative sampling algorithm distinguishes the correct guess t_c with k randomly selected negative samples $\{t_d|t_d \neq t_c\}$ using $k+1$ logistic regressions. $\mathbb{E}_{t_d \backsim P_n(t_c)}$ is a sampling function that samples a token t_d from the vocabulary D according to the noise distribution $P_n(t_c)$ constructed from D. By taking derivatives, respectively on $\vec{v'}_t$ and $\vec{\theta}_{f_s}$, we can calculate the gradients as follows:

$$\begin{aligned} \frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta) &= \frac{1}{3} \sum_{i}^{k} \mathbb{E}_{t_b \backsim P_n(t_c)}\big([\![t_b = t_c]\!] - f(\vec{v'}_t, \delta(in_j, f_s))\big) \\ &\quad \times \vec{v'}_t \\ \frac{\partial}{\partial \vec{v'}_t} J(\theta) &= [\![t = t_c]\!] - f(\vec{v'}_t, \delta(in_j, f_s)) \times \delta(in_j, f_s) \end{aligned} \tag{4.7}$$

By taking derivatives, respectively on $\vec{v}_{\mathcal{P}(in_{j+1})}$ and $\{\vec{v}_{t_b}|t_b \in \mathcal{A}(in_{j+1})\}$, we can calculate their gradients as follows. It will be the same equation for the previous instruction in_{j-1}, by replacing in_{j+1} with in_{j-1}.

$$\begin{aligned} \frac{\partial}{\partial \vec{v}_{\mathcal{P}(in_{j+1})}} J(\theta) &= \big(\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta)\big)[0 : d-1] \\ \frac{\partial}{\partial \vec{v}_{t_b}} J(\theta) &= \frac{1}{|\mathcal{A}(in_{j+1})|} \times \big(\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta)\big)[d : 2d-1] \\ &\quad t_b \in \mathcal{A}(in_{j+1}) \end{aligned} \tag{4.8}$$

After, we use back propagation to update the values of the involved vectors. Specifically, we update $\vec{\theta}_{f_s}$, all the involved $\vec{v}_t$, and involved $\vec{v'}_t$ according to their gradients, with a learning rate.

Example 2 Continuing from Example 1, where the target token t_c is 'push'. We calculate $\mathbf{P}(\vec{v'}_{push}|\delta(in_j, f_s))$ using negative sampling (Equation 4.6). Next, we calculate the gradients using Equation 4.7 and 4.8. We update all the involved vectors in these two examples, according to their respective gradient, with a learning rate. ■

4.3.3 Modeling Assembly Functions

In this section, we model an assembly function into multiple sequences. Formally, we treat each repository function $f_s \in \mathrm{RP}$ as multiple sequences $\mathcal{S}(f_s) = seq[1:i]$. The original linear layout of control flow graph covers some invalid execution paths. We cannot directly use it as a training sequence. Instead, we model the control flow graph as edge coverage sequences and random walks.

Selective Callee Expansion

Function inlining is a compiler optimization technique that replaces a function call instruction with the body of the called function. It extends the original assembly function and improves its performance by removing call overheads. It significantly modifies the control flow graph and is a major challenge in assembly clone search [17], [55].

BinGo [17] proposes to selectively inline callee functions into the caller function in the dynamic analysis process. We adopt this technique for static analysis. Function call instructions are selectively expanded with the body of the callee function. *BinGo* inlines all the standard library calls for the purpose of semantic correctness. We do not inline any library calls because the lexical semantics among library call tokens have been well captured by the model (see the visualization in Figure 4.2). *BinGo* recursively inlines callee, but we only expand the first-order callees in the call graph. Expanding callee functions recursively will include too many callees' bodies into the caller, which makes the caller function statically more similar to the callee.

The decoupling metric used by *BinGo* captures the ratio of in-degree and out-degree of each

callee function f_c:

$$\alpha(f_c) = \text{outdegree}(f_c)/(\text{outdegree}(f_c) + \text{indegree}(f_c)) \tag{4.9}$$

We adopt the same equation, as well as the same threshold value 0.01, to select a callee for expansion. Additionally, we find that if the callee function is longer than or has a comparable length to the caller, the callee will occupy too large of a portion of the caller. The expanded function appears similar to the callee. Thus, we add an additional metric to filter out lengthy callees:

$$\delta(f_s, f_c) = \text{length}(f_c)/\text{length}(f_s) \tag{4.10}$$

We expand a callee if δ is less than 0.6 or f_s is shorter than 10 lines of instructions. The second condition is to accommodate wrapper functions.

Edge Coverage

To generate multiple sequences for an assembly function, we randomly sample all of the edges from the callee-expanded control flow graph until all the edges in the original graph are covered. For each sampled edge, we concatenate their assembly code to form a new sequence. This way, we ensure that the control flow graph is fully covered. The model can still produce similar sequences, even if the basic blocks in the control flow graph are split or merged.

Random Walk

CACompare [55] uses a random input sequence to analyze the I/O behavior of an assembly function. A random input simulates a random walk on the valid execution flow. Inspired by this method, we extend the assembly sequences for an assembly function by adding multiple random walks on the expanded control flow graph. This way, the generated sequence is much longer than the edge sampling.

Dominator is a widely used concept in control flow analysis and compiler optimizations. A basic block dominates another if one has to pass this block in order to reach the other. Multiple random walks will put a higher probability to cover the basic blocks that dominate others. These

Algorithm 6 Training the Asm2Vec model for one epoch

function TRAIN(Repository RP)
    shuffle(RP)
    for each $f_s \in$ RP **do**
        for each $seq_i \in \mathcal{S}(f_s)$ **do**
            for $j = 1 \rightarrow (|seq_i| - 1)$ **do**
            ▷ Going through each instruction.
                lookup f_s's representation $\vec{\theta}_{f_s}$
                calculate $\mathcal{CT}(in_{j-1})$ by Equ. 4.3
                calculate $\mathcal{CT}(in_{j+1})$ by Equ. 4.3
                calculate $\delta(in_j, f_s)$ by Equ. 4.4
                for each $tkn \in in_j$ **do**
                ▷ Going through each token
                    $targets \leftarrow \mathbb{E}_{t_b \backsim P_n(tkn)} \cup \{tkn\}$
                    ▷ Sample tokens from $P_n(tkn)$
                    calculate and cumulate gradient for $\vec{\theta}_{f_s}$ (Equ. 4.7)
                    calculate gradient for $\vec{v'}_t$ (Equ. 4.7)
                    update $\vec{v'}_t$
                    calculate and cumulate gradient for in_{j-1} (Equ. 4.8)
                    calculate and cumulate gradient for in_{j+1} (Equ. 4.8)
                end for
                update vectors for tokens of in_{j-1}
                update vectors for tokens of in_{j+1}
                update $\vec{\theta}_{f_s}$
            end for
        end for
    end for
end function

---

function $\mathcal{S}$(Function f_s)
    $graph \leftarrow$ CFG(f_s)
    $graph \leftarrow$ ExpandSellectiveCallee($graph$)
    $sequences \leftarrow \{\}$
    for each $edg \in$ SampleEdge($graph$) **do**
        $seq \leftarrow$ source(edg) || target(edg)
        ▷ Concatenate the source and the target blocks
        $sequences \leftarrow sequences \cup \{seq\}$
    end for
    for $i \leftarrow$ numRandomWalk **do**
        $seq \leftarrow$ RandomWalk($graph$)
        $sequences \leftarrow sequences \cup \{seq\}$
    end for

    return $sequences$
end function

popular blocks can be the indicator of loop structures or cover important branching conditions. Using random walks can be considered as a natural way to prioritize basic blocks that dominate others.

4.3.4 Training, Estimating, and Searching

The training procedure corresponds to Algorithm 6. For each function in the repository, it generates sequences by edging sampling and random walks. For each sequence, it goes through each instruction and applies the *Asm2Vec* to update the vectors (Line 10 to 19). As shown in Algorithm 6, the training procedure does not require a ground-truth mapping between equivalent assembly functions.

The estimation step corresponds to Step 3 in Figure 4.3. For an unseen assembly function f_t as query $f_t \notin \mathrm{RP}$ that does not belong to the set of training assembly functions, we first associate it with a vector $\vec{\theta}_{f_t} \in \mathbb{R}^{2\times d}$, which is initialized to a sequence of small values close to zero. Then, we follow the same procedure in the training process, where the neural network goes through each sequence of f_t and each instruction of the sequence. In every prediction step, we fix all $\vec{v}_t$ and $\vec{v'}_t$ in the trained model and only propagate errors to $\vec{\theta}_{f_t}$. At the end, we have $\vec{\theta}_{f_t}$, while the vectors for all $f_s \in \mathrm{RP}$ and $\{\vec{v}_t, \vec{v'}_t | t \in D\}$ remain the same. To search for a match, vectors are flattened and compared using cosine similarity. Cosine similarity is a widely-used metric for embedding vectors.

Algorithm 7 Estimating a vector representation for a query

```
1:  function ESTIMATE(Query Function f_t)
2:      initialize f_t's representation θ⃗_{f_t}
3:      for each seq_i ∈ S(f_t) do
4:          for j = 1 → (|seq_i| − 1) do
5:              calculate CT(in_{j−1}) by Equ. 4.3
6:              calculate CT(in_{j+1}) by Equ. 4.3
7:              calculate δ(in_j, f_t) by Equ. 4.4
8:              for each tkn ∈ in_j do
9:                  targets ← E_{t_b ∽ P_n(tkn)} ∪ {tkn}
10:                 calculate gradient for θ⃗_{f_t} (Equ. 4.7)
11:                 update θ⃗_{f_t}
12:             end for
13:         end for
14:     end for
15: end function
```

Scalability is critical for binary clone search, as there may be millions of assembly functions inside a repository. It is practical to train *Asm2Vec* on a large-scale of assembly code. A similar model on text has been shown to be scalable to billions of text samples for training [89]. In this study, we only use pair-wise similarity for nearest neighbor searching. Pair-wise searching among low-dimensional fixed-length vectors can be fast. In our experiment in Section 4.4.3, there are 139,936 functions. The average training time for each function is 49 milliseconds. The average query response time is less than 300 milliseconds.

4.4 Experiments

We compare *Asm2Vec* with existing available state-of-the-art dynamic and static assembly clone search approaches. All the experiments are conducted with an Intel Xeon 6 core 3.60GHz CPU with 32G memory. To simulate a similar environment in related studies, we limit the JVM to only 8 threads. There are four experiments. First, we benchmark the baselines against different compiler optimizations with *GCC*. Second, we evaluate clone search quality against different heavy code obfuscations with *CLANG* and *O-LLVM*. Third, we use all the binaries of the previous two. In the last one, we apply *Asm2Vec* on a publicly available vulnerability search dataset. All binary files are stripped before clone search. In all of the experiments, we choose $d = 200$, 25 negative samples, 10 random walks, and a decaying learning rate 0.025 for *Asm2Vec*. 200 corresponds to the suggested dimensionality ($2d$) used in [78].

4.4.1 Searching with Different Compiler Optimization Levels

In this experiment, we benchmark the clone search performance against different optimization levels with the *GCC* compiler version 5.4.0. We evaluate *Asm2Vec* based on 10 widely used utility and numeric calculation libraries in Table 4.1. They are chosen according to an internal statistic of the prevalence of FOSS libraries. We first compile a selected library using the *GCC* compiler with four different compiler optimization settings, which results in four different binaries. Then, we test every combination of two of them, which corresponds to two different optimization levels. Given two binaries from the same library but with different optimization levels, we link their assembly functions using the compiler-output debug symbols and generate a clone mapping between func-

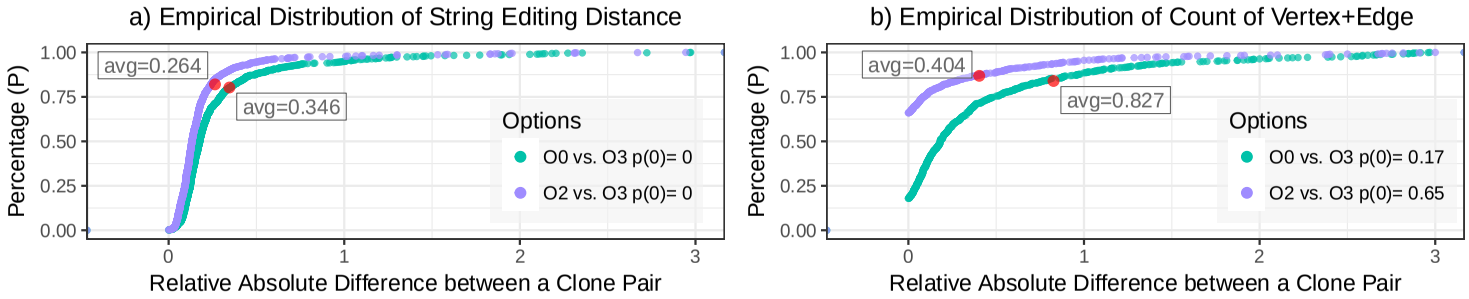

Figure 4.6: The difference between the O0/O2 optimized and the O3 optimized function. a) Relative string editing distance. 0.264 indicates that around 26.4% percent of bytes are different between two options for the same source code function. b) Relative absolute difference in the count of vertices and edges. 0.404% indicates that one function has 40.4% more vertices and edges than the other.

Compiler optimization O2 and O3													
Baselines	BusyBox	CoreUtils	Libgmp	ImageMagick	Libcurl	LibTomCrypt	OpenSSL	SQLite	zlib	PuTTYgen	Avg.	p	d
BinGo†	Ø	.490	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.490	○	NA
Composite	.789	.643	.910	.787	.842	.646	.783	.777	.813	.8IHI 38	.783	○	-0.97
Constant	.437	.338	.202	.711	.522	.440	.365	.368	.549	.571	.450	●	-1.00
Graphlet	.309	.268	.355	.262	.321	.297	.212	.313	.406	.148	.289	●	-1.00
Graphlet-C	.662	.581	.680	.678	.689	.559	.586	.687	.730	.795	.665	●	-1.00
Graphlet-E	.278	.225	.362	.270	.271	.219	.199	.280	.399	.355	.286	●	-1.00
MixedGram	.811	.663	.906	.792	.848	.652	.789	.804	.858	.865	.798	●	-0.97
MixedGraph	.445	.413	.436	.427	.486	.379	.350	.458	.564	.533	.449	●	-1.00
n-gram	.774	.644	.874	.739	.814	.593	.748	.760	.812	.781	.754	●	-1.00
n-perm	.803	.654	.912	.788	.848	.646	.785	.799	.850	.855	.793	●	-0.97
FuncSimSearch	.157	.169	.848	.514	.663	.698	.726	.533	.488	.363	.516	●	-1.00
PV(DM/DBOW)	.895	.899	.959	.952	.927	.945	.919	.898	.873	.823	.909	●	-0.43
Asm2Vec*	**.954**	**.929**	**.973**	**.971**	**.951**	**.991**	**.931**	**.926**	**.885**	**.891**	**.940**	○	0.00
Compiler optimization O0 and O3													
Baselines	BusyBox	CoreUtils	Libgmp	ImageMagick	Libcurl	LibTomCrypt	OpenSSL	SQLite	zlib	PuTTYgen	Avg.	p	d
BinGo†	Ø	.317	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	.317	○	NA
CACompare†	.844	Ø	Ø	**.893**	.794	Ø	**.795**	Ø	Ø	.717	.808	○	NA
Composite	.013	.031	.019	.017	.004	.005	.007	.004	.036	.127	.026	●	-1.00
Constant	.239	.128	.101	.610	.369	.258	.270	.182	.360	.439	.296	●	-1.00
Graphlet	.017	.008	.049	.010	.023	.011	.009	.014	.029	.016	.019	●	-1.00
Graphlet-C	.018	.020	.012	.022	.027	.001	.034	.012	.065	.102	.031	●	-1.00
Graphlet-E	.021	.011	.075	.019	.017	.003	.018	.019	.051	.058	.029	●	-1.00
MixedGram	.016	.033	.019	.018	.011	.005	.007	.006	.036	.116	.028	●	-1.00
MixedGraph	.034	.028	.062	.024	.039	.015	.023	.030	.064	.097	.042	●	-1.00
n-gram	.011	.029	.012	.021	.011	.010	.005	.003	.036	.129	.027	●	-1.00
n-perm	.017	.029	.021	.021	.011	.006	.007	.005	.036	.129	.028	●	-1.00
FuncSimSearch	.008	.019	.323	.039	.036	.030	.220	.011	.054	.040	.078	●	-1.00
PV(DM/DBOW)	.745	.677	.760	.802	.792	.821	.759	.758	.713	.615	.744	●	-0.61
Asm2Vec*	**.856**	**.781**	**.763**	.837	**.850**	**.921**	.792	**.776**	**.722**	**.788**	**.809**	○	0.00

Table 4.1: Clone search between different compiler optimization options using the *Precision at Position 1* (*Precision@1*) metric. It captures the ratio of assembly functions that are correctly matched at position 1. In this case, it equals *Recall at Position 1*. *Asm2Vec* is our proposed method. †denotes cited performance. ○ and ● respectively indicate $p > 0.05$ and $p \leq 0.01$ for Wilcoxon signed-rank test between *Asm2Vec* and each baseline. d denotes the *Cliff's Delta* value. *Holm-Bonferroni's Family-Wise Error Rate (FWER)* < 0.05.

tions. This mapping is used as the ground-truth data for evaluation only. We search the first against the second in RP, and after we search for the second against the first in RP. Only the binary in the repository is used for training. We take the average of the two.

A higher optimization level contains all optimization strategies from the lower level. The comparison between O2 and O3 is the easiest one (Figure 4.6). On average, 26% bytes of a function are modified and none of the functions are identical. 40% of a control flow graph is modified and 65% function pairs share similar graph complexity. It can be considered as the best situation where the optimization strategies used in two binaries are similar. The comparison between O0 and O3 is the most difficult one. It can be considered as the worst situation where there exists a large difference in the optimization strategies (Figure 4.6). On average, 34% bytes of a function are modified and none of the functions are identical. 82% of a control flow graph is modified, and 17% function pairs share similar graph complexity. Table 4.1 presents the results in these two situations. Due to the large number of cases, we only list the results for these two cases to demonstrate the best and worst situations. The results of other cases lie between these two and follow the same ranking.

Andriesse et al. [5] point out that using supervised machine learning may risk having invalid experiment results. For example, splitting *coreutils* binaries into training set and testing set may lead to an artificially inflated result because these binaries share a very similar code base. This issue is not applicable to our experiment. First, we follow the unsupervised learning paradigm, where the true clone mapping is only used for evaluation. Second, our training data is very different from the testing data, as shown in Figure 4.6 and Figure 4.7. For example, the *coreutils* library comes with many binaries but we statically linked them into a *single* binary. We train the O0-optimized binary and match the O3-optimized binary. These two binaries are very different.

We use the *Precision at Position 1* (*Precision@1*) metric. For every query, if a baseline returns no answer, we count the precision as zero. Therefore, *Precision@1* captures the ratio of assembly functions that are correctly matched, which is equal to *Recall at Position 1*. We benchmark nine feature representations proposed by Khoo, Mycroft, and Anderson [69]: mnemonic n-grams (denoted as n-gram), mnemonic n-perms (denoted as n-perm), Graphlets (denoted as *Graphlet*), Extended Graphlets (denoted as *Graphlet-E*), Colored Graphlets (denoted as *Graphlet-C*), Mixed Graphlets (denoted as *MixGraph*), Mixed n-grams/perms (denoted as *MixGram*), *Constants*, and the Composite of n-grams/perms and Graphlets (denoted as *Composite*). The idea of using *Graphlet* orig-

inated from Kruegel, Kirda, Mutz, *et al.* [75]. These baseline methods cover a wide range of popular features from token to graph substructure. These baselines are configured according to the reported best settings in the paper. We also include the original *PV-DM* model and *PV-DBOW* model as a baseline where each assembly function is treated as a document. We pick the best results and denote it as *PV-(DM/DBOW)*. We only tune the configurations for *PV-(DM/DBOW)* as well as *Asm2Vec* on the *zlib* dataset. *FuncSimSearch* is an open source assembly clone static search toolkit recently released by Google[3]. It has a default training dataset that contains a ground-truth mapping of equivalent assembly functions. The state-of-the-art dynamic approach *BinGo* [17] and *CACompare* [55] are unavailable for evaluation. However, we conduct the experiment in the same way using the same metric. Their reported results are included in Table 4.1. We also include the Wilcoxon signed-rank test across different binaries to see if the difference in performance is statistically significant.

As shown in Table 4.1, *Asm2Vec* significantly outperforms static features in both the best and worse situation. It also outperforms *BinGo*, a recent semantic clone search approach that involves dynamic features. It shows that *Asm2Vec* is robust against heavy syntax modifications and intensive inlining introduced by the compiler. Even in the worse case, the learned representation can still correctly match more than 75% of assembly functions at position 1. It even achieves competitive performance against the state-of-the-art dynamic approach *CACompare* for semantic clone detection. The difference is not statistically different, due to the small sample size. *Asm2Vec* performs stably across different libraries and is able to find clones with high precision. On average, it achieves more than 93% precision in detecting clones among compiler optimization options O1, O2, and O3. As the difference between two optimization levels increases, the performance of the *Asm2Vec* decreases. Nevertheless, it is much less sensitive than the other static features, which demonstrates its robustness.

Discovre and *Genius* are two recent static approaches that use descriptive statistics and graph matching. Neither of them are available for evaluation. *CACompare* has been shown to outperform *Discovre* [34], *Genius* [39], and *Blanket* [33]. Our approach achieves comparable performance to *CACompare*, which indirectly compares *Asm2Vec*'s performance to *Discovre* and *Genius*.

[3]Available at `https://github.com/google/functionsimsearch`

In the best situation where we compare between optimization level O2 and O3, the baseline static features' performance is inline with the result reported in the original paper, which shows the correctness of our implementation. In the worse case, we notice that the *Constant* model outperforms the other static features based on assembly instructions and graph structures. The reason is that constant tokens do not suffer from changes in assembly instructions and subgraph structures. We also notice that *BinGo*, in the worst case, outperforms static features. However, in the best case, its performance is not as good as static features, such as *Graphlet-C* and n-grams, because the noise at the symbolic logic level is higher than at the assembly code level. Logical expressions promote recall and can find clones when the syntax is very different. However, assembly instructions can provide more precise information for matching.

The largest binary, OpenSSL, has more than 5,000 functions. *Asm2Vec* takes on average 153 ms to train an assembly function and 20 ms to process a query. For OpenSSL, *CACompare* takes on average 12 seconds to fulfill a query.

4.4.2 Searching with Code Obfuscation

Obfuscator-LLVM (O-LLVM) [64] is built upon the *LLVM* framework and the *CLANG* compiler toolchain. It operates at the intermediate language level and modifies a program's logics before the binary file is generated. It increases the complexity of the binary code. O-LLVM uses three different techniques and their combination: *Bogus Control Flow Graph (BCF)*, *Control Flow Flattening (FLA)*, and *Instruction Substitution (SUB)*. Figure 4.7 shows the statistics on differences.

- *BCF* modifies the control flow graph by adding a large number of irrelevant random basic blocks and branches. It will also split, merge, and reorder the original basic blocks. BCF breaks CFG and basic block integrity (on average 149% vertices/edges are added).
- *FLA* reorganizes the original CFG using a complex hierarchy of new conditions as switches (see an example in Figure 4.1). The original instructions are heavily modified to accommodate the new entering conditions and variables. The linear layout has been completely modified (on average 376% vertices and edges are added). Graph-based features are oblivious to this technique. It is also unscalable for a dynamic approach to fully cover the CFG.
- *SUB* substitutes fragments of assembly code to its equivalent forms by going one pass over the

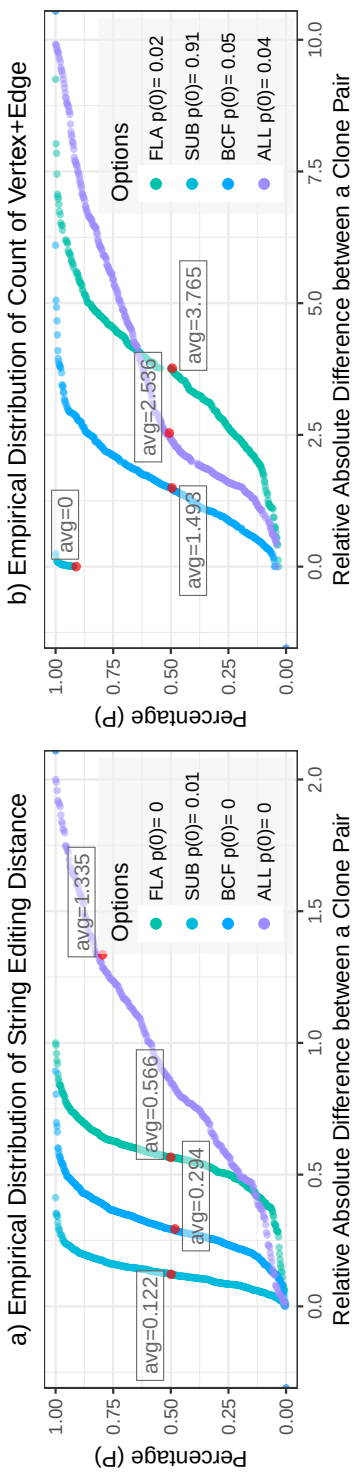

Figure 4.7: The difference between the original function and the obfuscated function. a) Relative string editing distance. 0.122 indicates that around 12.2% percent of bytes are modified. b) Relative absolute difference in the count of vertices and edges. 1.49% indicates the obfuscated function has 149% more vertices and edges in CFG.

	O-LLVM - Bogus Control Flow Graph (BCF)					O-LLVM - Instruction Substitution (SUB)				
Baselines	Libgmp	ImageMagick	LibTomCrypt	OpenSSL	Avg.	Libgmp	ImageMagick	LibTomCrypt	OpenSSL	Avg.
Composite	.226	.224	.312	.246	.252	.620	.675	.600	.766	.665
Constant	.130	.592	.412	.318	.363	.173	.622	.492	.360	.412
Graphlet	.003	.005	.033	.007	.012	.198	.158	.411	.308	.269
Graphlet-C	.112	.118	.165	.124	.130	.626	.572	.539	.585	.581
Graphlet-E	.026	.011	.050	.014	.025	.454	.216	.286	.271	.307
MixedGram	.220	.234	.375	.303	.283	.585	.642	.563	.743	.633
MixedGraph	.011	.007	.049	.014	.020	.356	.325	.495	.488	.416
n-gram	.134	.134	.295	.195	.190	.466	.516	.513	.670	.541
n-perm	.233	.224	.374	.274	.276	.557	.624	.558	.736	.619
FuncSimSearch	.109	.022	.029	.027	.047	.685	.442	.699	.330	.539
PV(DM/DBOW)	.784	.870	.842	.768	.816	.935	.968	.964	.958	.956
Asm2Vec *	**.802**	**.920**	**.933**	**.883**	**.885**	**.940**	**.960**	**.981**	**.961**	**.961**
	O-LLVM - Control Flow Flattening (FLA)					O-LLVM - SUB+FLA+BCF				
Baselines	Libgmp	ImageMagick	LibTomCrypt	OpenSSL	Avg.	Libgmp	ImageMagick	LibTomCrypt	OpenSSL	Avg.
Composite	.138	.129	.052	.027	.086	.219	.226	.015	.009	.117
Constant	.105	.480	.215	.209	.252	.137	.591	.173	.159	.265
Graphlet	.000	.002	.000	.000	.000	.000	.005	.000	.000	.001
Graphlet-C	.003	.008	.000	.001	.003	.107	.124	.000	.000	.058
Graphlet-E	.001	.002	.000	.000	.001	.020	.012	.000	.000	.008
MixedGram	.148	.143	.075	.036	.101	.221	.234	.018	.010	.121
MixedGraph	.003	.003	.000	.000	.002	.006	.010	.000	.000	.004
n-gram	.095	.093	.059	.030	.069	.154	.144	.013	.007	.079
n-perm	.133	.126	.055	.033	.087	.224	.222	.018	.008	.118
FuncSimSearch	.095	.001	.004	.008	.027	.110	.025	.003	.008	.037
PV(DM/DBOW)	**.852**	**.938**	.786	.763	.835	.780	.873	.639	.595	.722
Asm2Vec *	.772	.920	**.890**	**.795**	**.844**	**.854**	**.880**	**.830**	**.690**	**.814**

Table 4.2: Clone search between the original and obfuscated binary using the *Precision at Position 1* (*Precision@1*) metric. It captures the ratio of functions that are correctly matched at position 1, which is equal to *Recall at Position 1* (*Recall@1*) in this case. The difference between *Asm2Vec* and each baseline is significant (p <0.01 in a Wilcoxon signed-rank test).

function logic using predefined rules. This technique modifies the contents of basic blocks and adds new constants. For example, additions are transformed to $a = b - (-c)$. Subtractions are transformed to $r = rand(); a = b - r; a = a - c; a = a + r$. And operations are transformed to $a = (b \wedge \backsim c) \& b$. *SUB* does not change much of the graph structure (91% of functions keep the same number of vertices and edges).

- *BCF+FLA+SUB* uses all the obfuscation options above.

O-LLVM heavily modifies the original assembly code. It breaks the CFG and the basic blocks integrity. By design, most of the static features are oblivious to the obfuscation. By using the *CLANG* compiler with *O-LLVM*, we successfully compile four libraries used in the last experiment and evaluate *Asm2Vec* using them. There were compilation errors when compiling the other binaries with the *CLANG+O-LLVM* toolchain. The errors are caused by the obfuscation tool itself. According to Figure 4.7, there is a significant difference between the original and the ones obfuscated with BCF and FLA. BCF doubles the number of vertices and edges. FLA almost doubles the latter. With SUB, the number of assembly instructions significantly increases. We use the same set of baselines and configurations from the previous experiment except for *BinGo* and *CACompare*; they are unavailable for evaluation and the original papers do not include such an experiment.

We first compile a selected library without any obfuscation techniques applied. After, we compile the library again with a chosen obfuscation technique to have an original and an obfuscated binary. We link their assembly functions by using debug symbols and generate a one-to-one clone mapping between assembly functions. This mapping is used for evaluation purposes only. After stripping binaries, we search the original against the obfuscated. Then, we search for the obfuscated against the original. We report the average. We use the *Precision@1* as our evaluation measure. In this case, *Precision@1* equals *Recall@1* because we treat 'no-answer' for a query as a zero precision.

Table 4.2 shows the results for *O-LLVM*. We find that instructions substitution can significantly reduce the performance of n-gram. SUB breaks the sequence by adding instructions in between. n-perm performs better than n-gram, since it ignores the order of tokens. Graph-based features can still recover more than 60% of clones as the graph structure is not heavily modified. *Asm2Vec* can achieve more than 96% precision against assembly instruction substitution. Instructions are replaced with their equivalent form, which in fact still shares similar lexical semantic to the original.

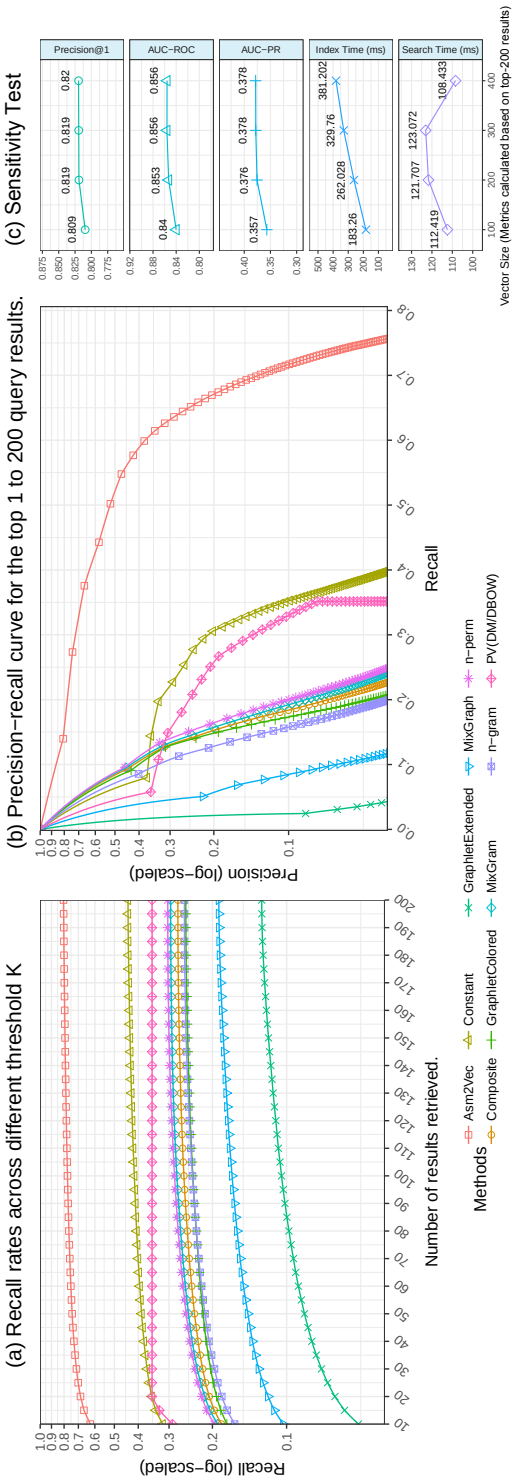

Figure 4.8: Baseline comparison for the third experiment. There are 139,936 assembly functions. We search each one against the rest. The set is a mixture of different compilers, compiler optimization settings, and *O-LLVM* obfuscation settings. a) Recall rates are plotted for different top-K retrieved results. b) Recall-Precision Curve. c) Sensitivity test on dimensionality.

This information is well captured by *Asm2Vec*.

After applying BCF obfuscation, *Asm2Vec* can still achieve more than 88% precision, where the control flow graph already looks very different from the original. It shows that *Asm2Vec* is resilient to the inserted junk code and faked basic blocks. The FLA obfuscation destroys all the subgraph structures. This is also reflected from the degraded performance of graph sub-structure features. Most of them have a precision value around zero. Even in such situations, *Asm2Vec* can still correctly match 84% of assembly function clones. It shows that *Asm2Vec* is resilient to sub-structure changes and linear layout changes. After applying all the obfuscation techniques, *Asm2Vec* can still recover around 81% of assembly functions.

Asm2Vec can correctly pinpoint and identify critical patterns from noise. Inserted junk basic blocks or noise instructions follow the general syntax of random assembly code, which can be easily predicted by neighbor instructions. The function representation in *Asm2Vec* captures the missing information that cannot be provided by neighbor instructions. It also weights this information to best distinguish one function from another.

4.4.3 Searching against All Binaries

In this experiment, we use all the binaries in the previous two experiments. We evaluate whether *Asm2Vec* can distinguish different assembly functions when the candidate set is large. We also evaluate its performance with varying retrieval thresholds to inspect whether true positives are ranked at the top. Specifically, there are in total 60 binaries that are a mixture of libraries compiled for different compiler options (O0-O3), different compilers (*GCC* and *CLANG*), and different *O-LLVM* obfuscation configurations. Following the experiment in *Genius* [39] and *Discovre* [34], we consider assembly functions that have at least 5 basic blocks. However, we do not use sampling. We use all of them. In total, there are 139,936 assembly functions. For each of them, we search against the rest to find clones. We sort the returned results and evaluate each in sequence. We use the same set of baselines and configuration from the last experiment except for *FuncSimSearch*, since it throws segmentation fault when indexing all the binaries.

We collect recall and precision at different top-k positions. We plot recall against k in Figure 4.8(a). We remove *Graphlet* from the figure, it does not perform any better than *Graphlet-*

		ESH [22]			Asm2Vec		
Vulnerability	CVE	FP	ROC	CROC	FP	ROC	CROC
Heartbleed	2014-0160	0	1	1	0	1	1
Shellshock	2014-6271	3	0.999	0.996	0	1	1
Venom	2015-3456	0	1	1	0	1	1
Clobberin' Time	2014-9295	19	0.993	0.956	0	1	1
Shellshock #2	2014-7169	0	1	1	0	1	1
ws-snmp	2011-0444	1	1	0.997	0	1	1
wget	2014-4877	0	1	1	0	1	1
ffmpeg	2015-6826	0	1	1	0	1	1

Table 4.3: Evaluating *Asm2Vec* on the vulnerability dataset [22] using the False Positives (FP), Receiver Operating Characteristic (ROC), and Concentrated ROC (CROC) metrics. For all the cases, *Asm2Vec* retrieves all results without any false positives.

Extended. Even with a large size of assembly functions, *Asm2Vec* can still achieve a recall of 70% for the top 20 results. It significantly outperforms other traditional token-based and graph-based features. Moreover, we observe that token-based approaches in general perform better than subgraph-based approaches.

We plot precision against recall for each baseline in Figure 4.8(b). This curve evaluates a clone search engine with respect to the trade-off between precision and recall, when varying the number of retrieved results. As shown in the plot, *Asm2Vec* outperforms traditional representations of assembly code. It achieves 82% precision for the returned top clone search result where $k = 1$. The false positives on average have 33 basic blocks ($\sigma = 231$). On the other hand, all the functions in the dataset on average have 47 basic blocks ($\sigma = 110$) as a prior. By using a one-sided Kolmogorov-Smirnov test, we can conclude that false positives have a smaller number of basic blocks than the overall population ($p < 2.2e^{-16}$). We conduct a sensitivity test based on top-200 results to evaluate different choices of vector size. Vector size is the dominant factor for the number of parameters and thus the model's complexity. Figure 4.8 (c) shows that with difference vector size, *Asm2Vec* is stable for both efficacy and efficiency. We tried to incorporate more neighbor instructions. However, this increases the possible patterns to be learned and requires more data. In our experiment, we did not find such design effective.

4.4.4 Searching Vulnerability Functions

In the above experiments, we evaluate *Asm2Vec*'s overall performance on matching general assembly functions. In this case study, we apply *Asm2Vec* on a publicly available vulnerability dataset[4] presented in [22] to evaluate its performance in actually recovering the reuse of the vulnerabilities in functions. Vulnerability retrieval is a typical use case for assembly clone search. The dataset contains 3,015 assembly functions. For each of the 8 given vulnerabilities, the task is to retrieve its variants from the dataset. The variants are either from different source code versions or generated by different versions of GCC, ICC, and CLANG compilers. This dataset is closely related to the real-life scenario.

Figure 4.9 shows an example of using *Asm2Vec* to search for the *Heartbleed* vulnerability in the dataset. The query is a function containing the *Heartbleed* vulnerability in *OpenSSL* version 1.0.1f, compiled with Clang 3.5. There are total 15 different functions containing this vulnerability. The pie chart in each ranked entry indicates the similarity. Each ranked entry contains the assembly function name and its corresponding binary file. As shown in the ranked list, *Asm2Vec* successfully retrieves all 15 candidates in the top 15 results. Therefore, it has a precision and recall of 1 for this query. The first entry corresponds to the same function as the query. However, it does not have a similarity of 1 since the query's representation is estimated, but the one in repository is trained. However, it is still ranked first.

We implement *Asm2Vec* as an open source vulnerability search engine and follow the same experimental protocol to compare its performance with the state-of-the-art vulnerability search solution in [22]. Table 4.3 shows the results. We use the same performance metrics as [22]: False Positives (FP), Receiver Operating Characteristic (ROC), and Concentrated ROC (CROC). For all the vulnerabilities, *Asm2Vec* has zero false positives and 100% recall. Therefore, it achieves a ROC and a CROC of 1. It outperforms [22].

Tigress [8] is another advanced obfuscator. It transforms the C Intermediate Language (CIL) using virtualization and Just-In-Time (JIT) execution. *Tigress* failed to obfuscate a complete library binary due to compilation errors. Therefore we were unable to evaluate *Asm2Vec* against *Tigress* in the same way as against *O-LLVM* in Section 4.4.2. We increase the difficulty on the vulnerability

[4] Available at https://github.com/nimrodpar/esh-dataset-1523

search by using the *Tigress* obfuscator. In this experiment, for each of the 8 different vulnerabilities, we obfuscate the query function with literals encoded, virtualization, and Just-In-Time execution. Then, we try to recover their original variants from the dataset. *Encode Literals*: Literal integers are replaced with opaque expressions. Literal strings are replaced with a function that generates them at runtime. *Virtualization*: This transformation turns a function into an interpreter with specialized byte code translation. By design, it is difficult for a static approach to detect clones protected by this technique. *JIT*: It transforms the function to generate its code at runtime. Almost every instruction is replaced with a function call. By design, a static approach can hardly recover any variants. Our result shows that *Asm2Vec* is still able to recover 97.2% with literals encoded, 35% with virualization, and 45% with JIT execution (see Table 4.4). We inspect the result and find that *Asm2Vec* tries to match any similar information neglected by the obfuscator. However, after applying three obfuscation techniques at the same time, *Asm2Vec* can no longer recover any clone.

4.5 Related Work

Static approaches such as *k-gram* [94], *LSH-S* [114], n-gram [69], *BinClone* [36], *ILine* [59], and *Kam1n0* [30] rely on operations or categorized operands as static features. *BinSequence* [56] and *Tracelet* [24] model assembly code as the editing distance between instruction sequences. All these features failed to leverage the semantic relationship between operations or categories. *TEDEM* [104] compares basic blocks by their expression trees. However, even semantically similar instructions result in different expressions and side effects, which make them sensitive to instruction changes. *ILine* [59], *Discovre* [34], *Genius* [39], *BinSign* [97], and *BinShape* [123] construct descriptive statistic features, such as ratio of arithmetic assembly instructions, ratio of transfer instructions, number of basic blocks, and number of function calls, among others. Instruction-based features failed to consider the relationships between instructions and are affected by instruction substitutions. In NLP tasks one usually penalizes frequent words by filtering, subsampling, or generalization. For assembly language we find that frequent words improve the robustness of the representation. Graph-based features are oblivious to CFG manipulations. *BinDiff* [31] and *BinSlayer* [12] rely on CFG matching, which is susceptible to CFG changes such as flattening. *Gitz* [23] is another static approach used at the IR level. However, it operates at the boundary of a basic block and assumes basic block integrity, which is vulnerable to splitting. [140] proposes a

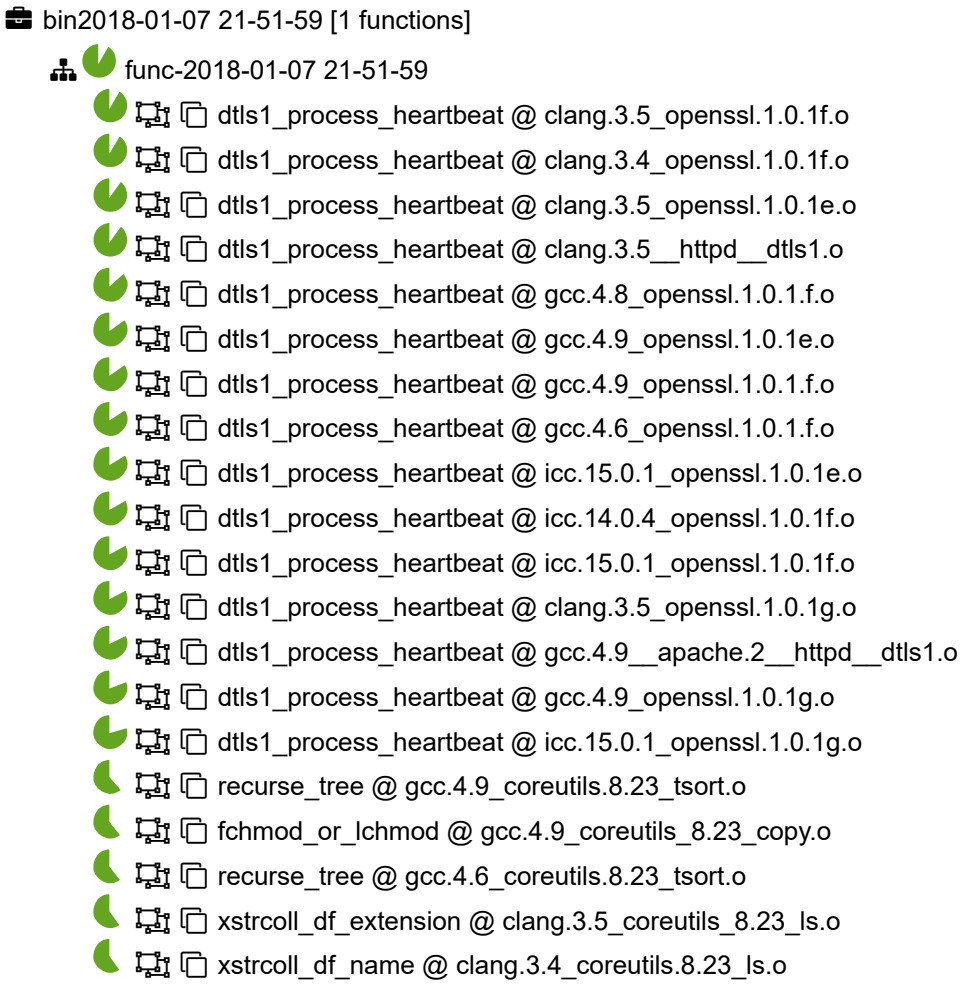

Figure 4.9: Searching the Heartbleed vulnerable function in the vulnerability dataset. The binary name indicates the compiler, library name, and library version. For example, clang.3.5_openssl.1.0.1f indicates that the binary is library *OpenSSL* version 1.0.1f compiled with *clang* version 3.5.

graph convolution approach. It might be able to mitigate graph manipulation. However, it relies on supervised learning and requires a ground-truth mapping of equivalent assembly functions to be trained. *Asm2Vec* enriches static features by considering the lexical semantic relationships between tokens appearing in assembly code. It also avoids direct use of the graph-based features and is more robust against CFG manipulations. However, the CFG is useful in some malware analysis scenarios, especially for matching template-generated and marco-generated functions that share similar CFG structure. One direction is to combine *Asm2Vec* and *Tracelet* [24] or subgraph search [30].

Dynamic methods measure semantic similarity by dynamically analyzing the behavior of the target assembly code. *BinHunt* [43], *iBinHunt* [90], and *ESH* [22] use a theorem prover to verify whether two basic blocks or strands are equivalent. *BinHunt* and *iBinHunt* assume basic blocks integrity. *ESH* assumes strand integrity. They are vulnerable to block splitting. Jiang et al. [62], *Blex* [33], *Multi-MH* [103], and *BinGo* [17] use randomly-sampled values to compare I/O values. Random sampling may not correctly discriminate two logics. Consider that one expression outputs 1 if $v! = 100$; otherwise, 0. Another expression outputs 1 if $v! = 20$, otherwise, 0. Given a widely-used sampling range $[-1000, 1000]$, they have a high chance of being equivalent. *CACompare* follows the similar idea used in [60], [147], [149]. In addition to I/O values, it records all intermediate execution results and library function calls for matching. Using similar experiments to match assembly functions, *CACompare* achieves the best performance among the binary clone search literature at the time of writing this paper. However, it depends on a single input value and only covers one execution path. As stated by the authors, it is vulnerable to CFG changes. *Asm2Vec* leverages the lexical semantic rather than the symbolic relationship which is more scalable and less vulnerable to added noisy logics. As a static approach, *Asm2Vec* achieves competitive performance compared to *CACompare*. *CryptoHunt* is a recent dynamic approach for matching cryptographic functions. It can detect wrapped cryptographic API calls. *Asm2Vec* focuses on assembly code similarity, which is different to *CryptoHunt*.

Source code clone is another related area. *CCFINDERX* [65] and *CP-Miner* [83] use lexical tokens as features to find code clones. Baxter et al. [10] and *Deckard* [61] leverage abstract syntax trees for clone detection. *ReDebug* [58] is another scalable source code search engine. Recently, deep learning has been applied on this problem [138].

4.6 Limitations and Conclusion

Asm2Vec suffers from several limitations. First, it is designed for a single assembly code language and the clone search engine is architecture-agnostic. At this stage, it is not directly applicable for semantic clones across architectures. In the future, we will align the lexical semantic space between two different assembly languages by considering their shared tokens, such as constants and libc calls. Second, the current selective callee expansion mechanism cannot determine the dynamic jumps, such as jump table. Third, as a black box static approach, *Asm2Vec* cannot explain or justify the returned results by showing the cloned subgraphs or proving symbolic equivalence. It has limited interpretability.

In this chapter, we propose a robust and accurate assembly clone search approach named *Asm2Vec* that learns a vector representation of an assembly function by discriminating it from the others. *Asm2Vec* does not require any prior knowledge such as the correct mapping between assembly functions or the compiler optimization level used. It learns lexical semantic relationships of tokens appearing in assembly code and represents an assembly function as an internally weighted mixture of latent semantics. Besides assembly functions, it can be applied on different granularities of assembly sequences, such as binaries, fragments, basic blocks, or functions. We conduct extensive experiments on assembly code clone search, using different compiler optimization options and obfuscation techniques. Our results suggest that *Asm2Vec* is accurate and robust against severe changes in the assembly instructions and control flow graph.

Searching with Obfuscation Options in Tigress									
Name	Heartbleed	ShellshocK	Venom	Clobberin' Time	Shellshock #2	ws-snmp	wget	ffmpeg	avg.
CVE	2014-0160	2014-6271	2015-3456	2014-9295	2014-7169	2014-4877	2014-4877	2015-6826	
# of Positives (k)	15	9	6	10	3	7	3	7	
Encode Literal	100%	77.8%	100%	100%	100%	100%	100%	100%	97.2%
Virtualization	0%	0%	100%	20%	100%	0%	66.7%	0%	35.8%
JIT Execution	53.3%	0%	83.3%	30%	33.3%	0%	0%	100%	37.5%

Table 4.4: True Positive Rate (TPR) of the top-k results searching the obfuscated vulnerable function against the dataset in [22]. k is chosen as the number of ground-truth clones in the dataset. For example, *Venom CVE 2015-3456-4877* has 6 variants in the dataset. By inspecting the top-6 results from *Asm2Vec* we recovered 100% (6/6) for the query with literals encoded, 100% (6/6) for the virtualized query, and 83.3% (5/6) for JIT-transformed query. After applying all the options at the same time, *Asm2Vec* cannot recover any true positives.

5

Going beyond Information Retrieval: Behavior Characterization

In 2013, approximately 82,000 new strains of malware were generated per day. In 2017, this number rose to 285,000 per day, according to the Panda Security Annual Report and the AV-Test Malware Statistics [6], [99], [101]. Out of 15 million new malware collected by Panda Security in 2017, less than 1% occurred more than once [101]. The reason behind this vast amount of new malware is polymorphism and metamorphism, by which malware change themselves constantly after every infection, even though they exhibit the same forms of behaviors [81], [101], [106], [113].

The polymorphism and metamorphism characteristics of modern malware cause static signature-based malware analyses to be ineffective, especially for the packed or obfuscated binaries. In contrast, behavior-based analysis, as a dynamic approach, executes the target binary file in an isolated environment and observes its behaviors. Since the modified binaries still exhibit similar malicious behaviors, behavior-based analysis addresses the issue of packing, obfuscation, polymorphism, and metamorphism. Behavior-based analysis includes an analytic environment. It can be a virtual machine or an emulator. Popular sandbox includes *Cuckoo* [49], *FireEye* [131], *Falcon* [26], *VMRay* [57], and *Joe* [120].

Sandboxes log low-level instructions, system calls, and contextual data associated with each event. However, a high-level malicious behavior is a chain of events. Moreover, the same behavior can be triggered by different events. In order to obtain a high-level understanding of the malware's

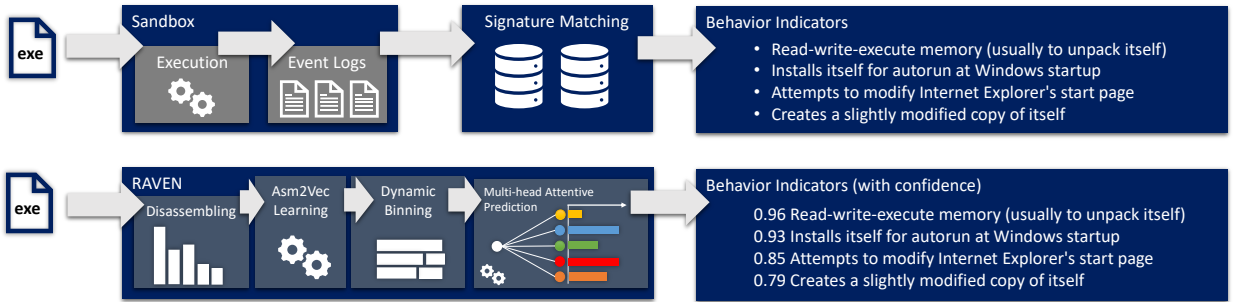

Figure 5.1: Malware behavioral indicator detection process. The sandbox-based analysis typically follows the first pipeline. RAVEN follows the second pipeline.

behavior from the collected logs, most sandboxes develop a signature database to match the low-level events to high level behavioral indicators. For example, the Cuckoo sandbox comes with a community-powered signature database[1]. Some sandboxes, such as VMRay, refer behavioral indicators as threat indicators or risk indicators. Figure 5.1 shows the general process and an example. Behavioral indicators give a global view of the malware's functionalities to security analysts and reverse engineers. It is used to characterize the malware of an unknown family. Behavioral indicators can also be the input to other components in the automated malware analysis pipeline. FireEye uses behavioral indicators to classify malware, in an attempt to reduce false positives [1].

However, malware have become more sophisticated and incorporate anti-sandbox techniques [40]. Sandbox environments can be detected by checking the presence of various signs: remote debugger, injected system hook, amount of memory, disk space, human interaction, special registry values, virtualized device driver, kernel debugger, CPUID, user name, directory name, and suspicious processes. Modularized design and available open source malware generation tools push malware development forward. *Pafish*[2] and *Al-Khaser*[3] are two popular toolkits for sandbox detection. One can easily wrap malicious payload behind the provided environment checker. Some sandboxes, such as Falcon[4], are publicly available as a free service. The malware developers can directly test their malware against the sandbox [48]. After the sandbox environment has been detected, a mal-

[1] https://github.com/cuckoosandbox/community
[2] https://github.com/a0rtega/pafish
[3] https://github.com/LordNoteworthy/al-khaser
[4] https://www.hybrid-analysis.com/

ware can cloak and avoid to unpack, decrypt, or execute itself. The malware can even crash the sandbox host. It is thus very challenging to build a bullet-proof sandbox environment. Building a perfect sandbox resembles the Turing test problem: fooling the malware that a human is using the presented environment.

Anti-evasion techniques have been systematically studied for years. Evasive and anti-evasive techniques on sandbox design ultimately formulate a 'cat-and-mouse' game [15]. Hardware emulation or full OS emulation gives the sandbox full control over program state and can force execution of malicious payloads [92], [143]. However, emulation-based systems are susceptible to timing attacks. Hypervisor-based sandboxes provide minimal overhead but can still be detected. Bare metal architecture is very difficult to detect, but still shows a different power consumption profile [93]. These techniques are complex and require attentive design. They take a significant amount of computational resources and are not scalable to millions of malware. The malware may simply execute stalling code and delay the malicious behaviors [72], [74], [98].

Alternatively, we develop a novel machine learning model, called RAVEN, that directly predicts the malicious behavioral indicators without using a sandbox. As an additional analysis layer, the output of this model can be directly used by security analysts or as the input to other malware analytic pipelines. It requires fewer hardware resources at inference time than using a sandbox. Signature matching can also apply to this problem. However, as we mentioned at the beginning, they become less effective due to packing, polymorphism, and metamorphism. A machine learning-based solution is less susceptible to these issues. If enough data is provided, the model can just learn from the packed byte sequences to find generalizable patterns [16], [107], [109].

In malware classification, each malware only belongs to one family. However, a malicious program could have multiple purposes and, therefore, multiple labels [25]. [126] and [127] also call for a multi-class behavior abstraction to characterize malware. By predicting the behavioral indicators, RAVEN is able to statically provide multiple high-level behavior descriptions. Compared to the existing malware classification problem, it is more challenging in two aspects:

C1 - Sparsity: The number of classes in the literature of malware classification and detection is limited. Most studies only consider fewer than 10 classes [107], [113]. However, the number of malware indicators can easily go beyond a hundred. The distribution is also very sparse. Some

behavioral indicators may only have tens of training samples.

C2 - Open-set Recognition: Most existing studies formulate malware classification as a closed-set classification problem, where a malware always belongs to only one family in the candidate set. Our problem is an open-set recognition problem, which is more difficult. The malware behaviors are either present or absent, and they are not exclusive. It requires *verifying* instead of *classifying* the behaviors.

Static features and models have been widely used to analyze malware [35], [45], [107], [113], [141], [144]. Considering the challenges above, we also aim to address the following problems in existing models:

P1 - Context: Existing byte-based models treat the binary file as a raw byte sequence. However, the same byte will have a different meaning when given a different context. In the text section, it indicates assembly instructions. In the 'rdata' section, it indicates a string character or a constant number. In the 'idata' section, it indicates imported symbols. We argue that modeling binaries as simply byte sequence does not provide sufficient semantic information.

P2 - Feature Engineering: The majority of machine learning models for malware analysis relies on the feature engineering process where security analysts create a set of static and dynamic features according to their expertise. This process represents a binary as a compressed view of the original input space. The representation may miss critical patterns for the target malware under study.

P3 - Memory Footprint: Memory footprint is one of the major challenges for modeling a binary file for an end-to-end learning solution. An executable can have more than one million lines of assembly instructions. Using them as the training samples for a neural network can lead to large intermediate tensors cached for gradient calculation.

To address the above challenges, we design a novel neural network to characterize malware behaviors. Our contributions can be summarized as follows:

- We are among the first to characterize malware behaviors with a machine learning approach. Typically, the behaviors are observed by using the sandbox-based dynamic analyses or matched using signatures. We formulate it as an open-set generation problem (C2).

- We propose a novel neural network architecture that separately models four aspects of a binary file: basic blocks, string constants, imported symbols, and data segments. The neural network addresses P1 by taking a richer contextual information than raw byte sequence. Also, we did not manually engineer features. Instead, this model solves the problem end-to-end (P2).
- We propose a two-level truncated attention architecture. The first level is a self-attention selective mechanism. It builds a compressed shared context representation for all the behaviors. Therefore, behavioral indicators that have only a few training samples can benefit from the others and mitigate the issue of C1. Moreover, this layer greatly reduces the memory footprint of training (P3).
- We introduce sequence fusing and a resettable gate to the recurrent neural network. Combining it with a greedy binning algorithm, we significantly improve the training speed and reduce the memory footprint (P3).
- We collected 45,126 malware and created a malware behavior dataset that will be publicly available to facilitate the research in this area. We benchmark the performance of RAVEN. It shows that RAVEN is accurate and achieves a more than 98% AUROC score. We conducted a case study to investigate its effectiveness.

This paper is organized as follows. Section 5.1 formally formulates the behavioral indicator recognition problem. Section 5.2 describes and justifies our design of the neural network model. Section 5.3 shows our experimental setups and results. Section 5.5 discusses relevant studies. Section 5.6 reflects on the limitations and concludes this paper.

5.1 Problem Definition

In this section, we define the open-set behavioral indicator prediction problem. As mentioned in P2, it is formulated as an open-set recognition problem.

Definition 11 *(Open-set Binary Behavior Indicator Recognition) Given a target malware binary file λ and a set of known malicious behaviors $\mathbb{M}$ observed from a collection of training binaries Λ, the problem is to verify if the given malware λ under analysis will exhibit any malicious behavior $m \in \mathbb{M}$ by giving a confidence value $c_m^{(\lambda)}$. A confidence value of 0 indicates that the malware is unlikely to exhibit the corresponding behavior, while a value of 1 indicates a high likelihood.*

The open-set recognition problem is more challenging than the typical closed-set classification problem used in the malware detection and classification literature. We also note that the behavior verification problem can also be treated as a generation problem typically used in the text summarization literature. The behavior descriptions, such as 'modify the browser homepage', can be used as a target summary to be generated by the model. However, we find that behavioral indicator descriptions are not highly semantically related. Each of them stands out very distinctively to the others regarding their words. Therefore, the descriptions are less generalizable to each other and it is more appropriate to approach it as a recognition problem. In this work, we only focus on Windows x86 or AMD64 PE binaries. However, the proposed neural network can be applied to other binary formats such as ELF.

5.2 Neural Network Architecture

In this section, we formulate our proposed neural network architecture, RAVEN, and describe our idea behind the choice of designs. Figure 5.2 shows the high-level architecture of the neural network. In general, it consists of six components: assembly code modeling, string constant modeling, import function modeling, data segment modeling, pre-selection by self-attention, and behavior-conditioned attention for the final behavioral indicator recognition. We assume that the given binary file λ may contain: code segment b, data segment d, import symbols r, and string constants s; all information can be extracted using a disassembler. We use the IDA Pro 7.1 disassembler[5].

5.2.1 Modeling the Code Segment

We formulate the code segment b as a list of basic blocks, and b_i is one of them. A basic block b_i is a sequence of non-branching assembly instructions $l^{(i)}$, and $l_j^{(i)}$ is one of them. Assembly instruction consists of an operation, such as 'mov' or 'add', and several operands such as registers or memory reference. They can be all considered as tokens. We refer an instruction $l_j^{(i)}$ as a list of assembly tokens $t^{(i,j)}$, and $t_k^{(i,j)}$ is one of them, where j denotes the corresponding instruction and i indicates the corresponding basic block.

We propose to model each basic block using a share-weight recurrent neural network. Fig-

[5] https://hex-rays.com/

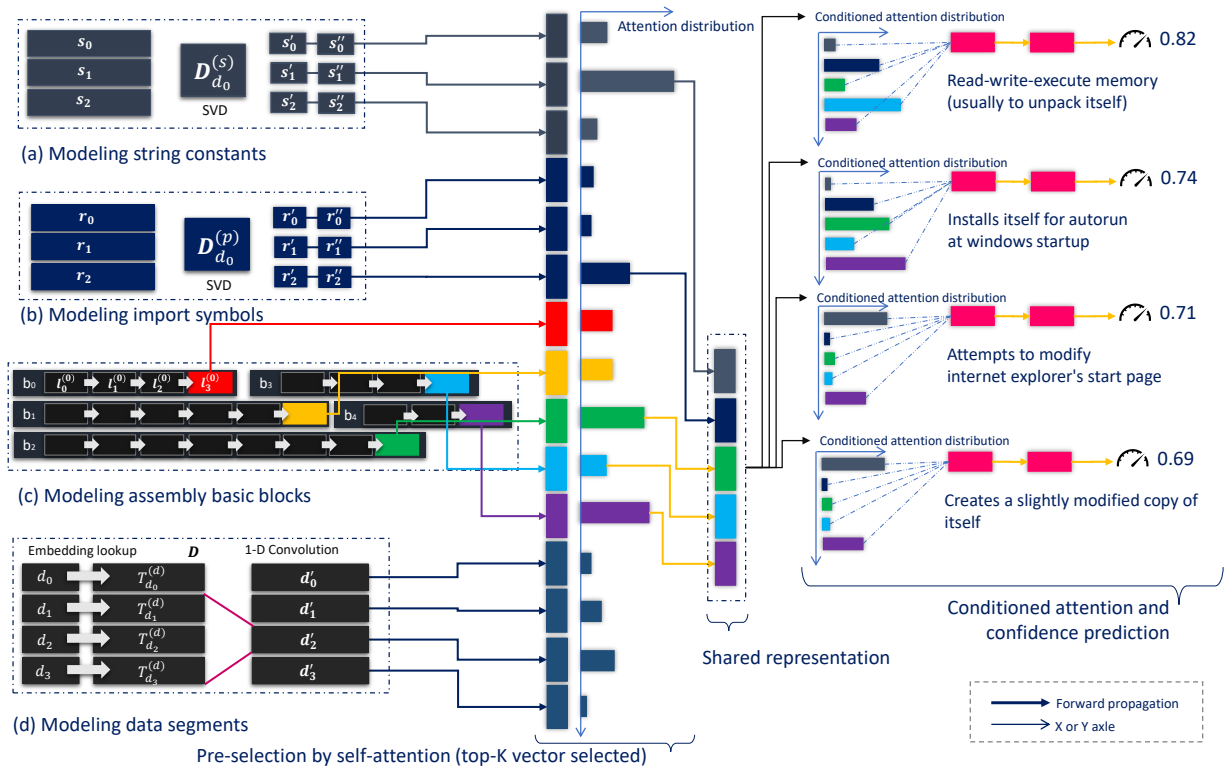

Figure 5.2: The neural network architecture in RAVEN. The input is multiple types of information extracted from a binary.

ure 5.2 (c) corresponds to this component. The general idea is that we first turn each token into a vector by using an assembly code representation learning algorithm *Asm2Vec* [29]. We then represent an instruction as the sum of the vectors of its token. Finally, we model the list of instructions in a basic block with the Quasi-Recurrent Neural Network [13].

Embedding Learning with Customized Asm2Vec

We start from the elements that have the smallest granularity: assembly tokens. Assembly tokens, similar to the word tokens in text data, are discrete and cannot be directly used as the input to the neural network. One needs to encode each unique token as a vector. One typical choice is the one-hot encoding, where each discrete token is represented as a zero-filled vector and only the value at

the index of its ID is one. One-hot encoding assumes that these discrete tokens are independent of each other. However, in assembly code, 'add' will be semantically similar to 'addc'. This issue can be mitigated by using an embedding layer.

In an embedding layer, an assembly token $t_k^{(i,j)}$ is mapped into a context vector by using an embedding matrix $\boldsymbol{T} \in \mathbb{R}^{V \times d_0}$. d_0 is a scalar hyperparameter chosen by the user, and V is the number of unique assembly tokens. One can retrieve a specific token's vector by looking for the row in $\boldsymbol{T}$ that corresponds to the token's integer ID. The relationship between two different tokens can be evaluated by using a distance function on their respective context vectors. $\boldsymbol{T}$ can be treated as the model parameter to be learned from the training data.

However, considering that the number of unique tokens in the assembly code can be large if we factor in all constants, the embedding matrix $\boldsymbol{T}$ can also be very large. A large trainable $\boldsymbol{T}$ introduces a high degree of freedom to the neural network model. With a large input sample that has over 10,000 time stamps, the neural network can easily overfit the training data. This results in a perfect performance on the training dataset and low accuracy on the future unknown data from the very beginning.

Typically, in natural language processing one can use a word embedding learning model, such as *word2vec* [88], or *Glove* [102], to pre-train the embedding layer. These models are designed specifically for text data. We adopt Asm2Vec, proposed in [29]. It is designed specifically for assembly language. We learn $\boldsymbol{T}$ by pre-training an Asm2Vec model on the training data. The original model is designed to jointly learn the vector representations of assembly tokens and the vector representations of the assembly functions. In our problem, we are only interested in the assembly token vectors. Therefore, we change the original model to fit our need by removing the feed-forward paths for assembly function. Figure 5.3 shows the updated model. In general, this model goes through assembly instructions sequence using a sliding window of three instructions. It tries to reconstruct the middle instruction by using its surrounding instructions and maximizes:

$$log\ \mathbf{P}(l_j^{(i)} | l_{j-1}^{(i)}, l_{j+1}^{(i)}) \tag{5.1}$$

Specifically, it predicts the tokens in the middle instruction by using the aggregated vectors of the two neighbor instructions in the sliding window. Formally, it maximizes the following log

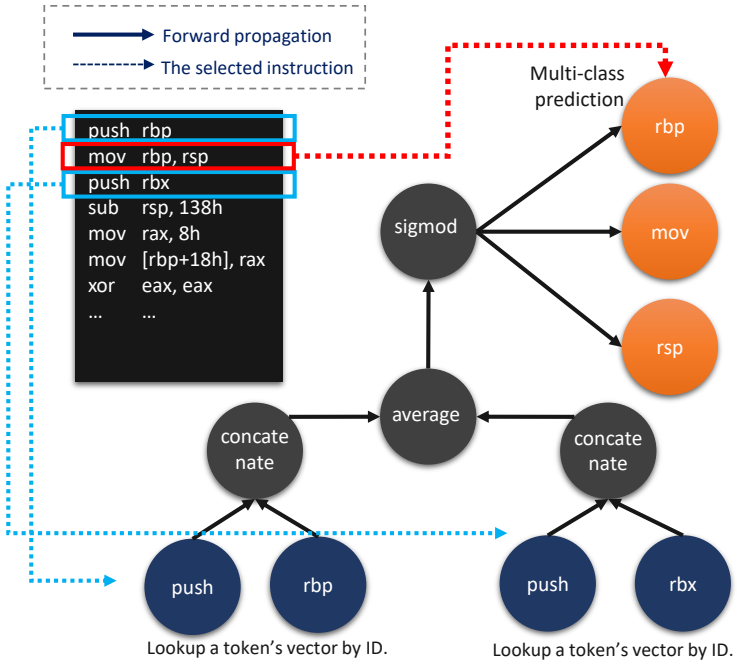

Figure 5.3: The customized Asm2Vec model. We remove all the feed-forward links for assembly functions representation since we only use the tokens representation in this problem.

probability by adjusting $\boldsymbol{T}$:

$$\arg\max \sum_{b_i}^{b} \sum_{l_j^{(i)}}^{l^{(i)}} \sum_{t_k^{(i,j)}}^{t^{(i,j)}} \mathbf{P}(t_k^{(i,j)} | f(t^{(i,j-1)}) + f(t^{(i,j+1)})))$$

$$f(t) = \boldsymbol{T}_{t_0} \frown \sum_{j=1}^{|t|} \boldsymbol{T}_{t_j}$$

Function f takes an assembly instruction, which is a list of assembly tokens t, as input. We first look up the operation t_0's vector and concatenate ($\frown$) it with the sum of the operands' vectors. Given a sliding window of three instructions, the objective is to predict each token in the middle instruction given the sum of two neighbor instructions with function f applied. The sliding

windows are generated by looping through all the basic blocks. We also use control flow graph random walks in [29] to generate sliding windows. The resulting contextual vectors for assembly token capture the semantic information among tokens based on how they are used together. For example, vector operations and SSE registers tend to be similar to each other. Function call *strcmp* is similar to *memcpy*.

After obtaining $\boldsymbol{T}$, we are able to map each unique assembly token in the code segment into a numeric vector. We model an instruction as a d_0-D vector by summing up the vectors of its tokens.

$$\boldsymbol{l}_j^{(i)} = \sum_{t_k^{(i,j)}}^{t^{(i,j)}} \boldsymbol{T}_{t_k^{(i,j)}} \qquad \mathbf{L}_{ij} = \boldsymbol{l}_j^{(i)} \tag{5.2}$$

Given $|b|$ basic blocks in a binary file and each basic block having at most η instructions, we can arrange the instruction vectors into a rank-3 tensor $\mathbf{L} \in \mathbb{R}^{|b| \times \eta \times d_0}$. If a basic block has fewer than η vectors, we pad it with d_0-D zero vectors. Figure 5.4 (a) shows an example of tensor $\mathbf{L}$.

Recurrent Neural Network on Assembly Instructions

The complete code segment can be treated as a sequence of assembly instructions. The semantic of an assembly instruction depends on the context created by its preceeding instructions. To model the code segment, a naive solution is to apply a recurrent neural network that can capture the sequential dependencies. However, for most binaries there are hundreds of thousands of assembly instructions. It is difficult to learn a recurrent neural network on such a long sequence due to the gradient vanishing issue: The gradient becomes smaller and smaller as we travel backward on the sequence for back propagation. Moreover, the original linear layout of assembly instructions on the code segment covers some invalid execution paths; it ignores the boundary of basic blocks.

Instead, we propose to model each basic block separately using the same recurrent neural network. Long Short Term Memory Unit (LSTM) network [46] and Gated Recurrent Unit (GRU) network are two popular choices for modeling sequential data [82]. However, their computation on each time stamp j completely depends on the previous time stamp $j-1$, which limits the parallelism and is slow for long sequences. Instead, we employ the Quasi-RNN [13], which pre-calculates the state update and recurrent gates using a convolutional network. Recall that a basic

block b_i consists of a list of assembly instructions $l^{(i)}$, and by using Asm2Vec, we can represent its j^{th} instruction as a numeric vector $\mathbf{L}_{ij}$. We also refer to the instruction index j as a time step and η, the maximum number of instructions, as the maximum time stamp. The recurrent network can be formulated as:

$$\begin{aligned} \mathbf{Z}_i &= \tanh(\text{Conv1d}(\boldsymbol{W}_z,\ \mathbf{L}_i)) \\ \mathbf{F}_i &= \sigma(\text{Conv1d}(\boldsymbol{W}_f,\ \mathbf{L}_i)) \\ \mathbf{H}_{ij} &= \mathbf{F}_{ij} \odot \mathbf{H}_{i,j-1} + (1 - \mathbf{F}_{ij}) \odot \mathbf{Z}_{ij} \end{aligned} \tag{5.3}$$

Here, $\mathbf{Z} \in \mathbb{R}^{|b| \times \eta \times d_1}$ is the matrix of the proposed new state, where $\mathbf{Z}_{ij}$ is the new proposed state for the basic block i time stamp j. d_1 is another user chosen model parameter. $\mathbf{F}$ has the same shape as $\mathbf{Z}$. It is the matrix of forget gate, controlling how much the previous state should be kept. $\mathbf{F}_{ij}$ is the forget gate vector for the basic block i time stamp j. These two matrices are calculated using a 1-D convolution function [79], denoted as Conv1d, on the input $\mathbf{L}^{(i)}$. $\boldsymbol{W}_z \in \mathbb{R}^{d_0 \times d_1}$ and $\boldsymbol{W}_f \in \mathbb{R}^{d_0 \times d_1}$ are their respective kernels to be learned. tanh denotes an element-wise hyperbolic tangent function, and σ denotes an element-wise sigmoid function.

$\mathbf{H}$ has the same shape as $\mathbf{Z}$. It denotes the new state vectors of all time stamps. At time stamp j of basic block b_i, the network calculates the new state $\mathbf{H}_{ij}$ by combining the previous state $\mathbf{H}_{ij}$ and the proposed new state $\mathbf{Z}_{ij}$ using the corresponding forget gate $\mathbf{H}_{ij}$. $\odot$ denotes an element-wise multiplication.

Sequence Binning and Resettable Recurrent Gate

Modeling each basic block as a separate sequence reduces the gradient vanishing problem and avoids invalid execution paths of the executable. Recall that $\mathbf{L}_{ij}$ denotes the vector for the j-th instruction in basic block i. For basic blocks that have instructions fewer than η, we pad them with zeros. By adopting the tensor notation, we are able to apply the above recurrent neural network in one run for all the sequences. This way, the calculation of different sequences is parallelized. Figure 5.4 (a) shows an example of $\mathbf{L}$.

Basic blocks have different numbers of instructions, ranging from 1 to more than 10,000. Padding a basic block of length from 1 to 10,000 results in a large amount of unnecessary memory

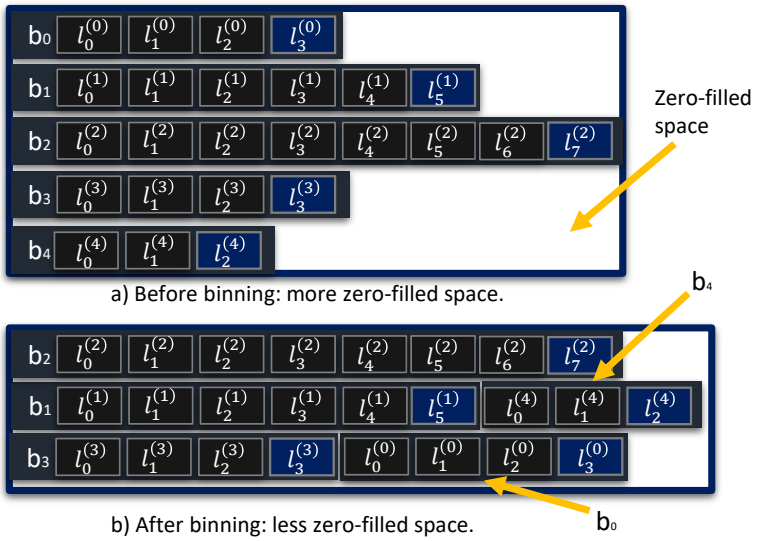

Figure 5.4: The basic block tensor $\mathbf{L}$ before dynamic binning (a) and after binning $\mathbf{L}'$ (b).

space and calculations. The distribution of block length is highly skewed. Especially for obfuscated codes, the maximum number of instructions in a basic block can significantly increase. One can use a sparse matrix to solve this problem. However, it is intrinsically difficult to implement a convolutional layer for sparse matrices.

To address this problem, we introduce a new dynamic binning approach that groups sequences into several bins and reduces the padding space. We use a new hard reset gate to separate the calculations between different sequences. Figure 5.4 (b) shows an example of the binned sequences. It retains a specific number of rows and maintains parallelism for the recurrent neural network. It reduces 87% of padding space compared to the original approach.

The binning approach starts from a fixed number of bins. We choose this value to be 64 by considering the level of parallelism for the recurrent neural network. We then follow a greedy approach to place the basic blocks into these 64 bins. First, we sort the basic blocks according to their respective number of instructions in descending order. We sort the basic blocks in a queue. We pull the first basic block from the queue and place it in the bin that has the least number of instructions. We repeat this process until the queue is empty. By using this simple strategy, the

training time has been reduced by more than five times.

We concatenate the basic blocks in a given bin as a single sequence. We denote the bin index as o and the binned $\mathbf{L}$ as $\mathbf{L}'$. The proposed new state and the forget gate can be calculated by replacing i with o in Equation 5.3:

$$\begin{aligned} \mathbf{Z}'_o &= \tanh(\text{Conv1d}(\boldsymbol{W}_z,\ \mathbf{L}'_o)) \\ \mathbf{F}'_o &= \sigma(\text{Conv1d}(\boldsymbol{W}_f,\ \mathbf{L}'_o)) \end{aligned} \tag{5.4}$$

They are the same as Equation 5.3 except that the first dimension of $\mathbf{Z}'$ and $\mathbf{F}'$ is now 64 instead of $|b|$. The new state can be calculated by:

$$\begin{aligned} \mathbf{H}'_{oj} &= \mathbf{F}'_{oj} \odot \mathbf{H}_{o,j-1} \odot \boldsymbol{G} + (1 - \mathbf{F}'_{oj}) \odot \mathbf{Z}'_{oj} \\ \boldsymbol{G}_{oj} &= \begin{cases} 0 & \text{if } j \text{ is the beginning of a basic block} \\ 1 & \text{otherwise} \end{cases} \end{aligned} \tag{5.5}$$

$\boldsymbol{G}$ is our hard reset gate. If the current time stamp j on bin n denotes the beginning of a basic block, its value is zero, which forces the network to forget the previous state. The last state of a basic block encodes the sequence of its instructions. We extract the last states of each basic block in $\mathbf{H}'$, by which each basic block is represented as a vector. We stack their vectors together as a matrix $\mathbf{B} \in \mathbb{R}^{|b| \times b_1}$. To this end, we have finished modeling the code segments in a binary as the matrix $\mathbf{B}$.

5.2.2 Modeling Strings and Import Symbols

String patterns extracted from the binary can provide useful hints on some malicious behaviors such as sandbox and debugger detection. They have been widely used as rules in signature matching tools such as YARA[6]. Imported library functions can also provide explicit hints on the involved behaviors. For example, some malware use Windows API to modify the registry in order to add themselves to the start-up applications. Some combinations of the import functions, such as *LoadLibrary* and *GetProcAddress*, also signal possible unpacking behavior. Moreover, some malware

[6] https://virustotal.github.io/yara/

use Windows API for encryption and decryption [139].

The one-hot encoding and its variants such as hashing can model the string constants and import functions. This approach has been used in malware detection and classification. It assumes that all string constants and import functions are independent of each other. However, different string constants may share a very similar semantic meaning. For example, 'password' and 'pass_word' may actually be used for the same intention. Some import functions are used interchangeably as well. For instance, 'RegKeyOpen' can be replaced by 'RegKeyOpenEx'. A better approach is to model them as character n-gram vectors. We extract all unique character n-grams where $n \in \{2, 3, 4\}$. 'ex' is an example of a character 2-gram. Each string constant or import function is modeled as a frequency vector on the n-grams. Therefore, the vector of 'RegKeyOpen' will be very similar to the one of 'RegKeyOpenEx'.

Given the set of binary files for training, let S denote the set of all unique strings and R, denote the set of all unique import symbols. First, we build two character n-gram vocabularies, VS and VR, for each of them. Then, we convert S and R into two sparse matrices $\boldsymbol{S} \in \mathbb{R}^{|S|\times|VS|}$ and $\boldsymbol{R} \in \mathbb{R}^{|R|\times|VR|}$. A row in $\boldsymbol{S}$ represents an n-gram frequency vector for a string constant. A row in $\boldsymbol{R}$ denotes a n-gram frequency vector for an import symbol.

By using the character n-gram as feature vectors, we model the relationship between two string constants or two import functions by considering how similar they are. However, the relationship between two assembly tokens in Section 5.2.1 is modeled by how similar the context they have been used for is. For example, the library function call 'exp' is similar to 'pow'. Even though they do not resemble each other, they are used in a similar context.

The number of unique character n-grams is large. Directly using them as the input to the model is infeasible. We seek to use the truncated Single Value Decomposition (SVD) [32] for dimensionality reduction. We choose SVD over the Principle Component Analysis (PCA), since it can be applied efficiently on the sparse matrix [32]. $\boldsymbol{S}$ can be approximated as:

$$\boldsymbol{S} \approx \boldsymbol{U}_{d_0} \boldsymbol{\Sigma}^{S}_{d_0} \boldsymbol{D}^{\intercal}_{d_0} \tag{5.6}$$

d_0 is the number of components. We set it to the same value as the embedding dimension used in Section 5.2.1. By solving the above equation, we can obtain a matrix $\boldsymbol{U}_{d_0} \in \mathbb{R}^{|S|\times d_0}$ with orthogo-

nal columns, a diagonal matrix $\mathbf{\Sigma}_{d_0} \in \mathbb{R}^{d_0 \times d_0}$, and another matrix $\boldsymbol{D}_{d_0}^{(s)} \in \mathbb{R}^{|VS| \times d_0}$ with orthogonal columns.

Given a target binary, we can extract all its string constants, denoted as s, and convert each of them into a character n-gram vector $\boldsymbol{s}$. We respectively apply the equation below to obtain a dimensionality-reduced d_0-D vector $\boldsymbol{s}'$:

$$\boldsymbol{s}' = \boldsymbol{s}\boldsymbol{D}_{d_0}^{(s)} \tag{5.7}$$

We then adopt a linear transformation for $\boldsymbol{s}'$:

$$\boldsymbol{s}'' = \boldsymbol{s}'\boldsymbol{W}_s + \boldsymbol{b}_s \tag{5.8}$$

$\boldsymbol{W}_s \in \mathbb{R}^{d_0 \times d_1}$ and $\boldsymbol{b}_s \in \mathbb{R}^{d_1}$ are the model parameters to be learned. By stacking all the $\boldsymbol{s}'' \in \mathbb{R}^{d_1}$ vectors of the binary file, we can obtain a matrix $\mathbf{S} \in \mathbb{R}^{|s| \times d_1}$.

After extracting all the import symbols r from the target binary and following the same procedure of SVD, we can obtain another matrix $\mathbf{R} \in \mathbb{R}^{|r| \times d_1}$. To this end, given a binary file, we can model its string constants as $\mathbf{S}$ and its import symbols as $\mathbf{R}$.

5.2.3 Modeling Data Segments

Data segments contain important information about the packed code. Generally, it is modeled as a single entropy value or an entropy histogram. Byte n-grams have been found to be effective in analyzing malware as well. We propose to model each data segment as a byte sequence and apply a 1-D convolutional layer. The convolutional layer acts as a scanner over regions of the byte sequence to find any interesting patterns.

Let d denote the data segment and $d_a \in [0, 255]$ indicate one of them. First, we map each byte into a vector representation by using an embedding matrix $\boldsymbol{T}^{(d)} \in \mathbb{R}^{256 \times d_0}$. The byte d_a's vector representation is $\boldsymbol{T}_{d_a}^{(d)}$. $\boldsymbol{T}^{(d)}$ is another variable to be learned. By using an embedding matrix, we can take the relationship between bytes into consideration. We map each $d_a \in d$ by using $\boldsymbol{T}^{(d)}$ and stack them together to obtain $\boldsymbol{D} \in \mathbb{R}^{|d| \times 256}$, where $|d|$ is the sequence length. After, we feed $\boldsymbol{D}$

into a 1-D convolutional layer:

$$\mathbf{D} = \text{Conv1d}(\boldsymbol{D}, nk, ns) \tag{5.9}$$

This convolutional function has three configurable properties: kernel size nk, stride size ns, and filter size d_1. The convolutional function applies a sliding window over the byte sequence. The kernel size controls the width of each scanned window. The stride size controls the number of time stamps to skip when moving the window forward. By choosing a stride of size ns, this function generates $\lceil |d|/ns \rceil$ windows and each has a shape of $[nk, d_0]$. Then, the convolution function applies a tensor-dot transformation over each window with a weight tensor $\mathbf{W}_d \in \mathbb{R}^{nk \times d_0 \times d_1}$. Now, each window has been transformed into a vector of dimension d_1. After applying an element-wise non-linear $ReLu$ function and stacking all the windows' vectors together, we obtain the matrix $\mathbf{D} \in \mathbb{R}^{\lceil |d|/ns \rceil \times d_1}$. To this end, we have modeled the data segment of a binary file as the matrix $\mathbf{D}$. As recommended in [107], we use a fixed kernel size of 500 and a stride of 500.

5.2.4 Pre-selection and Shared Representation

After modeling individual types of information, we now have the following four matrices for the target binary λ:

- $\mathbf{B} \in \mathbb{R}^{|b| \times d_1}$: A matrix for the code segment. $|b|$ denotes the number of basic blocks.
- $\mathbf{S} \in \mathbb{R}^{|s| \times d_1}$: A matrix for string constants. $|s|$ denotes the number of string constants.
- $\mathbf{R} \in \mathbb{R}^{|r| \times d_1}$: A matrix for import symbols. $|r|$ denotes the number of import symbols.
- $\mathbf{D} \in \mathbb{R}^{\lceil |d|/ns \rceil \times d_1}$: A matrix for the data segment. Its first dimension corresponds to the number of sliding windows.

By concatenating them along the second dimension, we have a unified matrix:

$$u = |b| + |s| + |r| + \lceil |d|/ns \rceil \tag{5.10}$$

$$\boldsymbol{U} \in \mathbb{R}^{u \times d_1} \tag{5.11}$$

Each row of the matrix represents a specific piece of information extracted from the binary. Next, we reduce this gigantic matrix into a single vector for the final prediction.

There are several available options. We can apply a convolutional layer or an attention layer to combine them into a single d_1 vector. Recurrent neural networks can achieve this as well, but there is no sequential relationship shown along the first dimension. We tried all of these solutions, but each generated a massive memory footprint during training due to the large input matrix. We consider all the binaries that are less than 6 MB, which is significantly larger than the binary files used in recent machine learning studies [107]. None of these options can only use a single GPU, even for only the forward pass on all our binaries. Having a small memory footprint enables massive distributed deployment for inference. It also facilitates the training speed across multiple GPUs by using data parallelization. Therefore, we seek to develop a specialized two-level attention mechanism that fits into our design. The attention layer provides a weighted average over a sequence of vector and the weights automatically learned based on the training dataset.

On the first level of the attention layer, we only select the top-k_1 rows before conducting any further analysis on a specific malicious behavioral indicator. We base our design on the following observation. For any malicious behavior that one tries to predict, there are some irrelevant patterns that can always be skipped by looking at the binary file. In a real-life malware analysis task, a large block of zeros, large blocks of 0xff, regular program entry point patterns, harmless imported symbols, or nop sleds are of less interest for the analyst. The same applies for a machine learning model. Most patterns from the input binary are not useful regarding all the malicious behaviors to be analyzed. By selecting the top-k_1 rows from U, we essentially are asking the model to identify the most useful patterns by just looking at the currently presented information. We set k_1 as 4096. This layer is backed by a self-attention mechanism [135]:

$$\boldsymbol{g} = (\boldsymbol{U}\boldsymbol{W}_c + \boldsymbol{b}_c) \cdot \boldsymbol{v}$$

$$\boldsymbol{p}_q = \frac{e^{g_q}}{\sum_k^u e^{g_k}} \qquad \boldsymbol{U}' = \text{top}(\boldsymbol{p}, k_1, \boldsymbol{U} \cdot \boldsymbol{p})$$

$\boldsymbol{U}$ is our concatenated matrices of different segments. $\boldsymbol{W}_c \in \mathbb{R}^{d_1 \times d_2}$, $\boldsymbol{b} \in \mathbb{R}^{d_2}$, and $\boldsymbol{v} \in \mathbb{R}^{d_2}$ are the parameters to be learned. $\cdot$ denotes a broadcasted dot product. $\boldsymbol{g}$ is of size u. d_2 is another configurable parameter of our model. After getting $\boldsymbol{g}$, we apply a soft-max function to get a probability

distribution. The resulting vector $\boldsymbol{p} \in \mathbb{R}^u$ represents the weights for reference. It sums up to 1. Then, we apply a function to extract the row vectors from $\boldsymbol{U}$ that are ranked in top-k_1 according to their respective weights in $\boldsymbol{p}$. The extracted matrix $\boldsymbol{U}' \in \mathbb{R}^{k_1 \times d_1}$ will be used for further analysis. By using such a filtering mechanism we significantly reduce the memory footprint. It allows the calculations presented in the next section to not grow with the size of the binary. We name $\boldsymbol{U}'$ as a shared representation for all behaviors.

5.2.5 Behavior-Conditioned Attention Mechanism

In this section, we predict each individual malicious behavioral indicator $m \in \mathbb{M}$. Given a pre-selected shared representation for binary λ, our goal is to predict a confidence value $c_m^{(\lambda)}$ to indicate the possibility of observing a specific behavior m. Different behaviors may focus on different patterns. In this layer, we also include an attention mechanism to select another top-k_2 vector from $\boldsymbol{U}'$ where $k_2 < k_1$:

$$\boldsymbol{g}^{(m)} = (\boldsymbol{U}'\boldsymbol{W}_m + \boldsymbol{b}_m) * \boldsymbol{v}_m$$
$$\boldsymbol{p}_q^{(m)} = \frac{e^{g_q}}{\sum_k^u e^{g_k}} \qquad \boldsymbol{U}'_m = \text{top}(\boldsymbol{p}^{(m)}, k_2, \boldsymbol{U}' \cdot \boldsymbol{p}^{(m)})$$

The selection mechanism is the same as the previous one. However, we use the behavior-specific weights $\boldsymbol{W}_m$, $\boldsymbol{b}_m$, and $\boldsymbol{v}_m$. They are the parameters to be learned. This way, we provide flexibility for the model to learn a behavior-specific attention. Then, we simply apply a max-pooling layer and a logistic layer for the final prediction:

$$c_m^{(\lambda)} = \text{sigmoid}(\text{reduce\_max}(\boldsymbol{U}'_m, 1) \cdot vw_m + b) \tag{5.12}$$

The reduce_max function selects the maximum value along each column and results in a vector of dimension k_2. Vector $\boldsymbol{w}_m \in \mathbb{R}^{k_2}$ and scalar b are two other behavior-specific weights to be learned. Here $\cdot$ denotes the vector dot product. Finally, the sigmoid function re-scales the output to $[0, 1]$. By following the same attention mechanism, we obtain all the predicted confidence values for binary λ: $\{c_m^{(\lambda)} | m \in \mathbb{M}\}$.

Let Λ denote the training dataset and $y_m^{(\lambda)}$ denote the ground-truth label for binary λ behavior

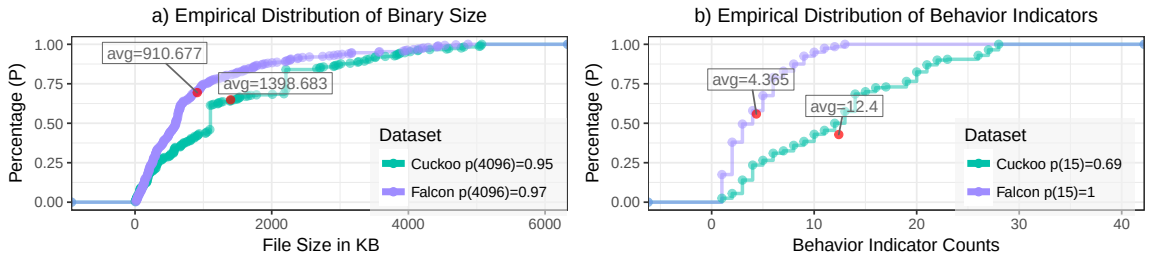

Figure 5.5: The empirical distribution of the size of binary files and the number of behavior indicators per-binary.

m. The final training objective is to minimize the logistic loss over all malicious behaviors:

$$\arg\min \sum_{\lambda}^{\Lambda} \sum_{m}^{\mathbb{M}} -y_m^{(\lambda)} \log(c_m^{(\lambda)}) - (1 - y_m^{(\lambda)}) \log(1 - c_m)$$

This objective function is optimized by using the stochastic gradient descend algorithm with the *Adam* [70] optimizer. Without the loss of generality, the formations in this section are based on a single behavioral indicator. Under the hood, we use the tensor notation to parallelize the calculation of all the behavioral indicators in one pass. We implemented RAVEN by using the *TensorFlow*[7] auto-gradient framework. RAVEN is open source[8].

5.3 Experiments

The experiments consist of two parts. In the first part we develop two labeled datasets for the behavioral indicator recognition problem and evaluate RAVEN against 6 state-of-the-art binary modeling approaches. In the second part we collect an additional known family of malware and conduct a case study with a RAVEN model trained with the Cuckoo dataset. The experiments are all carried out on a Windows server equipped with two Xeon E5-2697 CPUs (36 cores), 384 GB memory, and four Nvidia Titan XP GPU cards. Each card has 12 GB memory. RAVEN requires only one GPU to be trained, consuming less than 11G memory even with a binary file larger than 6 MB.

[7] http://www.tensorflow.org
[8] Available at [URL is temporally hidden according to the double-blind submission policy].

Family	Files	Family	Files	Family	Files
fareit	5585	loadmoney	272	somoto	110
installmonster	4177	autoit	270	prepscram	109
razy	3648	mikey	253	upatre	105
zusy	2237	gamarue	225	mywebsearch	104
gandcrab	829	firseria	224	safebytes	103
outbrowse	799	imali	212	virut	94
installerex	777	zbot	161	delf	92
startsurf	669	tinyloader	160	nakoctb	91
downloadadmin	583	softpulse	145	midie	90
multiplug	499	hotbar	139	wannacry	87
soft32downloader	472	speedingupmypc	139	istartsurf	87
installcore	324	dlhelper	122	noon	85
bundlore	304	downloadguide	115		
emotet	303	vittalia	110		

Table 5.1: Top-40 malware families in the Cuckoo dataset. There are total 1091 unique families.

5.3.1 Dataset Development

To the best of our knowledge, we present the first work that tries to understand malware behavior using a machine learning model instead of a sandbox. There is no available dataset for benchmark. We curate and release two labeled datasets for this problem to facilitate future studies in this area[9]. As defined in Section 5.1, the problem involves a collection of binaries Λ with their known behaviors as the basis of knowledge to predict the behaviors of an unknown binary λ. To develop a model and solve this problem we need to build a large collection of malware samples annotated with their malware behaviors.

It is very challenging to find a data source that can provide reasonably accurate behavior descriptions. At first, we looked into the malware encyclopedia for detailed descriptions of malware families. Given a collected known malware sample, we can first look up its malware family. Then, we search for the corresponding malware description against the malware encyclopedia. To implement this solution, we build our own malware encyclopedia by combining all the available ones online from Kapersky, ESET, Sophos, Symantec, and Microsoft. Given a malware family, it can

[9]Dataset available at [URL temporarily hidden according to the double-blind review policy].

output a description of the malware's high-level behaviors. However, this approach turns out to be infeasible. We find that less than 10% of the malware families in our collected dataset have a detailed analysis report. Most of them are generated using generic templates and do not include specific behaviors. Instead, we decide to characterize the malware behavior by ourselves using sandbox environments. We use two different sandboxes and curate two different datasets.

The Cuckoo Dataset

For this dataset, we retrieve the malware's behavioral indicators using our own in-house Cuckoo-based sandbox cluster. We first gather a collection of binaries from *MalShare* and *VirusShare* between January and May 2018. We only consider files that are smaller than 6 MB, which already account for more than 95% of the binaries that we observe. It is noted that, in typical malware detection or malware classification studies, the maximum file size is less than 4 MB [77], [107], [113]. Figure 5.5 characterizes the file size distribution. We filter out the benign binaries using the binaries' corresponding *VirusTotal* online reports. VirusTotal scans a binary file using multiple anti-virus engines. We include the binaries that are identified as malicious by more than 5% percent of the anti-virus engines. This way, the binary files in the dataset have a good chance to show several malicious behaviors. The resulting 36,730 binaries are fed into the sandbox environment for analysis. We denote this dataset as the Cuckoo dataset. Table 5.1 lists the top-40 malware families in it. There are 1,091 unique families. It's size is 349 GB, including all the extracted information.

Our mini Cuckoo-cluster consists of 5 virtual machines (VMs) connected through a virtualized network. Each has a varying amount of memory and processing cores. The host machine of the virtualized environment dispatches malware samples to the virtual machines for analysis. Cuckoo is an open source sandbox software that relies on a hood-based analytic environment. It uses system hooks to observe the malware's behaviors. After logging all the low-level activities, it uses a community-powered signature database to generate a list of high-level behavioral indicators. Figure 5.1 shows several examples. Given the fact modern malware mostly include evasive techniques, we apply the following steps to minimize the chance of a successful evasion.

- We use the *Pafish*[10] and the *Al-Khaser*[11] toolkits to analyze our virtualized environment. We fix

[10] https://github.com/a0rtega/pafish
[11] https://github.com/LordNoteworthy/al-khaser

Type	# of binaries	Specific technique, target, or resource.
antivm	28893	vbox, network, virtualpc, disk, queries, shared, vmware, generic, sandboxie, firmware, vpc, memory
packer	18792	polymorphic, entropy, vm_protect, upx
peid	6361	peid detected_packer (static)
antisandbox	9015	idle_time, restart, file, threat_track, sunbelt, unhook, foreground_windows, sleep, mouse, cuckoo, joe
antiemu	3807	wine
antidbg	339	devices, windows
antiav	395	detect_reg, avast, service_stop, detect_file bitdefender
antianalysis	40	detect_file

Table 5.2: Statistics of the grouped evasive behavioral indicators observed in the Cuckoo dataset. See Appendix D for the full description of each indicator.

Category	# of binaries	Technique, target, or resource.
allocates	29984	rwx, execute remote process
network	21887	wscript, bind, icmp, tor
injection	20274	create remote thread, run pe, modifies memory, write memory
exe	18626	drop exe
stealth	14855	hiddenfile, hidden icons, hidden extension, hide notifications, window, system procname
modifies	13101	desktop wallpaper, boot config, proxy autoconfig, security center warnings, proxy wpad, zoneid, certificates
dumped	11254	buffer2, buffer
suspicious	9246	powershell, process, write exe, command_tools
recon	9222	fingerprint, beacon, systeminfo, programs
persistence	8857	registry powershell, registry javascript, autorun, registry exe, ads
infostealer	8812	ftp, mail, keylogger, im, bitcoin, browser
console	8087	output
deletes	7462	executed files
browser	6666	startpage, security
queries	6385	programs
banker	3483	bancos, zeus p2p, spyeye mutexes
locates	2524	sniffer, browser
privilege	2427	luid check
disables	2056	windowsupdate, system restore, app launch, browser warn, spdy ie, proxy, security
creates	1771	largekey, shortcut, service, doc, hidden file

Table 5.3: Top-20 groups of dynamic malicious behavior indicator in the Cuckoo dataset. We group behavior indicators into categories and rank by their frequency. See Appendix D for the full description of each descriptor.

the detected flags if possible. However, not all the flags are fixable, as they are limited by the Cuckoo sandbox itself. For example, we cannot remove its API hood. Moreover, we are not able to introduce any human interaction.

- We create each virtual machine using a recent copy of a personal computer's file system. The folder naming and structure resemble a workstation used by an actual human. Also, the last modified time stamp for the user files are recent. We even include faked links in the user's recent file folder.
- Malware can check the active running processes to detect the VM environment. Therefore, we run multiple random programs and services in the virtual machine before each analytic task.
- Malware can detect the VM environment by checking the amount of assigned hardware resources, since VM typically is assigned less memory and storage than an actual workstation. We mitigate this issue by assigning the VM different resources. The smallest VM has only 2 core, 1G RAM, and 10G storage. The biggest VM, in contrast, has 4 core, 8 GB RAM, and 128G storage. We conduct the analysis in several rounds. If a malware only exhibits few behaviors in a VM, we move it to the queue of the others.
- Malware can also sleep and wait for a specific period of time for evasion because most analytic VMs are given only 1 to 5 minutes of execution time. Instead, we conduct the analysis in multiple rounds, from 1 minute to 1 hour.

This sandbox environment is not 100% bulletproof. However, the resulting analytic reports show that our setups are good enough to detect a variety of malicious activities and evasive techniques. We only consider the behavioral indicators that have more than 20 samples. There are a total of 139 unique malicious behavioral indicators. Table 5.3 lists the top-20 types of dynamic behaviors detected by the Cuckoo sandbox. Table 5.2 summarizes all the evasive techniques. The malware in this dataset exhibits a significant amount of evasive techniques. 80.1% of the binaries demonstrate at least one evasive technique. 4,617 binaries create a modified copy of itself, as indicated by the polymorphic behavioral indicator. Figure 5.5 characterizes the distribution of per-sample behavioral indicators. Refer to Appendix D for a full list of included behavioral indicators.

We further conduct analyses to guarantee that the majority of the annotated behaviors are detected based on dynamic activity logs rather than other static features. We assume that we have an ideal evasive technique that can detect any kind of sandbox. We equip all the malware in our

Family	Files	Family	Files	Family	Files
fareit	324	icloader	25	tiggre	14
emotet	117	mywebsearch	25	xtrat	13
gamarue	100	darkkomet	24	nymaim	13
autoit	55	genkryptik	23	trickbot	12
zusy	51	noon	23	gamehack	12
razy	46	zlob	22	farfli	11
gandcrab	43	swrort	21	dimnie	11
installcore	39	vobfus	18	ekstak	11
khalesi	37	nakoctb	17	occamy	10
ursnif	34	dealply	15	icedid	10
delf	34	banload	15	crysis	10
zbot	30	chapak	14	screenmate	10
ursu	30	driverpack	14		
yakes	29	upatre	14		

Table 5.4: Top-40 malware families in the Falcon dataset.

dataset with this technique. They will refuse to execute if they detect our sandbox. Instead of working out a wrapper for the malware, we simulate this ideal technique by modifying the sandbox: the sandbox will skip the dynamic execution step in the analysis. We collect all the reports after the modification. From these reports, we can recover less than 10% of the originally annotated behaviors, such as the packers reported by static analysis tool *PEiD* [12] and the patterns matched by Yara rules.

The Falcon Dataset

Following the same idea, we developed the second dataset using a different sandbox called Falcon, which is an industrial sandbox that has been hardened for years against evasive techniques. Unlike the Cuckoo sandbox's hook-based logging approach, Falcon observes the malware's behavior from the kernel space. It is more difficult to detect. We assume that the identified malware behavior is more accurate. *Hybrid Analysis* is an online binary analytic platform that is powered by the Falcon sandbox. It directly comes with the behavioral indicator reports generated by the Falcon sandbox. By using its online RESTful API, we are able to retrieve recent binary samples and analytic reports.

[12] https://www.aldeid.com/wiki/PEiD

Category	# of binaries	Technique, target, or resource.
fingerprint	5902	privilege, IDE drive, IP, installation date, username, scsi, language, GUID
remote	5618	listen, registry, internet-related hooks, steal password, RDT
persistence	4461	schedule, inject, startup, auto execute, startup repair, restore, take ownership, firewall
network	4290	traffic
evasive	2153	API, forensic tools, vm, timeout, machine name, decompressor, antivirus, sleep, security services, regsvnum
spyware	3045	keyboard, clipboard, input devices, post files
spreading	1833	mountpointmananger, drive letters, network ARP, workgroup share, browser
stealer	665	mail, tfp, IM
credential	486	password, browser
ransomware	166	ransomware, snapshots, tor, wallpaper, globeimposter
adware	44	start page, anti-adware tools
exploit	12	escaped byte string, shell code
banking	2	web certificate

Table 5.5: Statistics of the grouped dynamic malicious behavior sindicator in the Falcon dataset. We group behavior indicators into categories. See Appendix E for the full description of each individual behavioral indicator.

We only collect online samples that are flagged as malicious. In total, we gather 8,405 binaries and their reports. We denote this dataset as the Falcon dataset. Its size is 81 GB, including all the extracted information. It is much smaller than the Cuckoo dataset, since we are limited by the access rate. With far less training samples, it is more difficult to statically predict malicious behaviors in this dataset. Figure 5.5 compares the file size distribution. On average, the binary size of the Falcon dataset is smaller than that of the Cuckoo dataset.

Table 5.4 lists the top-40 malware families in the Falcon dataset. It shares some families with the Cuckoo dataset but with different rankings. The Falcon analytic reports contain a section named risk assessment that lists several malicious behaviors exhibited by the malware. We use them as the label for the dataset. We summarize all the malicious behaviors in Table 5.5. We group them into the available behavior categories in the report. Figure 5.5 also compares the behaviors per-sample count against the Cuckoo dataset. On average, less malicious behaviors are reported. In the Falcon dataset, 25% of malware exhibit evasive techniques, which is also lower than the Cuckoo dataset.

The Cuckoo Dataset												
	Validation Set						Testing Set					
Baselines	FPR	Precision	Recall	Brier	F1	AUROC	FPR	Precision	Recall	Brier	F1	AUROC
Byte n-gram	.008	.617	.441	.034	.514	.857	.008	.597	.431	.034	.501	.849
Opcode n-gram	.009	.665	.385	.025	.488	.917	.009	.625	.386	.026	.477	.914
Opcode n-gram+s	.009	.601	.356	.026	.447	.914	.009	.598	.361	.027	.450	.912
Opcode n-gram+s+r	.009	.604	.361	.026	.452	.914	.009	.607	.370	.027	.460	.911
DNN2dim	.020	.110	.084	.057	.095	.645	.025	.084	.079	.063	.081	.587
GIST	**.000**	.670	.512	.021	.581	.793	**.000**	.642	.518	.021	.573	.802
RAVEN*	.008	**.914**	**.852**	**.017**	**.882**	**.989**	.009	**.910**	**.846**	**.017**	**.877**	**.988**

The Falcon Dataset												
	Validation Set						Testing Set					
Baselines	FPR	Precision	Recall	Brier	F1	AUROC	FPR	Precision	Recall	Brier	F1	AUROC
Byte n-gram	.020	.357	.195	.071	.253	.659	.022	.249	.133	.074	.174	.632
Opcode n-gram	.017	.203	.068	.057	.101	.606	.016	.229	.066	.057	.102	.599
Opcode n-gram+s	.017	.236	.073	.057	.112	.621	.016	.244	.073	.056	.112	.606
Opcode n-gram+s+r	.017	.236	.073	.057	.112	.621	.016	.244	.073	.056	.112	.606
DNN2dim	.019	.071	.051	.061	.059	.507	.025	.113	.065	.060	.082	.532
GIST	**.000**	.273	.132	.062	.178	.614	**.000**	.251	.118	.064	.161	.598
RAVEN*	.020	**.697**	**.460**	**.052**	**.554**	**.910**	.019	**.717**	**.470**	**.050**	**.568**	**.910**

Table 5.6: Behavior recognition benchmark. Evaluated using *False Positive Rate (FPR)*, *Precision*, *Recall*, *Brier Score (Brier)*, *F-measure (F1)*, and the *Area Under the Receiver Operating Characteristic Curve (AUROC)* metric. *RAVEN* is our proposed model. We reported the performance on both the validation set and testing set.

5.3.2 Quantitative Benchmark

With the aforementioned datasets, we benchmark the performance of RAVEN against several state-of-the-art approaches for binary modeling. For each dataset we use stratified sampling to split it into three parts: the training set (80%), the validation set (10%), and the testing set(10%). The training set is used to train a given model. The validation set is used to fine-tune different configurations of a given model. The testing set is solely used for evaluation. For all the baselines, we report both the performance on the validation and testing set.

As mentioned in the introduction and the problem definition, the problem is formulated as a multi-label recognition problem. The model should give a confidence value $\in [0, 1]$ for each malicious behavior. Even though the baselines below are originally designed for malware classification or malware detection, in general they can be applied on this problem. We include a diverse set of static features.

- *Byte n-gram*. Byte n-gram is a simple but effective choice for binary modeling. This method

treats a binary file as a byte sequence and counts the frequency of any possible unique combination of n consecutive bytes. It has been used in malware classification [16], [117], [145]. [109], [145] find that 6-grams converge faster and perform the best. We fine-tune n using our validation set and reach a similar observation in the experiment. Following [109], we select the top 200,000 byte 6-grams ranked by frequency as features and apply a logistic model for recognition. These grams are ranked using a database. We build a logistic model for each indicator. We tune the C regularization strength on the validation set for better generalizablity.

- *GIST*. A binary file as byte sequence can be modeled as a gray scale image. Studies have applied this approach for malware triage [71] and malware classification [95]. We use the model in these two studies as a baseline. We determine the image width according to the byte sequence length and model a binary file as a gray scale image. Then, we extract the GIST image descriptor from the gray scale images. A k-Nearest-Neighbor model is used for final recognition. We choose k by tuning the model's performance on the validation set. We use a different model for each behavioral indicator.
- *Opcode n-gram*. Opcode denotes the operation of a single instruction. n-grams have been widely used in malware detection [116], [121] and classification [3], [142], [148]. It is shown to be the predictor against polymorphism and metamorphism [11]. We select top-k opcodes ranking by their frequency and use logistic regression for the final recognition. We tune k and the regularization strength C on the validation set.
- *Opcode n-gram+s*. The same as the last one, except we add another top-k string constant into the feature set. We tune k and the regularization strength C on the validation set.
- *Opcode n-gram+s+m*. The same as the last one, except we add another top-k import symbols into the feature set. We tune k and the regularization strength C on the validation set.
- *DNN2dim*. [118] is another neural network-based approach for malware detection. It models different segments of a binary file into a single vector by feature engineering. It includes static features extracted from the PE header, import symbols, string constants, and byte sequence by using hashing and histograms. We use the validation to control the training termination criteria with an adaptive learning rate.

For RAVEN, we fix d_0 and d_1 to 128, which is a typical choice of the number of hidden units in a neural network model layer. We use the validation set as the stopping criteria for training and use

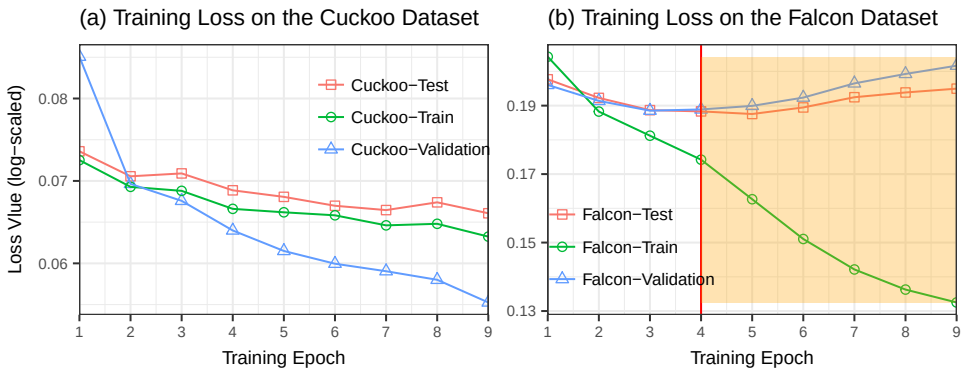

Figure 5.6: The loss values across RAVEN's training epochs. The orange area indicates a training set overfit.

an adaptive learning rate. It is a widely used training practice for neural networks [79]. We do not tune any other settings. Table 5.6 shows the benchmark results. We evaluate each model using *False Positive Rate (FPR)*, *Precision*, *Recall*, *Brier Score (Brier)*, *F-measure (F1)*, and the *Area Under the Receiver Operating Characteristic Curve (AUROC)* metric. These are typical metrics used in recognition tasks [79]. False positives rate and Precision both measure the accuracy of the positive predictions. We prefer a low FPR and a high precision value. Recall measures the percentage of the true positive labels, in our case the true behavioral indicators, that can be recovered. The Brier score is the mean square difference between the predicted probability and the actual label. AUROC measures the overall trade-off between true positive and false positive when one changes the threshold on the predicted probability.

In general, RAVEN achieves the best performance. It achieves .98 AUROC on the Cuckoo dataset and .91 AUROC on the Cuckoo testing dataset. It is able to retrieve more than 87% of dynamic behaviors with only a 0.09 false positive rate and .91 precision on the Cuckoo dataset. On the Falcon dataset it does not perform as well as on the Cuckoo dataset. It recalls 47% of behavioral indicators with a false positive rate of 0.02. Nevertheless, it still performs better than the baselines, achieving the best recall, precision, brier score, and AUROC score.

Consider the AUROC metric, which measures the overall performance. Byte n-gram ranks second in the Falcon dataset and Opcode-based methods rank second in the Cuckoo dataset. These results are consistent with the observation that opcode is a good predictor for polymorphism and

metamorphism [11]. However, byte n-gram tends to focus more on capturing string patterns than instructions [109], and the Cuckoo dataset contains more evasive behaviors than the Falcon sandbox. RAVEN combines information from both sources. GIST does not perform well but achieves the lowest false positive, less than 0.001. It should be noted that GIST does have false positives since the precision is not one. DNN2dim does not perform as well as the others. Its feature representation compresses too much information.

Overall, in the Falcon dataset all the baselines perform worse than they do in the Cuckoo dataset. We suspect that it is due to a smaller dataset size and a smaller behavior-per-sample ratio. The number of positive samples is small, but an input contains massive information. All the models tend to overfit the training dataset at the very beginning. Take RAVEN, for example. It starts to overfit the training dataset and stops generalizing the validation and testing set from the fourth epoch (see Figure 5.6).

5.4 Case Studies

We collect three variants of the *GandCrab* malware family for a case study. GandCrab is a family of ransomware widely spread in 2018. It is distributed through multiple spreading vectors such as spam emails, exploit kits, and malicious websites. These three variants are packed with a customized packer used in the GandCrab family. They also contain sandbox evasion techniques. They try to delay executing its malicious payload by calling Windows APIs in a loop. This simple approach is effective against sandboxes because they have limited computational resources, and the loop calculation is slow. Its string constants have also been obfuscated. This case study evaluates RAVEN's robustness against custom-packed binaries.

These malware samples are not used to train our model. We reuse the trained RAVEN model on the previous Cuckoo dataset. We build a web-based analytic service on top of the trained RAVEN model. We feed these three samples into RAVEN for analysis. Figure 5.7 shows the predicted behaviors in three radar charts, respectively for each sample. The blue area represents the observed behaviors in a sandbox, and the orange area represents the predicted behaviors. The red circle denotes the decision boundary of a 0.5 confidence. Outside the red circle, the area covered with blue but not orange represents a false negative. The area covered with orange but not blue indicates

a false positive.

According to the chart, RAVEN is able to detect all these three samples generated cryptographic keys using related Windows API, as indicated by the 'generates_crypto' indicator. It also detects that these three samples exhibit an unpacking behavior by allocating read-write-execute memory (denoted by the 'allocate_rwx' indicator). RAVEN detects that samples (b) and (c) try to delay the analysis task (denoted by the 'antisandbox_sleep'). According to the reports generated by the sandbox, sample (a) does not delay the analysis task, and RAVEN does not report a false positive as well. RAVEN also detects that these three samples check various environment flags such as memory, disk size, and CPU name. RAVEN also correctly predicts that these three samples modify network proxy for traffic interception.

There are two false predictions. RAVEN fails to predict the auto-run functionality of sample (a). It also predicts that sample (b) generates some ICMP traffic, which is not shown in the sandbox logs. Except for these two false positives, RAVEN's prediction is accurate, even though these samples are packed and obfuscated.

5.5 Related Works

Behavioral Analysis Dynamically generated behavior events have been widely used for malware analysis. On the malware family classification problem, [110] proposes to use API 4-grams. [111] uses a bag-of-words model on event logs. [112] proposes malware instruction set *MIST* and uses n-gram modeling. On the malware detection problem, [2] models API as a spatial-temporal transit matrix. [132] employs one-hot encoding of API sequence. [41] models behavior report as a term dictionary. [20] uses API n-grams. These methods classify malware into existing categories. However, as discussed in [25], malware could have more than one purpose and should thus have more than one label. [126] and [127] also call for a multi-class behavior abstraction to characterize malware. RAVEN follows a similar direction to study malware behavior. Instead of predicting the malware families, it tries to predict the high-level behaviors.

Static Analysis Various static features have been developed on malware for triage, detection, or classification. There are three typical models: byte sequence, opcode sequence, and image. [16], [117], [145] model binary as byte n-grams. [71], [95] model binary as image and extract the GIST

image descriptor. [50] treats the image as an entropy graph. [51], [52] model opcode sequence as images. [121], [148] use opcode n-grams. [142] models opcode sequence as bag-of-word. [116] combines dynamically generated opcodes and static opcodes. [107] models binary as a byte sequence. [108] models the header information as byte sequence. [118] models a binary by compressing the information from different sources into a single vector using hashing and histograms. RAVEN studies a different problem and models different parts of a binary file by considering their contexts.

Anti-Evasive Techniques The research on malware evasive techniques and anti-evasive techniques has a long history. [15] provides a comprehensive survey on related techniques and their trade-offs. Generally, there are two problems: evasion detection and evasion mitigation. Evasion detection methods identify anti-analysis techniques by static analysis [67] or dynamic analysis [7], [66], [76]. Evasion mitigation techniques directly interact with the binary by modification [134], path exploration [92], state modification [66], etc. More detailed models are discussed in [15]. RAVEN can be applied to the evasion detection problem. As an additional analytic layer, it can detect dynamic behaviors related to evasive techniques.

5.6 Limitations and Conclusion

RAVEN still suffers from several issues. First, the inherited design limits its application to the PE or ELF executables. It cannot directly handle other formats such as scripts, Word documents, PDF documents, and web pages. Second, it requires a large amount of labeled data, and its capability is limited to the behavioral indicators identified in the training data. Transfer-learning and domain-adaptation models that can leverage unlabeled data and cross-domain data will be our future directions.

In this paper, we propose to predict a malware's dynamic behavioral indicators using a static machine learning approach. We design a new lightweight neural network architecture that models different information from a binary executable by considering their context. We develop two benchmark datasets and evaluate RAVEN with other binary modeling methods. It shows that RAVEN is accurate and outperforms the state-of-the-art static models on this problem. RAVEN can be used as an additional binary analytic layer to mitigate the issues of polymorphism, metamorphism, and

evasive techniques. It also provides another behavioral abstraction of malware to security analysts.

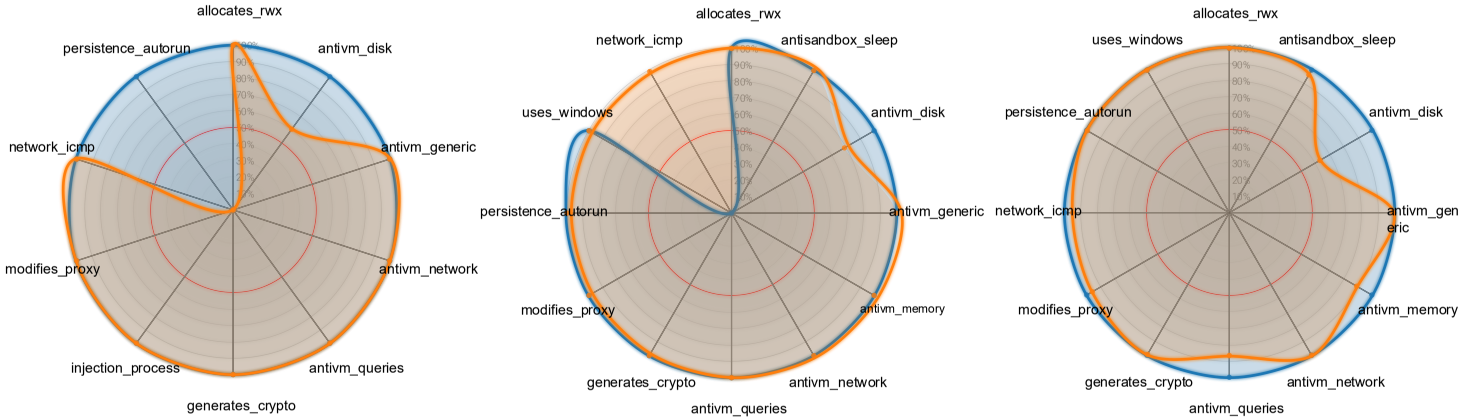

Figure 5.7: A case study using three variants of the *GandCrab* malware. The blue area in the radar charts indicates the true observed behaviors. The orange area indicates the predicted behaviors. Each axis represents a behavioral indicator's ID. The value on each axis represents the confidence value. The red circle in the middle represents the decision boundary of a 0.5 confidence. The description of the behavioral indicators can be found in Appendix D

6

Final Conclusion & Future Work

6.1 Final Conclusion

This thesis presents four original research projects that bridge the area of data mining, machine learning, and reverse engineering. By developing novel data mining and machine learning models and integrating them seamlessly into the reverse engineer's daily routine, these systems can intelligently leverage the knowledge derived from the vast amount of data and use it to reduce the manual analytic effort and cognitive load for reverse engineers. This thesis presents four individual systems: Kam1n0, Sym1n0, Asm2Vec, and RAVEN. Kam1n0, Sym1n0, and Asm2Vec are designed for assembly clone search. They focus on different scenarios. RAVEN focuses on behavioral analysis that provides an additional abstraction layer of malware analysis and can reduce the number of malware to be manually analyzed or simulated. Each system's merits and contributions have been extensively discussed in their corresponding chapters. Next, I briefly summarize each system's use cases, advantages, and limitations. In this chapter, features are denoted by +, and limitations are denoted by -.

Kam1n0 tries to solve the efficient subgraph search problem (i.e., graph isomorphism problem) for assembly functions. It takes only 1.3s on average for query and less than 30ms on average to index time, even when the repository has more than 2.3M functions. Given a target function, it can identify the cloned subgraphs among other functions in the repository.

- + Currently support Meta-PC, ARM, PowerPC, and TMS320c6 (experimental).
- + Support subgraph clone search within a certain assembly code family.
- + Interpretability of the result: explanation by showing subgraph clones.
- + Accurate for searching within the given code family.
- + Good for differing various patches or versions for big binaries.
- - Relatively more sensitive to instruction set changes, optimizations, and obfuscation.
- - Need to pre-define the syntax of the assembly code language.
- - Need to have assembly code of the same chosen family in the repository.

Sym1n0 enables semantic subgraph clone search by differentiated fuzz testing and constraint solving. It is efficient and scalable with a dynamic-static hybrid approach. It takes less than 1s on average for query and less than 100ms on average to index, even when the repository contains more than 1.5M functions. Given a target function, it can identify the cloned subgraphs among other functions of a different processor family in the repository. It also supports syntax visualization and intermediate representation visualization in the system.

- + Clone search by both symbolic execution and concrete execution.
- + Differentiation of functions based on their different I/O behavior.
- + Clone search conducted on the abstract syntax graph constructed from Vex IR (powered by LibVex).
- + Clone search across different assembly code families. For example, indexed x86 binaries but the query is ARM code.
- + Subgraph clone search.
- + Support of a wide range of families through LibVex. x86, AMD64, MIPS32, MIPS64, PowerPC32, PowerPC64, ARM32, and ARM64.
- + Efficient dynamic-static hybrid approach.
- + Analysis of firmware compiled for different processors.
- - Sensitive to heavy graph manipulation (such as a full flattening).
- - Sensitive to large-scale breakdown of basic block integrity.

6.1 Final Conclusion

Asm2Vec leverages representation learning. It understands the lexical semantic relationship of assembly code. For example, *xmm* registers are semantically related to vector operations such as *addps*. *memcpy* is similar to *strcpy*. Asm2Vec also tries to capture interesting patterns that cannot be generalized across all the data samples.

- + Leverage representation learning.
- + Understand the lexical semantic relationship of assembly code.
- + State-of-the-art for clone search against heavy code obfuscation techniques. (>0.8 accuracy for all options applied in O-LLVM, multiple iterations).
- + State-of-the-art for clone search against code optimization. (>0.8 accuracy between O0 and O3, >0.94 accuracy between O2 and O3).
- + Even better result than the most recent dynamic approach.
- + Much more efficient than recent dynamic approaches.
- + No need to define the architecture. It self-learns by reading a large volume of code.
- + Static approach: efficient and scalable.
- - No subgraphs.
- - Assume the assembly code comes from the same processor family.
- - Static approach: cannot recognize jump table, etc.

RAVEN is a neural network that scans a binary executable statically for malicious behavior indicators. It models different aspects of a binary executable using different mechanisms in machine learning. It does not require unpacking. Relying upon pattern recognition, it tries to find any generalizable pattern driven by the data.

- + Statically characterizes malware behaviors, therefore scalable and efficient.
- + Saves computational resources.
- + Reduces manual effort and assists manual analysis.
- - Relies on data to be trained.
- - Limited to the predefined labels presented in the dataset.

- - Black-box approach, unable to explain the decision-making process.

6.2 Future Directions

Data mining and machine learning provide theoretical building blocks to design efficient and effective data-driven solutions addressing the information needs and challenges in cybersecurity. They open the possibilities to those questions that we were not previously able to answer. Taking the presented four systems in this thesis as the starting point, the following topics extend both the depth and width of research in this direction in the future.

Binary Clone Search. The goal is to build a large-scale publicly available binary repository with a clone search engine. It will benefit the security analysts and researchers by reducing the cognitive workload in analyzing any integrated 3rd party libraries. It also benefits the software engineering community by providing the first scalable platform to study the chronological or genetic relationship among a large number of binary applications. My previous works propose different indexes for different use cases. However, this goal calls for a unified self-adaptive index mixture that automatically adjusts the index types and settings according to the characteristics of different indexed subsets. Second, to assist the user's exploration process, it needs to efficiently retrieve and visualize aggregative or clustering clone search results over a selected subset of the binaries, which has not been studied yet.

Malware Analysis. Studying malware behaviors using a data mining and machine learning approach provides an additional binary analytic layer to mitigate the issues of polymorphism, metamorphism, and other evasive techniques. It also provides another behavioral abstraction of malware to security analysts to characterize zero-day malware in real time. The neural network model developed in my previous work follows a black box approach to characterize the dynamic behaviors of malware. However, the prediction result is not interpretable. Given a predicted behavior, it will be more practical to also locate the specific pattern in the malware that correlates to the behavior. In this way the security analysis can develop a deeper understanding of the malware. Moreover, the behavior characterization problem can be formulated as a text generation process. In this way the model can leverage the information in the description and find generalizable patterns across different behavior descriptions. It also enables prediction of behavior description that does

not exist in the training data.

Provenance Analysis. Binary provenance denotes the 'characteristics of a program that derives from its path from source code to executable form'. Binary provenance is important in the domain of binary forensic and performance analysis. It provides an important evidential trial for cybersecurity investigators to track down the hackers behind the security incidences. For example, the *Lazarus* group is linked to the *Wannacry* incidence by code similarity. I mainly focus on two critical aspects: toolchain recovery and authorship analysis. Given a binary file, the task is to verify the compiler family, optimization techniques, obfuscation techniques, and authorship. A binary file contains text and non-text (i.e., binary) information, both carry the writing style of the programmers. I will study both of them.

Vulnerability Analysis. Machine learning-powered vulnerability detection models have been studied for different scenarios. However, the output of the vulnerability detection models is too simple to be useful. Given an identified vulnerability, the security analyst still needs to manually understand the pinpointed assembly functions for verification purposes, since it is a critical step to fix the vulnerability. However, the number of assembly functions to be verified could be large, and the manual verification process is time-consuming. I propose to develop vulnerability detection and classification models that can also assist this manual analytic process by locating the specific pattern of assembly code. In this way, the security analysis can better understand the model's decision-making process, and the vulnerability can be easily understood.

List of Publications and Achievements

At the time of writing, we have finished/published the following publications:

- Assembly Subgraph Clone Search (Chapter 2)

  S. H. H. Ding, B. C. M. Fung, and P. Charland, "Kam1n0: Mapreduce-based assembly clone search for reverse engineering," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, San Francisco, CA: ACM Press, Aug. 2016, 10 pages.

  [**acceptance ratio: 142/784 = 18%**]

- Assembly Clone Search Against Obfuscation and Optimization (Chapter 4)

  S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization," in *Proceedings of the 40th International Symposium on Security and Privacy (S&P)*, San Francisco, CA: IEEE Computer Society, May 2019, pp. 38–55.

  [**new paper acceptance ratio: 46/545 = 8.4%**]

- Cross Architecture Clone Search (Chapter 3, under review)

  S. H. H. Ding, B. C. M. Fung, and P. Charland, "Sym1n0: Large-scale Symbolic Expression Retrieval for Cross-architecture Assembly Clone Search," *IEEE Transactions on Software Engineering*, 2018.

- Neural Malware Behavior Characterization (Chapter 5, under review)

  S. H. H. Ding, B. C. M. Fung, S. McIntosh, S. M, and P. Charland, "Raven: Neural malware behavior characterization without sandbox," in *USENIX Security Symposium (USENIX Security)*, 2019, 18 pages.

In addition to academic publications, the implemented open-source binary analysis platform,

Kam1n0[1], has won the following awards:

- Kam1n0 won 2nd place in the 2015 Hex-Rays International Plug-in Contest[2]. Hex-Rays delivers the most widely used binary analysis software, IDA Pro, for reverse engineering.
- Kam1n0 won the Best Poster Award in the 2016 research showcase of The Smart Cybersecurity Network (SERENE-RISC).
- Kam1n0 has been presented at Google Montreal, EBTIC Research Centre established by British Telecommunications, Above Security Montreal, ESET Montreal, and Sophos Vancouver.
- Kam1n0 is now integrated by Cisco to generate malware signatures[3].
- Kam1n0 has been chosen by Tourism Montreal to be one of the five technological innovations from Montreal[4].

[1] https://github.com/McGill-DMaS/Kam1n0-Community
[2] https://hex-rays.com/contests/2015/
[3] https://www.talosintelligence.com/bass
[4] https://blog.mtl.org/en/5-technological-innovations-you-didnt-know-came-montreal

Bibliography

[1] F. H. Abbasi, A. Salam, and F. Shahzad, *Leveraging behavior-based rules for malware family classification*, US Patent App. 14/967,180, Mar. 2017.

[2] F. Ahmed, H. Hameed, M. Z. Shafiq, *et al.*, "Using spatio-temporal information in api calls with machine learning algorithms for malware detection," in *Proceedings of the 2nd ACM workshop on Security and artificial intelligence*, ACM, 2009, pp. 55–62.

[3] B. Anderson, C. Storlie, and T. Lane, "Improving malware classification: Bridging the static/dynamic gap," in *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, ACM, 2012, pp. 3–14.

[4] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, no. 1, 2008.

[5] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *Proceedings of the 2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017, pp. 177–189.

[6] AVTest. (2018). Avtest malware statistics, [Online]. Available: https://www.av-test.org/en/statistics/malware/ (visited on 08/21/2018).

[7] S. Bahram, X. Jiang, Z. Wang, *et al.*, "Dksm: Subverting virtual machine introspection for fun and profit," in *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, IEEE, 2010, pp. 82–91.

[8] S. Banescu, C. S. Collberg, V. Ganesh, *et al.*, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, 2016, pp. 189–200.

[9] M. Bawa, T. Condie, and P. Ganesan, "LSH forest: Self-tuning indexes for similarity search," in *Proc. of WWW'05*, 2005.

[10] I. D. Baxter, A. Yahin, L. Moura, *et al.*, "Clone detection using abstract syntax trees," in *Proceedings of International Conference on Software Maintenance*, IEEE, 1998.

[11] D. Bilar, "Opcodes as predictor for malware," *International Journal of Electronic Security and Digital Forensics*, vol. 1, no. 2, pp. 156–168, 2007.

[12] M. Bourquin, A. King, and E. Robbins, "Binslayer: Accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, ACM, 2013, p. 4.

[13] J. Bradbury, S. Merity, C. Xiong, *et al.*, "Quasi-Recurrent Neural Networks," in *Proceedings of the International Conference on Learning Representations (ICLR 2017)*, 2017.

[14] S. Brown. (2016). Binary diffing with kam1n0. Retrieved from https://www.whitehatters.academy/diffing-with-kam1n0/ (2017-01-20).

[15] A. Bulazel and B. Yener, "A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web," in *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, ACM, 2017, p. 2.

[16] S. Cesare, Y. Xiang, and W. Zhou, "Malwisean effective and efficient classification system for packed and polymorphic malware," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1193–1206, 2013.

[17] M. Chandramohan, Y. Xue, Z. Xu, *et al.*, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[18] M. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings on 34th Annual ACM STOC'02*, 2002.

[19] P. Charland, B. C. M. Fung, and M. R. Farhadi, "Clone search for malicious code correlation," in *Proc. of the NATO RTO Symposium on Information Assurance and Cyber Defense (IST-111)*, 2012.

[20] S. Das, Y. Liu, W. Zhang, *et al.*, "Semantics-based online malware detection: Towards efficient real-time protection against malware," *IEEE transactions on information forensics and security*, vol. 11, no. 2, pp. 289–302, 2016.

[21] M. Datar, N. Immorlica, P. Indyk, *et al.*, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proc. of ACM SoCG'04*, 2004.

[22] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016.

[23] ——, "Similarity of binaries through re-optimization," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, 2017, pp. 79–94.

[24] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proc. of SIGPLAN'14*, 2014.

[25] X. Deng and J. Mirkovic, "Malware analysis through high-level behavior," in *11th {USENIX} Workshop on Cyber Security Experimentation and Test ({CSET} 18)*, 2018.

[26] D. F. Diehl, D. Alperovitch, I.-A. Ionescu, *et al.*, *Kernel-level security agent*, US Patent 9,904,784, Feb. 2018.

[27] S. H. H. Ding, B. C. M. Fung, S. McIntosh, *et al.*, "Raven: Neural malware behavior characterization without sandbox," in *USENIX Security Symposium (USENIX Security)*, 2019, 18 pages.

[28] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Sym1n0: Large-scale Symbolic Expression Retrieval for Cross-architecture Assembly Clone Search," *IEEE Transactions on Software Engineering*, 2018.

[29] ——, "Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization," in *Proceedings of the 40th International Symposium on Security and Privacy (S&P)*, San Francisco, CA: IEEE Computer Society, May 2019, pp. 38–55.

[30] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Kam1n0: Mapreduce-based assembly clone search for reverse engineering," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, San Francisco, CA: ACM Press, Aug. 2016, 10 pages.

[31] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (english version)," *SSTIC*, vol. 5, no. 1, p. 3, 2005.

[32] S. T. Dumais, "Latent semantic analysis," *Annual review of information science and technology*, vol. 38, no. 1, pp. 188–230, 2004.

[33] M. Egele, M. Woo, P. Chapman, *et al.*, "Blanket execution: Dynamic similarity testing for program binaries and components," in *Proceedings of the 23rd USENIX conference on Security*, 2014.

[34] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "Discovre: Efficient cross-architecture identification of bugs in binary code," in *Proceedings of the 23rd Symposium on Network and Distributed System Security (NDSS)*, 2016.

[35] Y. Fan, S. Hou, Y. Zhang, *et al.*, "Gotcha-sly malware!: Scorpion a metagraph2vec based malware detection system," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ACM, 2018, pp. 253–262.

[36] M. R. Farhadi, B. C. M. Fung, P. Charland, *et al.*, "Binclone: Detecting code clones in malware," in *Proc. of the 8th International Conference on Software Security and Reliability*, 2014.

[37] M. R. Farhadi, B. C. M. Fung, Y. B. Fung, *et al.*, "Scalable code clone search for malware analysis," *Digital Investigation*, 2015.

[38] T. Fawcett, "An introduction to roc analysis," *Pattern recognition letters*, vol. 27, no. 8, 2006.

[39] Q. Feng, R. Zhou, C. Xu, *et al.*, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[40] O. Ferrand, "How to detect the cuckoo sandbox and to strengthen it?" *Journal of Computer Virology and Hacking Techniques*, vol. 11, no. 1, pp. 51–58, 2015.

[41] I. Firdausi, A. Erwin, A. S. Nugroho, *et al.*, "Analysis of machine learning techniques used in behavior-based malware detection," in *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on*, IEEE, 2010, pp. 201–203.

[42] J. Gan, J. Feng, Q. Fang, *et al.*, "Locality-sensitive hashing scheme based on dynamic collision counting," in *Proc. of SIGMOD'12*, 2012.

[43] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Proceedings of the International Conference on Information and Communications Security*, Springer, 2008.

[44] J. Gao, H. V. Jagadish, W. Lu, *et al.*, "Dsh: Data sensitive hashing for high-dimensional k-nnsearch," in *Proc. of SIGMOD'14*, ACM, 2014.

[45] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, *et al.*, Eds., ACM, 2018, p. 497. DOI: 10.1145/3180155.3182551.

[46] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," 1999.

[47] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proc. of the VLDB*, 1999.

[48] M. Graziano, D. Canali, L. Bilge, *et al.*, "Needles in a haystack: Mining information from public dynamic analysis sandboxes for malware intelligence," in *USENIX Security Symposium*, USENIX Association, 2015, pp. 1057–1072.

[49] C. Guarnieri, A. Tanasi, J. Bremer, *et al.*, *The cuckoo sandbox*, 2012.

[50] K. S. Han, J. H. Lim, B. Kang, *et al.*, "Malware analysis using visualized images and entropy graphs," *International Journal of Information Security*, vol. 14, no. 1, pp. 1–14, 2015.

[51] K. Han, B. Kang, and E. G. Im, "Malware analysis using visualized image matrices," *The Scientific World Journal*, vol. 2014, 2014.

[52] K. Han, J. H. Lim, and E. G. Im, "Malware analysis method using visualization of binary files," in *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, ACM, 2013, pp. 317–321.

[53] W. Han, J. Lee, and J. Lee, "Turbo$_{\text{iso}}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proc. of SIGMOD'13*, 2013.

[54] S. Har-Peled, P. Indyk, and R. Motwani, "Approximate nearest neighbor: Towards removing the curse of dimensionality," *Theory of Computing*, vol. 8, no. 1, 2012.

[55] Y. Hu, Y. Zhang, J. Li, *et al.*, "Binary code clone detection across architectures and compiling configurations," in *Proceedings of the 25th International Conference on Program Comprehension*, 2017.

[56] H. Huang, A. Youssef, and M. Debbabi, "Binsequence: Fast, accurate and scalable binary code reuse detection.," in *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, ACM Press, 2017.

[57] V. Inc. (2018). Vmray - the invisible malware sandbox, [Online]. Available: https://www.vmray.com/products/malware-sandbox-products/ (visited on 08/21/2018).

[58] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire os distributions," in *Proceedings of Security and Privacy (SP), 2012 IEEE Symposium on*, IEEE, 2012.

[59] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *Proc. of the 22th USENIX Security Symposium*, 2013.

[60] Y.-C. Jhi, X. Wang, X. Jia, *et al.*, "Value-based program characterization and its application to software plagiarism detection," in *Proceedings of Software Engineering (ICSE), 2011 33rd International Conference on*, IEEE, 2011.

[61] L. Jiang, G. Misherghi, Z. Su, *et al.*, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proc. of the 29th international conference on Software Engineering*, IEEE Computer Society, 2007.

[62] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the 18th International Symposium on Software Testing and Analysis*, ACM, 2009.

[63] E. Juergens *et al.*, "Why and how to control cloning in software artifacts," *Technische Universität München*, 2011.

[64] P. Junod, J. Rinaldini, J. Wehrli, *et al.*, "Obfuscator-llvm–software protection for the masses," in *Proceedings of 2015 IEEE/ACM 1st International Workshop on Software Protection (SPRO)*, IEEE, 2015.

[65] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE TSE*, vol. 28, no. 7, 2002.

[66] M. G. Kang, H. Yin, S. Hanna, *et al.*, "Emulating emulation-resistant malware," in *Proceedings of the 1st ACM workshop on Virtual machine security*, ACM, 2009, pp. 11–22.

[67] A. Kapravelos, Y. Shoshitaishvili, M. Cova, *et al.*, "Revolver: An automated approach to the detection of evasive web-based malware.," in *USENIX Security Symposium*, 2013, pp. 637–652.

[68] W. M. Khoo, "Decompilation as search," *University of Cambridge, Computer Laboratory, Technical Report*, 2013.

[69] W. M. Khoo, A. Mycroft, and R. J. Anderson, "Rendezvous: A search engine for binary code," in *Proc. of MSR'13*, 2013.

[70] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[71] D. Kirat, L. Nataraj, G. Vigna, *et al.*, "Sigmal: A static signal processing based malware triage," in *Proceedings of the 29th Annual Computer Security Applications Conference*, ACM, 2013, pp. 89–98.

[72] C. Kolbitsch, E. Kirda, and C. Kruegel, "The power of procrastination: Detection and mitigation of execution-stalling malicious code," in *Proceedings of the 18th ACM conference on Computer and communications security*, ACM, 2011, pp. 285–296.

[73] V. Komsiyski, "Binary differencing for media files," 2013.

[74] C. Kruegel, "Full system emulation: Achieving successful automated dynamic analysis of evasive malware," in *Proceedings of the 2014 BlackHat USA Security Conference*, 2014, pp. 1–7.

[75] C. Kruegel, E. Kirda, D. Mutz, *et al.*, "Polymorphic worm detection using structural information of executables," in *Proc. of RAID'06*, Springer, 2006.

[76] B. Lau and V. Svajcer, "Measuring virtual machine detection in malware using dsd tracer," *Journal in Computer Virology*, vol. 6, no. 3, pp. 181–195, 2010.

[77] Q. Le, O. Boydell, B. Mac Namee, *et al.*, "Deep learning at the shallow end: Malware classification for non-domain experts," *Digital Investigation*, vol. 26, S118–S126, 2018.

[78] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the International Conference on Machine Learning*, 2014, pp. 1188–1196.

[79] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[80] J. Lee, W. Han, R. Kasperovics, *et al.*, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," *PVLDB*, vol. 6, no. 2, 2012.

[81] X. Li, P. K. Loh, and F. Tan, "Mechanisms of polymorphic and metamorphic viruses," in *Proceedings of the 2011 European Intelligence and Security Informatics Conference (EISIC)*, IEEE, 2011.

[82] Y. Li, D. Tarlow, M. Brockschmidt, *et al.*, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[83] Z. Li, S. Lu, S. Myagmar, *et al.*, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, 2006.

[84] Y. Liu, J. Cui, Z. Huang, *et al.*, "SK-LSH: an efficient index structure for approximate nearest neighbor search," *PVLDB*, vol. 7, no. 9, 2014.

[85] L. Luo, J. Ming, D. Wu, *et al.*, "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*, 2014.

[86] C. D. Manning, P. Raghavan, H. Schütze, *et al.*, *Introduction to information retrieval*. Cambridge University Press, 2008, vol. 1.

[87] C. D. Manning and H. Schütze, *Foundations of statistical natural language processing*. MIT press, 1999.

[88] T. Mikolov, K. Chen, G. Corrado, *et al.*, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013.

[89] T. Mikolov, I. Sutskever, K. Chen, *et al.*, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the Advances in Neural Information Processing Systems*, 2013.

[90] J. Ming, M. Pan, and D. Gao, "Ibinhunt: Binary hunting with inter-procedural control flow," in *Proceedings of the International Conference on Information Security and Cryptology*, Springer, 2012.

[91] A. Mockus, "Large-scale code reuse in open source software," in *Proc. of FLOSS'07*, IEEE, 2007.

[92] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Security and Privacy, 2007. SP'07. IEEE Symposium on*, IEEE, 2007, pp. 231–245.

[93] S. Mutti, Y. Fratantonio, A. Bianchi, *et al.*, "Baredroid: Large-scale analysis of android apps on real devices," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ACM, 2015, pp. 71–80.

[94] G. Myles and C. Collberg, "K-gram based software birthmarks," in *Proceedings of the 2005 ACM Symposium on Applied Computing*, ACM, 2005, pp. 314–318.

[95] L. Nataraj, S. Karthikeyan, G. Jacob, *et al.*, "Malware images: Visualization and automatic classification," in *2011 International Symposium on Visualization for Cyber Security, VizSec '11, Pittsburgh, PA, USA, July 20, 2011*, ACM, 2011, p. 4.

[96] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, 2007.

[97] L. Nouh, A. Rahimian, D. Mouheb, *et al.*, "Binsign: Fingerprinting binary functions to support automated analysis of code executables," in *Proceedings of the IFIP International Conference on ICT Systems Security and Privacy Protection*, Springer, 2017.

[98] Y. Oyama, "Investigation of the diverse sleep behavior of malware," *JIP*, vol. 26, pp. 461–476, 2018. DOI: 10.2197/ipsjjip.26.461.

[99] PandaLabs, "Annual report pandalabs 2013 summary," 2013.

[100] ——, "Annual report pandalabs 2014 summary," 2014.

[101] ——, "Annual report pandalabs 2017 summary," 2017.

[102] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation.," in *EMNLP*, vol. 14, 2014, pp. 1532–43.

[103] J. Pewny, B. Garmany, R. Gawlik, *et al.*, "Cross-architecture bug search in binary executables," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, IEEE, 2015.

[104] J. Pewny, F. Schuster, L. Bernhard, *et al.*, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ACM, 2014.

[105] J. Qiu, X. Su, and P. Ma, "Library functions identification in binary code by using graph isomorphism testings," in *Proc. of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2015.

[106] B. B. Rad, M. Masrom, and S. Ibrahim, "Camouflage in malware: From encryption to metamorphism," *International Journal of Computer Science and Network Security*, vol. 12, no. 8, 2012.

[107] E. Raff, J. Barker, J. Sylvester, *et al.*, "Malware detection by eating a whole exe," *arXiv preprint arXiv:1710.09435*, 2017.

[108] E. Raff, J. Sylvester, and C. Nicholas, "Learning the pe header, malware detection with minimal domain knowledge," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, ACM, 2017, pp. 121–132.

[109] E. Raff, R. Zak, R. Cox, *et al.*, "An investigation of byte n-gram features for malware classification," *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 1, pp. 1–20, 2018.

[110] C. Ravi and R. Manoharan, "Malware detection using windows api sequence and machine learning," *International Journal of Computer Applications*, vol. 43, no. 17, pp. 12–16, 2012.

[111] K. Rieck, T. Holz, C. Willems, *et al.*, "Learning and classification of malware behavior," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2008, pp. 108–125.

[112] K. Rieck, P. Trinius, C. Willems, *et al.*, "Automatic analysis of malware behavior using machine learning," *Journal of Computer Security*, vol. 19, no. 4, pp. 639–668, 2011.

[113] R. Ronen, M. Radu, C. Feuerstein, *et al.*, "Microsoft malware classification challenge," *arXiv preprint arXiv:1802.10135*, 2018.

[114] A. Saebjornsen, "Detecting fine-grained similarity in binaries," PhD thesis, UC Davis, 2014.

[115] A. Sæbjørnsen, J. Willcock, T. Panas, *et al.*, "Detecting code clones in binary executables," in *Proc. of the 18th International Symposium on Software Testing and Analysis*, 2009.

[116] I. Santos, F. Brezo, J. Nieves, *et al.*, "Idea: Opcode-sequence-based malware detection," in *International Symposium on Engineering Secure Software and Systems*, Springer, 2010, pp. 35–43.

[117] I. Santos, J. Nieves, and P. G. Bringas, "Semi-supervised learning for unknown malware detection," in *International Symposium on Distributed Computing and Artificial Intelligence*, Springer, 2011, pp. 415–422.

[118] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on*, IEEE, 2015, pp. 11–20.

[119] M. Schubotz, A. Grigorev, M. Leich, *et al.*, "Semantification of identifiers in mathematics for better math information retrieval," in *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, ACM, 2016.

[120] J. Security. (2018). Joe sandbox cloud basic, [Online]. Available: https://www.joesandbox.com/ (visited on 08/21/2018).

[121] A. Shabtai, R. Moskovitch, C. Feher, *et al.*, "Detecting unknown malicious code by applying classification techniques on opcode patterns," *Security Informatics*, vol. 1, no. 1, p. 1, 2012.

[122] R. Shamir and D. Tsur, "Faster subtree isomorphism," in *Proc. of IEEE ISTCS'97*, 1997.

[123] P. Shirani, L. Wang, and M. Debbabi, "Binshape: Scalable and robust binary library function identification using function shape," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2017.

[124] M. Sojer and J. Henkel, "Code reuse in open source software development: Quantitative evidence, drivers, and impediments," *JAIS*, vol. 11, no. 12, 2010.

[125] S. Sonnenburg, G. Rätsch, C. Schäfer, *et al.*, "Large scale multiple kernel learning," *JMLR*, vol. 7, 2006.

[126] J. tastná and M. Tomáek, "Characterising malicious software with high-level behavioural patterns," in *International Conference on Current Trends in Theory and Practice of Informatics*, Springer, 2017, pp. 473–484.

[127] J. t'astná and M. Tomáek, "High-level malware behavioural patterns: Extractability evaluation," in *Computer Science and Information Systems (FedCSIS), 2017 Federated Conference on*, IEEE, 2017, pp. 569–572.

[128] Z. Sun, H. Wang, H. Wang, *et al.*, "Efficient subgraph matching on billion node graphs," *The VLDB Endowment*, vol. 5, no. 9, 2012.

[129] Y. Tao, K. Yi, C. Sheng, *et al.*, "Quality and efficiency in high dimensional nearest neighbor search," in *Proc. of SIGMOD'09*, 2009.

[130] ——, "Efficient and accurate nearest neighbor and closest pair search in high-dimensional space," *ACM TODS*, vol. 35, no. 3, 2010.

[131] E. Thioux, M. Amin, D. Kindlund, *et al.*, *Malicious content analysis using simulated user interaction without user involvement*, US Patent 9,104,867, Aug. 2015.

[132] R. Tian, R. Islam, L. Batten, *et al.*, "Differentiating malware from cleanware using behavioural analysis," in *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, IEEE, 2010, pp. 23–30.

[133] J. R. Ullmann, "An algorithm for subgraph isomorphism," *ACM JACM*, vol. 23, no. 1, 1976.

[134] A. Vasudevan and R. Yerraballi, "Cobra: Fine-grained malware analysis using stealth localized-executions," in *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE, 2006, 15–pp.

[135] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[136] J. Wang, H. T. Shen, J. Song, *et al.*, "Hashing for similarity search: A survey," *arXiv:1408.2927*, 2014.

[137] H. Welte, *Current developments in GPL compliance*, 2012.

[138] M. White, M. Tufano, C. Vendome, *et al.*, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ACM, 2016.

[139] D. Xu, J. Ming, and D. Wu, "Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping," in *Security and Privacy (SP), 2017 IEEE Symposium on*, IEEE, 2017, pp. 921–937.

[140] X. Xu, C. Liu, Q. Feng, *et al.*, "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*, ACM, 2017, pp. 363–376.

[141] Y. Ye, L. Chen, S. Hou, *et al.*, "Deepam: A heterogeneous deep learning framework for intelligent malware detection," *Knowl. Inf. Syst.*, vol. 54, no. 2, pp. 265–285, 2018. DOI: 10.1007/s10115-017-1058-9.

[142] Y. Ye, T. Li, Y. Chen, *et al.*, "Automatic malware categorization using cluster ensemble," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2010, pp. 95–104.

[143] H. Yin and D. Song, "Temu: Binary code analysis via whole-system layered annotative execution," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3, Jan. 2010.

[144] Z. Yuan, Y. Lu, Z. Wang, *et al.*, "Droid-sec: Deep learning in android malware detection," in *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, F. E. Bustamante, Y. C. Hu, A. Krishnamurthy, *et al.*, Eds., ACM, 2014, pp. 371–372. DOI: 10.1145/2619239.2631434.

[145] R. Zak, E. Raff, and C. Nicholas, "What can n-grams learn for malware detection?" In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, IEEE, 2017, pp. 109–118.

[146] R. Zanibbi and D. Blostein, "Recognition and retrieval of mathematical expressions," *International Journal on Document Analysis and Recognition (IJDAR)*, vol. 15, no. 4, 2012.

[147] F. Zhang, Y. Jhi, D. Wu, *et al.*, "A first step towards algorithm plagiarism detection," in *Proc. of ISSTA'12*, 2012.

[148] H. Zhang, X. Xiao, F. Mercaldo, *et al.*, "Classification of ransomware families with machine learning based on n-gram of opcodes," *Future Generation Computer Systems*, vol. 90, pp. 211–221, 2019.

[149] X. Zhang and R. Gupta, "Matching execution histories of program versions," in *ACM SIGSOFT Software Engineering Notes*, ACM, vol. 30, 2005.

Appendices

A

Kam1n0 Implementation Details

This appendix provides more technical details on the assembly clone search engine *Kam1n0*. Section A.1 elaborates details on the assembly code normalization scheme. Section A.2.1 provides theoretical analysis of the Adaptive Locality Sensitive Hashing (ALSH) index scheme. Section A.2.2 introduces how we implement the ALSH on top of the key-value based column store, which by default does not support a prefix tree.

A.1 Assembly code normalization

The equivalent assembly code fragments can be represented in different forms. In order to mitigate this issue we normalize the operands in assembly code during the preprocessing. We extend the normalization tree used in BinClone [36]. Each level of the tree represents a normalization level and there are three of them: *root*, *type*, and *specific*. Moreover, we determine how many bits are actually used given its modifier keyword such as `short`, `dword`, `qword`, etc. For example, given the assembly line `movq xmm0, qword ptr [eax+30h]`, operand `xmm0` is a 128-bit register. However, due to the `qword` modifier, only the lower 64 bits are used.

A.2 Adaptive Locality Sensitive Hashing

This section provides theoretical analysis of the ALSH scheme, followed by details of its implementation on Cassandra-like key-value column store.

A.2.1 Theoretical Analysis

In this section, we present our theoretical analysis on the ALSH index scheme. Following the Theorem 1, we have the following observation:

Observation 1 *g_t is a $(r_t, r_{t+1}, p_m, p_m^c)$-sensitive hash function.* □

Proof 2 *At position t, the r value is r_t, and the approximated r value is r_{t+1}. k_t hash functions are concatenated to form g_i. According to Equation 2.2 and Definition 4, we have:*

$$p_1 = (1 - \frac{r_t}{\pi})^{k_t} \tag{A.1}$$

$$= (1 - \frac{\pi \times (1 - p_m^{(1/k_t)})}{\pi})^{k_t} = p_m \tag{A.2}$$

$$p_2 = (1 - \frac{r_{t+1}}{\pi})^{k_t} \tag{A.3}$$

$$= (1 - \frac{\pi \times (1 - p_m^{(1/k_{t+1})})}{\pi})^{k_t} = p_m^{k_t/k_{t+1}} = p_m^c \tag{A.4}$$

Thus, g_t is a $(r_t, r_{t+1}, p_m, p_m^c)$-sensitive hash function.

Following Observation 1, we have two different hash functions, g_t and g_{t+1}, by moving from position t to $t+1$. g_t is $(r_t, r_{t+1}, p_m, p_m^c)$-sensitive and g_{t+1} is $(r_{t+1}, r_{t+2}, p_m, p_m^c)$-sensitive. In other words, by increasing the distance r, we still achieve the same p_1 and p_2 for the locality sensitive hashing function. It is the same rationale as described in LSB-Tree [130] and C2LSH [42]. By moving up the ALSH prefix tree we have the same effect of using different g_t with decreasing k_t and increasing r, and vice versa. At the same time, we guarantee the same p_1 and p_2.

In order to have g_t correctly working for the $(r_{t+1}/r_t, r_t)$-approximated ball cover problem, g_t also needs to satisfy both Property 1 and Property 2 with constant probability. Following [54], [129], we adopt l ASLH prefix trees to ensure the quality. The proof follows [54].

Proposition 1 *Given the $(r_t, r_{t+1}, p_m, p_m^c)$-sensitive hash function family $\mathcal{G}$, there exists an algorithm for $(r_{t+1}/r_t, r_t)$-NN ball cover problem under cosine similarity measure.* □

Proof 3 *Let P_1 be the probability that Property 1 holds, P_2 be the probability that Property 2*

holds, and q be the query point. It suffices to ensure that both P_1 and P_2 are strictly greater than half [54].

By setting $p_m^c = 1/n$, the probability that $P[g_t(p) = g_t(q)]$ for $p \notin$ ball $B(q, r_{t+1})$ is $1/n$. Thus, the expected number of collided points that $\notin B(q, r_{t+1})$ is at most $n \times 1/n = 1$ for g_t for a single ALSH tree. By getting points from l such trees, the expected number of such collisions is at most l (1 for each tree), so the probability that this number exceeds $2l$ is less than $1/2$ by Markov inequality. In other words, Property 2 holds with $P_2 > 1/2$.

The following proof shows that $P_1 > 1/2$. $P[g_t(p) = g_t(q)]$ for $p \in$ ball $B(q, r_{t+1})$ is bounded from below by $p_m = (1/n)^{1/c} = n^{-1/c}$. The probability that all the l indexes miss the point p is $(1 - n^{-1/c})^l$. Thus, $P_1 = 1 - (1 - n^{-1/c})^l$. By setting $l = n^{1/c}$, we have $P_1 > 1 - 1/e > 1/2$. Thus, the proposition holds.

Another parameter for the ALSH index is r_m, which controls the starting k_m value at the root value. It indicates the minimum distance that two points can be considered as valid neighbors. Really sparse points far away from each other are not considered as neighbors unless their distance is within r_m.

$$\begin{aligned} r_m &= (1 - p_m^{k_m}) \\ k_m &= \frac{ln(1 - r_m)}{ln(p_m)} = \frac{ln(1 - r_m)}{ln(n^{-1/c})} = -c\frac{ln(1 - r_m)}{ln(n)} \end{aligned} \tag{A.5}$$

We also set a maximum depth for the tree, i.e., the maximum k value k_0 at the level 0.

A.2.2 Implementation on the Key-value Database

Key-value databases, such as Apache Cassandra, are intrinsically difficult to accommodate prefix tree structures. The randomized data partitioner plays a critical role on load balancing across nodes, and it provides better read-write performance than a sorted-key based partitioner. Nonetheless, partitioning the data by randomly projecting the original key space to another one disables efficient prefix search. In this section, we present our design of a data module that fits the ALSH index to the Cassandra key-value storage.

A.2 Adaptive Locality Sensitive Hashing

Refer to Figure A.2. All the trees of an ALSH index are stored in a single column family. Basically, a column family consists of rows of records. Each row has a partition key and several clustering keys. A partition key is used to partition the rows into different cluster nodes in the database, and the clustering key can be sorted according to the key value. We maintain a row for each node in an ALSH tree. Suppose a node is at level t_{t+1}, and it has children at level t. Its partition key is a combination of the tree ID and the signature generated by g_{t+1}. A row has two fields: *Hids* and *Cids*. *Hids* is a set of unique ID that links to the data points of this node. If *Hids* is empty, it means that this node has more than $2l$ points and has been split. Another field, *Cids*, is a clustering key that contains the children of this node. The *Cid* for each child is the signature generated by the function g_t. Since they share the same prefix generated by g_{t+1}, we only store the distinct part. Correspondingly, for each child there is a *Hids* field link to data points. If we need to split a child we empty its *Hids* and create a new row accordingly.

Algorithm 8 Basic Block Semantic Search (BBSS)

Input the basic blocks B_t of the target function f_t
Output basic block clone pairs $\{\langle b_t, b_s \rangle, \dots\}$

```
1: result ← {}
2: for each b_t in B_t do
3:     b_t ← preprocess(b_t)
4:     q⃗ ← contructVector(b_t)
5:     points ← ALSH_Query(q⃗)
6:     for each p⃗ in points do
7:         b_s ← getSourceBlock(p⃗)
8:         result = result ⋃ ⟨b_t, b_s⟩
9:     end for
10: end for
11: return result
```

For example, in Figure A.2, the first row represents a node from tree of ID "0001". Its signature generated by g_t is "58". Its empty *Hids* implies that it has been split. It has a list of children and each of them is distinguished by *Cid*. Suppose we have to split its child "A0" at level t to level $t-1$. We first create a new row using its tree ID and signature "58A0" that is generated by g_t. Then we generate signatures for data points in cell "0001-58:A0" using g_{t-1}, and put them in the newly created row. To fulfill the query, we first locate the leaf cell as described in Section 2.5. In

ALSH we only index the unique data points. We also maintain another column family where data points are indexed by a strong hash signature on the feature vector. To index a new data point, we check whether it exists in the database by hashing.

By adopting the ALSH scheme, we can efficiently retrieve all the basic block clones by considering their semantic similarity revealed from their assembly code instructions. Since we use the cosine similarity, a clone between a short basic block and a long basic block can be detected. Recall that our input to the clone search engine is a target function f_t with its target blocks B_t and edges of the control flow graph E_t. Algorithm 8 provides the interface for the graph search algorithm in Section 2.6.

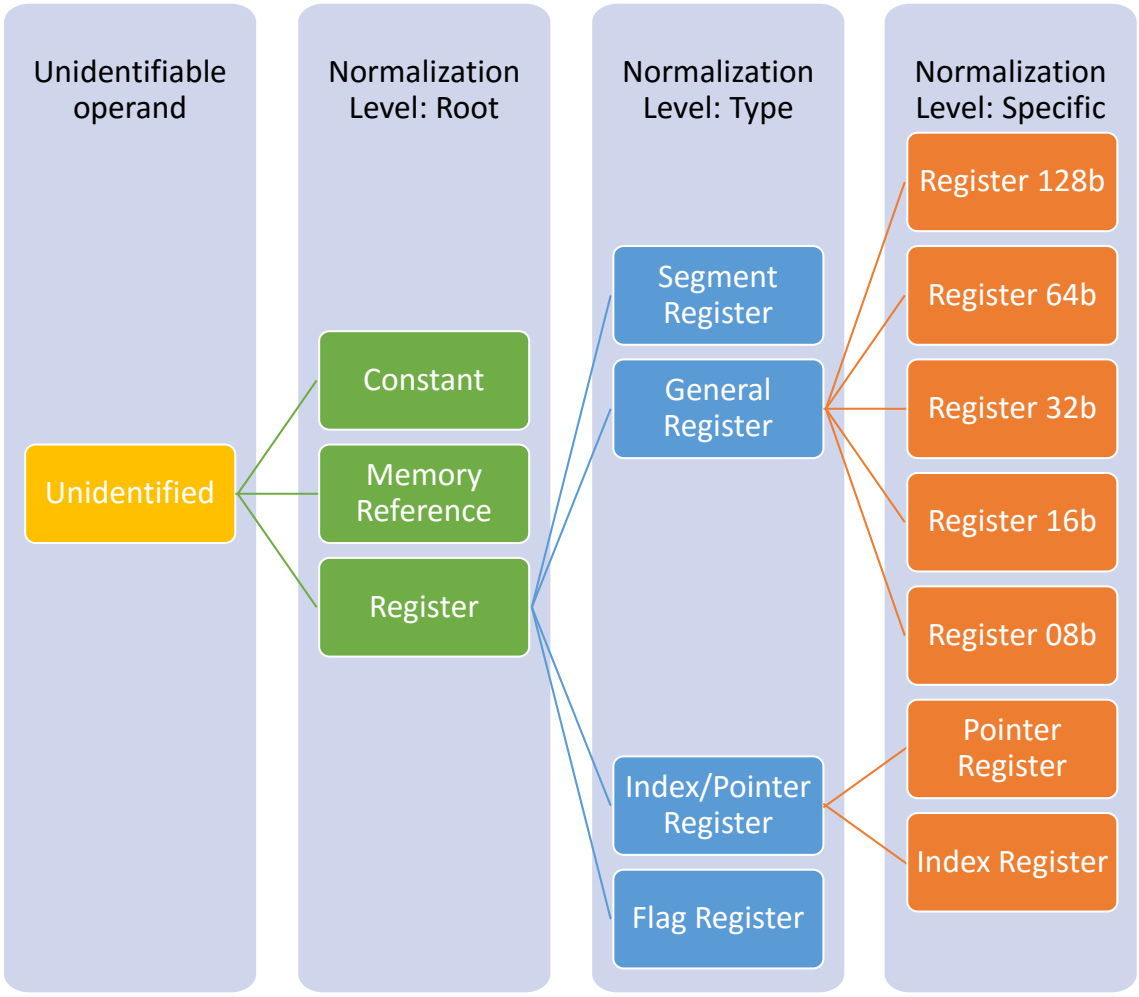

Figure A.1: The hierarchy used to normalize the operands. Each level of the tree represents a normalization level.

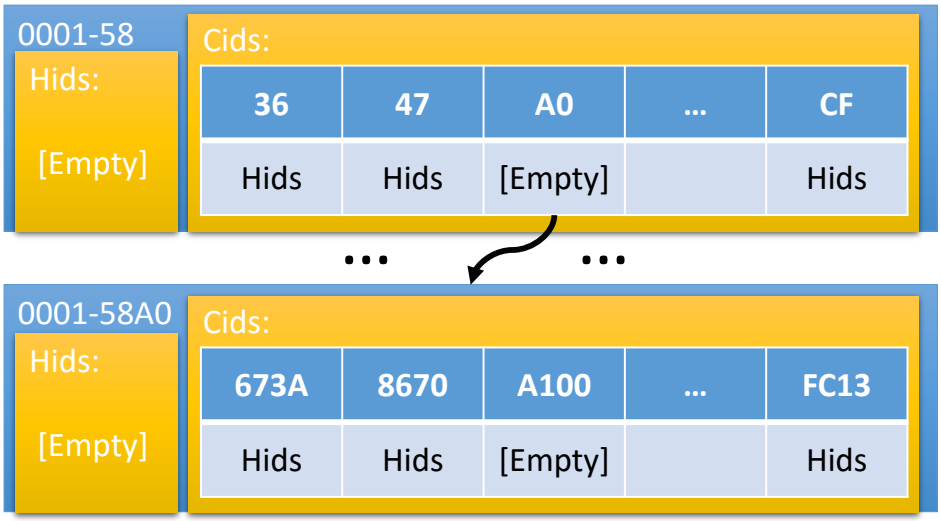

Figure A.2: The data module for ALSH.

B

Extended Formulation of Asm2Vec

This appendix extends the original description and the formulation of the *Asm2Vec* model training. Recall that at the beginning of Section 4.3 we define f_s as an assembly function in the repository. The *Asm2Vec* model tries to learn the following parameters:

$\vec{\theta}_{f_s} \in \mathbb{R}^{2\times d}$	The vector representation of the function f_s.
$\vec{v}_t \in \mathbb{R}^d$	The vector representation of a token t.
$\vec{v'}_t \in \mathbb{R}^d$	Another vector of token t, used for prediction.

Table B.1: Parameters to be estimated in training.

All $\vec{\theta}_{f_s}$ and $\vec{v}_t$ are initialized to small random values around zero. All $\vec{v'}_t$ are initialized to zeros. We use $2 \times d$ for f_s because we concatenate the vector for operation and operands to represent an instruction. We also define the following symbols according to the syntax of assembly language:

$\mathcal{S}(f_s) = seq[1:i]$	Multiple sequences generated from f_s.
$\mathcal{I}(seq_i) = in[1:j]$	Instructions of a sequence seq_i.
in_j	The j^{th} instruction in a sequence.
$\mathcal{A}(in_j)$	Operands of instruction in_j.
$\mathcal{P}(in_j)$	The operation of instruction in_j.
$\mathcal{T}(in_j)$	Represent the tokens of in_j.
$\mathcal{CT}(in_j) \in \mathbb{R}^{2\times d}$	Vector representation of an instruction in_j.
$\mathcal{CT}(in_{j-1}) \in \mathbb{R}^{2\times d}$	Vector representation of in_j's previous instruction.
$\mathcal{CT}(in_{j+1}) \in \mathbb{R}^{2\times d}$	Vector representation of an instruction in_j's next instruction.
$\delta(in_j, f_s)$	Vector representation of the joint memory of function f_s and in_j's neighbor instructions.

Table B.2: Intermediate symbols used in training.

For an instruction in_j, we treat the concatenation of its operation and operands as its tokens $\mathcal{T}(in_j)$: $\mathcal{T}(in_j) = \mathcal{P}(in_j) \mid\mid \mathcal{A}(in_j)$, where $\mid\mid$ denotes concatenation. $\mathcal{CT}(in)$ denotes the vector representation of an instruction in.

$$\mathcal{CT}(in) = \vec{v}_{\mathcal{P}(in)} || \frac{1}{|\mathcal{A}(in)|} \sum_{t}^{\mathcal{A}(in)} \vec{v}_{t_b}$$

The representation is calculated by averaging the vector representations of its operands $\mathcal{A}(in)$. The averaged vector is then concatenated to the vector representation $\vec{v}_{\mathcal{P}(in)}$ of its operation $\mathcal{P}(in)$.

As presented in Algorithm 6, the training procedure goes through each assembly function f_s in the repository and generates multiple sequences by calling $\mathcal{S}(f_s)$. For each sequence seq_i of function f_s, the neural network walks through the instructions from its beginning. We collect the current instruction in_j, its previous instruction in_{j-1}, and its next instruction in_{j+1}. We ignore the instructions that are out-of-boundary. We calculate $\mathcal{T}(in_{j-1})$ and $\mathcal{T}(in_{j+11})$ using the previous equation. By averaging f_s's vector representation $\vec{\theta}_{f_s}$ with $\mathcal{CT}(in_j - 1)$ and $\mathcal{CT}(in_j + 1)$, $\delta(in, f_s)$

models the joint memory of neighbor instructions:

$$\delta(in_j, f_s) = \frac{1}{3}(\vec{\theta}_{f_s} + \mathcal{CT}(in_{j-1}) + \mathcal{CT}(in_{j+1}))$$

For the current instruction in_j, the proposed model maximizes the following log probability:

$$\arg\max \sum_{t_c}^{\mathcal{T}(in_j)} log\ \mathbf{P}(t_c | f_s, in_{j-1}, in_{j+1})$$

It predicts each token in the current instruction in_j based on the joint memory of its corresponding function vector and its neighbor instruction vectors, as illustrated in Figure 4.5.

To model the above prediction one can use a typical softmax multi-class classification layer and maximize the following log probability:

$$\begin{aligned}
\mathbf{P}(t_c | \delta(in_j, f_s)) &= \mathbf{P}(\vec{v'}_{t_c} | \delta(in_j, f_s)) \\
&= \frac{f(\vec{v'}_{t_c}, \delta(in_j, f_s))}{\sum_d^D f(\vec{v'}_{t_d}, \delta(in_j, f_s))} \\
f(\vec{v'}_{t_c}, \delta(in_j, f_s)) &= Uh((\vec{v'}_{t_c})^T \times \delta(in_j, f_s))
\end{aligned}$$

D denotes the whole vocabulary constructed upon the repository RP. $Uh(\cdot)$ denotes a sigmoid function applied to each value of a vector. The total number of parameters to be estimated is $(|D| + 1) \times 2 \times d$ for each pass of the softmax layout. The term $|D|$ is too large to be efficient for the softmax classification.

Therefore we use the k negative sampling approach [78], [89] to approximate the log probability:

$$\begin{aligned}
&log\ \mathbf{P}(t_c | \delta(in_j, f_s)) \approx log\ f(\vec{v'}_{t_c} | \delta(in_j, f_s)) \\
&+ \sum_{i=1}^{k} \mathbb{E}_{t_d \backsim P_n(t_c)} \big([\![t_d \neq t_c]\!] log\ f(-1 \times \vec{v'}_{t_d},\ \delta(in_j, f_s)) \big)
\end{aligned}$$

By manipulating the value of the parameters listed in Table B.1 we can maximize the sum of the

above log-probability for all the instruction in_j.

We follow the parallel stochastic gradient decent algorithm. In a single training step we only consider a single token t_c of the current instruction in_j. We calculate the above log probability and its gradients with respect to the parameters that we are trying to manipulate. The gradients define the direction in which we should manipulate the parameters in order to maximize the log probability. The gradients are calculated by taking the derivatives with respect to each parameter defined in Table B.1. The table below defines the symbol of the gradients:

$\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta)$	The gradient for current function f_s's $\vec{\theta}_{f_s}$.
$\frac{\partial}{\partial \vec{v'}_t} J(\theta)$	The gradient for the token t_c of current instruction in_j.
$\frac{\partial}{\partial \vec{v}_{\mathcal{P}(in_{j+1})}}$	The gradient for the operation of instruction in_{j+1}.
$\frac{\partial}{\partial \vec{v}_{\mathcal{P}(in_{j-1})}}$	The gradient for the operation of instruction in_{j-1}.
$\frac{\partial}{\partial \vec{v}_{t_b}} J(\theta)$	The gradient for each operation of instruction in_{j+1} and in_{j-1}.

Table B.3: Gradients to be calculated in a training step.

The equations below calculate the gradients defined above.

$$\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta) = \frac{1}{3} \sum_{i}^{k} \mathbb{E}_{t_b \backsim P_n(t_c)} \big([\![t_b = t_c]\!] - f(\vec{v'}_t, \delta(in_j, f_s)) \big) \times \vec{v'}_t$$

$$\frac{\partial}{\partial \vec{v'}_t} J(\theta) = [\![t = t_c]\!] - f(\vec{v'}_t, \delta(in_j, f_s)) \times \delta(in_j, f_s)$$

It will be the same equation for the previous instruction in_{j-1}, by replacing in_{j+1} with in_{j-1}.

$$\frac{\partial}{\partial \vec{v}_{\mathcal{P}(in_{j+1})}} J(\theta) = \big(\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta)\big)[0 : d-1]$$
$$\frac{\partial}{\partial \vec{v}_{t_b}} J(\theta) = \frac{1}{|\mathcal{A}(in_{j+1})|} \times \big(\frac{\partial}{\partial \vec{\theta}_{f_s}} J(\theta)\big)[d : 2d-1]$$
$$t_b \in \mathcal{A}(in_{j+1})$$

After, we use back propagation to update the values of all the involved parameters according to their gradients in Table B.3, with a learning rate.

C

Extended Descriptive Statistics of the Dataset

This appendix provides additional descriptive statistics on the experimental dataset used in Section 4.4.1, Section 4.4.2, and Section 4.4.3.

In the compiler optimization experiment (Section 4.4.1, *ImageMagick*) generally has the largest number of assembly basic blocks, while *zlib* has the least. By adopting different compiler optimization options, the generated number of basic blocks greatly varies. Specifically, O0 is very different from the other optimization levels. O1 and O2 appear to share a similar number. O3 has the largest number of basic blocks, which is generated by intensive inlining. Figure C.2 shows the empirical distribution of the assembly functions length under different optimization levels. O3 tends to produce assembly functions that are much longer than O0, O1, and O2. O1 and O2 share similar distributions on function length.

In the *O-LLVM* obfuscation experiment (Section 4.4.2), we evaluate the clone search methods before and after obfuscation. *O-LLVM* significantly increases the complexity of the binary code. Table C.2 shows how the number of basic blocks have been changed across different obfuscation levels. Figure C.3 shows the empirical distribution of the assembly functions length under different obfuscation options. There are three different techniques and their combination:

- *BCF* modifies the control flow graph by adding a large number of irrelevant random basic blocks and branches. It will also split, merge, and reorder the original basic blocks. It almost doubles the number of basic blocks after obfuscation (see Table C.2).

	GCC O0	GCC O1	GCC O2	GCC O3
BusyBox	52,118	46,519	47,272	62,069
CoreUtils	38,176	36,168	35,117	41,421
Libgmp	12,919	15,534	14,602	16,234
ImageMagick	85,191	88,342	84,395	93,421
Libcurl	17,969	14,097	13,483	15,371
LibTomCrypt	12,021	10,135	10,258	13,451
OpenSSL	52,063	44,527	44,642	50,043
SQLite	27,621	24,978	29,332	38,699
zlib	2,898	2,747	2,668	3,706
PuTTYgen	5,495	4,957	5,065	7, 231
Total	306,471	288,004	286,834	341,646

Table C.1: Number of basic blocks for each selected library compiled using different optimization options.

	Original	BCF	FLA	SUB	All
Libgmp	20,168	54,738	103,258	20,168	55,007
ImageMagick	83,704	218,315	434,599	83,702	216,904
LibTomCrypt	10,044	19,534	35,608	10,115	62,895
OpenSSL	46,298	100,315	160,265	46,278	289,657
Total	160,214	392,902	733,730	160,263	624,463

Table C.2: Number of basic blocks for each selected library under different code obfuscation options.

- *FLA* reorganizes the original CFG using a complex hierarchy of new conditions as switches (see an example in Figure 4.1). Only the penultimate level of the CFG contains the modified original logics. It completely destroys the original CFG graph. The obfuscated binary on average contains 4 times more basic blocks than the original.

```
movzx   ecx, byte ptr [rbp+1]
shl     ecx, 8
or      ecx, eax
movzx   eax, byte ptr [rbp+2]
```

```
movzx   ecx, byte ptr [r13+1]
shl     ecx, 8
mov     edx, ecx
not     edx
mov     esi, eax
not     esi
and     edx, 8D8113F6h
and     ecx, 0EC00h
and     esi, 8D8113F6h
and     eax, 9
or      ecx, edx
or      eax, esi
xor     eax, ecx
movzx   edx, byte ptr [r13+2]
```

Figure C.1: An assembly fragment obfuscated by *O-LLVM* Instruction Substitution. Left: the original fragment. Right: the obfuscated fragment.

- *SUB* substitutes fragments of assembly code to its equivalent form by going one pass over the function logic using predefined rules. This technique modifies the contents of basic blocks and adds new constants. *SUB* does not change much of the graph structure. Figure C.1 shows an example. Figure C.3 shows that it increases the length of the original assembly function.

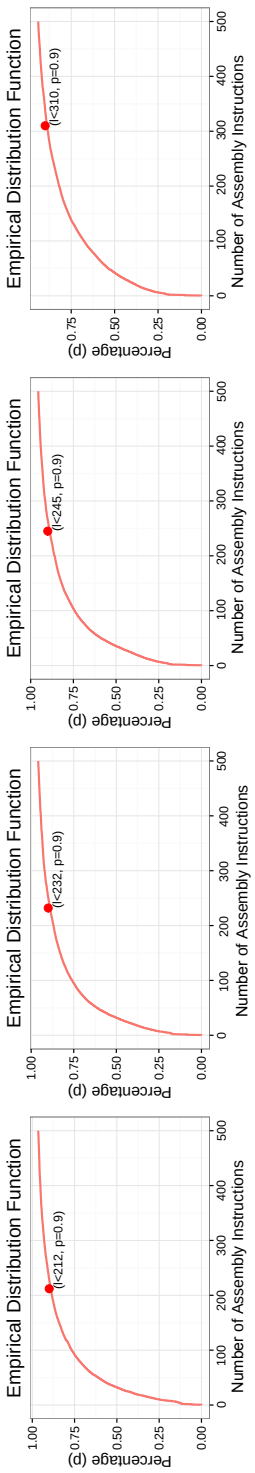

Figure C.2: The empirical distribution function of the assembly function length in terms of number of instructions. From left to right, each diagram corresponds to the optimization options O0, O1, O2, and O3.

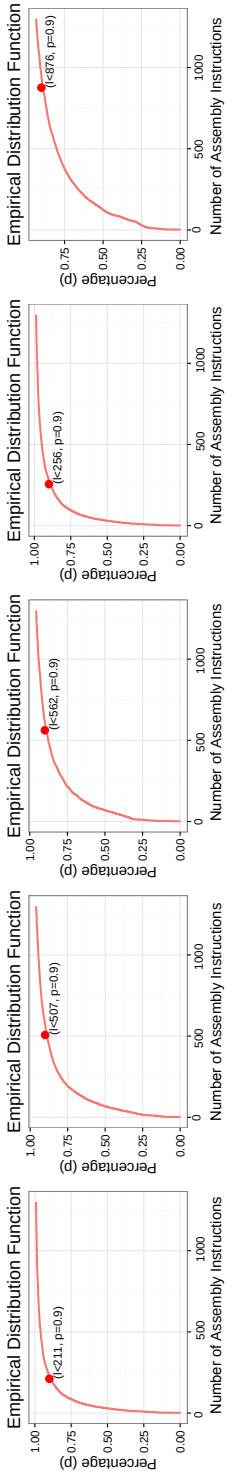

Figure C.3: The empirical distribution function of the assembly function length in terms of the number of instructions. Each diagram from left to right corresponds to the original binary, the obfuscation techniques Bogus Control Flow Graph, Control Flow Flattening, Instruction Substitution, and the application of all the obfuscation techniques provided by *Obfuscator-LLVM*.

D

Cuckoo Behavioral Indicators

Table D.1: Description of each behavioral descriptor.

Indicator	Files	Description
allocates execute remote process	8919	allocates execute permission to another process indicative of possible code injection
allocates rwx	23789	allocates read-write-execute memory (usually to unpack itself)
antianalysis detectfile	31	attempts to identify installed analysis tools by a known file location
antiav avast libs	71	detects avast antivirus through the presence of a library
antiav detectreg	129	attempts to identify installed av products by registry key
antiav servicestop	104	attempts to stop active services
antidbg devices	154	checks for the presence of known devices from debuggers and forensic tools
antidbg windows	241	checks for the presence of known windows from debuggers and forensic tools
antiemu wine	3071	detects the presence of wine emulator
antisandbox cuckoo files	31	attempts to detect cuckoo sandbox through the presence of a file

Continued from the preceding table.

Indicator	Files	Description
antisandbox foregroundwindows	1494	checks whether any human activity is being performed by constantly checking whether the foreground window changed
antisandbox idletime	104	looks for the windows idle time to determine the uptime
antisandbox mouse hook	4259	installs a hook procedure to monitor for mouse events
antisandbox sleep	1639	a process attempted to delay the analysis task
antisandbox unhook	24	tries to unhook windows functions monitored by cuckoo
antivm disk size	4905	queries the disk size, which could be used to detect virtual machine with small fixed size or dynamic allocation
antivm generic bios	273	checks the version of bios
antivm generic cpu	909	checks the cpu name from registry
antivm generic disk	196	queries information on disks
antivm generic scsi	2929	detects virtualization software with scsi disk identifier trick(s)
antivm generic services	24	enumerates services
antivm memory available	19684	checks amount of memory in system
antivm network adapters	13470	checks adapter addresses that can be used to detect virtual network interfaces
antivm queries computername	11850	queries for the computername
antivm sandboxie	24	tries to detect sandboxie
antivm vbox devices	41	detects virtualbox through the presence of a device
antivm vbox files	151	detects virtualbox through the presence of a file

Continued from the preceding table.

Indicator	Files	Description
antivm vbox keys	3110	detects virtualbox through the presence of a registry key
antivm vmware files	124	detects vmware through the presence of various files
antivm vmware in instruction	3090	detects vmware through the in instruction feature
antivm vmware keys	28	detects vmware through the presence of a registry key
av detect china key	62	checks for known Chinese av software registry keys
banker bancos	2782	creates known bancos banking Trojan files
banker zeus p2p	22	zeus p2p (banking trojan)
browser security	120	attempts to modify browser security settings
browser startpage	5205	attempts to modify internet explorer's start page
bypass firewall	171	operates on local firewall's policies and settings
checks debugger	4461	checks if process is being debugged by a debugger
console output	6459	command line console output was observed
creates doc	59	creates (office) documents on the filesystem
creates hidden file	227	creates hidden or system file
creates largekey	28	creates or sets a registry key to a long series of bytes
creates service	621	creates a service
creates shortcut	626	creates a shortcut to an executable file
credential dumping lsass	94	locates and dumps memory from the lsass.exe process indicative of credential dumping
cryptomining stratum command	41	a stratum cryptocurrency mining command was executed
dead host	109	connects to ip addresses that are no longer responding to requests (legitimate services will remain up-and-running usually)

Continued from the preceding table.

Indicator	Files	Description
deletes executed files	5967	deletes executed files from disk
detect putty	310	putty files
disables app launch	35	modifies system policies to prevent the launching of specific applications or executables
disables proxy	1417	disables proxy possibly for traffic interception
disables security	173	disables windows security features
disables system restore	71	attempts to disable system restore
dropper	1270	drops a binary and executes it
dumped buffer	9037	one or more potentially interesting buffers were extracted
dumped buffer2	5186	one or more of the buffers contains an embedded pe file
dyreza	39	creates known dyreza banking Trojan files
exe appdata	14876	drops an executable to the user appdata folder
generates crypto key	1026	uses windows apis to generate a cryptographic key
has pdb	2039	this executable has a pdb path
has wmi	1618	executes one or more wmi queries
infostealer bitcoin	26	attempts to access bitcoin/altcoin wallets
infostealer browser	2118	steals private information from local internet browsers
infostealer ftp	589	harvests credentials from local ftp client softwares
infostealer im	364	harvests information related to installed instant messenger clients
infostealer keylogger	4958	creates a windows hook that monitors keyboard input (keylogger)
infostealer mail	628	harvests credentials from local email clients
injection createremotethread	262	creates a thread using createremotethread in a non-child process indicative of process injection

Continued from the preceding table.

Indicator	Files	Description
injection modifies memory	436	manipulates memory of a non-child process indicative of process injection
injection network traffic	22	network communications indicative of possible code injection originated from the process explorer.exe
injection ntsetcontextthread	8793	used ntsetcontextthread to modify a thread in a remote process indicative of process injection
injection process search	1820	searches running processes potentially to identify processes for sandbox evasion
injection resumethread	15498	resumed a suspended thread in a remote process potentially indicative of process injection
injection runpe	8948	executed a process and injected code into it
injection write memory	8385	potential code injection by writing to the memory of another process
injection write memory exe	3449	code injection by writing an executable or dll to the memory of another process
installs bho	84	installs a browser helper object to thwart the users browsing experience
locates browser	1899	tries to locate where the browsers are installed
locates sniffer	96	tries to locate whether any sniffers are installed
locker taskmgr	27	disables windows' task manager
malicious document urls	171	potentially malicious url found in document
memdump urls	1347	potentially malicious urls were found in the process memory dump
modifies certificates	173	attempts to create or modify system certificates
modifies proxy wpad	10200	sets or modifies wpad proxy autoconfiguration file for traffic interception

Continued from the preceding table.

Indicator	Files	Description
modifies security center warnings	128	modifies security center warnings
modify uac prompt	24	attempts to modify uac prompt behavior
moves self	492	moves the original executable to a new location
multiple useragents	5594	network activity contains more than one unique useragent
network bind	40	starts servers listening
network document file	171	network communications indicative of a potential document or script payload download was initiated by the process wscript.exe
network icmp	17491	generates some icmp traffic
network wscript downloader	171	wscript.exe initiated network communications indicative of a script based payload download
origin langid	1239	foreign language identified in pe resource
packer entropy	10526	the binary likely contains encrypted or compressed data indicative of a packer
packer polymorphic	4617	creates a slightly modified copy of itself
packer upx	3673	the executable is compressed using upx
packer vmprotect	95	the executable is likely packed with vmprotect
pe features	13867	the executable contains unknown pe section names indicative of a packer (could be a false positive)
pe unknown resource name	11715	the file contains an unknown pe resource name possibly indicative of a packer
peid packer	5103	the executable uses a known packer
persistence ads	139	creates an alternate data stream (ads)
persistence autorun	7065	installs itself for autorun at windows startup
privilege luid check	1900	checks for the locally unique identifier on the system for a suspicious privilege

Continued from the preceding table.

Indicator	Files	Description
process interest	587	expresses interest in specific running processes
process martian	139	one or more martian processes was created
process needed	457	repeatedly searches for a not-found process
protection rx	6611	changes read-write memory protection to read-execute (probably to avoid detection when setting all rwx flags at the same time)
queries programs	5120	queries for potentially installed applications
raises exception	11804	one or more processes crashed
ransomware appends extensions	57	appends a new file extension or content to 71 files indicative of a ransomware file encryption process
ransomware dropped files	51	drops 835 unknown file mime types indicative of ransomware writing encrypted files back to disk
ransomware extensions	72	appends a known multi-family ransomware file extension to files that have been encrypted
ransomware file moves	60	performs 71 file moves indicative of a ransomware file encryption process
ransomware mass file delete	70	deletes a large number of files from the system indicative of ransomware
ransomware message	36	writes a potential ransom message to disk
ransomware shadowcopy	41	removes the shadow copy to avoid recovery of the system
rat fynloski	91	creates known fynloski/darkcomet files
rat xtreme	27	creates known xtremerat files
reads user agent	186	reads the systems user agent and subsequently performs requests
recon beacon	67	a process performed obfuscation on information about the computer or sent it to a remote location indicative of cnc traffic/preparations

Continued from the preceding table.

Indicator	Files	Description
recon fingerprint	3103	collects information to fingerprint the system (machineguid)
recon programs	4175	collects information about installed applications
removes zoneid ads	415	attempts to remove evidence of file being downloaded from the internet
self delete bat	65	creates and runs a batch file to remove the original binary
spreading autoruninf	84	creates an autorun.inf file
stealth hidden extension	115	attempts to modify explorer settings to prevent file extensions from being displayed
stealth hiddenfile	627	attempts to modify explorer settings to prevent hidden files from being displayed
stealth hide notifications	28	attempts to modify user notification settings
stealth system procname	4638	created a process named as a common system process
stealth window	7107	a process created a hidden window
stops service	66	stops windows services
suspicious command tools	116	uses suspicious command line tools or windows utilities
suspicious powershell	27	creates a suspicious powershell process
suspicious process	7149	creates a suspicious process
suspicious write exe	319	the process wscript.exe wrote an executable file to disk, which it then attempted to execute
sysinternals tools usage	91	uses sysinternals tools in order to add additional command line functionality
terminates remote process	269	terminates another process

Continued from the preceding table.

Indicator	Files	Description
uses windows utilities	10598	uses windows utilities for basic windows functionality
wmi antivm	1020	executes one or more wmi queries that can be used to identify virtual machines

E

Falcon Behavioral Indicators

Table E.1: Description of each behavioral descriptor.

Indicator	Files	Description
adware_0	38	possibly checks for the presence of an adware detecting tool
credential_stealer_0	390	scans for artifacts that may help identify the target
evasive_0	66	possibly checks for the presence of a forensicsmonitoring tool
evasive_1	29	executes wmi queries known to be used for vm detection
evasive_11	452	possibly tries to evade analysis by sleeping many times
evasive_14	107	references security related windows services
evasive_17	177	reads the windows product ID
evasive_18	513	possibly checks for the presence of an antivirus engine
evasive_19	107	reads the keyboard layout followed by a significant code branch decision
evasive_3	40	may try to detect deepfreeze frozen state

Continued from the preceding table.

Indicator	Files	Description
evasive_5	24	reads the registry for vmware specific artifacts
evasive_6	231	reads the systemvideo bios version
evasive_7	41	queries firmware table information may be used to fingerprintevade
evasive_8	371	tries to sleep for a long time more than two minutes
fingerprint_13	4615	reads the active computer name
fingerprint_3	27	contains ability to look up the windows account name
fingerprint_4	255	tries to identify its external ip address
fingerprint_6	2573	reads the cryptographic machine guid
fingerprint_8	196	found a dropped file containing the windows username possible fingerprint attempt
fingerprint_9	216	reads the windows installation date
network	3398	contact hosts or domains
persistence_0	716	spawns a lot of processes
persistence_1	34	persists itself using autoexecute at a hidden registry location
persistence_13	78	schedules a task to be executed at a specific time and date
persistence_14	3016	writes data to a remote process
persistence_15	36	modifies firewall settings
persistence_16	235	injects into explorer
persistence_18	83	modifies system certificates settings
persistence_2	54	creates a fake system process
persistence_3	939	modifies autoexecute functionality by settingcreating a value in the registry
persistence_4	248	injects into remote processes

Continued from the preceding table.

Indicator	Files	Description
persistence_6	288	interacts with the primary disk partition drnum
ransomware_0	31	deletes volume snapshots often used by ransomware
ransomware_1	48	detected indicator that file is ransomware
ransomware_5	33	contains ability to createswitch the desktop
ransomware_7	50	the analysis extracted a known ransomware file
remote_access_0	344	contains ability to listen for incoming connections
remote_access_1	354	contains a remote desktop related string
remote_access_6	3925	reads terminal service related keys often rdp related
remote_access_7	734	uses network protocols on unusual ports
spreading_1	204	tries to access unusual system drive letters
spreading_2	29	detected a large number of arp broadcast requests network device lookup
spreading_3	1292	opens the mountpointmanager often used to detect additional infection locations
spyware/leak_0	720	contains ability to retrieve keyboard strokes
spyware/leak_1	859	posts files to a webserver
spyware/leak_3	720	contains ability to open the clipboard
spyware_2	34	sets a global windows hook to intercept keystrokes
spyware_5	884	accesses potentially sensitive information from local browsers
stealer/phishing_2	445	tries to steal ftp credentials
stealer/phishing_4	111	touched instant messenger related registry keys