UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Secure Software Engineering

Bachelor's Thesis

Submitted to the Secure Software Engineering Research Group
in Partial Fulfilment of the Requirements for the Degree of

Bachelor of Science

Tailoring Code Property Graphs to Jimple

by
Michael Youkeim

Thesis Supervisor:
Prof. Dr. Eric Bodden
and
Prof. Dr.-Ing. Juraj Somorovsky

Paderborn, November 26, 2024

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

______________________________	______________________________
Ort, Datum	Unterschrift

Abstract. The increased complexity of modern software has led to much more sophisticated attack vectors. As a result, we require newer vulnerability detection methods to ensure software security without compromising efficiency.
The *Code Property Graph* (CPG) is a program representation that provides a comprehensive overview of program behavior, combining abstract syntax trees, control flow graphs, and program dependence graphs. With such a detailed data structure, we can detect patterns that characterize known vulnerabilities and identify various security threats. Querying the combined data structure instead of the individual graphs enables the detection of multidimensional scenarios.
This work aims to integrate the advantages of CPGs into software systems that utilize the Jimple intermediate representation. We introduce *JimNode*, a novel approach for generating CPGs specifically tailored to Jimple. Despite the model incompatibility, our evaluation, which covered approximately 50,800 methods, reveals an 88.07% similarity of the inter-statement edges compared to Joern, the state-of-the-art tool for CPG generation. We provide a detailed analysis of our methodology and discuss why it is better suited for Jimple programs than Joern's language-agnostic approach.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem Statement . . . 2
1.3 Thesis Structure . . . 3

2 Background **5**
2.1 Program Analysis . . . 5
2.1.1 Data Flow Analysis . . . 5
2.1.2 Control Flow Analysis . . . 6
2.2 Jimple . . . 6
2.2.1 Challenges of Bytecode . . . 6
2.2.2 Advantages of Jimple . . . 8
2.2.3 Impact on Program Analysis . . . 9
2.3 Property Graphs . . . 9
2.3.1 Core Concepts and Structure . . . 9
2.3.2 Querying Property Graphs . . . 10
2.3.3 Formalizing Property Graphs . . . 11

3 Code Property Graphs **13**
3.1 Structure and Components . . . 13
3.1.1 Abstract Syntax Trees . . . 13
3.1.2 Control Flow Graphs . . . 14
3.1.3 Program Dependence Graphs . . . 15
3.1.4 Constructing the CPG . . . 16
3.2 Relevance and Applications . . . 17
3.2.1 Vulnerability Characterization . . . 17
3.2.2 Vulnerability Detection . . . 18

4 Contribution **19**
4.1 Theoretical Foundations . . . 19
4.1.1 Control Dependence . . . 19
4.1.2 Data Dependence . . . 23
4.2 Methodology and Model Design . . . 25
4.2.1 Design and Model Structure of JimNode . . . 25
4.2.2 Advantages Over Language-Agnostic Models . . . 27
4.2.3 Practical Disadvantages of Joern . . . 27
4.3 Implementation and Integration . . . 28

4.3.1 Adopting SootUp's CFG and AST 28
4.3.2 Building the CDG and DDG 28
4.3.3 Model Considerations 28

5 Evaluation **31**
5.1 Research Questions and Criteria 31
5.2 Evaluation Methodology 31
5.3 Results and Discussion 33
5.3.1 RQ1: Similarity Analysis 33
5.3.2 RQ2: Performance Analysis 35

6 Related Work **37**
6.1 Vulnerability Detection 37
6.2 Static Analysis Frameworks 37
6.3 Standardizing Property Graphs 38

7 Threats to Validity **39**
7.1 Technical and Tool-Specific Considerations 39
7.1.1 Tool-Specific Limitations 39
7.1.2 Variability and Bias in Jimple Code Generation 39
7.2 Methodological Concerns 40
7.2.1 Conversion and Transformation Bias 40
7.2.2 Measurement and Evaluation Criteria 40
7.3 Data Selection and Reliability 41
7.3.1 Selection Bias 41
7.3.2 API and Data Source Reliability 41

8 Conclusion and Future Work **43**
8.1 Conclusion 43
8.2 Future Work 44
8.2.1 Advanced Query Capabilities 44
8.2.2 Performance Optimization 44
8.2.3 Database Integration 44
8.2.4 Graph Visualization Enhancements 44

Bibliography **45**

1 Introduction

In the following, we first detail the motivations guiding this research and the specific objectives and challenges we aim to tackle in Section 1.1. The Problem Statement in Section 1.2 further clarifies the particular issues we address. Section 1.3 then outlines the organization of the chapters, providing a clear framework for navigating this thesis.

1.1 Motivation

As the complexity and size of software systems continue to expand, the task of detecting vulnerabilities has become increasingly challenging. Manual methods and traditional tools are often inadequate for the nuanced demands of modern software security [Gra][BBC$^+$].

Detecting vulnerabilities in software is an intricate and multifaceted challenge. Vulnerabilities encompass a multitude of aspects, each adding to the complexity of the task [AW].

Consider, for instance, the case of buffer overflow vulnerabilities. Buffer overflow vulnerabilities involve the improper handling of memory within a program. When a program attempts to store more data in a memory buffer than it can safely accommodate, it can lead to a breach of the system's security [BEHW]. These vulnerabilities, while seemingly straightforward, encompass various dimensions of software security [PT].

Identifying buffer overflow vulnerabilities requires a deep understanding of the logic and control flow of the code. These vulnerabilities often depend on the exact sequence of instructions that lead to the overflow, making manual detection a cumbersome and error-prone process [TZWL]. Detecting buffer overflows also requires monitoring how data is read into and manipulated in memory buffers, which is also tedious given the complexity of modern software. The multifaceted nature of vulnerabilities underscores the necessity for advanced techniques capable of integrating comprehensive analyses [AW]. Such techniques would provide a structured approach to effectively tackle the different dimensions of vulnerabilities and ensure security. In addition, buffer overflows often depend on external inputs, which makes it difficult to predict when and how they might occur. This unpredictability makes detection even more difficult.

Another reason that contributes to the amount of bugs in modern software is its large scale. With software systems comprising millions of lines of code, attempting to manually detect all vulnerabilities is a daunting task fraught with limitations. Modern software is characterized by its sheer scale and complexity. From operating systems to web applications, the lines of code have multiplied exponentially. Large-scale software systems involve intricate code, numerous dependencies, and frequent updates. In such a landscape, manual vulnerability detection

becomes impractical [BBC+] [AHSM]. Furthermore, the rapid pace of software development demands quick and continuous updates. Vulnerabilities can be introduced or fixed with each update, making it a race against time. Manual detection methods simply cannot keep up with this velocity. This underscores the need for automation. Automated vulnerability scanning tools that utilize techniques such as static and dynamic analysis can efficiently scan large code bases and identify potential vulnerabilities quickly and accurately [TZWL]. These tools are able to handle the scale and complexity of modern software and provide comprehensive coverage and timely detection.

Java is one of the most widely used programming languages, known for its ability to run on any device using the *Java Virtual Machine* (JVM). Java's design utilizes the JVM to ensure stable performance and security across different computing environments. This thesis addresses these challenges by integrating *Code Property Graph* (CPG) into the Java analysis and optimization environment, with a specific focus on tailoring them to Jimple, an intermediate representation of Java bytecode. Unlike conventional methods that primarily focus on syntax, this implementation enables a multi-dimensional analysis of software, encompassing not only the syntactical elements but also the behavioral and semantic aspects of code.

CPGs provide a practical framework for vulnerability detection. The main contributor to their effectiveness is their ability to represent the source code comprehensively. The integration of the *Abstract Syntax Tree* (AST), *Control Flow Graph* (CFG), and acpdf subgraphs facilitates a more thorough analysis, as the process of searching for vulnerabilities involves querying the integrated structure of the CPG using queries that include aspects of all of the three subgraphs.

Yamaguchi et al.demonstrated the effectiveness of CPGs in detecting software vulnerabilities by identifying 18 previously unknown vulnerabilities in the Linux kernel [YGAR]. These vulnerabilities span various types, including buffer overflows, integer overflows, format string vulnerabilities, and memory disclosure. By using graph traversals, which are similar to database queries, they were able to systematically examine combined properties of the code for signs of vulnerabilities. This method is beneficial for identifying subtle and intricate vulnerabilities that conventional, isolated methods of static code analysis might miss. The effectiveness of CPGs in this study underscores their potential as a powerful tool for improving code security through detailed and proactive vulnerability detection.

1.2 Problem Statement

While there exist tools that support the generation of CPGs, like the language-agnostic Joern, they are not tailored to the specific structure of Jimple [joe]. This lack of specialization results in queries that are not fully type-supported and do not efficiently utilize the unique types and structures of Jimple. Moreover, these general models tend to be excessively intricate as they cater to various scenarios across different programming languages without accounting for the inherent simplicity of Jimple. This thesis aims to address these issues by developing CPGs specifically for Jimple. By focusing on the characteristic aspects of Jimple, this specialized approach aims to improve the efficiency and accuracy of analysis processes for Java applications and overcome challenges that broader, language-agnostic tools may not be able to fully overcome.

This integration of CPGs into Jimple marks a significant step towards optimizing the use of CPGs in bytecode optimization frameworks and ensures that static analysis tools can analyze complex code structures more effectively. This specialization in the use of CPGs for Jimple facilitates a deeper understanding of the inherent vulnerabilities of Java applications.

The broader goal of this thesis is to enrich the toolkit available for vulnerability detection in Java, contributing to the ongoing efforts to secure increasingly complex software systems. In doing so, it aims to bring about a more nuanced and thorough approach to understanding and

mitigating software vulnerabilities, particularly in Java-based environments.

We focus on addressing two primary challenges. The first challenge involves the detailed alignment of Jimple's unique structure within the CPG framework. This entails accurately capturing the intricacies of Jimple programs in CPGs to ensure that the CPG representation reflects the specific properties and semantics of Jimple code. The second challenge concerns the efficient generation of CPGs tailored to Jimple. This aspect requires leveraging efficient analyses and algorithms to generate the CPGs. Addressing these challenges is critical to the accurate and efficient application of CPGs to Jimple and paves the way for more targeted and effective vulnerability analysis within Java's unique ecosystem.

1.3 Thesis Structure

Following, we provide a detailed roadmap of the entire thesis, outlining its structure and providing a preview of the subsequent chapters.

In Chapter 2, we lay the foundation by discussing the fundamental concepts of static analysis in the context of software security. We explore common vulnerability detection techniques, setting the stage for the more advanced approaches presented. We then delve into the intricacies of the Jimple intermediate representation and discuss its relationship to Java bytecode. Finally, we introduce the concept of property graphs and illustrate their role as a versatile means of representing structured data.

In Chapter 3, we explain the structure and essential components of code property graphs. We demonstrate how these components work together to form a powerful representation of program properties. We then explore their advantages and potential limitations. Finally, we delve into their practical applications in the context of detecting common vulnerability patterns. In addition, we present practical examples and case studies that illustrate how these structures are effectively utilized to improve software security.

In Chapter 4, we delve into the methodology adopted throughout this research. We provide a comprehensive explanation of the approaches and techniques used to achieve our research objectives. We then offer detailed insights into the implementation of the solution proposed in this thesis.

Moving on to Chapter 5, we assess the correctness of the generated CPGs in representing the specified input accurately. The purpose of this assessment is to ensure that the guidelines produced are in line with our objectives and expectations. In addition to correctness, we also evaluate the performance of the proposed solution.

Chapter 6 discusses other studies and tools related to our project. We review the approaches of other studies to similar problems and where our work fits into the overall picture.

Following the discussion of related work, Chapter 7 addresses the threats to validity of our research. This chapter carefully examines potential limitations and biases inherent in our study's design, methodology, and analysis. By acknowledging these threats, we discuss the implications for the reliability and generalizability of our findings and describe the measures employed to minimize their impact. This critical evaluation is essential for ensuring the robustness of our research conclusions.

Finally, in the Chapter 8, we provide a concise summary of the work performed and the results achieved throughout the thesis. We then explore avenues for future work, suggesting how the completed research could be expanded to achieve further goals.

2 Background

In the following, we provide the foundational knowledge necessary for understanding the context of our work. In Section 2.1, we provide an introduction to program analysis. We then explore the specifics of Jimple in Section 2.2 and conclude with an introduction to property graphs in Section 2.3.

2.1 Program Analysis

Program analysis is an essential aspect of software development that involves examining, understanding, and enhancing the code [NNH]. Program analysis can be broadly categorized into various types, each with its own methodologies, goals, and applications. The two most fundamental types are *static analysis* and *dynamic analysis*. However, there are various other specialized categories such as behavioral analysis, fuzz testing, and symbolic execution [BCD+][LPJ+][SR]. Each of these techniques offers unique insights into the behavior and performance of software systems. In this section, we mainly lay our focus on static analysis.

In static analysis, the source code of a program is examined without being executed. One of the main objectives of static analysis is to identify areas for optimization to improve a program's overall efficiency. This involves analyzing algorithms for efficiency, detecting redundant code, and identifying opportunities for code refactoring. In addition, static analysis helps to optimize resource usage, e. g., memory consumption, by analyzing the code structure and applying the appropriate improvements.

2.1.1 Data Flow Analysis

Data-flow analysis is among the static analysis techniques used to examine and optimize programs [AC][VJB+]. Data-flow analysis tracks how variables are defined and used in the program, offering insights into their usage and the paths data takes. This is crucial for identifying segments of code where data processing can be optimized. It helps uncover inefficiencies such as unused variables or unreachable code segments. Furthermore, there are several data flow analyses that enhance the security of the code, such as taint analysis. Taint analysis tracks the flow of potentially insecure or "tainted" data through the program. This is significant in identifying security vulnerabilities, especially in scenarios where untrusted input can flow into sensitive program areas, leading to issues such as SQL injection or cross-site scripting (XSS) vulnerabilities. This analysis helps to find the exact places in the code where data sanitization or validation checks

should be implemented. We will further delve into the theoretical foundations and practical applications of data flow analysis in Chapter 4.

2.1.2 Control Flow Analysis

Another critical aspect within the domain of static analysis is *Control Flow Analysis*. Control flow analysis is a technique used to understand the order in which a program's individual statements, instructions, or function calls are executed or evaluated. This analysis is essential in identifying the different paths a program might take during its execution, which is crucial for both optimizing performance and ensuring correct program behavior. By analyzing the control-flow graph of a program, compilers and static analysis tools can identify unreachable code and dead code, i.e., code that does not affect the program's output, and can perform optimizations to enhance the program's performance.

2.2 Jimple

Jimple is an intermediate representation of Java bytecode that is more human-readable and more suited for analysis than bytecode [VrCG+].

2.2.1 Challenges of Bytecode

Java bytecode is designed for efficient execution by the JVM, not for human readability or ease of analysis. Its dense and complex nature makes it difficult for developers and analysts to intuitively understand the program's logic and flow.

Stack-based Nature

```
1 iload_0
2 iconst_2
3 irem
```

Listing 2.1: Bytecode Example for Computing the Remainder of x Divided by 2

Bytecode operates on a stack-based mechanism. This stack-based nature of bytecode introduces complexity, especially for certain types of analyses.

For example, in a simple operation example, computing $x\%2$ requires three instructions, demonstrating the granularity of bytecode operations. The sequence to perform this calculation, as illustrated in Listing 2.1, involves the following operations.

1. Loading the value of x onto the stack with an `iload` instruction, which pushes x onto the top of the operand stack.

2. Pushing the constant value '2' onto the stack, typically achieved with an `iconst_2` instruction, preparing it for the modulus operation.

3. Executing the modulus operation using the `irem` instruction, which pops the two top values from the stack (x and '2'), computes the remainder of their division, and pushes the result back onto the stack.

This example shows how bytecode decomposes even simple expressions into multiple instructions. This granularity necessitates examining all preceding instructions to fully understand the computation being performed.

For more complex expressions, the number of required instructions increases significantly, potentially becoming arbitrarily large. Consequently, to determine the expression being computed, the analysis must examine all preceding instructions.

Low-Level Orientation

Bytecode's low-level and machine-oriented nature lacks high-level abstractions, which complicates human readability and analysis. The design of bytecode focuses on machine efficiency rather than ease of understanding or analysis, making it challenging for humans to directly interpret or analyze the code without significant effort.

Complex Control Flow

The control flow within bytecode is influenced by its stack-based architecture. Complex expressions in bytecode often require multiple instructions and intermediate results stored on the operand stack. This leads to intricate control flow patterns, especially when expressions involve nested or chained operations. In addition, evaluating these complex expressions can make understanding the control flow of the program particularly difficult, which highlights the challenges of working directly with low-level code representations for program analysis and optimization.

These aspects underscore the challenges associated with directly working with and analyzing bytecode, particularly when compared to more abstracted intermediate representations like Jimple.

```
1 if (x > y) {
2     x = x + 1;
3 }
4 // Following code ..
```

Listing 2.2: A simple if-condition example in Java

Consider the simple Java if-condition shown in Listing 2.2. In Java bytecode, this translates to a sequence of operations involving loading the variables, performing a comparison, and then a conditional jump based on the comparison's outcome.

```
1 iload_1
2 iload_2
3 if_icmple L1
4 iinc 1, 1
5 L1:
6 // Following code ..
```

Listing 2.3: A simple if-condition example in Bytecode

In the corresponding bytecode representation in Listing 2.3, the `iload` instructions load the variables x and y onto the stack. The `if_icmple` instruction pops the two top values from

the stack (x and y), compares them, and jumps to the label `L1` if the condition $x \leq y$ is true, effectively inverting the original $x > y$ condition for the jump logic. The `iinc` instruction is then used to increment the variable x by 1 if the jump is not performed, indicating the condition $x > y$ was true.

This transformation requires careful analysis to understand and trace the original high-level conditional logic. Unlike the clear and structured if-else blocks in Java, bytecode represents these conditions with conditional jump instructions and direct variable manipulation commands like `iinc`, which can be harder to follow and understand, especially in more complex conditional structures with nested if-else statements.

This complexity in representing and understanding conditional logic in bytecode exemplifies the challenges of working directly with low-level code representations for program analysis and optimization.

2.2.2 Advantages of Jimple

Jimple converts bytecode's stack-based operations into a three-address code format, significantly enhancing readability, simplicity, and the capacity for analysis.

Higher Level of Abstraction

Jimple is far easier for a human to understand due to its higher level of abstraction. Operations are explicitly stated, resembling high-level language constructs more closely than bytecode.

Stackless Representation

Jimple eliminates the stack mechanism used in bytecode, replacing it with additional local variables. This transformation makes references explicit rather than implicit, aiding in clarity and analysis [VrCG$^+$].

Three-Address Code

Jimple's use of three-address code simplifies each operation to the form of `x = y op z`, where an assignment involves at most three operands. This format is crucial for readability and makes program analysis and optimization more straightforward [VrCG$^+$].

Compactness

Jimple significantly simplifies the complexity of bytecode by mapping its over 200 distinct operations to only 16 different statement types. This consolidation not only enhances readability but also streamlines the analysis process by reducing the diversity of operations that must be understood and processed.

Typed and Named Variables

Jimple introduces typed and named variables, enhancing the accuracy of analyses. For example, interface invocations can sometimes be statically resolved using the type information provided, facilitating optimizations such as loop unrolling, method inlining, and branch prediction [VrCG$^+$].

Now, consider the representation of the same Java program, as detailed in Listing 2.2, within the Jimple intermediate language. The Jimple representation, shown in Listing 2.4, maintains the structure and logic of the Java source code, but with the aforementioned improvements for analysis.

```
1      int x, y;
2      if (x <= y) goto label1;
3      x = x + 1;
4  label1:
5      // Following code
```

Listing 2.4: Jimple representation of a simple Java if-condition

This example demonstrates Jimple's approach to representation, which is straightforward, stackless, uses three-address code, is compact, and employs typed and named variables. These characteristics significantly contribute to Jimple's suitability for detailed program analysis and optimization.

2.2.3 Impact on Program Analysis

The clarity and structure of Jimple greatly facilitate the analysis of dynamic Java features, such as reflection and dynamic method invocation. By abstracting complex bytecode operations into more understandable structures, Jimple allows for deeper and more effective program analysis, highlighting optimization opportunities more clearly than bytecode [VRGH+].

Jimple's design addresses the challenges posed by bytecode's complexity and lack of readability. By providing a more accessible, stackless, and structured format, Jimple enables more effective program analysis and optimization.

2.3 Property Graphs

Property graphs represent a versatile and powerful model for organizing and analyzing complex data relationships. They have evolved over time to become a fundamental tool in various domains, from social network analysis to complex system modeling [ABD+]. The essence of a Property Graph lies in its ability to not only represent entities and their relationships but also to enrich these elements with detailed attributes.

2.3.1 Core Concepts and Structure

At its core, a property graph consists of nodes, edges, properties, and labels. Nodes typically represent entities or objects, while edges denote the relationships or interactions between these entities. Both nodes and edges can have properties associated with them, which are key-value pairs that provide additional information about the entity or relationship. Labels, assigned to both nodes and edges, categorize them into distinct types or classes. For example, node labels might identify whether an entity is a *Person*, *Organization*, or *Event*, and edge labels might specify the nature of the relationships, such as *FriendOf*, *EmployedBy*, or *Attended*. Property graphs are multigraphs, i. e., the same pair of nodes can have several edges between them. Additionally, property graphs can be enriched with schemas that provide a predefined structure for the properties and labels associated with nodes and edges, thereby further organizing and defining the graph's data model.

These features make property graphs particularly useful for modeling real-world scenarios where entities have multiple attributes and can be connected in various ways [HBC+]. The support for multiple edges between the same pair of nodes enables the representation of more complex relationships and connections in the data. Additionally, the ability to operate without

a strict schema, yet allowing for its inclusion when necessary, provides flexibility that makes it suitable for a wide range of applications and use cases.

To illustrate the practical utility of property graphs, consider the example of a social network platform like Facebook, where users are represented as vertices and friendships as edges. Each user (node) can have properties such as name, age, and interests, while the friendships (edges) may include properties like the date the friendship was established or the strength of the connection. Figure 2.1 shows a potential property graph representation that captures this scenario and illustrates the model's ability to encapsulate complex relational data in a structured yet flexible way.

Alice, Bob, Carol, and David are depicted as nodes in the graph, each with their respective attributes and the edges between them represent different types of social interactions or relationships. For instance, Alice follows both Bob and Carol, as indicated by the edges labeled "follows" connecting Alice to Bob and Carol. Furthermore, the graph captures additional interactions such as liking posts. For instance, Alice has liked a post by Bob (edge labeled "liked post"), and similar interactions are depicted between other users in the network. The graph also demonstrates the concept of multiple edges between the same pair of nodes. For example, Bob and Carol have both followed each other, resulting in two edges labeled "follows" between them. Moreover, the graph showcases edges with the same label but different properties. For instance, Bob and Carol have both liked posts by Alice, each with a distinct post identifier.

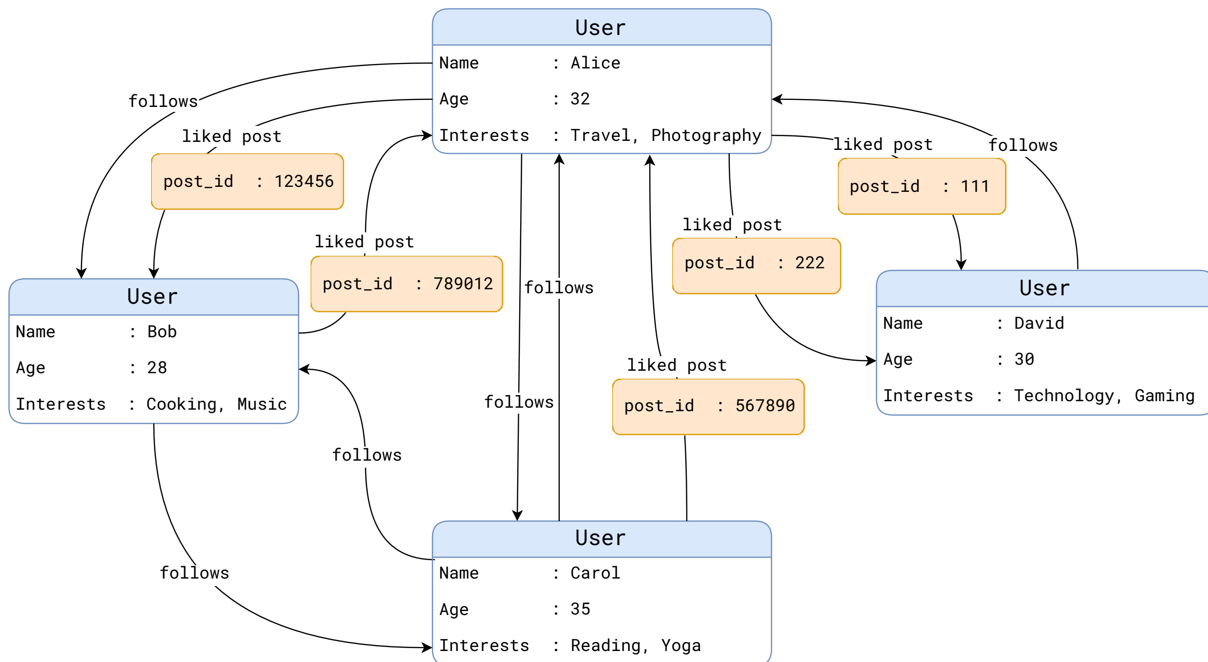

Figure 2.1: Property graph illustrating social network interactions.

2.3.2 Querying Property Graphs

The structure of property graphs lends itself to powerful querying and analysis capabilities. Graph databases that support property graphs often provide query languages like Cypher by Neo4j, which enable intricate queries based on both the structure of the graph and the properties of nodes and edges [Neo24]. This feature allows for more sophisticated analyses, like finding the shortest path considering not just the distance but also other factors like traffic conditions or route safety [AAB+].

2.3.3 Formalizing Property Graphs

Currently, property graphs do not have a single, universally agreed-upon formal definition but rather are understood through a commonly accepted conceptual framework. Efforts toward standardizing the concept and query languages for property graphs include the development of openCypher by Neo4j and the ongoing work by ISO/IEC to standardize *GQL* (Graph Query Language) as a query language for property graphs [GJK+][Int24]. These initiatives aim to provide more formal definitions and interoperability standards for property graph technologies.

In the context of this work, property graphs will serve as the basis for different program representations. For that purpose, we will provide a formal definition of property graphs tailored to our specific needs.

Definition 2.1 (Property Graph)**.** *A property graph G is defined as a tuple $G = (V, E, P_v, P_e, \phi_v, \phi_e, L_v, L_e, \lambda_v, \lambda_e)$. V denotes the set of vertices in the graph. E denotes the set of directed edges, with each edge being a pair (u, v) where $u, v \in V$, indicating a directed edge from vertex u to vertex v. P_v and P_e represent sets of property keys associated with vertices and edges, each property being a pair $(key, value) \in K \times V$, where K is the domain of keys and V is the domain of values. The functions $\phi_v : V \to 2^{P_v}$ and $\phi_e : E \to 2^{P_e}$ map vertices and edges to a set of their properties, respectively. L_v and L_e indicate separate sets of labels for vertices and edges, with $\lambda_v : V \to L_v$ and $\lambda_e : E \to L_e$ as the functions assigning these labels.*

Applying the given definition to our social network example in Figure 2.1, we identify the components of the tuple G as follows. The set of vertices V includes $\{$Alice, Bob, Carol, David$\}$, representing individuals in the network. The set of directed edges E consists of tuples such as $(Alice, Bob)$, each indicating a directed relationship from one vertex to another. Properties associated with vertices (P_v) include keys like "Age" and "Interests", with corresponding values for each individual, for instance, $(\text{"Age"}, 32)$ for Alice. Similarly, properties associated with edges (P_e) include keys relevant to the interactions, such as "post_id" for edges labeled "liked post". The functions ϕ_v and ϕ_e map these vertices and edges to their associated properties, respectively. Labels for vertices and edges (L_v and L_e, respectively) further classify the graph elements, with λ_v and λ_e assigning these labels. For example, λ_e would map an edge from Alice to Bob with the action "follows" to its respective label.

There are simpler formulations of property graphs in the literature, such as the definition by Yamaguchi et al.[YGAR]. We opted for this particular extended definition, where L_v and L_e are uniquely addressed for labeling nodes and edges, respectively, due to the need to capture the notion of different labeling domains. This distinction is crucial in our context since we use labels to convey the notion of object or entity types. We assume that the set of types relevant to edges generally differs from that for vertices.

3 Code Property Graphs

In the following, we delve into the essence and functionality of code property graphs. Section 3.1 breaks down their structure and components, while Section 3.2 illustrates their relevance and applications through practical examples.

3.1 Structure and Components

CPGs present an innovative framework for source code analysis by synthesizing various dimensions of code representation into a cohesive structure. This concept, initially introduced by Yamaguchi et al. [YGAR], integrates key aspects of code analysis—*Abstract Syntax Tree* (AST), *Control Flow Graph* (CFG), and *Program Dependence Graph* (PDG)—directly into a unified graph-based format.

3.1.1 Abstract Syntax Trees

ASTs form the backbone of the CPG, representing the syntactic structure of the source code. Each node in an AST corresponds to a construct in the source code, such as variables, operators, method calls, and control structures. The hierarchical nature of ASTs mirrors the nested structure of programming constructs, providing a detailed view of the code's syntax.

Formal Representation of ASTs as Property Graphs

Given the formal definition of a property graph provided in Definition 2.1, we can represent an AST within this framework as follows.

- **Vertices** (V): Each vertex corresponds to a syntactic construct in the source code, such as variables, operators, method calls, and control structures. These syntactic constructs are the nodes (V) of the AST.
- **Edges** (E): Directed edges represent the syntactic hierarchy and relationships between nodes, such as parent-child relationships in the tree structure, included in E.
- **Properties of Vertices** (P_v) and **Edges** (P_e): Nodes (vertices) carry properties (P_v) like positional information (e. g.statement position). While basic AST edges primarily signify structural connections without additional attributes, enhanced ASTs might include properties (P_e) on edges to capture more nuanced relationships, such as the sequence of arguments in a function call.

- **Property Functions** (ϕ_v, ϕ_e): The function $\phi_v : V \to 2^{P_v}$ maps each node to its set of properties and extends the AST with detailed contextual information. Similarly, $\phi_e : E \to 2^{P_e}$ would apply to edges if they carry properties.
- **Labels** (L_v, L_e): Labels are used to explicitly denote the type of syntactic constructs and relationships. Each node is assigned a label from L_v (e. g., `VariableDeclaration`, `Assignment`, `MethodCall`), signifying its role in the source code. Edges are labeled from L_e to denote the type of syntactic relation (e. g., `ParentChild`, `ArgumentSequence`).
- **Labeling Functions** (λ_v, λ_e): The function $\lambda_v : V \to L_v$ assigns the appropriate syntactic type label to each node, while $\lambda_e : E \to L_e$ labels each edge with its relationship type.

3.1.2 Control Flow Graphs

CFGs are integrated into CPGs to represent the flow of control in a program. They illustrate how execution progresses from one block of code to another, highlighting the paths that might be taken during execution. In CFGs, nodes can represent either basic blocks — a sequence of consecutive statements or instructions with only one entry point and one exit point — or single statements, and edges represent the flow of control from one block to another. This graphical representation provides a detailed view of all possible execution paths and is instrumental in identifying areas like loops, conditional branches, and exit points.

Formal Representation of CFGs as Property Graphs

In alignment with our formal definition of a property graph in Definition 2.1, we detail the representation of a Control Flow Graph (CFG) as follows.

- **Vertices** (V): Each vertex corresponds to a single statement in the source code, and collectively, these statements form the nodes (V) of the CFG, representing the granular execution points within a program.
- **Edges** (E): The directed edges between the vertices represent the control flow from one statement to another and capture the possible execution paths through the program. These edges are contained in E and describe how the execution can progress or branch depending on the conditions fulfilled at runtime.
- **Properties of Vertices** (P_v) and **Edges** (P_e): Each node can be associated with properties (P_v) such as statement position. Edges might carry properties (P_e) that describe the conditions under which control flow transitions occur, like boolean conditions for branches.
- **Property Functions** (ϕ_v, ϕ_e): The function $\phi_v : V \to 2^{P_v}$ maps each statement to its relevant properties, enriching the CFG with contextual details. Similarly, $\phi_e : E \to 2^{P_e}$ maps edges to their properties, if any, enhancing the understanding of control flow dynamics.
- **Labels** (L_v, L_e): Labels denote the type of control flow constructs represented by nodes and edges. Nodes (V) receive labels from L_v indicating their statement type (e. g., `Assignment`, `IfStatement`, `Loop`), while edges (E) are labeled from L_e to reflect the nature of control flow (e. g., `TrueBranch`, `FalseBranch`, `LoopBack`).
- **Labeling Functions** (λ_v, λ_e): $\lambda_v : V \to L_v$ assigns a specific type label to each statement, which clarifies its role in the program logic. $\lambda_e : E \to L_e$ provides the edges with labels that indicate the type of control flow relationship they represent, thereby offering insights into the program's execution pathways.

3.1.3 Program Dependence Graphs

PDGs, introduced by Ferrante et al., capture the dependencies between different parts of the code [FOW]. They illustrate how data flows through the program and how different operations depend on each other. This is crucial for understanding the impact of changes in one part of the code on other parts, and for identifying potential side effects. PDGs consist of two primary subgraphs: the *Control Dependence Graph* (CDG) and the *Data Dependence Graph* (DDG).

Control Dependence Graphs

The CDG records control dependencies within the program. Control dependencies are relationships that indicate that the execution of a piece of code depends on the outcome of a particular condition or decision point in another piece of code. In a CDG, the nodes represent program statements or code blocks, while the edges represent control dependencies. For example, in an if-else statement, the code blocks within the if and else sections are control-dependent on the conditional expression of the if statement. This means that the execution of these blocks depends on the result of the evaluation of the conditional expression.

Data Dependence Graph

The DDG illustrates data dependencies within the program. Data dependencies occur when a piece of code depends on data produced by another part of the program. In the DDG, nodes again represent program statements or operations, and edges signify the data dependencies between these operations.

Formal Representation of CDGs and DDGs as Property Graphs

Following the formal definition of a property graph provided in Defintion 2.1, we describe the representation of a CDG as follows.

- **Vertices** (V): Each vertex corresponds to a single statement or control structure in the source code, where collectively, these elements form the nodes (V) of the CDG. This representation emphasizes the points in the program where the execution path diverges based on conditional statements.
- **Edges** (E): Directed edges between vertices represent control dependencies, indicating that the execution of one statement is conditionally dependent on another. These dependencies form the edges (E) of the CDG and outline the influence of the decision points on the execution sequence.
- **Properties of Vertices** (P_v) and **Edges** (P_e): Nodes may carry properties (P_v) such as the condition leading to a control dependency or the statement position. Edges, characterizing the nature of control dependencies, could have properties (P_e) detailing the specific conditions under which control is passed from one node to another.
- **Property Functions** (ϕ_v, ϕ_e): The function $\phi_v : V \to 2^{P_v}$ maps each node to its set of properties. Similarly, $\phi_e : E \to 2^{P_e}$ assigns properties to edges to clarify the conditions of dependency.
- **Labels** (L_v, L_e): Nodes are labeled with types from L_v that reflect their role in control dependencies (e. g. `Conditional`, `LoopStart`, `MergePoint`). Edges are labeled from L_e to indicate the type of control dependency (e. g. `TrueDependency`, `FalseDependency`, `LoopDependency`).

- **Labeling Functions** (λ_v, λ_e): Through $\lambda_v : V \to L_v$, each control statement or structure is assigned a label identifying its type. $\lambda_e : E \to L_e$ categorizes the control dependencies by illustrating how execution decisions impact the program's flow.

Finally, for the DDGs, the vertices (V) and their properties (P_v) are defined similarly to the CDGs. The special features of the DDG are described below.

- **Edges** (E): In DDGs, directed edges are particularly significant, representing data dependencies. These dependencies indicate that the execution or value of one statement directly influences another. This relationship is crucial for understanding data flow and potential side effects in program execution.
- **Properties of Edges** (P_e): The properties associated with edges in DDGs might include information about the data dependency (e. g., the dependency variables).
- **Labels** (L_e): Edges in DDGs are labeled to reflect the specific kind of data dependency they represent (e. g., `ReachingDef`.).

3.1.4 Constructing the CPG

The construction of a CPG involves the integration of multiple subgraphs—namely, the AST, CFG, CDG, and DDG—each representing different facets of program semantics. The unification process is formalized as follows, adhering to the principles of property graph $G = (V, E, P_v, P_e, \phi_v, \phi_e, L_v, L_e, \lambda_v, \lambda_e)$ defined in Definition 2.1:

1. **Vertex Integration** (V): The set of vertices in the CPG, V_{CPG}, is the union of vertices from the AST, CFG, CDG, and DDG. Each vertex represents a unique construct or statement in the source code, ensuring no duplication:

$$V_{\text{CPG}} = V_{\text{AST}} \cup V_{\text{CFG}} \cup V_{\text{CDG}} \cup V_{\text{DDG}}$$

2. **Edge Integration** (E): The set of edges in the CPG, E_{CPG}, combines the edges from the AST, CFG, CDG, and DDG, preserving their original semantics while avoiding overlap:

$$E_{\text{CPG}} = E_{\text{AST}} \cup E_{\text{CFG}} \cup E_{\text{CDG}} \cup E_{\text{DDG}}$$

3. **Property Aggregation** (P_v, P_e): Properties of vertices and edges $(P_{v_{\text{CPG}}}, P_{e_{\text{CPG}}})$ are aggregated from their counterparts in the subgraphs. This aggregation preserves the original context and adds a layer of integration for analysis:

$$\phi_{v_{\text{CPG}}} : V_{\text{CPG}} \to 2^{P_v}\text{CPG}, \quad \phi_{e_{\text{CPG}}} : E_{\text{CPG}} \to 2^{P_e}\text{CPG}$$

4. **Label Harmonization** (L_v, L_e): The labeling functions $\lambda_{v_{\text{CPG}}}$ and $\lambda_{e_{\text{CPG}}}$ ensure that each vertex and edge in the CPG is assigned a label that reflects its role across the combined semantics of the subgraphs. This step is crucial for maintaining the integrity and interpretability of the merged graph:

$$\lambda_{v_{\text{CPG}}} : V_{\text{CPG}} \to L_{v_{\text{CPG}}}, \quad \lambda_{e_{\text{CPG}}} : E_{\text{CPG}} \to L_{e_{\text{CPG}}}$$

5. **Semantic Preservation**: Throughout the integration process, it is imperative to preserve the semantic relationships inherent in each subgraph. This involves careful mapping of control and data dependencies, ensuring that the unified graph accurately represents the program's behavior and structure.

This formal methodology outlines the comprehensive approach to constructing a CPG by integrating distinct but related subgraphs, thereby facilitating a multidimensional analysis of source code through a unified, semantically rich graph model.

While, in practice, entities such as statements inherently carry the same properties across different subgraphs, we adopt a formal approach to property integration. This strategy is implemented to ensure comprehensiveness and to avoid any potential confusion that might arise from implicit assumptions about property uniformity. By formally integrating properties, we avoid inadvertently omitting nuanced details and ensure that each entity is represented in the CPG with the greatest possible completeness and accuracy, reflecting the full range of its characteristics as described in the individual subgraphs.

3.2 Relevance and Applications

CPGs stand out due to their ability to encapsulate multiple facets of source code. By combining the syntactic structure provided by ASTs, the execution flow depicted in CFGs, and the interdependencies highlighted in PDGs, CPGs offer a comprehensive representation of both the static and dynamic characteristics of code. This multifaceted approach enables a more holistic analysis, crucial for a range of applications from security vulnerability detection to code quality assessment.

3.2.1 Vulnerability Characterization

Whereas traditional code analysis methods might focus on a single aspect of the code, CPGs provide a layered and interconnected view. This allows for an enhanced program analysis, since CPGs reveal additional insights into how different components of the code interact and impact each other.

Consider, for instance, the *Time-of-check to time-of-use* (TOCTTOU) vulnerability. The TOCTTOU vulnerability exemplifies a scenario where the multifaceted analysis capability of CPGs becomes indispensable.

```
1 public class FileAccess {
2   public void handleFile(String filePath) {
3     File file = new File(filePath);
4
5     // Time-of-Check
6     if (isNotSymbolicLink(file)) {
7       // Time-of-Use: Potential TOCTTOU Vulnerability
8       processFile(file);
9     }
10  }
11
12  private boolean isNotSymbolicLink(File file) {
13    return !Files.isSymbolicLink(file.toPath());
14  }
15
16  private void processFile(File file) {
17    // Perform some operation on the file
18  }
19 }
```

Listing 3.1: Example of the TOCTTOU Vulnerability.

In a TOCTTOU scenario, the victim first ensures that a particular file is not a symbolic link (symlink). Once confirmed, they believe it is safe to work on the file. However, in the intermediary time between the verification and the subsequent action, an attacker stealthily replaces the original file with a symlink. Unaware of this change, the victim then accesses the file, assuming that it is not a symlink. Such a vulnerability poses significant risks. For example, if a password file is mapped to the symlink, this can lead to the unintentional disclosure of sensitive data.

This type of vulnerability highlights the importance of considering multiple facets of code behavior—sensitive operations, type usage, attacker control, and sanitization measures——in conjunction [YGAR]. CPGs, by encapsulating these aspects, enable the detection of such complex vulnerability patterns that would otherwise evade more traditional, single-faceted analysis techniques.

3.2.2 Vulnerability Detection

The methodology of Yamaguchi et al. for characterizing vulnerabilities using CPGs represents a paradigm shift in static code analysis by transitioning to a model that accounts for the entirety of code structural and semantic components [YGAR]. Analyses that focus on isolated aspects of the code fall short of identifying complex vulnerabilities, such as TOCTTOU, where specific preconditions must be met. While some tools attempt to remedy that by assuming that all preconditions are always satisfied, this often leads to a high rate of false positives [SNAA]. CPGs, by providing a multi-dimensional view of the code, pave the way for more accurate and nuanced vulnerability detection, reducing the reliance on broad assumptions and subsequent manual filtering of results.

4 Contribution

In the following, we present the cornerstone contributions of our research. Section 4.1 lays the foundation by outlining the fundamental principles underlying the construction of CPGs universally. Section 4.2 examines our methodology and the rationale behind the design decisions of JimNode. Lastly, section 4.3 discusses the practical implementation of our theoretical framework and design principles.

4.1 Theoretical Foundations

The construction of a CPG entails constructing and assembling its subgraphs. This section discusses the theoretical underpinnings required for constructing the CPG subgraphs. Since we were only concerned with the construction of the CDG and DDG based on an already established CFG, we will focus exclusively on the construction of CDG and DDG.

4.1.1 Control Dependence

The principles of dominance and post-dominance within a CFG are pivotal for building the CDG. In the following, we explore how these foundational concepts form the basis for its construction.

Dominance in Control Flow Analysis

The foundational concept of dominance in control flow analysis was first systematically articulated in the seminal work of Prosser [Pro]. Dominance within a CFG is defined by the principle that a node is considered dominant over another if every path to the latter must traverse through the former. This concept is integral to understanding the flow of control in programs and is particularly significant in compiler optimization and control flow analysis.

Dominance analysis is employed to systematically dissect the control paths within a CFG. In the broader scope of program analysis, dominance is a key factor in accurately determining points of control convergence and divergence, which are critical in understanding program behavior and structure [CFR+]. In the context of our research, we leverage the concept of dominance to identify the control dependencies between the nodes of the CFG and consequently construct the CDG. The choice of dominance analysis for the construction of the CDG due to its robustness in mapping control flow dependencies as well as its operational efficiency [CHK]. Dominance between two nodes in a CFG is defined as follows.

Definition 4.1 (Dominance)**.** *A node A in a CFG is said to dominate another node B if every path from the initial node to B passes through A. This is denoted as A dom B.*

$$D(A) = \{B \in CFG \,|\, \forall \, paths \, P \, from\ the\ start\ node\ to\ B,\, A \in P\}$$

The dominators of a node are thus the nodes that are present in every path from the start node to this node. Similarly to the concept of dominance, we can introduce the concept of post-dominance as follows.

Definition 4.2 (Post-Dominance)**.** *A node A in a CFG is said to post-dominate another node B if every path from B to the exit node passes through A. This is denoted as A pdom B.*

$$PD(A) = \{B \in CFG \,|\, \forall \, paths \, P \, from\ B\ to\ the\ exit\ node,\, A \in P\}$$

Dominance Frontiers

While dominance indicates the nodes that influence the execution path of other nodes, control dependencies are more specific. They exist between two nodes A and B if the execution of B depends on a decision made at A. To derive control dependencies from the established dominance relationships, we introduce the concept of dominance frontiers.

Definition 4.3 (Dominance Frontier)**.** *The dominance frontier of a node A is the set of all nodes B for which A does not dominate B, but A dominates an immediate predecessor of B in the CFG. In other words, the dominance frontier represents those nodes that are just outside the reach of A's dominance. This can be denoted as follows.*

$$DF(A) = \{B \in CFG \,|\, \neg(A\ dom\ B)\ and\ \exists\, C\ pred\ B,\ A\ dom\ C\}$$

Similarly, we define post-dominance frontiers as follows.

Definition 4.4 (Post-Dominance Frontier)**.** *The post-dominance frontier of a node A is the set of all nodes B for which A does not post-dominate B, but A post-dominates an immediate successor of B in the CFG. In other words, the post-dominance frontier represents those nodes that are just outside the reach of A's post-dominance. This can be denoted as follows.*

$$PDF(A) = \{B \in CFG \,|\, \neg(A\ pdom\ B)\ and\ \exists\, C\ succ\ B,\ A\ pdom\ C\}$$

Dominance frontiers and post-dominance frontiers are critical in identifying the points in the program where control paths diverge and converge, respectively. They are particularly important in the construction of CDGs as they help to determine where control dependencies arise in the presence of branching structures in the program. Similarly, we define post-dominance frontiers as follows.

Dominance Trees

Building on the concept of dominance, where the dominators of a node include all nodes present in every path from the start node to the given node, we transition to a more specific aspect known as immediate dominance. Immediate dominance refines the general notion of dominance by identifying the closest dominator to a node within the CFG. Specifically, a node A is considered the immediate dominator of node B if it directly precedes B in the hierarchy of dominators, with no intermediate nodes exhibiting dominance over B. This implies that A is the last node through which all paths from the start node to B must pass, making it a pivotal control point in the flow of the program. Immediate dominance thus narrows down the broad spectrum of

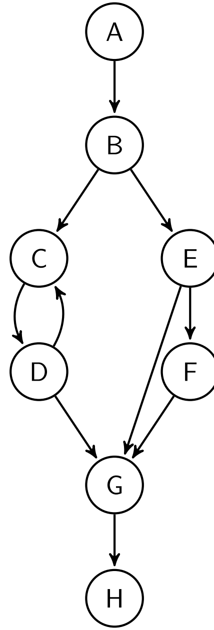

Figure 4.1: Example of a CFG

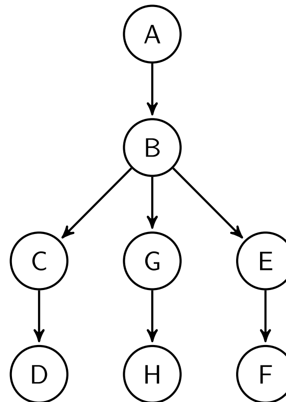

Figure 4.2: Dominator tree for Figure 4.1

dominators to highlight the most direct influence on a node's execution path. Consider the CFG in Figure 4.1 with the entry node A. Since node A is the entry point, it dominates all other nodes. Similarly, node B dominates all nodes in the graph, except for A. Node C dominates only itself and node D, but it does not dominate node G due to the presence of paths $ABEFG$ and $ABEG$. Likewise, node E dominates only itself and node F. Node G dominates only itself and node H. Nodes D, F, and H dominate only themselves. Based on this, we can identify the immediate dominators and construct the dominance tree shown in Figure 4.2.

Algorithm for Finding Dominators

In the realm of compiler design and program analysis, determining the dominators within a CFG is a pivotal task. This process is foundational for constructing the dominator tree, which is essential for optimizing code, identifying loops, and enhancing program analysis techniques [Ram]. There are several algorithms designed to find dominators in a CFG, each varying in complexity and efficiency.

Algorithm 1 outlines the the naïve approach to finding dominators in a CFG. It involves

Algorithm 1 Naive Approach for Calculating Dominators

Require: Control flow graph CFG with start node S
Ensure: Dominator set Dom for each node in CFG

```
1:  for each node N in CFG do
2:      Dom[N] ← {S}                          ▷ Initialize dominator set with start node
3:      if N ≠ S then
4:          Dom[N] ← Dom[N] ∪ {N}             ▷ A node dominates itself
5:          for each node M in CFG excluding S and N do
6:              isDominator ← true
7:              for each path P from S to N do
8:                  if M ∉ P then
9:                      isDominator ← false
10:                     break
11:                 end if
12:             end for
13:             if isDominator then
14:                 Dom[N] ← Dom[N] ∪ {M}     ▷ Add M to N's dominators
15:             end if
16:         end for
17:     end if
18: end for
```

a simple but exhaustive examination of all paths within the graph. For a given node B, this method checks every possible path from the entry node of the CFG to B, ensuring that a candidate dominator node A is present in all such paths. If A is found in every path leading to B, A is considered a dominator of B.

While conceptually straightforward, the naïve method is computationally intensive, especially for complex CFGs with a large number of nodes and intricate control flows. The primary limitation of this approach lies in its requirement to examine all paths within the CFG, which can grow exponentially with the size of the graph. This exhaustive path analysis makes the naïve method less practical for use in real-world applications, where CFGs can be extensive and complex.

One of the most common approaches in dominance analysis is the Cooper-Harvey-Kennedy algorithm by Cooper et al. [CHK]. It is known for its simplicity and efficiency.In the JimNode implementation, the CHK algorithm was utilized to compute the post-dominance frontiers.

Building the CDGs from Post-Dominance Frontier

In their foundational work on PDGs, Ferrante et al. define control dependence based on the concept of dominance. A node Y becomes control dependent on node X if and only if there is a path from X to Y, with all intermediate nodes post-dominated by Y without X being post-dominated by Y. This relationship is represented in the CDG edges connecting nodes to their respective post-dominators [FOW].

The conventional computation of control dependence utilizes two primary methods based on dominance principles. Initially, one can construct a post-dominator tree, establishing control dependencies by directly connecting nodes to their post-dominators.

The alternative approach, proposed by Cytron et al., utilizes post-dominance frontiers to determine control dependencies [CFR$^+$].

Our work in JimNodedraws upon these established methods, particularly the insights from

Cytron et al.. To compute control dependence relations, we reverse the CFG, incorporate a virtual start node, and then compute the dominance frontiers within the reversed graph. This adaptation, which is in line with common practices, leverages the idea that dominators in the reversed CFG correspond to post-dominators in the original graph.

4.1.2 Data Dependence

Data dependence is a fundamental concept in data flow analysis that plays a crucial role in understanding the behavior of programs. It refers to the relationships and constraints that exist between different data elements within a program. These dependencies determine how data values propagate through the program and affect the control and flow of execution. In the following, we introduce the basic principles and concepts underlying data dependence.

Data Flow Analysis

Data flow analysis is one of the main approaches to program analysis. It emerged as a systematic method to analyze the flow of data across the control flow of a program. The technique is instrumental in identifying how data values are defined, used, and propagated through a program, with applications ranging from dead code elimination to register allocation and constant propagation [NNH] [AC].

One of the fundamental concepts in data flow analysis is the notion of a "*data flow graph*," which represents the flow of information within a program. It illustrates how data values are generated, used, and modified across different parts of the code.

Data flow analysis operates on the CFG. The primary goal of data flow analysis is to deduce information about the possible set of values calculated at various points in a program and how these values "flow" from one part of the program to another.

At the heart of data flow analysis lies the concept of solving data flow equations that abstractly represent the flow of information across the program's control paths. At the heart of data flow analysis lies the concept of solving data flow equations that abstractly represent the flow of information across the program's control paths. Two fundamental analyses—forward and backward analysis—serve as the basis for various optimization strategies.

- **Forward Analysis:** Propagates information from the entry point towards the exit points of a CFG. Examples include constant propagation and reaching definitions.
- **Backward Analysis:** Information flows from exit points back to the entry. Examples include live variable analysis and dead code elimination.

These analyses rely on the iterative solution of data flow equations until a fixed point is reached, where no new information is generated upon further iteration.

Tabular representations, or data flow tables, play a pivotal role in visualizing and solving data flow equations. Each row in a data flow table typically corresponds to a basic block or a statement in the program, while columns represent the data flow facts being analyzed (e. g., definitions reaching a point, live variables at a point).

Two principal tables emerge in the analysis:

1. **Gen and Kill Sets:** The "Gen" (generate) set identifies the data flow facts generated by executing a basic block, and the "Kill" set identifies facts that are no longer valid after the block's execution.
2. **In and Out Sets:** For each basic block, the "In" set represents the data flow facts that hold at the entry point of the block, and the "Out" set contains facts that hold at the exit.

By iteratively updating these sets based on the relationships defined by the data flow equations——considering the CFG's structure and the gen/kill effects of each block—the analysis can reach a state where the in and out sets stabilize, providing a comprehensive view of the program's data flow properties.

In the following, we will apply the above principles to conduct the reaching definitions analysis.

Reaching Definitions Analysis

```
1     x = 5;
2     y = x + 1;
3     x = y + 2;
4     if x > y goto label6;
5     y = x + 3;
6 label6:
7     x = y + 4;
```

Listing 4.1: Example Program Segment for Reaching Definitions Analysis

Reaching definitions analysis is a fundamental data flow analysis technique used to determine which definitions of variables "reach" a certain point in the program without being overwritten. This analysis helps identify potential redundancies and optimizations by tracking where variable values come from and how they propagate through the program's control flow.

Consider the Jimple program in Listing 4.1. In reaching definitions analysis, we are interested in identifying which assignments (definitions) of variables are available (i. e., reach) at each point in the program.

The analysis process involves creating tables that summarize the "gen" (generate) and "kill" sets for each statement, as well as the "in" and "out" sets that represent definitions reaching the entry and exit of each statement, respectively.

Without going into specific algorithmic details, which might closely resemble existing literature, a generalized approach to solving the reaching definitions problem involves iteratively updating the "in" and "out" sets for each node in the CFG based on the "gen" and "kill" sets until no more changes occur [Kil].

For instance, at line 1, "gen" is x=5 because it generates a definition of "x", and "kill" would include any prior definitions of "x" that are overwritten by this new definition. The "in" and "out" sets for each line are computed by considering the flow of definitions through the program's CFG.

Line	Gen	Kill	In	Out
1	{x=5}	{}	{}	{x=5}
2	{y=x+1}	{}	{x=5}	{x=5, y=x+1}
3	{x=y+2}	{x=5}	{x=5, y=x+1}	{x=y+2, y=x+1}
4	{}	{}	{x=y+2, y=x+1}	{x=y+2, y=x+1}
5	{y=x+3}	{y=x+1}	{x=y+2}	{x=y+2, y=x+3}
6	{x=y+4}	{x=y+2}	{y=x+3}	{x=y+4}

Table 4.1: Reaching Definitions Analysis for the Program Segment in Listing 4.1

Table 4.1 demonstrates how variables are defined and used throughout the program. Typically, defining the specifics of an analysis involves detailed discussions on several theoretical aspects: the domain of analysis, the direction (forward or backward), the meet operator, transfer functions, and the initial and boundary conditions. However, due to the comprehensive coverage of these foundational concepts in seminal texts, and in order to maintain conciseness and clarity in our discussion, we have omitted a detailed exposition of these aspects. For comprehensive insights into these foundational concepts, the seminal texts by Aho, Sethi, and Ullman, and by Muchnick, are highly recommended [ASU86][Muc].

Building the DDG

Transitioning from reaching definitions analysis to constructing the DDG is a direct process. Essentially, the DDG is a visual representation of the dependencies identified through reaching definitions analysis, where each node in the graph corresponds to a statement in the program, and edges represent the dependencies between these statements based on the data flow.

In reaching definitions analysis, we identify where and how variables are defined (generated) and where these definitions are "reached' ' or used within the program. This analysis uncovers the dependencies between the variable definitions and their subsequent uses and forms the basis for the DDG. Constructing the DDG involves constructing the nodes and edges as specified below.

- **Nodes:** Each statement in the program that defines or uses a variable becomes a node in the DDG.
- **Edges:** For each use of a variable, an edge is drawn from the statement that defines the variable (if that definition reaches the use) to the statement in which the variable is used. This edge signifies a data dependency, which indicates that executing the using instruction depends on the value defined by the defining instruction.

4.2 Methodology and Model Design

Unlike traditional language-agnostic models, like Joern, which often struggle to capture the intricacies of specific programming languages and software systems, JimNode adopts a tailored approach that aligns closely with the underlying semantics and structures of the code. By incorporating language-specific knowledge into its design, JimNode offers a more nuanced and accurate representation of programs.

4.2.1 Design and Model Structure of JimNode

At the heart of JimNode is a design philosophy that aims to represent the concrete syntax and semantic structure of Jimple code within the framework of a CPG. This alignment allows the representation of the code to maintain adherence to the original Jimple constructs and enables a more accurate and deeper analysis. It avoids the generic abstractions often used by language-specific tools, which can dilute the specificity and context of the language constructs.

The structure of JimNode's CPG, as depicted in Figure 4.3, is a manifestation of this approach. It comprises specialized nodes and edges that correspond directly to the elements of Jimple code, thus preserving its unique syntax and semantics. This detailed mapping ensures that the intricacies of Jimple are not lost in translation to a generic model.

The `StmtPropertyGraphNode` and `ValuePropertyGraphNode` are pivotal to this design. They are not just generic placeholders but are extended to represent specific Jimple types.

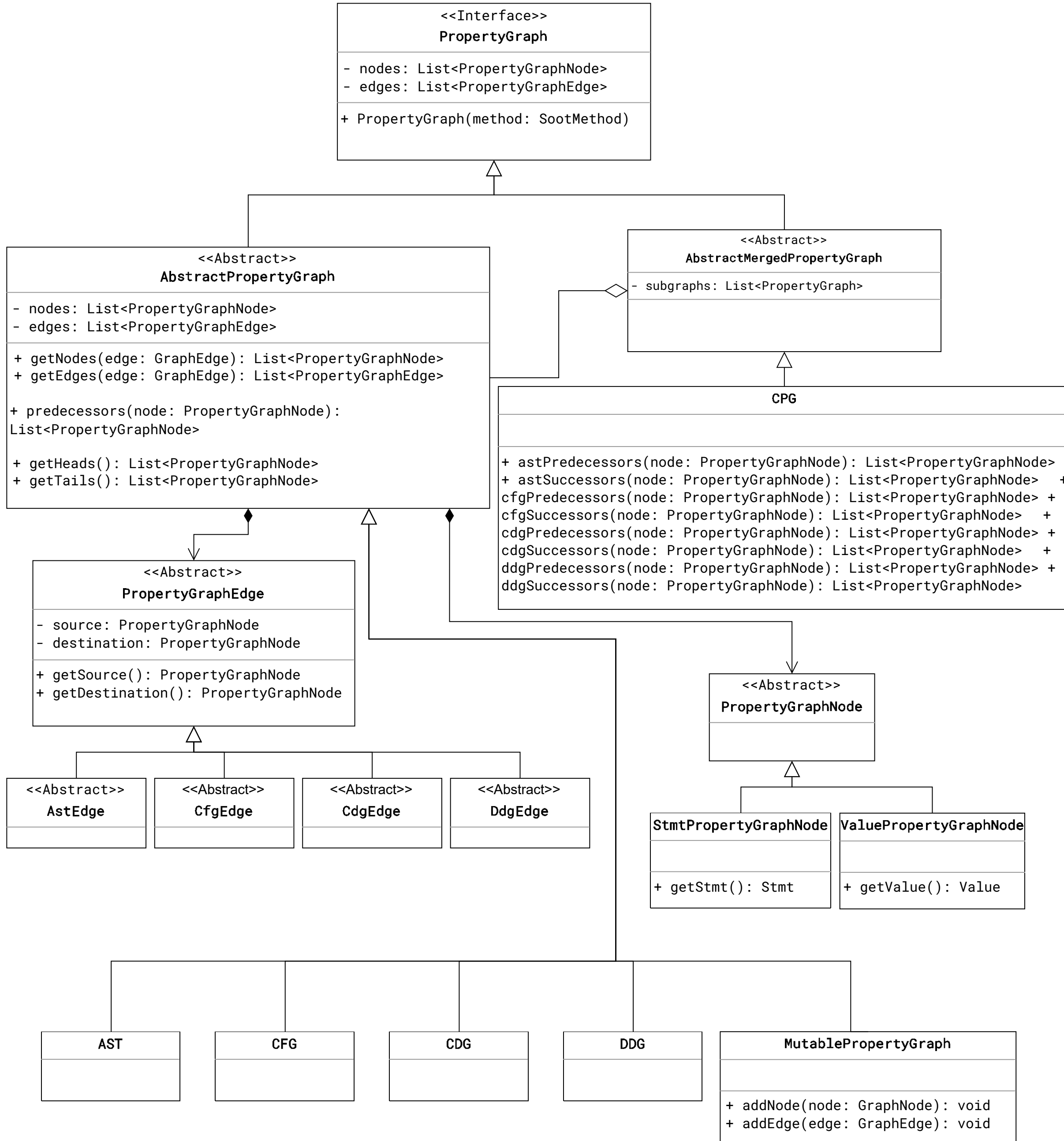

Figure 4.3: Basic design of the CPG.

This means that the CPG not only contains the structure of the code but also rich, typed information about each statement and value.

While Joern provides representations for elements such as local variables and array types, these are often defined in a generic manner; for instance, types are denoted as strings rather than specific objects, leading to a representation that is flexible but may lack the precision of a language-specific approach. This level of detail is not typically captured in a language-agnostic model like Joern's, which abstracts away such specifics into a unified representation, potentially obscuring the language's unique features.

By adopting this approach, JimNode facilitates a more granular level of analysis. Researchers and developers can query the graph directly for specific Jimple constructs, which

greatly simplifies the process of program analysis. Such a direct mapping from the Jimple code to the CPG also aids in reducing the cognitive load for the user, as there is a clear correspondence between the code being analyzed and its representation within the graph.

4.2.2 Advantages Over Language-Agnostic Models

The tailored approach of JimNode presents several advantages over language-agnostic models such as Joern. Following, we enumerate the key benefits, which underscore the value brought by a model intricately aligned with the semantics of Jimple code.

MA1 **Semantic Richness:** The JimNode model leverages the specific syntactic and semantic features of Jimple, offering a rich set of constructs that are directly aligned with the language's own abstractions. This level of semantic richness enables more precise static analysis, vulnerability detection, and program understanding.

MA2 **Ease of Use:** By mirroring the actual constructs of Jimple, the learning curve for new users is significantly reduced. Analysts can intuitively navigate the CPG, as it closely resembles the source code structure they are familiar with, leading to a more efficient analysis process.

MA3 **Enablement of Advanced Analysis:** The model specificity empowers researchers to perform advanced analyses that are often cumbersome or impossible with generalized models. For example, specialized data flow analyses that leverage Jimple's unique features can be more effectively implemented within the JimNode framework.

These Model Advantages, denoted from **MA1** to **MA3**, constitute the foundation for the significant enhancements in JimNode's CPG model. In the following, we highlight the practical disadvantages of the most commonly used language-agnostic tool for generating CPGs, Joern.

4.2.3 Practical Disadvantages of Joern

While Joern excels at providing a robust framework for analyzing code across a variety of programming languages, its application to the nuanced features of languages like Jimple reveals certain limiations. These shortcomings underscore the limitations of a generalized approach to static code analysis, especially when dealing with language-specific features of Jimple. We categorize these disadvantages as follows.

PD1 **Excluding Identity Statements:** Joern's approach to control flow graphs (CFGs) omits identity statements, which are crucial for representing the initial state of parameters and fields in Jimple. The exclusion of these statements, as discussed in a recent update to Joern's repository (Joern Pull Request 1094), results in an incomplete and potentially misleading CFG representation [Pro22].

PD2 **Inaccuracies with Goto Statements:** The CFGs in Joern have been found to inaccurately represent goto statements, including multiple targets for a single statement. This misrepresentation could lead to incorrect assumptions about the program's control flow, impacting the accuracy of subsequent analyses.

PD3 **Ambiguities in Invoke Statement Types:** Joern's language-agnostic model does not adequately differentiate between various types of invoke statements, as `specialinvoke`, `interfaceinvoke`, `dynamicinvoke`, or `virtualinvoke`. This lack of specificity can obscure the precise nature of method invocations in Jimple, complicating the task of accurate method call analysis.

PD4 Lack of Edge Information: The edges in Joern's CPG model lack detailed information, making it challenging to discern the exact nature of control flow, especially in conditional structures. Analysts are often required to infer the meaning of edges based on the context, which is not always evident or accurate.

PD5 Learning Curve for Joern Query Language: Joern introduces its own query language, which, while powerful, has a steep learning curve and is more naturally suited to languages like C and C++ for which Joern was originally developed. This creates an additional barrier for users working primarily with Java or Jimple, who must adapt to a query language that may not align well with these languages' idioms.

These Practical Disadvantages, denoted from **PD1** to **PD5**, highlight the challenges posed by language-agnostic models in accurately capturing the complexities of Jimple code and reinforce the need for tailored solutions like *JimNode* for program analysis in this domain.

4.3 Implementation and Integration

The practical realization of JimNode involved the development of a new module within the SootUp framework, an overhaul of the Soot framework first released in December 2022 [Soo23b]. This section briefly outlines the integration process and the technical aspects regarding the construction of the various components of the CPG.

4.3.1 Adopting SootUp's CFG and AST

SootUp serves as the foundational platform for JimNode's implementation. The framework's `StmtGraph` provided the basis for constructing the CFG component of the CPG. For the AST, JimNode leverages simple parsing techniques on statement objects within SootUp to ensure seamless integration of the AST property graph, which is built on SootUp's robust object model. The implementation details and the module's source code can be accessed at the following link.

`https://github.com/michaelyoukeim/SootUp`

4.3.2 Building the CDG and DDG

The development of the CDG and DDG represents JimNode's core contribution to enhancing the analytical capabilities of the CPG.

4.3.3 Model Considerations

The development of JimNode within the SootUp framework aimed to refine the granularity of the CPG for Jimple, as described in section 4.1. Although the project succeeded in improving the level of detail of the nodes beyond the capabilities of generalized models such as Joern, the desired typed structure could not be fully achieved due to time constraints. While the nodes were elaborated in great detail, the edge labels were implemented as strings, which are intended for later refactoring towards a more type-specific schema.

The aspiration for developing a fully typed model that would provide a nuanced and detailed perspective on Jimple statements could not be realized within the timeframe of the project. The need to prioritize the correctness and stability of the current features led to this goal being put on hold. Consequently, the focus of the project shifted to creating a robust foundation that, despite the preliminary use of string labels for edges, significantly improves the analysis capabilities over the existing models.

This strategic approach ensures that the current iteration of JimNode not only represents a significant improvement in static code analysis tools, but also provides a solid foundation for future developments. The foundation laid by this project describes a clear path to achieving the comprehensive, type-enriched CPG that was originally proposed and underlines the success of the project in balancing immediate practical requirements with long-term goals. Conceptually, transforming the current version to the aspired model only requires slight refactoring.

5 Evaluation

In the following, we evaluate our research's effectiveness and efficiency. This includes setting the research questions and criteria in Section 5.1, describing the evaluation methodology in Section 5.2, and discussing the results in Section 5.3, providing a critical analysis of our findings.

5.1 Research Questions and Criteria

The evaluation of the conducted work is based on the following research questions.

- **RQ1:** Does the generated CPG accurately represent the specified input?
- **RQ2:** How well does the provided solution perform regarding processing time and resource utilization?

In our research questions, we select Joern as the baseline because it is the primary tool for CPG generation [joe]. This choice allows us to validate our JimNode's performance directly compared to the established standard.

RQ1 aims to verify the correctness of the generated CPGs, particularly their ability to accurately represent the syntax, control flow, and data flow of an input Java program transformed into Jimple. A successful assessment would confirm that our method reliably translates the Jimple code into a CPG format that fulfills the criteria established by Yamaguchi et al.[YGAR]. To this end, we perform a comparative analysis between the CPGs generated by JimNode and the CPGs generated by Joern to provide a direct benchmark for accuracy.

RQ2 evaluates the efficiency of JimNode in generating CPGs from Jimple code, focusing on processing time and resource utilization. By comparing our performance metrics against Joern's, we aim to highlight the relative enhancements and benefits JimNode introduces.

5.2 Evaluation Methodology

The core of our evaluation methodology revolves around assessing the correctness of the CPGs generated for a given JAR file. Figure 5.1 illustrates the process we follow to assess the CPG generation accuracy. We first generate CPGs for the file using JimNode and Joern. Joern relies internally on Soot for processing Jimple code. Given that Jimple version produced by Soot is not fully compatible with that produced by SootUp, we have to ensure that the CPGs produced by both platforms are based on the same version of Jimple. To achieve this, we

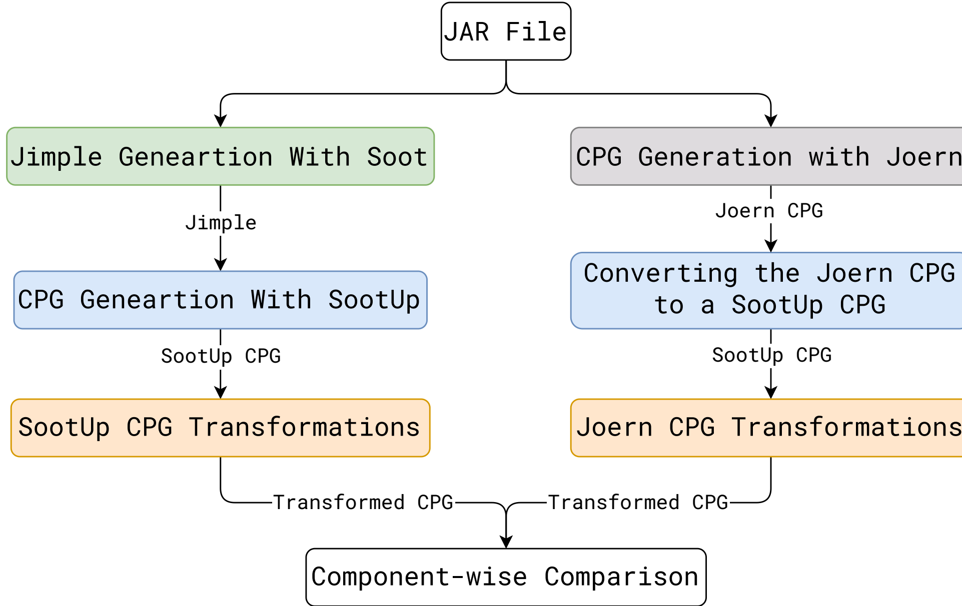

Figure 5.1: Evaluation methodology for the comparison of CPGs generated from Jimple code with SootUp and Joern.

use Soot to convert the JAR file into Jimple code. This code is then used to generate the CPGs with SootUp. Following this, for the CPGs generated with Joern, we undertake a crucial step of interpreting and converting the Joern-generated CPG to a format that aligns with the SootUp model. This conversion is vital for maintaining consistency across the comparisons and facilitating the application of the transformations.

Subsequently, we apply a set of transformations to each CPG to obtain compatible representations. We then assess the level of similarity between the two transformed property graphs. Since the CPG is composed of different components such as AST, CFG, CDG and DDG, with potentially common nodes, but disjoint edges, we assess these subgraphs individually.

To maintain the integrity of our comparison, we only included those methods whose Jimple code was successfully interpreted by SootUp. In addition, we considered the available methods in both SootUp and Joern exclusively. Since the CPGs of both platforms follow different schemes, we performed specific transformations on both platforms to obtain comparable representations. The applied transformations are described in detail below.

- **Excluding Identity Statements:** As noted in **PD1**, the CFGs generated by Joern no longer incorporate identity statements [Pro22]. To align the representations, we removed identity statements from the JimNode CPGs.

- **Excluding Incorrect Goto Statements in Joern:** One of the issues identified with Joern's output , as outlined in **PD2**, is the inclusion of incorrect goto statements. To rectify this, we refined our analysis by filtering out the incorrect goto statements in Joern's representation and only considered the goto statements that point to valid target statement positions.

- **Normalizing Invoke Statements:**

  Since Joern does not provide a straightforward way to determine whether a statement is a `specialinvoke`, `interfaceinvoke`, `dynamicinvoke`, or `virtualinvoke` (see **PD3**), we normalized all invoke statements in both representations to `virtualinvoke`.

After applying the aforementioned conversions and transformations to the Joern CPG, we evaluate the degree of similarity between the two graphs using a similarity score. This score is calculated as the percentage of edges that are identical in both graphs relative to the total number of edges in the SootUp CPG. This method provides a precise indicator of their comparative alignment. In this respect, edges are considered identical when they connect nodes that align completely in detail. For example, within control flow graphs, nodes—typically representing statements—must match precisely in all their components to be deemed similar. This standard ensures that the similarity score accurately reflects not only an exact match between the graphs but also considers both the structural arrangement and the detailed correspondence of node attributes.

In our evaluation, we did not assess the AST component due to its complex matching requirements. Matching parent statements is a binary outcome process; a successful match means a 100% similarity in AST components, while failure leads to a complete mismatch. This nuanced matching process exceeded our current evaluation scope. Due to the intricate process required for matching parent statements in the AST and the complexity arising from model mismatches, our evaluation specifically focused on inter-statement edges—those connections existing directly between statements.

To evaluate runtime efficiency, we compared the time taken by Joern to generate a CPG with the time required by our newly integrated module to perform the same task. To identify the relationship between memory allocation and performance, we conducted our assessments across various memory allocation configurations.

5.3 Results and Discussion

To carry out the evaluation methods described above, we developed the evaluation tool CpgEval [You24]. This tool was utilized to assess the 30 most frequently used Maven projects. For the selection of these projects, we relied on the API provided by libraries.io[Lib]. Our search was specifically narrowed to Java packages, and to enable automated package downloading, we further restricted our query to include only those packages available on the Maven platform. In the subsequent text, we detail our findings related to efficiency and runtime performance.

5.3.1 RQ1: Similarity Analysis

Graph	Total Methods	Total Edges	Same Edges	Avg. Sim. %
CFG	50,759	305,370	296,660	97,14 %
CDG	50,787	120,092	102,894	85,67 %
DDG	50,787	212,999	162,737	76,40 %

Table 5.1: Similarity Assessment of CPG Components in JIMNODE and Joern.

The evaluation of graph similarities between JIMNODE and Joern is summarized in Table 5.1. This table presents a comprehensive comparison across three different types of graphs: CFGs, CDGs, and DDGs. The analysis includes different totals of methods for each graph type, due to the creteria specified in Section 5.1 for method coverage. The CFG comparison demonstrates a high degree of similarity, with an average similarity percentage of 93.9556%, indicating that the structural representation of control flow in SootUp closely aligns with Joern. The CDG analysis shows a slightly lower average similarity of 81.9565%, reflecting differences in how call dependencies are represented. The DDG comparison highlights a further distinct pattern of similarity at 76.40%. These findings underscore the effectiveness of JIMNODE in generating

graph representations that are largely consistent with Joern, with specific distinctions observed in each graph type.

To calculate the overall similarity percentage, we first summed the similar edges for all graph types, obtaining a total of $296,660+102,894+162,737 = 562,291$ same edges. Then, we summed the total edges for all graph types, resulting in $305,370 + 120,092 + 212,999 = 638,461$ total edges. The overall similarity percentage is thus calculated as follows:

$$\text{Overall Similarity \%} = \left(\frac{562,291}{638,461}\right) \times 100 \approx 88.07\%$$

This calculation reveals an 88.07% similarity in the inter-statement edges between the CPGs generated by JimNode and Joern, underscoring the effectiveness of our approach in producing comparable graph structures despite the model incompatibilities.

Several factors contribute to the less than perfect similarity percentages observed in our benchmarks. Primarily, even after the transformations, both models of Joern and JimNode have major incompabitibilities. For example, the DDG and CDG in Joern could include expresssions, while JimNode only includes statements as CDG and DDG Nodes.

Additionaly, the process of transforming the CPG into a format compatible with the SootUp model presents significant challenges. This transformation leverages Joern's query capabilities to extract insights from the CPGs, a process that, while powerful, is intricate and susceptible to errors. As a result, CPGs that might semantically match could be inaccurately deemed non-matching due to errors in this conversion process. Additionally, the Joern representation occasionally lacks specific information, such as statement positions, and fails to account for identity statements. Despite efforts to normalize these discrepancies, mismatches between the representations may still occur.

Furthermore, inaccuracies in CPG generation, attributable to either Joern or SootUp, could also explain the observed deviations. On the SootUp side, these inaccuracies might stem from bugs within the proposed solution or the underlying SootUp framework itself. However, pinpointing the exact cause of each discrepancy is not straightforward. The interplay of these factors—complex transformation processes, missing information in representations, and potential inaccuracies in CPG generation—collectively accounts for the observed variance from the ideal similarity percentages.

Below are the specifications of the evaluation environment we used for the similarity and performance evaluation.

- **Device Name:** michael-pc
- **CPU:** Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz, 2.60 GHz
- **Memory:** 16.0 GB RAM (15.9 GB usable)
- **System Type:** 64-bit operating system, x64-based processor
- **Operating System:** Windows 10 Pro
- **Java Version:** Oracle OpenJDK 20.0.1
- **Joern Version:** Maven dependency of version 1.2.35
- **SootUp Version:** 1.2.0

SootUp (m:s)	Joern (m:s)
01:02	18:41

Table 5.2: CPG Generation Time for the 30 Most Used Maven Projects

5.3.2 RQ2: Performance Analysis

Table 5.2 outlines the comparative performance of Joern and SootUp in generating CPG for the 30 most commonly used Maven projects. The recorded times - 18:41 for Joern and 01:02 for SootUp - represent the complete duration from start to finish of the CPG generation process. For SootUp, this includes the time from the start of generation to the point at which the CPG is fully prepared and available in a runtime variable. In contrast, Joern's duration includes the entire process initiated by the CPG creation command. This evaluation reveals a significant disparity in favor of SootUp. This difference is primarily attributed to several key operational and architectural distinctions between the two frameworks.

Firstly, Joern's methodology necessitates writing the CPG data to disk before it can be accessed or queried. This additional step introduces latency not present in SootUp's processing pipeline. Furthermore, the CPGs generated by Joern are comprehensive, encompassing detailed information about the code's structure, including the call graph as well as various other aspects. While this breadth of information is valuable for in-depth analysis, it inherently requires more time to generate and manage. Additionally, while Joern's generalized representation allows for a versatile analysis across different programming languages, this sophistication leads to a larger volume of data being saved. The abstraction process introduces extra computational steps and overhead, contributing to both a slower performance and increased data storage requirements.

In contrast, SootUp's modular architecture allows for selective utilization of its components and thus a targeted analysis that focuses exclusively on relevant aspects of the code. This approach not only optimizes the analysis process, but also reduces the computing overhead, resulting in faster runtime performance. Additionally, while Joern relies on the older Soot framework to generate its graphs, SootUp has implemented optimizations that surpass the original Soot's efficiency, further contributing to its accelerated performance.

6 Related Work

In the following, we review existing literature and research efforts that overlaps with our study.

6.1 Vulnerability Detection

Apart from the work done by Yamaguchi et al., several other studies laveraged the power CPGs for vulnerabilty detection [YGAR]. Li et al.applied their deep learning-based system, VulDeePecker, to three software products and successfully detected four vulnerabilities that were not reported in the National Vulnerability Database [LZX+][XLQ+][Şa].

Banse et al. introduced the *Cloud Property Graph* (CloudPG), which aims to enhance the security of cloud services by providing a comprehensive framework that integrates both static and dynamic security analysis [BKSW]. Their graph is based on the ontology, functionalities, and security features of cloud resources. Weiss et al. focused on developing of a platform that enables the generation of a language-independent CPG representation. Their use of fuzzy parsing enabled the generation of even incomplete or non-compilable code. Brito et al.used CPGs to identify vulnerabilities in WebAssembly binaries [Bri]. Their study addressed the challenge of detecting vulnerabilities in WebAssembly binaries developed in memory-unsafe languages [WB].

6.2 Static Analysis Frameworks

Wala is a static analysis framework developed by IBM that provides features for analyzing Java bytecode and related languages [WAL23]. It provides support for creating call graphs, point-to-analysis and various other compiler optimizations and analyses. Opal is a project that focuses on the static analysis of Java bytecode and emphasizes the precision and efficiency of its analyses [OPA23]. It excels in providing a solid foundation for the construction of practical program analysis and tools aimed at both academic research and industrial applications. Soot is a Java optimization framework that serves as a versatile tool for the analysis and transformation of Java and Android applications [Soo23a]. It supports four different representations of code: Jimple (an intermediate representation), Shimple (an SSA variant of Jimple), Grimp (an aggregated version of Jimple that is suitable for decompilation and code inspection), and Baf (a simplified representation of bytecode). SootUp is an overhaul of the Soot framework that aims to improve its usability and extend its capabilities [Soo23b].

6.3 Standardizing Property Graphs

Currently, efforts towards the standardization of property graphs are still ongoing [Int24]. GQL aims to become an international standard for property graph querying, based on the initial efforts by languages such as Cypher, Gremlin, and SPARQL [GJK$^+$] [Neo24]. Thakkar et al. emphasize the lack of standardization in various graph-based data management systems [TPAV]. Green et al. provided precise mathematical definitions for property graphs in GQL [GGL]. Additionally, Francis et al. introduced GPC, a pattern calculus for property graphs, aligning with the standards adopted by the GQL Standard committee [FGG$^+$]. The discussions including transformation from other representations, like RDF-star, included the work conducted by Abuoda et al.[ADKH].

7 Threats to Validity

In the following, we identify and discuss the potential threats to the validity of our research. Through Sections 7.1 to 7.3, we critically examine the limitations of our methodology, tools, and data, ensuring a transparent evaluation of our study's robustness.

7.1 Technical and Tool-Specific Considerations

In the following, we describe the technical and tool-specific limitations encountered.

7.1.1 Tool-Specific Limitations

Our evaluation method is primarily based on the comparison between the results generated by our newly integrated module and those generated by Joern. While Joern is widely known for its ability to generate CPGs accurately, its process of converting Jimple code specifically is not entirely accurate. As mentioned in the Evaluation section, Joern disregards identity statements and provides insufficient information about statement positions and method invocation statement types. Joern's limitations restrict the quality and precision of our evaluation. Furthermore, our approach makes extensive use of the SootUp framework for the creation of the CFG and AST components of our CPGs. The development of the DDG component also relies on the advanced data flow analysis capabilities provided by the SootUp framework. Consequently, existing bugs or limitations in these tools could affect the integrity and reliability of the CPGs we produce.

Given this reliance on external tools, our evaluation process acknowledges the complexity of accurately comparing both CPG formats. It is worth noting that although the leveraged tools and features play an important role in our analysis, they bring their own challenges and limitations. For example, the accuracy of the CPG outputs depends not only on the correctness of the Jimple code conversion, but also on how comprehensively these tools can represent the various components of the code in graph format. The effectiveness of our evaluation is therefore closely tied to the performance and capabilities of the underlying frameworks. As we navigate these dependencies, our methodology is aware of the potential for discrepancies and the need for careful interpretation of the results.

7.1.2 Variability and Bias in Jimple Code Generation

The process of generating Jimple code from Java bytecode, primarily through Soot, inherently involves a certain degree of variability. This is partly due to the use of different versions of Soot

or changes to configuration settings during Jimple creation. Although we aimed to generate the Jimple code with the same options that are used in Joern, there is a possibility that we made an oversight that resulted in the generation of different Soot codes. This potential discrepancy could be due to variations in the configured of the Soot options.

Additionally, since SootUp interprets only specific portions of the Jimple files for evaluation, this selective processing introduces a bias. The alternative approach to using Soot for the Jimple code code generation was to directly convert the JAR files using SootUp. However, due to compatibility issues between the Jimple code produced by Soot and SootUp, executing evaluations through this method proved to be significantly challenging. These compatibility issues arise from differences in the way Soot and SootUp process and interpret Java bytecode, resulting in mismatches that could affect the accuracy and consistency of the generation of CPG. As a result, the variability introduced by the Soot-based Jimple code generation process remains a critical factor that affects the reliability of our comparative analysis.

7.2 Methodological Concerns

In the following, we outline our methodological concerns.

7.2.1 Conversion and Transformation Bias

As outlined in the evaluation section, our method for comparing the CPGs involved performing a series of transformations on the outputs of both frameworks. A critical procedure in this process was the interpretation and subsequent conversion of the CPGs generated by Joern to ensure compatibility with the SootUp model. This step naturally entailed the possibility of inaccuracies. Efforts to make Joern's and SootUp's CPGs comparable may not capture every detail or subtle difference in the way each tool presents information.

For the conversion of Joern's CPG, we worked with the data that was provided via the query interface, which allows for querying the generated CPGs. A major challenge was that much of this data was in a text-based format. Compared to object-based representations, text-based data is inherently more complex to interpret. To overcome this complexity, we occasionally resorted to minor heuristics. These heuristics were used to infer the types or values of certain components within the graph nodes based on the available textual information.

While these heuristics were invaluable in advancing the conversion process, they are also a potential source of error. By relying on inference rather than direct interpretation, there is a risk that the information encoded in the Joern CPGs may be misrepresented or oversimplified. This methodological necessity, though carefully considered, highlights one of the inherent challenges in ensuring the accuracy of our comparative analysis. Every step taken to bridge the gap between the different formats of CPGs was done with recognition of these limitations. Our focus was to minimize these errors and provide a fairly comprehensive and accurate comparison.

7.2.2 Measurement and Evaluation Criteria

The criteria chosen for computing edge similarity, and thus the overall similarity score between CPGs, may not represent the most comprehensive approach to understanding graph equivalence. Our method quantifies similarity primarily based on the percentage of matching edges and thus focuses on a direct attribute comparison. While this provides a clear, quantifiable metric, it may not capture the full complexity of graph equivalence, particularly in aspects that go beyond mere structural congruence.

Conventionally, more nuanced approaches to assessing similarity could consider the semantic relationships and functional roles of nodes and edges within graphs, beyond their mere presence

or absence. These methods could potentially provide deeper insight into graph equivalence by evaluating how changes in one part of the graph affect its overall behavior or meaning. In contrast, a purely quantitative measure based on edge matching could miss such subtleties and potentially misrepresent the true similarity between graphs.

Furthermore, the chosen criteria assume that all edges contribute equally to the semantics of the graph, which is not always the case. In reality, some edges might be more important due to their role in the control flow, data dependencies or security vulnerabilities. A more refined approach could weight edges differently based on these considerations, providing a more accurate reflection of graph similarity that is consistent with conventional practices in CPG analysis.

7.3 Data Selection and Reliability

7.3.1 Selection Bias

In our study, we focused on analyzing the 30 most commonly used Maven projects. It is important to recognize that projects that are not based on Maven or those with different characteristics might yield different results. There is a likelihood that projects from different domains have unique semantic structures that were not considered in our analysis. The selection of widely used projects suggests that they are mostly in line with common and recommended practices and thus have a certain structural uniformity. Furthermore, it is conceivable that the most widely used libraries contain methods with relatively small scale that adhere to best practices, which would imply that the analyzed control flow graphs do not represent more complex configurations. Nonetheless, the rationale behind selecting the most popular projects was to maximize the likelihood of capturing a wide range of patterns. However, there remains an inherent bias, as our analysis may primarily reflect patterns that are more common. While this approach is strategic, it also recognizes the potential limitation of the scope of patterns that our assessment could encompass.

7.3.2 API and Data Source Reliability

In our assessment, we used libraries.io API to determine the 30 most used packages [Lib]. We assumed that the data provided was both accurate and comprehensive. Nevertheless, there is a possibility that this data does not accurately reflect actual usage, which could result in a selection of projects that do not truly represent the top 30 in terms of actual usage.

8 Conclusion and Future Work

In the following, we summarize the thesis's main findings and contributions in Section 8.1. Section 8.2 then outlines potential directions for future research and suggests possible pathways for building on the foundation laid in this thesis.

8.1 Conclusion

In this work, we aimed to improve the analysis of Jimple code by tailoring CPGs specifically to its unique intermediate representation. Through the development of JimNode, a solution designed to generate CPGs optimized for Jimple, this work aimed to bridge the gap between generic CPG generation tools and the specific needs of Jimple code analysis. The motivation behind JimNode was rooted in the hypothesis that a tailored approach could offer substantial improvements over the state-of-the-art tool, Joern, particularly in terms of simplicity and direct applicability to Jimple code.

Our comprehensive exploration and development process led to the creation of a model that not only aligns more closely with the structure and semantics of Jimple but also simplifies the CPG creation process. The evaluation of JimNode, through a detailed comparison with CPGs generated by Joern, underscored the accuracy of our approach. By examining the presence of equivalent edges in both JimNode-generated and Joern-generated CPGs, without directly measuring similarity due to the intentional divergence in model complexity, we provided empirical evidence supporting the effectiveness of our tailored solution. Despite differences in the underlying models, our comprehensive evaluation, which spanned approximately 50,800 methods, showed an 88.07% similarity in the inter-statement edges when compared with Joern, the leading tool for CPG generation.

This thesis underscores the importance of model adaptation in the realm of code analysis, highlighting how tailored solutions can lead to more efficient, accurate, and meaningful analyses. In comparing JimNode with Joern, we not only validated our model's effectiveness but also initiated a dialogue on the potential for specialized tools to complement or even surpass generalized counterparts in specific contexts.

In conclusion, the development and evaluation of JimNode represent a significant step forward regarding the generation of CPGs, particularly for Jimple code. The insights gained from this work pave the way for further exploration into tailored analysis tools, encouraging a more nuanced approach to software analysis that considers the intricacies of different programming languages.

8.2 Future Work

There are several avenues for future exploration that promise to make JimNode even more powerful and user-friendly. Below are key areas where future work could significantly contribute to the advancement of this project.

8.2.1 Advanced Query Capabilities

During the development of this project, the primary focus was laid on the foundational aspects of JimNode, particularly on generating and tailoring CPGs for Jimple. Consequently, the query capabilities received less attention in this initial phase. Looking ahead, there is a vast potential for improvement in this area. By supporting more advanced query capabilities, JimNode could enable the application of sophisticated patterns on the generated CPGs. This enhancement would allow users to conduct deeper and more nuanced analyses of their codebases, identifying complex patterns and vulnerabilities with greater ease.

8.2.2 Performance Optimization

As datasets grow in size and complexity, optimizing the performance of queries becomes increasingly important. Future work could focus on further improving the efficiency of data retrieval, ensuring that JimNode can handle large-scale CPGs without compromising on speed.

8.2.3 Database Integration

Integrating JimNode with various database systems would significantly expand its usability. This integration would allow JimNode to directly interact with graph databases, enabling the storage, retrieval, and analysis of CPGs within a database environment. Such support would facilitate the management of persistent graphs and enhance the tool's applicability in real-world scenarios.

8.2.4 Graph Visualization Enhancements

Currently, JimNode offers visualization through simple DOT graphs, which serve as a foundational tool for representing the structure of CPGs. While effective for basic visualization needs, there's considerable room for improvement to meet the diverse and complex requirements of users. Enhancing the graph visualization capabilities of JimNode is thus a promising direction for future work. By introducing more sophisticated visualization techniques, including interactive exploration features, customizable layouts, and detailed node and edge representations, JimNode could significantly improve the user's ability to understand and analyze the intricate relationships within CPGs.

Bibliography

[AAB+] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. 50(5):68:1–68:40.

[ABD+] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith Hare, Jan Hidders, Victor Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savković, Michael Schmidt, Juan Sequeda, Slawek Staworko, and Dominik Tomaszuk. PG-keys: Keys for property graphs. pages 2423–2436.

[AC] F. E. Allen and J. Cocke. A program data flow analysis procedure. 19(3):137.

[ADKH] Ghadeer Abuoda, Daniele Dell'Aglio, Arthur Keen, and Katja Hose. Transforming RDF-star to property graphs: A preliminary analysis of transformation approaches – extended version.

[AHSM] Nikolaos Alexopoulos, Sheikh Mahbub Habib, Steffen Schulz, and Max Mühlhäuser. The tip of the iceberg: On the merits of finding security bugs. 24(1):3:1–3:33.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[AW] Andrew C. M. Austin and Laurie Williams. One technique is not enough: A comparison of vulnerability discovery techniques.

[BBC+] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. 53(2):66–75.

[BCD+] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. 51(3). Place: New York, NY, USA Publisher: ACM.

[BEHW] Matt Bishop, Sophie Engle, Damien Howard, and Sean Whalen. A taxonomy of buffer overflow characteristics. 9(3):305–317.

[BKSW] Christian Banse, Immanuel Kunz, Angelika Schneider, and Konrad Weiss. Cloud property graph: Connecting cloud security assessments with static code analysis. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 13–19. ISSN: 2159-6190.

[Bri] Tiago Brito. Wasmati: An efficient static vulnerability scanner for WebAssembly.

[CFR+] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. 13(4):451–490. Number: 4.

[CHK] Keith Cooper, Timothy Harvey, and Ken Kennedy. A simple, fast dominance algorithm.

[FGG+] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoč. GPC: A pattern calculus for property graphs.

[FOW] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. 9(3):319–349. Number: 3.

[GGL] Alastair Green, Paolo Guagliardo, and Leonid Libkin. Property graphs and paths in GQL: Mathematical definitions.

[GJK+] Alastair Green, Martin Junghanns, Max Kiessling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. openCypher: New directions in property graph querying.

[Gra] Andy Gray. An historical perspective of software vulnerability management. 8(4):34–44.

[HBC+] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D'amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge graphs. 54(4):71:1–71:37.

[Int24] International Organization for Standardization. Information technology – database languages – gql. `https://www.iso.org/standard/76120.html`, 2024.

[joe] Joern: A robust code analysis platform. `https://joern.io/`.

[Kil] Gary A. Kildall. A unified approach to global program optimization. pages 194–206. Conference Name: the 1st annual ACM SIGACT-SIGPLAN symposium Place: Boston, Massachusetts Publisher: ACM Press.

[Lib] Libraries.io api. `https://libraries.io/api`.

[LPJ+] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. 67(3):1199–1218. Conference Name: IEEE Transactions on Reliability.

[LZX+] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A deep learning-based system for vulnerability detection.

[Muc] S. S. Muchnick. Advanced compiler design and implementation.

[Neo24] Neo4j. Neo4j graph database. `https://neo4j.com/`, 2024.

[NNH] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. Data flow analysis. In Flemming Nielson, Hanne Riis Nielson, and Chris Hankin, editors, *Principles of Program Analysis*, pages 35–139. Springer.

[OPA23] OPAL Project. Opal - a scala library for the static analysis, and the development of analysis frameworks, for java bytecode and frameworks that are written in java, 2023.

[Pro] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, IRE-AIEE-ACM '59 (Eastern), pages 133–138. Association for Computing Machinery.

[Pro22] Joern GitHub Project. Pull request 1094: Exclusion of identity statements in joern, 2022.

[PT] Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan. Defending against buffer-overflow vulnerabilities. 44(11):53–60.

[Ram] Ganesan Ramalingam. On loops, dominators, and dominance frontiers. 24.

[SNAA] Aamir Shahab, Muhammad Nadeem, Mamdouh Alenezi, and Raja Asif. An automated approach to fix buffer overflows. 10:3777.

[Soo23a] Soot Team. Soot - a java optimization framework, 2023.

[Soo23b] SootUp Team. Sootup - enhancing the soot framework, 2023.

[SR] Raunak Shakya and Akond Rahman. A preliminary taxonomy of techniques used in software fuzzing. In *Proceedings of the 7th Symposium on Hot Topics in the Science of Security*, HotSoS '20, pages 1–2. Association for Computing Machinery.

[TPAV] Harsh Thakkar, Dharmen Punjani, Sören Auer, and Maria-Esther Vidal. Towards an integrated graph algebra for graph pattern matching with gremlin.

[TZWL] Ye Tao, Lingming Zhang, Linzhang Wang, and Xuandong Li. An empirical study on detecting and fixing buffer overflow bugs.

[VJB+] Peter Vanbroekhoven, Gerda Janssens, Maurice Bruynooghe, Henk Corporaal, and Francky Catthoor. Advanced copy propagation for arrays. 38(7):24–33.

[VrCG+] Raja Vallee-rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework.

[VRGH+] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In David A. Watt, editor, *Compiler Construction*, Lecture Notes in Computer Science, pages 18–34. Springer.

[WAL23] WALA Team. Wala - t.j. watson libraries for analysis, 2023. GitHub repository.

[WB] Konrad Weiss and Christian Banse. A language-independent analysis platform for source code.

[XLQ+] Xiaojun Xu, Chang Liu, Feng Qian, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection.

[YGAR] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. ISSN: 2375-1207.

[You24] Michael Youkeim. CpgEval GitHub Repository. `https://github.com/michaelyoukeim/CpgEval/tree/master`, 2024.

[Şa] Canan Batur Şahin. Semantic-based vulnerability detection by functional connectivity of gated graph sequence neural networks.