Flow2Vec: Value-Flow-Based Precise Code Embedding

YULEI SUI, University of Technology Sydney, Australia
XIAO CHENG, Beijing University of Posts and Telecommunications, China
GUANQIN ZHANG, University of Technology Sydney, Australia
HAOYU WANG, Beijing University of Posts and Telecommunications, China

Code embedding, as an emerging paradigm for source code analysis, has attracted much attention over the past few years. It aims to represent code semantics through distributed vector representations, which can be used to support a variety of program analysis tasks (e.g., code summarization and semantic labeling). However, existing code embedding approaches are intraprocedural, alias-unaware and ignoring the asymmetric transitivity of directed graphs abstracted from source code, thus they are still ineffective in preserving the structural information of code.

This paper presents Flow2Vec, a new code embedding approach that precisely preserves interprocedural program dependence (a.k.a value-flows). By approximating the high-order proximity, i.e., the asymmetric transitivity of value-flows, Flow2Vec embeds control-flows and alias-aware data-flows of a program in a low-dimensional vector space. Our value-flow embedding is formulated as matrix multiplication to preserve context-sensitive transitivity through CFL reachability by filtering out infeasible value-flow paths.

We have evaluated Flow2Vec using 32 popular open-source projects. Results from our experiments show that Flow2Vec successfully boosts the performance of two recent code embedding approaches code2vec and code2seq for two client applications, i.e., code classification and code summarization. For code classification, Flow2Vec improves code2vec with an average increase of 21.2%, 20.1% and 20.7% in precision, recall and F1, respectively. For code summarization, Flow2Vec outperforms code2seq by an average of 13.2%, 18.8% and 16.0% in precision, recall and F1, respectively.

CCS Concepts: • **Software and its engineering → Automated static analysis**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: Flow2Vec, code embedding, value-flows, asymmetric transitivity

ACM Reference Format:
Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: Value-Flow-Based Precise Code Embedding. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 233 (November 2020), 27 pages. https://doi.org/10.1145/3428301

1 INTRODUCTION

Static program analysis (or source code analysis) aims to reason about the runtime behavior of a program without actually running it. It is the cornerstone of many clients, such as program optimization [Bodík and Anik 1998; Ferrante et al. 1987], program slicing [Gallagher and Lyle 1991; Weiser 1981], change impact analysis [Acharya and Robinson 2011; Canfora and Cerulo 2005],

Authors' addresses: Yulei Sui, yulei.sui@uts.edu.au, University of Technology Sydney, 15 Broadway, Sydney, NSW, Australia, PO Box 123; Xiao Cheng, jackiecheng@bupt.edu.cn, Beijing University of Posts and Telecommunications, No 10, Xitucheng Road, Haidian District, Beijing, China, 100876; Guanqin Zhang, 13426770@student.uts.edu.au, University of Technology Sydney, 15 Broadway, Sydney, NSW, Australia, PO Box 123; Haoyu Wang, haoyuwang@bupt.edu.cn, Beijing University of Posts and Telecommunications, No 10, Xitucheng Road, Haidian District, Beijing, China, 100876.

2475-1421/2020/11-ART233
https://doi.org/10.1145/3428301

bug detection [Shi et al. 2018, 2010], code classification [Alon et al. 2019b; Frantzeskou et al. 2008; Ugurel et al. 2002] and code summarization [Alon et al. 2019a; Kamiya et al. 2002; Sajnani et al. 2016]. A key challenge in static analysis is to develop an effective code representation that can precisely capture code semantics to support a wide range of client applications.

Existing efforts and limitations. The recent success of embedding techniques in natural language processing has opened up new opportunities to develop effective code representations. *Code embedding*, as an emerging paradigm to support code analysis, produces low-dimensional vector representations of code (so called distributed code representation), which enables the application of existing machine learning techniques to various code analysis tasks. Many previous efforts in code embedding have treated source code as textual tokens by using Word2Vec-like techniques [Allamanis et al. 2016; Hindle et al. 2012; Pradel and Sen 2018; Raychev et al. 2014]. Later, abstract syntax trees have been used to extract the structural information of code in addition to shallow textual features [Alon et al. 2019b; Chen et al. 2018; Hu et al. 2018; Maddison and Tarlow 2014; Zhang et al. 2019]. Very recently, there have been a few attempts at investigating richer code semantics in the form of graphs, e.g., the data-flow graphs of a program [Allamanis et al. 2018; Ben-Nun et al. 2018; Li et al. 2018; Zhao and Huang 2018; Zhou et al. 2019].

However, existing approaches suffer from the following three limitations: (1) *Intraprocedural embedding*. Almost all existing embedding techniques are intraprocedural. These approaches extract the abstract syntax tree [Chen et al. 2018] or the data-flow information [Ben-Nun et al. 2018; Zhou et al. 2019] within each program method, but data-flows across methods are not preserved. The generated embedding vectors of individual methods are isolated with no calling context information. Thus, the resulting data dependence in the latent embedding space is context-insensitive. (2) *Alias-unaware*. One of the key research in data dependence analysis is memory aliasing. The state-of-the-art approaches are all alias-unaware. They either extract the data-flow information from ASTs [Allamanis et al. 2016; Hu et al. 2018; Zhang et al. 2019] or directly from an intermediate representation, e.g., LLVM-IR [Ben-Nun et al. 2018], without considering pointer alias information, which results in partial data dependence being embedded in the latent space. (3) *Ignoring asymmetric transitivity*. Since the data- or control-flow graphs of a program are directed, the current approaches which apply an existing network embedding technique (e.g., Gated Graph Neural Networks [Allamanis et al. 2018, 2016] or Skip-Gram models [Ben-Nun et al. 2018]) fail to preserve the long-range asymmetric transitivity of context-sensitive program dependence.

Insights and challenges. To address these limitations, a precise code embedding approach needs to operate on a compact and comprehensive code representation (e.g., interprocedural value-flow graphs [Choi et al. 1991; Hardekopf and Lin 2011; Sui and Xue 2016]), which contains both control-flows and alias-aware data-flows of a program. The approach is also expected to precisely preserve deep code semantics extracted from the target code representation (e.g., value-flow graphs) in a low-dimensional vector space.

Traditional graph embedding approaches [Grover and Leskovec 2016; Perozzi et al. 2014; Tang et al. 2015] leverage random walks on a graph to identify its structure by exploring the bidirectional connectivity between two nodes through first-order proximity (by looking at directly connected nodes [Belkin and Niyogi 2002]) or second-order proximity (by looking at nodes with shared neighbors [Tang et al. 2015]). As such, two nodes with similar proximity are forced to be close to one another in the latent embedding space. Recently, graph embedding approaches based on high-order proximity [Lian et al. 2018; Ou et al. 2016; Zhang et al. 2018] have been proposed to capture the high-level structural information of a graph. The technique is particularly useful for precisely preserving long-range program dependence information, which often manifests as many multi-hop def-use (value-flows) edges between program statements within and across methods.

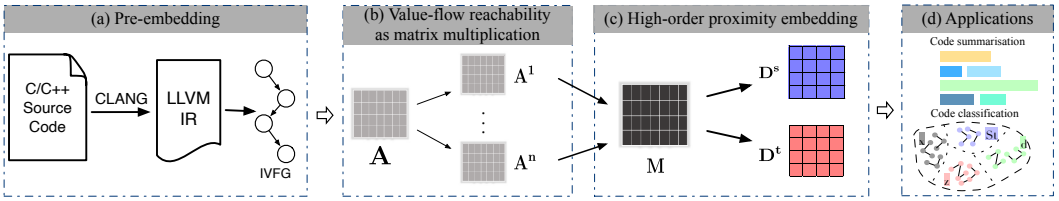

Fig. 1. Overview of our approach.

Inspired by these approaches, this paper aims to investigate, for the first time, high-order proximity based code embedding by preserving asymmetric transitivity of interprocedural alias-aware value-flows. Another challenge for interprocedural embedding is to identify and embed "realizable paths" [Reps 1998] on the value-flow graph in low-dimensional space by distinguishing different calling contexts leading to a procedure, thus producing a precise context-sensitive code representation to better support subsequent client applications (e.g., code classification and summarization).

Our solution. We present Flow2Vec, a new code embedding approach that preserves asymmetric transitivity of interprocedural value-flows of a program. To approximate a high-order proximity, our value-flow embedding is formulated as matrix multiplication by preserving context-sensitive transitivity through CFL reachability on the interprocedural value-flow graph (IVFG). A higher proximity from node i to node j implies the more feasible and the shorter value-flow paths from i to j on the IVFG. The high-order matrix is then decomposed to generate two types of embedding vectors, source and target vectors, for each node to represent its incoming and outgoing transitivity on the IVFG. The reachability information from node i to node j is then translated into computing the inner product between i's source vector and j's target vector. In the resulting embedding space, two nodes that can reach one another along value-flow paths on the IVFG are mapped to close numerical vectors, thus preserving high-level structural information of code.

Unlike many existing code embedding approaches [Allamanis et al. 2018; Alon et al. 2019a,b; Ben-Nun et al. 2018; Zhou et al. 2019] that rely on supervised learning and require ground truth data to train the model, our value-flow embedding does not depend on prior knowledge. Benefiting from fast matrix factorization techniques, the new code embedding supports tunable precision and efficiently solves reachability by approximating high-order proximity through Katz Index [Hochstenbach 2009; Ou et al. 2016], with theoretical guarantees on the Root Mean Squared Error (RMSE). Furthermore, our value-flow embedding, which preserves alias-aware value-flow reachability, serves as a new code representation. It can also be used as inputs to support subsequent client applications, such as code classification and summarization.

Framework overview. Figure 1 provides an overview of Flow2Vec with its four phases:

(a) *Pre-embedding.* A program is first compiled into LLVM-IR. The interprocedural value-flow graph (IVFG) is built on top of the IR using Andersen's pointer analysis [Andersen 1994]. The IVFG is then transformed into an adjacency matrix $\mathbf{A}$ with call/return value-flow edges represented by symbolic elements (i.e., opening/closing parentheses with their corresponding callsite information).

(b) *Value-flow reachability via matrix multiplication.* Value-flow reachability is formulated as a chain-matrix-multiplication problem. The number of value-flow paths between any two nodes with a path length of n is obtained by the n-th power of matrix $\mathbf{A}$. To achieve context-sensitive results, the symbolic elements in the resulting matrix are resolved by matching calls and returns so as to filter out unrealizable inter-procedural paths as a balanced-parentheses problem based on the CFL-reachability [Kodumal and Aiken 2004; Reps 1998].

(c) *High-order proximity embedding.* Given the matrices representing value-flow reachability for different path lengths, this phase approximates a high-order proximity matrix $\mathbf{M}$ via Katz

Index [Katz 1953] and decomposing **M** into source and target embedding vectors for each node on the IVFG. The reachability between node i and node j is measured as the dot products between i's source and j's target vector to preserve asymmetric transitivity of value-flows.

(d) ***Application scenarios.*** We demonstrate the effectiveness of Flow2Vec in boosting the performance of code2vec and code2seq, two state-of-the-art learning-based code embedding approaches for two important clients, i.e., code classification and summarization in terms of increased precision, recall and F1.

Our major contributions are as follows:

- We introduce Flow2Vec, a new approach to code embedding by preserving interprocedural alias-aware value-flows.
- We formulate value-flow reachability as matrix multiplication to embed context-sensitive value-flows through CFL reachability in the low-dimensional space.
- We present a comprehensive evaluation of Flow2Vec's precision, recall and F1-scores for both code classification and code summarization using 32 popular open-source C/C++ projects consisting of over 5 million lines of LLVM instructions. The code classification experiments show that Flow2Vec outperforms code2vec with an average increase of 21.2% in precision, 20.1% in recall and 20.7% in F1. The results for the code summarization show that Flow2Vec improves upon code2seq's performance by an average of 13.2% in precision, 18.8% in recall and 16.0% in F1.

2 BACKGROUND

This section sets out the preliminary knowledge of our work, including LLVM-IR, IVFG and graph embedding.

2.1 LLVM-IR

Modern compilers (e.g., LLVM) normally support multiple front-ends that compile programs written in different programming languages into a uniform Intermediate Representation (IR), e.g., LLVM-IR, for subsequent analysis and transformation tasks. Without loss of generality, our approach works on top of LLVM-IR, a popular and robust code representation used by many existing program analyses [Balatsouras and Smaragdakis 2016; Ben-Nun et al. 2018; Lei and Sui 2019; Lhoták and Chung 2011; Li et al. 2011]. In the LLVM-IR of a program, the set of all variables $\mathcal{V}$ is separated into two subsets, $\mathcal{O}$ which contains all possible abstract objects, i.e., the *address-taken variables* of a pointer, and $\mathcal{P}$ which contains all *top-level variables*, including stack virtual registers (the symbols starting with "%") and global variables (the symbols starting with "@") which are explicit, i.e., directly accessed. Address-taken variables in $\mathcal{O}$ are implicit, i.e., accessed indirectly at LLVM's load or store instructions via top-level variables.

After the SSA conversion, the LLVM-IR of a program is represented by five types of instructions: $p = \&o$ (AddrOf), $p = q$ (Copy), $p = \&q \rightarrow f_i$ (Field), $p = *q$ (Load) and $*p = q$ (Store), where $p, q \in \mathcal{P}$ and $o \in \mathcal{O}$ are address-taken variables. o is either a stack, global or a dynamically created heap object for an AddrOf $p = \&o$, known as an *allocation site*.

Figures 2(a) and (b) show a code fragment (with variable type information ignored for brevity) and its corresponding LLVM-IR, where $p, q, r, t_1, x \in \mathcal{P}$ and $a, b \in \mathcal{O}$. Note that a is indirectly accessed at a store $*p = t_1$ by introducing a top-level pointer t_1 in the LLVM's partial SSA form [Lattner and Adve 2004].

2.2 Interprocedural Value-Flow Graph (IVFG)

The IVFG [Hardekopf and Lin 2011; Shi et al. 2018; Sui and Xue 2016] of a program is built upon LLVM-IR by considering program control-flow and alias-aware data-flows. IVFG = (N, E)

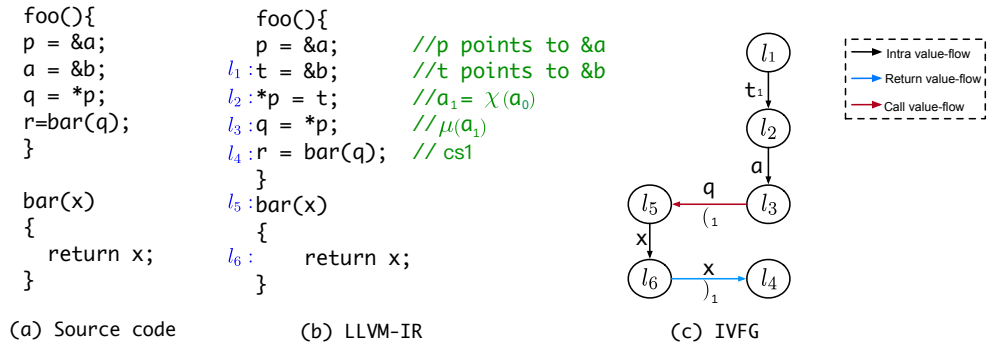

Fig. 2. A C code fragment and its LLVM instructions and interprocedural value-flow graph (IVFG).

is a multi-edged directed graph that captures its def-use chains flow-sensitively. N is the set of nodes representing all instructions and E is the set of edges representing all def-use chains. More specifically, an edge $\ell \xrightarrow{v} \ell'$, where $v \in \mathcal{V}$, from the instruction ℓ to ℓ' signifies a def-use relation for v with its def at ℓ and use at ℓ'.

As top-level variables are in LLVM's SSA form, their uses have unique definitions (with ϕ functions inserted at confluence points of a program's control-flow graph). A def-use chain $\ell \xrightarrow{t} \ell'$, where $t \in \mathcal{P}$, represents a *direct value-flow* of t. Such def-use chains can be found easily without the need for pointer analysis.

However, address-taken variables are not in LLVM's partial SSA form, so their indirect uses at loads may be defined indirectly at multiple stores. Their def-use chains are obtained by building the interprocedural memory SSA form following [Chow et al. 1996; Hardekopf and Lin 2011]. Figure 2 provides an example. First, the points-to information in the program is computed by Andersen's pointer analysis [Andersen 1994]. The points-to targets of the pointers p and t_1 are shown beside the first two instructions respectively. Second, a store, e.g., $*p = t_1$ is annotated with a function $a = \chi(a)$ for each variable $a \in O$ that may be pointed to by p to represent a potential def and use of a at the store. If a can be strongly updated [Lhoták and Chung 2011], then a receives the value on the right hand side of the store, i.e., "t_1" and the old contents in a are killed. Otherwise, a must also incorporate its old contents, resulting in a weak update to a. Likewise, a load $q = *p$ is annotated with a function $\mu(a)$ for each variable $a \in O$ that may be pointed to by p to represent a potential use of a at the load. Third, we convert all the address-taken variables into SSA form, with each $\mu(a)$ treated as a use of a and each $a = \chi(a)$ as both a def and use of a. Finally, we obtain the indirect def-use chains for $a \in O$ as follows. For a use of a identified as a_n (with its version identified by n) at a load or store ℓ, its unique definition of a is a_n at a store ℓ'. Then, an indirect def-use chain $\ell' \xrightarrow{a} \ell$ is added to represent the potentially *indirect value-flow* of a from ℓ' to ℓ (e.g., $\ell_2 \xrightarrow{a} \ell_3$ in Figure 2(c)). The opening/closing parentheses with the callsite information are put on the call/return edges to differentiate the intra-procedural value-flow edges from the inter-procedural ones. $\ell' \xrightarrow[(_i]{v} \ell$ denotes the call value-flow of v from ℓ' in a caller to ℓ in its callee via callsite i (e.g., $\ell_3 \xrightarrow[(_1]{q} \ell_5$ in Figure 2(c)). Similarly, $\ell_6 \xrightarrow[)_1]{r} \ell_4$ denotes the return edge from ℓ_6 to ℓ_4 via callsite $cs1$. Figure 2(c) shows the final IVFG of the LLVM-IR. The intra-procedural edges marked as the black arrows and call/return edges are the red/blue arrows.

2.3 Graph Embedding

Graph embedding [Cai et al. 2018; Cui et al. 2018] or network embedding is to transform a graph into a distributed representation in the form of a numerical vector or a set of vectors by capturing the graph's topology and vertex-to-vertex relations. The resulting embedding provides a compact data format, from which to efficiently conduct a wide variety of tasks, such as pattern recognition, clustering and classification, can be conducted efficiently in both time and space. For example, computing the (context-sensitive) reachability between two nodes on an IVFG, which previously requires costly enumeration of all possible paths between any two nodes, can be approximated by fast high-order proximity [Cui et al. 2018]. As another example, machine learning methods, which are powerful for solving some software engineering tasks, such as code summarization, require a precise structure-preserving code representation. In turn, this requires an effective embedding approach.

Preserving high-level graph structures, such as the reachability relations between two arbitrary nodes, is a fundamental requirement of graph embedding. Preserving asymmetric transitivity has recently become an important topic for precisely embedding directed graphs [Ou et al. 2016; Sun et al. 2019]. Given a directed graph $G = (N, E)$, where $N = \{v_1, v_2, \ldots, v_N\}$ is a set of nodes and E is a set of directed edges. An edge from v_i to v_j is represented as $e_{ij} = (v_i, v_j) \in E$. The embedding takes G's adjacency matrix (denoted as $\mathbf{A}$) as its input and outputs a high-order proximity matrix $\mathbf{M}$, where $\mathbf{M}_{i,j}$ preserves the asymmetric transitivity between v_i and v_j. Note that $\mathbf{M}_{i,j}$ denotes the element at the i-th row and j-th column of $\mathbf{M}$, and the bold lowercase symbol $\mathbf{s}_i$ represents the i-th row of $\mathbf{M}$). There are several commonly used models (e.g., random walk [Mikolov et al. 2013; Perozzi et al. 2014] and matrix factorization [De Lathauwer et al. 2000; Ou et al. 2016]) to transform a graph from its original graph space $\mathbf{A}$ to its high-order proximity space $\mathbf{M}$.

Given $\mathbf{M}$, the embedding vectors are extracted as $\mathbf{D} = [\mathbf{D}^s, \mathbf{D}^t]$, where the i-th row, $\mathbf{d}_i$, denotes the embedding vector of v_i and K is the embedding dimension. Unlike the embedding which maintains only one type of vectors for an undirected graph, asymmetric transitivity preservation on a directed graph produces two types of vectors: a source vector $\mathbf{D}^s$ and a target vector $\mathbf{D}^t$, in the embedding space $\mathcal{R}^{|N|\times K}$. The inner product between $\mathbf{d}_i^s$ and $\mathbf{d}_j^t$, i.e., $\mathbf{d}_i^s \cdot \mathbf{d}_j^{t\top}$ represents the approximated proximity from v_i to v_j. This proximity value is often normalized between 0 and 1. A higher proximity means more and shorter paths from v_i to v_j. A proximity value that fails under the user-defined threshold indicates that v_i are unlikely to reach v_j.

The fundamental difference between code and natural languages is the structural information available (e.g., IVFG). Exploiting rich code semantics via structure-preserving graph embedding becomes increasingly important. This paper fills the gap between the demand for comprehensive code representation and the lack of code embedding techniques that precisely preserve interprocedural context-sensitive program dependence.

3 A MOTIVATING EXAMPLE

Figure 3 illustrates the key idea of Flow2Vec. The objective is to show that our interprocedural code embedding can precisely preserve the context-sensitive value-flow reachability in the embedding space.

(a) Pre-embedding. The code example in Figure 3(a) is extracted from a real-world program `bison`. The IVFG of the code snippet is constructed by following Section 2.2. Each node on the IVFG represents a definition of a variable at an instruction with its corresponding line number. We then transform the IVFG into a symbolic matrix $\mathbf{A}$, with each element denoting the first-order reachability (two directly connected nodes) as depicted in Figure 3(a). If there is an intraprocedural value-flow between ℓ_i and ℓ_j for any two instructions ℓ_i and ℓ_j, $\mathbf{A}_{i,j} = 1$ and 0 otherwise. The

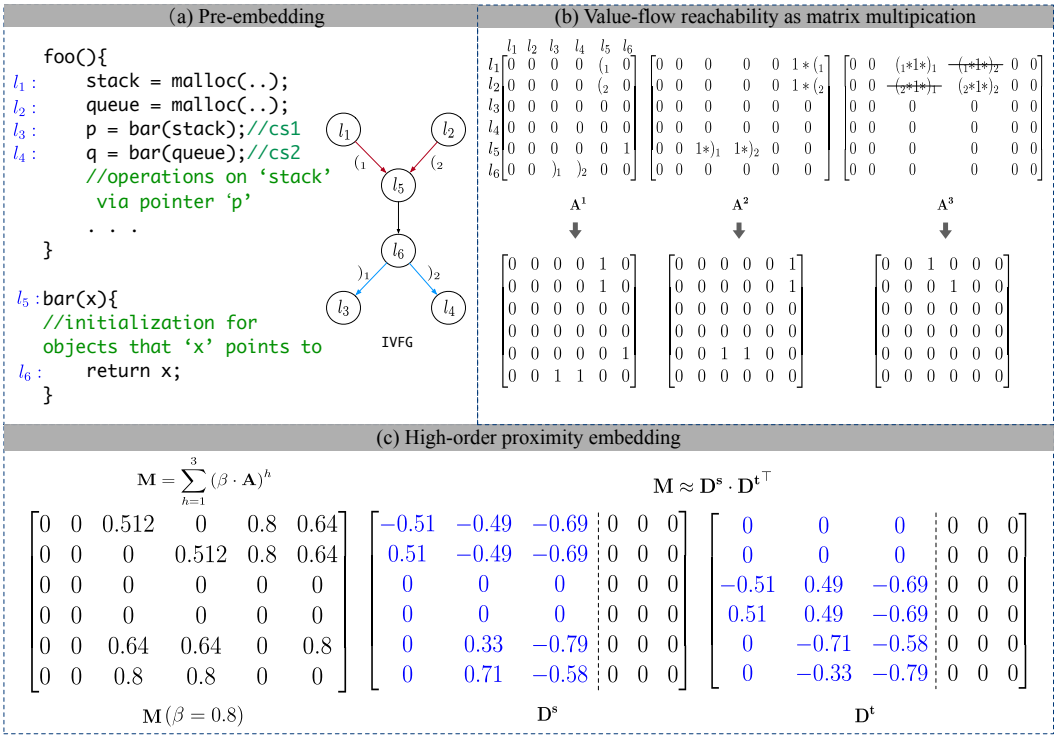

Fig. 3. A motivating example. (a) shows the code and its corresponding IVFG; (b) gives $\mathbf{A}^1$, $\mathbf{A}^2$ and $\mathbf{A}^3$ which represent context-sensitive reachability between any two IVFG nodes with a path length of 1, 2 or 3; and (c) shows a high-order proximity matrix $\mathbf{M}$, the source and target embedding vectors $\mathbf{D}^s$ and $\mathbf{D}^t$. The inner product between $\mathbf{d}_i^s \cdot \mathbf{d}_j^{t\,T}$ is the proximity from ℓ_i to ℓ_j.

interprocedural value-flows between ℓ_i and ℓ_j are symbolized using their callsite information, i.e., $\mathbf{A}_{i,j} = (_{csId}$ and $\mathbf{A}_{i,j} =)_{csId}$ for the call and return edges of the callsite $csId$, respectively. For the purposes of context-sensitive analysis, this later can be instantiated to either 1 or 0 for a feasible flow between i and j under different call paths. In Figure 3(a), $\mathbf{A}_{5,6} = 1$ signifies the intraprocedural reachability from ℓ_5 to ℓ_6, and $\mathbf{A}_{1,5} = (_1$ represents the interprocedural value-flow from ℓ_1 to ℓ_5 via callsite $cs1$.

(b) Value-flow reachability as matrix multiplication. The asymmetric transitivity of value-flows is formulated as a chain-matrix-multiplication problem as shown in Figure 3(b). The element at the i-th row and j-th column of $\mathbf{A}^h$, which is the h-th power[1] of $\mathbf{A}$, denotes the number of value-flow paths with a length of h from node i to node j. A symbolic element, e.g., $\mathbf{A}^3_{1,3} = (_1 * 1 *)_1$, indicates that a feasible path of length 3 from i and j involves one call edge, one intra-procedural edge, and one return edge. This multiplication of symbolic elements represents a value-flow concatenation by strictly preserving the order of call and return edges when traversing on the IVFG. Here, we only care about the call/return sequence to filter out "unrealizable" interprocedural paths by matching calls and returns. For example, $\mathbf{A}^3_{1,4} = (_1 * 1 *)_2$ signifies an infeasible path from ℓ_1 to ℓ_4 due to

[1]The power of matrix $\mathbf{A}$ is defined as $\mathbf{A}^h = \underbrace{\mathbf{A} \cdot \ldots \cdot \mathbf{A}}_{\times h}$ and $\mathbf{A}^0$ is the identity matrix.

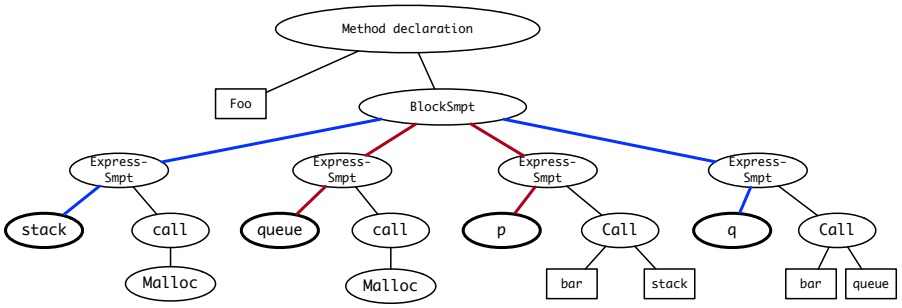

Fig. 4. The AST of foo in Figure 3(a) with two spurious paths, i.e., the path between stack and q (blue line) and the path between queue and p (red line).

unbalanced parentheses, which implies that *stack* can not pass its value to *q*. Once all the symbolic elements in a matrix of *h*-th power of **A** have been resolved based on CFL-reachability [Reps 1998] and replaced with either 1 or 0, yielding two context-sensitive reachability matrices $\mathbf{A}^2$ and $\mathbf{A}^3$ for paths of lengths 2 and 3.

(c) High-order proximity embedding. To preserve the high-level code structure (e.g., the ensemble of all feasible value-flow paths) and approximate the graph's high-order proximity, we choose Katz Index [Katz 1953]. The resulting proximity matrix $\mathbf{M} = \sum_{h=1}^{H} (\beta \cdot \mathbf{A})^h$ preserves the value-flow reachability, where H is the maximum limit for power of $\mathbf{A}$ and β is the decay parameter that determines how fast the weight of an edge decays when the length of a path grows. This guarantees convergence. We choose $H = 3$ and $\beta = 0.8$ in our example. We adopt the approach in [Ou et al. 2016] to reduce the computational cost of SVD (singular value decomposition) when approximating and decomposing the high-order proximity matrix into K-dimensional embedding vectors $\mathbf{D}^s$ and $\mathbf{D}^t$, as shown in Figure 3(c). The parameter K, represents the number of feature dimensions to support tunable precision for our embedding. The higher value of K becomes, the more precise the embedding is.

Figure 3(c) also illustrates the reachability computation using the 3-dimensional embedding vectors (the first three dimensions of $\mathbf{D}^s$ and $\mathbf{D}^t$ highlighted in blue). A closer distance between a source $\mathbf{d}_i^s$ and a target $\mathbf{d}_i^t$ vector means their inner product $\mathbf{d}_i^s \cdot \mathbf{d}_j^{t\,T}$ will have a higher proximity value. Compared to ℓ_6, ℓ_5 is closer to ℓ_1 on the IVFG. Accordingly, the value $\mathbf{d}_1^s \cdot \mathbf{d}_5^{t\,T} = 0.75$ is higher than $\mathbf{d}_1^s \cdot \mathbf{d}_6^{t\,T} = 0.71$ in the embedding space. A proximity value under a user-defined threshold indicates that ℓ_i unlikely reaches ℓ_j (our empirical threshold value of 0.003 under 110 feature dimensions can achieve 96.4% precision and 94.5% recall in our evaluation). Thus, Flow2Vec precisely preserves reachability from ℓ_1 to ℓ_3 with the proximity value of $\mathbf{d}_1^s \cdot \mathbf{d}_3^{t\,T} = 0.5$ and it also identifies two infeasible value-flows from ℓ_1 to ℓ_4 with $\mathbf{d}_1^s \cdot \mathbf{d}_4^{t\,T} = -0.02$ and from ℓ_2 to ℓ_3 with $\mathbf{d}_2^s \cdot \mathbf{d}_3^{t\,T} = -0.02$. Note that $\mathbf{d}_1^t$ and $\mathbf{d}_2^t$ are zero vectors, implying that no directed edges go to these two nodes.

(d) Applications. Our embedding provides a new code representation that is useful to support, for example, code classification and summarization tasks that require comprehensive interprocedural alias-aware program dependence information. Unlike code2vec [Alon et al. 2019b] and code2seq [Alon et al. 2019a] which embed the paths on a program's ASTs, Flow2Vec extracts deeper code structure information by exploring context-sensitive value-flow paths. Figure 4 shows foo's AST, which is intra-procedural and flow- and context-insensitive by nature. There two spurious paths on the AST. One is between *stack* and *q* (blue); the other is between *queue* to *p* (red). These spurious paths suggest false information that *p* may operate on the *queue*. In fact, *p* only accesses *stack* and there is no operation on *queue* in the remaining part of foo, as indicated in

the comments (green). Therefore, imprecise code information, such as spurious AST paths, can adversely affect the accuracy of code classification and summarization, as also demonstrated in our experiments in Section 5.2 and our case studies in Figure 12.

4 FLOW2VEC APPROACH

This section details our value-flow embedding approach, including pre-embedding, value-flow reachability via matrix multiplication and high-order proximity approximation by preserving asymmetric transitivity of value-flows in low-dimensional embedding space.

4.1 Pre-Embedding

Our pre-embedding phase accepts the source code of a project and compiles the code into LLVM-IR. The memory SSA form and the interprocedural value-flow graph are then built using Andersen's pointer analysis [Andersen 1994; Sui and Xue 2016]. The indirect function calls are resolved using the points-to information from pointer analysis. The IVFG is then translated into a symbolic matrix $\mathbf{A}$ with the call site IDs as placeholders for interprocedural value-flows. For example, if the two value-flows are passing to and returning from a callee through the same callsite i, the call and the return value-flow edges are labeled with an opening parenthesis $(_i$ and a closing parenthesis $)_i$ respectively.

4.2 Value-Flow Reachability as Matrix Multiplication

The context-sensitive value-flow reachability is formulated as a chain-matrix-multiplication problem to compute a high-order proximity matrix $\mathbf{M}$ given the adjacency matrix $\mathbf{A}$ of an IVFG. Note that $\mathbf{A}$ is sparse based on our sparse value-flow graph (Section 2.2). In Section 4.2.1, we introduce the matrix multiplication to obtain the context-insensitive value-flow reachability. Then, in Section 4.2.2, we move to context-sensitive reachability.

4.2.1 ***Context-insensitive value-flow reachability.*** For context-insensitive value-flow reachability, the symbolic matrix $\mathbf{A}$ is treated as a binary adjacency matrix, where the variable name on a value-flow edge and the callsite information on an interprocedural edge are ignored for context-insensitive analysis. The element $\mathbf{A}_{i,j}$ of $\mathbf{A}$ is 1 if there a directed edge from i to j (with a path of length of 1) exists in the graph, otherwise $\mathbf{A}_{i,j}$ is 0. The matrix element $\mathbf{A}_{i,j}$ in the h-th power of the adjacency matrix represents the number of paths from node i to j with a length of h. For example, the element at the i-th row and j-th column of $\mathbf{A}^2$ is 1 if there exists one path with a length of 2 from node i to node j.

$$\mathbf{A}^{sum} = \sum_{h=1}^{H} \mathbf{A}^h, \tag{1}$$

Equation 1 formulates value-flow reachability as the sum of all powers of the adjacency matrix $\mathbf{A}$ over the natural number lattice $\mathbb{N}$. The equation can be used to compute the reachability information between any two nodes on IVFG for path lengths from 1 to H.

Example 1. Figure 5 depicts a context-insensitive value-flow graph without labeled edges. $\mathbf{A}^1$ represents the first-order reachability and $\mathbf{A}^2$ represents the second-order reachability through paths of length of 2 between any two nodes. $\mathbf{A}^{sum}$ is the final matrix considering all reachability closures for path lengths of both 1 and 2. $\mathbf{A}^{sum}_{1,3} = 2$ means that there are two paths on the graph from ℓ_1 to ℓ_3, i.e., $\ell_1 \rightarrow \ell_3$ of path length 1 and $\ell_1 \rightarrow \ell_2 \rightarrow \ell_3$ of length 2.

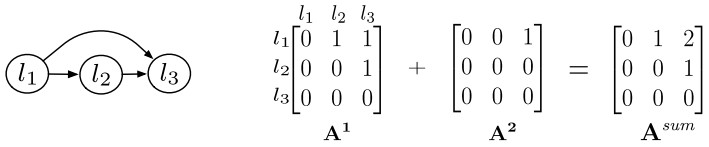

Fig. 5. Value-flow reachability via matrix multiplication ($\mathbf{A}^{sum}$ represent value-flow reachability between any two nodes for path lengths from 1 to 2).

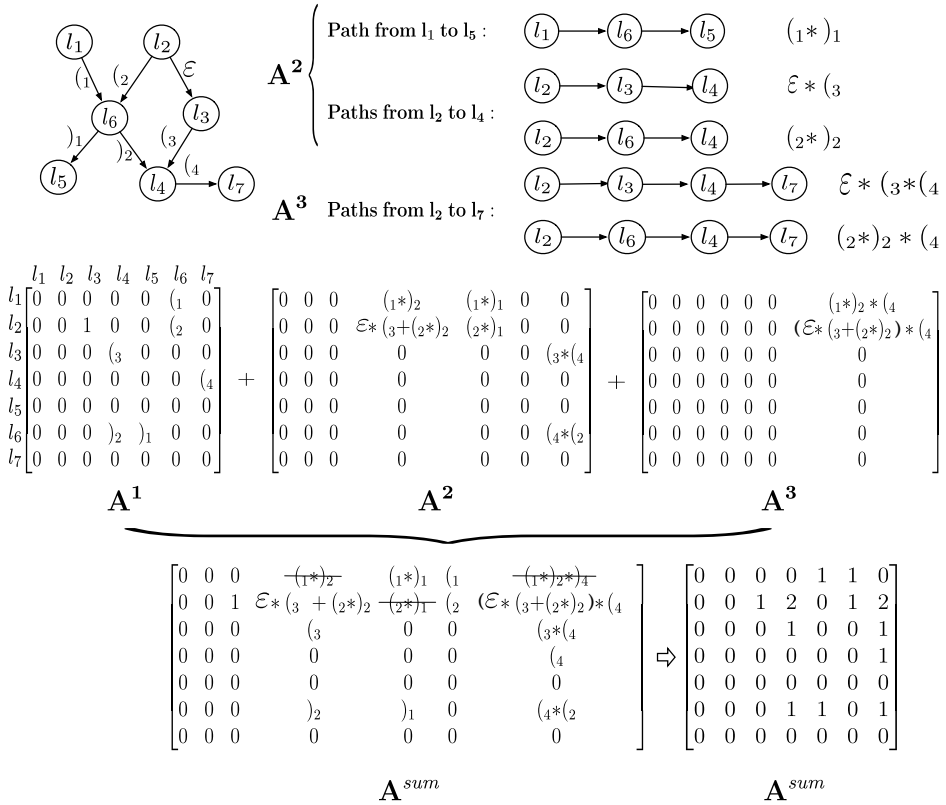

Fig. 6. Context-sensitive value-flow reachability via symbolic matrix.

4.2.2 ***Context-sensitive value-flow reachability.*** When applying Equation 1 to the symbolic matrix $\mathbf{A}$ for context-sensitive analysis, the value of an element $\mathbf{A}^{sum}_{i,j}$ in the resulting matrix $\mathbf{A}^{sum}$ may not be a numeric value, but it can be a symbolic expression, which represents the interprocedural value-flow path(s). The multiplication of two symbolic elements represents a value-flow concatenation by preserving the order of call and return IVFG edges. The addition of two symbolic elements, same as the context-insensitive analysis, represents that there are multiple possible paths between two nodes.

Unlike in the insensitive analysis, an interprocedural value-flow path involving call and/or return edges between two nodes can not be simply treated as feasible. Rather, it is recognized as a

feasible interprocedural path using CFL-reachability [Reps 1998; Sridharan and Bodík 2006]. CFL-reachability is an extension of standard graph reachability which allows filtering out unrealizable paths on the graph. We define the context-free grammar $CFG = (Q, N, P, S)$ for our analysis, where Q is the alphabet over the sets of opening and closing parentheses, i.e., $\{ (_1, (_2, ..., (_n \}$ and $\{)_1,)_2, ...,)_n \}$ and an empty string ϵ. N denotes nonterminal symbol. P is a set of production rules and S is the start symbol. The language L of size n (the number of callsites in a program) using the context-free grammar CFG is defined as follows:

$$L : C = CC \mid (_1\ C\ )_1 \mid \ \ldots\ \mid (_n\ C\ )_n \mid \epsilon$$

where C is the start symbol and the only nonterminal in the grammar. This language supports a special and restricted CFL, also called Dyck-CFL, which only generates strings of properly balanced parentheses [Kodumal and Aiken 2004]. Given an IVFG G with edge labels taken from alphabet Q. Note that intraprocedural edges are labeled with ϵ. Each path p in G is labeled with a string $s(p)$ in Q, obtained by concatenating edge labels in order. We say p is a L-path if $s(p) \in L$. A L-path is a realizable path if after entering a method m from callsite i, it exits from m back to callsite i. Following [Sridharan and Bodík 2006], a realizable also allows partially balanced parentheses (i.e., a prefix with unbalanced closed parentheses and a suffix with unbalanced open parentheses) since a realizable path may not start and end in the same method.

Example 2. Figure 6 shows an IVFG consisting of one intraprocedural edge (labeled with ϵ) and four call edges and two return edges. The symbolic matrix $\mathbf{A}^{sum} = \mathbf{A}^1 + \mathbf{A}^2 + \mathbf{A}^3$ represents the value-flow reachability with path lengths of 1, 2 and 3. The symbolic element $\mathbf{A}^{sum}_{2,4} = \epsilon * (_3 + (_2 *)_2$ represents two paths $\ell_2 \xrightarrow{\epsilon} \ell_3 \xrightarrow{(_3} \ell_4$ and $\ell_2 \xrightarrow{(_2} \ell_6 \xrightarrow{)_2} \ell_4$. According to our CFL-reachability analysis, both paths are realizable, one partially and the other fully-balanced. Similarly, the element $\mathbf{A}^{sum}_{2,7} = \left(\epsilon * (_3 + (_2 *)_2\right) * (_4$ represents two feasible paths from ℓ_2 to ℓ_7 with a common suffix $)_4$. However, $\mathbf{A}^{sum}_{1,4} = (_1 *)_2$ signifies an infeasible path from ℓ_1 to ℓ_4 because the string $(_1 *)_2$ is unbalanced, which violates our CFL grammar.

4.3 High-Order Proximity Embedding

After obtaining the matrices with different path lengths, this section details the high-order proximity embedding and decomposing the high-order proximity matrix into source and destination matrices for the nodes on the IVFG to preserve context-sensitive asymmetric transitivity of value-flows. We also mathematically measure the approximated error of our graph embedding in the low-dimensional space. The key is to choose an appropriate high-order proximity of the IVFG to precisely approximate the two embedding matrices:

$$\mathbf{M} \approx \mathbf{D}^s \cdot \mathbf{D}^{t\top}, \tag{2}$$

where $\mathbf{D}^s$ is the source embedding matrix (with N source vectors for N nodes) and $\mathbf{D}^t$ is the target embedding matrix (with N vectors).

As one of many high-order proximity measurements, the Katz Index [Katz 1953], which ensembles all paths between each pair in a directed graph, is a commonly used measurement for all-pair reachability:

$$\mathbf{M} = \sum_{h=1}^{H} (\beta \cdot \mathbf{A})^h, \tag{3}$$

where the high-order proximity matrix $\mathbf{M}$ is a weighted summarization over the paths between two nodes. A symbolic matrix $\mathbf{A}^h$ is concretized by mapping each of its symbolic elements to a numerical value based on CFL-reachability (Section 4.2.2). $\mathbf{M}$ adds a decay parameter β to Equation 1

to determine the speed at which the weight of a path decays when the length of a path extends. Therefore, a higher proximity value of an matrix element, e.g., $\mathbf{M}_{i,j}$ from node i to node j implies the more and the shorter feasible value-flow paths from i to j.

The final high-order proximity matrix $\mathbf{M}$ is then decomposed through SVD (Singular Value Decomposition) to produce the source and target embedding matrices:

$$\mathbf{M} = \mathbf{U}\Sigma\mathbf{V}^\top = \sum_{i=1}^{N} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top, \tag{4}$$

where σ_i is the i^{th} singular value, $\mathbf{u}_i$ and $\mathbf{v}_i$ are the corresponding left singular vector and right singular vector. The largest K singular values and the corresponding singular vectors are then used to generate the K-dimensional embedding vectors:

$$\mathbf{D}^s = \left[\sqrt{\sigma_1}\cdot\mathbf{u}_1, \cdots, \sqrt{\sigma_K}\cdot\mathbf{u}_K\right] = \left[\mathbf{d}_1^s, \cdots, \mathbf{d}_N^s\right] \tag{5}$$

$$\mathbf{D}^t = \left[\sqrt{\sigma_1}\cdot\mathbf{v}_1, \cdots, \sqrt{\sigma_K}\cdot\mathbf{v}_K\right] = \left[\mathbf{d}_1^t, \cdots, \mathbf{d}_N^t\right] \tag{6}$$

where K-dimensional vector $\mathbf{d}_i^s$ ($\mathbf{d}_i^t$) is the i-th row of $\mathbf{D}^s$ ($\mathbf{D}^s$) in the embedding space $\mathcal{R}^{N*K}$. The inner product between $\mathbf{d}_i^s$ and $\mathbf{d}_j^t$ (i.e., $\mathbf{d}_i^s \cdot \mathbf{d}_j^{t\top}$) is used to represent the proximity from node i to node j.

We have adopted the algorithm in [Ou et al. 2016] to approximate the embedding vectors directly, without the calculation of the high-order proximity matrix $\mathbf{M}$ if all the elements of $\mathbf{A}$ are numerical. The decay parameter β is set to be less than the spectral radius of $\mathbf{A}$ to ensure its invertibility. With the Jacobi-Davidson iteration of GSVD [Hochstenbach 2009], the total time complexity is $O(|E|\cdot K^2\cdot L)$, where $|E|$ is the number of edges on the IVFG, K is the embedding dimension and L is the number of iterations. When $K << N$, the time complexity is linear to the number of edges. Note that IVFG is usually a sparse graph [Choi et al. 1991; Hardekopf and Lin 2011], making $|E|$ much smaller than N^2.

Approximation error, recall and precision. Approximation errors (losses) are common in graph embedding techniques due to their approximation nature (e.g., dimensionality reduction to trade precision for efficiency) [Ou et al. 2016; Perozzi et al. 2014; Song et al. 2009; Tang et al. 2015]. This loss can cause imprecise and/or unsound preservation of value-flow reachability. The aim of this section is to mathematically define the approximation error of Flow2Vec to quantify its precision, with theoretical guarantees on the Root Mean Squared Error (RMSE), which is often missed by many blackbox learning-based approaches [Allamanis et al. 2018; Ben-Nun et al. 2018; Li et al. 2018; Zhou et al. 2019]. For a given high-order proximity matrix $\mathbf{M}$ and the learned approximation $\tilde{\mathbf{M}} = \mathbf{D}^s\cdot\mathbf{D}^{t\top}$, the error in the Frobenius Norm is calculated as $\left\|\mathbf{M}-\tilde{\mathbf{M}}\right\|_F = \sqrt{\sum_{i=K+1}^{N}\sigma_i^2}$, and the standard RMSE is:

$$RMSE = \sqrt{\frac{\left\|\mathbf{M}-\tilde{\mathbf{M}}\right\|_F^2}{N^2}} = \frac{\sqrt{\sum_{i=K+1}^{N}\sigma_i^2}}{N}, \tag{7}$$

where σ_i is the i^{th} singular value of $\mathbf{M}$ and K is the number of embedding dimensions. Additionally, for a given embedding dimension K, the lower the rank of $\mathbf{M}$ is, the smaller the error is. When K is equal to $rank(\mathbf{M})$, the error becomes zero. The precision and recall (soundness) are computed as follows:

$$Precision = \frac{|\mathbf{E}_c|}{|\mathbf{E}_p|}, \quad Recall = \frac{|\mathbf{E}_c|}{|\mathbf{E}_o|}, \tag{8}$$

where $\mathbf{E}_p$ is the set of predicted reachability between any two nodes, $\mathbf{E}_o$ is the set of observed reachability relations on the original graph, and $\mathbf{E}_c = \{(i,j) \mid (i,j) \in (\mathbf{E}_p \cap \mathbf{E}_o\})$ Note that the high-level reachability information can still be preserved with very high precision and soundness

given a small approximation error in the low-level embedding space, as also explained in Example 3 and evaluated in Figure 5.2 in Section 7.

Example 3. Let us revisit the motivating example to measure the approximation error of $\mathbf{D}^s$ and $\mathbf{D}^t$. The singular values for $\mathbf{M}$ are $\sigma = [1.92, 1.09, 0.51, 0.49, 0.00, 0.00]$. In a 3-dimensional embedding (i.e., $K = 3$), RMSE= $\frac{\sqrt{\sum_{i=4}^{6} \sigma_i^2}}{6} = 0.08$, while in the 2-dimensional embedding, RMSE= $\frac{\sqrt{\sum_{i=3}^{6} \sigma_i^2}}{6} = 0.12$. Given 0.25 as the threshold to determine the inter product between two embedding vectors, the 3-dimensional embedding perfectly preserves the graph's context-sensitive reachability with 100% precision and recall, while the 2-dimensional embedding achieves 97% precision and 97% recall in this example.

4.4 Applications of Flow2Vec

This section discusses the Flow2Vec's embedding results as inputs to boost the performance of two recent learning-based approaches, i.e., code2vec for code classification [Alon et al. 2019b] and code2seq for code summarization applications [Alon et al. 2019a].

4.4.1 ***Code classification.*** As the name implies, code classification aims to perform semantic labeling of a code fragment. This application is essentially a multi-class classification problem. Given a code fragment x (in the form of a program method), the objective is to train a prediction model to learn the label distribution conditioned on the code, i.e., $P(y_i|x)$, where the label y_i is from a pre-defined vocabulary Y, that is a set of tags/names in the training corpus. The predicted distribution of the model $q(y_i)$ is computed using a softmax function, the dot product between the code vector $\mathbf{v}$ of x and the vector representation lab_i of each label $y_i \in Y$.

$$for\ y_i \in Y : q(y_i) = \frac{\exp(\mathbf{v}^\top \cdot lab_j)}{\sum_{y_j \in Y} \exp(\mathbf{v}^\top \cdot lab_j)}, \tag{9}$$

Unlike Word2Vec-like techniques [Mikolov et al. 2013] which use textual code tokens to produce the code vector, whereas the code2vec [Alon et al. 2019b] approach leverages the structural features, that is, the abstract syntax tree (AST) of each program method. The approach aggregates multiple syntactic paths on an AST into a single code vector representation $\mathbf{v}$, which is then used for multi-class classification. To further improve the precision, code2vec includes an attention mechanism to focus on important AST paths that contribute more to predication accuracy [Allamanis et al. 2016; Luong et al. 2015].

However, by its nature, a tree data structure is an undirected graph that can not preserve asymmetric transitivity. Moreover, the paths on an AST can not reflect the alias-aware data-flow information. To demonstrate that Flow2Vec can preserve more comprehensive structural information than code2vec, we embed value-flow paths rather than AST paths to generate the code vector for code classification. Given the asymmetric vectors $\mathbf{d}_i^s \in \mathbf{D}^s$ and $\mathbf{d}_j^s \in \mathbf{D}^t$ for a code fragment computed by Equations 5 and 6, the path vector $\mathbf{c}_n$ of a value-flow path vfp_n from statement ℓ_i to ℓ_j is:

$$\mathbf{c}_n = \tanh(\mathbf{W} \cdot embed(\langle \ell_i, vfp_n, \ell_j \rangle)) = \tanh(\mathbf{W} \cdot [\ell_i; \mathbf{d}_i^s \cdot \mathbf{d}_j^{t\,\top}; \ell_j]) \tag{10}$$

where $\mathbf{W}$ is a learned weights matrix that serves as a fully connected layer before aggregating all the path vector and *tanh* is the monotonic nonlinear activation function. The value of the dot product $\mathbf{d}_i^s$ and $\mathbf{d}_j^s$ indicates two important distance relations, i.e., the length of each path and the number of paths between i and j, relative to all other value-flows in the embedding space. A higher proximity value means the more and the shorter paths from i to j.

The final code vector $\mathbf{v}$ is an aggregation of all the path vectors constructed by embedding S value-flow paths between any two nodes on the IVFG of a code fragment:

$$\mathbf{v} = \sum_{n=1}^{S} \alpha_n \cdot \mathbf{c}_n, \tag{11}$$

where $\alpha_n = \frac{\exp(\mathbf{c}_n{}^\top \cdot \mathbf{a})}{\sum_{j=1}^{S} \exp(\mathbf{c}_j{}^\top \cdot \mathbf{a})}$ is the attention weight of vector $\mathbf{c}_n$ and $\mathbf{a}$ is a global attention vector learned during model training [Alon et al. 2019b].

4.4.2 ***Code summarization.*** The aim of code summarization is to translate a code fragment into a sequence of words to describe the semantic meaning of the code. The summarization is performed under an encoder-decoder framework, which has been widely used in the neural machine translation [Luong et al. 2015] and image captioning [Xu et al. 2015].

The encoder first transforms the code snippet $\mathbf{x}$ into a sequence of hidden states in the form of a code vector $\mathbf{v}$, while the decoder generates one word y_{t+1} at a time step. Generating the summary is conditioned on all previously generated words $y_{1:t}$ and the hidden states from the encoder. Hence, the objective of code summarization is to model the conditional probability: $P(y_1, \ldots, y_m \mid \mathbf{x}) = \prod_{t=1}^{m} P(y_t \mid y_{1:t-1}, \mathbf{v})$.

As with the code classification task, the value-flow paths are encoded to generate the code vector $\mathbf{v}$ as opposed to the AST paths from CODE2SEQ [Alon et al. 2019a]. Note that, to improve precision, both approaches use the same attention mechanism to produce the final attentional code vector.

5 EVALUATION

The objective of this evaluation is to determine whether FLOW2VEC is effective at preserving value-flows in low dimensional embedding space. For this, we evaluate two aspects of FLOW2VEC: (1) approximation error, precision and recall through graph reconstruction, and (2) the performance improvement over CODE2VEC and CODE2SEQ on code classification and code summarization tasks.

5.1 Dataset and Evaluation Methodology

Implementation and Datasets. The interprocedural alias-aware value-flow graph of a program is built on top of LLVM-IR (with LLVM version 9.0.0) and its sub-project SVF [Sui and Xue 2016]. Program functions, which are represented as global variables in the LLVM-IR, are modeled as address-taken variables (Section 2.1). The callgraph of a program is built on-the-fly (with indirect calls discovered) when performing Andersen's pointer analysis [Andersen 1994]. The value-flows across procedures are soundly captured on top of the conservative callgraph built using Andersen's analysis. We have implemented the CFL-reachability algorithm (Section 4.2.2) to recognize realizable context-sensitive value-flow paths on IVFG. Katz Index [Katz 1953] is used to produce high-order proximity matrix, through which the asymmetric embedding vectors are generated.

We evaluated FLOW2VEC using 32 popular open-source C/C++ projects (with their statistics shown in Table 1) across a variety of domains: `a2ps` (postscript filter tool), `bash` (sh-compatible shell), `bc` (basic calculator), `bison` (parser generator), `ctypes_test` (builtin test tool), `dc` (desk calculator), `decimal` (python tool), `echogs` (port upgrate), `gdbserver` (remote debugging), `gzip` (file compress), `grep` (commandline search tool), `hashcat` (passwd recovery), `htop` (process viewer), `keepalived` (IP link checker), `less` (pager program), `lua` (interpreter), `libsass` (sass engine), `make` (executable builder), `msg-convert` (messagepack switcher), `msg-structure` (messagepack pattern former),`msg-map` (messagepack multi-keys API), `mkromfs` (file system), `mocktracer` (a code tracer), `ossaudiodev` (python audio API), `redis-client` (database client), `redis-server` (database server), `redis-benchmark` (redis benchmark), `sample` (Unix selection tool) `screen` (terminal emulator),

Table 1. The statistics of the open-source programs. #LOI denotes the number of lines of LLVM instructions. #Function, #Pointer, #Object, #Call, |V| and |E| are the numbers of functions, pointers, objects, method calls, IVFG nodes and IVFG edges, respectively.

	#LOI	#Method	#Pointer	#Objects	#Call	|V|	|E|
a2ps	118173	490	94173	6912	9036	122345	210783
bash	312144	783	235887	17630	24998	372242	503357
bc	18683	86	14744	909	1583	23207	32202
bison	98575	400	79820	3404	15978	160506	237364
ctypes_test	1467	93	1275	150	232	1203	1013
dc	11629	97	9167	871	1132	14651	17825
decimal	72546	193	57496	3278	22291	85989	98107
echogs	1034	18	743	46	146	589	575
gdbserver	64324	348	53196	3225	15091	108522	119497
gzip	32411	136	24944	1950	2030	32846	49806
grep	37796	221	28134	1639	8141	39532	52788
hashcat	177602	412	150885	5318	16281	234648	459262
htop	62554	293	29170	1456	7416	37367	43980
keepalived	214163	575	99354	3066	24142	162032	215899
less	37372	224	28843	2524	3333	81584	91222
lua	68222	346	52585	1869	4764	100701	106830
libsass	818989	2734	605654	39805	47343	553499	882694
make	42676	190	33200	2027	7209	82157	99750
msg-convert	241287	449	18417	3447	1757	25854	28499
msg-structure	222986	498	20065	3856	1873	35354	35650
msg-map	293360	640	24370	4784	2204	24876	34965
mkromfs	18751	155	15614	589	3196	19054	28448
mocktracer	214038	3688	176632	40666	19342	197830	300333
ossaudiodev	3107	61	2990	302	769	2833	2481
redis-client	322115	557	172664	5262	66297	241070	284082
redis-server	735275	1875	401520	14461	119216	654495	778854
redis-benchmark	299719	505	183609	4527	59041	253210	295056
sample	69720	383	55608	4797	5965	65894	94460
screen	117552	286	90984	3601	13899	321269	597668
sed	40708	177	30401	1570	5944	52405	83748
sendmail	153184	616	121604	6216	25384	529537	744380
tar	87418	386	66191	5052	17279	183538	207659
Total	4922162	17529	2913748	190157	536033	4637301	6531578

`sed` (Unix editor), `sendmail` (mail agent), `tar` (file compression). Our experiments were conducted on a machine with Intel Xeon Gold 6132 @ 2.60GHz CPUs and 128GB of RAM.

Evaluation Methodology. Like all existing embedding approaches, Flow2Vec does not aim to achieve 100% precision and recall, but rather to obtain a comprehensive code representation to better support subsequent client applications. To validate Flow2Vec's effectiveness, we conducted three evaluation tasks, graph reconstruction, code classification and code summarization.

Graph reconstruction is a typical method to evaluate graph embedding results [Ou et al. 2016; Perozzi et al. 2014; Tang et al. 2015]. We reconstructed the graph edges on the IVFG of each of our 32 benchmarks based on the reconstructed proximity, i.e., the inner products between the embedding vectors for each node pair. Note that H and β in Equation 3 were set to be 3 and 0.01 to produce the proximity matrix. Nodes with no incoming or outgoing edges on the IVFG were not included in the proximity matrix to reduce computational overhead. Following [Ou et al. 2016], we calculated the ratio of real links for the top k pairs of nodes as the reconstruction precision by randomly sampling about 0.1% of the node pairs to evaluate the approximation error, precision and recall for graph reconstruction.

To further validate the effectiveness and practicality of our approach, we choose two important and challenging clients, classification and summarization, for which prior results still have much room for improvement [Allamanis et al. 2016; Alon et al. 2019a, 2018, 2019b; Iyer et al. 2016]. We then compared Flow2Vec with the two recent open-source tools code2vec [Alon et al. 2019b] and code2seq [Alon et al. 2019a]. From the 32 open-source projects, we extracted 11,291 program methods with informative comments, including the method names and the comments of each method, with the AST path of each method extracted by pathminer [Kovalenko et al. 2019]. A method is not used for our training if it does not have any comment or it is an unused method in the project (i.e., it never calls other methods or being called by others). For both clients, we randomly split our collected programs into three subsets, of which the proportions are about 80%, 10% and 10% for training, validation and testing, respectively. Note that code2vec and code2seq have already shown that they have better performance compared with CodeNN [Iyer et al. 2016], ConvAttention [Allamanis et al. 2016], Paths+CRFs [Alon et al. 2018] and TreeLSTM [Tai et al. 2015], therefore, we did not compare Flow2Vec with these approaches.

To produce the path vectors based on Equation 10, we collected a set of IVFG nodes (with each node being a definition of a variable) for each program method. The context-sensitive value-flows between any two nodes are represented by the dot product of the nodes' embedding vectors. Note that the value-flow paths between two nodes in the same method can be interprocedural (e.g., p and *stack* in Figure 3(a)). Following [Alon et al. 2019b], the value-flow paths are randomly sampled for each method (i.e., S = 200 in Equation 11). The path vectors are then aggregated into a fixed-length code vector using the attention mechanism (Equation 11). The attention network is trained in a batch-wise fashion and the batch size is set to 512. The dropout is set to 0.25 and the number of epochs is set to 20. ADAM [Kingma and Ba 2015], a commonly-used adaptive gradient descent method, is used for training. We used grid search to perform hyper parameter tuning in order to determine the optimal values for a given model. Note that our focus here is on evaluating the effectiveness of our value-flow embedding, not on different backend learning models. Picking different models and feeding Flow2Vec's embedding vectors into these backend models to further enhance the performance is an orthogonal topic.

The evaluation metrics used to assess code classification and summarization performance are precision, recall and F1-score. The generated sequence of words were evaluated in a case-insensitive manner and calculated over (sub)tokens following previous studies [Allamanis et al. 2015, 2016; Alon et al. 2019a,b]. The key idea is to measure the quality of a sequence prediction by decomposing it into sub-tokens and calculate correct predictions based on them. Take method name prediction for example, given a method named `swapElements`, we first decompose it into `swap` and `elements`. A prediction of `elementsSwap` is an exact match, while a prediction of `swapAbstractElements` has a full recall but low precision and a prediction of `swap` has a full precision but low recall. Furthermore, the unseen sub-token in the token vocabulary from the test label is considered as a false negative. We used a similar strategy to evaluate the code summarization by measuring precision and recall based on tokens of the generated code comments.

5.2 Experimental Results and Analysis

Precision, recall and RMSE. Figure 7 gives the precision, recall and approximation error of the reconstructed graph based on Equations 8 and 7 for each of the 32 projects. RMSE is normally used when evaluating graph embedding techniques [Ou et al. 2016; Song et al. 2009] to measure the gaps between the prediction result and the ground truth. A smaller RMSE indicates a better embedding result. During value-flow embedding in the latent space, we assessed different feature dimensions, ranging from 10 to 210.

From the results in Figure 7, we can see that the error rate (RMSE) of each program decreases dramatically, while the precision and recall increase very quickly, particularly, in the beginning part (less than 60 dimensions). The results demonstrate that Flow2Vec is effective at preserving the asymmetric transitivity with low-dimensional embeddings. With 110 dimensions, the average error, precision and recall for the 32 projects were $2.46^{*}10^{-4}$, 96.9% and 95.5% respectively and an error up to $8.3^{*}10^{-4}$ for program `mocktracer`. Additionally, the RMSE was very small, but precision and recall are high both for the smallest program, i.e., `ossaudiodev` with $1.58^{*}10^{-4}$ error, 0.99 precision and 0.99 recall, and the largest program, i.e., `libsass` with $6.2^{*}10^{-4}$, 0.83 and 0.87, respectively. These results confirm that Flow2Vec precisely preserves asymmetric transitivity of value-flows in the embedding space.

Efficiency. Figure 8 shows the total running time (in seconds) of Flow2Vec, including the times for value-flow construction (red) and high-order proximity embedding (blue). The embedding dimension was set to 110. In summary, Flow2Vec finished analyzing the largest program `libsass` within 64.2 mins and for all programs consisting of over 5 million lines of LLVM instructions with 272.5 mins, showing that Flow2Vec scales to large projects for value-flow construction and graph embedding using fast high-order proximity computation. Note that although our interprocedural embedding consumes longer analysis time than intraprocedural analysis (e.g., collecting AST paths of individual methods [Alon et al. 2019a,b]), the embedding process is only done once to produce the distributed code representation before feeding into subsequent learning tasks. Moreover, the increase in precision, recall and F1 outweighs the interprocedural analysis overhead as also illustrated in Table 2 and Figure 9.

Code classification. The second and third Columns in Table 2 compare the results of Flow2Vec and code2vec for code classification. It is clear that Flow2Vec significantly outperforms code2vec in terms of precision, recall and F1-score with an average improvement of 21.2%, 20.1% and 20.7%, respectively. This confirms that our comprehensive code representation yields better results than the approaches that extract shallow code structure information to predict the names of code fragments.

Figure 9 gives the changes of the F1-scores as the size of a program method grows. Flow2Vec gave the best (or equal best) results for all code lengths. With small lengths of code (e.g., 1-10 lines) both code2vec and Flow2Vec observe their best performance. The F1-scores of both approaches decreased as the size of the method increased. However, at larger sizes, i.e., 35 lines and above, Flow2Vec was more stable and performed consistently better than code2vec. Note that code2vec's F1-score in Table 2 is lower than that presented in the code2vec paper, in which code classification was performed on Java and C# programs. There are two reasons behind. First, the method body of a program in C/C++ is generally larger than that in Java or C#, where the value-flows of each code fragment is simpler. In our C/C++ dataset, the average length of a method is around 35 lines, while in code2vec's dataset, it is around 7 lines per method. The F1-score only becomes stable in after round 35 lines as shown in Figure 9. Second, from our manual investigation of our code bases, we found that many methods in C/C++ have more complicated data and control dependence

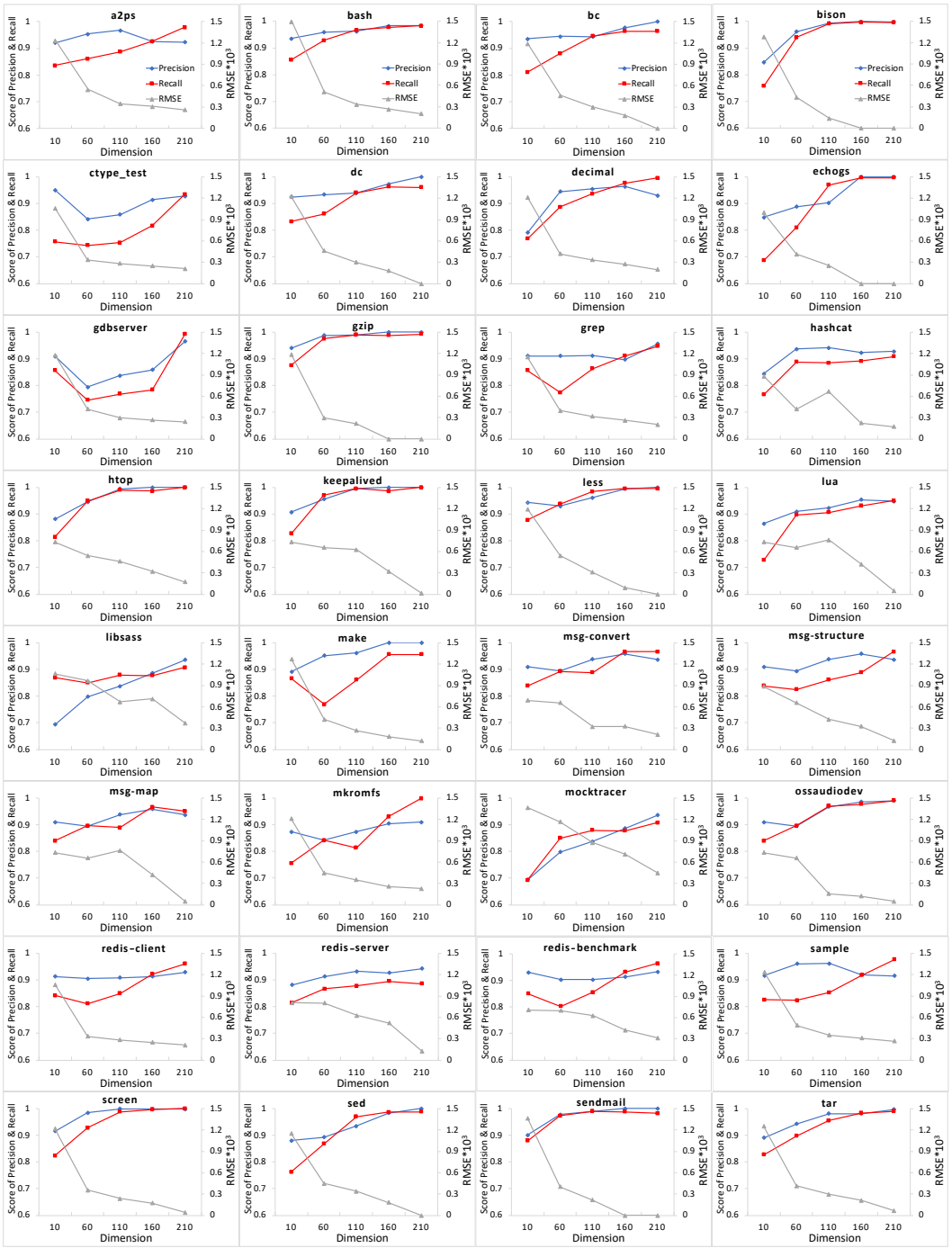

Fig. 7. Precision, Recall and RMSE of Flow2Vec under different embedding dimensions after graph reconstruction.

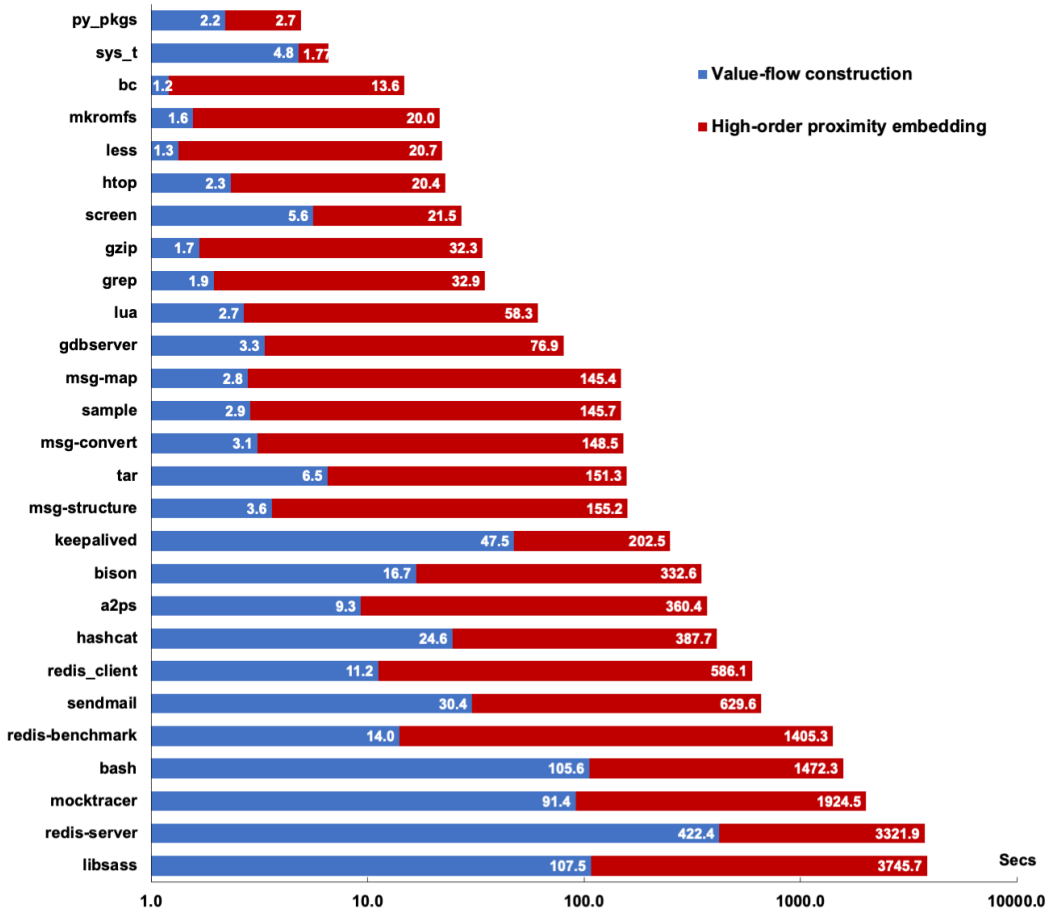

Fig. 8. Total running time (secs).

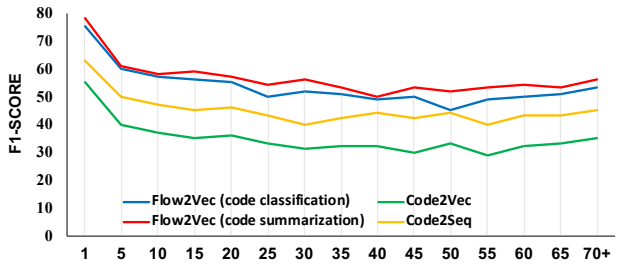

Fig. 9. F1-score under different the lengths of code.

than those in Java or C# programs, which makes code classification more difficult for C/C++ than Java/C#. It also demonstrates the necessity for a more comprehensive code representation.

Table 2. Precision, recall and F1 for the two applications.

Client	Model Name	Precision(%)	Recall(%)	F1-score(%)
Code Classification	CODE2VEC	35.5	34.2	34.8
	FLOW2VEC	**56.7**	**54.3**	**55.5**
Code Summarization	CODE2SEQ	43.9	41.1	42.5
	FLOW2VEC	**57.1**	**59.9**	**58.5**

Code summarization. The second task we used for comparison was code summarization, pitting FLOW2VEC against CODE2SEQ [Alon et al. 2019a] using the same set of methods with their corresponding comments. In Table 2, we can see that FLOW2VEC performed better than CODE2SEQ in all the three metrics. FLOW2VEC's F1-score was 58.5% compared to 42.5% of CODE2SEQ. FLOW2VEC's precision and recall were also higher, at 57.1% and 59.9% respectively, while CODE2SEQ only achieved 43.9% and 41.1% respectively.

Figure 9 shows the F1-scores for FLOW2VEC and CODE2SEQ with different code lengths. The trend here is similar to the comparison with CODE2VEC. Overall, FLOW2VEC outperformed CODE2SEQ for all code lengths. The F1-scores for both FLOW2VEC and CODE2SEQ were at or above 50% with small code lengths (i.e. under 5 lines), but both became stable as the length increases. Notably, FLOW2VEC delivered similar performance for both code classification and code summarization, whereas CODE2SEQ provided a much better result at code summarization than CODE2VEC. This is because we used the identical structural information of code (i.e., value-flow paths) for both tasks. Additionally, CODE2VEC encodes AST paths monolithically, so it cannot represent paths unless they are included in the path vocabulary, while CODE2VEC improves this by using a bi-directional LSTM to embed AST paths token-by-token. Thus, it can represent any syntactic path assembled from a limited size of path token vocabulary.

Note that CODE2VEC and CODE2SEQ perform path analysis on top of ASTs of a method through intra-procedural analysis and their dataset contains only disparate methods with each averaging 7 lines, rather than complete projects, making embedding less challenging under the small-code-length setting. Analyzing them one at a time, though the large number, is less costly than analyzing them together. However, adding more methods is subject to diminishing returns on precision. The limitations of its intraprocedural approach are further exemplified when evaluated against our complex large-scale projects where FLOW2VEC's interprocedural embedding yields better results by extracting substantial structural information of code, i.e., flow- and context-sensitive value-flows. The result is a more comprehensive code representation and, in turn, improved performance for code classification and summarization tasks.

Embedding dimension. Figure 10 further evaluates FLOW2VEC's F1-scores under different feature dimensions during our value-flow embedding. There are similar trends in both tasks with an increased F1-score by around 17% from dimension 10 to 110. The F1-scores are capped at around 56.0% and 59.0% for code classification and summarization, respectively, when the number of dimensions become even larger. This is consistent to the trends of the approximation error, precision and recall when performing graph reconstruction under different feature dimensions, where the results become stable and precise enough when the number of dimensions is around 110, a relatively low dimensional space. Note that the performance did not improve much when over 110 dimensions. This is due to the nature of the distributed representation of code whose semantic meaning is distributed across its vector components (dimensions). In our evaluation, 110

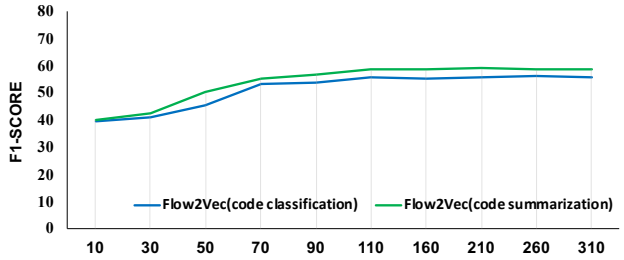

Fig. 10. F1-score under different embedding dimensions.

dimensions is already precise enough to preserve the value-flows of a program in the latent space. Adding more dimensions yields diminishing returns in performance as shown in Figure 10.

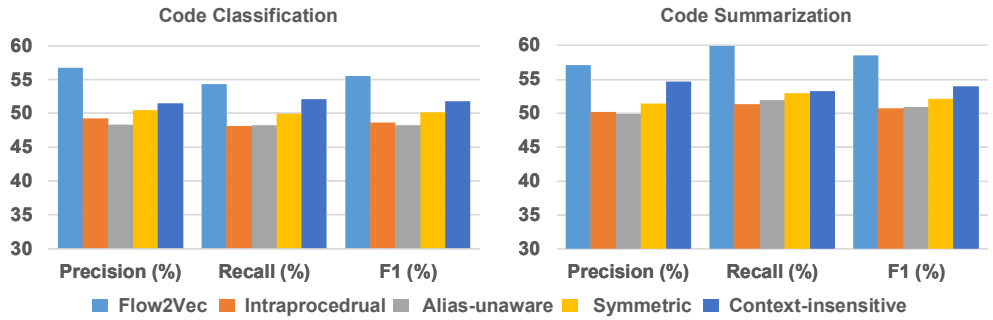

Fig. 11. Ablation analysis for inter-procedural analysis, pointer aliases, asymmetric transitivity and context-sensitivity.

Ablation analysis. We conducted ablation analysis to evaluate each of Flow2Vec's four properties, i.e., interprocedural analysis, aliases, asymmetric-transitivity and context-sensitivity, to further understand their contributions to the final accuracy for the two tasks. We evaluated Flow2Vec's performance by removing each property at a time. **(1)** We removed the method-call edges to see the influence of inter-procedural information. **(2)** We removed the indirect value-flow edges (Section 2.2) to remove the pointer analysis results to observe the effectiveness of pointer aliases. **(3)** We transformed the adjacency matrix of a value-flow graph to symmetric matrix when doing graph embedding to evaluate the importance of asymmetric transitivity. And **(4)** we produced a context-insensitive matrix rather than a symbolic matrix for the IVFG of a program without performing CFL-reachability analysis in order to evaluate the context-sensitive property.

Figure 11 shows the result of ablation analysis for each property of Flow2Vec. Overall, Flow2Vec (with four properties combined) performs better when one of its properties is disabled, with an improvement ranging from 3.7% to 7.3% and 4.6% to 7.8% in terms of F1-score for code classification and summarization respectively. It also demonstrates that the usefulness of all the four properties for producing the final distributed code representation. The properties of interprocedural analysis and pointer aliases have relatively high contributions to the precision/recall/F1 compared to the other two properties when evaluating using our dataset. By looking into the methods collected from our real-world projects, 91.0% methods have inter-procedural method calls. On average, there

Souce code (A)

```
int ____________(dict *d){
  int minimal;
  minimal = d->ht[0].used;
  if (minimal<INITIAL_SIZE) minimal=INITIAL_SIZE;
  return dictExpand(d, minimal);
}
int dictExpand(dict *d, unsigned long size){
  dictht n;
  unsigned long realsize = _dictNextPower(size);
  n.table = zcalloc(realsize*sizeof(dictEntry*));
  return DICT_OK;
}
```

Ground truth	resize to the minimal
Code2Vec	isMinimal
Code2Seq	Expand size
Flow2Vec	reset minimal memory

Souce code (B)

```
SIG __________(char* inp){
    if (!is_valid(inp)) return FAULT;
    CMD *cmd = parse(inp);
    return process(cmd);
}
SIG process(CMD *cmd){
    Executor *e = get_glb_executor();
    if(e->is_full) return FULL;
    e->execute(cmd);
    return SUCESS;
}
```

Ground truth	parse and execute command
Code2Vec	check
Code2Seq	parse
Flow2Vec	parse command and execute

Souce code (C)

```
void _____________(NODE *cur_node){
    NODE *c = fd_node(cur_node);
    NODE *n = fd_node(c->next);
    char * value = node_value_get(c);
    if (n != NULL)
      node_value_set(n, value);
}
```

Ground truth	set current value to next node
Code2Vec	getNodeValue
Code2Seq	set node value
Flow2Vec	set next node current value

Souce code (D)

```
char* ________(const char* name){
  Dir* directory = find_dir(name);
  if(directory==NULL) return NULL;
  Pro* p = init_pro(directory);
  Dir* r = p->get_root();
  return generate_file(r, p->sz);
}
Pro* init_pro(Dir* dir){
  Pro* program = create_pro();
  program->set_root(dir);
  return program;
}
```

Ground truth	generate file under directory
Code2Vec	generateFile
Code2Seq	get file
Flow2Vec	generate file in directory

Fig. 12. Case studies.

are 30 interprocedural value-flows coming in and going out of a method. The 91.0% methods have indirect value-flows with an average of 37 indirect value-flows per method. It demonstrates the importance for considering interprocedural and alias-aware value-flows for code embedding. When looking at the other two properties (i.e., asymmetric-transitivity and context-sensitivity), their precision benefit is slightly lower but the average improvement still reaches approximately 5% in terms of F1-score. The results confirm that all four properties are essential to comprehend and preserve the structural feature of source code to deliver better results for complicated tasks (e.g., code classification and summarization).

Case study. Figure 12 gives four real-world code snippets together with the ground truth of their corresponding code comments, which are extracted from `redis-server`, `bison`, `bash` and `libsass` respectively. Source code (A) and (B) are used to demonstrate the effectiveness of Flow2Vec's interprocedural analysis. Source code (C) and (D) are used to show the usefulness of context-sensitivity and pointer aliases preserved by Flow2Vec.

The first method of source code (A) is to resize `hash table` to the minimal size that contains all the elements. It can be seen that Flow2Vec can precisely capture the key words `minimal` and `resize` from the ground truth, while code2vec failed to predict the token `resize` and code2seq can not distinguish `resize` and `expand`. This is because the semantic of the current method depends on

its callee `dictExpand`. code2vec and code2seq generated partial and imprecise results due to their intra-procedural analysis. On the other hand, Flow2Vec's inter-procedural analysis successfully comprehended and incorporate the callee information into the summary, yielding more precise prediction results.

The first method of in source code (B) is to parse a command sequence and then invoke a global executor to execute the command. It is clear that Flow2Vec produced a more precise prediction by capturing the three key words `parse`, `command` and `execute` through analyzing value-flows across methods, while code2seq and code2vec failed to capture all these important tokens due to its intra-procedural analysis nature.

The method in source code (C) is similar to our motivating example in Figure 3. The code snippet is a helper method used to set the value of next node given current node's value. As can be seen from the prediction results of Flow2Vec and the two baselines, the summary produced by Flow2Vec is more precise because it distinguishes the different operations of current node and the next node. The prediction result is closer to the ground truth since our context-sensitive analysis is able to recognize that operations on object `c->next` only occur through pointer `n`. Similarly, object `cur_node` only operates through `c` when considering different calling contexts to callee `fd_node`. code2vec and code2seq, however, are unable to differentiate the current node from its next node. Because the intra-procedural paths on the AST of the code (as also explained in Figure 4) will establish false relations between `c` and `c->next` and `n` and `cur_node`, as such, the final results are imprecise.

The method presented in source code (D) is to generate files under a given directory. Flow2Vec precisely identified the key word `directory` by capturing the alias relation between `directory` and `r` through `set_root` and `get_root` to infer the more expressive name `directory` of the first parameter at callsite `generate_file`. However, code2vec and code2seq which failed to capture this complicated interprocedural alias relation, produced the imprecise summary result.

6 RELATED WORK

We limit our discussion to the work that is most relevant to Flow2Vec, namely, program dependence analysis, graph embedding and code embedding.

Program dependence analysis. Program dependence analysis or value-flow analysis, which reasons about the control- and data-flows of a program, has been well-studied over the past decades. Initially, the program dependence analysis has been used for compiler optimizations [Ferrante et al. 1987; Kuck et al. 1981]. Later, the analysis has been shown as an enabling technique for a wide variety of client applications, such as program slicing [Weiser 1981], software maintenance [Gallagher and Lyle 1991], change impact analysis [Acharya and Robinson 2011], program complexity metrics [Rilling and Klemola 2003] and precise software bug detection [Livshits and Lam 2003; Shi et al. 2018]. By leveraging the advances in pointer alias analysis [Barbar et al. 2020; Hardekopf and Lin 2007, 2011], some recent works [Shi et al. 2018; Sui and Xue 2016] have developed a more sound and precise approach to program dependence analysis by considering memory aliases. The result is a so-called sparse value-flow graph representation that captures the def-use relations of both the top-level and address-taken variables. Flow2Vec is the first technique that can embed a comprehensive sparse value-flow representation into low-dimensional space while preserving asymmetric value-flow transitivity to support two important clients, code classification and summarization.

Graph Embedding and code embedding. Graph embedding transforms nodes in a graph into distributed representations and effectively preserves the network structure. There have been some significant advances recently in graph embedding in a wide range of applications, including classification [Sen et al. 2008], clustering [Wang et al. 2017], link prediction [Wang et al. 2014]

and data representation for machine learning methods. For many software analysis tasks, directed graphs are the major representation for source code. Preserving asymmetric transitivity of graphs is the fundamental requirement for an accurate graph embedding. HOPE [Ou et al. 2016] approximates the high-order proximity to preserve the high-order structure of directed graphs. Later, ATP [Sun et al. 2019] has subsequently emerged to improve upon HOPE with an ability to solve cycles on a directed graph during high-order proximity embedding. Inst2Vec [Ben-Nun et al. 2018] presents a distributed representation of code statements based on contextual data-flow on top of LLVM IR. However, the underlying data-flow graph used is intraprocedural and alias-unaware. Allamanis et al. [Allamanis et al. 2018] proposes a code embedding technique that embeds a program's data-flows using gated graph neural networks (GGNNs) for analyzing variable renaming and misuses in C# programs. This approach is alias-unaware and the underlying embedding technique does not support context-sensitivity. DeepSim [Zhao and Huang 2018] transforms the control- and data-flows of a program using a feed-forward neural network to measure functional code similarity. ASTNN [Zhang et al. 2019] is an intraprocedural and context-insensitive embedding technique mainly for code clone detection based on the ASTs of a program. Like CODE2VEC [Alon et al. 2019b] and CODE2SEQ [Alon et al. 2019a], the embedding does not preserve a program's data- and control-flows.

7 CONCLUSION

This paper presents FLOW2VEC, a new code embedding approach that preserves interprocedural, context-sensitive and alias-aware value-flows in the low-dimensional vector space. The value-flow reachability is formulated as a chain-matrix-multiplication problem to support filtering out infeasible value-flow paths through CFL-reachability. The resulting embedding vectors can be used as a comprehensive code representation to better support subsequent learning tasks. We have evaluated FLOW2VEC using 32 popular open-source projects with over 5 million lines of code. When evaluated using graph reconstruction, FLOW2VEC achieved a high precision (96.9%) and recall (95.5%) by successfully preserving interprocedural alias-aware value-flows. When evaluated using the two client applications, FLOW2VEC significantly boosted the performance of CODE2VEC and CODE2SEQ, two recent embedding approaches. For code classification, FLOW2VEC outperformed CODE2VEC with an average increase of 21.2% in precision, 20.1% in recall, and 20.7% in F1. For code summarization, FLOW2VEC improved CODE2SEQ by an average of 13.2% in precision, 18.8% in recall, and 16.0% in F1.

8 ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their helpful comments. This research is supported by Australian Research Grants DP200101328.

REFERENCES

Mithun Acharya and Brian Robinson. 2011. Practical change impact analysis based on static program slicing for industrial software systems. In *ICSE'11*. ACM, 746–755.

Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *FSE'15*. 38–49.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. (2018).

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *ICML '16*. 2091–2100.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019a. code2seq: Generating sequences from structured representations of code. *ICLR '19* (2019).

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-Based Representation for Predicting Program Properties. In *PLDI '18*. 404–419. https://doi.org/10.1145/3192366.3192412

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019b. code2vec: Learning distributed representations of code. *ACM POPL* 3 (2019), 40.

Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph.D. Dissertation. University of Cophenhagen.

George Balatsouras and Yannis Smaragdakis. 2016. Structure-sensitive points-to analysis for C and C++ . In *SAS '16*. Springer, 84–104.

Mohamad Barbar, Yulei Sui, and Shiping Chen. 2020. Flow-Sensitive Type-Based Heap Cloning. In *ECOOP '20*.

Mikhail Belkin and Partha Niyogi. 2002. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *NeurIPS '02*. 585–591.

Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: a learnable representation of code semantics. In *NeurIPS '18*. 3585–3597.

Rastislav Bodík and Sadun Anik. 1998. Path-sensitive value-flow analysis. In *POPL '98*. 237–251.

Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. 2018. A comprehensive survey of graph embedding: Problems, techniques, and applications. *TKDE '18* 30, 9 (2018), 1616–1637.

Gerardo Canfora and Luigi Cerulo. 2005. Impact analysis by mining software and change request repositories. In *METRICS '05*. IEEE, 9—-pp.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *NeurIPS '18*. 2547–2557.

Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. 1991. Automatic construction of sparse data flow evaluation graphs. In *POPL '91*. 55–66.

Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. 1996. Effective representation of aliases and indirect memory operations in SSA form. In *CC '96*. Springer, 253–267.

Peng Cui, Xiao Wang, Jian Pei, and Wenwu Zhu. 2018. A survey on network embedding. *TKDE* 31, 5 (2018), 833–852.

Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. 2000. A multilinear singular value decomposition. *SIAM journal on Matrix Analysis and Applications* 21, 4 (2000), 1253–1278.

Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM TOPLAS* 9, 3 (1987), 319–349.

Georgia Frantzeskou, Stephen MacDonell, Efstathios Stamatatos, and Stefanos Gritzalis. 2008. Examining the significance of high-level programming features in source code author classification. *Journal of Systems and Software* 81, 3 (2008), 447–460.

Keith Brian Gallagher and James R Lyle. 1991. Using program slicing in software maintenance. *IEEE TSE* 17, 8 (1991), 751–761.

Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *KDD '16*. ACM, 855–864.

Ben Hardekopf and Calvin Lin. 2007. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI '07*. ACM, 290–299.

Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *CGO '11*. 289–298.

Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *ICSE '12*. IEEE, 837–847.

M E Hochstenbach. 2009. A Jacobi–Davidson type method for the generalized singular value problem. *Linear Algebra Appl.* 431, 3-4 (2009), 471–487.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *ICPC '18*. 200–210.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *ACL '16*. Berlin, Germany, 2073–2083. https://www.aclweb.org/anthology/P16-1195

Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE TSE* 28, 7 (2002), 654–670.

Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* 18, 1 (1953), 39–43.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR '15*, Yoshua Bengio and Yann LeCun (Eds.).

John Kodumal and Alex Aiken. 2004. The set constraint/CFL reachability connection in practice. *PLDI '04* 6 (2004), 207–218.

Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. 2019. PathMiner: a library for mining of path-based representations of code. In *MSR '19*. 13–17.

David J Kuck, Robert H Kuhn, David A Padua, Bruce Leasure, and Michael Wolfe. 1981. Dependence graphs and compiler optimizations. In *POPL '81*. ACM, 207–218.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*. IEEE, 75–86.

Yuxiang Lei and Yulei Sui. 2019. Fast and precise handling of positive weight cycles for field-sensitive pointer analysis. In *SAS '19*. Springer, 27–47.

Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-To Analysis with Efficient Strong Updates. In *POPL '11*. 3–16.

L Li, C Cifuentes, and N Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *FSE '11*. 343–353.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. *NDSS '18* (2018).

Defu Lian, Kai Zheng, Vincent W Zheng, Yong Ge, Longbing Cao, Ivor W Tsang, and Xing Xie. 2018. High-order proximity preserving information network hashing. In *KDD '18*. ACM, 1744–1753.

V Benjamin Livshits and Monica S Lam. 2003. Tracking pointers with path and context sensitivity for bug detection in C programs. *FSE '03* 28, 5 (2003), 317–326.

M T Luong, H Pham, and C D Manning. 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* (2015).

Chris Maddison and Daniel Tarlow. 2014. Structured generative models of natural source code. In *ICML '14*. 649–657.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *NeurIPS '13*. 3111–3119.

Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. 2016. Asymmetric transitivity preserving graph embedding. In *KDD '16*. ACM, 1105–1114.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *KDD '14*. ACM, 701–710.

Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *PLDI '14*. ACM, 419–428.

Thomas Reps. 1998. Program analysis via graph reachability. *IST* 40, 11-12 (1998), 701–726.

Juergen Rilling and Tuomas Klemola. 2003. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In *IEEE International Workshop on Program Comprehension*. IEEE, 115–124.

Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling code clone detection to big-code. In *ICSE '16*. IEEE, 1157–1168.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective classification in network data. *AI magazine* 29, 3 (2008), 93.

Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *PLDI '18*. ACM, 693–706.

Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. 2010. Do I use the wrong definition?: DeFuse: definition-use invariants for detecting concurrency and sequential bugs. *OOPSLA '10* 45, 10 (2010), 160–174.

Han Hee Song, Tae Won Cho, Vacha Dave, Yin Zhang, and Lili Qiu. 2009. Scalable proximity estimation and link prediction in online social networks. In *ACM SIGCOMM*. ACM, 322–335.

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. *PLDI* 41, 6 (2006), 387–400.

Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *CC '16*. ACM, 265–266.

Jiankai Sun, Bortik Bandyopadhyay, Armin Bashizade, Jiongqian Liang, P Sadayappan, and Srinivasan Parthasarathy. 2019. Atp: Directed graph embedding with asymmetric transitivity preservation. In *AAAI '19*, Vol. 33. 265–272.

Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *IJCNLP*. Beijing, China, 1556–1566.

Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. Line: Large-scale information network embedding. In *WWW '15*. 1067–1077.

Secil Ugurel, Robert Krovetz, and C Lee Giles. 2002. What's the code?: automatic classification of source code archives. In *KDD '02*. ACM, 632–638.

Xiao Wang, Peng Cui, Jing Wang, Jian Pei, Wenwu Zhu, and Shiqiang Yang. 2017. Community preserving network embedding. In *AAAI '17*.

Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. 2014. Knowledge graph embedding by translating on hyperplanes. In *AAAI '14*.

Mark Weiser. 1981. Program slicing. In *ICSE '81*. IEEE Press, 439–449.

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. 2015. Show, attend and tell: Neural image caption generation with visual attention. In *ICML '15*. 2048–2057.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *ICSE '19*. 783–794.

Ziwei Zhang, Peng Cui, Xiao Wang, Jian Pei, Xuanrong Yao, and Wenwu Zhu. 2018. Arbitrary-order proximity preserved network embedding. In *KDD '18*. ACM, 2778–2786.

Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *FSE '18*. ACM, 141–151.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *NeurIPS '19*. 10197–10207.