Title
Graph Learning for Robust Embedded and Cyber-Physical Systems

Author
YU, SHIH-YUAN

Publication Date
2023

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA,
IRVINE

Graph Learning for Robust Embedded and Cyber-Physical Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Shih-Yuan Yu

Dissertation Committee:
Professor Mohammad Al Faruque, Chair
Professor Pramod P. Khargonekar
Assistant Professor Yanning Shen

2023

Chapter 3 is adapted from the Arxiv version of the paper submitted to ICSE SEIP 2023.
The work involved in Chapter 3 is supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract Number N66001-20-C-4024.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I first want to thank my committee chair, Professor *Mohammad Abdullah Al Faruque*, for allowing me to return from industry to academia and pursue my Ph.D. There were ups and downs; however, Professor Al Faruque constantly guided me through the research and gave me valuable insights. Without his excellent supervision and persistent mentoring, I would not have been able to complete this thesis. I also want to thank my Ph.D. committee members: Professor *Pramod Khargonekar* and Assistant Professor *Yanning Shen*, for sparing precious time from their hectic schedules to provide supervision, support, and insightful comments for the thesis. Besides, I appreciate the help from Assistant Professor *Zhou Li* and *Alfred Chen* during my qualification examination.

I thank Dr. *Sujit Rokka Chhetri* for engaging and inspiring me to get involved in novel research ideas. Without his mentorship, I would not have gained this many achievements throughout my Ph.D. I thank my fellow lab-mates for engaging me in stimulating discussions. I particularly want to appreciate the precious time and efforts spared by *Arnav Malawade* and *Rozhin Yasaei*. Together, we have made some marks in the research communities. My sincere thanks go to my fellow collaborators, Dr. *Deepan Muthirayan*, Dr. *Gustavo Quirós*, Dr. *Arquimedes Canedo*, and Adjunct Professor *Lawrence Kulinsky*; they gave me opportunities to widen my vision and the scope of my research. Besides, I want to thank the principal investigators and leaders in Siemens Technologies, Dr. *Nemeth Laszlo*, Assistant Professor *Sharma Tushar*, Dr. *Ramamurthy Arun*, Dr. *Akrotirianakis Ioannis*, Dr. *Amar Viswanathan* for allowing me to collaborate with their research groups and broaden my horizons. I wish everyone the best in enjoying their following research journeys.

I also thank my fellow colleagues from the *Autonomous and Intelligent Cyber-Physical Systems* (AICPS) laboratory, *Mohanad Odema*, *Luke Chen*, *Yonatan Gizachew*, *Ahmet Aksakal*, *Trier Mortlock*, *Nafiul Rashid*, and *Anomadarshi Barua* for collaborating in various research endeavors. I want to thank many of my undergraduate students, *Aung Myat Thu*, *Tommy Nguyen*, *Emily Lam*, *Yasamin Moghaddas*, *FangZhou Du*, *George Chin-Yuan Ting*, *Qingrong Zhou*, *Chonghan Wang*, and *Lelin Pan*, for making contributions to my research works. I wish everyone the best in their lives.

Last but not least, I would like to thank my beloved mother and father-like brother for their unconditional love and belief in me. I want to appreciate the social support from my miscellaneous friends for making these difficult times relaxing. Particularly, I want to express my since gratitude to my wife, *Pin-Chun Chen*, for the much-needed love and support that helped me carry on through my Ph.D. during the unfortunate pandemic.

VITA

Shih-Yuan Yu

EDUCATION

Doctor of Philosophy in Computer Engineering **2023**
University of California, Irvine (UCI), *Irvine, California*

Master of Science in Computer Science and Information Engineering **2014**
National Taiwan University (NTU), *Taipei, Taiwan*

Bachelor of Science in Computer Science and Information Engineering **2012**
National Taiwan University (NTU) *Taipei, Taiwan*

RESEARCH EXPERIENCE

Graduate Research Assistant **Sep. 2018 - Jun. 2023**
University of California, Irvine (UCI) *Irvine, California*

Compute Systems Performance & Reliability Intern **Jun.-Sep. 2022**
TuSimple Inc. *San Diego, California*

Automation Runtime Systems Internship **Jun.-Sep. 2021**
Siemens Technology *Princeton, New Jersey*

Research Assistant **Jul. 2012 - Jun. 2014**
National Taiwan University (NTU) *Taipei, Taiwan*

TEACHING EXPERIENCE

Teaching Assistant for EECS 111 **2023 Spring**
System Software *Irvine, California*

Teaching Assistant for EECS 118 **2023 Winter**
Introduction to Artificial Intelligence *Irvine, California*

Teaching Assistant for EECS 111 **2022 Spring**
System Software *Irvine, California*

REFERRED JOURNAL AND CONFERENCE PUBLICATIONS

CFG2VEC: Hierarchical Graph Neural Network for Cross-Architectural Software Reverse Engineering **2023**
International Conference on Software Engineering / Software Engineering in Practice

Hardware Trojan Detection Using Graph Neural Networks **2022**
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Roadscene2vec: A Tool for Extracting and Embedding Road Scene-Graphs **2022**
Knowledge-Based Systems 242, 108245

Artificial Intelligence Algorithms Enable Automated Characterization of The Positive and Negative Dielectrophoretic Ranges of Applied Frequency **2022**
Micromachines 13 (3), 399

Spatiotemporal Scene-Graph Embedding for Autonomous Vehicle Collision Prediction **2022**
IEEE Internet of Things Journal 9 (12), 9379-9388

HW2VEC: A Graph Learning Tool for Automating Hardware Security **2021**
IEEE International Symposium on Hardware Oriented Security and Trust

Graph Learning for Cognitive Digital Twins in Manufacturing Systems **2021**
IEEE Transactions on Emerging Topics in Computing 10 (1), 34-45

Establishing Digital Recognition and Identification of Microscopic Objects for Implementation of Artificial Intelligence (AI) Guided Microassembly **2021**
Proceedings of World Congress on Micro and Nano Manufacturing 2021

Scene-Graph Augmented Data-Driven Risk Assessment of Autonomous Vehicle Decisions **2021**
IEEE Transactions on Intelligent Transportation Systems

Cognitive Digital Twin for Manufacturing Systems **2021**
IEEE/ACM Design Automation and Test in Europe Conference (DATE'21)

GNN4TJ: Graph Neural Networks for Hardware Trojan Detection at Register Transfer Level **2021**
IEEE/ACM Design Automation and Test in Europe Conference (DATE'21)

GNN4IP: Graph Neural Network for Hardware Intellectual Property Piracy Detection **2021**
Design automation Conference (DAC)

Pykg2vec: A python library for knowledge graph embedding **2021**
The Journal of Machine Learning Research 22 (1), 754-759

Multimodal Knowledge Graph for Deep Learning Papers and Code **2020**
Conference on Information and Knowledge Management (CIKM)

Sabotage Attack Detection for Additive Manufacturing Systems **2020**
IEEE Access

ABSTRACT OF THE DISSERTATION

Graph Learning for Robust Embedded and Cyber-Physical Systems

By

Shih-Yuan Yu

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2023

Professor Mohammad Al Faruque, Chair

Since the appearance of microprocessors, miscellaneous categories of computer systems, including *Embedded and Cyber-Physical Systems* (ECPS), have become an integral part of our modern society. In contrast to general-purpose computers, an ECPS is a computing system designed to perform a dedicated or narrow range of functions with minimal human intervention. Today, the advanced standards in device networking, ubiquitous access to the Internet, miniaturization of processors, and reduced power consumption have taken this field to the next level. However, despite these technological growths, designing a robust ECPS remains an open research challenge where the goal is to achieve better functionalities, encompass complex, uncertain, and changing environments, and ensure system security. Minor failures on ECPS may cause severe collapses or cyberattacks, impeding progress toward increasing automation and modernizing our computing environments.

To achieve robustness, this dissertation studies the embedding of *Computational Intelligence* (CI) into emerging *Graph Learning* (GL) technologies. The CI paradigms mimic the nature of humans, aiming at solving complex problems and exhibiting a cognitive ability to learn or adapt to new situations to generalize, abstract, discover, and associate. As a prerequisite of this research direction, *Machine Learning* (ML) has become increasingly ubiquitous, with existing works exploring various fields, from self-driving vehicles, facial recognition systems,

and real-time language translation to security surveillance, innovative home applications, and health monitoring. However, conventional ML algorithms typically require appropriate vectorized representations crafted by domain experts to accomplish the desired goals. Graph-structured data have imposed unprecedented challenges on ML due to their inherent complexity. Unlike text, audio, and images, graphs are embedded in an irregular dimension, making some essential operations of ML inapplicable. GL has attracted much attention to new research ideas in several fields. To date, many researchers have proven the usefulness of GL in social computing, information retrieval, computer vision, bioinformatics, economics, and e-commerce. However, its applications in the subfields of ECPS still need to be explored.

In this dissertation, we will cover GL applications for ECPS in robust Integrated Circuit (IC) design analysis for hardware security, robust binary analysis for enabling software security, and ultimately how GL brings the scene-understanding capabilities of autonomous driving systems to the next level. In IC design analysis (Chapter 2), we explore how GL can be leveraged to resolve challenging problems in hardware security. We propose *HW2VEC* and demonstrate how it can be utilized for *Hardware Trojan* (HT) detection and *Intellectual Property* (IP) piracy detection. Next, in the binary analysis (Chapter 3), we demonstrate how a more advanced GL methodology, called *CFG2VEC*, can reverse-engineer the semantic source-level information lost during the compilation process, thus making the binary software patching tasks more efficient. Then, we describe our novel methodology, *SG2VEC*, to enhance autonomous driving systems' scene understanding capabilities (Chapter 4). To enable this cognitive capability, we propose to use scene-graph to encode the surrounding traffic participants and a pipeline of spatiotemporal scene-graph embedding networks to process scene-graphs and learn toward goals.

Chapter 1

Introduction and Background

1.1 Embedded System

An *Embedded System* (ES) can be broadly defined as a device that contains tightly coupled hardware and software components to perform a single function [94]. Since the decade 1970, this term has been widely used as we have witnessed the development of the first microprocessor design, TMS1000, which was used in calculators. Typically, an ES forms a part of a larger computer system and is designed not to be independently programmable by the user. Most ES directly interacts with the environments, making decisions in real-time based on their inputs. Therefore, it has to be reactive, consuming inputs in real-time and ensuring proper operations given their outputs. Regardless of the function performed, an ES comprises two components, hardware components (e.g., microcontroller) and software components (e.g., firmware). The ES takes the system inputs and coordinates the hardware component and the software component, eventually determining the system outputs sent back to the human or environment. As time goes on, the complexity of ES grows significantly, and it has been for a more complicated application purpose where the system has to be broken

into several interacting ESs. For example, for a multimedia player, the system includes audio input/output, a digital camera, a video processing system, a hard drive, a user interface (keys, a touchscreen, and a graphic display), power management, and digital communication components where each of these components is supported individually by an ES integrated into the application system. Modern ES has become ubiquitous in almost every aspect of our modern lives, including cellular phones, televisions, and automobiles, and even hidden in some safety-critical systems such as anti-lock brakes, electronic surveillance, and defense systems.

1.2 Cyber-Physical System and Robustness

More recently, around 2006, the terminology *Cyber-Physical System* (CPS) emerged, coined by Helen Gill at the *National Science Foundation* (NSF). As an advanced version of ES, a full-fledged CPS is typically designed as a "network" of interacting elements with physical input and output instead of as standalone devices [87]. CPS consists of a complex interaction between heterogeneous (different types of computation and communication platforms) and hybrid (discrete and continuous) components. The goal of CPS is to bridge the cyber-domain with physical processes where sensing, actuation, computation, and communication functions are deeply integrated to improve the overall performance, security, and reliability of physical systems. This deep integration has broadened the potential of CPSs in several directions, including intervention (e.g., collision avoidance); precision (e.g., robotic surgery and nano-level manufacturing); operation in dangerous or inaccessible environments (e.g., search and rescue, firefighting, and deep-sea exploration); coordination (e.g., air traffic control, war-fighting); efficiency (e.g., zero-net energy buildings); and augmentation of human capabilities (e.g., in healthcare monitoring and delivery). Today, the advanced standards in device networking, ubiquitous access to the Internet, miniaturization of processors, reduc-

tion in power consumption, and availability of Simulators and Emulators for system design have taken this field to a new generation which is called *Intelligent Cyber-Physical Systems* (iCPS) [47]. The growing demand for autonomy and the need to reduce both the decision-making time and the transmission bandwidth are pushing designers to integrate CPSs with intelligent mechanisms, ultimately reaching real Automation and improving the quality of human life. However, despite these technological growths, designing an iCPS is challenging as it has to be robust against complex, uncertain, and changing environments. Minor system failures may cause severe collapses or cyberattacks, impeding progress toward increasing automation and modernizing our computing environments.

As shown in the literature, a robust design of CPS design includes three features (3S) [86], which *stability*, *security* and *systematicness*. For stability, the CPS system should always reach a stable decision result eventually, regardless of how the environment generates noise and uncertain factors. For security, CPSs should be able to detect and countermeasure the cyber–physical interaction attacks. Lastly, systematicness refers to the fact that the cyber and physical components should be seamlessly integrated together into a systematic design. In the past decade, it has been seen that *Artificial Intelligence* has become increasingly ubiquitous in everyday life, such as self-driving vehicles, facial recognition systems, and real-time interpretation of different languages to security surveillance, innovative home applications, and health monitoring. Particularly, *Machine Learning* has been explored in making CPS intelligent, aiming at automating the processes as far as possible and achieving autonomy. One example of such a vision is Industry 4.0, in which emerging technologies such as collaborative robotics, big data, internet of things, artificial intelligence, and virtual and augmented reality jointly create a new paradigm, offering a more automated and efficient system [103]. In this case, self-X properties are of paramount importance in making these CPSs self-adapt or self-organize to cope with high dynamism and high changing environments [56], and such a capability relies on a proper cognition level of the system.

1.3 Why Graph Learning?

To date, many intelligent systems generally rely on ML algorithms handling various types of data. However, despite their ubiquity, graph data have imposed unprecedented challenges on ML due to their inherent complexity. Unlike text, audio, and images, graph data are embedded in an irregular domain, making some essential operations of existing ML algorithms inapplicable. For this, many research works have focused on this new field of study, *Graph Learning* (GL) [176]. Graphs, also called networks, can represent many real-world relations among various entities. A graph is often defined by two sets, i.e., node set and edge set, where the node represents the entity in a graph, and the edge stands for the relationship between a pair of entities. GL refers to ML on graphs where the methods of GL can map the features of a graph to feature vector(s) with the same dimensions in the embedding space. The goal of GL models is to extract the desired features of a graph and the output can be easily used by downstream tasks such as node classification, link prediction, and graph classification without an explicitly embedding process or feature engineering process. Many researchers and practitioners have proven the usefulness of GL in different areas, such as social computing, information retrieval, computer vision, bioinformatics, economics, and e-commerce.

To increase the robustness of ESs and CPSs, this dissertation studies the embedding of *Computational Intelligence* (CI) with the emerging *Graph Learning* (GL) technologies. *Computational Intelligence* (CI) paradigms involve adaptive mechanisms to facilitate intelligent behavior in a complex, uncertain, and changing environment. These paradigms mimic the nature of solving complex problems and exhibit an ability to learn or adapt to new situations to generalize, abstract, discover, and associate. Recently, cognitive intelligence (a.k.a. cognitive computing) [33]has attracted much research attention in various fields originating from neuroscience. One key in cognitive computing is that the human brain consists of two parts: the first relates to distinguishment and determination (perception), and the second

relates to reasoning and explanation. Generally speaking, it aims at understanding the presented information, interpreting its contextual meaning, and drawing deductions through correlation analysis or casual analysis. This derives an idea of *Intermediate Representation* (IR) where a human also adopts a similar mechanism during the cognitive process as it has a computational advantage for only considering a modest number of primitives and their relations and perceptual advantage in viewpoint invariance and robustness to noise based on qualitative discriminations [21]. However, most of the ML-based approaches implicitly assume the existence of the first part (in vectorized form) through the proper manual feature engineering by domain experts, only automating the second part through various types of ML models and learning processes.

1.4 Research Scope

In this dissertation, we will explore how to utilize GL to automate the first part (e.g., perception) and make the system more robust regarding performance, generalization, and explainability. As shown in Figure 1.1, specifically, we will cover GL applications for ES and CPS in robust Integrated Circuit (IC) design analysis for hardware security, robust binary analysis for enabling software security, and ultimately how GL brings the scene-understanding capabilities of autonomous driving systems to the next level. In IC design analysis (Chapter 2), we explore how GL can be leveraged to resolve challenging problems in hardware security. We propose *HW2VEC* and demonstrate how it can be utilized for *Hardware Trojan* (HT) detection and *Intellectual Property* (IP) piracy detection. Next, in the binary analysis (Chapter 3), we demonstrate how a more advanced GL methodology, called *CFG2VEC*, can reverse-engineer the semantic source-level information lost during the compilation process, thus making the binary software patching tasks more efficient. Then, we describe our novel methodology, *SG2VEC*, to enhance autonomous driving systems' scene

understanding capabilities (Chapter 4). To enable this cognitive capability, we propose to use scene-graph to encode the surrounding traffic participants and a pipeline of spatiotemporal scene-graph embedding networks to process *scene-graphs* and learn toward goals. During the courses of these projects, we have developed many research tools to confront the technical challenges (mentioned in Appendices). Lastly, we conclude the thesis with key findings and future steps in Chapter 5.

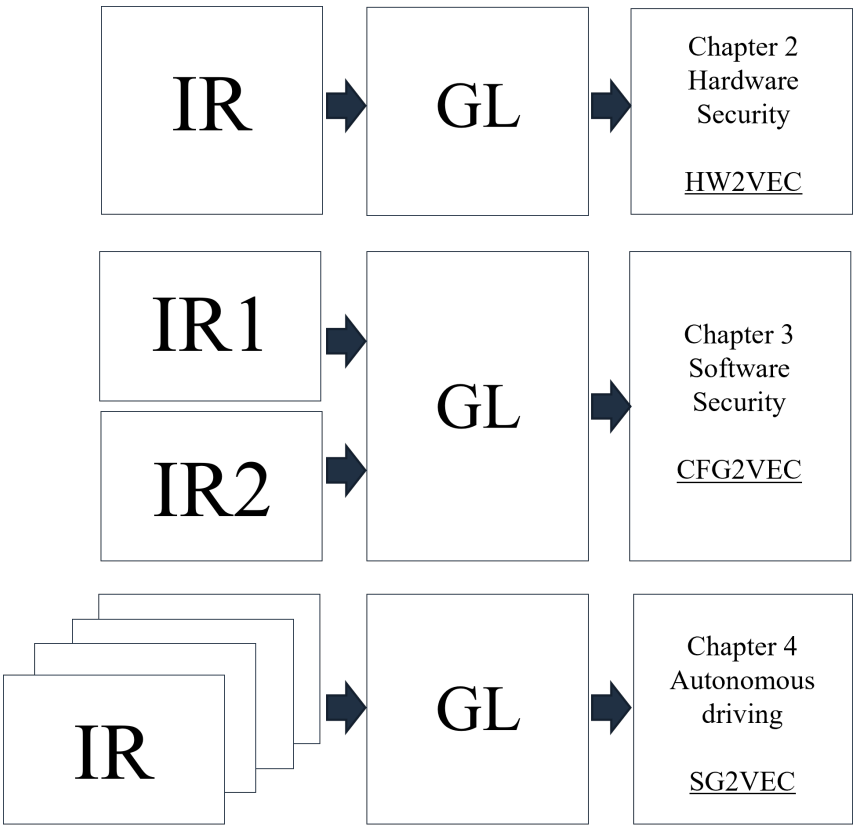

Figure 1.1: The illustration of the research scope in this dissertation.

Chapter 2

Graph Learning for Enhancing the Security of Hardware Designs

The globalized *Integrated Circuit* (IC) supply chain has brought various security threats in each phase of the chain. Although the counteracting solutions based on *Machine Learning* (ML) exist, most can only achieve the desired performance when security experts define a robust feature extraction process for IC designs. As the hardware designs are non-Euclidean data, designing a robust feature extraction process is challenging and requires manual effort. In this chapter, as a result, I will discuss my proposed research tool *HW2VEC* that models these IC designs using *Graph Learning* to enhance the early phases of the IC supply chain.

2.1 Introduction

In past decades, the growing design complexity and the time-to-market pressure have jointly contributed to the globalization of the *Integrated Circuit* (IC) supply chain [146]. Along this globalized supply chain, IC designers tend to leverage third-party *Electronic Design*

Automation (EDA) tools and *Intellectual Property* (IP) cores or outsource costly services to reduce their overall expense. This results in a worldwide distribution of IC design, fabrication, assembly, deployment, and testing [24, 95, 139]. However, such globalization can also make the IC supply chain vulnerable to various hardware security threats such as *Hardware Trojan Insertion*, *IP Theft*, *Overbuilding*, *Counterfeiting*, *Reverse Engineering*, and *Covert & Side Channel Attacks* [143**?**].

As the consequences of not promptly addressing these security threats can be severe, countermeasures and tools have been proposed to mitigate, prevent, or detect security threats [88]. For example, hardware-based primitives, physical unclonable functions (PUFs) [82], true random number generator (TRNG) [128], and cryptographic hardware can all intrinsically enhance architectural security. The countermeasures built into hardware design tools are also critical for securing the hardware in the early phases of the IC supply chain. Some Machine Learning (ML) based approaches have been proven effective for detecting *Hardware Trojans* (HT) from hardware designs in both Register Transfer Level (RTL) and Gate-Level Netlist (GLN) [73, 77]. Besides, [90] automates the process of identifying the counterfeited ICs by leveraging Support Vector Machine (SVM) to analyze the sensor readings from on-chip hardware performance counters (HPCs). However, as indicated in [159], effectively applying ML models is a non-trivial task as the defenders must first identify an appropriate input representation based on hardware domain knowledge. Therefore, ML-based approaches can only achieve the desired performance with a robust feature representation of a circuit (non-Euclidean data) which is more challenging to acquire than finding the one for Euclidean data such as images, texts, or signals.

In IC design flow, many fundamental objects such as netlists or layouts are natural graph representations [110]. These graphs are non-Euclidean data with irregular structures, thus making it hard to generalize basic mathematical operations and apply them to conventional Deep Learning (DL) approaches [30]. Also, extracting a feature that captures structural

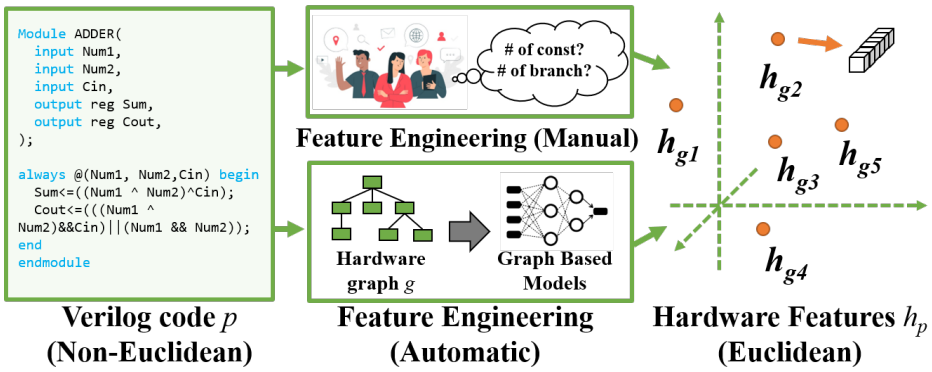

Figure 2.1: The illustration of the process that extracts features for hardware analysis.

information requires a non-trivial effort to achieve the desired performance. To overcome these challenges, many *graph learning* approaches such as *Graph Convolutional Networks* (GCN), *Graph Neural Networks* (GNN), or *Graph Autoencoder* (GAE) have been proposed and applied in various applications such as computer vision, natural language processing, and program analysis [98, 175]. In the EDA field, some works tackle netlists with GCNs for test point insertion [111] or with GNNs for fast and accurate power estimation in pre-silicon simulation [198]. As Figure 2.1 shows, these approaches typically begin with extracting the graph representation (g) from a hardware design p, then use the graph-based models as an alternative to the manual feature engineering process. Lastly, by projecting each hardware design onto the Euclidean space (h_g), these designs can be passed to ML models for learning tasks. However, only a few works have applied GNN-based approaches for securing hardware during IC design phases due to the lack of supporting tools [187, 188].

To attract more research attention to this field, we propose *HW2VEC*, an open-source graph learning tool for enhancing hardware security. *HW2VEC* provides automated pipelines for extracting graph representations from hardware designs and leveraging graph learning to secure hardware in design phases. Besides, *HW2VEC* automates the processes of engineering features and modeling hardware designs. To the best of our knowledge, *HW2VEC* is the first open-source research tool that supports applying graph learning methods to hardware designs in different abstraction levels for hardware security. In addition, *HW2VEC* supports

transforming hardware designs into various graph representations such as the *Data-Flow Graph* (DFG), or the *Abstract Syntax Tree* (AST). In this paper, we also demonstrate that *HW2VEC* can be utilized in resolving two hardware security applications: *Hardware Trojan Detection* and *IP Piracy Detection* and can perform as good as the state-of-the-art GNN-based approaches.

Research Contribution: for the hardware security research community,

- We propose an automated pipeline to convert a hardware design in RTL or GLN into various graph representations.
- We propose a GNN-based tool to generate vectorized embeddings that capture the behavioral features of hardware designs from their graph representations.
- We demonstrate *HW2VEC*'s effectiveness by showing that it can perform similarly compared to state-of-the-art GNN-based approaches for various real-world hardware security problems, including *Hardware Trojan Detection* and *IP Piracy Detection*.
- We open-source *HW2VEC* as a Python library[1] to contribute to the hardware security research community.

2.2 Related Works

This section depicts hardware security problems in the IC supply chain and countermeasures against these security threats. Then, it covers the research works related to ML-based approaches for hardware security. Lastly, we introduce the works that utilize graph learning methods in both EDA and hardware security.

[1]The *HW2VEC* is publicly available at `https://github.com/AICPS/hw2vec/`. Our readers can refer to [114] for more information about implementation.

2.2.1 Hardware Security Threats in IC Supply Chain

In the *Integrated Circuit* (IC) supply chain, each IC is passed through multiple processes, as shown in Figure 2.2. First, the specification of a hardware design is turned into a behavioral description written in a *Hardware Design Language* (HDL) such as Verilog or VHDL. Then, it is transformed into a design implementation in terms of logic gates (i.e., netlist) with *Logic Synthesis*. *Physical Synthesis* implements the netlist as a layout design (e.g., a GDSII file). Lastly, the resulting GDSII file is handed to a foundry to fabricate the actual IC. Once a foundry produces the IC (Bare Die), several tests are performed to guarantee its correct behavior. The verified IC is then packaged by the assembly and sent to the market to be deployed in systems.

For a *System-on-Chip* (SoC) company, all of the mentioned stages of the IC supply chain require a vast investment of money and effort. For example, it costs $5 billion in 2015 to develop a new foundry [189]. Therefore, to lower R&D costs and catch up with the competitive development cycle, an SoC company may choose to outsource the fabrication to a third-party foundry, purchase third-party IP cores, and use third-party EDA tools. The use of worldwide distributed third parties makes the IC supply chain susceptible to various security threats [177] such as *Hardware Trojan Insertion*, *IP Theft*, *Overbuilding*, *Counterfeiting*, *Reverse Engineering*, and *Covert & Side Channel Attacks*, etc. Not detecting or preventing these threats can lead to severe outcomes. For example, in 2008, a suspected nuclear installation in Syria was bombed by Israeli jets because a backdoor in its commercial off-the-shelf microprocessors disabled Syrian radar [3]. In another instance, the IP-intensive industries of the USA lose between $225 to $600 billion annually as the companies from China steal American IPs, mainly in the semiconductor industry [152].

Among the mentioned security threats, the insertion of *Hardware Trojan* (HT) can cause the infected hardware to leak sensitive information, degrade its performance, or even trigger

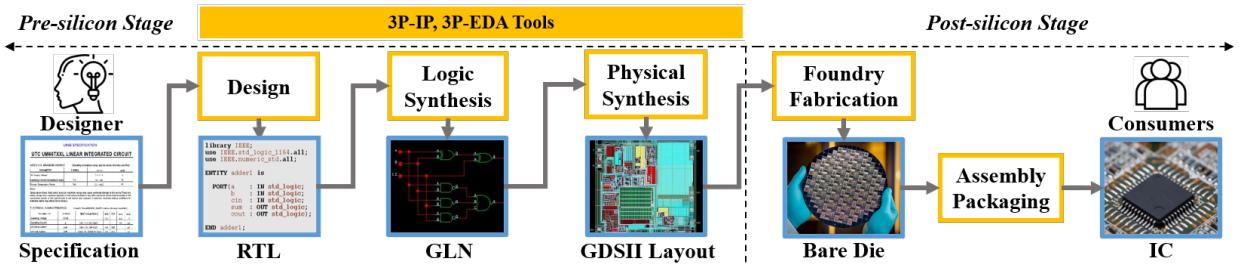

Figure 2.2: The illustration of the IC supply chain demonstrating the hardware design flow from a specification to the behavioral description (RTL), logic implementation (GLN), physical implementation (GDSII), and the actual chip (Bare Die or IC).

a Denial-of-Service (DoS) attack. In SoC or IC designs, *IP Theft*, the illegal usage and distribution of an IP core can occur. The third-party foundries responsible for outsourced fabrication can *overbuild* extra chips for their benefits without the designer's permission. Moreover, selling the *Counterfeited* designs in the name of its original supplier leads to financial or safety damage to its producer or even the national security if the target is within essential infrastructures or military systems. *Reverse engineering* (RE) recovers the high-level information from a circuit available in its lower-level abstraction. Although RE can be helpful in the design and verification process, an attacker can misuse the reconstructed IC designs for malicious intentions. *Covert Channel* uses non-traditional communication (e.g., shared cache) to leak critical information about a circuit. In contrast, *Side Channel* exists among the hardware components that are physically isolated or not even in proximity (e.g., power or electromagnetic channel) [199, 17, 5, 60, 40, 36, 39, 38, 35, 42, 44].

2.2.2 The Countermeasures Against Hardware Security Threats

Due to the globalization of the IC supply chain, hardware is susceptible to security threats such as IP piracy (unlicensed usage of IP), overbuilding (unauthorized manufacturing of the circuit), counterfeiting (producing a faithful copy of the circuit), reverse engineering, hardware Trojan (malicious modification of circuit), and side-channel attacks [11]. In the literature, countermeasures and tools have been proposed to mitigate, prevent, or detect

security threats [88]. For example, a cryptographic accelerator is a hardware-based countermeasure that can reinforce the build-in instead of the add-on defense against security threats. *True Random Number Generator* (TRNG) and *Physical Unclonable Function* (PUF) are two other effective security primitives [82, 128]. These solutions are critical for security protocols and unique IC identification, and they rely on the physical phenomena for randomness, stability, and uniqueness, such as process variations during fabrication [159]. In addition to hardware-based solutions, countermeasures enhancing security during the hardware design process are also present in the literature. For example, side-channel analysis for HT detection using various models such as hierarchical temporal memory [59, 18] and DL [58] has grabbed lots of attention recently. However, they postpone the detection to *post-silicon* stage.

To defend the HT earlier in the *pre-silicon* stage, most existing detection techniques fall into one of the following categories. First, as HT typically remains inconspicuous during testing and only gets triggered by a particular event, the authors of [142] attempt to propose an automated *Test Pattern Generation* method to generate effective test vectors and increase the probability of triggering the HT. However, it is infeasible to cover all the possible testing scenarios. Secondly, *Formal Verification* (FV) is a *pre-silicon* algorithmic method that converts the 3PIP to a proof checking format and checks if the IP satisfies some predefined security properties [93, 156]. Although FV leverages the predefined security properties in IP for HT detection, its detection scope is limited to certain types of HTs because they are not comprehensive enough to cover all kinds of malicious behaviors [132]. Some works employ model checking but are not scalable to large designs as model checking is NP-complete and can suffer from state explosion [133]. The third existing category is *Code Analysis* (CA), which analyzes the hardware design programs using metrics such as line, statement, finite state machine, and toggle coverage to ascertain the suspicious signals that imitate the HT [169, 201]. However, CA requires the designer to manually inspect the suspicious code regions and localize the HTs. Finally, the last category is the graph-based analysis methods as the graph is an intuitive representation of a hardware design. The authors of [125] propose

analyzing the data/control flow graph of the IC design to pinpoint the HTs by referencing a library of known HTs graphs and searching for the graph using sub-graph matching. However, sub-graph matching is NP-complete and thus incapable of applying to large designs. In more recent research, [65] proposes a new graph similarity heuristic customized for hardware security to improve accuracy and computation time. However, once again, it can only detect the HTs with the same graph representation as known HTs in their library, while in practice, attackers design a variety of HTs.

As for IP theft prevention, *watermarking* and *fingerprinting* are two approaches that embed the IP owner and legal IP user's signatures into a circuit to prevent infringement [127, 129]. *Hardware metering* is an IP protection method in which the designer assigns a unique tag to each chip for chip identification (passive tag) or enabling/disabling the chip (active tag) [100]. *Obfuscation* is another countermeasure for IP theft [34] which comprises two main approach; *Logic Locking* and *Camouflaging*. In *Logic Locking*, the designer inserts additional gates such as XOR into non-critical wires. The circuit will only be functional if the correct key is presented in a secure memory out of reach of the attacker [178]. *Camouflaging* modifies the design such that cells with different functionalities look similar to the attacker and confuses the reverse engineering process [131]. Lastly, another countermeasure is to split the design into separate ICs and have them fabricated in different foundries (e.g., *Split Manufacturing*) so that none of them can access the whole design to perform malicious activities [124, 200].

In [88], several academic and commercial tools have been proposed to secure hardware. For example, *VeriSketch*, *SecVerilog*, etc., are open-source academia verification tools for securing hardware. *SecureCheck* from *Mentor Graphics*, *JasperGold Formal Verification Platform* from *Cadence*, and *Prospect* from *Tortuga Logic* are all commercial verification tools ready in the market. *PyVerilog* [157] is a hardware design tool that allows users to parse HDL code and perform *pre-silicon* formal verification side-by-side with functional verification. In short, though many approaches have been proposed to counteract security threats, security

is still an afterthought in hardware design. Therefore, new countermeasures will be needed against new security threats.

2.2.3 Machine Learning for Hardware Security

In the last few decades, advancements in Machine Learning (ML) have revolutionized the conventional methods and models in numerous applications throughout the design flow. Defenders can use ML with hardware-based observations for detecting attacks, while attackers can also use ML to steal sensitive information from an IC, breaching hardware security [159]. Some ML-based countermeasures have been proven effective for detecting HT from hardware designs in both Register Transfer Level (RTL) or gate-level netlists (GLN) [73, 77]. In [73], the circuit features are extracted from the *Abstract Syntax Tree* (AST) representations of RTL codes and fed to a gradient boosting algorithm to train the ML model to construct an HT library. [77] extracts 11 Trojan-net feature values from GLNs and then trains a *Multi-Layer Neural Network* on them to classify each net in a netlist as a normal netlist or Trojan. Similarly, researchers have applied ML to automate the process of detecting other threats. For instance, SVM can be used to analyze the on-chip sensor readings (e.g., HPCs) to identify counterfeited ICs and detect HT in real-time [90, 101]. However, as indicated in [159], effectively applying ML models is not a trivial task, as the defenders must first identify an appropriate input representation for a hardware design. Unlike Euclidean data such as images, texts, or signals, finding a robust feature representation for a circuit (Non-Euclidean data) is more challenging as it requires domain knowledge in both hardware and ML. To overcome this challenge, *HW2VEC* provides more effective graph learning methods to automatically find a robust feature representation for a non-Euclidean hardware design.

2.2.4 Graph Learning for Hardware Design and Security

Although conventional ML and DL approaches can effectively capture the features hidden in Euclidean data, such as images, text, or videos, there are still various applications where the data is graph-structured. As graphs can be irregular, a graph can have a variable size of unordered nodes, and nodes can have a different number of neighbors, thus making mathematical operations used in deep learning (e.g., 2D Convolution) challenging to be applied [30]. Also, extracting a feature that captures structural information requires challenging efforts to achieve the desired performance. To address these challenges, recently, many *graph learning* approaches such as *Graph Convolutional Networks* (GCN), *Graph Neural Networks* (GNN), or *Graph Autoencoder* (GAE) have been proposed and applied in various applications [98, 175]. Only by projecting non-Euclidean data into low-dimensional embedding space can the operations in ML methods be applied.

In EDA applications, many fundamental objects such as Boolean functions, netlists, or layouts are natural graph representations [110]. Some works tackle netlists with GCNs for test point insertion [111] or with GNNs for fast and accurate power estimation in *pre-silicon* simulation [198]. [198] uses a GNN-based model to infer the toggle rate of each logic gate from a netlist graph for fast and accurate average power estimation without gate-level simulations, which is a slower way to acquire toggle rates compared to RTL simulation. They use GLNs, corresponding input port, and register toggle rates as input features and logic gate toggle rates as ground-truth to train the model. The model can infer the toggle rate of a logic gate from input features acquired from RTL simulation for average power analysis computed by other power analysis tools.

As for hardware security, only a few works utilizing GNN-based approaches against security threats exist [187, 188, 185]. [188] utilizes a GNN-based approach for detecting HT in *pre-silicon* design phases without the need for golden HT-free reference. Besides, using the GNN-

based approach allows the extraction of features from Data-Flow graphs to be automated. In [187], the proposed GNN-based approach can detect IP piracy without extracting hardware overhead to insert signatures to prove ownership. Specifically, the Siamese-based network architecture allows their approach to capturing the features to assess the similarity between hardware designs as a Data-Flow Graph. In short, these works have shown the effectiveness of securing hardware designs with graph learning approaches. To further attract attention, we propose *HW2VEC* as a convenient research tool that lowers the threshold for newcomers to make research progress and for experienced researchers to explore this topic more in-depth.

2.3 HW2VEC Architecture Introduction

As Figure 2.3 shows, *HW2VEC* contains HW2GRAPH and GRAPH2VEC. During the IC design flow, a hardware design can have various levels of abstraction, such as *High-Level Synthesis* (HLS), RTL, GLN, and GDSII, which are fundamentally non-Euclidean data. Overall, in *HW2VEC*, a hardware design p is first turned into a graph g by HW2GRAPH, which defines the pairwise relationships between objects that preserve the structural information. Then, GRAPH2VEC consumes g and produces the Euclidean representation h_g for learning.

2.3.1 HW2GRAPH: from hardware design to graph

The first step is to convert each hardware design code p into a graph g. HW2GRAPH supports the automatic conversion of raw hardware code into various graph formats such as *Abstract Syntax Tree* (AST) or *Data-Flow Graph* (DFG). AST captures the syntactic structure of hardware code, while DFG indicates the relationships and dependencies between the signals and gives a higher-level expression of the code's computational structure.

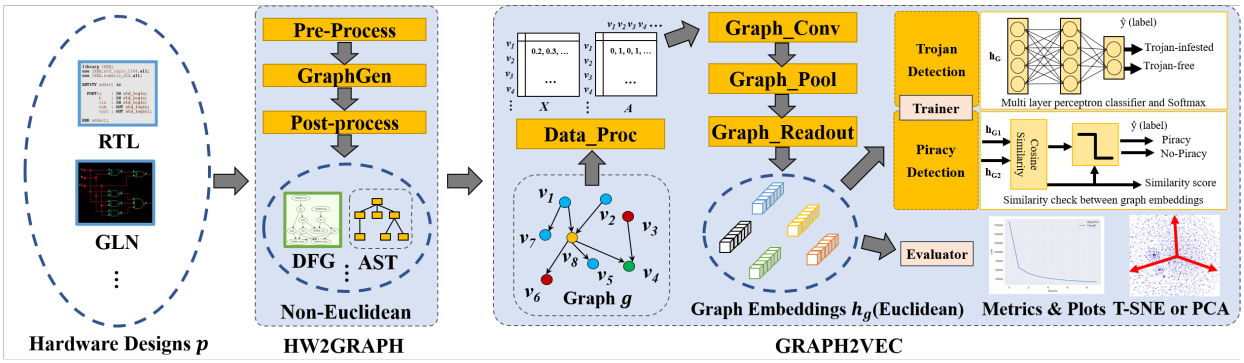

Figure 2.3: The architecture of *HW2VEC*. Beginning with hardware design objects (RTL or GLN), the HW2GRAPH leverages PRE_PROC, GRAPH_GEN, and POST_PROC to extract graph representations from hardware designs in the form of node embedding matrix ($\mathbf{X}$) and adjacency matrix ($\mathbf{A}$). These graphs are then passed to GRAPH2VEC to acquire the graph embeddings for graph learning tasks of hardware security.

HW2GRAPH consists of three primary modules: *pre-processing*, *graph generation engine*, and *post-processing*.

Pre-processing (PRE_PROC): in this module, we have several automatic scripts for pre-processing a raw hardware code p. As a hardware design can contain several modules stored in separate files, the first step is to combine them into a single file (i.e., flattening). Next, to automatically locate the "entry point" top module of p, the script scans the flattened code for the keyword "module" and extracts the module names and the number of repetitions in p. Then, the script analyzes the list of discovered module names and takes the one that appears only once, which means the module is not instantiated by any other module, as the top module. Here, we denote the pre-processed hardware design code as p'.

Graph Generation Engine (GRAPH_GEN): we integrate PyVerilog [158], a hardware design toolkit for parsing the Verilog code, into this module. The pre-processed code p' is first converted by a lexical analyzer, YACC (Yet Another Compiler-Compiler), into a corresponding parse tree. Then, we recursively iterate through each node in the parse tree with Depth-First Search (DFS). At each recursive step, we determine whether to construct a collection of name/value pairs, an ordered list of values, or a single name/value pair based on

the token names used in Verilog AST. To acquire DFG, the AST is further processed by the data flow analyzer to create a signal DFG for each signal in the circuit such that the signal is the root node. Lastly, we merge all the signal DFGs. The resulting graph, either DFG or AST, is denoted as $g = (V, E)$. The AST is a tree type of graph in which the nodes V can be operators (mathematical, gates, loop, conditional, etc.), signals, or attributes of signals. The edges E indicate the relation between nodes. The DFG shows data dependency where each node in V represents signals, constant values, and operations such as xor, and, concatenation, branch, or branch condition, etc. Each edge in E stands for the data dependency relation between two nodes. Specifically, for all v_i, v_j pairs, the edge e_{ij} belongs to E ($e_{ij} \in E$) if v_i depends on v_j, or if v_j is applied on v_i.

Post-processsing (POST_PROC) The output from *Graph Generation Engine* is in JSON (JavaScript Object Notation) format. In this phase, we convert a JSON-formatted graph into a NetworkX graph object. NetworkX is an efficient, scalable, and highly portable framework for graph analysis. Several popular geometric representation learning libraries (PyTorch-Geometric and Deep Graph Library) take this format of graphs as the primary data structure in their pipelines.

2.3.2 GRAPH2VEC: from graph to graph embedding

Once HW2GRAPH has converted a hardware design into a graph g, we begin to process g with the modules in GRAPH2VEC, including *Dataset Processor*, *Trainer*, and *Evaluator* to acquire the graph embedding h_g.

Dataset Processor: this module handles the low-level parsing tasks such as caching the data on disk to optimize the tasks that involve repetitive model testing, performing train-test split, and finding the unique set of node labels among all the graph data instances. One important task of the *dataset processor* is to convert a graph $g = (V, E)$ into the tensor-like

inputs $\mathbf{X}$ and $\mathbf{A}$ where $\mathbf{X}$ represents the node embeddings in matrix form and $\mathbf{A}$ stands for the adjacency information of g. The conversion between E and $\mathbf{A}$ is straightforward. To acquire $\mathbf{X}$, *Dataset Processor* performs a *normalization* process and assigns each of the nodes a label that indicates its type, which may vary for different kinds of graphs (AST or DFG). Each node gets converted to an initial vectorized representation using one-hot encoding based on its type label.

Graph Embedding Model: in this module, we break down the graph learning pipeline into multiple network components, including graph convolution layers (*GRAPH_CONV*), graph pooling layers (*GRAPH_POOL*), and graph readout operations (*GRAPH_READOUT*). In *HW2VEC*, the *GRAPH_CONV* is inspired by the Spatial Graph Convolution Neural Network (SGCN), which defines the convolution operation based on a node's spatial relations. In literature, this phase is also referred to as *message propagation phase*, which involves two sub-functions: **AGGREGATE** and **COMBINE** functions. Each input graph $g = (V, E)$ is initialized in the form of node embeddings and adjacency information ($\mathbf{X}^{(0)}$ and $\mathbf{A}$). For each k-th iteration, the process updates the node embeddings $\mathbf{X}^{(k)}$ using each node representation $h_v^{(k-1)}$ in $\mathbf{X}^{(k-1)}$, given by,

$$a_v^{(k)} = \mathbf{AGGREGATE}^{(k)}(\{h_u^{(k-1)} : u \in N(v)\}),$$
$$h_v^{(k)} = \mathbf{COMBINE}^{(k)}(h_v^{(k-1)}, a_v^{(k)})$$

where $h_v^{(k)} \in R^{C^k}$ denotes the feature vector after k iterations for the v-th node and $N(v)$ returns the neighboring nodes of v-th node. Essentially, the **AGGREGATE** collects the features of the neighboring nodes to extract an aggregated feature vector $a_v^{(k)}$ for the layer k. The **COMBINE** combines the previous node feature $h_v^{(k-1)}$ with $a_v^{(k)}$ to output next feature vector $h_v^{(k)}$. This message propagation is carried out for a pre-determined number of layers k. We denote the final propagation node embedding $\mathbf{X}^{(k)}$ as $\mathbf{X}^{prop}$. Next, in *GRAPH_POOL*, the node embedding $\mathbf{X}^{prop}$ is further processed with an attention-based graph pooling layer.

As indicated from [108, 190], the integration of a graph pooling layer allows the model to operate on the hierarchical representations of a graph. It hence can better perform the graph classification task. Besides, such an attention-based pooling layer allows the model to focus on a local part of the graph and is considered part of a unified computational block of a GNN pipeline [99]. In this layer, we perform *top-k filtering* on nodes according to the scoring results, as follows:

$$\alpha = \mathrm{SCORE}(\mathbf{X}^{prop}, \mathbf{A}),$$
$$\mathbf{P} = \mathrm{top}_k(\alpha)$$

where α stands for the coefficients predicted by the graph pooling layer for nodes. $\mathbf{P}$ represents the indices of the pooled nodes, which are selected from the top k of the nodes ranked according to α. The number k used in *top-k filtering* is calculated by a pre-defined pooling ratio, pr using $k = pr \times |V|$, where we consider only a constant fraction pr of the embeddings of the nodes of the DFG to be relevant (i.e., 0.5). One example of the scoring function is to utilize a separate trainable GNN layer to produce the scores so that the scoring method considers both node features and topological characteristics [108]. We denote the node embeddings and edge adjacency information after pooling by $\mathbf{X}^{pool}$ and $\mathbf{A}^{pool}$, which are calculated as follows:

$$\mathbf{X}^{pool} = (\mathbf{X}^{prop} \odot \tanh(\alpha))_{\mathbf{P}},$$
$$\mathbf{A}^{pool} = \mathbf{A}^{prop}{}_{(\mathbf{P},\mathbf{P})}$$

where $\odot$ represents an element-wise multiplication, $()_{\mathbf{P}}$ refers to the operation that extracts a subset of nodes based on P, and $()_{(\mathbf{P},\mathbf{P})}$ refers to the information of the adjacency matrix between the nodes in this subset. Lastly, in *GRAPH_READOUT*, the overall graph-level feature extraction is carried out by either summing up or averaging up the node features

$\mathbf{X}^{pool}$. We denote the graph embedding for each graph g as $h_g^{(k)}$, computed as follows:

$$h_g^{(k)} = GRAPH\_READOUT(\{h_v^{(k)} : v \in V\}) \tag{2.1}$$

We use the graph embedding $h_g^{(k)}$ to model the behavior of circuits (use h_g for simplicity). After this, the fixed-length embeddings of hardware designs become compatible with ML algorithms. In practice, these network components can be combined in various ways depending on the type of the tasks (node-level task, graph-level task) or the complexity of the tasks (simple or complex network architecture). In GRAPH2VEC, one default option is to use one or multiple *GRAPH_CONV*, followed by a *GRAPH_POOL* and a *GRAPH_READOUT*. Besides, in conjunction with Multi-Layer Perceptron (MLP) or other ML layers, this architecture can transform the graph data into a form that we can use in calculating the loss for learning. In GRAPH2VEC, we reserve the flexibility for customization, so users may also choose to combine these components in a way that is effective for their tasks.

Trainer and Evaluator: the *Trainer* module takes training datasets, validating datasets, and a set of hyperparameter configurations to train a GNN model. *HW2VEC* currently supports two types of *Trainer*, *graph-trainer* and *graph-pair-trainer*. To be more specific, *graph-trainer* uses GRAPH2VEC's model to perform graph classification learning and evaluation, while *graph-pair-trainer* considers pairs of graphs, calculates their similarities, and ultimately performs the graph similarity learning and evaluation. Some low-level tasks are also handled by *Trainer* module, such as caching the best model weights evaluated from the validation set to the disk space or performing mini-step testing. Once the training is finished, the *Evaluator* module plots the training loss and commonly used metrics in ML-based hardware security applications. To facilitate the analysis of the results, *HW2VEC* also provides utilities to visualize the embeddings of hardware designs with t-SNE-based dimensionality reduction [165]. Besides, *HW2VEC* provides multiple exporting functionalities so that the learned embeddings can be presented in standardized formats, and users can also choose

other third-party tools such as *Embedding Projector* [149] to analyze the embeddings.

2.4 HW2VEC Use-cases

In this section, we describe *HW2VEC* use-cases. First, Section 2.4.1 exhibits a fundamental use-case in which a hardware design p is converted into a graph g and then into a fixed-length embedding h_g. Next, the use-cases of *HW2VEC* for two hardware security applications (detecting hardware Trojan and hardware IP piracy) are described in Section 2.4.2 and Section 2.4.3, respectively.

2.4.1 Use-case 1: Converting a Hardware Design to a Graph Embedding

The first use-case demonstrates the transformation of a hardware design p into a graph g and then into an embedding h_g. As Algorithm 1 shows, HW2GRAPH uses *preprocessing* (*PRE_PROC*), *graph generation* (*GRAPH_GEN*) and *post-processing* (*POST_PROC*) modules to convert each hardware design into the corresponding graph. The g is fed to GRAPH2VEC with the uses of *Data Processing* (*DATA_PROC*) to generate X and A. Then, X and A are processed to generate the graph embedding h_g. This resulting h_g can be further inspected with the utilities of *Evaluator*. In *HW2VEC*, we provide Algorithm 1's implementation in `use_case_1.py` of our repository.

2.4.2 Use-case 2: Hardware Trojan Detection (GNN4TJ)

In this use-case, we demonstrate how to use *HW2VEC* to detect HTs, which has been a major hardware security challenge for decades. An HT is an intentional, malicious circuit

Algorithm 1: Use-case - *HW2VEC*

Input: A hardware design program p.
Output: A graph embedding h_p for p.
def *HW2GRAPH*(p)**:**
    $p' \leftarrow$ PRE_PROC(p);
    $g \leftarrow$ GRAPH_GEN(p');
    $g\prime \leftarrow$ POST_PROC(g);
    return $g\prime$;
def *GRAPH2VEC*(g)**:**
    $X, A \leftarrow$ DATA_PROC(g)
    $X^{prop}, A^{prop} \leftarrow GRAPH\_CONV(X, A)$
    $X^{pool}, A^{pool} \leftarrow GRAPH\_POOL(X^{prop}, A^{prop})$
    $h_g \leftarrow GRAPH\_READOUT(X^{pool})$
    return h_g
$g \leftarrow$ *HW2GRAPH*(p);
$h_g \leftarrow$ *GRAPH2VEC*(g);

modification by an attacker [140]. The capability of detection at an early stage (particularly at the RTL level) is crucial, as removing HTs at later stages could be very expensive. The majority of existing solutions rely on a golden HT-free reference or cannot generalize detection to previously unseen HTs. In our prior work ([188]), we propose a GNN-based approach to model the circuit's behavior and identify the presence of HTs.

To realize [188] using *HW2VEC*, we first use HW2GRAPH to convert each hardware design p into a graph g. Then, we transform each g to a graph embedding h_g. Lastly, h_g is used to make a prediction $\hat{y}$ with an MLP layer. To train the model, the cross-entropy loss L is calculated collectively for all the graphs in the training set (see Equation 2.2).

$$L = H(Y, \hat{Y}) = \sum_i y_i * log_e(\hat{y}_i), \tag{2.2}$$

where H is the loss function. Y stands for the set of ground-truth labels (either TROJAN or NON_TROJAN) and $\hat{Y}$ represents the corresponding set of predictions. Once trained by minimizing L, we use the model and Algorithm 2 to perform HT detection (can also be done with a pre-trained model). In practice, we provide an implementation in `use_case_2.py` in

our repository.

Algorithm 2: Use-case - Hardware Trojan Detection

Input: A hardware design program p.
Output: A label indicating whether p contains Hardware Trojan.
def *use_case_2*(p)**:**
    $g \leftarrow$ HW2GRAPH(p);
    $h_g \leftarrow$ GRAPH2VEC(g);
    $\hat{y} \leftarrow$ MLP(h_g);
    if $\hat{y}[0] > \hat{y}[1]$ **then**
        return TROJAN;
    else
        return NON_TROJAN;
$\hat{Y} \leftarrow$ *use_case_2*(p);

2.4.3 Use-case 3: Hardware IP Piracy Detection (GNN4IP)

This use-case demonstrates how to leverage *HW2VEC* to confront another major hardware security challenge – determining whether one of the two hardware designs is stolen from the other or not. The globalization of the IC supply chain poses a high risk of theft for design companies that share their most valuable assets, IPs, with other entities. IP piracy is a serious issue in the current economy, with a drastic need for an effective detection method. According to the U.S. Department of Commerce study, 38% of the American economy is composed of IP-intensive industries [172] that lose between $225 billion to $600 billion annually because of Chinese companies stealing American IPs mainly in the semiconductor industry, based on the U.S. Trade Representative report [152].

To implement [187], the GNN model has to be trained with a graph-pair classification trainer in GRAPH2VEC. The first step is to use HW2GRAPH to convert a pair of circuit designs p_1, p_2 into a pair of graphs g_1, g_2. Then, GRAPH2VEC transforms both g_1 and g_2 into graph embeddings h_{g_1}, h_{g_2}. To train this GNN model for assessing the similarity of h_{g_1} and h_{g_2}, the cosine similarity is computed as the final prediction of piracy, denoted as $\hat{y} \in [-1, 1]$.

Algorithm 3: Use-case - Hardware IP Piracy Detection

Input: A pair of hardware design programs p_1, p_2.
Output: A label indicating whether p_1, p_2 is piracy.
def *use_case_3*(p_1, p_2)**:**
    $g_1, g_2 \leftarrow$ HW2GRAPH(p_1), HW2GRAPH(p_2);
    $h_{g_1}, h_{g_2} \leftarrow$ GRAPH2VEC(g_1), GRAPH2VEC(g_2);
    $\hat{y} \leftarrow$ COSINE_SIM(h_{g_1}, h_{g_2});
    if $\hat{y} > \delta$ **then**
        return PIRACY;
    else
        return NON-PIRACY;
$\hat{Y} \leftarrow$ *use_case_3*(p_1, p_2);

The loss between a prediction $\hat{y}$ and a ground-truth label y is calculated as Equation 2.3 shows. Lastly, the final loss L is computed collectively with a loss function H for all the graphs in the training set (see Equation 2.4).

$$G(y, \hat{y}) = \begin{cases} 1 - \hat{y}, & \texttt{if } y = 1 \\ \text{MAX}(0, \hat{y} - \text{MARGIN}) & \texttt{if } y = -1 \end{cases} \tag{2.3}$$

$$L = H(Y, \hat{Y}) = \sum_i G(y_i, \hat{y_i}), \tag{2.4}$$

where Y stands for the set of ground-truth labels (either PIRACY or NON_PIRACY) and $\hat{Y}$ represents the corresponding set of predictions. The MARGIN is a constant to prevent the learned embedding from becoming distorted (always set to 0.5 in [187]). Once trained, we use this model and Algorithm 3 with δ, which is a decision boundary used for making a final judgment to detect piracy. In practice, we provide the implementation of Algorithm 3 in `use_case_3.py`.

2.5 Experimental Results

In this section, we evaluate the *HW2VEC* through various experiments using the use-case implementations described earlier.

2.5.1 Dataset Preparation

For evaluation, we prepare one RTL dataset for HT detection (*TJ-RTL*) and both RTL and GLN datasets (*IP-RTL* and *IP-GLN*) for IP piracy detection.

The TJ-RTL dataset: we construct the *TJ-RTL* dataset by gathering the hardware designs with or without HT from the Trust-Hub.org benchmark [164]. From Trust-Hub, we collect three base circuits, `AES`, `PIC`, and `RS232`, and insert 34 varied types of HTs into them. We also include these HTs as standalone instances to the *TJ-RTL* dataset. Furthermore, we insert these standalone HTs into two other circuits (`DES` and `RC5`) and include the resulting circuits to expand the *TJ-RTL* dataset. Among the five base circuits, `AES`, `DES`, and `RC5` are cryptographic cores that encrypt the input plaintext into the ciphertext based on a secret key. For these circuits, the inserted HTs can leak sensitive information (i.e., secret key) via side channels such as power and RF radiation or degrade the performance of their host circuits by increasing the power consumption and draining the power supply. `RS232` is an implementation of the UART communication channel, while the HT attacks on `RS232` can affect the functionality of either transmitter or receiver or can interrupt/disable the communication between them. The `PIC16F84` is a well-known Power Integrated Circuit (PIC) microcontroller, and the HTs for PIC fiddle with its functionality and manipulate the program counter register. Lastly, we create the graph datasets, *DFG-TJ-RTL* and *AST-TJ-RTL*, in which each graph instance is annotated with a Trojan or Non_Trojan label.

The IP-RTL and IP-GNL datasets: to construct the datasets for evaluating piracy de-

tection, we gather RTL and GLN of hardware designs in Verilog format. The RTL dataset includes common hardware designs such as single-cycle and pipeline implementation of MIPS processors which are derived from available open-source hardware design on the internet or designed by a group of in-house designers who are given the same specification to design hardware in Verilog. The GLN dataset includes ISCAS'85 benchmark [74] which includes 7 different hardware designs (`c432`, `c499`, `c880`, `c1355`, `c1908`, `c6288`, `c7552`) and their obfuscated instances derived from TrustHub. Obfuscation complicates the circuit and confuses reverse engineering but does not change the behavior of the circuit. Our collection comprises 50 distinct circuit designs and several hardware instances for each circuit design that sums up 143 GLN and 390 RTL codes. We form a graph-pair dataset of 19,094 similar pairs and 66,631 different pairs, dedicating 20% of these 85,725 pairs for testing and the rest for training. This dataset comprises pairs of hardware designs labeled as PIRACY (positive) or NO-PIRACY (negative).

2.5.2 The Evaluation of Hardware Trojan Detection (GNN4TJ)

Here, we evaluate the capability of *HW2VEC* in identifying the existence of HTs from hardware designs. We leverage the implementation mentioned in Section 2.4.2. During the experiments, we used 2 GCN layers with 200 hidden units for each layer. For the graph pooling layer, we use the pooling ratio 0.8 to perform *top-k filtering*. For **READOUT**, we use *max-pooling* for aggregating node embeddings of each graph. GNN4TJ uses 1 MLP layer that reduces the number of hidden units from 200 to 2 used in $\mathbf{h}_G$ for predicting the result of HT detection. In training, we append a dropout layer with a rate of 0.5 after each GCN layer. We train the model for 200 epochs using the batch gradient descent algorithm with batch size 4 and a learning rate 0.001. In *HW2VEC*, we directly use this hyperparameter setting which is stored as a preset in a YAML configuration file. For performance metrics, we count the True Positive (TP), False Negative (FN) and False Positive (FP) for deriving

Precision $P = TP/(TP+FP)$ and Recall $R = TP/(TP+FN)$. R manifests the percentage of HT-infested samples that the model can identify. As the number of HT-free samples incorrectly classified as HT is also critical, we compute P that indicates what percentage of the samples that the model classifies as HT-infested actually contains HT. F_1 score is the weighted average of precision and recall that better presents performance, calculated as $F_1 = 2 \times P \times R/(P+R)$.

To demonstrate whether the learned model can generalize the knowledge to handle the unknown or unseen circuits, we perform a variant *leave-one-out* cross-validation to experiment. We perform a train-test split on the *TJ-RTL* dataset by leaving one base circuit benchmark in the testing set and use the remaining circuits to train the model. We repeat this process for each base circuit and average the metrics we acquire from evaluating each testing set. The result tested with *HW2VEC* tool is presented in Table 2.1, indicating that *HW2VEC* can reproduce comparable results to [188] in terms of F_1 score (0.926 versus 0.940) if we use DFG as the graph representation. The difference in performance can be due to the use of different datasets. When using AST as the graph representation for detecting HT, *HW2VEC* performs worse in terms of F_1 score, indicating that DFG is a better graph representation because it captures the data flow information instead of simply the syntactic information of a hardware design code. All in all, these results demonstrate that our *HW2VEC* can be leveraged for studying HT detection at design phases.

Method	Graph	Dataset	Precision	Recall	F1
HW2VEC	**DFG**	RTL	0.87334	0.98572	0.92596
HW2VEC	**AST**	RTL	0.90288	0.8	0.8453
[188]	**DFG**	RTL	**0.923**	**0.966**	**0.940**

Table 2.1: The performance of HT detection using *HW2VEC*.

2.5.3 The Evaluation of Hardware IP Piracy Detection (GNN4TP)

Besides HT detection, we also evaluate the power of *HW2VEC* in detecting IP piracy. We leverage the usage example mentioned in Section 2.4.3, which examines the cosine-similarity score $\hat{y}$ for each hardware design pair and produces the final prediction with the decision boundary. Using the *IP-RTL* dataset and the *IP-GNL* dataset (mentioned in Section 2.5.1), we generate graph-pair datasets by annotating the hardware designs that belong to the same hardware category as SIMILAR and the ones that belong to different categories as DISSIMILAR. We perform a train-test split on the dataset so that 80% of the pairs will be used to train the model. We compute the accuracy of detecting hardware IP piracy, which expresses the correctly predicted sample ratio and calculates the F_1 score as the evaluating metric. We refer to [187] for the selection of hyperparameters (stored in a YAML file). Specifically, we use 2 GCN layers with 16 hidden units for each layer. For the *graph_pool*, we use the pooling ratio of 0.5 to perform top-k filtering. For the *graph_readout*, we use *max-pooling* for aggregating node embeddings of each graph. In training, we apply dropout with a rate of 0.1 after each GCN layer. We train the model using the batch gradient descent algorithm with batch size 64 and a learning rate to be 0.001.

Table 2.2 is the result that indicates that *HW2VEC* can reproduce comparable results to [187] in terms of piracy detection accuracy. When using DFG as the graph representation, *HW2VEC* underperforms [187] by 3% at RTL level and outperforms [187] by 4.2% at GLN level. Table 2.2 also shows a similar observation with Section 2.5.2 that using AST as the graph representation can lead to worse performance than using DFG. Figure 2.4 visualizes the graph embeddings that *HW2VEC* exports for every processed hardware design, allowing users to inspect the results manually. For example, by inspecting Figure 2.4, we may find a clear separation between `mips_single_cycle` and `AES`. Certainly, *HW2VEC* can perform better with more fine-tuning processes. However, the evaluation aims to demonstrate that *HW2VEC* can help practitioners study the problem of IP piracy at RTL and

GLN levels.

Method	Graph	Dataset	Accuracy	F1
HW2VEC	**DFG**	RTL	0.9438	0.9277
HW2VEC	**DFG**	GLN	0.9882	0.9652
HW2VEC	**AST**	RTL	0.9358	0.9183
[187]	**DFG**	RTL	**0.9721**	–
[187]	**DFG**	GLN	**0.9461**	–

Table 2.2: The results of detecting IP piracy with *HW2VEC*.

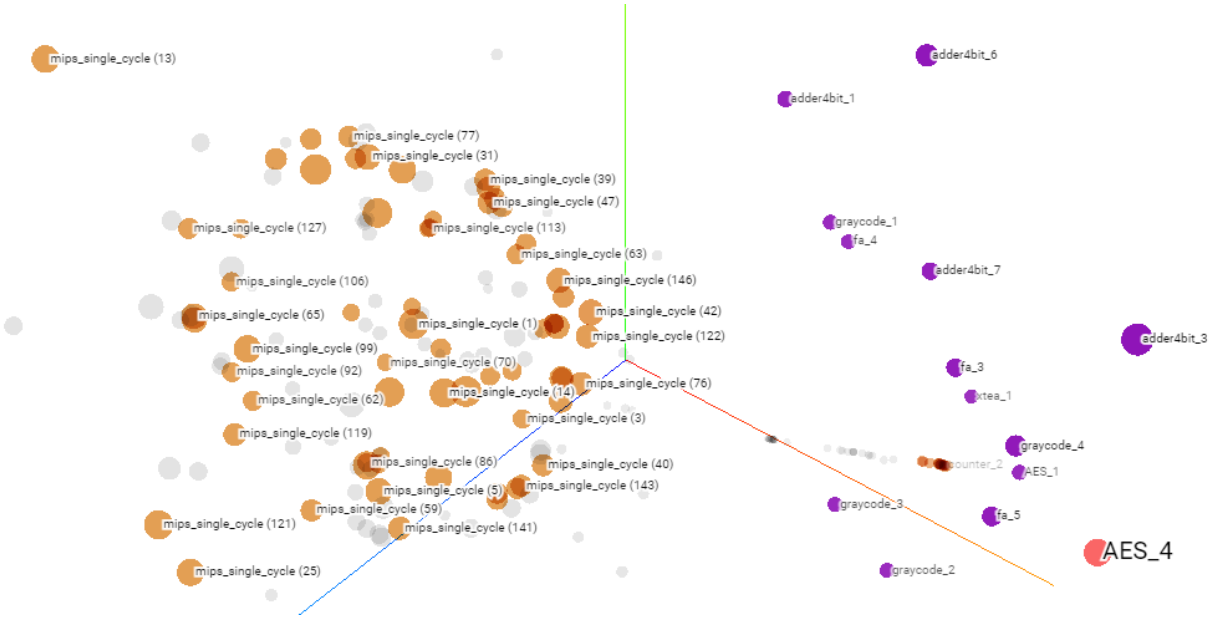

Figure 2.4: The embedding visualization with 3D t-SNE.

2.5.4 The Evaluation of HW2VEC Tool Timing

To evaluate the time required for training and testing, we test the models on a server with NVIDIA TITAN-XP and NVIDIA GeForce GTX 1080 graphics cards. Table 2.3 indicates that the time taken by training and inference are both below 15 milliseconds, and the time taken by training is more than inference as it includes the time for performing back-propagation. As *HW2VEC* aims to serve as a research tool, our users must evaluate their applications within a reasonable time duration. We believe that the time spent by the graph learning pipelines of *HW2VEC* should be acceptable for conducting research. For practically

deploying the models, the actual timing can depend on the computation power of hosting devices and the complexity of the models for the applications. Suppose our users need an optimized performance for real-time applications. In that case, they can implement the models with performance-focused programming languages (C or C++) or ML frameworks (e.g., TensorFlow) using the best model settings found using *HW2VEC*. As for specialized hardware that can accelerate the processing of GNNs, it is still an open challenge as indicated in [1].

Table 2.4 indicates that the time that *HW2VEC* spends in converting the raw hardware code into ASTs is on average 1.98 seconds. Although [73] takes 1.37 seconds on average per hardware code, it requires domain knowledge to find a deterministic way to perform feature extraction. For DFG extraction, HW2VEC takes on average 244.58 seconds per graph as it requires recursive traversals to construct the whole data flow. In our datasets, `AES` and `DES` are relatively more complex, so *HW2VEC* takes 472.46 seconds on average to process them while the rest of the data instances take 16.70 seconds on average. Certainly, *HW2VEC* performs worse in DFG extraction, but manual feature engineering possibly requires a much longer time. In design phases, even for an experienced hardware designer, it can take 6-9 months to prototype a complex hardware design [161] so the time taken by *HW2VEC* is acceptable and not slowing down the design process. However, as the first open-source tool in the field, *HW2VEC* will keep evolving and embrace the contributions from the open-source community.

	TJ-RTL-AST	*IP-RTL-AST*
training time	10.5 (ms)	13.5 (ms)
testing time	6.8 (ms)	12.4 (ms)

Table 2.3: The time profiling for training/inference.

	TJ-DFG-RTL	*IP-DFG-GLN*	*TJ-AST-RTL*
# of node	7573.58	7616.16	971.01
# of edge	8938.11	9495.97	970.01
Exec time	244.58 (s)	14.61 (s)	1.98 (s)

Table 2.4: The graph extraction time profiling. For *TJ-DFG-RTL*, the hardware `AES` and `DES` jointly take 472.46 seconds on average for DFG extraction while the rest of the data instances take 16.7 seconds on average.

2.5.5 The Evaluation of HW2VEC Applicability

In Section 2.5.2 and Section 2.5.3, we have discussed the performance of the GNN-based approach in resolving two hardware security problems: hardware Trojan detection and IP piracy detection. In Section 2.5.2, our evaluation shows that *HW2VEC* can successfully be leveraged to perform HT detection on hardware designs, particularly on the unseen ones, without the assistance of golden HT-free reference. The capability to model hardware behaviors can be attributed to using a natural representation of the hardware design (e.g., DFG) and the use of the GNN-based method for capturing both the structural information and semantic information from the DFG and co-relating this information to the final HT labels. Similarly, Section 2.5.3 indicates that *HW2VEC* can be utilized to assess the similarities between circuits and thus can be a countermeasure for IP piracy. The use of graph representation for a hardware design and a Siamese GNN-based network architecture are the keys in [187] to perform IP piracy detection at both RTL and GLN levels. For other hardware security applications, the flexible modules provided by *HW2VEC* (*Trainer* and *Evaluator*) can be adapted easily to different problem settings. For example, by adjusting the *Trainer* to train the GNN models for node classification, *HW2VEC* can be adapted to localize the HT(s) or hardware bug(s) that exist in the hardware designs. Also, the cached models provided by *HW2VEC* can be used in learning other new hardware design-related tasks through the transfer of knowledge from a related task that has already been learned as the idea of *Transfer Learning* suggests [162].

2.6 Summary and Future Directions

As technological advancements continue to grow, the fights between attackers and defenders will rise in complexity and severity. In this chapter, to contribute to the hardware security research community, we propose *HW2VEC*: a graph learning tool for automating hardware security. *HW2VEC* provides an automated pipeline for hardware security practitioners to extract graph representations from a hardware design in either RTL or GLN. Besides, the toolbox of *HW2VEC* allows users to realize their hardware security applications flexibly. Our evaluation shows that *HW2VEC* can be leveraged and integrated for counteracting two critical hardware security threats: *Hardware Trojan Detection* and *IP Piracy Detection*.

- For HT detection, *HW2VEC* can be turned into a novel golden reference-free methodology to find unknown HT in RTL. We generate the DFGs of RTL codes and employ the GNN to construct a model on the generated graphs. GNN4TJ automatically extracts the features of graphs and learns the behavior of the hardware design. Our model is trained and tested on a DFG dataset created by expanding the Trustub benchmarks. The results indicate that GNN4TJ discovers HT with 97% recall very fast in 21.1ms.

- For IP piracy detection, *HW2VEC* can also be converted into a novel IP piracy detection methodology, called *GNN4IP*, which does not have existing countermeasures shortcomings such as overhead and vulnerability to attacks. Our automated framework extracts the DFGs from RTL codes and gate-level netlist. Then, *HW2VEC*, our graph neural network generates embeddings for graphs according to the similarity between designs. Based on embeddings, we infer IP piracy between circuits with 96% accuracy.

As discussed in the chapter, we certainly anticipate that *HW2VEC* can provide more straightforward access for practitioners and researchers to apply graph learning approaches to hardware security applications. While in the meantime, we expect more researchers can

apply graph learning for many hardware security applications differently from as described earlier, we hereby point out another critical research challenge, that is *Domain Shift* (DS) issue. In practice, the DS issue arises when these ML-based or GL-based approaches fail to handle *Out-Of-Distribution* (OOD) data during testing, as pictured in Figure 2.5. The key reason is the assumption that the training and testing data are drawn from the same data distribution. Therefore, as a critical future step of *HW2VEC* project and the hardware

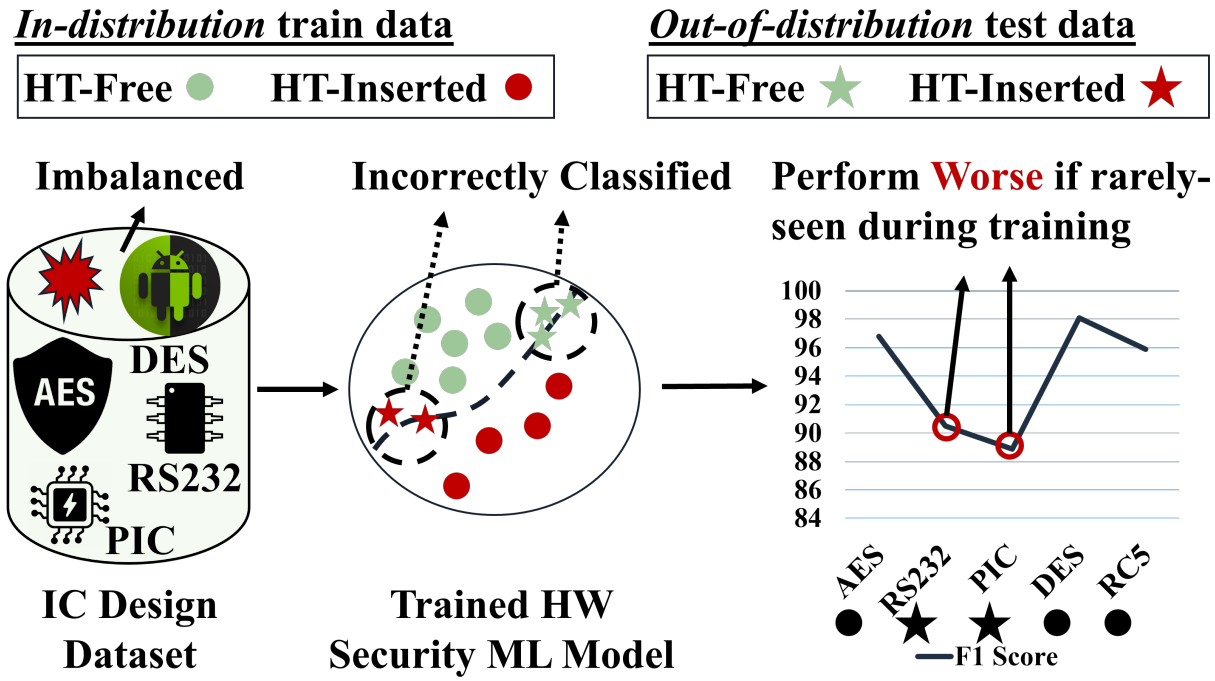

Figure 2.5: The illustration of our motivation where a GL-based approach can fail in generalizing to OOD IC designs.

security field, it is required to revisit the problem of HT detection from a new perspective considering the DS issue, aiming at advancing existing ML-based approaches towards the goal of truly defending against *zero-day* attacks. Generally speaking, to do so, one potential solution is to propose a new domain generalization framework that enhances the existing ML-based solutions regarding robustness and generalization. Besides, a new attack model also needs to be introduced, which is called the OOD HT insertion attack. OOD HT insertion attacks assume the existence of a strong attacker who can craft unseen types of HTs

that are sampled from an OOD of the defender training set. Against such attacks, this new framework with domain generalization capability can make ML-based or GL-based approaches capable of learning domain-invariant features to generalize better against OOD IC designs.

Chapter 3

Graph Learning for Embedded Software Analysis

In this chapter, to improve *Software Reverse Engineering* (SRE) tools, we propose *CFG2VEC*, a Hierarchical *Graph Neural Network* (GNN) based approach. Mission-critical embedded software is critical to our society's infrastructure but can be subject to new security vulnerabilities as technology advances. When security issues arise, *Reverse Engineers* (REs) use SRE tools to analyze vulnerable binaries. However, existing tools have limited support, and REs undergo a time-consuming, costly, and error-prone process that requires experience and expertise to understand software behaviors and vulnerabilities. In the chapter, we introduce a novel representation for analyzing software binaries, called *Graph-of-Graph* (GoG) representation, combining the information of control-flow and function-call graphs. Specifically, our *CFG2VEC* learns how to represent each binary function compiled from various CPU architectures with GoG, utilizing hierarchical GNN and the Siamese network-based supervised learning architecture.

3.1 Introduction

In mission-critical systems, embedded software is vital in manipulating physical processes and executing missions that could pose risks to human operators. Recently, the *Internet of Things* (IoT) has created a market valued at 19 trillion dollars and drastically grown the number of connected devices to approximately 35 billion in 2025 [202, 122, 50]. However, while IoT brings technological growth, it unintendedly exposes mission-critical systems to novel vulnerabilities [44, 143, 19]. The reported number of IoT cyberattacks increased by 300% in 2019 [57], while the discovered software vulnerabilities rose from 1.6k to 100k [48]. The consequence can be detrimental, as indicated in [69], the *Heartbleed* bug [81] can lead to a leakage of up to 64K memory, threatening not only personal but also organizational information security. Besides, *Shellshock* is a bash command-line interface shell bug, but it has existed for 30 years and remains a threat to enterprises today [148, 154]. For mission-critical systems, unexpected disruptions can incur millions of dollars even if they only last for a few hours or minutes [97]. As a result, timely analyzing these impacted software and patching vulnerabilities becomes critical.

However, mission-critical systems usually use software that can last for decades due to the criticality of the missions. Over time, these systems become legacy, and the number of newly-discovered threats can increase (as illustrated in Figure 3.1). Typically, for legacy software, the original development environment, maintenance support, or source code might no longer exist. To address vulnerabilities, vendors offer patches in the form of source code changes based on the current software version (e.g., ver 0.9). However, the only available data in the legacy system is binary based on its source code (e.g., ver 0.1). Such a version gap poses challenges in applying patches to the legacy binaries, leaving the only solution for applying patches to legacy software as direct binary analysis. Today, as Figure 3.2 shows, *Reverse Engineers* (REs) have to leverage *Software Reverse Engineering* (SRE) tools such as *Ghidra* [4], *HexRays* [84], and *radare2* [160] to first disassemble and decompile

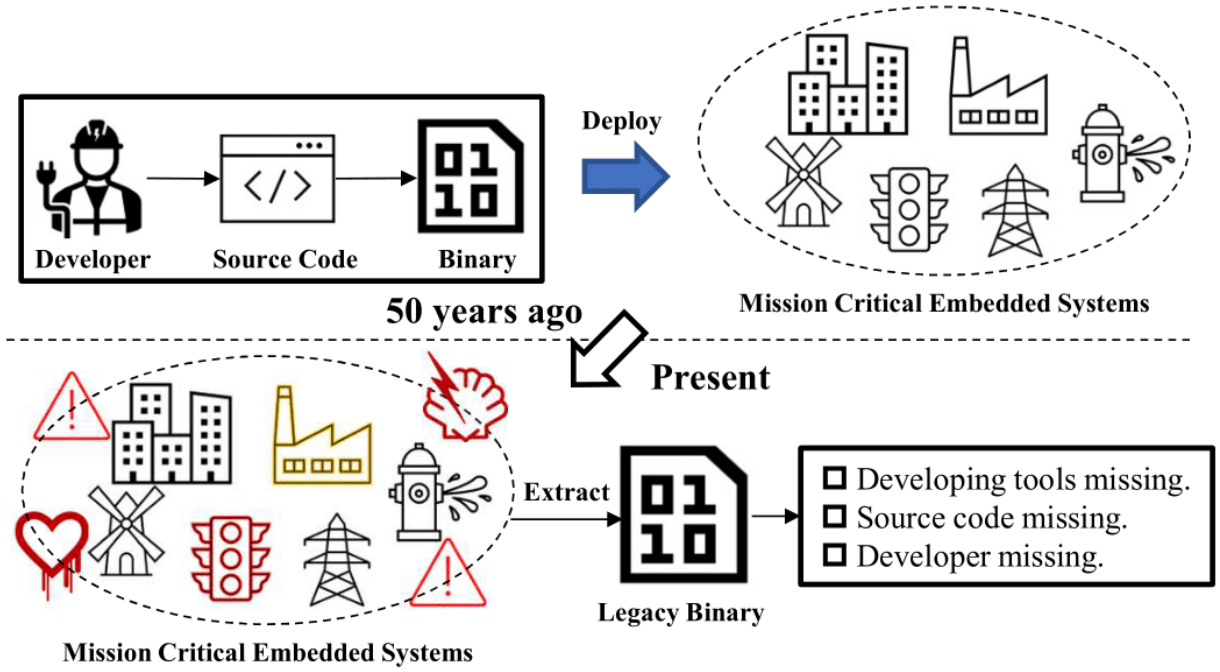

Figure 3.1: Legacy software life cycle.

binaries into higher-level representations (e.g., C or C++). Typically, these tools take the debugging information, strings, and the symbol-table and binary to reconstruct function names and variable names, allowing REs to rebuild a software's structure and functionality without access to source code [96]. For REs, these symbols encode the context of the source code and provide invaluable information that could help them to understand the program's logic as they work to patch vulnerable binaries. However, symbols are often excluded for optimizing the binary's footprint in mission-critical legacy systems with limited memory. Because recovering symbols from *stripped binaries* is not straightforward, most decompilers assign meaningless symbol names to code elements. As for understanding the software semantics, REs have to leverage their experience and expertise to consume the information and then interpret the semantics of each coding element.

Recent works tackle these challenges with *Machine Learning* (ML), aiming to recover the program's information from raw binaries. For example, [79], and [102] associate code features to function names and model the relationships between such code features and the

corresponding source-level information (variable names in [102], variable & function names in [79]). Meanwhile, [51] and [66] use an encoder-decoder network structure to predict function names from stripped binary functions based on instruction sequences and control flows. However, none of them support cross-architectural debug information reconstruction. On the other side, there exist works focusing on the cross-platform in their ML models [61, 180, 75]. These works focus on modeling the binary code similarity, extracting a real-valued vector from each control-flow graph (CFG) with attributed features, and then computing the *Structural Similarity* between the feature vectors of binary functions built from different CPU architectures.

In this paper, as part of a multi-industry-academia joint initiative between Siemens, the Johns Hopkins University Applied Physics Laboratory (JHU/APL), BAE Systems (BAE), and UCI, we propose *CFG2VEC*, which utilizes a hierarchical *Graph Neural Network* (GNN) for reconstructing the name of each binary function, aiming to develop the capacity for quick patching of legacy binaries in mission-critical systems. Our *CFG2VEC* forms a *Graph-of-Graph* (GoG) representation, combining CFG and FCG to model the relationship between binary functions' representation and their semantic names. Besides, *CFG2VEC* can tackle cross-architectural binaries thanks to the design of Siamese-based network architecture, as shown in Figure 3.3. One crucial use case of cross-architectural decompilation is *patching*, where the goal is to identify a known vulnerability or a bug and apply a patch. However, there can be architecture gaps when software with a bug can be compiled into many devices with diverse hardware architectures. For example, it is challenging to patch a stripped binary from an exotic embedded architecture compiled ten years ago that is vulnerable to a known attack such as *Heartbleed* [81]. While the reference patch is available in software, the reference architecture may not be readily available or documented, or the vendor may no longer exist. Under such circumstances, mapping code features across architectures is very helpful. It would allow for identifying similarities in code between a stripped binary that is vulnerable and its reference patch, even if the patch was built for a different type of CPU

architecture. For *CFG2VEC*, our targeted contributions are as follows:

- We propose representing binary functions in *Graph-of-Graph* (GoG) and demonstrate its usefulness in reconstructing function names from stripped binaries.
- We propose a novel methodology, *CFG2VEC*, that uses a hierarchical *Graph Neural Network* (GNN) to model control-flow and function-calling relations in binaries.
- We propose using cross-architectural loss when training, allowing *CFG2VEC* to capture the architecture-agnostic representations of binaries.
- We release *CFG2VEC* in a GitHub repository: `https://github.com/AICPS/mindsight_cfg2vec`.
- We integrate our *CFG2VEC* into an experimental Ghidra plugin, assisting the realistic scenarios of patching DARPA *Assured MicroPatching* (AMP) challenge binaries.

3.2 Related Works

This section introduces software reverse engineering backgrounds, discusses the related works using machine learning to improve reverse engineering, and ultimately covers graph learning for binary analysis.

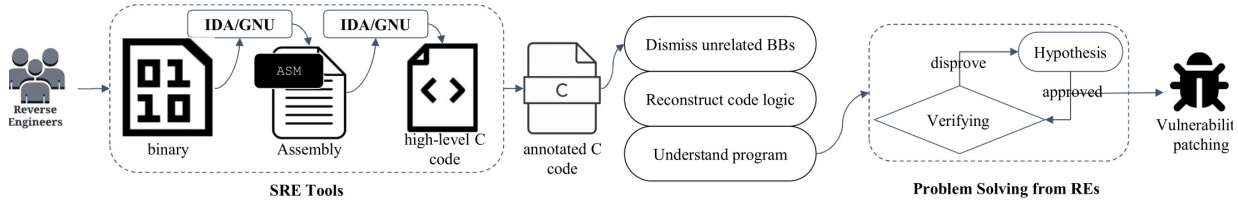

Figure 3.2: The RE flow to solve security issues.

3.2.1 Software Reverse Engineering

Software Reverse Engineering (SRE) aims at understanding the behavior of a program without having access to its source code, often being used in many applications such as detecting malware [182, 55], discovering vulnerabilities, and patching bugs in *legacy software* [166, 28]. One primary tool that *Reverse Engineers* (REs) use to inspect programs is *disassembler* which translates a binary into low-level assembly code. Examples of such tools include *GNU Binutils' objdump* [23], *IDA* [84], Binary Ninja [22], and Hopper [85]. However, even with these tools, reasoning at the assembly level still requires considerable cognitive effort from RE experts.

More recently, REs use *decompilers* such as *Hex-Rays* [83], or *Ghidra* [4] to reverse the compiling process by further translating the output of disassemblers into the code that ensembles high-level programming languages such as C or C++ to reduce the burden of understanding assembly code. From assembly instructions, these decompilers can use program analysis and heuristics to reconstruct variables, types, functions, and control flow structure of a binary. However, the decompilation is incomplete even if these decompilers generate a higher-level output for better code understanding. The reason is that the compilation process discards the source-level information and lowers its abstraction level in exchange for a smaller footprint size, faster execution time, or even security considerations. The source-level information such as comments, variable names, function names, and idiomatic structure can be essential for understanding a program but is typically unavailable in the output of these decompilers.

As Figure 3.2 demonstrated, REs use disassemblers or decompilers to generate high-level source code. Besides, [120] indicates REs will take notes and grant a name to those critical functions related to the vulnerabilities. This will create an annotated source code based on the high-level machine-generated source code. While annotating the source code, REs also analyze the significant part related to the vulnerability and ignore those general instructions

or unrelated codes. At the same time, understanding the logic flow among functions is another major task they must focus on resolving their tasks. After classification, annotation, and understanding, REs experiment with several viable remedies to find the correct patch to fix the vulnerability.

3.2.2 Machine Learning for Reverse Engineering

Software binary analysis is a straightforward first step to enhance security as developers usually deploy software in binaries [147]. Usually, experts conduct the patching process or vulnerability analysis by understanding the compilation source, function signatures, and variable information. However, after the compilation, such information is usually stripped or confuscated deliberately (e.g., *obfuscation*). Software binary analysis becomes more challenging in this case as developers have to recover the source-level information based on their experience and expertise. The early recovery work for binaries focuses on manual completion but suffers from low efficiency, high cost, and the error-prone nature of reverse engineering.

As *Machine Learning* (ML) has significantly advanced in its reasoning capability, applying ML and reconstructing higher-level source code information as an alternative to manual-based approaches has attracted considerable research attention. For example, [46] was the first approach that used neural network-based and graph-based models, predicting the function types to assist the reverse engineer in understanding the binary. [7] also predicted function names with neural networks, aggregating the related features of sections of binary vectors. Then, it analyzes the connections between each function in the source code (e.g., Java) and their corresponding function names for function name prediction. [79], on the other hand, did not use a neural network. It combined a decision-tree-based classification algorithm and a structured prediction with a probabilistic graphical model, then matched the function name by analyzing symbol names, types, and locations. However, [79] can only

predict from a predetermined closed set, incapable of generalizing to new names.

As the languages for naming functions are similar to natural language, recent research works start leaning toward the use of *Natural Language Processing* (NLP)[9, 51, 66]. Precisely, these models predict semantic tokens based on the function names in the library, comprising the function name during inference. The underlying premise is that each token corresponds in some way to the attributes and functionality of the function. [51] uses *Control-Flow Graph* (CFG) to predict function names. It combined static analysis with LSTM and transformer neural model to realize the name of functions. However, the dataset with unbalanced data and insufficient features was limited and hindered utter performance. [9] was designed to solve the limitation of the dataset. It provided *UbuntuDataset* that contained more than 9 million functions in 22K software. [66] demonstrated the framework's effectiveness by building a large dataset. It considers the fine-grained sequence and structure information of assembly code when modeling and realizing function name prediction. Meanwhile, [66] reduced the diversity of data (instructions or words) while keeping the basic semantics unchanged, similar to word stemming and semantics in NLP. However, these works have low precision scores for prediction tasks, exampled by [66], only achieving around 41% in correctly predicting the function name subtokens. Moreover, the metrics for the inference of unknown functions are substantially lower [66], making it difficult for REs to find it helpful in practice.

Although many existing works can reconstruct source-level information, none of them supports reconstructing cross-platform debug information. Cross-compilation is becoming more popular in the development of software. Hardware manufacturers, for instance, often reuse the same firmware code base across several devices running on various architectures [136]. A tool that performs cross-architecture function name prediction/matching would be beneficial if we have a stripped binary compiled for one architecture and a binary of a comparable program compiled for another architecture with debug symbols. We may use the binary with the

debug symbols to predict the names of functions in the stripped binary, which significantly aids debugging. A tool that could capture the architecture-agnostic characteristics of binaries would also help in malware detection as the source code of malware can be compiled in different architectures [136, 167]. Comparing two binaries of different architectures becomes more complicated because they will have different instruction sets, calling conventions, register sets, etc. Furthermore, assembly instructions from different architectures cannot often be compared directly due to the slightly different behavior of different architectures [6]. Cross-architecture function name prediction will assist in finding a malicious function in a program compiled for different architectures by learning its features from a binary compiled for just one architecture. The tools mentioned above are not architecture-agnostic; thus, we cannot utilize them for such applications. To address the flaws mentioned above, aid in creating more efficient decompilers, and make reverse engineering more accessible, we propose *CFG2VEC*. Incorporating the cross-architectural siamese network architecture, our *CFG2VEC* can learn to extract robust features encompassing platform-independent features, enhancing the state-of-the-art by achieving function name reconstruction across cross-architectural binaries.

3.2.3 Graph Learning for Binary Analysis

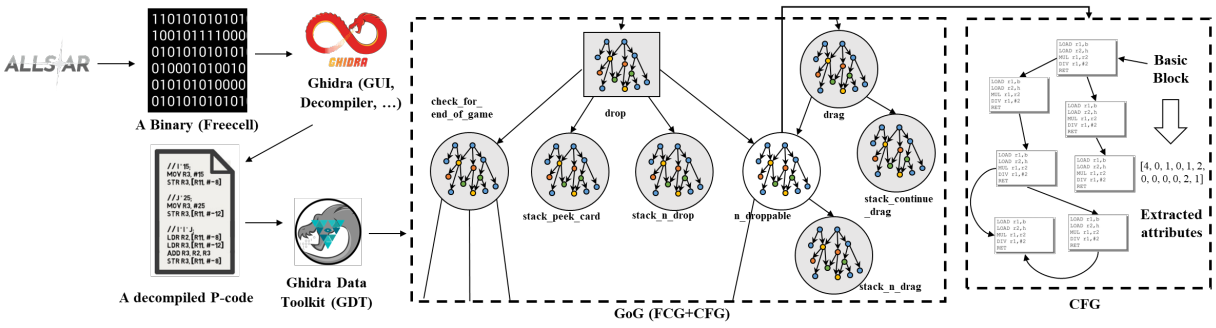

Figure 3.3: An example of a *Graph-of-Graph* (GoG) of a binary compiled from a package Freecell with `amd64` CPU architecture.

Graph learning has become a practical approach across fields [71, 175, 186, 193]. Although conventional ML can effectively capture the features hidden in Euclidean data, such as

images, text, or videos, our work focuses more on the application where the core data is graph-structured. Graphs can be irregular, and a graph may contain a variable size of unordered nodes; moreover, nodes can have a varying number of neighboring nodes, making deep learning mathematical operations (e.g., 2D Convolution) challenging to apply. The operations in conventional ML methods can only be applied by projecting non-Euclidean data into low-dimensional embedding space. In graph learning, *Graph Embeddings* (GE) can transform a graph into a vector (embedding of a graph) or a set of vectors (embedding of nodes or edges) while preserving the relevant and structural information about the graph [71]. *Graph Neural Network* (GNN) is a model aiming at addressing graph-related tasks in an end-to-end manner, where the main idea is to generate a node's representation by aggregating its representation and the representations of its neighbors [175]. GNN stacks multiple graph convolution layers, graph pooling layers, and a graph readout to generate a low-dimensional graph embedding from high-dimensional graph-structured data.

In software binary analysis, many approaches use *Control-Flow Graphs* (CFGs) as the primary representations. For example, *Genius* forms an *Attributed Control-Flow Graph* (ACFG) representation for each binary function by extracting the raw attributes from each *Basic Block* (BB), a straight-line code sequence with no branching in or out except at the entry and exit, in an ACFG [61]. *Genius* measures the similarity of a pair of ACFGs through a bipartite graph matching algorithm, and the ACFGs are then clustered based on similarity. *Genius* leverages a codebook for retrieving the embedding of an ACFG based on similarity. Another approach, *Gemini*, proposes a deep neural network-based model along with a Siamese architecture for modeling binary similarities with greater efficiency and accuracy than other state-of-the-art models of the time [180]. *Gemini* takes in a pair of ACFGs extracted from raw binary functions generated from known vulnerability in code and then embeds them with a shared *Structure2vec* model in their network architecture. Once embedded, *Gemini* trains its model with a loss function that calculates the cosine similarities between two embedded representations. *Gemini* outperforms models like *Genius* or other

approaches such as bipartite graph matching. In literature, there exist other works that consider the *Function Call Graph* (FCG) as their primary data structures in binary analysis for malware detection [78]. Our *CFG2VEC* extracts relevant platform-independent features by combining the usage of CFG and FCG, resulting in a *Graph-of-Graph* (GoG) representation for cross-architectural high-level information reconstruction tasks (e.g., function name).

3.3 CFG2VEC Architecture

This section begins with problem formulation. Next, as Figure 3.4 shows, we depict how our *CFG2VEC* extracts the *Graph-of-Graph* (GoG) representation from each software binary. Lastly, we describe the network architecture in *CFG2VEC*.

3.3.1 Problem Formulation

In our work, given a binary code, denoted as p, compiled from different CPU architectures, we extract a graph-of-graph (GoG) representation, $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ where $\mathcal{V}$ is the set of nodes and $\mathcal{A}$ is the adjacency matrix (As Figure 3.3 shows). The nodes in $\mathcal{V}$ represent functions and the edges in $\mathcal{A}$ indicate their cross-referencing relationships. That says, each of the node $f_i \in \mathcal{V}$ is a CFG, and we denote it as $f_i = (B, A, \phi)$ where the nodes in B represent the basic blocks and the edges in A denote their dependency relationships. ϕ is a mapping function that maps each basic block in the assembly form to its corresponding extracted attributes $\phi(v_i) = C^k$ where C is a numeric value, and k is the number of attributes for the basic block (BB). Whereas the CFG structure is meant to provide more information at the lower BB level, the GoG structure is intended for recovering information at the overarching function level between the CFGs. Figure 3.3 is an example of a partial GoG structure with a closer inspection of one of its CFG nodes and another of a single CFG BB node, showing the set

of features corresponding to that BB node. The goal is to design an efficient and effective graph embedding technique that can be used for reconstructing the function names for each function $f_i \in \mathcal{V}$.

3.3.2 Ghidra Data ToolKit for Graph Extraction

To extract the structured representation required for *CFG2VEC* we leverage the state-of-the-art decompiler *Ghidra* [4] and the *Ghidra Headless Analyzer* [1]. The *headless analyzer* is a command-line version of *Ghidra* allowing users to perform many tasks (such as analyzing a binary file) supported by *Ghidra* via a command-line interface. For extracting GoG from a binary, we developed our *Ghidra Data Toolkit* (GDT); GDT is a set of Java-based metadata extraction scripts used for instrumenting *Ghidra Headless Analyzer*. First, GDT programmatically analyzes the given executable file and stores the extracted information in the internal Ghidra database. Ghidra provides a set of APIs to access the database and retrieve information about the analyzed binary. GDT uses these APIs to export information such as Ghirda's PCode and call graph for each function. Specifically, the *FunctionManager* API allows us to manipulate the information of each decompiled function in the binary and acquire the cross-calling dependencies between functions. For each function, we utilized another Ghidra API *DecompInterface*[2] to extract 12 attributes associated with each basic block in a function. These attributes precisely correspond to the total number of instructions, including arithmetic, logic, transfer, call, data transfer, SSA, compare, and pointer instructions, as well as other instructions not falling within those categories and the total number of constants and strings within that BB. Lastly, by integrating all of the information, we form a GoG representation $\mathcal{G}$ for each binary p. We repeat this process until all binaries are converted to the GoG structure. We feed the resulting GoG representations to our model

[1] *Ghidra Headless Analyzer*: `https://ghidra.re/ghidra_docs/analyzeHeadlessREADME.html`

[2] Documentation of *Ghidra API DecompInterface*: `https://ghidra.re/ghidra_docs/api/ghidra/app/decompiler/DecompInterface.html`

in batches, with the batch size denoted as B.

3.3.3 Hierarchical Graph Neural Network

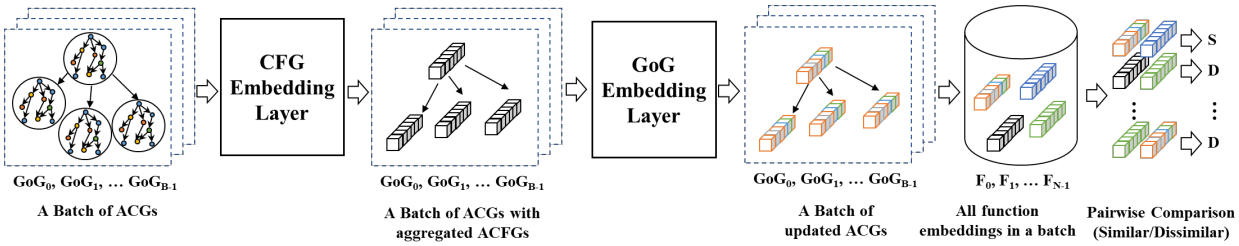

Figure 3.4: The architecture of *CFG2VEC* with a supervised hierarchical graph neural network approach.

Once $\mathcal{G}$ is extracted from the GDT, we then feed it to our hierarchical network architecture (inspired from [76]) that contains both *CFG Graph Embedding* layer and *GoG Graph Embedding Layer* as Figure 3.4 shows. For each GoG structure, we denote it as $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ where $\mathcal{V}$ is a set of functions associated with $\mathcal{G}$ and $\mathcal{A}$ indicates the calling relationships between the functions in $\mathcal{V}$. Each function in $\mathcal{V}$ is in the form of CFG $f_i = (B, A, \phi)$ where each node $b \in B$ is a BB represented in a fixed-length attributed vector $b \in R^d$, and d is the dimension that we have mentioned earlier. A encodes the pair-wise control-flow dependency relationships between these BBs.

CFG Graph Embedding Layer: our network architecture first feeds all functions in a batch of GoGs to the *CFG Graph Embedding Layer* consisting of multiple graph convolutional layers and a graph readout operations. The input to this layer is a function $f_i = (B, A, \phi)$ and the output is the fixed-dimensional vector representing a function. For each BB b_k we let $b_k^0 = b_k$, and we update b_k^t to b_k^{t+1} with the graph convolution operation shown as follows:

$$b_k^{t+1} = f_G(W b_k^t + \sum_{b_m \in A_k} M b_m^t)$$

where f_G is a non-linear activation function such as ReLU, A_k is the list of adjacent BBs for

b_k, and $W \in R^{d \times d}$ and $M \in R^{d \times d}$ are the weights to be learned during the training. We run T iterations of such a convolution, which can be a tunable hyperparameter in our model. During the updates, each BB gradually aggregates the global information of the control-flow dependency relations into its representation, utilizing the representation of its neighbor. We obtain the final representation for each BB as b_k^T. To acquire the representation for the function f_i, we apply a graph readout operation such as *sum-readout*, described as follows,

$$g^{(T)} = \sum_{b_k \in B} b_k^T \tag{3.1}$$

We assign the value of $g^{(T)}$ (a.k.a. CFG embedding) to f_i. The graph readout operation can be replaced with *mean-readout* or *max-readout*.

GoG Graph Embedding Layer: once all the functions have been converted to fixed-length graph embeddings, we then feed $\mathcal{G}$ to the second layer of *CFG2VEC*, the *GoG Embedding Layer*. Here, for each function f_i we apply another L iterations of graph convolution with $\mathcal{F}$ and $\mathcal{C}$. The updates can be illustrated as follows,

$$f_k^{(l+1)} = f_{GoG}(U f_k^l + \sum_{f_m \in C_k} V f_m^{(l)}) \tag{3.2}$$

where f_{GoG} is a non-linear activation function and C_k is the list of adjacent functions (calling) for the function f_k and $U \in R^{d \times d}$ and $V \in R^{d \times d}$ are the weights to be learned during the training. Lastly, we take the $f_k^{(L)}$ as the representation that considers both CFG and GoG graph structures. We use these updated representations to perform cross-architecture function similarity learning.

Siamese-based Cross-Architectural Function Similarity: given a batch of GoGs $B = \{GoG_1, GoG_2, ..., GoG_B\}$, we apply the hierarchical graph neural network to acquire the set of updated function embeddings, denoted as $B_F = \{f_1^{(T)}, f_2^{(T)}, ..., f_K^{(T)}\}$. We calculate the function similarity for each function pair with cosine similarity, denoted as $\hat{y} \in [-1, 1]$. The

loss function J between $\hat{Y}$ and a ground-truth label y, which indicates whether a pair of functions have the same function or not, is calculated as follows,

$$J(\hat{y}, y) = \begin{cases} 1 - y, & \text{if y=1,} \\ MAX(0, \hat{y} - m), & \text{if y=-1,} \end{cases} \tag{3.3}$$

the final loss L is then calculated as follows,

$$L = H(Y, \hat{Y}) = \sum_{i} (J(\hat{y}_i, y_i)), \tag{3.4}$$

where Y stands for ground-truth labels (either similarity or dissimilarity), and $\hat{Y}$ represents the corresponding predictions. More specifically, we denote a pair of functions as similar if they are the same but compiled with different CPU architectures. The m is a constant to prevent the learned embeddings from becoming distorted (by default, 0.5). To maintain the balance between positive and negative training samples, we developed a custom batching algorithm. The function leverages the knowledge gained by adding a binary of some package to a given batch to find and add a binary for the same package, built for a different architecture, to the provided batch as a positive sample. It will also include a binary from another package as a negative sample. This will give any batch a balanced proportion of positive and negative samples. Finally, we use the loss L to update all the associated weights in our neural networks with an *Adam* optimizer. Once trained, we use the model to perform function name reconstruction tasks.

3.4 Experimental Results

In this section, we evaluate *CFG2VEC*'s capability in predicting function names. We first describe the dataset preparation and the training setup processes. Then, we present the

comparison of *CFG2VEC* against baseline in predicting function names. Although many baseline candidates tackle the same problem [79, 9, 51, 66], some require purchasing a paid version of IDA Pro to preprocess datasets, and some even do not open source their implementations. Therefore, [79] was the only feasible choice, as running other models using our datasets was almost impossible. Next, we also show the result of the ablation study over *CFG2VEC*. Besides, we exhibit that our *CFG2VEC* can perform architecture-agnostic prediction better than the baseline. Lastly, we illustrate the real-world use case where our *CFG2VEC* is integrated as a *Ghidra* plugin application for assisting in resolving challenging reverse engineering tasks. We conducted all experiments on a server equipped with Intel Core i7-7820X CPU @3.60GHz with 16GB RAM and two NVIDIA GeForce GTX Titan Xp GPUs.

3.4.1 Dataset Preparation

Our evaluating data source is the ALLSTAR (*Assembled Labeled Library for Static Analysis Research*) dataset, hosted by *Applied Physics Laboratory* (APL) [153]. It has over 30,000 Debian Jessie packages pre-built from `i386`, `amd64`, `ARM`, `MIPS`, `PPC`, and `s390x` CPU architectures for software reverse engineering research. The authors used a modified *Dockcross* script in docker to build each package for each supported architecture. Then, they saved each resulting ELF with its symbols, the corresponding source code, header files, intermediate files (.o, .class, .gkd, .gimple), system headers, and system libraries altogether.

To form our datasets, we selected the packages that have ELF binaries built for the `amd64`, `armel`, `i386`, and `mipsel` CPU architectures. `i386` and `amd64` are widely used by general computers, especially in the Intel and AMD products, respectively. `MIPS` and `ARM` are crucial in embedded systems, smartphones, and other portable electronic devices [53]. In practice, we excluded the packages with only one CPU architecture in the ALLSTAR dataset. Ad-

ditionally, due to our limited local computing resources, we eliminated packages that were too large to handle. We checked each selected binary on whether the ground-truth symbol information exists using the *Ghidra* decompiler and Linux `file` command and removed the ones that do not have them. Lastly, we assembled our primary dataset, called the *AS-4cpu-30k-bin* dataset, that consists of 27572 pre-built binaries from 1117 packages and 4 CPU architectures, as illustrated in Table 3.1.

Our preliminary experiment revealed that the evaluation had a data leakage issue when splitting the dataset randomly. Therefore, we performed a non-random variant train-test split with a 4-to-1 ratio on the *AS-4cpu-30k-bin* dataset, selecting roughly 80% of the binaries for the training dataset and leaving the rest for the testing dataset. We referenced [180] for their splitting methods, aiming to ensure that the binaries that belong to the same packages stay in the same set, either the training or testing sets. Such a variant splitting method allows us to evaluate *CFG2VEC* truly.

Next, we converted binaries in the *AS-4cpu-30k-bin* dataset into their *Graph-of-Graph* (GoG) representations leveraging the GDT mentioned previously in Section 3.3.2. Notably, we processed a batch of binaries related to one package at one time as developers might define user functions in different modules of the same package while putting prototype declarations in that package's main module. For this case, *Ghidra* indeed recognizes two function instances while one only contains the function declaration and another has its actual function content. As these two instances correspond to the same function name and one contains only dummy instructions, they can thus create noise in our datasets, thus affecting our model's learning. To cope with this, our GDT also searches from other binaries of the same package for the function bodies. If found, our GDT associates that user function with the function graph node with the actual content data. Besides user functions, library function calls may exist, and searching their function bodies in the same package would fail for dynamically loaded binaries. Under such circumstances, *Ghidra* would recognize these functions as *ThunkFunc-*

tions[3] which only contain one dummy instruction. As a workaround, we removed these *ThunkFunctions* from our data as they might mislead the model's learning. Applying this workaround indicates that our model works in predicting function names for the user and statically linked functions.

Table 3.1: The statistics of datasets used in our experiments.

Dataset / #	pkg/bin	func node/edge[1]	bb node/edge[2]
AS-4cpu-30k-bin	1117/27,572	51.17/97.14	14.12/19.98
AS-3cpu-9k-bin	633/9,000	44.01/79.06	12.24/17.07
AS-i386-3k-bin	633/3,000	45.31/87.70	11.45/15.97
AS-amd64-3k-bin	633/3,000	42.28/74.07	12.28/17.18
AS-armel-3k-bin	633/3,000	44.45/75.41	13.00/18.07

[1] # of average functions and edges in each binary
[2] # of average bb blocks and edges from each function

Model	Training dataset	Testing dataset	P@1[1]	P@2[1]	P@3[1]	P@4[1]	P@5[1]
CFG2VEC	*AS-4cpu-30k-bin*	*AS-noMipsel-300-bin*	97.05%	99.47%	99.47%	99.47%	99.47%
CFG2VEC	*AS-4cpu-20k-bin*	*AS-noMipsel-300-bin*	74.22%	75.76%	75.78%	75.78%	78.78%
CFG2VEC	*AS-3cpu-9k-bin*	*AS-amd-100-bin*	69.18%	69.98%	69.98%	69.98%	69.98%
		AS-i386-100-bin	69.41%	70.39%	70.39%	70.39%	70.39%
		AS-armel-100-bin	70.66%	71.04%	71.11%	71.11%	71.11%
		AS-noMipsel-300-bin	69.75%	70.47%	70.50%	70.50%	70.50%
[79]-`amd64`[2]	*AS-amd64-3k-bin*	*AS-amd-100-bin*	29.32%	-	-	-	-
[79]-`i386`[2]	*AS-i386-3k-bin*	*AS-i386-100-bin*	52.64%	-	-	-	-
[79]-`armel`[2]	*AS-armel-3k-bin*	*AS-armel-100-bin*	53.65%	-	-	-	-

[1] P@k measures if the actual function name is in the top k of the predicted function names.
[2] These models only provide the top 1 function name prediction; hence they only have P@1 value.

Table 3.2: The performance evaluation of cfg2vec for function name prediction against [79].

We experimented [79] with our datasets, referencing to their implementation[4]. As [79] used a dataset with 3,000 binaries for experiments, we followed accordingly, preparing datasets with smaller but similar sizes. We achieved this by downsampling from our primary *AS-4cpu-30k-bin* dataset, creating the *AS-3cpu-9k-bin* dataset which has 9,000 binaries for `i386`, `amd64`, and `armel` CPU architectures. Furthermore, as [79] supports only one CPU architecture at a time, we then separated the *AS-3cpu-9k-bin* dataset into different CPU architectures, generating three training datasets for testing [79]: *AS-i386-3k-bin*, *AS-amd64-3k-bin*, and

[3]ThunkFunction Manual: `https://ghidra.re/ghidra_docs/api/ghidra/program/model/listing/ThunkFunction.html`

[4]Debin's [79] repository: `https://github.com/eth-sri/debin`

AS-armel-3k-bin. For training, we utilized the `strip` Linux command, converting our original data into three: the original binaries (*debug*), stripped binaries with debug information (*stripped*), and stripped binaries without debug information (*stripped_wo_symtab*) to follow [79]'s required data format. For evaluation, we sampled 100 binaries from our primary dataset for each CPU architecture, labeled *AS-amd-100-bin*, *AS-i386-100-bin*, *AS-armel-100-bin*, and *AS-mipsel-100-bin*. We also have another evaluation dataset called *AS-noMipsel-300-bin*, which contains roughly 300 binaries produced for the `amd64`, `i386`, and `armel` platforms. Table 3.1 summarizes the data statistics for all these datasets, including the numbers of packages and binaries, the average number of function nodes, edges, and BB nodes. The following sections will detail how we utilized these datasets during our experiments.

3.4.2 Evaluation: Function Name Prediction

Table 3.2 demonstrates the results of *CFG2VEC* in predicting function names. For the baseline, we followed [79]'s best setting where the feature dimension of register or stack offset are both 100 to train with our prepared datasets. For *CFG2VEC*, we used three GCN layers and one GAT convolution layer in both graph embedding layers. For evaluation, we calculate the p@k (e.g., precision at k) metric, which refers to an average hit ratio over the top-k list of predicted function names. Specifically, we feed each binary represented in GoG into our trained model, converting each function $f \in F$ and acquiring its function embedding h_f. Then, we calculate pair-wise cosine similarities between h_f and all the other function embeddings, forming a top-k list by selecting k names in which their embeddings are top-kth similar to h_f. If the ground-truth function name is among the top-k list of function name predictions, we regard that as a hit; otherwise, it is a miss. During experiments, we set the top-k value to be 5, so our model can recommend the best five possible names for each function in a binary.

As shown in Table 3.2, *CFG2VEC*, trained with the *AS-3cpu-9k-bin* dataset, can achieve a 69.75% prediction accuracy (e.g., p@1) in inferring function names. For [79], we had to train their models for each CPU architecture separately as it cannot train in a cross-architectural manner. Even so, for `amd64` binaries, [79] only achieves 29.32% precision, while for `i386` and `armel`, it performs 52.64% and 53.65%, respectively. This result indicates that in any case, our *CFG2VEC* outperforms [79]. Besides, while [79] only yields one prediction, our *CFG2VEC* suggests five choices, making it flexible for our users (e.g., REs) to select what they believe best fits the function among the best k predicted names. The p@2 to p@5 in Table 3.2 demonstrate that our *CFG2VEC* can provide enough hints of function names for users. For example, p@5 of *CFG2VEC* trained with our *AS-3cpu-9k-bin* dataset can achieve 70.50% precision across all the CPU architecture binaries. We also experimented our *CFG2VEC* with larger datasets. From Table 3.2, we can observe that *CFG2VEC* can have 5.04% performance gain in correctly predicting function names (e.g., p@1). Moreover, the gain increases to 28% when training *CFG2VEC* with the *AS-4cpu-30k-bin* dataset. We believe training on a larger dataset implies training with a more diversified set of binaries. This allows our model to acquire more knowledge, thus being capable of extracting more robust features for binary functions. In summary, this result indicates that compared to the baseline, our model can effectively provide contextually relevant names for functions in the decompiled code to our users.

Table 3.3: The comparison between *CFG2VEC* and its ablated variations.

Arch	[79]	*GCN-GAT*	*2GCN*	*2GCN-GAT*	*CFG2VEC*
`amd64`	29.32%[1]	61.59%	69.49%	69.56%	70.66%
`armel`	52.64%[2]	66.40%	68.59%	68.92%	69.19%
`mipsel`	53.65%[3]	66.47%	68.17%	68.56%	69.41%
Overall	45.20%	64.82%	68.75%	69.01%	69.75%

[1] Evaluation results for [79]-`amd64` model.
[2] Evaluation results for [79]-`i386` model.
[3] Evaluation results for[79]-`armel` model.

We also experimented with various ablated network setups to study how each component of

CFG2VEC contributes to performance. First, we simplified our *CFG2VEC* by stripping one GCN layer from the original experimental setup. As shown in Table 3.3, we called this setup *2GCN-GAT* which slightly decreased the performance by 0.75%. Then, from *2GCN-GAT* setup, we further removed the GAT layer, calling it *2GCN*. We again observed a marginal performance decrease (<1%). Next, we eliminated another GCN layer from *2GCN-GAT*, constructing the *GCN-GAT* setup. For *GCN-GAT*, we saw a drastic drop (4.2%) which highlights that the number of GCN layers can be an essential factor in the performance. Specifically, we found that going from 1 to 2 GCN layers improves prediction accuracy by more than 4%. However, we do not observe a significant performance gain when increasing the number of GCN layers to more than three. Therefore, we retained the original *CFG2VEC* model with its three GCN layers. All in all, as shown in Table 3.3, all these ablated models, still outperform [79], which we attributed to the GoG representation we made for each binary in the dataset.

3.4.3 Evaluation: Architectural-agnostic Prediction

Table 3.4 demonstrates our *CFG2VEC*'s capability in terms of cross-architecture support. As [79] supports training one CPU architecture at a time, we had to train it multiple times during experiments. Specifically, we trained [79] on three datasets: *AS-amd64-3k-bin*, *AS-i386-3k-bin*, and *AS-armel-3k-bin*, calling resulting trained models, [79]-`amd64`, [79]-`i386`, and [79]-`armel`, respectively. For these baseline models, we observe that they perform well when tested with the binaries built on the same CPU architecture but poorly with the ones built on different CPU architectures. For instance, [79]-`amd64` achieves 29.3% accuracy for `amd64` binaries, but performs worse for `i386` and `armel` binaries (13.8% and 7.1%). Similarly, [79]-`i386` achieves 52.6% accuracy for `i386` binaries, but performs worse for `amd64` and `armel` binaries (6.2% and 1.1%). Lastly, [79]-`armel` achieves 53.6% accuracy for `armel` binaries, but performs worse for `amd64` and `i386` binaries (11.8% and 8.9%). We used the top-1 prediction

generated from *CFG2VEC* (a.k.a., p@1) as the comparing metric as [79] produces only one prediction per each function. From the results, we observe that *CFG2VEC* outperforms [79] across all three tested CPU architectures. The fact that *CFG2VEC* performs consistently well across all CPU architectures indicates that our *CFG2VEC* supports cross-architecture prediction.

To evaluate the capability of generalizing the learned knowledge, we tested all models with the *AS-mipsel-100-bin* dataset, which has binaries built from another famous CPU architecture, `mipsel`, that our *CFG2VEC* does not train before. For [79], it has lower performance when testing on binaries built from the CPU architectures that it did not train before, exampled by the highest accuracy of [79] to be 13.84% when trained on *amd64* binaries and evaluated on `i386` binaries. In our work, as Table 3.4 shows, our *CFG2VEC* achieves 36.69% accuracy when trained with `amd64`, `i386`, and `armel` binaries but tested on `mipsel` binaries. For [79], it does not even support analyzing `mipsel` binaries. In short, these results demonstrate that our *CFG2VEC* outperforms our baseline in the function name prediction task on cross-architectural binaries and generalizes better to the binaries built from unseen CPU architectures. To further investigate *CFG2VEC*'s cross-architecture performance, we trained it on three datasets, each consisting of binaries built for two different architectures. We then gave the resulting trained models names that indicated the architectures from which the binaries were derived: *CFG2VEC*-armel-i386, *CFG2VEC*-amd64-i386, and *CFG2VEC*-armel-amd64. These results show that our model performs well in the function name prediction job across all of these scenarios, including when tested on binaries compiled for unknown CPU architectures.

Table 3.4: The cross-architectural comparison between cfg2vec and [79]

Model	Testing dataset	P@1
CFG2VEC-3-Arch	*AS-amd-100-bin*	69.18%
	AS-i386-100-bin	70.66%
	AS-armel-100-bin	69.41%
	*AS-mipsel-100-bin**	36.69%
CFG2VEC-amd64-armel	*AS-amd-100-bin*	68.53%
	*AS-i386-100-bin**	39.23%
	AS-armel-100-bin	69.11%
	*AS-mipsel-100-bin**	32.21%
CFG2VEC-amd64-i386	*AS-amd-100-bin*	68.59%
	AS-i386-100-bin	69.09%
	*AS-armel-100-bin**	34.20%
	*AS-mipsel-100-bin**	38.26%
CFG2VEC-armel-i386	*AS-amd-100-bin**	42.96%
	AS-i386-100-bin	67.45%
	AS-armel-100-bin	63.86%
	*AS-mipsel-100-bin**	36.61%
[79]-`amd64`	*AS-amd-100-bin*	29.32%
	*AS-i386-100-bin**	13.84%
	*AS-armel-100-bin**	7.08%
	*AS-mipsel-100-bin**	-
[79]-`i386`	*AS-amd-100-bin**	6.23%
	AS-i386-100-bin	52.64%
	*AS-armel-100-bin**	1.05%
	*AS-mipsel-100-bin**	-
[79]-`armel`	*AS-amd-100-bin**	11.82%
	*AS-i386-100-bin**	8.86%
	AS-armel-100-bin	53.65%
	*AS-mipsel-100-bin**	-

* indicates that dataset was not used during the training.

3.4.4 The Practical Usage of CFG2VEC

In this section, we demonstrate how *CFG2VEC* assists REs in dealing with *Defense Advanced Research Projects Agency* (DARPA) *Assured MicroPatching* (AMP) challenges binaries. The AMP program aims at enabling fast patching of legacy mission-critical system binaries,

enhancing decompilation and guiding it toward a particular goal of a *Reverse Engineer* (RE) by integrating the existing source code samples, the original build process information, and historical software artifacts.

The MINDSIGHT project: our multi-industry-academia initiative between Siemens, JHU/APL, BAE, and UCI jointly developed a project, *Making Intelligible Decompiled Source by Imposing Homomorphic Transforms* (MINDSIGHT). Our team focused on building an automated toolchain integrated with *Ghidra*, aiming to enable the decompilation process with (1) a less granular identification of modular units, (2) an accurate reconstruction of symbol names, (3) the lifting of binaries to stylized C code, (4) a principled and scalable approach to reason about code similarity, and (5) the benchmarking of new decompilation techniques using state-of-the-art embedded software binary datasets. To date, our team has developed an open-source tool, *CodeCut*[5], to improve the accuracy and completeness of *Ghidra*'s module identification, providing an automated script-based decompilation analysis toolchain to ease the RE's expert interpretation. Besides, we also developed a *Homomorphic Transform Language* (HTL) to describe transformations on *Abstract Syntax Tree* (AST) languages and the rules of their composition. Our tool, integrated with *ghidra*, allows developers to transform the decompiled code syntactically while keeping it semantically equivalent. The key idea is to use this HTL to morph a *Ghidra* AST into a GCC AST to lift the decompiled binary to a high-level C representation. This process can make it easier for REs to comprehend the binary code. *CFG2VEC* is another tool developed in the MINDSIGHT project, enabling the reconstruction of function names, saving the manual guesswork from REs.

The cfg2vec plugin: in *MINDSIGHT* project, we incorporated *CFG2VEC* into *Ghidra* decompiler as a plugin application. Our *CFG2VEC* plugin assists REs in comprehending the binaries by providing a list of potential function names for each function without its name. Technically, like all *Ghidra* plugins, our *CFG2VEC* plugin bases on Java with its

[5] *CodeCut*'s repository: `https://github.com/DARPAMINDSIGHT/CodeCut`

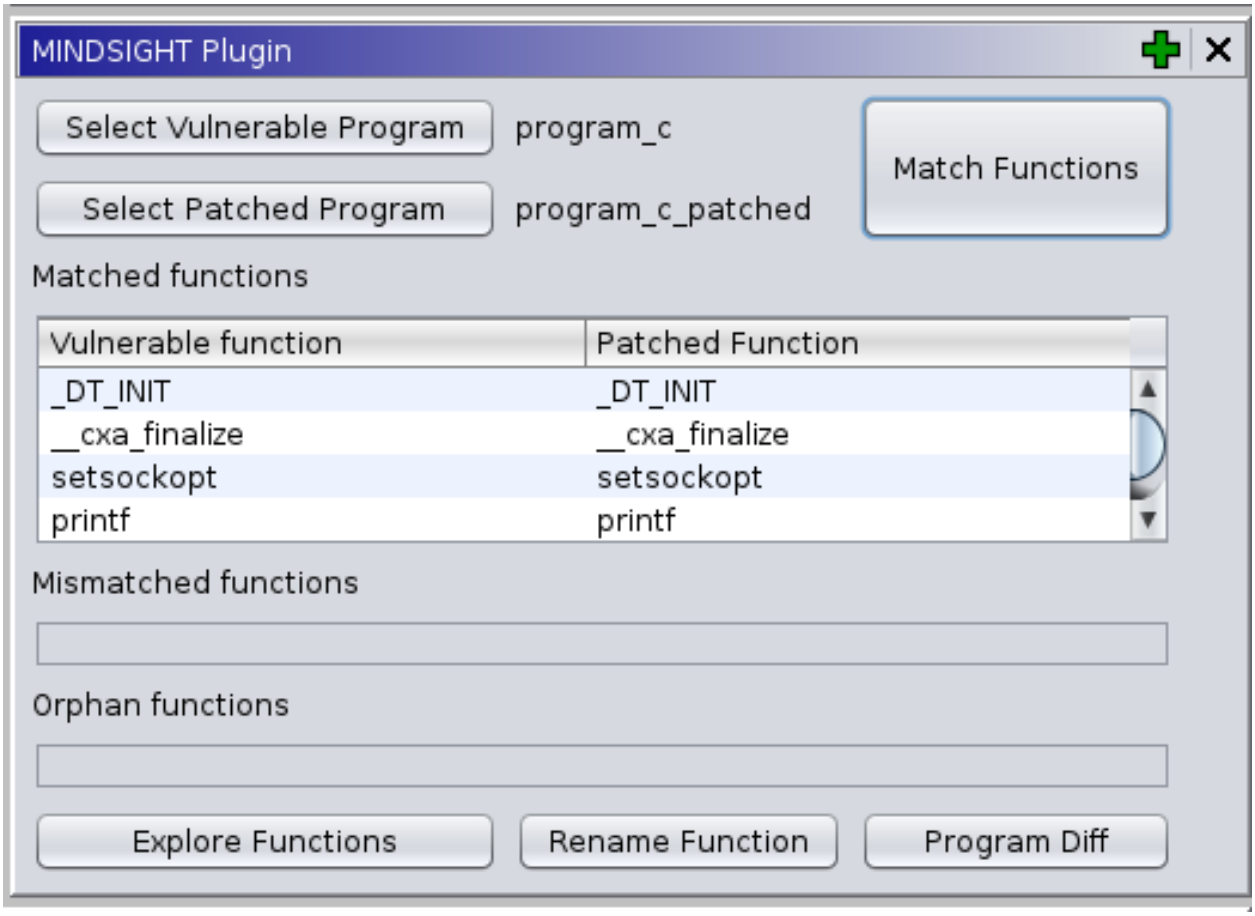

Figure 3.5: The plugin screenshot integrated into Ghidra.

core inference modules implemented as a REST API in Python 3.8. Once the metadata of a stripped binary is extracted from *Ghidra* decompiler, it is then sent to the *CFG2VEC* end-point, which calculates and returns the inferred mappings for all the functions. Figure 3.5 demonstrates the user interface of our *CFG2VEC* plugin. In this scenario, the user must provide the vulnerable and the reference binary with extra debug information, such as function names. The "Match Functions" button triggers *CFG2VEC* functionality and displays the function mapping results in three tables:

- *Matched Table*: displays the mapping of similar functions.
- *Mismatched Table*: displays the mapping of *dissimilar* functions and, therefore, candidates for patching.

- *Orphan Table*: displays the mapping of functions with a low confidence score.

The groupings reduce REs' workload. Rather than inspecting all functions, they can focus on patching candidate functions (mismatched functions) and the orphans. The "Explore Functions" button invokes Ghidra's function explorer, where the two functions can be compared side-by-side, as shown in Figure 3.5. This utility allows the user to switch between C and assembly language, thus assisting in confirming or modifying the mappings from the three tables. Regarding *CFG2VEC*'s function prediction, the "Rename Function" button takes the selected row from the tables and imposes the name from the patched binary in the vulnerable binary. When the "Match Functions" button fires, we invoke the FCG and CFG generators for the two programs (vulnerable and patched).

The use-case for AMP challenge binaries: DARPA AMP challenges is about REs to patch a vulnerability regarding a weak encryption algorithm where the encryption of communication traffic was accomplished with a deprecated cipher suite, Triple DES or 3DES [31]. For this challenge, REs have to analyze the vulnerable binary, identify functions and instructions to be patched, *3DES cipher suite* in this case, and patch 3DES-related function calls and instructions with the ones for AES [20]. All these steps happen at the decompiled binary level, and the vulnerable binaries are optimized by a compiler and stripped of the debugging information and function names. Furthermore, these binaries are sometimes statically linked against libraries such as GNU C Library [141] or OpenSSL, which introduce many extra functions to the binary (some of which will never be called/used). Given these complications, it becomes a non-trivial task for an RE to make sense of all these functions, find the problem, and successfully patch the problem. The direct usage of our *CFG2VEC* plugin was to pick a function of interest with stripped information and see predictions of potential function names or matching functions from the available reference binary to confirm that whether this function is in the critical path during RE's problem solving. As Figure 3.5 shows, our plugin allows users to see possible matches between functions from a stripped vulnerable

binary and functions from a patched (reference) binary with extra information. REs may then leverage such information and make appropriate notes for that function, allowing them to complete their jobs more efficiently. The main feedback we received from REs who used the tool was that this is the functionality REs would like to have. However, the accuracy and usability of the tool were not high enough to truly utilize the tool's potential.

3.5 Conclusion

In this chapter, we present *CFG2VEC*, a Hierarchical Graph Neural Network-based approach for software reverse engineering. Building on top of *Ghidra*, our *CFG2VEC* plugin can extract a *Graph-of-Graph* (GoG) representation for binary, combining the information from Control-Flow Graphs (CFG) and Function-Call Graphs (FCG). *CFG2VEC* utilizes a hierarchical graph embedding framework to learn the representation for each function in binary code compiled into various architectures. Lastly, our *CFG2VEC* utilizes the learned function embeddings for function name prediction, outperforming the state-of-the-art [79] by an average of 24.54% across all tested binaries. By increasing the amount of data, our model achieved 51.84% better. While [79] requires training once for each CPU architecture, our *CFG2VEC* still can outperform consistently across all the architectures, only with one training. Besides, our model generalizes the learning better [79] to the binaries built from untrained CPU architectures. Lastly, we demonstrate that our *CFG2VEC* can assist the real-world REs in resolving *Darpa Assured MicroPatching* (AMP) challenges.

Chapter 4

Graph Learning for Autonomous Driving Systems

This chapter explores *Graph Learning* for *Autonomous Driving Systems* (ADSs). In literature, there is considerable evidence that evaluating the subjective risk level of driving decisions can improve the safety of self-driving cars in both typical and complex driving scenarios. In this chapter, we propose a novel data-driven approach that uses *scene-graphs* as intermediate representations for modeling the subjective risk of driving maneuvers. This novel approach, nailed as SG2VEC, can assess the risk of various driving maneuvers more accurately than state-of-the-art. Besides, its model transfers the knowledge learned from a lane-changing synthesized dataset more effectively than the state-of-the-art model. Moreover, adding spatial and temporal attention layers improves its performance and explainability. Even more, SG2VEC runs faster and consumes lesser energy than the state-of-the-art method, making it more suitable for implementation on the edge device.

4.1 Introduction

Autonomous Driving Systems (ADSs) have advanced significantly in recent years. However, navigation is still challenging in complex urban environments since the scenarios are highly variable and complex [115, 194, 121]. The continued reports of autonomous vehicle crashes only highlight these challenges [106, 52, 113, 107]. A risk-based approach for autonomous driving has the potential to address this challenge and better assure driving safety. Within this context, the effectiveness of understanding the driving scenes and quantifying the risk of driving decisions becomes particularly crucial for ADSs.

Several papers have leveraged state-of-the-art deep learning architectures for modeling subjective risk [194, 195]. Such methods typically use *Convolutional Neural Networks* (CNNs) and *Long-Short Term Memory Networks* (LSTMs). They have proven effective at capturing features essential for modeling subjective risk in both spatial and temporal domains [195]. However, it is unclear whether these methods can capture critical higher-level information, such as the relationships between traffic participants in a given scene. Failing to capture these relationships can result in poor ADS performance in complex scenarios.

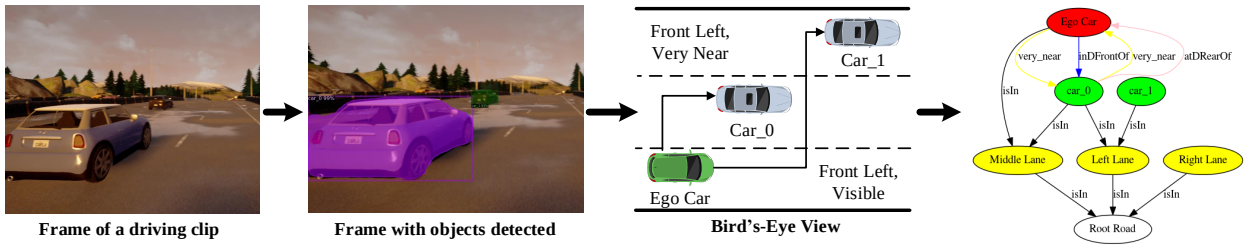

Figure 4.1: An illustration of *scene-graph* extraction using the *Real Image Pipeline*. In this process, the first step is to detect a list of objects on each clip frame. Then, we project each frame to its bird's-eye view to better approximate the spatial relations between objects. Finally, we construct a *scene-graph* using the list of detected objects and their attributes.

Research Challenges and Contributions: overall, designing a risk assessment system for ADSs using data-driven approaches presents the following challenges:

1. Designing a reliable method that can handle a wide range of complex and unpredictable traffic scenarios,

2. Building a model that is transferable from the simulation setting to the real-world setting because the real-world datasets for supervised training are limited,

3. Building a model that can provide explainable decisions.

To overcome Research Challenge 1, deep learning based subjective risk assessment methods must be trained on large datasets covering a wide range of "corner cases" (especially risky driving scenarios), which are expensive and time-consuming to generate [54]. Researchers use synthesized datasets containing many examples of these corner cases to address this issue. However, for these to be valuable, a model must be able to transfer the knowledge gained from simulated training data to real-world situations (Research Challenge 2). A standard method for measuring a model's ability to generalize is *transferability*, where a model's accuracy on a dataset different from the training dataset is evaluated. If a model can transfer the knowledge gained from a simulated training set to a real-world testing set effectively, it will likely perform better in unseen real-world scenarios.

Even if these existing methods can transfer knowledge well, the predictions of such methods lack *explainability*, which is crucial for establishing trust between ADSs and human drivers [16, 12, 2]. *Explainability* refers to the ability of a model to effectively communicate the factors that influenced its decision-making process for a given input, particularly those that might lead the model to make incorrect decisions [2, 99]. Suppose a model can give attention to the aspects or entities in a traffic scene that make the scenario risky or non-risky. In that case, it can improve its decision, and its decisions become more explainable [168] (Research Challenge 3).

To address these limitations, we propose a *scene-graph* augmented data-driven approach for assessing the subjective risk of driving maneuvers, where the *scene-graphs* serve as *Interme-*

diate Representations (IR) as shown in Figure 4.1. The key advantage of using *scene-graph* as IR is that they allow us to model the relationships between the participants in a traffic scene, thus potentially improving the model's understanding of a scene. Our proposed architecture consists of three major components: (i) a pipeline to convert the images of a driving clip to a sequence of *scene-graphs*, (ii) an MR-GCN to convert each of the *scene-graphs* to an embedding (a vectorized representation), and (iii) an LSTM for temporally modeling the sequence of embeddings of the respective *scene-graphs*. Our model also contains multiple attention layers: (i) a node attention layer before the embedding of a *scene-graph* is computed, and (ii) an attention layer on top of the LSTM, both of which can further improve its performance and explainability. For training the model, we formulate the problem of subjective risk assessment as a supervised *scene-graph* sequence classification problem. The key contributions are as follows:

- We present a novel *scene-graph* augmented data-driven approach for assessing the risk of driving actions in autonomous vehicles.
- We demonstrate that our approach outperforms existing methods of risk assessment across a wide range of scenarios using lane change as a use case.
- We demonstrate that multi-level attention in our proposed approach provides better explainability.
- We demonstrate that our *scene-graph* based approach can better transfer knowledge gained from simulated environments to real-world risk assessment tasks.

4.2 Related Works

4.2.1 ADS Design Philosophies

Two broad approaches for designing ADSs are (i) modular design and (ii) end-to-end design [194]. Most modular approaches comprise a pipeline of separate components from the sensory inputs to the actuator outputs, while end-to-end approaches generate output directly from their sensory inputs [126, 25]. One advantage of a modular design approach is the division of a task into an easier-to-solve set of sub-tasks that have been addressed in other fields such as robotics [105], computer vision [92] and vehicle dynamics [130]. Therefore, prior knowledge from these fields can be leveraged when designing the components corresponding to the sub-tasks. However, one disadvantage of such an approach is the complexity of the whole pipeline [194]. End-to-end approaches can achieve good performance with a smaller network size because they perform feature extraction from sensor inputs implicitly through the network's hidden layers [25]. However, the authors in [32] point out that the needed level of supervision is too weak for the end-to-end model to learn critical controlling information (e.g., from image to steering angle), so it can fail to handle complicated driving maneuvers.

A third approach was first proposed by DeepDriving [32], called the *direct perception* approach. In their approach, a set of *affordance indicators*, such as the distance to lane markings and other cars in the current and adjacent lanes, are extracted from an image and serve as an IR for generating the final control output. They prove this IR is effective for simple driving tasks such as lane following and generalizing to real-world environments. Authors in [14] use a collection of filtered images, each representing a piece of distinct information, as the IR. They state that the IR used in their approach allows the training to be conducted on real or simulated data, facilitating testing and validation in simulations before testing on a real car. Moreover, they show that it is easier to synthesize perturbations to the driving trajectory at the mid-level representations than at the level of raw sensors, enabling them to

produce non-expert behaviors such as off-road driving and collisions. The authors in [195] use Mask-RCNN [80] to color the vehicles in each input image, producing a form of IR. In contrast to the works mentioned above, our approach uses a *scene-graph* IR that encodes the spatial and semantic relations between all the traffic participants in a frame.

4.2.2 AV Scene-Graphs and Optimization Techniques

Several works have proposed graph-based methods for scene understanding. For example, [116] proposed a multi-relational graph convolutional network (MR-GCN) that uses both spatial and temporal information to classify vehicle driving behavior. Similarly, in [109], an *Ego-Thing* and *Ego-Stuff* graph are used to model and classify the ego vehicle's interactions with moving and stationary objects, respectively. In our prior work, we demonstrated that a *scene-graph* sequence embedding approach assesses driving risk better than the state-of-the-art CNN-LSTM approach [193]. In [193], we utilized an architecture consisting of MR-GCN layers for spatial modeling and an LSTM with attention for temporal modeling; however, this architecture was only capable of performing binary sequence-level classification over a complete video clip. Thus, although our prior architecture could accurately assess the subjective risk of complete driving sequences, it was not capable of predicting the future state of a scene.

Current autonomous driving systems consume a substantial amount of power (up to 500 Watts for the Nvidia DRIVE AGX Pegasus), demanding more robust cooling and power delivery mechanisms. Thus, many have tried to optimize AV tasks for efficiency without sacrificing performance. Existing approaches have proposed methods for jointly optimizing power consumption and latency for localization [10], perception [13], and control [89]. However, to the best of our knowledge, no work has explored this optimization for AV safety systems, such as collision prediction systems.

4.2.3 Risk Assessment

In prior research, the problem of risk assessment for autonomous driving has been tackled by modeling either the *objective risk* or the *subjective risk* [72, 64, 16]. The *objective risk* is defined as the objective probability of an accident occurring and is usually determined by statistical analysis [72]. Some works have focused on minimizing the *objective risk* by modeling the trajectories of vehicles [104, 173] to guarantee safe driving. *Subjective risk* refers to the driver's perceived risk and is an output of the driver's cognitive process [64, 16]. One primary reason why assessing subjective risk is important is because it accounts for the human behavior perspective and its critical role in anticipating risks, as many works point out [15, 16, 64]. Further, studies such as [163, 72] provide direct evidence that a driver's subjective risk assessment is inversely related to the risk of traffic accidents. Similarly, [16] suggests that augmenting an objective risk assessment system with *subjective risk* techniques can improve overall risk assessment performance. For these reasons, our goal in this work is to build a model for subjective risk assessment. In Figure 4.2, we present examples of a lane change that is both *subjectively* and *objectively* risky and a lane change that is *subjectively safe* but *objectively risky* from our driving dataset. An objective and subjective approach would likely identify the obvious risk factor of the close-proximity, high-speed vehicle in (a). However, an objective risk assessment approach may incorrectly consider (b) to be risky because of the ego car's perceived trajectory while this is, in fact, a safe lane change.

Several works have studied subjective risk assessment for autonomous driving systems [64, 16, 195, 196]. In [196], *Hidden Markov Models* (HMMs) and Language Models are used to detect unsafe lane change events. The approach taken in [195] is the most related to our work as it infers the risk level of overall driving scenes with a deep Spatiotemporal neural network architecture. Using Mask-RCNN [80] to generate an IR for each image, their approach achieves a 3% performance gain in risk assessment. They show that the architecture with *Semantic Mask Transfer* (SMT) + CNN + LSTM can perform 25% better than the

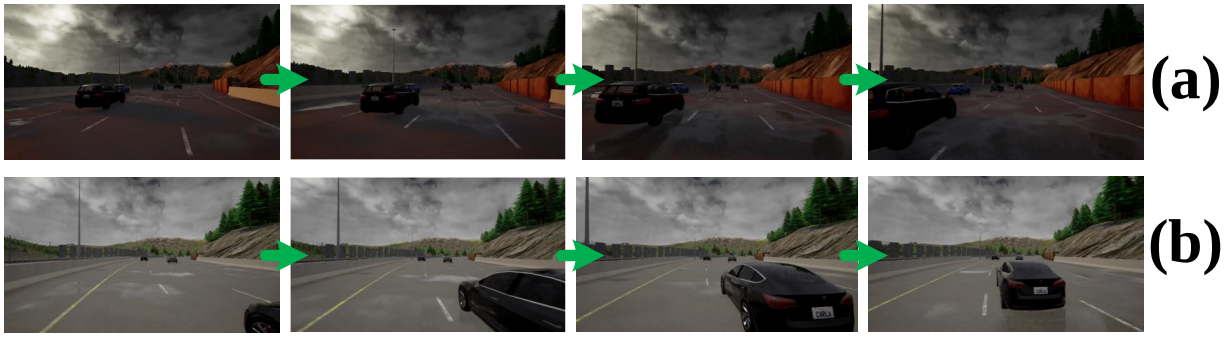

Figure 4.2: An example of (a) a lane change that is both *subjectively* and *objectively* risky as well as (b) a *subjectively safe* but *objectively risky* lane change from our driving dataset. In (a), the ego car starts a safe lane change, but a high-speed vehicle suddenly appears in its blind spot and nearly collides with it. In (b), the ego car appears to drive directly toward the adjacent vehicle but is making a safe lane change on a curved road.

architecture with *Feature Transfer* (FT) + *Frame-by-Frame* (FbF). This result indicates that capturing the spatial and temporal features from a single camera can be useful in modeling subjective risk. However, this approach only considers the spatial features (the latent vector output of the CNN layers) of a frame instead of the relations between all the traffic participants. Our work uses *scene-graphs* as IRs to capture the high-level relationships between all the traffic participants of a scene.

4.2.4 Early Collision Prediction

Since collision prediction is key to the safety of AVs, a wide range of solutions have been proposed by academia and industry. As mentioned earlier, current consumer vehicles use statistics-based SBTMs for collision prediction but can perform poorly in complex situations [91, 49] or react too late to avoid collisions [118, 151]. Expanding on these approaches, companies like Mobileye and Nvidia have proposed more comprehensive mathematical models for ensuring AV safety, namely Responsibility-Sensitive Safety (RSS) [145] and Nvidia Safety Force Field [119], respectively. However, these models are heavily rule-based and can thus be fragile in complex situations with high uncertainty. Additionally, computing future

trajectory constraints with RSS is non-trivial and can require vehicle-specific calibration [68].

Model-based probabilistic and deep learning approaches for collision prediction have also been proposed. For example, [8] proposes a model-based probabilistic technique that uses the roadway geometry, ego trajectory, and position/velocity of road objects to predict future object positions. However, this model is highly conservative and is likely to have a high false-positive rate. Similarly, [171] and [197] use model-based approaches but require significant domain knowledge about the driving scene, such as road geometry information as well as accurate vehicle position and velocity information. [170] proposes a deep learning collision prediction approach. Still, due to its use of pre-processed trajectory data captured from cameras overlooking a highway, it is not ego-centric and cannot be practically used for on-vehicle collision prediction. In a different approach, [155] proposes a Deep Predictive Model (DPM) that used a Bayesian Convolutional LSTM for collision risk assessment where image data, vehicle telemetry data, and driving inputs were all factors in the risk assessment decision. However, this approach was only evaluated on simulated street scenes containing two vehicles and no other dynamic objects. Thus, DPM's performance may suffer when evaluated on more complex road scenarios.

In contrast to these existing works, we propose SG2VEC which captures structural and relational information of a road scene in a *scene-graph* representation and computes a spatio-temporal embedding to predict collisions. Additionally, we perform experiments that were not done in many prior works, such as evaluating each model's capability to transfer knowledge, efficiency on AV hardware, performance on a complex real-world crash dataset, and ability to predict collisions early. We primarily compare our methodology with the DPM as it is the state-of-the-art data-driven collision prediction framework for AVs that considers both spatial and temporal factors. Although the DPM uses multiple modalities for sensing, the results in [155] show that it achieves an accuracy (of 81.95%) that is just 0.24% less using just the image sensing modality. In this work, we compare our proposed SG2VEC method-

ology and the DPM on image-only datasets, which is fair because the DPM's performance does not vary much with the inclusion of other modalities.

4.3 SG2VEC: Scene-Graph Augmented Methodology

4.3.1 Problem Formulation

In our work, we make the same assumption in [195] that the set of driving sequences can be partitioned into two jointly exhaustive and mutually exclusive subsets: risky and safe. We denote the sequence of images of length T by $\mathbf{I} = \{I_1, I_2, I_3, ..., I_T\}$. We assume the existence of a spatiotemporal function f that outputs whether a sequence of driving actions x is safe or risky via a risk label y, as given in Equation 4.1.

$$y = f(\mathbf{I}) = f(\{I_1, I_2, I_3, ..., I_{T-1}, I_T\}), \tag{4.1}$$

where

$$y = \begin{cases} (1,0), & \text{if the driving sequence is safe} \\ (0,1), & \text{if the driving sequence is risky.} \end{cases} \tag{4.2}$$

In this section, we propose a suitable model for approximating the function f. In the model we propose, the first step is the extraction of the *scene-graph* G_t from each image I_t of the video clip $\mathbf{I}$. This step is achieved by a series of processes that we collectively call the *Scene-Graph Extraction Pipeline*. In the second step, these *scene-graphs* are passed through graph convolution layers and an attention-based graph pooling layer. The graph-level embeddings of each *scene-graph*, $\mathbf{h}_{G_t}$, are then calculated using a graph readout operation. Next, these *scene-graph* embeddings are passed sequentially to LSTM cells to acquire the *spatio-temporal*

representation, denoted as $\mathbf{Z}$, of each *scene-graph* sequence. Lastly, we use a Multi-Layer Perceptron (MLP) layer with a *Softmax* activation function to acquire the final inference, denoted as $\hat{y}$, of the risk for each driving sequence $\mathbf{I}$. We describe more details regarding each of our model's components in Section 4.3.3.

4.3.2 Scene-Graph Extraction

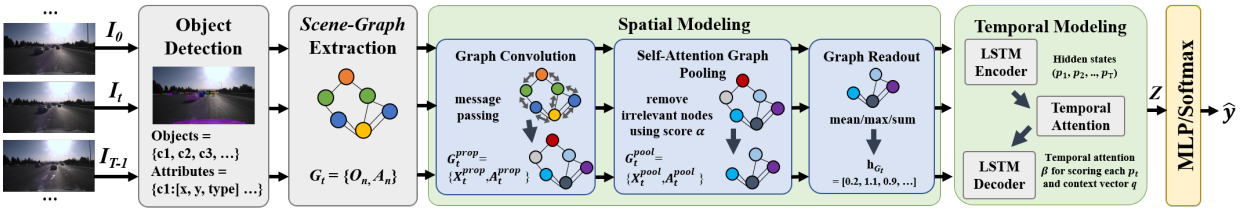

Figure 4.3: An illustration of our model's architecture. First, each image $I_t \in \mathbf{I}$ is converted to a *scene-graph* G_t via the *Scene-Graph Extraction Pipeline*. Next, each *scene-graph* G_t is converted to its corresponding *scene-graph* embedding $\mathbf{h}_{G_t}$ via the graph convolution, pooling, and readout operations in the Spatial Modeling block. Then, the resulting *scene-graph* embeddings are sequentially processed by LSTM and temporal attention layers to acquire the spatiotemporal representation $\mathbf{Z}$ for a *scene-graph* sequence. Finally, the risk inference $\hat{y}$ of the sequence is calculated from $\mathbf{Z}$ using an MLP with a *Softmax* activation function.

Several approaches have been proposed for extracting scene-graphs from images by detecting the objects in a scene and then identifying their visual relationships [179, 183]. However, these works have focused on single general images instead of a sequence of images as it arises in autonomous driving, where higher accuracy is demanded. Thus, we adopted a partially rule-based process to extract objects and their attributes from images called the *Real Image Pipeline*. Besides, to evaluate how our approach performs with *scene-graphs* containing ground truth information, we use the *Carla Ground Truth (GT) Pipeline* as a surrogate for the ideal situation where the attributes for each object can be correctly extracted. We discuss each of these components in detail below.

Real Image Pipeline: In this pipeline, object attributes and bounding boxes are extracted

directly from images using state-of-the-art image processing techniques. As Figure 4.1 shows, we first convert each image I_t into a collection of objects O_t using Detectron2, a state-of-the-art object detection model based on Faster RCNN [174, 137]. Next, we use OpenCV's perspective transformation library to generate a top-down perspective of the image, commonly known as a "birds-eye view" projection [27]. This projection allows us to approximate each object's location relative to the road markings and the ego vehicle. Next, for each detected object in O_t, we use its estimated location and class type (cars, motorcycles, pedestrians, lanes, etc.) to compute the attributes required in building the *scene-graph*.

Carla Ground Truth Pipeline: Object detection and location estimation with solely a monocular camera can be unstable because of factors such as weather and camera position [29], which can impact the correctness of our image-based *scene-graph* construction pipeline and thus our approach's performance. To evaluate our methodology under the assumption that object attributes can be extracted without error, we build our *scene-graphs* using the ground-truth location and class information for each vehicle in the *Carla GT Pipeline*. We extract this information directly from Carla simulator [54] without any image processing steps.

Scene-Graph Construction: After collecting the list of objects in each image and their attributes, we begin constructing the corresponding *scene-graphs* as follows. For each image I_t, we denote the corresponding *scene-graph* by $G_t = \{O_t, A_t\}$ and model it as a directed multi-graph where multiple types of edges connect nodes. The nodes of a *scene-graph*, denoted as O_t, represent the objects in a scene such as lanes, roads, traffic signs, vehicles, pedestrians, etc. The edges of G_t are represented by the adjacency matrix A_t, where each value in A_t represents the type of the corresponding edge in G_t. The edges between two nodes represent the different kinds of relations between them (e.g., near, Front_Left, isIn, etc.).

In assessing the risk of driving behaviors, traffic participants' relations that we consider use-

ful are the distance relations and the directional relations. The assumption made here is that the local proximity and positional information of one object will influence the other's motion only if they are within a certain distance. Therefore, in this work, we extract only the location information for each object and adopt a simple rule to determine the relations between the objects using their attributes (e.g., relative location to the ego car), as shown in Figure 4.1. For distance relations, we assume two objects are related by one of the relations $r \in \{$*Near_Collision* (4 ft.), *Super_Near* (7 ft.), *Very_Near* (10 ft.), *Near* (16 ft.), *Visible* (25 ft.)$\}$ if the objects are physically separated by a distance that is within that relation's threshold. In the case of the directional relations, we assume two objects are related by the relation $r \in \{$*Front_Left*, *Left_Front*, *Left_Rear*, *Rear_Left*, *Rear_Right*, *Right_Rear*, *Right_Front*, *Front_Right*$\}$ based on their relative positions if they are within the *Near* threshold distance from one another.

In addition to directional and distance relations, we also implement the *isIn* relation that connects vehicles with their respective lanes. For the *Carla GT Pipeline*, we extract the ground-truth lane assignments for each vehicle from the simulator directly. For the *Real Image Pipeline*, we use each vehicle's horizontal displacement relative to the ego vehicle to assign vehicles to either the *Left Lane*, *Middle Lane*, or *Right Lane* based on a known lane width. Our abstraction only includes these three-lane areas, and, as such, we map vehicles in all left lanes to the same *Left Lane* node and all vehicles in right lanes to the *Right Lane* node. If a vehicle overlaps two lanes (i.e., during a lane change), we assign it an *isIn* relation to both lanes. Figure 4.1 illustrates an example of resultant *scene-graph*.

4.3.3 Scene-Graph Sequence Model Architecture

The model we propose consists of three major components: a spatial model, a temporal model, and a risk inference component. The spatial model outputs the embedding h_{G_t} for

each scene-graph G_t. The temporal model processes the sequence of *scene-graph* embeddings $\mathbf{h}_I = \{h_{G_1}, h_{G_2}, ..., h_{G_T}\}$ and produces the spatio-temporal embedding $\mathbf{Z}$. The risk inference component outputs each driving clip's final risk assessment, denoted as $\hat{Y}$, by processing the Spatio-temporal embedding $\mathbf{Z}$. The overall network architecture is shown in Figure 4.1. We discuss each of these components in detail below.

Spatial Modeling: The spatial model we propose uses MR-GCN layers to compute the embedding for a *scene-graph*. MR-GCN allows us to capture multiple types of relations on each *scene-graph* $G_t = \{O_t, A_t\}$. In the *Message Propagation* phase, a collection of node embeddings and their adjacency information serve as the inputs to the MR-GCN layer. Specifically, the l-th MR-GCN layer updates the node embedding, denoted as $\mathbf{h}_v^{(l)}$, for each node v as follows:

$$\mathbf{h}_v^{(l)} = \mathbf{\Phi}_0 \cdot \mathbf{h}_v^{(l-1)} + \sum_{r \in \mathbf{A_t}} \sum_{u \in \mathbf{N}_r(v)} \frac{1}{|\mathbf{N}_r(v)|} \mathbf{\Phi}_r \cdot \mathbf{h}_u^{(l-1)}, \tag{4.3}$$

where $N_r(v)$ denotes the set of neighbor indices of node v with the relation $r \in A_t$. $\mathbf{\Phi}_r$ is a trainable relation-specific transformation for relation r in MR-GCN layer. Since the information in $(l-1)$-th layer can directly influence the representation of the node at l-th layer, MR-GCN uses another trainable transformation $\mathbf{\Phi}_0$ to account for the self-connection of each node using a special relation [144]. Here, we initialize each node embedding $\mathbf{h}_v^{(0)}$, $\forall v \in O_t$, by directly converting the node's type information to its corresponding one-hot vector.

Typically, the node embedding becomes more refined and global as the number of graph convolutional layers, L, increases. However, the authors in [181] also suggest that the features generated in earlier iterations might generalize the learning better. Therefore, we consider the node embeddings generated from all the MR-GCN layers. To be more specific, we calculate the embedding of node v at the final layer, denoted as $\mathbf{H}_v^L$, by concatenating the

features generated from all the MR-GCN layers, as follows,

$$\mathbf{H}_v^L = \mathbf{CONCAT}(\{\mathbf{h}_v^{(l)}\}|l = 0, 1, ..., L). \tag{4.4}$$

We denote the collection of node embeddings of *scene-graph* G_t after passing through L layers of MR-GCN as $\mathbf{X}_t^{prop}$ (L can be 1, 2 or 3).

The node embedding $\mathbf{X}_t^{prop}$ is further processed with an attention-based graph pooling layer. As stated in [99], such an attention-based pooling layer can improve the explainability of predictions and is typically considered a part of a unified computational block of a graph neural network (GNN) pipeline. In this layer, nodes are pooled according to the scores predicted from either a trainable simple linear projection [67] or a separate trainable GNN layer [108]. We denote the graph pooling layer that uses the **SCORE** function in [67] as *TopkPool* and the one that uses the **SCORE** function in [108] as *SAGPool*. The calculation of the overall process is presented as follows:

$$\alpha = \mathbf{SCORE}(\mathbf{X}_t^{prop}, \mathbf{A_t}), \tag{4.5}$$

$$\mathbf{P} = \text{top}_k(\alpha), \tag{4.6}$$

where α stands for the coefficients predicted by the graph pooling layer for nodes in G_t and $\mathbf{P}$ represents the indices of the pooled nodes which are selected from the top k of the nodes ranked according to α. The number k of the nodes to be pooled is calculated by a pre-defined pooling ratio, pr, and using $k = pr \times |O_t|$, where we consider only a constant fraction pr of the embeddings of the nodes of a scene-graph to be relevant (i.e., 0.25, 0.5, 0.75). We denote the node embeddings and edge adjacency information after pooling by $\mathbf{X}_t^{pool}$ and $\mathbf{A_t^{pool}}$ and

are calculated as follows:

$$\mathbf{X}_t^{pool} = (\mathbf{X}_t^{prop} \odot \tanh(\alpha))_{\mathbf{P}}, \tag{4.7}$$

$$\mathbf{A_t^{pool}} = \mathbf{A_t^{prop}}_{(\mathbf{P},\mathbf{P})}. \tag{4.8}$$

where $\odot$ represents an element-wise multiplication, $()_{\mathbf{P}}$ refers to the operation that extracts a subset of nodes based on P and $()_{(\mathbf{P},\mathbf{P})}$ refers to the formation of the adjacency matrix between the nodes in this subset. Finally, our model aggregates the node embeddings of the graph pooling layer, $\mathbf{X}_t^{pool}$, using a graph **READOUT** operation, to produce the final graph-level embedding $\mathbf{h}_{G_t}$ for each *scene-graph* G_t as given by

$$\mathbf{h}_{G_t} = \mathbf{READOUT}(\mathbf{X}_t^{pool}), \tag{4.9}$$

where the **READOUT** operation can be either summation, averaging, or selecting the maximum of each feature dimension, over all the node embeddings, known as *sum-pooling*, *mean-pooling*, or *max-pooling*, respectively. The process until this point is repeated across all images in $\mathbf{I}$ to produce the sequence of embedding, $\mathbf{h}_I$.

Temporal Modeling: The temporal model we propose uses an LSTM for converting the sequence of scene-graph embeddings $\mathbf{h}_I$ to the combined spatiotemporal embedding $\mathbf{Z}$. For each timestamp t, the LSTM updates the hidden state p_t and cell state c_t as follows,

$$p_t, c_t = \mathbf{LSTM}(\mathbf{h}_{G_t}, c_{t-1}), \tag{4.10}$$

where $\mathbf{h}_{G_t}$ is the final *scene-graph* embedding from timestamp t. After the LSTM processes all the scene-graph embeddings, a temporal readout operation is applied to the resultant output sequence to compute the final Spatio-temporal embedding Z given by

$$\mathbf{Z} = \mathbf{TEMPORAL\_READOUT}(p_1, p_2, ..., p_T) \tag{4.11}$$

where the **TEMPORAL_READOUT** operation could be extracting only the last hidden state p_T (LSTM-last), or be a temporal attention layer (LSTM-attn).

In [12], adding an attention layer b to the encoder-decoder-based LSTM architecture is shown to achieve better performance in Neural Machine Translation (NMT) tasks. For the same reason, we include *LSTM-attn* in our architecture. *LSTM-attn* calculates a context vector q using the hidden state sequence $\{p_1, p_2, ..., p_T\}$ returned from the LSTM encoder layer as given by

$$q = \sum_{t=1}^{T} \beta_t p_t \tag{4.12}$$

where the probability β_t reflects the importance of p_t in generating q. The probability β_t is computed by a *Softmax* output of an energy function vector e, whose component e_t is the energy corresponding to p_t. Thus, the probability β_t is formally given by

$$\beta_t = \frac{\exp(e_t)}{\sum_{k=1}^{T} \exp(e_k)}, \tag{4.13}$$

where the energy e_t associated with p_t is given by $e_t = b(s_0, p_t)$. The temporal attention layer b scores the importance of the hidden state p_t to the final output, which in our case is the risk assessment. The variable s_0 in the temporal attention layer b is computed from the last hidden representation p_T. The final Spatio-temporal embedding for a video clip, Z, is computed by feeding the context vector q to another LSTM decoder layer.

4.3.4 Risk Inference

To evaluate SG2VEC for the subjective risk assessment task, on top of spatial modeling and temporal modeling layers, we define the risk inference layer that computes the risk assessment prediction $\hat{Y}$ using the spatiotemporal embedding $\mathbf{Z}$. This component comprises

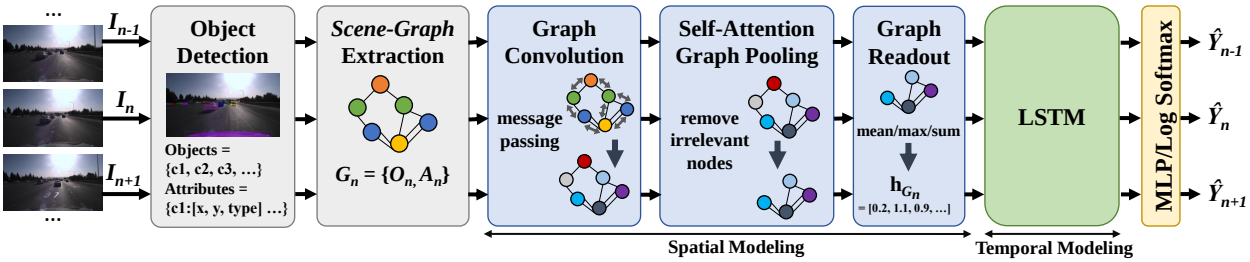

Figure 4.4: An illustration of SG2VEC's architecture.

an MLP layer followed by a *Softmax* activation function. Thus, the prediction $\hat{Y}$ is given by

$$\hat{Y} = Softmax(\mathbf{MLP}(Z)) \tag{4.14}$$

The loss for the prediction is calculated as follows,

$$\arg\min \mathbf{CrossEntropyLoss}(Y, \hat{Y}) \tag{4.15}$$

For training our model, we use a mini-batch gradient descent algorithm that updates its parameters by training on a batch of *scene-graph* sequences. To account for label imbalance, we apply class weighting when calculating loss. Besides, several dropout layers are inserted into the network to reduce overfitting.

4.4 Spatiotemporal Scene-Graph Embedding for collision prediction

4.4.1 Problem Formulation

In SG2VEC, we formulate the problem of collision prediction as a time-series classification problem where the goal is to predict if a collision will occur in the *near future*. Our goal is

to accurately model the spatiotemporal function f, where

$$\mathbf{Y_n} = f(\{I_1, ..., I_{n-1}, I_n\}), \mathbf{Y_n} \in \{0, 1\}, \text{for } n > 2, \tag{4.16}$$

where $\mathbf{Y_n} = 1$ implies a collision in the near future and $\mathbf{Y_n} = 0$ otherwise. Here the variable I_n denotes the image captured by the onboard camera at time n. The interval between each frame varies with the camera sampling rate.

SG2VEC consists of two parts (Figure 4.4) : (i) the *scene-graph* extraction, and (ii) collision prediction through spatiotemporal embedding.

4.4.2 Scene-Graph Extraction

The first step of our methodology is the extraction of *scene-graphs* for the images of a driving scene. The extraction pipeline forms the *scene-graph* for an image as in [183, 179] by first detecting the objects in the image and then identifying their relations based on their attributes. The difference from prior works lies in the construction of a *scene-graph* that is designed for higher-level AV decisions. We propose extracting a *minimal* set of relations such as directional relations and proximity relations. From our design space exploration, we found that adding many relation edges to the *scene-graph* adds noise and impacts convergence while using too few relation types reduces our model's expressivity. The best approach we found across applications involves constructing mostly ego-centric relations for a moderate range of relation types. Figure 4.5 shows an example of the graph extraction process.

We denote the extracted *scene-graph* for the frame I_n by $G_n = \{O_n, A_n\}$. Each *scene-graph* G_n is a directed, heterogeneous multi-graph, where O_n denotes the nodes and A_n is the adjacency matrix of the graph G_n. As shown in Fig. 4.5, nodes represent the identified objects such as lanes, roads, traffic signs, vehicles, pedestrians, etc., in a traffic scene. The adjacency

matrix A_n indicates the pair-wise relations between each object in O_n. The extraction pipeline first identifies the objects O_n by using Mask R-CNN [80]. Then, it generates an inverse perspective mapping (also known as a "birds-eye view" projection) of the image to estimate the locations of objects relative to the ego car, which are used to construct the pair-wise relations between objects in A_n. For each camera angle, we calibrate the birds-eye view projection settings using known fixed distances, such as the lane length and width, as defined by the highway code. This enables us to estimate longitudinal and lateral distances accurately in the projection. For datasets captured by a single vehicle, this step only needs to be performed once. However, for datasets with a wide range of camera angles such as the *620-dash* dataset introduced later in the paper, this process needs to be performed once per vehicle. With a human operator, we found that this calibration step takes approximately 1 minute per camera angle on average.

The extraction pipeline identifies three kinds of pair-wise relations: *proximity* relations (e.g. *visible*, *near*, *very_near*, etc.), *directional* (e.g. *Front_Left*, *Rear_Right*, etc.) relations, and *belonging* (e.g. car_1 *isIn* left_lane) relations. Two objects are assigned the *proximity* relation, $r \in \{$*Near_Collision* (4 ft.), *Super_Near* (7 ft.), *Very_Near* (10 ft.), *Near* (16 ft.), *Visible* (25 ft.)$\}$ provided the objects are physically separated by a distance that is within that relation's threshold. The *directional relation*, $r \in \{$*Front_Left*, *Left_Front*, *Left_Rear*, *Rear_Left*, *Rear_Right*, *Right_Rear*, *Right_Front*, *Front_Right*$\}$, is assigned to a pair of objects, in this case between the ego-car and another car in the view, based on their relative orientation and only if they are within the *Near* threshold distance from one another. Additionally, the *isIn* relation identifies which vehicles are on which lanes (see Fig. 4.5). We use each vehicle's horizontal displacement relative to the ego vehicle to assign vehicles to either the *Left Lane*, *Middle Lane*, or *Right Lane* using the known lane width. Our abstraction only considers three-lane areas, and, as such, we map vehicles in all left lanes and all right lanes to the same *Left Lane* node *Right Lane* node respectively. If a vehicle overlaps two lanes (i.e., during a lane change), it is mapped to both lanes.

4.4.3 Early Collision Prediction

As shown in Figure 4.4, in our collision prediction methodology, each image I_n is first converted into a *scene-graph* $G_n = \{O_n, A_n\}$. Each node $v \in O_n$ is initialized by a one-hot vector (*embedding*), denoted by $\mathbf{h}_v^{(0)}$. Then, the MR-GCN [144] layers are used to update these embeddings via the edges in A_n. Specifically, the l-th MR-GCN layer computes the node embedding for each node v, denoted as $\mathbf{h}_v^{(l)}$, as follows:

$$\mathbf{h}_v^{(l)} = \mathbf{\Phi}_0 \cdot \mathbf{h}_v^{(l-1)} + \sum_{r \in \mathbf{A_n}} \sum_{u \in \mathbf{N}_r(v)} \frac{1}{|\mathbf{N}_r(v)|} \mathbf{\Phi}_r \cdot \mathbf{h}_u^{(l-1)}, \tag{4.17}$$

where $N_r(v)$ denotes the set of neighbors of node v with respect to the relation $r \in A_n$, $\mathbf{\Phi}_r$ is a trainable relation-specific transformation for relation r, and $\mathbf{\Phi}_0$ is the self-connection for each node v that accounts for the influence of $\mathbf{h}_v^{(l-1)}$ on $\mathbf{h}_v^{(l)}$ [144]. After the input is passed through multiple MR-GCN layers, the set of node embeddings output by each layer is collected and concatenated along the feature dimension to produce the final embedding for each node v, denoted by $\mathbf{H}_v^L = \mathbf{CONCAT}(\{\mathbf{h}_v^{(l)}\}|l = 0, 1, ..., L)$, where L is the index of the last layer. Thus, if the model uses two MR-GCN layers with output size 64, the final embedding for each node will contain 128 features.

The final embeddings for *scene-graph* G_n, denoted by $\mathbf{X}_n^{prop}$, are then passed through a graph pooling layer to filter out irrelevant nodes from the graph, creating the pooled set of node embeddings $\mathbf{X}_n^{pool}$ and their edges $\mathbf{A}_n^{pool}$. The pooling layer is implemented as follows:

$$\alpha = \mathbf{SCORE}(\mathbf{X}_n^{prop}, \mathbf{A}_n^{prop}), \tag{4.18}$$

$$\mathbf{P} = \text{top}_k(\alpha), \tag{4.19}$$

where **SCORE** can either be implemented as a top-k pooling (*Top-K*) [67] or self-attention graph pooling function (*SAGPool*) [108], α contains the score of each node in G_n, and $\mathbf{P}$ is

the set of k highest scoring nodes in G_n. After pooling, the node embeddings and adjacency matrix are denoted as $\mathbf{X}_t^{pool}$ and $\mathbf{A_t^{pool}}$ computed as follows:

$$\mathbf{X}_t^{pool} = (\mathbf{X}_t^{prop} \odot \tanh(\alpha))_{\mathbf{P}}, \tag{4.20}$$

$$\mathbf{A_t^{pool}} = \mathbf{A_t^{prop}}_{(\mathbf{P},\mathbf{P})} \tag{4.21}$$

where $\odot$ represents element-wise multiplication, $()_{\mathbf{P}}$ refers to the operation that selects only the subset of nodes defined by P and $()_{(\mathbf{P},\mathbf{P})}$ refers to the formation of the adjacency matrix between the nodes in this subset. Then, for each *scene-graph* G_n, the corresponding $\mathbf{X}_n^{pool}$ is passed through the graph **READOUT** operation that condenses the node embeddings to a single graph embedding $\mathbf{h}_{G_n}$ as follows:

$$\mathbf{h}_{G_n} = \mathbf{READOUT}(X_t^{pool}) \tag{4.22}$$

where **READOUT** can be an operation such as averaging (*mean-readout*), summation (*add-readout*), or retrieving the maximum (*max-readout*) in each feature dimension for the set of pooled node embeddings X_t^{pool}.

Then, this spatial embedding $\mathbf{h}_{G_n}$ is passed to the temporal model (LSTM) to generate a spatiotemporal embedding z_n as follows:

$$z_n, s_n = \mathbf{LSTM}(h_{G_n}, s_{n-1}) \tag{4.23}$$

Where s_{n-1} represents the hidden state of the LSTM after the previous time step. For each timestamp n, the LSTM produces an output embedding z_n and updates its hidden state s_n. Since the hidden state is carried over to the next time step $n+1$ and used to compute z_{n+1}, it enables the LSTM to model how the spatial embeddings h_{G_n} change over time.

Lastly, each spatiotemporal embedding z_n is then passed through a Multi-Layer Perceptron

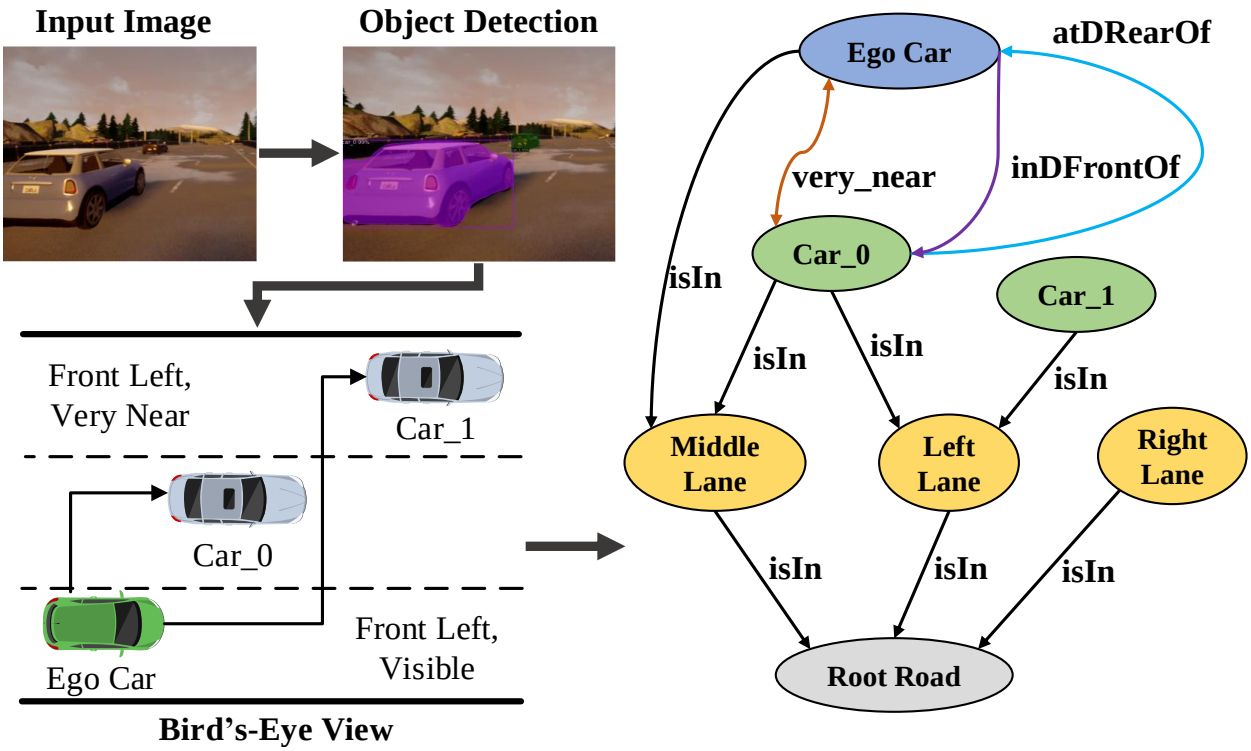

Figure 4.5: An illustration of our *scene-graph* extraction process.

(MLP) that outputs each class's confidence value. The two outputs of the MLP are compared, and $\hat{Y}_n$ is set to the index of the class with the greater confidence value (0 for no-collision or 1 for collision). During training, we calculate the cross-entropy loss between each set of non-binarized outputs $\hat{Y}_n$ and the corresponding labels for backpropagation.

4.5 Experimental Results for Risk Assessment

For this work, we provide extensive experimental results to illustrate the accuracy of our model and its ability to transfer knowledge (transferability) for a specific driving maneuver: lane changes. This task by itself is crucial, given that 7.62% of all traffic accidents between light vehicles can be attributed to improper execution of lane changes [117]. Besides, we also evaluate our model's capability for turning and entering or leaving the traffic flow of a road (merging and branching, respectively). We evaluate our approach by comparing our model

and a state-of-the-art SMT+CNN+LSTM-based risk assessment model [195]. We refer to this model as the *baseline*. Besides, we provide results for our model's best hyper-parameter setting and perform an ablation study to evaluate the contribution of each major component in our model.

4.5.1 Experimental Preparation

We prepared two types of datasets for our experiments (i) synthesized lane-changing datasets and (ii) real-world driving datasets. To create the synthesized datasets, we developed a tool to generate lane-changing clips using the Carla[1] and Carla Scenario Runner[2]. We generated the real-world dataset by extracting lane change clips and other driving actions from the Honda Driving Dataset (HDD) [134].

Carla is an open-source driving simulator [54] that allows users to control a vehicle in either manual or autopilot mode. The Carla Scenario Runner contains a set of atomic controllers that enable users to control a car in a driving scene and perform complex driving maneuvers. We modified the user script in Carla so that it can (i) select one autonomous car randomly and switch its mode to manual mode and then (ii) utilize Scenario Runner's function to force the vehicle to change lanes.

The data-generating tool allows us to fabricate lane-changing clips directly instead of extracting them from long-driving clips. We generated a wide range of simulated lane changes using the various presets in Carla that allowed us to specify the number of cars, pedestrians, weather and lighting conditions, etc. Also, through the APIs provided by the Traffic Manager (TM) of the Carla simulator, we were able to customize the driving characteristics of every autonomous vehicle, such as the intended speed considering the current speed limit, the chance of ignoring the traffic lights, or the chance of neglecting collisions with other

[1] `https://github.com/carla-simulator/carla`
[2] `https://github.com/carla-simulator/scenario_runner`

vehicles. This allowed us to simulate a wide range of realistic urban driving environments and generate synthesized datasets suitable for training and testing a model.

We generated two synthesized lane-changing datasets: a *271-syn* dataset and a *1043-syn* dataset, containing 271 and 1,043 lane-changing clips, respectively. In addition, we subsampled the *271-syn* and *1043-syn* datasets further to create two balanced datasets that have a 1:1 distribution of risky to safe lane changes: *96-syn* and *306-syn*. For real driving datasets, we processed the HDD dataset to create a dataset called *1361-honda*. Specifically, *1361-honda* contains 571 lane changing, 350 turning, 297 branching, and 149 merging video clips. For evaluating the capability of the model to transfer knowledge after training on *271-syn* and *1043-syn* datasets, we subsampled *1361-honda* to create a real-driving lane-changing dataset which contains 571 lane-changing clips, denoted as the *571-honda* dataset.

To label the video clips in both the real-world and synthesized datasets, we performed an annotation process similar to the one used in [195]. The process starts with multiple human annotators assigning a risk score to each clip that ranges from -2 to 2, where 2 implies a highly risky driving scenario and -2 implies the safest driving scenario. Then for each video clip, the risk labels of all the annotators are averaged and converted to a binary label y as follows: if the average is ≤ 0, then the label $y = 0$ (safe) is assigned, else label $y = 1$ (risky) is assigned.

In our work, we used five different anonymous annotators. After the annotation process, the annotators were required to write down the criteria and rationale they used when annotating the video clips. This process ensured that the annotators paid attention while labeling, reducing the odds of trivial misjudgment of a driving scene's risk level. The risk factors common among the five annotators were the distance to other cars and the side curbs, the speed relative to other vehicles, the sizes of adjacent vehicles, the presence of bikers or pedestrians, and the traffic light status. Besides, the sudden or random appearance of objects in the scene was also a critical factor in determining a driving scene's risk level.

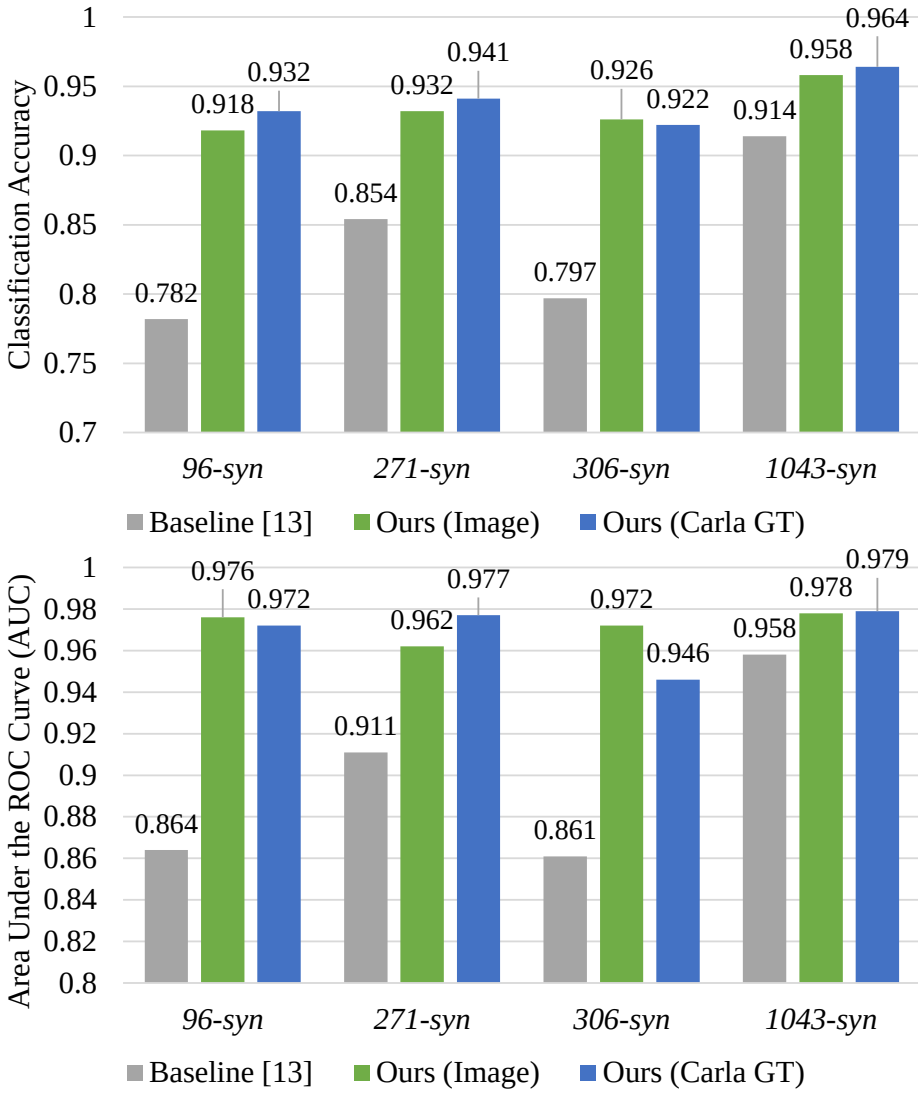

Figure 4.6: Accuracy and AUC comparison between our approaches (Real Image and Carla GT) and [195] on different datasets.

We randomly split each dataset into a training set and a testing set by the ratio 7:3 such that the split is stratified, i.e., the proportion of risky to safe lane change clips in each of the splits is the same. The models are first trained on the training set before being evaluated on the testing set. The final score of a model on a dataset is computed by averaging over the testing set scores for ten different stratified train-test splits of the dataset.

Our models were implemented using *PyTorch* and *PyTorch-Geometric* [123, 62]. We used the ADAM optimizer for the training algorithm. We considered three learning rates: {0.0005,

0.0001, 0.00005}, and a weight decaying rate of 5×10^{-4} per epoch. We used a batch size of 16 sequences for each training epoch. In our experiments, we trained each model for 200 epochs. Regarding the setting of hyper-parameters, we considered the options described in Section 4.3.3. From our experimentation, we found that the best option for the hyper-parameters of our model is a mini-batch size of 16 sequences, a learning rate of 0.00005, two MR-GCN layers with 100 hidden units, a SAGPool pooling layer with a ratio of 0.5, *sum-pooling* for graph readout operation and *LSTM-attn* for temporal modeling.

To ensure a fair comparison between our model and the baseline, we reported the performance of the model configurations with the lowest validation loss throughout the training in the results section. All the experiments were conducted on a server with one NVIDIA TITAN-XP graphics card and one NVIDIA GeForce GTX 1080 graphics card. For implementing the baseline model [195], we used the source code available at their open-source repository[3]. The source code and *scene-graph* datasets used in our experiments are open-sourced at `https://github.com/louisccc/sg-risk-assessment`.

4.5.2 The Evaluation of Subjective Risk Assessment

We evaluate each model's performance by measuring its classification accuracy and the Area Under the Curve (AUC) of the Receiver Operating Characteristic (ROC) for each dataset. The classification accuracy is the ratio of the number of correct predictions on the test set of a dataset to the total number of samples in the testing set. AUC sometimes called a balanced accuracy measure [150], measures the probability that a binary classifier ranks a positive sample more highly than a random negative sample. This is a more balanced measure for measuring accuracy, especially with imbalanced datasets (i.e., *271-syn*, *1043-syn*, *571-honda*).

[3] `https://github.com/Ekim-Yurtsever/DeepTL-Lane-Change-Classification`

Figure 4.6 compares our model's performance and the baseline model [195] for all the synthetic datasets. The results show that our approach consistently outperforms [195] across all the datasets in terms of both classification accuracy and AUC. Particularly, on the *1043-syn* dataset, our Image-based and GT pipelines outperform [195] in classification accuracy by 4.4% and 5% respectively (i.e., the accuracy of 95.8% and 96.4% compared to 91.4% for the baseline).

We found that the performance difference between our approach and the baseline increased when the training datasets were smaller. Figure 4.6 shows that the difference in the accuracy between our approach using the GT pipeline and the baseline [195] is 5% for the *1043-syn* dataset and 8.7% for the *271-syn* dataset. This result indicates that our approach can learn an accurate model even from a smaller dataset. We postulate this is a direct result of its use of a *scene-graph* based IR.

We also found that our approach performs better than the baseline on balanced datasets. Among the datasets used for evaluation of the models, the datasets *271-syn* and *306-syn* contain roughly the same number of clips but different distribution of safe to risky lane changes (2.30:1 for *271-syn* vs. 1:1 for *306-syn*). We found that the performance difference between our image-based approach and the baseline on these datasets is 12.9% on the *306-syn* dataset compared to 7.8% on the *271-syn* dataset, indicating that our approach can discriminate between the two classes better than the baseline.

We also evaluated the contribution of each functional component in our proposed model by conducting an ablation study. The results of the study are shown in Table 4.1. From Table 4.1 we find that the simplest of the models, with no MR-GCN layer (replaced with an MLP layer) and a simple average of the embeddings in $\mathbf{h}_I$ for the temporal model (denoted as *mean* in Table 4.1), achieves a classification accuracy of 75%. Replacing *mean* with an LSTM layer for temporal modeling yields a 10.5% increase in performance. We also find that including a single MR-GCN with 64 hidden units and *sum-pooling* to the simplest model

	Spatial Modeling	Temporal Modeling	Avr. Acc.	Avr. AUC
Ablation Study	No MR-GCN	*mean*	0.762	0.823
	No MR-GCN	*LSTM-last*	0.867	0.929
	1 MR-GCN	*mean*	0.910	0.960
	1 MR-GCN	*LSTM-last*	**0.943**	**0.977**
Temporal Attention	No MR-GCN	*LSTM-last*	0.867	0.929
	No MR-GCN	*LSTM-attn*	0.868	0.928
	1 MR-GCN	*LSTM-last*	0.943	0.977
	1 MR-GCN	*LSTM-attn*	**0.950**	**0.977**
Spatial Attention	1 MR-GCN	*mean*	0.910	0.960
	1 MR-GCN, *TopkPool*	*mean*	0.886	0.930
	1 MR-GCN, *SAGPool*	*mean*	**0.937**	**0.968**

Table 4.1: The results of the Carla GT approach on *1043-syn* dataset with various spatial and temporal modeling settings. In these experiments, we used MR-GCN layers with 64 hidden units and *sum-pooling* as the graph readout operation.

results in a 14.8% performance gain over the simplest model. The performance gain achieved by just including the MR-GCN layer suggests the effectiveness of considering the relations between objects. Finally, we find that the model with one MR-GCN with 64 hidden units and *sum-pooling* plus the LSTM layer for temporal modeling yields the maximum gain of 18.1% over the simplest model. These results demonstrate the importance of each component in the model we propose.

4.5.3 The Impact of Attention Mechanisms on Risk Assessment

Here, we evaluate the various attention components of our proposed model. To evaluate the benefit of attention over the spatial domain, we tested our model with three different graph attention methods: no attention, *SAGPool*, and *TopkPool*. To evaluate the impact of attention on the temporal domain, we tested our model with the following temporal models: *mean*, *LSTM-last*, and *LSTM-attn*. The detailed results that elucidate the effectiveness of these different attention mechanisms are presented in Table 4.1.

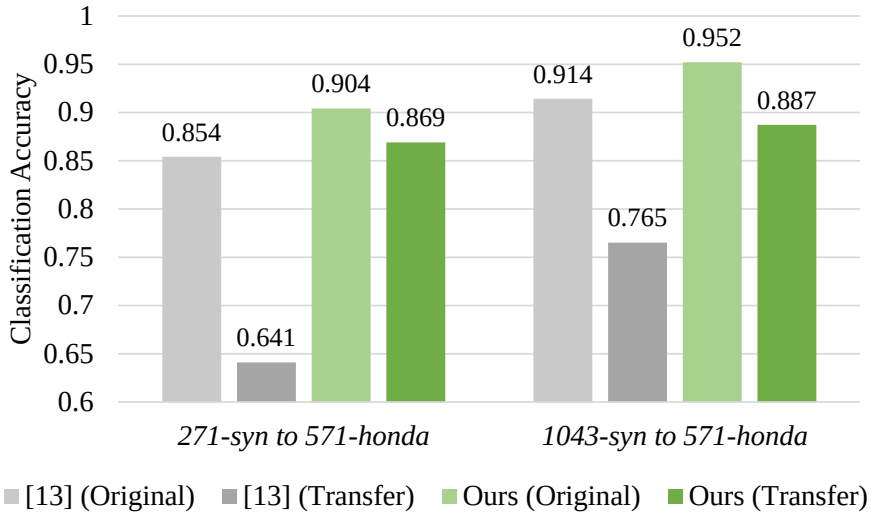

Figure 4.7: The results of comparing transferability between our Real Image model and [195]. In this experiment, we trained our model using our best hyper-parameters on both *271-syn* dataset and *1043-syn* dataset. Then we tested the accuracy of our approach on both the original dataset and *571-honda* dataset. We followed the same procedure to train and test [195].

For evaluating the benefits of graph attention, we start with an attention-free model: one MR-GCN layer with *sum-pooling* + *mean*. In comparison, the model that uses *SAGPool* for attention on the graph shows a 2.7% performance gain over the attention-free model. This result indicates that attention over both nodes and relations allows *SAGPool* to filter out better irrelevant nodes from each *scene-graph*. We found that the model using *TopkPool* as the graph-attention layer became relatively unstable, resulting in a 2.4% performance drop compared to the attention-free model. This is likely because *TopkPool* ignores the relations between nodes when calculating α. Another reason for this instability could be the random initialization of weights in *TopkPool*, which can exponentially affect the overall performance as stated in [99].

For evaluating the impact of attention on the temporal model, we evaluated the effects of adding a temporal attention layer to the following two models: (i) with no MR-GCN layers and no temporal attention and (ii) with one MR-GCN layer and no temporal attention. Compared to the model with no MR-GCN layer and no temporal attention, the model's

performance with no MR-GCN and *LSTM-attn* was found to be 0.1% higher. We also found that adding *LSTM-attn* to the model with one MR-GCN layer increases its performance by 0.7% over the same model with no temporal attention. These results demonstrate that the inclusion of temporal attention does improve performance, though only marginally. The reason why we only see a marginal improvement can be that the temporal attention layer is less relevant to the dataset that our model was trained on. When preparing these datasets, we manually removed the frames irrelevant to a lane change, exactly the set of frames that temporal attention would have given less attention to, thus minimizing its effect.

Figure 4.8 demonstrates our model's capability to pinpoint the critical factors in assessing driving risk in both temporal and spatial domains. As Equation 4.7 shows, the node attention weights α are used by our graph pooling layer to filter the objects in a scene-graph that are less significant for assessing the risk. In the temporal domain, the attention weights, β, allow the LSTM encoder to score each intermediate hidden state (p_t) and retain only the useful information in Z for the final risk assessment. Table 1 demonstrates the benefit of applying attention layers in risk assessment. Specifically, it shows that the addition of the *SAGPool* attention layer improves the accuracy of the baseline model (1 MR-GCN + mean) by 2.7% and that including *LSTM-attn* to the baseline model (1 MR-GCN + *LSTM-last*) increases the performance by 0.3%.

In addition to improving our model's performance, including graph and temporal attention improves the explainability of our model's risk assessment decisions. We demonstrate this capability using the visualization of both graph and temporal attention provided in Figure 4.8. Figure 4.8 shows the trend of the attention scores $\beta_1, \beta_2, ...\beta_T$ for a risky lane changing clip. Intuitively, the frame with a higher attention score α_t contributes more to the context vector c (shown in Equation 4.12), thus playing a more critical role in calculating h_{G_t} and contributing to the final risk assessment decision. In this risky lane-changing example, the temporal attention scores progressively increase between frames 19 and 32 during the lane

change; and the highest frame attention weights appear in frames 33 and 34, which are the frames immediately before the collision occurs. Figure 4.8 also shows the projection scores for the node attention layer, where a higher score for a node indicates that it contributes more to the final risk assessment prediction. As shown in this example, as the ego car approaches the yellow vehicle, the node attention weights for the ego car and the yellow vehicle are increased proportionally to the scene's overall risk. In the first few frames, the risk of collision is low; thus, the node attention weights are low; however, in the last few frames, a collision between these two vehicles is imminent; thus, the attention weights for the two cars are much higher than for any other nodes in the graph. This example demonstrates our model's capability to pinpoint the critical factors in a *scene-graph* that contributed to its risk assessment decision. This capability can be valuable for debugging edge cases at design time, thus reducing the chances of ADS making unexpected, erroneous decisions in real-world scenarios and improving human trust in the system.

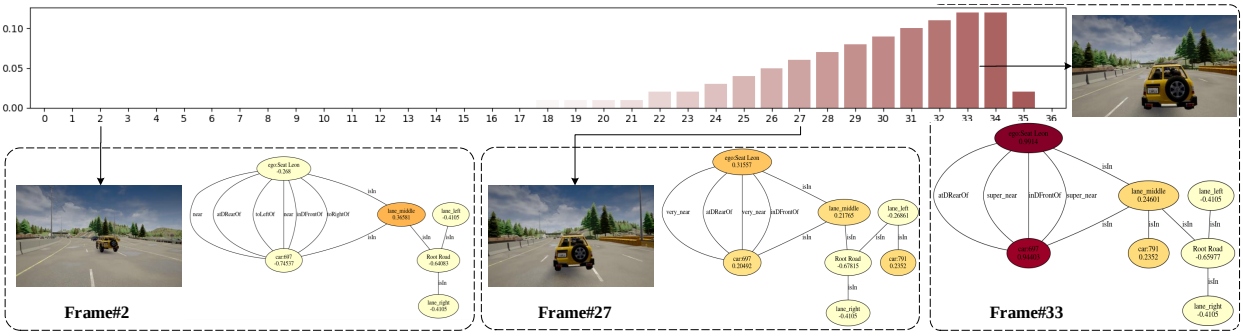

Figure 4.8: The visualization of attention weights in both spatial (α) and temporal (β) domains using a risky lane-changing clip as an example. We used a gradient color from light yellow to red for visualizing each node's projection score that indicates its importance in calculating a *scene-graph* embedding. We also used a gradient colored (white to red) bar chart to visualize the temporal attention coefficients $\beta_1, \beta_2, ...\beta_{36}$ used for calculating the context vector c.

4.5.4 Evaluating Transferability: From Virtual To Real-Driving

Here, we demonstrate our approach's capability to effectively transfer the knowledge learned from a simulated dataset to a real-world dataset. To demonstrate this capability, we use the model weights and parameters learned from training on the *271-syn* dataset or the *1043-syn* dataset directly for testing on the real-world driving dataset: *571-honda*. We also compare the transferability of our model with that of the baseline method [195]. The results are shown in Figure 4.7.

As expected, the performance of both our approach and the baseline degrades when tested on *571-honda* dataset. However, as Figure 4.7 shows, the accuracy of our approach only drops by 6.7% and 3.5% when the model is trained on *271-syn* and *1043-syn*, respectively, while the baseline's performance drops drastically by a much higher 21.3% and 14.9%, respectively. The results categorically show that our proposed model can transfer knowledge more effectively than the baseline.

4.5.5 Evaluating Risk Assessment for Other Driving Maneuvers

We show results from evaluating our model's performance on various other driving scenarios available in the HDD: turning, branching, merging, etc. The results for training and evaluating our model on the *1361-honda* dataset are shown in Table 4.2. From Table 4.2, we can observe that our approach significantly outperforms [195] in both overall accuracy (0.86 v.s. 0.58) and overall AUC (0.91 v.s. 0.61), indicating that our approach can better assess risk across various driving scenarios. In Table 4.2 we also show the performance for each action type. The results show that our approach also outperforms [195] on each type of driving scenario. Our approach slightly underperforms on turning scenarios compared to its performance on other action types. This discrepancy is likely because turning scenarios are intrinsically more complicated than straight-road driving scenarios (lane change, branch,

merge). Another reason could be that the heading of vehicles is a contributing factor in complicated scenarios, while the *scene-graph* used in our work contains only distance and directional relations.

Metric	Action Type	**Ours**	[195]
Accuracy	Overall	**0.8655**	0.5844
	Lane Changing	**0.8710**	0.5714
	Merging	**0.8462**	0.5854
	Branching	**0.9101**	0.5556
	Turning	**0.8211**	0.6218
AUC	Overall	**0.9124**	0.6078
	Lane Changing	**0.9105**	0.5877
	Merging	**0.9395**	0.6526
	Branching	**0.9462**	0.5807
	Turning	**0.8645**	0.6400

Table 4.2: The breakdown of risk assessment performance by driver action types (Lane-Changing, Merging, Branching, and Turning) evaluated on *1361-Honda* dataset.

4.6 Experimental Results for Early Collision Prediction

4.6.1 Experimental Preparation

This section provides extensive experimental results to demonstrate SG2VEC's performance, efficiency, and transferability compared to the state-of-the-art collision prediction model, DPM [155]. For SG2VEC, we used 2 MR-GCN layers, each of size 64, one *SAGPooling* layer with a pooling ratio of 0.25, one *add-readout* layer, one LSTM layer with hidden size 20, one MLP layer with an output of size 2, and a LogSoftmax to generate the final confidence value for each class. For the DPM, we followed the architecture used in [155], which uses one 64x64x5 Convolutional LSTM (ConvLSTM) layer, one 32x32x5 ConvLSTM layer, one 16x16x5 ConvLSTM layer, one MLP layer with output size 64, one MLP layer with output

size 2, and a Softmax to generate the final confidence value. For both models, we used a dropout of 0.1 and ReLU activation. The learning rates were 0.00005 for SG2VEC and 0.0001 for DPM. We ran the experiments shown in Sections 4.6.2 and 4.6.3 on a Windows PC with an AMD Ryzen Threadripper 1950X processor, 16 GB RAM, and an Nvidia GeForce RTX 2080 Super GPU.

Figure 4.9: Examples of driving scenes from our a) synthetic datasets, b) typical real-world dataset, and c) complex real-world dataset. In a), all driving scenes occur on highways with the same camera position and clearly defined road markings; lighting and weather are dynamically simulated in CARLA. In b) driving scenes occur on multiple types of clearly marked roads but lighting, camera angle, and weather are consistent across scenes. c) contains a much broader range of camera angles as well as more diverse weather and lighting conditions, including rain, snow, and night-time driving; it also contains a large number of clips on unpaved or unmarked roadways, as shown.

Dataset Preparation: We prepared three types of datasets for our experiments: (i) synthesized datasets, (ii) a typical real-world driving dataset, and (iii) a complex real-world driving dataset. Examples from each dataset are shown in Figure 4.9. Our synthetic datasets focus on the highway lane change scenario as it is a common AV task. To evaluate the transferability of each model from synthetic datasets to real-world driving, we prepared a typical real-world dataset containing lane-change driving clips. Finally, we prepared the complex real-world driving dataset to evaluate each model's performance on a challenging dataset containing a broad spectrum of collision types, road conditions, and vehicle maneuvers. All datasets were collected at a 1280x720 resolution, and each clip spans 1-5 seconds.

To synthesize the datasets, we developed a tool using CARLA [54], an open-source driving simulator, and CARLA Scenario Runner[4] to generate lane change video clips with/without collisions. We generated a wide range of simulated lane changes with different numbers of cars, pedestrians, weather and lighting conditions, etc. We also customized each vehicle's driving behavior, such as their intended speed, probability of ignoring traffic lights, or the chance of avoiding collisions with other vehicles. We generated two synthetic datasets: a *271-syn* dataset and a *1043-syn* dataset, containing 271 and 1,043 video clips, respectively. These datasets have no-collision:collision label distributions of 6.12:1 and 7.91:1, respectively. In addition, we sub-sampled the *1043-syn* dataset to create *306-syn*: a balanced dataset that has a 1:1 distribution. Our synthetic *scene-graph* datasets[5] and our source code[6] are open-source and available online.

As for the typical real-world driving dataset, we denoted it as *571-honda* which is a subset of the *Honda Driving Dataset* (HDD) [135] containing 571 lane-change video clips from real-world driving with a distribution of 7.21:1. The HDD was recorded on the same vehicle during mostly safe driving in the California Bay Area.

Our complex real-world driving dataset, denoted as *620-dash*, contains very challenging real-world collision scenarios drawn from the Detection of Traffic Anomaly dataset [184]. This dataset contains a wide range of drivers, car models, driving maneuvers, weather/road conditions, and collision types, as recorded by on-board dashboard cameras. Since the original dataset contains only collision clips, we prepared *620-dash* by splitting each clip in the original dataset into two parts: (i) the beginning of the clip until 1 second before the collision, and (ii) from 1 second before the collision until the end of the collision. We then labeled part (i) as 'no-collision' and part (ii) as 'collision.' The *620-dash* dataset contains 315 collision video clips and 342 non-collision driving clips.

[4] `https://github.com/carla-simulator/scenario_runner`
[5] `https://dx.doi.org/10.21227/c0z9-1p30`
[6] `https://github.com/AICPS/sg-collision-prediction`

Labeling and Pre-Processing: We labeled the synthetic datasets and the *571-honda* dataset using human annotators. The final label assigned to a clip is the average of the labels assigned by the human annotators rounded to 0 (no collision) and 1 (collision/near collision). Each frame in a video clip is given a label identical to the entire clip's label to train the model to identify the preconditions of a future collision.

For SG2VEC, all the datasets were pre-processed using the *scene-graph* extraction pipeline to construct the *scene-graphs* for each video clip. For a given sequence, SG2VEC can leverage the full history of prior frames for each new prediction. For the DPM, the datasets were pre-processed to match the input format used in its original implementation [155]. Thus, the DPM uses 64x64 grayscale versions of the clips in the datasets turned into sets of subsequences J_n for a clip of length l defined as follows.

$$J_n = \{I_n, I_{n+1}, I_{n+2}, I_{n+3}, I_{n+4}\}, \text{for } n \in [1, l-4] \tag{4.24}$$

Since DPM only uses five prior frames to make each prediction, we also present results for SG2VEC using the same length of history, denoted as SG2VEC (5-frames) in the results.

4.6.2 The evaluation of SG2VEC on Collision Prediction

We evaluated SG2VEC and the DPM using classification accuracy, the area under the ROC curve (AUC) [26], and Matthews Correlation Coefficient (MCC) [45]. MCC is considered a balanced measure of performance for binary classification even on datasets with significant class imbalances. The MCC score outputs a value between -1.0 and 1.0, where 1.0 corresponds to a perfect classifier, 0.0 to a random classifier, and -1.0 to an always incorrect classifier. Although class re-weighting helps compensate for the dataset imbalance during training, classification accuracy is typically less reliable for imbalanced datasets, so the primary metric we use to compare the models is MCC. We used stratified 5-fold cross-validation

Dataset	Model	Accuracy	AUC	MCC
271-syn	SG2VEC (5-frames)	**0.8979**	**0.9541**	**0.5362**
271-syn	SG2VEC	0.8812	0.9457	0.5145
271-syn	DPM	0.8733	0.8939	0.2160
306-syn	SG2VEC (5-frames)	0.7946	0.8653	0.5790
306-syn	SG2VEC	**0.8372**	**0.9091**	**0.6812**
306-syn	DPM	0.6846	0.6881	0.3677
1043-syn	SG2VEC (5-frames)	**0.9142**	**0.9623**	0.5323
1043-syn	SG2VEC	0.9095	0.9477	**0.5385**
1043-syn	DPM	0.8834	0.9175	0.2912
620-dash	SG2VEC (5-frames)	0.6534	0.7113	0.3053
620-dash	SG2VEC	**0.7007**	**0.7857**	**0.4017**
620-dash	DPM	0.4890	0.4717	-0.0366

Table 4.3: Classification accuracy, AUC, and MCC for SG2VEC (Ours) and DPM.

to produce the final results shown in Table 4.3 and Figure 4.10.

The performance of SG2VEC and the DPM on our synthetic datasets is shown in Table 4.3. We find that our SG2VEC achieves higher accuracy, AUC, and MCC on every dataset, even when only using five prior frames as input. In addition to predicting collisions more accurately, SG2VEC also infers **5.5x** faster than the DPM on average. We attribute this to the differences in model complexity between our SG2VEC architecture and the much larger DPM model. Interestingly, SG2VEC (5-frames) achieves slightly better accuracy and AUC than SG2VEC on the imbalanced datasets and slightly lower overall performance on the balanced datasets. This is likely because a large number of safe lane changes in the imbalanced datasets adds noise during training and makes the full-history version of the model perform slightly worse. However, the full model can learn long-tail patterns for collision scenarios and performs better on balanced datasets.

The DPM achieves relatively high accuracy and AUC on the imbalanced *271-syn* and *1043-syn* datasets but suffers significantly on the balanced *306-syn* dataset. This drop indicates that the DPM could not identify the minority class (collision) well and tended to over-predict the majority class (no-collision). In terms of MCC, the DPM scores higher on the *306-syn*

Exp.	Spatial Model	Graph Pooling	Temporal Model	Acc.	MCC
Ablation Study	MLP	none	none	0.7605	0.2612
	MLP	none	LSTM	0.7660	0.2874
	MRGCN	none	none	0.8605	0.4792
	MRGCN	none	LSTM	**0.8931**	**0.5561**
Graph Attn. and Pooling	MRGCN	Top-K	none	0.8288	0.3458
	MRGCN	SAGPool	none	0.8738	0.5032
	MRGCN	Top-K	LSTM	**0.9014**	**0.5565**
	MRGCN	SAGPool	LSTM	**0.9076**	**0.5407**

Table 4.4: SG2VEC ablation study on the *1043-syn* dataset.

dataset than it scores on the other datasets. This result is because the *306-syn* dataset has a balanced class distribution compared to the other datasets, which could enable the DPM to improve its prediction accuracy on the collision class.

In contrast, the SG2VEC methodology performs well on both balanced and imbalanced synthetic datasets with an average MCC of **0.5860**, an average accuracy of **87.97%**, and an average AUC of **0.9369**. Since MCC is scaled from -1.0 to 1.0, SG2VEC achieves a **14.72%** higher average MCC score than the DPM model.

The results from our SG2VEC ablation study are shown in Table 4.4 and support our hypothesis that spatial modeling with MRGCN and temporal modeling with LSTM are core to SG2VEC's collision prediction performance. However, the MRGCN appears to be slightly more critical to performance than the LSTM. Interestingly the choice of pooling layer (no pooling, Top-K pooling, or SAG Pooling) does not seem to significantly affect performance at this task as long as LSTM is used; when no LSTM is used SAG Pooling presents a clear performance improvement.

The performance of both the models significantly drops on the highly complex real-world *620-dash* dataset due to the variations in the driving scenes and collision scenarios. This drop is to be expected as this dataset contains a wide range of driving actions, road environments, and collision scenarios, increasing the difficulty of the problem significantly. We took several

steps to try and address this performance drop. First, we improved the birds-eye view (BEV) calibration on this dataset in comparison to the other datasets. Since the varying camera angles and road conditions in this dataset impact our ability to properly calibrate SG2VEC's BEV projection in a single step, we created custom BEV calibrations for each clip in the dataset, which improved performance somewhat. However, as shown in Figure 4c, a significant part of the dataset consists of driving clips on roads without any discernible lane markings, such as snowy, unpaved, or unmarked roadways. These factors make it challenging to correlate known fixed distances (i.e., the width and length of lane markings) with the projections of these clips. To further improve performance on this particular dataset, we performed extensive architecture and hyperparameter tuning. We found that, with one MRGCN layer of size 64, one LSTM layer with the hidden size 100, no SAGPooling layer, and a high learning rate and batch size, we achieved significantly better performance than the model architecture discussed at the beginning of Section 4.6.1 (2 MRGCN layers of size 64, one LSTM layer with hidden size 20, and a SAGPooling layer with a keeping ratio of 0.5). We believe this indicates that the temporal features of each clip in this dataset are more closely related to collision likelihood than the spatial features in each clip. As a result, the additional spatial modeling components were likely causing overfitting and skewing the spatial embedding output. The spatial embeddings remained more general with a simpler spatial model (1 MRGCN and no SAGPooling). This change, combined with using a larger LSTM layer, enabled the model to capture more temporal features when modeling each clip and better generalize to the testing set. Model performance on this dataset and similar datasets could likely be improved by acquiring more consistent data via higher-resolution cameras with fixed camera angles and more accurate BEV projection approaches. However, as collisions are rare events, there are little to no datasets containing real-world collisions that meet these requirements. Despite these limitations, SG2VEC outperforms the DPM model by a significant margin, achieving **21.17%** higher accuracy, **31.40%** higher AUC, and a **21.92%** higher MCC score. Since DPM achieves a negative MCC score, its performance on

Dataset	Model	ATP	Avg. Seq. Len.	Ratio
271-syn	**sg2vec (Ours)**	10.004	33.920	**0.2949**
271-syn	DPM	17.399	32.899	0.5289
1043-syn	**sg2vec (Ours)**	6.442	37.343	**0.1725**
1043-syn	DPM	9.018	37.856	0.2382

Table 4.5: Average time of prediction (ATP) for collisions.

this dataset is worse than that of a random classifier (MCC of 0.0). Consistent with the synthetic dataset results, *sg2vec* using all frames performs better on the balanced *620-dash* dataset than SG2VEC (5-frames). Overall, these results show that, on very challenging and complex real-world driving scenarios, SG2VEC can perform much better than the current state-of-the-art.

Time of Prediction: Since collision prediction is a time-sensitive problem, we evaluated our methodology and the DPM on their average time-of-prediction (ATP) for video clips containing collisions. To calculate the ATP, we recorded the first frame index in each collision clip when the model correctly predicts that a collision would occur. We then averaged these indices and compared them with the average collision video clip length. Essentially, ATP gives an estimate of how early each model can predict a future collision. These results are shown in Table 4.5. On the *1043-syn* dataset, SG2VEC achieves 0.1725 for the ratio of the ATP and the average sequence length while the DPM achieves a ratio of 0.2382, indicating that SG2VEC predicts future collisions **39.07%** earlier than the DPM on average. In the context of real-world collision prediction, the average sequence in the *1043-syn* dataset represents 1.867 seconds of data. Thus, our methodology predicted collisions **122.7** milliseconds earlier than DPM on average. This extra time can be critical for ensuring that the AV avoids an impending collision.

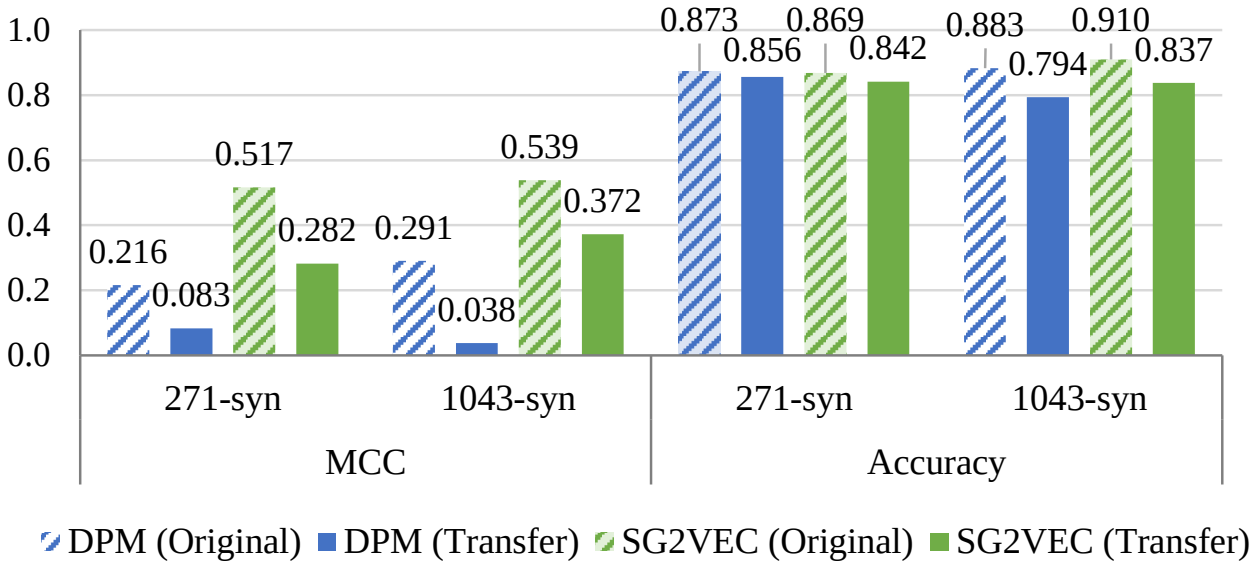

Figure 4.10: Performance after transferring the models trained on synthetic *271-syn* and *1043-syn* datasets to the real-world *571-honda* dataset.

4.6.3 The evaluation of SG2VEC's Transferability for Collision Prediction

The collision prediction models trained on simulated datasets must be transferable to real-world driving as they can differ significantly from simulations. To evaluate each model's ability to transfer knowledge, we trained each model on a synthetic dataset before testing it on the *571-honda* dataset. No additional domain adaptation was performed. We did not evaluate transferability to the *620-dash* dataset because it contains a wide range of highly dynamic driving maneuvers that were not present in our synthesized datasets. As such, evaluating transferability between our synthesized datasets and the 620-dash dataset would yield poor performance and would not provide insight. Figure 4.10 compares the accuracy and MCC for both the models on each training dataset and the *571-honda* dataset after transferring the trained model.

We observe that the SG2VEC model achieves a significantly higher MCC score than the DPM model after the transfer, suggesting that our methodology can better transfer knowledge from

a synthetic to a real-world dataset compared to the state-of-the-art DPM model. The drop in MCC values observed for both the models when transferred to the *571-honda* dataset can be attributed to the characteristic differences between the simulated and real-world datasets; the *571-honda* dataset contains a more heterogeneous set of road environments, lighting conditions, driving styles, etc., so a drop in performance after the transfer is expected. We also note that the MCC score for the SG2VEC model trained on *271-syn* dataset drops more than the model trained on the *1043-syn* dataset after the transfer, likely due to the smaller training dataset size. Regarding accuracy, the SG2VEC model trained on *1043-syn* achieves 4.37% higher accuracy and the model trained *271-syn* dataset achieves 1.47% lower accuracy than the DPM model trained on the same datasets. The DPM's similar accuracy after transfer likely results from the class imbalance in the *571-honda* dataset. Overall, we hypothesize that SG2VEC's use of an intermediate representation (i.e., *scene-graphs*) inherently improves its ability to generalize and thus results in an improved ability to transfer knowledge compared to CNN-based deep learning approaches.

4.6.4 Experimental Results: Evaluation on Industry-Standard AV Hardware

To demonstrate that the SG2VEC is implementable on industry-standard AV hardware, we measured its inference time (milliseconds), model size (kilobytes), power consumption (watts), and energy consumption per frame (milli-joules) on the industry-standard Nvidia DRIVE PX 2 platform, which was used by Tesla for their Autopilot system from 2016 to 2018 [63]. Our hardware setup is shown in Figure 4.11. For the inference time, we evaluated the average inference time (AIT) in milliseconds taken by each algorithm to process each frame. We recorded power usage metrics using a power meter connected to the power supply of the PX 2. To ensure that the reported numbers only reflected each model's power consumption and not that of background processes, we subtracted the hardware's idle power consumption

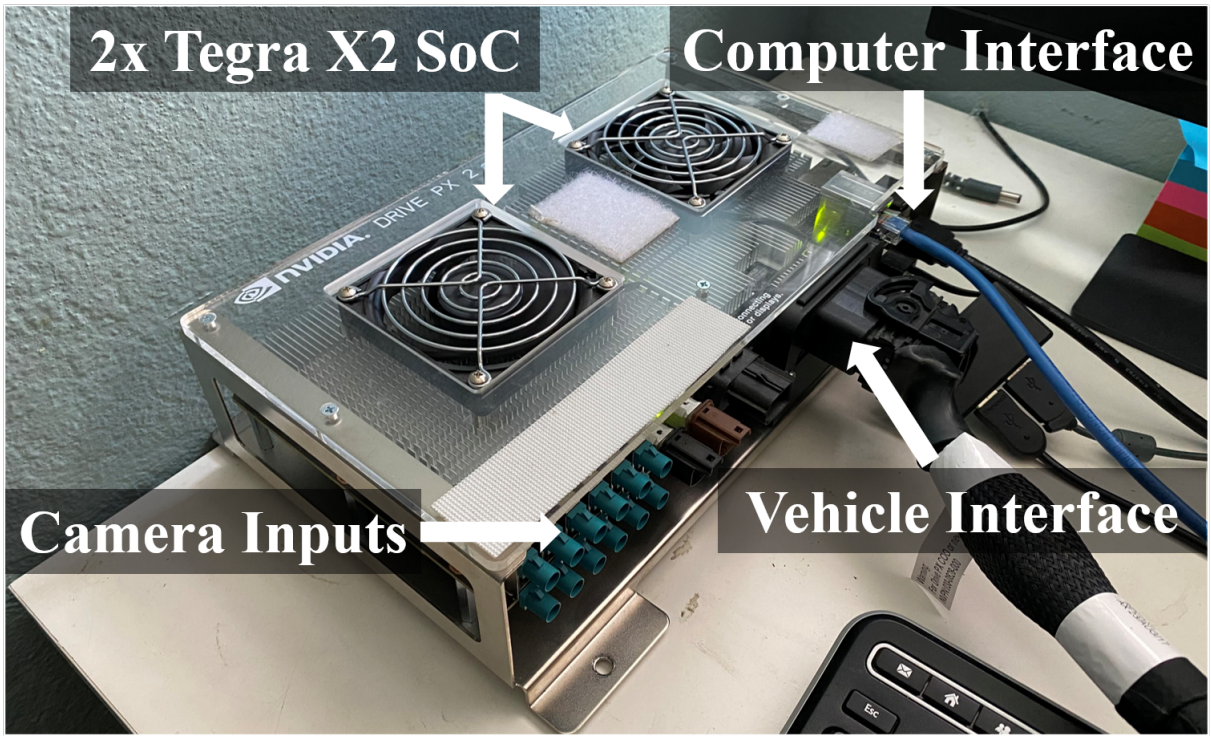

Figure 4.11: Our experimental setup for evaluating SG2VEC and DPM on the industry-standard Nvidia DRIVE PX 2 hardware.

Model	PC AIT (ms)	PX2 AIT (ms)	Size (KB)	Power (W)	Energy/frame (mJ)
sg2vec	**0.2549**	**0.4828**	**331**	**2.99**	**1.44**
DPM	1.393	4.535	2,764	4.42	20.0

Table 4.6: Performance evaluation of inference on *271-syn* on the Nvidia DRIVE PX 2.

from the averages recorded during each test. For a fair comparison, we captured the metrics for the core algorithm (i.e., the SG2VEC and DPM model), excluding the contribution from data loading and pre-processing. Both models were run with a batch size of 1 to emulate the real-world data stream where images are processed as they are received. For comparison, we also show the AIT on a PC for the two models.

Our results are shown in Table 4.6. SG2VEC performs inference **9.3x** faster than the DPM on the PX 2 with an **88.0%** smaller model and **32.4%** less power, making it undoubtedly more practical for real-world deployment. Our model also uses **92.8%** less energy to process each frame, which can be beneficial for electric vehicles with limited battery capacity. With

an AIT of 0.4828 ms, SG2VEC can theoretically process up to 2,071 frames/second (fps). In contrast, with an AIT of 4.535 ms, the DPM can only process up to 220 fps. In the context of real-world collision prediction, this means that SG2VEC could easily support multiple 60 fps camera inputs from the AV while DPM would struggle to support more than three.

4.7 Summary

In this chapter, we summarize graph learning applications on two critical AV safety-driven tasks. Subjective risk assessment is a challenging, safety-critical problem that requires a good semantic understanding of many possible road scenarios. Our results show that our scene-graph augmented approach outperforms state-of-the-art techniques at risk assessment tasks regarding accuracy (95.8% vs. 91.4%) and AUC (0.978 vs. 0.958). We also show that our approach can learn with much less training data than these techniques, as our approach achieves 91.8% accuracy on the *96-syn* dataset compared to 78.2% accuracy achieved by [195]. Additionally, our results show that our approach can better transfer knowledge gained from simulated datasets to real-world datasets (5.0% avg. acc. drop for our approach vs. 18.1% avg. acc. drop for [195]). We also show that using spatial and temporal attention components improves our approach's performance and explainability. Finally, our results illustrate that our model can more accurately assess the risk of diverse driving maneuvers than the state-of-the-art model (86.5% for our approach vs. 58.4% for [195]).

For collision prediction, we demonstrated that our SG2VEC outperforms the state-of-the-art method, DPM, in terms of average MCC (0.5055 vs. 0.2096), average inference time (0.255 ms vs. 1.39 ms), and average time of prediction (39.07% sooner than DPM). Additionally, we demonstrated that SG2VEC could transfer knowledge from synthetic datasets to real-world driving datasets more effectively than the DPM, achieving an average transfer MCC of 0.327 vs. 0.060. Finally, we showed that our methodology performs faster inference than

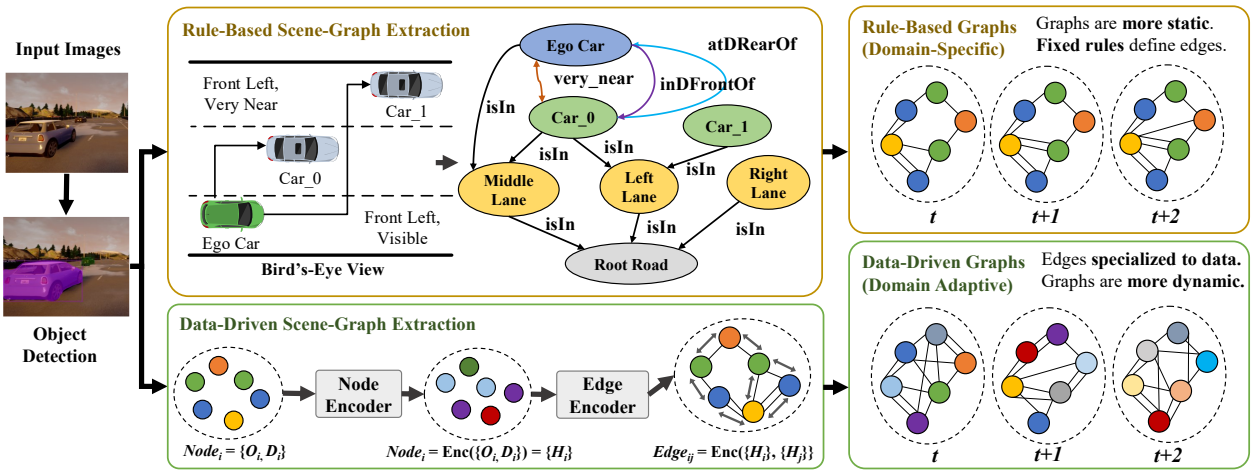

Figure 4.12: The differences between the scene-graphs generated by *RS2G* and those extracted with a rule-based method for a driving scene.

the DPM (0.4828 ms vs. 4.535 ms) with a smaller model size (331 KB vs. 2,764 KB) and reduced power consumption (2.99 W vs. 4.42 W) on the industry-standard Nvidia DRIVE PX 2 autonomous driving platform. In the context of real-world collision prediction, these results indicate that SG2VEC is a more practical choice for AV safety and could significantly improve consumer trust in AVs. Few works have explored graph-based solutions for other complex AV challenges such as localization, path planning, and control. These are open research problems that we reserve for future work.

To further increase the robustness and performance, it can be observed that our approaches previously mentioned in this chapter may be constrained by the domain-specific rules used for graph extraction that can vary in effectiveness across domains. As shown in Figure 4.12, our SG2VEC approach uses a rule-based *scene-graph* extraction and embedding approach, enabling GL algorithms to explicitly model inter-object relationships for scene understanding tasks such as risk assessment or collision prediction. However, SG2VEC requires domain-knowledge rules to define when to build graph edges for each relation type, which can limit its generalization ability. One solution to increase the generalization is by proposing integrating a data-driven *scene-graph* extraction component into the SG2VEC approach. As an ongoing work, we propose a new data-driven approach, *RS2G*, for learning the rules for

building relations from understanding driving scenarios [112] and demonstrate its benefits in advancing the performance of scene understanding tasks.

Chapter 5

Conclusion

Overall, through the exploration across many subfields, the results support the claim that graph learning can assist in automating Cognitive Intelligence for *Embedded Systems* (ES) and *Cyber-Phsycial Systems* (CPS) and making them more robust in terms of performance, generalization, as well as making explainable decisions. Chapter 2 and Chapter 3 show that graph learning can better generalize the knowledge from one dataset to various testing scenarios. Specifically, Chapter 2 demonstrates that *HW2VEC* automates the feature extraction process from the graphical representation of hardware designs. By doing so, therefore, it can model the behavior hardware designs into vectorized representations and use them to implement several downstream tasks. For detecting hardware Trojans, we showed that it could process hardware designs at the RTL level and extract meaningful information for identifying hardware Trojans from designs. Similarly, *HW2VEC* can help increase the abstraction of IP piracy identification to a behavioral level without any adding extra hardware cost. In Chapter 3, more sophisticated *Graph-of-Graph* (GoG) representations were proposed to model the behavioral information for software binaries. By imposing the architectural similarities supervision, the *CFG2VEC* methodology combines the control-flow and function call information, capable of learning the CPU architectural-agnostic features and generalizing the

task of function name prediction for binary functions across various CPU architectures.

In Chapter 4, we further demonstrate the power of graph learning in improving the scene understanding capabilities for cyber-physical systems, namely, autonomous driving systems. The proposal of *scene-graph* representation allows us to capture the relationships between traffic participants in a more structured and simplified way. Combined with the spatiotemporal graph embedding network, this novel framework, *SG2VEC*, can better identify the driving risk and predict the collision to improve the safety of autonomous driving systems. Because of such an intermediate representation between perception and reasoning processes, *SG2VEC* can understand the scenarios more robustly in generalizing the learning from synthetic data to real-world driving and easily transferring the learned knowledge to different driving domains.

Although this dissertation demonstrates that graph learning is a powerful tool in advancing the cognitive intelligence of embedded systems and cyber-physical systems, we want to argue that graph learning is not the only tool that can achieve the same goal. As a human who can have a very robust cognitive intelligence, our thought process is indeed much more complicated than we expected. As mentioned in [70], a human-like mind can be structured as a high-level breakdown of several sub-processes, such as reactive processes, deliberative processes, and meta-cognitive processes, and so on, interact with each other to process the information sent from the preception subsystem and complete the decision process and command our action subsystem. To make such a cognitive system more robust, enhancing the methodology to have better generalization is still a challenging research problem but has several benefits in various application fields. As evidenced in Chapter 2, in hardware security, new types of hardware trojan appear very frequently so whether the machine learning models or methodologies can generalize the knowledge of identifying hardware Trojans to the known designs and even completely unseen designs becomes critically important. In Chapter 4, automating the *scene-graph* extraction is a natural next step for the graph learning research

on autonomous driving systems because domain experts' graph extraction rules may only apply to a set of application traffic scenarios. Besides, automating the *scene-graph* extraction process also allows more information to be fused into the representations. Lastly, no matter what direction we are seeking, the most critical question is: how to embed human-like level computational intelligence into the embedded system and cyber-physical system. Answering this question remains an open challenge yet important as it can get us closer to the true "*Autonomy*" and drastically improve human lives.

Bibliography

[1] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón. Computing graph neural networks: A survey from algorithms to accelerators. *arXiv preprint arXiv:2010.00130*, 2020.

[2] A. Adadi and M. Berrada. Peeking inside the black-box: A survey on explainable artificial intelligence (xai). *IEEE Access*, 6:52138–52160, 2018.

[3] S. Adee. The hunt for the kill switch. In *IEEE Spectrum*, 2008.

[4] N. S. Agency. Ghidra - software reverse engineering framework., 2019.

[5] M. A. Al Faruque, S. R. Chhetri, A. Canedo, and J. Wan. Forensics of thermal side-channel in additive manufacturing systems. *University of California, Irvine*, 2016.

[6] M. Alhanahnah, Q. Lin, Q. Yan, N. Zhang, and Z. Chen. Efficient signature generation for classifying cross-architecture iot malware. In *2018 IEEE conference on communications and network security (CNS)*, pages 1–9. IEEE, 2018.

[7] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[8] M. Althoff, O. Stursberg, and M. Buss. Model-based probabilistic collision detection in autonomous driving. *IEEE Transactions on Intelligent Transportation Systems*, 10(2):299–310, 2009.

[9] F. Artuso, G. A. Di Luna, L. Massarelli, and L. Querzoni. In nomine function: Naming functions in stripped binaries with neural networks. *arXiv preprint arXiv:1912.07946*, 2019.

[10] B. Asgari, R. Hadidi, N. S. Ghaleshahi, and H. Kim. Pisces: Power-aware implementation of slam by customizing efficient sparse algebra. *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[11] M. AshrafiAmiri et al. Towards side channel secure cyber-physical systems. In *Real-Time and Embedded Systems and Technologies*, 2018.

[12] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[13] S. Baidya, Y.-J. Ku, H. Zhao, J. Zhao, and S. Dey. Vehicular and edge computing for emerging connected and autonomous vehicle applications. *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[14] M. Bansal, A. Krizhevsky, and A. Ogale. Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst. *arXiv preprint arXiv:1812.03079*, 2018.

[15] N. Bao, A. Carballo, C. Miyajima, E. Takeuchi, and K. Takeda. Personalized subjective driving risk: Analysis and prediction. *Journal of Robotics and Mechatronics*, 32(3):503–519, 2020.

[16] N. Bao, D. Yang, A. Carballo, Ü. Özgüner, and K. Takeda. Personalized safety-focused control by minimizing subjective risk. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 3853–3858. IEEE, 2019.

[17] A. Barua, Y. G. Achamyeleh, and M. A. A. Faruque. A wolf in sheep's clothing: Spreading deadly pathogens under the disguise of popular music. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 277–291, 2022.

[18] A. Barua, D. Muthirayan, P. P. Khargonekar, and M. A. Al Faruque. Hierarchical temporal memory-based one-pass learning for real-time anomaly detection and simultaneous data prediction in smart grids. *IEEE Transactions on Dependable and Secure Computing*, 19(3):1770–1782, 2020.

[19] A. Barua, L. Pan, and M. A. A. Faruque. Bayesimposter: Bayesian estimation based. bss imposter attack on industrial control systems. In *Annual Computer Security Applications Conference*, pages 440–454, 2022.

[20] C. Bernstein and M. Cobb. What is the advanced encryption standard (aes)? definition from searchsecurity, Sep 2021.

[21] I. Biederman and E. E. Cooper. Priming contour-deleted images: Evidence for intermediate representations in visual object recognition. *Cognitive psychology*, 23(3):393–419, 1991.

[22] Binary ninja. (2022) interactive disassembler, decompiler, and binary analysis platform, 2022.

[23] Binutils. (2019) objdump, 2019.

[24] D. Board. Defense science board (dsb) study on high performance microchip supply. *URL www. acq. osd. mil/dsb/reports/ADA435563. pdf,[March 16, 2015]*, 2005.

[25] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[26] A. P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

[27] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[28] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 353–368, 2013.

[29] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom. nuscenes: A multimodal dataset for autonomous driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11621–11631, 2020.

[30] H. Cai, V. W. Zheng, and K. C.-C. Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 30(9):1616–1637, 2018.

[31] J. Callas. Triple des: How strong is the data encryption standard?, May 2017.

[32] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[33] M. Chen, F. Herrera, and K. Hwang. Cognitive computing: architecture, technologies and intelligent applications. *Ieee Access*, 6:19774–19783, 2018.

[34] J. Chen et al. Decoy: Deflection-driven hls-based computation partitioning for obfuscating intellectual property. In *Design Automation Conference (DAC)*, 2020.

[35] S. R. Chhetri and M. A. Al Faruque. Side channels of cyber-physical systems: Case study in additive manufacturing. *IEEE Design & Test*, 34(4):18–25, 2017.

[36] S. R. Chhetri, A. Barua, S. Faezi, F. Regazzoni, A. Canedo, and M. A. Al Faruque. Tool of spies: Leaking your ip by altering the 3d printer compiler. *IEEE Transactions on Dependable and Secure Computing*, 18(2):667–678, 2019.

[37] S. R. Chhetri, A. Canedo, and M. A. A. Faruque. Kcad: kinetic cyber-attack detection method for cyber-physical additive manufacturing systems. In *Proceedings of the 35th international conference on Computer-Aided Design*, page 74. ACM, 2016.

[38] S. R. Chhetri, A. Canedo, and M. A. A. Faruque. Confidentiality breach through acoustic side-channel in cyber-physical additive manufacturing systems. *ACM Transactions on Cyber-Physical Systems*, 2(1):1–25, 2017.

[39] S. R. Chhetri, S. Faezi, and M. A. Al Faruque. Information leakage-aware computer-aided cyber-physical manufacturing. *IEEE Transactions on Information Forensics and Security*, 13(9):2333–2344, 2018.

[40] S. R. Chhetri, S. Faezi, A. Canedo, and M. A. Al Faruque. Thermal side-channel forensics in additive manufacturing systems. In *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS)*, pages 1–1. IEEE, 2016.

[41] S. R. Chhetri, S. Faezi, A. Canedo, and M. A. A. Faruque. Quilt: quality inference from living digital twins in iot-enabled manufacturing systems. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, pages 237–248. ACM, 2019.

[42] S. R. Chhetri, S. Faezi, and M. A. A. Faruque. Fix the leak!: an information leakage aware secured cyber-physical manufacturing system. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 1412–1417. European Design and Automation Association, 2017.

[43] S. R. Chhetri, S. Faezi, N. Rashid, and M. A. Al Faruque. Manufacturing supply chain and product lifecycle security in the era of industry 4.0. *Journal of Hardware and Systems Security*, 2:51–68, 2018.

[44] S. R. Chhetri, J. Wan, and M. A. Al Faruque. Cross-domain security of cyber-physical systems. In *2017 22nd Asia and South Pacific design automation conference (ASP-DAC)*, pages 200–205. IEEE, 2017.

[45] D. Chicco and G. Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC genomics*, 21(1):6, 2020.

[46] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang. Neural nets can learn function type signatures from binaries. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 99–116, USA, 2017. USENIX Association.

[47] D. Cogliati, M. Falchetto, D. Pau, M. Roveri, and G. Viscardi. Intelligent cyber-physical systems for industry 4.0. In *2018 First International Conference on Artificial Intelligence for Industries (AI4I)*, pages 19–22. IEEE, 2018.

[48] CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160., Dec. 3 2013.

[49] J. Dahl, G. R. de Campos, C. Olsson, and J. Fredriksson. Collision avoidance: A literature review on threat-assessment techniques. *IEEE Transactions on Intelligent Vehicles*, 4(1):101–113, 2018.

[50] S. Dange and M. Chatterjee. Iot botnet: the largest threat to the iot network. In *Data Communication and Networks*, pages 137–157. Springer, 2020.

[51] Y. David, U. Alon, and E. Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

[52] A. Davies. Google's self-driving car caused its first crash. *Wired*, 2016.

[53] Debian gnu linux installation guide, 2004.

[54] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. Carla: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017.

[55] L. Ďurfina, J. Křoustek, and P. Zemek. Psybot malware: A step-by-step decompilation case study. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 449–456. IEEE, 2013.

[56] L. A. Estrada-Jimenez, T. Pulikottil, R. S. Peres, S. Nikghadam-Hojjati, and J. Barata. Complexity theory and self-organization in cyber-physical production systems. *Procedia CIRP*, 104:1831–1836, 2021.

[57] F-secure. Attack landscape h1 2019, 2019.

[58] S. Faezi, R. Yasaei, and M. Al Faruque. Htnet: Transfer learning for golden chip-free hardware trojan detection. *IEEE/ACM Design Automation and Test in Europe Conference (DATE'21)*, 2021.

[59] S. Faezi et al. Brain-inspired golden chip free hardware trojan detection. *IEEE Transaction on Information Forensics and Security (IEEE TIFS'21)*, 2021.

[60] A. Faruque, M. Abdullah, S. R. Chhetri, A. Canedo, and J. Wan. Acoustic side-channel attacks on additive manufacturing systems. In *Proceedings of the 7th International Conference on Cyber-Physical Systems*, page 19. IEEE Press, 2016.

[61] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, 2016.

[62] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[63] L. Fred. All new Teslas are equipped with NVIDIA's new Drive PX 2 AI platform for self-driving - Electrek. https://electrek.co/2016/10/21/all-new-teslas-are-equipped-with-nvidias-new-drive-px-2-ai-platform-for-self-driving, Oct 2016. [Online; accessed 9. Nov. 2020].

[64] R. Fuller. Towards a general theory of driver behaviour. *Accident analysis & prevention*, 37(3):461–472, 2005.

[65] M. Fyrbiak et al. Graph similarity and its applications to hardware security. *IEEE Tran. on Computers*, 2020.

[66] H. Gao, S. Cheng, Y. Xue, and W. Zhang. A lightweight framework for function name reassignment based on large-scale stripped binaries. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 607–619, 2021.

[67] H. Gao and S. Ji. Graph u-nets. *arXiv preprint arXiv:1905.05178*, 2019.

[68] B. Gassmann, F. Pasch, F. Oboril, and K.-U. Scholl. Integration of formal safety models on system level using the example of responsibility sensitive safety and carla driving simulator. In *International Conference on Computer Safety, Reliability, and Security*, pages 358–369. Springer, 2020.

[69] I. Ghafoor, I. Jattala, S. Durrani, and C. M. Tahir. Analysis of openssl heartbleed vulnerability for embedded systems. In *17th IEEE International Multi Topic Conference 2014*, pages 314–319. IEEE, 2014.

[70] B. Goertzel, M. Iklé, and J. Wigmore. The architecture of human-like general intelligence. In *Theoretical foundations of artificial general intelligence*, pages 123–144. Springer, 2012.

[71] P. Goyal and E. Ferrara. Graph embedding techniques, applications, and performance: A survey. *CoRR*, abs/1705.02801, 2017.

[72] G. Grayson, G. Maycock, J. Groeger, S. Hammond, and D. Field. Risk, hazard perception and perceived control. *TRL Report TRL560. TRL Ltd., Crowthorne, UK*, 2003.

[73] T. Han, Y. Wang, and P. Liu. Hardware trojans detection at register transfer level based on machine learning. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2019.

[74] M. C. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the iscas-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers*, 16(3):72–80, 1999.

[75] I. U. Haq and J. Caballero. A survey of binary code similarity. *arXiv preprint arXiv:1909.11424*, 2019.

[76] S. Harada, H. Akita, M. Tsubaki, Y. Baba, I. Takigawa, Y. Yamanishi, and H. Kashima. Dual graph convolutional neural network for predicting chemical networks. *BMC bioinformatics*, 21:1–13, 2020.

[77] K. Hasegawa, Y. Shi, and N. Togawa. Hardware trojan detection utilizing machine learning approaches. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 1891–1896. IEEE, 2018.

[78] M. Hassen and P. K. Chan. Scalable function call graph-based malware classification. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 239–248, 2017.

[79] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680, 2018.

[80] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[81] The heartbleed vulnerability., 2014.

[82] C. Herder, M.-D. Yu, F. Koushanfar, and S. Devadas. Physical unclonable functions and applications: A tutorial. *Proceedings of the IEEE*, 102(8):1126–1141, 2014.

[83] Hex-rays. (2019) the hex-rays decompiler., 2019.

[84] S. Hex-Rays. Ida disassembler, 2017.

[85] Hopper. (2022) reverse engineering tool that lets you disassemble, decompile and debug your applications, 2022.

[86] F. Hu, Y. Lu, A. V. Vasilakos, Q. Hao, R. Ma, Y. Patil, T. Zhang, J. Lu, X. Li, and N. N. Xiong. Robust cyber–physical systems: Concept, models, and implementation. *Future generation computer systems*, 56:449–475, 2016.

[87] J. Hu, H. Niu, J. Carrasco, B. Lennox, and F. Arvin. Fault-tolerant cooperative navigation of networked uav swarms for forest fire monitoring. *Aerospace Science and Technology*, 123:107494, 2022.

[88] W. Hu, C.-H. Chang, A. Sengupta, S. Bhunia, R. Kastner, and H. Li. An overview of hardware security and trust: Threats, countermeasures and design tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[89] C. Huang, S. Xu, Z. Wang, S. Lan, W. Li, and Q. Zhu. Opportunistic intermittent control with safety guarantees for autonomous systems. *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[90] K. Huang, J. M. Carulli, and Y. Makris. Parametric counterfeit ic detection via support vector machines. In *2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 7–12. IEEE, 2012.

[91] H. L. D. Institute. Volvo collision avoidance features: initial results. *Highway Loss Data Institute Bulletin*, 29(5), 2012.

[92] R. Jain, R. Kasturi, and B. G. Schunck. *Machine vision*, volume 5. McGraw-Hill New York, 1995.

[93] Jasper. Jaspergold: Security path verification app. Technical report, Cadence, 2014.

[94] M. Jiménez, R. Palomera, and I. Couvertier. *Introduction to embedded systems*. Springer, 2013.

[95] S. Jose. Innovation is at risk as semiconductor equipment and materials. *Semiconductor Equipment and Material Industry (SEMI)*, 2008.

[96] A. Keliris and M. Maniatakos. Icsref: A framework for automated reverse engineering of industrial control systems binaries. *arXiv preprint arXiv:1812.03478*, 2018.

[97] S.-H. Kim, M. A. Cohen, S. Netessine, and S. Veeraraghavan. Contracting for infrequent restoration and recovery of mission-critical systems. *Management Science*, 56(9):1551–1567, 2010.

[98] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[99] B. Knyazev et al. Understanding attention and generalization in graph neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[100] F. Koushanfar. Active hardware metering by finite state machine obfuscation. *Hardware Protection through Obfuscation*, pages 161–187, 2017.

[101] A. Kulkarni, Y. Pino, and T. Mohsenin. Svm-based real-time hardware trojan detection for many-core platform. In *2016 17th International Symposium on Quality Electronic Design (ISQED)*, pages 362–367. IEEE, 2016.

[102] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019.

[103] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann. Industry 4.0. *Business & information systems engineering*, 6:239–242, 2014.

[104] C. Laugier, I. E. Paromtchik, M. Perrollaz, M. Yong, J. Yoder, C. Tay, K. Mekhnacha, and A. Nègre. Probabilistic analysis of dynamic scenes and collision risks assessment to improve driving safety. *IEEE Intelligent Transportation Systems Magazine*, 3(4):4–19, 2011.

[105] J.-P. Laumond et al. *Robot motion planning and control*, volume 229. Springer, 1998.

[106] D. Lavrinc. This is how bad self-driving cars suck in rain, 2014.

[107] T. B. Lee. Autopilot was active when a tesla crashed into a truck, killing driver. *ARS Technica*, 2019.

[108] J. Lee et al. Self-attention graph pooling. *arXiv preprint arXiv:1904.08082*, 2019.

[109] C. Li, Y. Meng, S. H. Chan, and Y.-T. Chen. Learning 3d-aware egocentric spatial-temporal interaction via graph convolutional networks. *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8418–8424, 2020.

[110] Y. Ma, Z. He, W. Li, L. Zhang, and B. Yu. Understanding graphs in eda: From shallow to deep learning. In *ISPD*, pages 119–126, 2020.

[111] Y. Ma, H. Ren, B. Khailany, H. Sikka, L. Luo, K. Natarajan, and B. Yu. High performance graph convolutional networks with applications in testability analysis. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[112] A. V. Malawade, S.-Y. Yu, J. Wang, and M. A. A. Faruque. Rs2g: Data-driven scene-graph extraction and embedding for robust autonomous perception and scenario understanding. *arXiv preprint arXiv:2304.08600*, 2023.

[113] M. McFarland. Who's responsible when an autonomous car crashes? *CNN*, 2016.

[114] Y. Moghaddas, T. Nguyen, S.-Y. Yu, R. Yasaei, and M. A. A. Faruque. Technical report for hw2vec – a graph learning tool for automating hardware security. Technical Report TR-21-02, Center for Embedded and Cyber-Physical Systems University of California, Irvine, Irvine, CA 92697-2620, USA, July 2021.

[115] R. Mudge, D. Montgomery, E. Groshen, J. Groshen, S. Helper, and C. Carson. America's workforce and the self-driving future realizing productivity gains and spurring economic growth. *no. june*, 2018.

[116] S. Mylavarapu, M. Sandhu, P. Vijayan, K. M. Krishna, B. Ravindran, and A. Namboodiri. Towards accurate vehicle behaviour classification with multi-relational graph convolutional networks. *arXiv preprint arXiv:2002.00786*, 2020.

[117] W. G. Najm, J. D. Smith, M. Yanagisawa, et al. Pre-crash scenario typology for crash avoidance research. Technical report, United States. National Highway Traffic Safety Administration, 2007.

[118] J. Nilsson, A. C. Ödblom, and J. Fredriksson. Worst-case analysis of automotive collision avoidance systems. *IEEE Transactions on Vehicular Technology*, 65(4):1899–1911, 2015.

[119] D. Nistér, H.-L. Lee, J. Ng, and Y. Wang. The safety force field. *NVIDIA White Paper*, 2019.

[120] M. Nyre-Yu, K. Butler, and C. Bolstad. A task analysis of static binary reverse engineering for security, Jan 2022.

[121] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on intelligent vehicles*, 1(1):33–55, 2016.

[122] A. Panarello, N. Tapas, G. Merlino, F. Longo, and A. Puliafito. Blockchain and iot integration: A systematic survey. *Sensors*, 18(8):2575, 2018.

[123] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[124] S. Patnaik et al. Raise your game for split manufacturing: Restoring the true functionality through beol. In *Design Automation Conference (DAC)*, 2018.

[125] L. Piccolboni et al. Efficient control-flow subgraph matching for detecting hardware trojans in rtl models. *ACM Tran. on Embedded Computing Systems (TECS)*, 2017.

[126] D. A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pages 305–313, 1989.

[127] P. Poudel et al. Flashmark: watermarking of nor flash memories for counterfeit detection. In *Design Automation Conference (DAC)*, 2020.

[128] M. T. Rahman, K. Xiao, D. Forte, X. Zhang, J. Shi, and M. Tehranipoor. Ti-trng: Technology independent true random number generator. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2014.

[129] S. Rai et al. Hardware watermarking using polymorphic inverter designs based on reconfigurable nanotechnologies. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019.

[130] R. Rajamani. *Vehicle dynamics and control*. Springer Science & Business Media, 2011.

[131] J. Rajendran et al. Security analysis of integrated circuit camouflaging. In *ACM conference on Computer & communications security*, 2013.

[132] J. Rajendran et al. Detecting malicious modifications of data in third-party intellectual property cores. In *ACM/IEEE Design Automation Conference (DAC)*, 2015.

[133] J. Rajendran et al. Formal security verification of third party intellectual property cores for information leakage. In *International Conference on VLSI Design and Embedded Systems (VLSID)*, 2016.

[134] V. Ramanishka, Y.-T. Chen, T. Misu, and K. Saenko. Toward driving scene understanding: A dataset for learning driver behavior and causal reasoning. In *Conference on Computer Vision and Pattern Recognition*, 2018.

[135] V. Ramanishka, Y.-T. Chen, T. Misu, and K. Saenko. Toward driving scene understanding: A dataset for learning driver behavior and causal reasoning. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7699–7707, 2018.

[136] K. Redmond, L. Luo, and Q. Zeng. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. *arXiv preprint arXiv:1812.09652*, 2018.

[137] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[138] S. Rokka Chhetri. *Data-Driven Modeling of Cyber-Physical Systems using Side-Channel Analysis*. PhD thesis, UC Irvine, 2019.

[139] M. Rostami, F. Koushanfar, and R. Karri. A primer on hardware security: Models, methods, and metrics. *Proceedings of the IEEE*, 102(8):1283–1295, 2014.

[140] M. Rostami, F. Koushanfar, J. Rajendran, and R. Karri. Hardware security: Threat models and metrics. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 819–823. IEEE, 2013.

[141] T. Rothwell and J. Youngman. The gnu c reference manual. *Free Software Foundation, Inc*, page 86, 2007.

[142] S. Saha et al. Improved test pattern generation for hardware trojan detection using genetic algorithm and boolean satisfiability. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 2015.

[143] A. Sargolzaei, A. Abbaspour, M. A. Al Faruque, A. Salah Eddin, and K. Yen. Security challenges of networked control systems. In *Sustainable interdependent networks*, pages 77–95. Springer, 2018.

[144] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pages 593–607. Springer, 2018.

[145] S. Shalev-Shwartz, S. Shammah, and A. Shashua. On a formal model of safe and scalable self-driving cars. *arXiv preprint arXiv:1708.06374*, 2017.

[146] K. Shamsi, M. Li, K. Plaks, S. Fazzari, D. Z. Pan, and Y. Jin. Ip protection and supply chain security through logic obfuscation: A systematic overview. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24(6):1–36, 2019.

[147] W. Shao, Q. Yang, X. Guo, and R. Cai. A survey of available information recovery of binary programs based on machine learning. In *2022 5th International Conference on Artificial Intelligence and Big Data (ICAIBD)*, pages 125–132. IEEE, 2022.

[148] Shellshock: All you need to know about the bash bug vulnerability, 2014.

[149] D. Smilkov, N. Thorat, C. Nicholson, E. Reif, F. B. Viégas, and M. Wattenberg. Embedding projector: Interactive visualization and interpretation of embeddings. *arXiv preprint arXiv:1611.05469*, 2016.

[150] M. Sokolova and G. Lapalme. A systematic analysis of performance measures for classification tasks. *Information processing & management*, 45(4):427–437, 2009.

[151] S. Sontges, M. Koschi, and M. Althoff. Worst-case analysis of the time-to-react using reachable sets. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1891–1897. IEEE, 2018.

[152] Special 301 report, 2017.

[153] J. Staff. Assembled labeled library for static analysis research (allstar) dataset, Dec 2019.

[154] M. Stone. Shellshock in-depth: Why this old vulnerability won't go away, 2020.

[155] M. Strickland, G. Fainekos, and H. B. Amor. Deep predictive models for collision risk assessment in autonomous driving. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8, 2018.

[156] P. Subramanyan and D. Arora. Formal verification of taint-propagation security properties in a commercial soc design. In *Design, Automation & Test in Europe Conference (DATE)*, 2014.

[157] S. Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, volume 9040 of *Lecture Notes in Computer Science*, pages 451–460. Springer International Publishing, Apr 2015.

[158] S. Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *International Symposium on Applied Reconfigurable Computing*, 2015.

[159] B. Tan and R. Karri. Challenges and new directions for ai and hardware security. In *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 277–280. IEEE, 2020.

[160] R. Team. *Radare2 Book*. GitHub, 2017.

[161] J. Teel. How long does it take to develop a new product and get it to market?, Oct 2017.

[162] L. Torrey and J. Shavlik. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI global, 2010.

[163] U. Tränkle, C. Gelau, and T. Metker. Risk perception and age-specific accidents of young drivers. *Accident Analysis & Prevention*, 22(2):119–125, 1990.

[164] Trusthub, 2016.

[165] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[166] M. J. Van Emmerik. *Static single assignment for decompilation*. University of Queensland, 2007.

[167] D. Vasan, M. Alazab, S. Venkatraman, J. Akram, and Z. Qin. Mthael: Cross-architecture iot malware detection based on neural network advanced ensemble learning. *IEEE Transactions on Computers*, 69(11):1654–1667, 2020.

[168] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[169] A. Waksman et al. Fanci: identification of stealthy malicious logic using boolean functional analysis. In *ACM SIGSAC Conference on Computer and Communications Security*, 2013.

[170] X. Wang, J. Liu, T. Qiu, C. Mu, C. Chen, and P. Zhou. A real-time collision prediction mechanism with deep learning for intelligent transportation system. *IEEE transactions on vehicular technology*, 69(9):9497–9508, 2020.

[171] Y. Wang, Z. Liu, Z. Zuo, Z. Li, L. Wang, and X. Luo. Trajectory planning and safety assessment of autonomous vehicles based on motion prediction and model predictive control. *IEEE Transactions on Vehicular Technology*, 68(9):8546–8556, 2019.

[172] Copyrights and patents, piracy and theft, 2018.

[173] H. Woo, Y. Ji, Y. Tamura, Y. Kuroda, T. Sugano, Y. Yamamoto, A. Yamashita, and H. Asama. Advanced adaptive cruise control based on collision risk assessment. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 939–944, 2018.

[174] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick. Detectron2, 2019.

[175] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 2020.

[176] F. Xia, K. Sun, S. Yu, A. Aziz, L. Wan, S. Pan, and H. Liu. Graph learning: A survey. *IEEE Transactions on Artificial Intelligence*, 2(2):109–127, 2021.

[177] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor. Hardware trojans: Lessons learned after one decade of research. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(1):1–23, 2016.

[178] Y. Xie et al. Delay locking: Security enhancement of logic locking against ic counterfeiting and overproduction. In *Design Automation Conference (DAC)*, 2017.

[179] D. Xu, Y. Zhu, C. B. Choy, and L. Fei-Fei. Scene graph generation by iterative message passing. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5410–5419, 2017.

[180] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

[181] K. Xu et al. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[182] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 158–177. IEEE, 2016.

[183] J. Yang, J. Lu, S. Lee, D. Batra, and D. Parikh. Graph r-cnn for scene graph generation. In *Proceedings of the European conference on computer vision (ECCV)*, pages 670–685, 2018.

[184] Y. Yao, X. Wang, M. Xu, Z. Pu, E. Atkins, and D. Crandall. When, where, and what? a new dataset for anomaly detection in driving videos. *arXiv preprint arXiv:2004.03044*, 2020.

[185] R. Yasaei, S. Faezi, and M. A. Al Faruque. Golden reference-free hardware trojan localization using graph convolutional network. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(10):1401–1411, 2022.

[186] R. Yasaei, S.-Y. Yu, and M. A. Al Faruque. Gnn4tj: Graph neural networks for hardware trojan detection at register transfer level. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1504–1509. IEEE, 2021.

[187] R. Yasaei, S.-Y. Yu, and M. A. A. Faruque. Gnn4ip: Graph neural network for hardware intellectual property piracy detection. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Ieee, 2021.

[188] R. Yasaei, S.-Y. Yu, and M. A. A. Faruque. Gnn4tj: Graph neural networks for hardware trojan detection at register transfer level. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Ieee, 2021.

[189] A. Yeh. Trends in the global ic design service market. *DIGITIMES research*, 2012.

[190] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec. Hierarchical graph representation learning with differentiable pooling. *arXiv preprint arXiv:1806.08804*, 2018.

[191] S. Y. Yu, S. R. Chhetri, A. Canedo, P. Goyal, and M. A. A. Faruque. Pykg2vec: A python library for knowledge graph embedding. *ArXiv*, abs/1906.04239, 2019.

[192] S.-Y. Yu, A. V. Malawade, S. R. Chhetri, and M. A. Al Faruque. Sabotage attack detection for additive manufacturing systems. *IEEE Access*, 8:27218–27231, 2020.

[193] S.-Y. Yu, A. V. Malawade, D. Muthirayan, P. P. Khargonekar, and M. A. Al Faruque. Scene-graph augmented data-driven risk assessment of autonomous vehicle decisions. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[194] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda. A survey of autonomous driving: Common practices and emerging technologies. *arXiv preprint arXiv:1906.05113*, 2019.

[195] E. Yurtsever, Y. Liu, J. Lambert, C. Miyajima, E. Takeuchi, K. Takeda, and J. H. Hansen. Risky action recognition in lane change video clips using deep spatiotemporal networks with segmentation mask transfer. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 3100–3107. IEEE, 2019.

[196] E. Yurtsever, S. Yamazaki, C. Miyajima, K. Takeda, M. Mori, K. Hitomi, and M. Egawa. Integrating driving behavior and traffic context through signal symbolization for data reduction and risky lane change detection. *IEEE Transactions on Intelligent Vehicles*, 3(3):242–253, 2018.

[197] L. Zhang, W. Xiao, Z. Zhang, and D. Meng. Surrounding vehicles motion prediction for risk assessment and motion planning of autonomous vehicle in highway scenarios. *IEEE Access*, 8:209356–209376, 2020.

[198] Y. Zhang, H. Ren, and B. Khailany. Grannite: Graph neural network inference for transferable power estimation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[199] Y. Zhang, R. Yasaei, H. Chen, Z. Li, and M. A. Al Faruque. Stealing neural network structure through remote fpga side-channel analysis. *IEEE Transactions on Information Forensics and Security*, 16:4377–4388, 2021.

[200] B. Zhang et al. Analysis of security of split manufacturing using machine learning. In *Design Automation Conference (DAC)*, 2018.

[201] J. Zhang et al. Veritrust: Verification for hardware trust. *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems*, 2015.

[202] K. Zhidanov, S. Bezzateev, A. Afanasyeva, M. Sayfullin, S. Vanurin, Y. Bardinova, and A. Ometov. Blockchain technology for smartphones and constrained iot devices: A future perspective and implementation. In *2019 IEEE 21st Conference on Business Informatics (CBI)*, volume 2, pages 20–27. IEEE, 2019.

Appendix A

Other Research Areas

A.1 Overview

In addition to the research areas discussed in the Chapters of this dissertation, I have also studied several other relevant research topics. Below, these topics and their related publications are briefly summarized.

A.2 Machine Learning for Sabotage Attack Detection in Advanced Additive Manufacturing

Additive Manufacturing (AM), or 3D Printing, is seeing practical use for the rapid prototyping and production of industrial parts. The digitization of such systems not only makes AM a crucial technology in Industry 4.0 but also presents a broad attack surface that is vulnerable to sabotage attacks [43]. In the field of AM security, sabotage attacks are cyber-attacks that introduce inconspicuous defects to a manufactured component at any specific

process of the AM digital process chain, resulting in the compromise of the component's structural integrity and load-bearing capabilities [37]. Defense mechanisms that detect such attacks using side-channel analysis have been studied [35, 41, 37]. However, most current works focus on modeling the state of AM systems using a single side-channel, thus limiting their effectiveness at attack detection. To address this challenge, we present a novel multi-modal sabotage attack detection system for Additive Manufacturing (AM) machines in [192]. By utilizing multiple side-channels (*e.g.*, vibration, acoustic, magnetic, power), we improve system state estimation significantly over that of existing methods. Besides, we analyze the value of each side-channel for performing attack detection in terms of mutual information shared with the machine control parameters. More details on how side-channels can be leveraged in manufacturing systems are provided in [138].

A.3 Pykg2vec: A python library for knowledge graph embedding

Pykg2vec is a Python library for learning the representations of the entities and relations in knowledge graphs [191]. *Pykg2vec*'s flexible and modular software architecture currently implements 25 state-of-the-art knowledge graph embedding algorithms and is designed to easily incorporate new algorithms. The goal of *Pykg2vec* is to provide a practical and educational platform to accelerate research in knowledge graph representation learning. *Pykg2vec* is built on top of PyTorch and Python's multiprocessing framework and provides modules for batch generation, Bayesian hyperparameter optimization, evaluation of KGE tasks, embedding, and result visualization. *Pykg2vec* is released under the MIT License and is also available in the Python Package Index (PyPI).