Check for updates

June 16, 2025
Seoul, Republic of Korea

Association for Computing Machinery

Advancing Computing as a Science & Profession

SOAP '25

Proceedings of the 14th ACM SIGPLAN International Workshop on

the State Of the Art in Program Analysis

Edited by:
Kihong Heo and Luca Negrini

Sponsored by:
ACM SIGPLAN

Co-located with:
PLDI '25

Association for Computing Machinery, Inc.
1601 Broadway, 10th Floor
New York, NY 10019-7434
USA

Copyright © 2025 by the Association for Computing Machinery, Inc (ACM). Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted.

To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept. ACM, Inc.
Fax +1-212-869-0481 or E-mail permissions@acm.org.

For other copying of articles that carry a code at the bottom of the first or last page, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, USA.

ACM ISBN: 979-8-4007-1922-6

Cover photo:
Title: “Traditional Pavilion in Seoul”
Photographer: Woosuk Lee
Cropped from private photo at Changdeokgung Palace, Seoul

Production: Conference Publishing Consulting
D-94034 Passau, Germany, info@conference-publishing.com

Welcome from the Chairs

The 14th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP'25) is co-located with the 46th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'25). In line with past workshops, SOAP'25 aims to bring together members of the program analysis community to share new developments and shape innovations in program analysis.

This edition of SOAP had 13 submissions, each reviewed by three reviewers, and seven were accepted (54% acceptance rate) in the end. Along with the accepted papers, SOAP'25 features three keynotes by leading members of the program analysis community: Xavier Rival (Ecole Normale Supérieure), Yulei Sui (University of New South Wales) and Charles Zhang (Hong Kong University of Science and Technology).

We would like to commend the efforts of the eleven members of the program committee, who donated their valuable time and effort to make the reviewing process possible. We also thank the PLDI chairs, the workshops chairs and the ACM staff for their continued support in making this workshop possible. We hope you enjoy SOAP'25 and look forward to enlightening discussions.

June 2025

Kihong Heo
Luca Negrini

Program Chairs

Kihong Heo (KAIST, Korea)
Luca Negrini (Ca' Foscari University of Venice, Italy)

Program Committee

Elizabeth Dinella (Bryn Mawr College, USA)
Karoliine Holter (University of Tartu, Estonia)
Minseok Jeon (Korea University, Korea)
HeuiChan Lim (Davidson College, USA)
Mukund Raghothaman (University of Southern California, USA)
Michael Schwarz (TU Munich, Germany)
Sunbeom So (GIST, Korea)
Tian Tan (Nanjing University, China)
Aditya V. Thakur (University of California at Davis, USA)
Ignacio Tiraboschi (Eclypsium, Argentina)
Milla Valnet (Sorbonne Université, France)

Steering Committee

Pietro Ferrara (Università Ca' Foscari, Italy)
Liana Hadarean (Amazon, USA)
Laure Gonnord (Grenoble-INP/LCIS, France)
Neville Grech (University of Malta, Malta)
Ben Hermann (Technische Universität Dortmund, Germany)
Padmanabhan Krishnan (Oracle Labs, Australia)
Thierry Lavoie (Synopsys, Canada)
Raphaël Monat (Inria and University of Lille, France)
Lisa Nguyen Quang Do (Google, Switzerland)
Christoph Reichenbach (Lund University, Sweden)
Cindy Rubio-González (University of California at Davis, USA)
Laura Titolo (NIA/NASA LaRC, USA)
Omer Tripp (Amazon, USA)
Caterina Urban (INRIA and École Normale Supérieure, France)

Contents

Frontmatter

Papers

Beyond Affine Loops: A Geometric Approach to Program Synthesis*

Erdenebayar Bayarmagnai
erdenebayar.bayarmagnai@kuleuven.be
KU Leuven
Belgium

Fatemeh Mohammadi
fatemeh.mohammadi@kuleuven.be
KU Leuven
Belgium

Rémi Prébet
remi.prebet@inria.fr
Inria
France

Abstract

Ensuring software correctness remains a fundamental challenge in formal program verification. One promising approach relies on finding polynomial invariants for loops. Polynomial invariants are properties of a program loop that hold before and after each iteration. Generating polynomial invariants is a crucial task for loops, but it is an undecidable problem in the general case. Recently, an alternative approach to this problem has emerged, focusing on synthesizing loops from invariants. However, existing methods only synthesize affine loops without guard conditions from polynomial invariants. In this paper, we address a more general problem, allowing loops to have polynomial update maps with a given structure, inequations in the guard condition, and polynomial invariants of arbitrary form.

In this paper, we use algebraic geometry tools to design and implement an algorithm that computes a finite set of polynomial equations whose solutions correspond to all loops satisfying the given polynomial invariants. In other words, we reduce the problem of synthesizing loops to finding solutions of polynomial systems within a specified subset of the complex numbers. The latter is handled in our software using an SMT solver.

CCS Concepts: • **Applied computing** → **Invariants; Logic and verification**; • **Computing methodologies** → **Symbolic and algebraic manipulation**.

*We are grateful to Teresa Krick for her very helpful comments. The third author is also affiliated with Inria, CNRS, ENS de Lyon, Université Claude Bernard Lyon 1, LIP (UMR 5668), 69342 Lyon Cedex 07, France. The research is partially supported by the KU Leuven grant iBOF/23/064, the FWO grants G0F5921N and G023721N and the ANR grants ANR-20-CE48-0014 NuSCAP and ANR-24-CE48-4035 CNACS

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOAP '25, Seoul, Republic of Korea
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1922-6/25/06
https://doi.org/10.1145/3735544.3735581

Keywords: Program synthesis, Polynomial invariants

ACM Reference Format:
Erdenebayar Bayarmagnai, Fatemeh Mohammadi, and Rémi Prébet. 2025. Beyond Affine Loops: A Geometric Approach to Program Synthesis. In *Proceedings of the 14th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '25), June 16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3735544.3735581

1 Introduction

Loop invariants are properties that hold before and after each iteration and are essential for automating program verification, which ensures correctness before execution. Established methods such as the Floyd-Hoare inductive assertion technique [11] and termination verification via ranking functions [26] rely on loop invariants. A polynomial invariant is a loop invariant expressed as a polynomial equation or inequality.

In this work, instead of generating polynomial invariants for a given loop, we address the reverse problem, that is synthesizing a loop that satisfies given polynomial invariants.

We consider polynomial loops $\mathcal{L}(\mathbf{a}, h, F)$ of the form

$$\begin{array}{l}\mathbf{x} := (x_1, \ldots, x_n) \leftarrow \mathbf{a} := (a_1, \ldots, a_n)\\ \textbf{while } h(\mathbf{x}) \neq 0 \textbf{ do}\\ \quad \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \xleftarrow{F} \begin{pmatrix} F_1 \\ F_2 \\ \vdots \\ F_n \end{pmatrix}\\ \textbf{end while}\end{array}$$

where x_i are the program variables with initial values a_i, $h \in \mathbb{C}[\mathbf{x}]$, and $F = (F_1, \ldots, F_n)$ is a sequence of polynomials in $\mathbb{C}[\mathbf{x}]$. The inequation $h(\mathbf{x}) \neq 0$, called the *guard*, is assumed to be a single inequality for simplicity (we can replace $h_1 \neq 0, \ldots, h_k \neq 0$ with their product $h_1 \cdot \ldots \cdot h_k \neq 0$ without loss of generality). Where there is no h we simply write $\mathcal{L}(\mathbf{a}, 1, F)$, which corresponds to an infinite loop.

Previous work [16, 19, 22] has focused on update maps that are restricted to be linear. In this paper, we go beyond this limitation by allowing update maps to be arbitrary polynomial functions. We present a method for computing a system of equations whose common solutions characterize all coefficient assignments for update maps of loops that satisfy a given set of polynomial invariants. To illustrate our main objective, we now present a motivating example.

Example 1. Consider the following polynomial loop:

$(x_1, x_2, x_3) \leftarrow (1, 1, -1)$
while true **do**
$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \overset{F}{\leftarrow} \begin{pmatrix} \lambda_1 x_1^3 + \lambda_2 x_2^2 \\ \lambda_3 x_1 + \lambda_4 x_2^2 \\ \lambda_5 x_1 \end{pmatrix}$
end while

By applying Algorithm 2, we construct polynomials $P_1, \ldots, P_4$, such that $(\lambda_1, \ldots, \lambda_5)$ is a common root of these polynomials if and only if the loop satisfies the polynomial invariants $g_1 = x_2^2 - x_1$ and $g_2 = x_3^3 + 2x_2^2 - x_1$. For instance, $(-3, 3, 1, -1, 0)$ is a common root of $\{P_1, \ldots, P_4\}$. Thus, if the update map is $F(x_1, x_2, x_3) = (-3x_1^3 + 3x_2^2, x_1 - x_2^2, 0)$, the loop satisfies the invariants g_1 and g_2. See Example 3 for further details.

Related works. The computation of polynomial invariants for loops has been a prominent area of research over the past two decades [3, 9, 17, 20, 23, 24, 27–29]. However, computing invariant ideals is undecidable for general loops [18]. As a result, efficient methods have been developed for restricted classes of loops, particularly those where the assertions are linear or can be transformed into linear assertions.

The converse problem of synthesizing loops from given invariants has received less attention, primarily focusing on linear and affine loops. These studies vary in the types of invariants considered: linear invariants in [34], a single quadratic polynomial in [16], and pure difference binomials in [22]. In contrast, [19] explores general polynomial invariants but lacks the completeness properties of earlier work and does not address guard conditions.

Higher degree loops have been studied in a more general framework in [12, 13], which also takes polynomial inequalities as input. However, the loops synthesized by this algorithm are restricted to those where the input invariants are inductive, meaning that if the invariant holds after one iteration, it holds for all subsequent iterations.

Our contributions. We consider the problem of generating polynomial loops with guards from arbitrary polynomial invariants. As a first step, we construct a polynomial system whose solutions correspond precisely to loops with a given structure that satisfy the specified invariants. We then discuss strategies for solving this system. In practice, in most cases a satisfying solution is found using an SMT solver.

Section 2 recalls the definition of invariant sets and key results from [2]. Section 3 introduces a method for identifying multiple polynomial invariants (Proposition 3.5) and proves that the set $\mathscr{C}(\mathbf{a}, h, \mathbf{f}; \boldsymbol{g})$ of coefficients of polynomial maps of loops satisfying the invariants $\boldsymbol{g}$ forms an algebraic variety. We also present Algorithm 2, which computes the polynomials defining $\mathscr{C}(\mathbf{a}, h, \mathbf{f}; \boldsymbol{g})$. Section 4 discusses strategies for solving polynomial systems and their limitations, while Section 5 presents our algorithm's implementation and experimental results.

2 Preliminaries

In the following, we present some terminology from algebraic geometry. For further details, we refer the reader to [7, 21, 32]. We denote the field of complex numbers by $\mathbb{C}$. Throughout the paper, $\mathbf{x}$ denotes indeterminates $x_1, \ldots, x_n$, and $\mathbb{C}[\mathbf{x}]$ the multivariate polynomial ring in these variables.

Ideals. A polynomial ideal I is a subset of $\mathbb{C}[\mathbf{x}]$ that is closed under addition, $0 \in I$ and for any $f \in \mathbb{C}[\mathbf{x}]$ and $g \in I$, $fg \in I$. Given a subset S of $\mathbb{C}[\mathbf{x}]$, the ideal generated by S is

$$\langle S \rangle = \{a_1 f_1 + \ldots + a_m f_m \mid a_i \in \mathbb{C}[\mathbf{x}], f_i \in S, m \in \mathbb{N}\}.$$

By Hilbert Basis Theorem [7, Theorem 4, Chap. 2], every ideal is finitely generated. For a subset $X \subset \mathbb{C}^n$, the defining ideal $I(X)$ consists of all polynomials vanishing on X. The radical of an ideal $I \subset \mathbb{C}[\mathbf{x}]$ is defined as

$$\sqrt{I} = \{f \in \mathbb{C}[\mathbf{x}] \,|\, f^m \in I \text{ for some } m \in \mathbb{N}\}.$$

Varieties. For $S \subset \mathbb{C}[x]$, the algebraic variety $\boldsymbol{V}(S)$ is the common zero set of all polynomials in S. Moreover, $\boldsymbol{V}(S) = \boldsymbol{V}(\langle S \rangle)$ and so every algebraic variety is the vanishing locus of finitely many polynomials.

Polynomial maps. A map $F : \mathbb{C}^n \longrightarrow \mathbb{C}^m$ is a polynomial map if there exist $f_1, \ldots, f_m \in \mathbb{C}[x_1, \ldots, x_m]$ such that

$$F(x) = (f_1(x), \ldots, f_m(x))$$

for all $x \in \mathbb{C}^n$. For simplicity, we will identify polynomial maps and their corresponding polynomials.

Here, we introduce the main object of this paper and build upon key results from [2] by recalling and extending them.

Definition 2.1. *Let $F : \mathbb{C}^n \longrightarrow \mathbb{C}^n$ be a map and X be a subset of $\mathbb{C}^n$. The invariant set of (F, X) is defined as:*

$$S_{(F,X)} = \{x \in X \mid \forall m \in \mathbb{N}, F^{(m)}(x) \in X\},$$

where $F^{(0)}(x) = x$ and $F^{(m)}(x) = F(F^{(m-1)}(x))$ for $m > 1$.

The following proposition enables the computation of invariant sets using tools from algebraic geometry, relying on the Hilbert Basis Theorem, which implies that descending chains of algebraic varieties eventually stabilize.

Proposition 2.2. *Let $X \subseteq \mathbb{C}^n$ be an algebraic variety and consider a polynomial map $F : \mathbb{C}^n \longrightarrow \mathbb{C}^n$. We define*

$$X_m = X \cap F^{-1}(X) \cap \ldots \cap F^{-m}(X)$$

for all $m \in \mathbb{N}$. Then, there exists $N \in \mathbb{N}$ such that $X_N = X_{N+1}$, and for any such index $X_N = S_{(F,X)}$.

Algorithm 1 computes invariant sets via an iterative outer approximation that converges in finitely many steps, based on Proposition 2.2. It relies on the following procedures:

- "Compose" takes as input two sequences of polynomials $\boldsymbol{g} = (g_1, \ldots, g_m)$ and $F = (F_1, \ldots, F_n)$ in $\mathbb{Q}[\mathbf{x}]$ and outputs the polynomials

$$g_1\big(F_1(\mathbf{x}), \ldots, F_n(\mathbf{x})\big), \ldots, g_m\big(F_1(\mathbf{x}), \ldots, F_n(\mathbf{x})\big).$$

- "InRadical" takes as input a sequence of polynomials $\widetilde{\boldsymbol{g}}$ and a finite set S in $\mathbb{Q}[\mathbf{x}]$ and outputs "True" if $\widetilde{\boldsymbol{g}} \subset \sqrt{\langle S \rangle}$; "False" otherwise.

These procedures are classic routines in symbolic computations, and can be performed using various efficient techniques such as Gröbner bases [7]. See [3] for more details.

Algorithm 1 InvariantSet

Input: $\boldsymbol{g}$ and $F = (F_1, \ldots, F_n)$ are sequences in $\mathbb{Q}[\mathbf{x}]$.
Output: Polynomials whose common zero-set is $S_{(F, V(\boldsymbol{g}))}$.

```
S ← {g};
g̃ ← Compose(g, F);
while InRadical(g̃, S) == False do
    S ← S ∪ {g̃};
    g̃ ← Compose(g̃, F);
end while
return S;
```

The proof of the termination and correctness of Algorithm 1 follows from Proposition 2.2 and [2, Theorem 2.4].

Example 2. Let us compute the invariant set of a polynomial map $F(x_1, x_2) = (2x_1 - 3x_2, x_1 + x_2)$ and an algebraic variety $X = \boldsymbol{V}(x_1^2 - x_2^2 + x_1x_2)$. On input F and $g = x_1^2 - x_2^2 + x_1x_2$, Algorithm 1 computes the invariant set of F and X through the following steps.

- At Step 1, S gets $\{g\}$ and at Step 2, $\widetilde{g}$ gets
$$\text{Compose}(g, F) = 5x_1^2 - 15x_2x_2 + 5x_2^2.$$
- At Step 3, a Gröbner basis of the ideal $\langle g, 1 - t\widetilde{g} \rangle$ is computed to verify that
$$\text{InRadical}(\widetilde{g}, S) = \texttt{False}.$$
- At Step 4, S gets $\{g, g \circ F\}$ and at Step 5, $\widetilde{g}$ is set to
$$\text{Compose}(\widetilde{g}, F) = -5x_1^2 - 35x_1x_2 + 95x_2^2$$
- This time, $\text{InRadical}(\widetilde{g}, F) = \texttt{True}$, so the while loop terminates.

Therefore, the invariant set $S_{(F, V(g))}$ is the vanishing locus of polynomials $\{g, g \circ F\}$.

3 From Loops to Polynomial Systems

In this section, we present a method for identifying all loops that satisfy given polynomial invariants by formulating them as solutions of a polynomial system via invariant sets. This reduces the problem to solving a polynomial system, which we explore in the next section.

Definition 3.1. *A polynomial g is an invariant of the loop $\mathcal{L}(\mathbf{a}, h, F)$ if, for any $m \in \mathbb{Z}_{\geq 0}$, either:*
$$g(F^{(m)}(\mathbf{a})) = 0,$$
or there exists $m \in \mathbb{Z}_{\geq 0}$ such that $g(F^{(m)}(\mathbf{a})) = h(F^{(m)}(\mathbf{a})) = 0$ and for every $l < m$:
$$g(F^{(l)}(\mathbf{a})) = 0 \text{ and } h(F^{(l)}(\mathbf{a})) \neq 0.$$

Definition 3.2. *Let $\mathcal{L}(\mathbf{a}, h, F)$ be a polynomial loop. The set of all polynomial invariants for $\mathcal{L}(\mathbf{a}, h, F)$ is called the invariant ideal of $\mathcal{L}$ and is denoted by $I_{\mathcal{L}(\mathbf{a},h,F)}$.*

The invariant ideal is an ideal of the ring $\mathbb{C}[x_1, \ldots, x_n]$ where $x_1, \ldots, x_n$ are program variables [27]. Let $f_1, \ldots, f_n$ be polynomials in $\mathbb{C}[x_1, \ldots, x_n]$. We denote by $span\{f_1, \ldots, f_n\}$ the vector space they generate.

Definition 3.3. *$(F_1, \ldots, F_n)$ and for $1 \leq i \leq n$ let $\mathbf{f}_i = (f_{i,1}, \ldots, f_{i,l_i})$ be sequences of polynomials in $\mathbb{C}[\mathbf{x}]$ such that for every i,*
$$F_i = \sum_{j=1}^{l_i} b_{i,j} f_{i,j}$$
for some $b_{i,j}$'s $\in \mathbb{C}$. Let $\mathbf{f} = (\mathbf{f}_1 \ldots, \mathbf{f}_n)$ and define the map
$$F_{\mathbf{f},\mathbf{b}} : \mathbb{C}^n \longrightarrow \mathbb{C}^n$$
with $F_{\mathbf{f},\mathbf{b}}(x) = (F_1(x), \ldots, F_n(x))$.

The object defined below is the primary focus of this paper. We prove that it is an algebraic variety and compute the polynomial equations that define it.

Definition 3.4. *Using the notations of Definition 3.3, let $h \in \mathbb{C}[\mathbf{x}]$, $\boldsymbol{g} = (g_1, \ldots, g_m)$ be a sequence of polynomials in $\mathbb{C}[\mathbf{x}]$, and $\mathbf{a} \in \mathbb{C}^n$. Then, the polynomial loop, structured by $\mathbf{f}$, with invariants including $\boldsymbol{g}$, is defined by the coefficient set:*
$$\mathscr{C}(\mathbf{a}, h, \mathbf{f}; \boldsymbol{g}) = \{\mathbf{b} \in \mathbb{C}^{l_1 + \ldots + l_n} \mid \boldsymbol{g} \subset I_{\mathcal{L}(\mathbf{a},h,F_{\mathbf{f},\mathbf{b}})}\}.$$

In simple words, $\mathscr{C}(\mathbf{a}, h, \mathbf{f}; \boldsymbol{g})$ is the set of all vectors $\mathbf{b} \in \mathbb{C}^{l_1 + \cdots + l_n}$ such that all polynomials in $\boldsymbol{g}$ are polynomial invariants of the following loop:

```
x ← a
while h(x) ≠ 0 do
    x ← F_{f,b}(x)
end while
```

We now give a necessary and sufficient condition for checking whether given polynomials are loop invariants. This extends [3, Proposition 2.7], where the following statement was proven for a single polynomial invariant.

Proposition 3.5. *Let $h, g_1, \ldots, g_m$ be polynomials in $\mathbb{C}[\mathbf{x}]$. Let z be a new indeterminate and let*
$$X = \boldsymbol{V}(zg_1, \ldots, zg_m) \subset \mathbb{C}^{n+1}.$$
Let $F : \mathbb{C}^n \longrightarrow \mathbb{C}^n$ be a polynomial map and define
$$G_h(\mathbf{x}, z) = (F(\mathbf{x}), zh(\mathbf{x})).$$
Then, for $\mathbf{a} \in \mathbb{C}^n$, $\boldsymbol{g} \subset I_{\mathcal{L}(\mathbf{a},h,F)}$ if and only if $(\mathbf{a}, 1) \in S_{(G_h,X)}$.

Proof. Let $X_i = \boldsymbol{V}(zg_i) \subset \mathbb{C}^{n+1}$ be an algebraic variety for $i \in \{1, \ldots, m\}$. For any $i \in \{1, \ldots, m\}$, by [3, Proposition 2.7], $g_i \in I_{\mathcal{L}(\mathbf{a},h,F)}$ if and only if $(\mathbf{a}, 1) \in S_{(G_h,X_i)}$. Therefore, $g_1, \ldots, g_m \in I_{\mathcal{L}(\mathbf{a},h,F)}$ if and only if
$$(\mathbf{a}, 1) \in S_{(G_h,X_1)} \cap \ldots \cap S_{(G_h,X_m)}.$$

It follows directly from Definition 2.1 that

$$S_{(G_h,X)} = S_{(G_h,X_1)} \cap \ldots \cap S_{(G_h,X_m)}.$$

Thus, $g_1, \ldots, g_m \in I_{\mathcal{L}(\mathbf{a},h,F)}$ if and only if $(\mathbf{a}, 1) \in S_{(G_h,X)}$. □

In the following proposition, we present a necessary and sufficient condition for loops with a given structure to satisfy specified polynomial invariants.

Proposition 3.6. *For $1 \leq i \leq n$, let $\mathbf{f}_i = (f_{i,1}, \ldots, f_{i,l_i})$ be sequences of polynomials in $\mathbb{C}[\mathbf{x}]$ and define $\mathbf{f} = (\mathbf{f}_1, \ldots, \mathbf{f}_n)$. Let $z, y_{1,1}, \ldots, y_{1,l_1}, y_{n,1}, \ldots, y_{n,l_n}$ be new indeterminates, let $h \in \mathbb{C}[\mathbf{x}]$ and define the following polynomial map in $\mathbb{C}[\mathbf{x}, \mathbf{y}, z]$:*

$$G_{\mathbf{y},h}(\mathbf{x}, \mathbf{y}, z) = \left(\sum_{i=1}^{l_1} y_{1,i} f_{1,i}(\mathbf{x}), \ldots, \sum_{i=1}^{l_n} y_{n,i} f_{n,i}(\mathbf{x}), \mathbf{y}, zh(\mathbf{x}) \right).$$

Let $\boldsymbol{g} = (g_1, \ldots, g_m)$ be a sequence of polynomials in $\mathbb{C}[\mathbf{x}]$, seen as elements of $\mathbb{C}[\mathbf{x}, \mathbf{y}, z]$, let

$$X = \boldsymbol{V}(zg_1, \ldots, zg_m) \subset \mathbb{C}^{n+l_1+\ldots+l_n+1}.$$

Then, for any $\mathbf{a} \in \mathbb{C}^n$,

$$\mathscr{C}(\mathbf{a}, h, \mathbf{f}; \boldsymbol{g}) = \{\mathbf{b} \in \mathbb{C}^{l_1+\ldots+l_n} \mid (\mathbf{a}, \mathbf{b}, 1) \in S_{(G_{\mathbf{y},h},X)}\}.$$

Proof. Write $G_{\mathbf{y},h}(\mathbf{x}, \mathbf{y}, z) = (G_{\mathbf{y}}(\mathbf{x}, \mathbf{y}), zh(\mathbf{x}))$ where

$$G_{\mathbf{y}}(\mathbf{x}, \mathbf{y}) = \left(\sum_{i=1}^{l_1} y_{1,i} f_{1,i}(\mathbf{x}), \ldots, \sum_{i=1}^{l_n} y_{n,i} f_{n,i}(\mathbf{x}), \mathbf{y} \right).$$

By Proposition 3.5, $g_1, \ldots, g_m$ are polynomial invariants of $\mathcal{L}((\mathbf{a}, \mathbf{b}), h, G_{\mathbf{y}})$ if and only if $(\mathbf{a}, \mathbf{b}, 1) \in S_{(G_{\mathbf{y},h},X)}$. Since the value of $\mathbf{y}$ remains $\mathbf{b}$ after any iteration, we know that

$$F_y^N(\mathbf{x}, \mathbf{b}) = (F^N(\mathbf{x}), \mathbf{b})$$

for any $N \in \mathbb{N}$. Therefore, $\boldsymbol{g} \subset I_{\mathcal{L}(\mathbf{a},h,F_{\mathbf{f},\mathbf{b}})}$ if and only if $\boldsymbol{g} \subset I_{\mathcal{L}((\mathbf{a},\mathbf{b}),h,G_{\mathbf{y}})}$, that is $(\mathbf{a}, \mathbf{b}, 1) \in S_{(G_{\mathbf{y},h},X)}$ by above. □

Since the invariant set is an algebraic variety,by Proposition 3.6, $\mathscr{C}(\mathbf{a}, h, \mathbf{f}; \boldsymbol{g})$ is also an algebraic variety. Specifically, its defining equations are obtained by substituting $\mathbf{x} = \mathbf{a}$ and $z = 1$ into the system defining the invariant set.

We now present our main algorithm for computing the polynomials that define $\mathscr{C}(\mathbf{a}, h, \mathbf{f}; \boldsymbol{g})$.

Algorithm 2 GenerateLoops

Input: $h, g_1, \ldots, g_m, f_{1,1}, \ldots, f_{1,l_1}, \ldots, f_{n,1}, \ldots, f_{n,l_n}$ in $\mathbb{Q}[\mathbf{x}]$ and $\mathbf{a} \in \mathbb{Q}^n$.

Output: A set of polynomials $\{P_1, \ldots, P_s\}$ such that $\mathscr{C}(\mathbf{a}, h, \mathbf{f}; g)$ is $\boldsymbol{V}(P_1, \ldots, P_s)$

1: $G_{\mathbf{y},h} \leftarrow \left(\sum_{i=1}^{l_1} y_{1,i} f_{1,i}(\mathbf{x}), \ldots, \sum_{i=1}^{l_n} y_{n,i} f_{n,i}(\mathbf{x}), \mathbf{y}, zh(\mathbf{x}) \right)$;
2: $\{Q_1, \ldots, Q_s\} \leftarrow$ InvariantSet$((zg_1, \ldots, zg_m), G_{\mathbf{y},h})$;
3: $\{P_1(\mathbf{y}), \ldots, P_s(\mathbf{y})\} \leftarrow \{Q_1(\mathbf{a}, \mathbf{y}, 1), \ldots, Q_s(\mathbf{a}, \mathbf{y}, 1)\}$
4: **return** $\{P_1, \ldots, P_s\}$;

We now prove the correctness of Algorithm 2.

Theorem 3.7. *Let $h \in \mathbb{C}[\mathbf{x}], \boldsymbol{g} = (g_1, \ldots, g_m)$ and $\mathbf{f}_1 = (f_{1,1}, \ldots, f_{1,l_1}), \ldots, \mathbf{f}_n = (f_{n,1}, \ldots, f_{n,l_n})$ be sequences of polynomials in $\mathbb{Q}[\mathbf{x}]$. Let $\mathbf{a} \in \mathbb{Q}^n$ and define $\mathbf{f} = (\mathbf{f}_1, \ldots, \mathbf{f}_n)$. On input $\mathbf{a}, h, \mathbf{f}, \boldsymbol{g}$, Algorithm 2 outputs a set of polynomials such that the vanishing locus of the polynomials is $\mathscr{C}(\mathbf{a}, h, \mathbf{f}; \boldsymbol{g})$.*

Proof. Let $X = \boldsymbol{V}(zg_1, \ldots, zg_m) \subset \mathbb{C}^{n+l_1+\ldots+l_n+1}$. At Step 2, by [2, Theorem 2.4], InvariantSet outputs

$$\{Q_1(\mathbf{x}, \mathbf{y}, z), \ldots, Q_s(\mathbf{x}, \mathbf{y}, z)\}$$

such that $S_{(G_{\mathbf{y},h},X)}$ is the common solution of $\{Q_1, \ldots, Q_s\}$. Therefore, by Proposition 3.6, $g_1, \ldots, g_m$ are polynomial invariants of $\mathcal{L}(\mathbf{a}, h, F_{\mathbf{f},\mathbf{b}})$ if and only if

$$Q_1(\mathbf{a}, \mathbf{b}, 1) = \ldots = Q_s(\mathbf{a}, \mathbf{b}, 1) = 0.$$

Consequently, $g_1, \ldots, g_m \in I_{\mathcal{L}(\mathbf{a},h,F_{\mathbf{f},\mathbf{b}})}$ if and only if

$$P_1(\mathbf{b}) = \ldots = P_s(\mathbf{b}) = 0$$

which proves the correctness of Algorithm 2. □

Example 3. Consider the loop $\mathcal{L}((1, 1, -1), 1, F)$ together with polynomial invariants g_1 and g_2 from Example 1. Let us compute all loops with the polynomial map F that satisfy the polynomial invariants g_1 and g_2 That is, we determine all loops of the given form with the precondition $(x_1 = 1, x_2 = 1, x_3 = -1)$ and the postcondition $(x_2^2 - x_1 = 0, x_3^3 + 2x_2^2 - x_1 = 0)$. Let $F = (F_1, F_2, F_3)$, $\boldsymbol{g} = \{g_1, g_2\}$ and define

$$\mathbf{f} = \{\{x_1^3, x_2^2\}, \{x_1, x_2^2\}, \{x_1\}\}.$$

From the structure of a loop, we know that

$$F_1 \in span\{x_1^3, x_2^2\}, F_2 \in span\{x_1, x_2^2\}, F_3 \in span\{x_1\}.$$

Thus, the input to Algorithm 2 is $(\{1\}, \boldsymbol{g}, \mathbf{f}, \{1, 1, -1\})$. Let $X = \boldsymbol{V}(zg_1, zg_2) \subset \mathbb{C}^9$ and define a polynomial map

$$G_{\mathbf{y},h}(\mathbf{x}, \mathbf{y}, z) = (y_1 x_1^3 + y_2 x_2^2, y_3 x_1 + y_4 x_2^2, y_5 x_1, \mathbf{y}, z)$$

At Step 2, on input $G_{\mathbf{y},h}$ and X, algorithm "InvariantSet" computes polynomials whose vanishing locus is the invariant set $S_{(G_{\mathbf{y},h},X)}$ and outputs $\{Q_1(\mathbf{x}, \mathbf{y}, z), \ldots, Q_6(\mathbf{x}, \mathbf{y}, z)\}$. After substituting the initial values of the loop into the polynomials, we obtain only four non-zero polynomials $P_1(\mathbf{y})$, $P_2(\mathbf{y})$, $P_3(\mathbf{y})$ and $P_4(\mathbf{y})$ whose common vanishing locus is $\mathscr{C}((1, 1, -1), 1, \mathbf{f}; \boldsymbol{g})$:

$$P_1 = (y_3 + y_4)^2 - y_1 - y_2; \quad P_2 = y_5^3 + 2(y_3 + y_4)^2 - y_1 - y_2;$$

$$P_3 = 2y_3^4 y_4^2 + 8y_3^3 y_4^3 + 12y_3^2 y_4^4 + 8y_3 y_4^5 + 2y_4^6 + y_1^3 y_5^3 + 3y_1^2 y_2 y_5^3 + 3y_1 y_2^2 y_5^3 + y_2^3 y_5^3 + 4y_1 y_3^3 y_4 + 4y_2 y_3^3 y_4 + 8y_1 y_3^2 y_4^2 + 8y_2 y_3^2 y_4^2 + 4y_1 y_3 y_4^3 + 4y_2 y_3 y_4^3 - y_1^4 - 3y_1^3 y_2 - 3y_1^2 y_2^2 - y_1 y_2^3 + 2y_1^2 y_3^2 + 4y_1 y_2 y_3^2 + 2y_2^2 y_3^2 - y_2 y_3^2 - 2y_2 y_3 y_4 - y_2 y_4^2;$$

$$P_4 = y_3^4 y_4^2 + 4y_3^3 y_4^3 + 6y_3^2 y_4^4 + 4y_3 y_4^5 + y_4^6 + 2y_1 y_3^3 y_4 + 2y_2 y_3^3 y_4 + 4y_1 y_3^2 y_4^2 + 4y_2 y_3^2 y_4^2 + 2y_1 y_3 y_4^3 + 2y_2 y_3 y_4^3 - y_1^4 - 3y_1^3 y_2 - 3y_1^2 y_2^2 - y_1 y_2^3 + y_1^2 y_3^2 + 2y_1 y_2 y_3^2 + y_2^2 y_3^2 - y_2 y_3^2 - 2y_2 y_3 y_4 - y_2 y_4^2.$$

4 Polynomial System Solving

In the previous section, we have seen how Algorithm 2 can compute a system of multivariate polynomials whose solutions correspond exactly to the loops with the given structures and invariants. This approach reduces the problem of loop synthesis to solving polynomial systems.

However, since we aim to find loops with finite, exact representations on computers, we focus on *rational solutions*. Unlike solutions in $\mathbb{C}$ (by Hilbert's Nullstellensatz [15], see e.g., [7, Chap. 4, §1]) or $\mathbb{R}$ (via Tarski-Seidenberg's theorem [31, 35], see e.g., [1]), determining whether a multivariate polynomial equation has a rational solution is a major open problem in number theory [33]. For integer solutions, this is Hilbert's Tenth Problem, which is undecidable [33]. We outline three main strategies to address this, though none is fully satisfactory or complete.

4.1 Exploit Structure

Although no general algorithm is known to compute (or even decide the existence of) rational solutions to multivariate polynomials using exact methods, many real-world cases can still be successfully addressed with existing techniques. To illustrate this, consider the polynomial system obtained at the end of Example 3. While this example may appear overly simplistic or too specific, it turns out that all benchmarks presented in Section 5 can be tackled in a similar manner.

Example 4. The variety $V(P_1, \ldots, P_4)$ can be decomposed into finitely many irreducible components, which in turn decompose the system into potentially simpler subsystems. This can be done using classical methods from computer algebra [7, Chap. 4, §6], and here we apply the "`minimalPrimes`" command from Macaulay2 [14]. We obtain the following five components:

$$\begin{aligned}
\mathbf{V_1} &= V(y_5, y_3 + y_4, y_1 + y_2)\\
\mathbf{V_2} &= V(y_5 + 1, y_3 + y_4 - 1, y_1 + y_2 - 1)\\
\mathbf{V_3} &= V(y_5 + 1, y_3 + y_4 + 1, y_1 + y_2 - 1)\\
\mathbf{V_4} &= V(y_3 + y_4 - 1, y_1 + y_2 - 1, y_5^2 - y_5 + 1)\\
\mathbf{V_5} &= V(y_3 + y_4 + 1, y_1 + y_2 - 1, y_5^2 - y_5 + 1)
\end{aligned}$$

Thus, $\mathcal{L}((1, 1, -1), 1, F)$ satisfies the polynomial invariants $\{g_1, g_2\}$ if and only if $(\lambda_1, \ldots, \lambda_5)$ lies in one of the above irreducible components. The well-structured polynomials defining each component allow us to fully solve the problem.

For the first three components, which involve only linear polynomials, the problem reduces to a standard linear algebra routine, benefiting from optimized methods (see e.g., [6]). More specifically, if μ_1 and μ_2 are parameters in $\mathbb{Q}$, we obtain the following loop map:

$$\begin{aligned}
\mathbf{V_1}\colon\quad & F_1(x_1, x_2, x_3) = \big(\mu_1(x_1^3 - x_2^2), \mu_2(x_1 - x_2^2), 0\big)\\
\mathbf{V_2}\colon\quad & F_2(x_1, x_2, x_3) = \big(\mu_1 x_1^3 + (1-\mu_1)x_2^2, \mu_2 x_1 + (1-\mu_2)x_2^2, -1\big)\\
\mathbf{V_3}\colon\quad & F_3(x_1, x_2, x_3) = \big(\mu_1 x_1^3 - (1+\mu_1)x_2^2, \mu_2 x_1 + (1-\mu_2)x_2^2, -1\big)
\end{aligned}$$

For the last two components, there is no rational solution, because the last equation $y_5^2 - y_5 + 1$ has no rational roots.

Another specific case occurs when the polynomial system computed by Algorithm 2 (or a subsystem of it) has only finitely many solutions. Geometrically, this means the associated variety (or an irreducible component) consists of finitely many points, i.e., has *dimension zero*. In this case, such systems can be reduced (see e.g., [10, §3] and [30]) to polynomial systems with rational coefficients of the form:

$$Q_1(x_1) = 0 \quad \text{and} \quad x_i = Q_i(x_1) \quad \text{for all } i \geq 2.$$

This reduces to finding all rational solutions of a univariate polynomial, which is either done using arithmetic-based modular algorithms [25] or efficient factoring algorithms to identify its linear factors in $\mathbb{Q}[x_1]$ [36, Theorem 15.21].

Thus, when the system has finitely many solutions, we can always find all rational ones. Efficient software, such as AlgebraicSolving.jl[1], based on the msolve library [4], is designed for such tasks. Many of the benchmarks in the next section involve polynomial systems of this type, which we expect when the degree and support of F are close.

4.2 Numerical Methods

Numerical methods for solving polynomial systems, such as HomotopyContinuation.jl [5], provide powerful tools for approximating solutions. These methods use numerical algebraic geometry, particularly homotopy continuation. Unlike symbolic approaches that rely on Gröbner bases or resultants, numerical methods can efficiently handle large and complex systems. Using these methods, we can find numerical solutions of the polynomial system generated by Algorithm 2 and check if close integers or rational numbers are valid solutions of the polynomial system.

Example 5. Consider the polynomial system $\{P_1, \ldots, P_4\}$ from Example 3. This system either has no solutions or infinitely many, as there are 5 variables and only 4 equations. A classic approach to handle this is to introduce a random linear form with integer coefficients. Using HomotopyContinuation.jl, we find 15 numerical real solutions to the augmented system, 12 of which correspond to valid integer solutions.

Despite their efficiency, numerical methods have several drawbacks. First, they provide approximate solutions, which may lack exact algebraic structure and require further validation. Second, these methods can struggle with singular solutions. Additionally, homotopy continuation techniques may occasionally miss solutions.

4.3 Satisfiability Modulo Theories (SMT) Solvers

SMT solvers determine the satisfiability of logical formulas combining Boolean logic with mathematical theories, such as integer or real arithmetic. By verifying the satisfiability of

$$\exists x_1 \ldots \exists x_n (f_1(x_1, \ldots, x_n) = 0) \wedge \ldots \wedge (f_m(x_1, \ldots, x_n) = 0),$$

one can determine the existence of integer or rational solutions to a system of polynomial equations.

[1] https://algebraic-solving.github.io

SMT solvers leverage automated reasoning, efficient decision procedures, and integration with other logical theories. However, they can struggle with high-degree polynomials and may fail to find general integer solutions due to undecidability. Nevertheless, as discussed in Section 5, the SMT solver Z3 we used successfully finds integer solutions for most systems of polynomials generated by Algorithm 2.

5 Implementation and Experiments

In this section, we present the implementation of Algorithm 2 and report experiments on benchmarks from related works [16, 19, 22], which are limited to generating linear loops. The implementation in Macaulay2 [14] is available at

https://github.com/Erdenebayar2/Synthesizing_Loops.git

After running Algorithm 2, we use the SMT solver Z3 [8] to find a common nonzero integer solution for the polynomials.

5.1 Implementation Details

The experiments below were performed on a laptop equipped with a 4.8 GHz Intel i7 processor, 16 GB of RAM and 25 MB L3 cache. This prototype implementation is primarily based on the one in [2] to which we refer for further details.

5.2 Experimental Results

Let $\mathbf{f}_1 = (f_{1,1}, \ldots, f_{1,l_1}), \ldots, \mathbf{f}_n = (f_{n,1}, \ldots, f_{n,l_n})$ and $\boldsymbol{g} = (g_1, \ldots, g_m)$ be sequences of polynomials in $\mathbb{C}[\mathbf{x}]$, and define $\mathbf{f} = (\mathbf{f}_1, \ldots, \mathbf{f}_n)$. Let $h \in \mathbb{C}[\mathbf{x}]$. In Tables 1 and 2, we present the outputs and timings for Algorithm 2, which computes polynomials whose vanishing locus is $\mathscr{C}(\mathbf{a}, h, \mathbf{f}; g)$, along with timings for Z3 [8] for finding a common non-zero integer solution to the polynomials generated by Algorithm 2. The first row shows the benchmarks, while the first column describes the structures of the polynomial maps of loops. The benchmarks sources are available at

https://github.com/Erdenebayar2/Synthesizing_Loops/software/loops

Table 1 presents the execution timings for Algorithm 2 and Z3 [8], both using polynomial equations generated by Algorithm 2. Timings are in seconds, with a timeout of 300 seconds. "F" indicates Z3 failed to find an integer solution, while "NI" denotes no input was provided to Z3 when the algorithm reached the time limit.

In the following tables, n denotes the number of program variables, m is the number of polynomial invariants $\boldsymbol{g}$ and d is the maximal degree of polynomial invariants $\boldsymbol{g}$. Moreover, D denotes the maximal degrees of polynomials in $\mathbf{f}_1, \ldots, \mathbf{f}_n$, and $l = l_1 + \cdots + l_n$.

In most cases, when Algorithm 2 terminates, Z3 quickly finds a common non-zero integer solution for the generated polynomials within 0.3 seconds. Thus, finding a non-zero integer solution is not a bottleneck in our approach. The primary computational bottleneck lies in generating the polynomial systems for loop generation. Additionally, we observe that as more polynomial invariants are provided, Algorithm 2 terminates faster.

Table 1. Timings for Algorithm 2 in seconds;

Polynomial map				D=1, $l=3$		D=1, $l=4$		D=1, $l=5$		D=2, $l=2$		D=2, $l=3$	
Benchmark	n	m	d	Alg. 2	Z3	Alg. 2	Z3	Alg. 2	Z3	Alg. 2	Z3	Alg. 2	Z3
Ex2	2	1	4	0.02	F	31.1	F	TL	NI	0.01	0.06	TL	NI
Ex3	3	2	3	0.01	0.06	0.04	0.06	4.4	0.2	0.01	0.06	0.018	0.07
Ex3Ineq	3	2	3	0.009	0.06	11.4	0.06	TL	NI	0.008	0.05	0.03	0.05
Ex4	2	1	2	0.03	0.17	0.16	TL	TL	NI	0.01	0.07	0.13	0.07
sum1	3	2	2	0.02	0.06	0.13	0.06	1.1	0.06	0.01	0.05	0.02	0.06
square	2	1	2	0.01	0.06	0.5	TL	TL	NI	0.01	0.06	0.015	0.26
square_conj	3	2	2	0.019	0.06	0.14	0.09	0.39	0.06	0.007	0.05	0.015	0.06
fmi1	2	1	2	1.19	0.06	161.74	0.06	TL	NI	237.1	0.06	TL	NI
fmi2	3	2	2	0.01	0.06	0.015	0.06	0.017	0.07	0.009	0.07	0.01	0.07
fmi3	3	2	2	0.01	0.06	0.03	0.05	0.39	0.09	0.01	0.06	0.2	0.11
intcbrt	3	2	2	0.25	0.07	16.6	0.08	TL	NI	0.01	0.06	0.65	0.17
cube_square	3	1	3	0.01	0.07	0.018	TL	TL	NI	0.15	0.06	0.96	106

Table 2 presents the output of Algorithm 2, which is a polynomial system whose solution set exactly corresponds to $\mathscr{C}(\mathbf{a}, h, \mathbf{f}; g)$. For each choice of D and l, we report the number s of non-zero polynomials in the output system and indicate whether or not there are finitely many solutions.

Table 2. Data on outputs of Algorithm 2

Polynomial map				D=1, $l=3$		D=1, $l=4$		D=1, $l=5$		D=2, $l=2$		D=2, $l=3$	
Benchmark	n	m	d	s	#sols	s	#sols	s	#sols	s	#sols	s	#sols
Ex2	2	1	4	3	∞	4	∞	TL	TL	2	∞	TL	TL
Ex3	3	2	3	2	∞	4	∞	6	∞	4	$<\infty$	4	$<\infty$
Ex3Ineq	3	2	3	2	∞	4	∞	TL	TL	4	$<\infty$	4	$<\infty$
Ex4	2	1	2	3	∞	3	∞	TL	TL	3	$<\infty$	3	∞
sum1	3	2	2	4	∞	6	∞	6	∞	2	$<\infty$	4	$<\infty$
square	2	1	2	2	∞	4	∞	TL	TL	2	$<\infty$	3	∞
square_conj	3	2	2	4	$<\infty$	6	∞	6	∞	4	$<\infty$	4	$<\infty$
fmi1	2	1	2	0	∞	0	∞	TL	TL	0	∞	TL	TL
fmi2	3	2	2	2	∞	4	∞	4	∞	2	$<\infty$	4	$<\infty$
fmi3	3	2	2	4	$<\infty$	4	∞	6	∞	2	$<\infty$	4	$<\infty$
intcbrt	3	2	2	4	$<\infty$	4	∞	TL	TL	2	$<\infty$	4	$<\infty$
cube_square	3	1	3	2	∞	3	∞	TL	TL	3	$<\infty$	5	$<\infty$

We observe that Macaulay2 [14] computed the irreducible decompositions of varieties defined by the polynomials generated by Algorithm 2 in most cases within a few seconds. In many instances, the irreducible components of the varieties are defined by linear equations. In Table 2, even when the dimension of the varieties is 0, the varieties are not empty in all cases, indicating that they consist of finitely many points.

6 Conclusion

We presented a method for generating polynomial loops with specified structures that satisfy given polynomial invariants, reducing the problem to solving a system of polynomial equations. Along with an algorithm for constructing this system, we discussed various strategies for solving it. While previous work has primarily focused on linear maps, our approach extends beyond this limitation, marking a significant advancement in the synthesis of non-linear programs.

This work opens up several promising avenues for future research. A natural next step would be to explore the synthesis of more complex loop structures, such as branching and nested loops. Additionally, extending the approach to handle polynomial invariants in the form of inequalities, rather than just equations, could greatly enhance the flexibility and applicability of the method.

References

[1] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. 2006. *Algorithms in Real Algebraic Geometry* (2nd revised and extended 2016 ed.). Springer International Publishing. doi:10.1007/3-540-33099-2

[2] Erdenebayar Bayarmagnai, Fatemeh Mohammadi, and Rémi Prébet. 2024. Algebraic Tools for Computing Polynomial Loop Invariants. In *Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation* (Raleigh, NC, USA) *(ISSAC '24)*. Association for Computing Machinery, New York, NY, USA, 371–381. doi:10.1145/3666000.3669710

[3] Erdenebayar Bayarmagnai, Fatemeh Mohammadi, and Rémi Prébet. 2024. Algebraic Tools for Computing Polynomial Loop Invariants (Extended Version). *arXiv preprint arXiv:2412.14043* (2024).

[4] Jérémy Berthomieu, Christian Eder, and Mohab Safey El Din. 2021. msolve: A Library for Solving Polynomial Systems. In *2021 International Symposium on Symbolic and Algebraic Computation*. ACM, Saint Petersburg, Russia, 51–58. doi:10.1145/3452143.3465545

[5] Paul Breiding and Sascha Timme. 2018. HomotopyContinuation.jl: A Package for Homotopy Continuation in Julia. In *Mathematical Software – ICMS 2018*. Springer International Publishing, Cham, 458–465. doi:10.1007/978-3-319-96418-8_54

[6] William Cook and Daniel E. Steffy. 2011. Solving Very Sparse Rational Systems of Equations. *ACM Trans. Math. Softw.* 37, 4, Article 39 (Feb. 2011), 21 pages. doi:10.1145/1916461.1916463

[7] David Cox, John Little, and Donal O'Shea. 2013. *Ideals, varieties, and algorithms: an introduction to computational algebraic geometry and commutative algebra*. Springer Science & Business Media. doi:10.1007/978-3-319-16721-3

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. doi:10.1007/978-3-540-78800-3_24

[9] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. 2017. Synthesizing invariants by solving solvable loops. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 327–343. doi:10.1007/978-3-319-68167-2_22

[10] Clémence Durvye and Grégoire Lecerf. 2008. A concise proof of the Kronecker polynomial system solver from scratch. *Expositiones Mathematicae* 26, 2 (2008), 101–139. doi:10.1016/j.exmath.2007.07.001

[11] Robert Floyd. 1993. Assigning meanings to programs. In *Program Verification: Fundamental Issues in Computer Science*. Springer, 65–81. doi:10.1007/978-94-011-1793-7_4

[12] Amir Goharshady, S. Hitarth, Fatemeh Mohammadi, and Harshit Motwani. 2023. Algebro-geometric Algorithms for Template-based Synthesis of Polynomial Programs (Full Version including Appendices). https://hal.science/hal-04012686. (2023).

[13] Amir Kafshdar Goharshady, S Hitarth, Fatemeh Mohammadi, and Harshit Jitendra Motwani. 2023. Algebro-geometric Algorithms for Template-based Synthesis of Polynomial Programs. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 727–756. doi:10.1145/3586052

[14] Daniel R Grayson and Michael E Stillman. 2002. Macaulay2, a software system for research in algebraic geometry.

[15] David Hilbert. 1893. Ueber die vollen Invariantensysteme. *Math. Ann.* 42, 3 (1893), 313–373. doi:10.1007/BF01444162

[16] S. Hitarth, George Kenison, Laura Kovács, and Anton Varonka. 2024. Linear Loop Synthesis for Quadratic Invariants. In *41st International Symposium on Theoretical Aspects of Computer Science (STACS 2024)*, Vol. 289. 41:1–41:18. doi:10.4230/LIPIcs.STACS.2024.41

[17] Ehud Hrushovski, Joël Ouaknine, Amaury Pouly, and James Worrell. 2018. Polynomial invariants for affine programs. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 530–539. doi:10.1145/3209108.3209142

[18] Ehud Hrushovski, Joël Ouaknine, Amaury Pouly, and James Worrell. 2023. On strongest algebraic program invariants. *J. ACM* 70, 5 (2023), 1–22. doi:10.1145/3614319

[19] Andreas Humenberger, Daneshvar Amrollahi, Nikolaj Bjørner, and Laura Kovács. 2022. Algebra-Based Reasoning for Loop Synthesis. *Form. Asp. Comput.* 34, 1 (2022), 31 pages. doi:10.1145/3527458

[20] Michael Karr. 1976. Affine relationships among variables of a program. *Acta Informatica* 6 (1976), 133–151. doi:10.1007/BF00268497

[21] G. Kempf. 1993. *Algebraic Varieties*. Cambridge University Press. doi:10.1017/CBO9781107359956

[22] George Kenison, Laura Kovács, and Anton Varonka. 2023. From Polynomial Invariants to Linear Loops. In *Proceedings of the 2023 International Symposium on Symbolic and Algebraic Computation* (Tromsø, Norway) *(ISSAC '23)*. Association for Computing Machinery, New York, NY, USA, 398–406. doi:10.1145/3597066.3597109

[23] Laura Kovács. 2008. Reasoning algebraically about p-solvable loops. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 249–264. doi:10.1007/978-3-540-78800-3_18

[24] Laura Kovács. 2023. Algebra-Based Loop Analysis. In *Proceedings of the 2023 International Symposium on Symbolic and Algebraic Computation*. 41–42. doi:10.1145/3597066.3597150

[25] Rüdiger Loos. 1983. Computing Rational Zeros of Integral Polynomials by p-Adic Expansion. *SIAM J. Comput.* 12, 2 (1983), 286–293. doi:10.1137/0212017

[26] Zohar Manna and Amir Pnueli. 2012. *Temporal verification of reactive systems: safety*. Springer Science & Business Media. doi:10.1007/978-1-4612-4222-2

[27] Enric Rodríguez-Carbonell and Deepak Kapur. 2004. Automatic generation of polynomial loop invariants: Algebraic foundations. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*. 266–273. doi:10.1145/1005285.1005324

[28] Enric Rodríguez-Carbonell and Deepak Kapur. 2007. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming* 64, 1 (2007), 54–75. doi:10.1016/j.scico.2006.03.003

[29] Enric Rodríguez-Carbonell and Deepak Kapur. 2007. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation* 42, 4 (2007), 443–476. doi:10.1016/j.jsc.2007.01.002

[30] Fabrice Rouillier. 1999. Solving Zero-Dimensional Systems Through the Rational Univariate Representation. *Applicable Algebra in Engineering, Communication and Computing* 9, 5 (1999), 433–461. doi:10.1007/s002000050114

[31] A. Seidenberg. 1954. A New Decision Method for Elementary Algebra. *Annals of Mathematics* 60, 2 (1954), 365–374. https://doi.org/10.2307/1969640

[32] Igor Rostislavovich Shafarevich and Miles Reid. 1994. *Basic algebraic geometry*. Vol. 1. Springer. doi:10.1007/978-3-642-37956-7

[33] Alexandra Shlapentokh. 2011. Defining Integers. *The Bulletin of Symbolic Logic* 17, 2 (2011), 230–251. doi:10.2178/bsl/1305810912

[34] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From program verification to program synthesis. *SIGPLAN Not.* 45, 1 (Jan. 2010), 313–326. doi:10.1145/1707801.1706337

[35] Alfred Tarski and J. C. C. McKinsey. 1951. *A Decision Method for Elementary Algebra and Geometry* (dgo - digital original, 1 ed.). University of California Press. https://doi.org/10.1525/9780520348097

[36] Joachim Von Zur Gathen and Jürgen Gerhard. 2013. *Modern computer algebra*. Cambridge university press. doi:10.1017/CBO9781139856065

Received 2025-03-05; accepted 2025-04-25

Optimizing Type Migration for LLM-Based C-to-Rust Translation: A Data Flow Graph Approach

Qingxiao Xu
Texas A&M University
College Station, USA
qingxiao@tamu.edu

Jeff Huang
Texas A&M University
College Station, USA
jeff@cse.tamu.edu

Abstract

Translating legacy C codebases to Rust is crucial for improving memory safety, yet automating this process remains challenging for real-world projects. In this paper, we document key technical challenges we encountered in manually translating the Gzip GNU package using state-of-the-art LLMs (e.g., the OpenAI o1 model). To address the incompatibility between C and Rust type systems, we propose an optimized method aimed at providing the appropriate context and prompts to ensure correct variable types at declaration stage. We extract essential code semantics by generating data flow graphs (DFG). For global variables that may induce multiple borrowing, we prompt LLM to generate enums, and for pointer variables (scalar pointers, array pointers and pointer arithmetic), we prompt LLM to map them to correct Rust types. By leveraging DFG context information and prompts with clear instructions, our approach enables o1 to achieve more accurate type translation compared to direct translation. Our results are publicly available and demonstrate the effectiveness of our approach in improving type migration.

CCS Concepts: • **Software and its engineering → General programming languages**.

Keywords: Code translation, Type migration, Large language model, Data flow graph

ACM Reference Format:
Qingxiao Xu and Jeff Huang. 2025. Optimizing Type Migration for LLM-Based C-to-Rust Translation: A Data Flow Graph Approach. In *Proceedings of the 14th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '25), June 16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3735544.3735582

1 Introduction

Code translation is a critical yet challenging task that often requires expertise in both the source and target languages. Many rule-based [7] and learning-based methods [1, 8, 9] have been proposed to automate this process, including approaches leveraging Large Language Models (LLMs). LLM-based code translation has shown considerable promise, gaining significant interest from researchers [3–5, 11, 13, 14] due to its ability to generate syntactically and semantically valid translations without extensive manual effort.

However, current research has not achieved a fully automated translation of C repositories into Rust [6, 10]. Existing efforts typically stall due to incomplete compilations, highlighting a fundamental gap in LLM-based translation. Wang et al. proposed a benchmark for repository-level code translation, named RepoTransBench [12]. They found that even the best-performing LLM achieved a *Success*@1 score of only 21%. While LLMs have demonstrated effectiveness in translating small C code snippets to Rust, the specific barriers preventing their success in repository-level code migration remain unclear.

In this work, through manually translating the Gzip GNU package [2] with the assistance of LLMs (e.g., o1), we identify a key challenge in LLM-based translation: type incompatibility. Rust enforces strict memory safety guarantees that differ fundamentally from C's type system, making LLM-based translation error-prone. To address this problem, we propose an optimized type migration framework that incorporates essential program semantics. This framework includes: (1) the use of Data Flow Graphs (DFGs) to assist in type migration decisions, (2) the generation of enum indicators for parameter borrowing, and (3) pointer type mapping to convert C pointers. By analyzing variable usage patterns, this framework enables more precise type selection [1].

In our experiments with Gzip, our approach significantly outperforms the baseline of direct translation (i.e., direct LLM prompting without our optimizations). It successfully eliminates issues with multiple mutable borrows and ensures better handling of type migration by introducing 14 new enumerators and correctly declaring 17 pointer types. Additionally, our method accurately translates pointer types without relying on reference-counting cells, enabling pointer accessing and arithmetic that retain the original semantics of C pointers. In contrast, the baseline approach retains excessive global variables, loses some variables due to missing data flow information, and struggles with pointer translation.

[1]Our codes are available at https://github.com/KITKATXU/gzip-rust and https://github.com/KITKATXU/DFG-C2Rust.

This work is licensed under a Creative Commons Attribution 4.0 International License.

SOAP '25, Seoul, Republic of Korea
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1922-6/25/06
https://doi.org/10.1145/3735544.3735582

Previous work has explored various approaches to enhancing C-to-Rust translation, such as passing the callee function context and replacing function return types with their Rust equivalents [4, 5, 11]. The closest related work to ours is by [5], which identifies the main challenge as the *m*-to-*n* correspondence between C and Rust types. Their approach prompts an LLM to generate candidate Rust signatures and then compares them to select optimal types. When the generated code fails to compile, they iteratively fix type errors using compiler feedback. Although this comparison-based type selection is a sensible starting point, we found it insufficient. Many compilation errors arise from low-level issues, making their evaluation metrics, such as error message counts and the variety of Rust types, potentially unreliable.

Our contributions are summarized as follows:

- We manually translated the Gzip GNU C package to Rust. To the best of our knowledge, it is the first successful case of repository-level C-to-Rust translation. Through this effort, we identified key gaps in aligning type systems between C and Rust in current LLM-based methods, highlighting that "*type migration matters*."
- We introduce a series of methods that automatically extract essential program logic and present relevant information to facilitate Rust type selection. To address the recurring issues in parameter borrowing, we introduce the use of enum indicators. Furthermore, we classify C pointers based on their intent and map them to three corresponding Rust types.
- We evaluate our approach on the Gzip project. The results demonstrate the importance of generating enum indicators and implementing pointer mapping for both global and local variables. However, achieving fully accurate and automated translation remains an open challenge. Through a detailed analysis of manual translations, we identify additional gaps that future research must address.

2 The Complexities of Type Migration

This section analyzes the root causes of type incompatibilities, which often arise from incorrect variable translation by LLMs and cannot be resolved through local patches.

2.1 Multiple Mutable References

Because Rust discourages global variables to ensure memory and concurrency safety, global variables and functions in C must be translated into struct fields and methods in Rust, as illustrated in Fig. 1.

Methods in Rust are defined within an `impl` block, which gives them access to the struct's fields and enables additional functionality. When writing a Rust method, the first parameter is conventionally `self`, referencing the struct instance on which the method is called. However, if a C function originally takes global variables as parameters, a direct translation might produce signatures like (`self, self.a`) in Rust. Because `self.a` is part of `self`, Rust's borrowing rules interpret this as multiple mutable references to the same underlying data within a single call, violating its safety guarantees. In practice, `self` won't appear before `self.a`, making it harder for LLMs to detect and fix errors. This violates Rust's safety rules and leads to borrowing multiple parts of `self` within one method call.

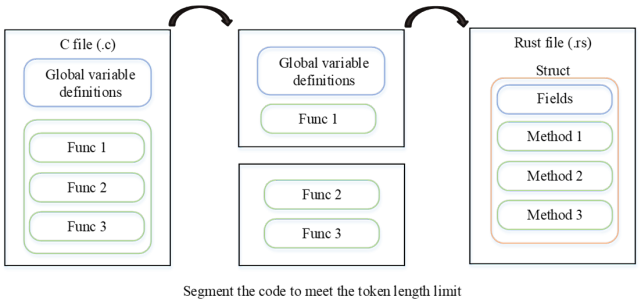


Figure 1. Global variables and functions in C are translated into struct fields and methods in Rust.

Example: Conflicts from Translating Global Variables to Struct Fields . (Listing 1) Consider the following C code snippet, where functions pass global variables among themselves.

```c
// global variables
int x = 10;
int y = 20;
// modify functions
void modify(int* value){
    (*value)++;
}
void modify_x(){modify(&x);}
void modify_y(){modify(&y);}
int main(){
    modify_x();
    modify_y();
    return 0;
}
```

Listing 1. C functions that pass global variables

In Rust, these global variables become struct fields, and the corresponding function calls lead to multiple parts of `self` being borrowed simultaneously (Listing 2).

If the compiler emits errors, LLMs often suggest incorrect fixes such as passing cloned data, an approach that deviates significantly from the original code's semantics. This underscores the challenge of handling Rust's strict borrowing rules when translating C code that relies heavily on global variables.

```rust
struct MyStruct {x: i32, y: i32}
```

```rust
impl MyStruct {
    fn new() -> Self{
        MyStruct{x: 10, y: 20}
    }
    // Requires &mut self and another &mut param
    fn modify(&mut self, x:&mut i32){
        *X += 1;
    }
    fn modify_x(&mut self){
        // Error! Attempting to borrow self twice
        self.modify(&mut self.x);
    }
    fn modify y(&mut self){
        // Error! Attempting to borrow self twice
        self.modify(&mut self.y);
    }
}
```

Listing 2. Borrow multiple parts of self in one method call

2.2 Implicit Pointer Semantics

In C, pointers can be incremented or decremented to directly access or manipulate memory addresses. By contrast, Rust's `Vec` only allows element access through indexing and does not support pointer arithmetic. Consequently, pointers from C cannot be naively translated into Rust's `Box`, `reference`, `Vec`, or `slice` types. LLM-based translations often fail to make this distinction because C uses the same declaration syntax for both scalar and array pointers (see Listing 6, Listing 8).

C pointer arithmetic further complicates translation. The language does not require tracking the starting position for pointer operations; Rust, on the other hand, relies on constructs like `slice`.

While `slice` can safely manage contiguous memory regions, they introduce restrictions on overlapping mutable references. If the original code expects multiple pointers into the same array (with potential overlap), mechanically converting these pointers into `slice` can trigger Rust's borrow checker errors.

Example: Pointer Arithmetic Misinterpreted as Slices. Listing 3 demonstrates a scenario where pointer arithmetic should not be translated as `slice`.

```c
int main(){
    int arr[]={1,2,3,4};
    int *ptr1 = arr;
    int *ptr2 = arr+2;
    ptr1[0]= 10;
    ptr2[0]= 20;
    return 0;
}
```

Listing 3. C Code snippet with pointer arithmetic

In practice, safe translation to `slice` often requires reordering or refactoring the code to avoid overlapping mutable borrows. If the function is large, incorrectly placed `slice` can lead to borrowing conflicts. Listing 4 shows both the original code sequence and a refactored version translated to `slice`.

```rust
//Keep the original code sequence: cannot borrow
    arr as mutable more than once at a time.
fn main(){
    let mut arr =[1,2,3,4];
    let slice1 = &mut arr[..];
    let slice2 = &mut arr[2..4];
    slice1[0] = 10;
    slice2[0] = 20;
}
//Change the code sequece: correct translation.
fn main(){
    let mut arr =[1,2,3,4];
    let slice1 = &mut arr[..];
    slice1[0] = 10;
    let slice2 = &mut arr[2..4];
    slice2[0] = 20;
}
```

Listing 4. Translate to slice

2.3 Code Segmentation and Intent Loss

Translating repositories often exceeds the token limit of LLMs, necessitating truncation before input. To address this, Shiraishi and Shinagawa [11] propose a context-aware code segmentation approach that parses both C and Rust units, generating contextual metadata to include in the prompt. Despite these measures, type declarations are still frequently translated without considering how they are subsequently used. As a result, variables lack full context regarding their intended roles, leading to partial intent capture and ultimately causing type information loss.

3 Our Approach

In this section, we present our methodology for translating C code to Rust. We leverage Data Flow Graphs (DFGs) to guide type selection, and we introduce two specialized mechanisms for handling challenging C-to-Rust mappings: enum indicators to address multiple mutable borrowing and pointer mapping to handle pointer semantics.

3.1 A Data Flow Graph Approach

We employ DFGs to track how data propagates between operations and functions, focusing on each variable's definitions, assignments, and usages. Each DFG is built specifically for a target variable, capturing only the essential information. Unrelated code is omitted to keep the graph compact and thus reduce the LLM's input size. By examining how a variable is used in subsequent code, the LLM can select an appropriate Rust type that aligns with future operations.

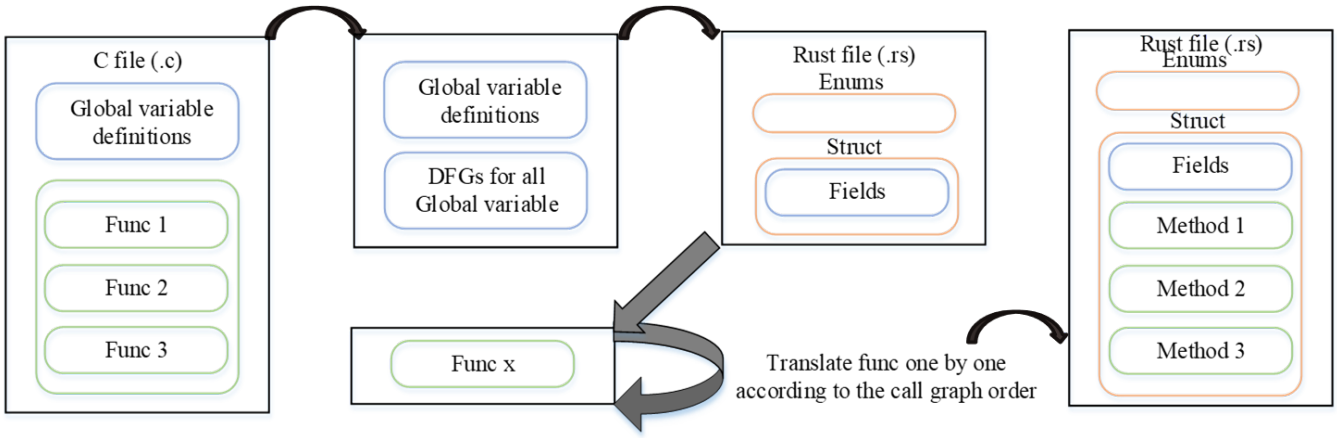


Figure 2. Translation of global variables using DFGs.

For example, if a DFG indicates that a global variable is frequently passed mutably, we generate an enum indicator (Section 3.2), whereas evidence of pointer arithmetic leads us to a pointer-to-index mapping approach in pointer type selection (Section 3.3).

Fig. 2 shows the complete translation process for global variables using DFGs. First, we use Clang's static analysis tool to generate a DFG for each global variable. If a global variable is passed as a parameter to any function in its corresponding DFG, we generate an enum indicator. Then, we augment the translation of each function with Rust definitions for these parameters, incorporating any relevant enum indicators. Leveraging the DFG in this manner also helps to maintain compliance with token length limits. In our experiments, this approach also eliminated erroneous references when global variables function as members of other structures, underscoring the effectiveness of DFG-based analysis.

However, recursive cyclic calls pose a challenge for the DFG approach: while DFGs of small programs often fit within the context window and capture recursion, those of larger programs may be truncated due to token limitations.

3.2 Generating Enum Indicator Support

Since Rust does not support global variables, C global variables are translated into struct fields. This leads to an issue where methods that take these fields as parameters often borrow multiple parts of the same `self`, violating Rust's safety rules. To address this, we introduce **enum indicators** for selected field definitions at their initial declaration.

We examined three possible solutions, each demonstrated with the example in Listing 1: (1) redesign the code; (2) set auxiliary functions; and (3) use enumerators. See Supplemental Material for a comparison. In this section, we focus on (3).

Use of enum indicators. (Listing 5). Replacing direct field references with an enum that identifies which field to modify avoids passing multiple mutable references. This method requires only minimal changes to the original C code, preserving both readability and functionality.

```
struct MyStruct {X: i32, y: i32}
enum Field {X, Y}
impl MyStruct {
    fn new() -> Self {
        MyStruct { x: 10, y: 20 }
    }
    // Use enum to specify which field to modify
    fn modify(&mut self, field: Field) {
        match field {
            Field::X -> self.x += 1,
            Field::Y -> self.y += 1,
        }
    }
    fn modify_x(&mut self){self.modify(Field::x);}
    fn modify_y(&mut self){self.modify(Field::Y);}
}
```

Listing 5. Use of enum indicators

3.3 Pointer Type Mapping

Fig. 3 describes our strategy for distinguishing among scalar pointers, array pointers, and pointer arithmetic.

We declare all pointers first, then translate the remainder of the functionality:

1. **Generate Pointer DFGs.** For each local pointer variable, Clang's static analysis tool creates a DFG outlining key pointer operations, such as memory allocation (e.g., `malloc`) or pointer arithmetic.
2. **Determine Pointer Type.** The pointer declaration and its DFG are fed to the LLM, which decides among `Box<>`, `reference`, `Vec`, `Box<[]>`, or `Index`.
   - *Unallocated array pointer*: Use `Vec` or `Box<[]>`.

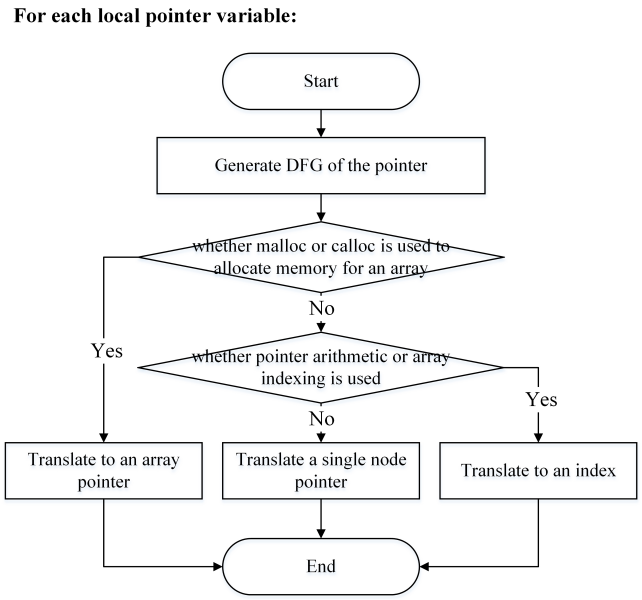


Figure 3. Select Rust types based on the DFGs.

- *Pointer to array elements*: Use `Index` type that stores the pointer's initial position and handles pointer arithmetic.
- *Other pointers*: Default to `Box<>` or `reference`, depending on usage.

Examples: Translation of C Pointers to Rust Types. We illustrate three scenarios (Listings 6–11) that require distinct pointer handling at declaration in Rust, but are not demanded by C's uniform pointer syntax.

```
int value = 42;
int* ptr;
ptr = &value;
free(ptr);
```

Listing 6. C code snippet with scalar pointer

```
// Option 1: Box(owned)
let ptr: Box<i32>= Box::new(42);
// Option 2: Reference
let value = 42;
let ptr: &i32 = &value;
```

Listing 7. Rust equivalent for scalar pointer

```
int* ptr;
ptr = malloc(size*sizeof(int));
ptr[0]= 1;
free(ptr);
```

Listing 8. C code snippet with array pointer

```
// Option 1:vec(dynamic array)
let arr_ptr:vec<i32>= vec::with_capacity(size);
// Option 2:Box<[T]>(fixed-size array)
let arr _ptr: Box<[i32]>=vec![o; size].into
    boxed_slice();
```

Listing 9. Rust equivalent for array pointer

```
int* ptr;
int arr[]={1,2,3,4,5};
ptr = &arr[2]; // points to third element
ptr++; // move to next element
```

Listing 10. C code snippet with pointer arithmetic

```
let arr=vec![1,2,3,4,5];
// Option 1: using index
let index:usize =2;
let value = arr[index];
// Option 2: using slice
let slice:&[i32]=&arr[2..];// start from index 2
```

Listing 11. Rust equivalent for pointer arithmetic

In section 4, two pointers that should have referred to arrays were incorrectly translated into index values. The primary cause of this issue was that the Data Flow Graphs (DFGs) of callee functions is missing in the context: some pointers were allocated within callee functions.

Fig. 4 shows the optimized pointer-type mapping process, with a simplified call graph illustrating the relationship between caller and callee functions.

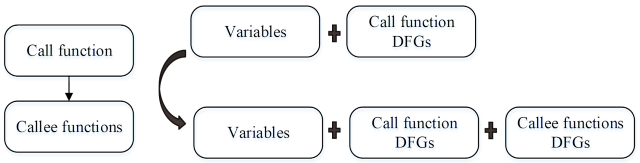


Figure 4. Optimized approach for pointer type mapping.

3.4 Pre and Post-processing by Manual Intervention

The DFG approach is not fully automated and requires the involvement of developers with a basic understanding of C and Rust. Thus, manual intervention remains necessary to ensure producing compilable code, posing challenges to automation and scalability.

Specifically, the manual tasks include:

1. Feeding the automatically generated Data Flow Graph (DFG) from Clang and variable definitions to the LLM to iteratively translate to Rust variables types;
2. Segmenting the function bodies and passing them along with the translated variables to the LLM for translation;
3. Integrating the translated code fragments;
4. Fixing compilation errors.

4 Experiments

Our approach targets repository-level translation, where translation quality is ideally measured by the number of passed tests after compilation. However, like all existing methods, ours struggles to produce compilable code, and compilation error counts do not reliably reflect translation quality. We therefore manually verified translation accuracy by comparing variable types against those in our fully manually translated, compilable code.

4.1 Setup and Implementation

We translated 151 global variables from 12 C files in the Gzip C package, as well as 76 local variables from 8 functions in inflate.c. For our experiment, we used ChatGPT-O1 as the language model. The prompt templates is in Supplemental Material.

Baseline Comparison. To provide a baseline, we also conducted a translation task in which the model received only the C variables and function headers, without any additional context or DFG information.

4.2 Results

The experimental results of both global variable and pointer variable translation are presented in this subsection.

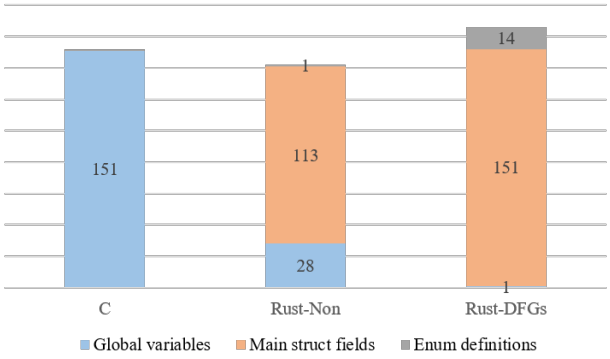


Figure 5. Translation results of global variables.

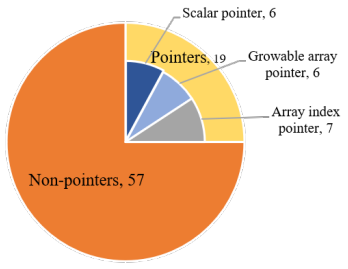


Figure 6. The distribution of local variables in inflate.c.

Translation of Global Variables. Global variables in C were translated into Rust struct fields to ensure safe memory management and satisfy Rust's lifetime requirements (see Section 3). Because certain global variables are passed as parameters to functions or stored in other structs, our method creates enum indicators at the declaration stage. Fig. 5 summarizes the number of translated global variables (constants and static variables), newly introduced enum indicators, and the state struct fields.

Among all 151 global variables, our approach introduced 14 enum indicators. This highlights the necessity of these indicators to prevent multiple mutable borrowing. By contrast, the baseline approach lacked the data flow information required to trigger enum generation. Consequently, it retained most global variables in their original form and introduced no new enum values.

Translation of Pointer Variables. We selected this inflate C file because it is largely self-contained, defining its own tree node structure and avoiding calls to external C files. Fig. 6 illustrates the distribution of local variables in inflate.c.

Our approach processes each function in two steps:

1. Identify function boundary (i.e., its start and end line).
2. Use Clang's static analysis to build the DFG for all variables in that function.

The resulting DFGs, together with the prompt templates, form the input to the LLM. In contrast, the baseline method translates the variables without data flow information.

Pointers need be explicitly distinguished at the declaration stage in Rust. As shown in Table 1, our approach successfully identified the correct pointer types whereas the baseline resorted to reference-counting cells. This fallback does not guarantee a single mutable reference and struggles to handle pointer arithmetic.

Table 1. Translation results of pointers

Type	Direct translation	Translation-DFGs
Scalar pointer	12	6
Array pointer	0	4
Index	0	9
RefCell	7	0

5 Conclusion

We have identified key gaps in translating variable types between C and Rust in existing LLM-based methods, and presented a new approach based on DFGs. For global variables, we propose using enums to eliminate repeated borrowing errors. For C pointers, we prompt the LLM to select appropriate Rust types, including dynamic arrays and indexes. The evaluation results on translating Gzip C indicate that our approach is effective in improving type migration from C to Rust.

References

[1] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems* 31 (2018).

[2] Kunpeng Compute. 2024. gzip. https://github.com/kunpengcompute/gzip Accessed: 2024-12-20.

[3] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2024. Towards Translating Real-World Code with LLMs: A Study of Translating to Rust. *arXiv preprint arXiv:2405.11514* (2024).

[4] Jaemin Hong and Sukyoung Ryu. 2024. Don't Write, but Return: Replacing Output Parameters with Algebraic Data Types in C-to-Rust Translation. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 716–740. https://doi.org/10.1145/3656406

[5] Jaemin Hong and Sukyoung Ryu. 2025. Type-migrating C-to-Rust translation using a large language model. *Empirical Software Engineering* 30, 1 (2025), 3. https://doi.org/10.1007/s10664-024-10573-2

[6] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Repository-level compositional code translation and validation. *arXiv preprint arXiv:2410.24117* (2024).

[7] Immunant. [n. d.]. C2Rust: Migrate C code to Rust. https://github.com/immunant/c2rust. Accessed: 2024-10-23.

[8] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511* (2020).

[9] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584. https://doi.org/10.1109/ASE.2015.36

[10] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. https://doi.org/10.1145/3597503.3639226

[11] Momoko Shiraishi and Takahiro Shinagawa. 2024. Context-aware Code Segmentation for C-to-Rust Translation using Large Language Models. *arXiv preprint arXiv:2409.10506* (2024).

[12] Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, et al. 2024. RepoTransBench: A Real-World Benchmark for Repository-Level Code Translation. *arXiv preprint arXiv:2412.17744* (2024).

[13] Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. Codetransocean: A comprehensive multilingual benchmark for code translation. *arXiv preprint arXiv:2310.04951* (2023).

[14] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1585–1608. https://doi.org/10.1145/3660778

Received 2025-02-28; accepted 2025-04-25

Compositional Static Callgraph Reachability Analysis for WhatsApp Android App Health

Ákos Hajdu
Meta
London, UK
akoshajdu@meta.com

Roman Lee
Meta
Vancouver, Canada
romanlee@meta.com

Gavin Weng
Meta
Menlo Park, USA
gavinweng@meta.com

Nilesh Agrawal
Meta
Menlo Park, USA
niles@meta.com

Jérémy Dubreil*
Unaffiliated
Nantes, France
jeremy.dubreil@lacework.net

Abstract

We report on an industrial use case of static callgraph reachability analysis to improve WhatsApp Android app health. We collaborated with engineers dedicated to app health to annotate/specify the source code. We leveraged the Infer static analyzer to prevent regressions during code changes and to periodically find pre-existing issues on the latest revision. Within three months, the analysis prevented almost a hundred regressions from being introduced and resulted in fixes for a handful of pre-existing issues, including examples with end-user measurable impact.

CCS Concepts: • **Software and its engineering → Automated static analysis**.

Keywords: Static analysis, callgraph reachability

ACM Reference Format:
Ákos Hajdu, Roman Lee, Gavin Weng, Nilesh Agrawal, and Jérémy Dubreil. 2025. Compositional Static Callgraph Reachability Analysis for WhatsApp Android App Health. In *Proceedings of the 14th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '25), June 16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3735544.3735583

1 Introduction

WhatsApp is a messaging application serving more than 2 billion users in over 180 countries every day. One of WhatsApp's goals is to provide fast and reliable communication, including lower-end consumer devices and poor network conditions. In order to achieve this goal, special emphasis needs to be put on app health and code quality. In this paper we describe a static callgraph reachability analysis on WhatsApp Android app's codebase. The analysis finds potential issues where certain performance critical functions can end up (transitively) calling computationally expensive functions. In order to deploy such an analysis, we needed (1) to specify functions of interest (performance critical and computationally expensive ones), and (2) an automated tool that takes the specification and the source code, and performs the analysis.

Specification is a well-known challenge in program analysis: it is often hard to motivate developers to write specifications, especially at a large scale in a fast-paced company. We tackled this challenge by collaborating with engineers from a team specifically working on app health to provide an initial set of specifications, which was then expanded by other developers as we gradually rolled out the analysis.

The main challenge for the analysis is that it needs to be fast enough to give timely feedback on code changes submitted by developers, while also being sufficiently sound and precise. To tackle this, we leveraged Infer [3, 4], a static analysis platform with multiple language frontends and analyzers (also called checkers). Infer had a so-called *annotation reachability* checker, which could perform static callgraph reachability analysis based on in-code annotations. A distinguishing feature of this checker is that it does not construct and traverse a global callgraph, but uses Infer's compositional and interprocedural abstract interpretation framework. This checker's functionalities were mostly fit for our purposes, but it was mostly unused and unmaintained. We "revived" this checker and added some new features needed for our use case (e.g., supporting regexp-based specification of library functions). In addition, we formalized the algorithm of the checker in this paper for the first time.

We deployed the analysis to run on every code change (*diff*) to prevent regressions, and also to run periodically on the latest revision to find pre-existing issues. Over a period of 3 months, Infer reported 174 issues on diffs with 92 being fixed (53% fix rate). Furthermore, 7 pre-existing issues were also fixed, which is uncommon, as Infer has traditionally

*The author was affiliated with Meta while contributing to this work.

This work is licensed under a Creative Commons Attribution 4.0 International License.
SOAP '25, Seoul, Republic of Korea
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1922-6/25/06
https://doi.org/10.1145/3735544.3735583

been more successful at diff-time deployments [6]. One notable example issue found by Infer resulted in measurable impact for end-users: app not responding (ANR) issues were reduced by 0.56% in particular regions, and chat loading got 1.25% faster globally.

2 Background

Reachability. Given a program P consisting of a set of functions F and sets of distinguished *source* functions $F_{src} \subseteq F$ and *sink* functions $F_{snk} \subseteq F$, the goal of reachability analysis is to check if any source function can (potentially transitively) call a sink function, that is, whether an execution exists in P with a sequence of calls $f_1 \to f_2 \to \ldots \to f_n$ for some $n > 1$, where $f_1 \in F_{src}$, $f_n \in F_{snk}$, and $f_i \in F$ for $1 \leq i \leq n$. Reachability can be extended with an additional set of *sanitizers* $F_{san} \subseteq F$ imposing an extra condition $f_i \notin F_{san}$ for $1 \leq i \leq n$. In other words, we are not interested in call sequences that pass through any sanitizer function.

Example. Consider the program in Figure 1 with `somethingOnTheUI` being the single source and `readFromDatabase` being the single sink. Then, a sequence of calls `somethingOnTheUI` → `checkSomething` → `readFromDatabase` exists from source to sink. However, if we consider `checkSomething` to be a sanitizer, then there are no source-to-sink call sequences anymore.

```
1 @PerfCrit void somethingOnTheUI() {
2   checkSomething();
3 }
4 void checkSomething() {
5   readFromDatabase();
6 }
7 @Expensive void readFromDatabase() {
8   /* Slow stuff */
9 }
```

Figure 1. Example code with annotated functions.

Static callgraphs. As usual with non-trivial program properties, callgraph reachability is an undecidable problem[1] in theory. In this work, we use *static callgraphs* to solve reachability in an approximate way. The static callgraph of a program P is a directed graph $G = (F, E)$ where the vertices F consist of the functions of the program, and there is a directed edge $(f_1, f_2) \in E$ between functions $f_1, f_2 \in F$ if the body of f_1 contains a call instruction of the form $f_2(\ldots)$. An approximate solution to the reachability problem is to check if a path $f_1 \to \ldots \to f_n$ exists in G where $f_1 \in F_{src}$, $f_n \in F_{snk}$, and $f_i \notin F_{san}$ for $1 \leq i \leq n$.

This approximation can be solved, e.g., by computing a transitive closure of the graph [7]. However, it can report paths that are not feasible in any actual execution (e.g., calls guarded by infeasible conditions) and also miss real call chains (e.g., if dynamic dispatch or lambdas are involved). For us, this is acceptable: we are not aiming for a sound over-approximation. False negatives are acceptable as long as the analysis is fast enough to run on every code change, and can detect enough real issues with a sufficiently low level of false positives to make an impact in practice.

3 Reachability Analysis with Infer

Infer [3, 4] is a static analysis framework that supports multiple language frontends (including Java and Kotlin) and analyzer backends (checkers) through a common intermediate language [2]. One of Infer's checkers is called *annotation reachability*[2] which can perform the aforementioned reachability analysis of programs based on the static callgraph; with a few practical differences and extensions. This checker has been around in Infer for a while, originally developed to solve similar problems on other Android apps, but – to the best of our knowledge – this is the first paper to present it in a formal way. As described later in this section, a key feature of this algorithm is that it does not construct a global callgraph, but rather uses compositional reasoning and abstract interpretation. We have also added some improvements and extensions to the checker to support our particular use case at WhatsApp (Section 3.4).

3.1 Specification

As the name of the checker suggests, the sources, sinks and sanitizers can be defined via sets of Java *annotations*. Given the set of all annotations A, let $A(f) \subseteq A$ denote the annotations of a function[3] f and let $A_{src}, A_{snk}, A_{san} \subseteq A$ denote the source, sink and sanitizer annotations, respectively. Without the loss of generality, we can assume that no sink or source annotation is a sanitizer at the same time ($A_{src} \cap A_{san} = \emptyset$ and $A_{snk} \cap A_{san} = \emptyset$), as otherwise any path containing them would be immediately sanitized. We can derive the set of source, sink and sanitizer functions as follows: $F_{src} := \{f \in F \mid A(f) \cap A_{src} \neq \emptyset\}$, $F_{snk} := \{f \in F \mid A(f) \cap A_{snk} \neq \emptyset\}$ and $F_{san} := \{f \in F \mid A(f) \cap A_{san} \neq \emptyset\}$. Note that we only assume that a source/sink annotation cannot be a sanitizer annotation; a source/sink function can still be a sanitizer at the same time if it has both source/sink and sanitizer annotations.

Additionally, the checker takes as input multiple sets of sources, sinks and sanitizers $(A^1_{src}, A^1_{snk}, A^1_{san}), \ldots, (A^k_{src}, A^k_{snk}, A^k_{san})$, treating each tuple as an independent reachability *property* (but solving all of them in one pass). For example, the configuration in Figure 2 defines a single property with one source, one sink and no sanitizers: ($A^1_{src} = \{$`@PerfCrit`$\}$, $A^1_{snk} = \{$`@Expensive`$\}$, $A^1_{san} = \emptyset$).

[1] We can append a synthetic function call *end*() to the end of a program P and the reachability of *end*() is equivalent to the halting of P.

[2] https://fbinfer.com/docs/checker-annotation-reachability

[3] Infer's Java/Kotlin frontend also looks at annotations from the enclosing class and from overridden functions from base classes / interfaces.

```
"annotation-reachability-custom-pairs": [{
  "sources": ["PerfCrit"],
  "sinks": ["Expensive"]
}]
```

Figure 2. Infer's annotation reachability configuration.

3.2 Analysis

The checker does not explicitly construct a global callgraph to traverse. Instead, it formulates the problem in Infer's compositional and interprocedural abstract interpretation framework. Infer analyzes each function independently by propagating an abstract state through its instructions to obtain a *summary* which can then be applied in every calling context (without having to re-analyze the callee). This also means that the analysis does not need an entry point or a main function: it can start from a set of arbitrary functions, and dependencies are analyzed *on-demand*. As demonstrated in the results (Section 4.3), this allows us to run the checker on every code change at the scale of WhatsApp within a reasonable time. However, recursion needs a special treatment (due to circular dependency in the analysis): either a fixed-point computation, or unrolling to a fixed depth. Infer currently does an unrolling to a depth of one,[4] i.e., performing an under-approximation.

Abstract domain. In annotation reachability, an abstract state $s \subseteq F \times \mathbb{N} \times F \times A$ is a set of tuples, where a tuple $(g, n, h, a) \in s$ represents the information that the current function f being analyzed calls some function g on line n, which then ends up calling sink h, annotated with $a \in A^i_{snk}$ for some i. If $g = h$ in a tuple, it means that the current function f calls a sink *directly*, whereas $g \neq h$ means that the call is *transitive*. Intuitively, this abstract domain implies that we only compute the relevant parts of the callgraph (that can potentially be part of a source-to-sink call chain), and each node only stores a single step towards a given sink. Paths can be reconstructed later, and we have the option to limit the number of paths per source/sink pair, allowing us to scale to large codebases.

Instructions are traversed from the beginning of the function in-order, however, as described below, the join operation and the transfer function is defined in a path- and flow-insensitive way for scalability.[5] The initial state $s_0 := \emptyset$ is the empty set, the join operation is set union $join(s_1, s_2) := s_1 \cup s_2$, and the summary of a function f – denoted by $sum(f)$ – is the abstract state at the exit point[6] of f.

The transfer function $T(s, instr_n)$ takes an abstract state s (the precondition) and an instruction $instr_n$ at line n, and returns a new abstract state that encapsulates the effects (postcondition) of executing $instr_n$ on s. If $instr_n$ is not a call, or the callee cannot be resolved statically (e.g., some unknown external function), then $T(s, instr_n) := s$, that is, the state does not change. If $instr_n$ is a call to some function g then the state gets extended with *direct* and *transitive* entries corresponding to g. Direct entries s_d are added if g itself is a sink, and transitive entries s_t are added for each entry in the summary of g (unless f or g is a sanitizer). Formally, $T(s, instr_n) = s \cup s_d \cup s_t$ where

- $s_d := \{(g, n, g, a) \mid a \in A(g) \wedge \exists i.\, a \in A^i_{snk} \wedge f, g \notin F^i_{san}\}$ and
- $s_t := \{(g, n, h, a) \mid (h', n', h, a) \in sum(g) \wedge \exists i.\, a \in A^i_{snk} \wedge f, g \notin F^i_{san}\}$.

3.3 Reporting

Once we finish analyzing a function f and it happens to be a source ($f \in F^i_{src}$ for some i), we check its summary $sum(f)$ to report if some sink g can be reached. In practice, we can have multiple sets of sources/sinks/sanitizers so we also report which exact annotations are responsible for the call chain. Formally, we report $\{(f, a_f, g, a_g) \mid (h, n, g, a_g) \in sum(f), \exists i, a_f.\, a_f \in A^i_{src} \wedge a_g \in A^i_{snk} \wedge f \notin F^i_{san}\}$. A reported tuple (f, a_f, g, a_g) should be interpreted as "source f annotated with a_f calls sink g annotated with a_g". In Infer's terminology, this is called an *issue*.

Summaries also allow us to reconstruct all distinct paths between a source and a sink. However, in practice, this can lead to an exponential number of paths, so the checker only reports one path per issue. A path for an issue (f, a_f, g, a_g) can be reconstructed recursively:

- $path(f, g, a_g) := f \rightarrow g$ if $\exists n \in \mathbb{N}$ with $(g, n, g, a_g) \in sum(f)$,
- $path(f, g, a_g) := f \rightarrow path(h, g, a_g)$ for some entry $(h, n, g, a_g) \in sum(f)$ otherwise. In practice, the checker picks the entry with the lowest n, i.e., the first callsite that leads towards the sink.

Consider the example code and configuration in Figures 1 and 2 and let us abbreviate each function with its initial letter. The summary $sum(\texttt{r}) = \emptyset$ is empty because `r` does not contain any call. The summary $sum(\texttt{c}) = \{(\texttt{r}, 5, \texttt{r}, \texttt{@Expensive})\}$ contains a single direct call to a sink: `c` calls `r` directly on line 5. The summary $sum(\texttt{s}) = \{(\texttt{c}, 2, \texttt{r}, \texttt{@Expensive})\}$ contains a single transitive call to a sink: `s` calls `r` indirectly via the call to `c` on line 2. The only source is `s`, so Infer would report a single issue: (`s`, `@PerfCrit`, `r`, `@Expensive`).

[4] Infer used to have fixed-point computation, but in practice, using an unrolling to a depth of one still found the majority of the relevant issues in a simpler and more efficient way, that is, the benefits of deeper unrolling were too small compared to the performance loss.

[5] We have also experimented with a path- and flow-sensitive underapproximate checker called Pulse [13], but within our time constraints, it had too many false negatives.

[6] Infer represents functions as control-flow graphs with one common exit node.

The path is reconstructed as $path(\texttt{s}, \texttt{r}, \texttt{@Expensive}) = \texttt{s} \rightarrow path(\texttt{c}, \texttt{r}, \texttt{@Expensive}) = \texttt{s} \rightarrow \texttt{c} \rightarrow \texttt{r}$.

3.4 Checker Improvements

In order to apply Infer to our use case at WhatsApp Android, we had to make some improvements to the annotation reachability checker itself.

Regular expressions. We extended the checker so that functions can be treated (*modeled*) as if they were annotated, based on regular expressions over their (fully qualified) names. This was necessary for library functions (e.g., Android or Java libraries) where the source code is external and annotations could not be added directly in the code. This allows us to detect call paths where the source is in the application code, but the sink is several call steps inside a library. In addition, this also helped to specify a larger set of functions in a concise way (e.g., marking all UI event handler overrides as sources). Figure 3 shows an example, where the configuration specifies that any method (`.*`) of class `MyClass` in package `com.library` should be treated (modeled) as if it was annotated with `@Expensive`. As a side benefit, this opened up the support for languages without annotations.

```
"annotation-reachability-custom-models": {
 "Expensive": ["com\.library\.MyClass\..*"]
}
```

Figure 3. Using regular expressions to model annotations.

Path minimization. Sources or sinks can also appear as intermediate steps in a path. Such paths contain other paths as a sub-path and it might be desirable to only report the shortest, because eliminating that call chain also removes the extended paths. We formalize this by defining minimal source and sink paths. A path $f_1 \rightarrow \ldots \rightarrow f_n$ (where $f_1 \in F_{src}$ and $f_n \in F_{snk}$) is *source-minimal* if $f_i \notin F_{src}$ for $i > 1$ and *sink-minimal* if $f_i \notin F_{snk}$ for $i < n$. In other words, the only source in a source-minimal path is the first function, and the only sink in a sink-minimal path is the last function. In practice though, we are using annotations: given an issue $(f_1, a_{src}, f_n, a_{snk})$ with a path $f_1 \rightarrow \ldots \rightarrow f_n$, we check if f_1 is the only function of the path annotated with a_{src} (source-minimal) and if f_n is the only function annotated with a_{snk} (sink-minimal). Formally, $a_{src} \notin A(f_i)$ for $1 < i \leq n$ must hold for source-minimal paths, and $a_{snk} \notin A(f_i)$ for $1 \leq i < n$ must hold for sink-minimal paths. Note that we only compute one path for each issue, so an issue is source/sink-minimal if and only if its single corresponding path is source/sink-minimal.

Loop highlighting. Consider a source to sink path $f_1 \rightarrow \ldots \rightarrow f_n$ where f_i calls f_{i+1} in a loop (for some index i). We wanted to flag such paths in a special way because executing something computationally expensive within a loop can have a higher impact than just executing it once. Infer's intermediate language is based on control flow graphs (CFG) and there is an implementation of Tarjan's strongly connected components algorithm [16] to compute loops. We added an option to the analyzer that checks for each edge $f_i \rightarrow f_{i+1}$ whether the CFG node in f_i that contains the call to f_{i+1} is part of a loop. If yes, the edge $f_i \rightarrow f_{i+1}$ is highlighted with a special tag in the trace during reporting.

4 Reachability Analysis for WhatsApp Android

The main goal of the use case at WhatsApp Android is to detect potential issues where certain performance critical functions can end up (transitively) calling computationally expensive functions. To achieve this goal, we partnered with engineers from WhatsApp to annotate the code, configure and deploy the checker, gather feedback and make improvements in an iterative process.

4.1 Specification and Configuration

As of writing the paper, there are two reachability properties (sets of sources, sinks and sanitizers) with 8 annotations and 11 regular expressions in total (see breakdown in Table 1). The sources, sinks and sanitizers are defined by Android engineers, including a dedicated team responsible for the app's health. In the first property, source annotations usually refer to something performance critical, such as event handlers on the user interface (overriding functions from Android UI libraries). Sink annotations correspond to (potentially) computationally expensive functions, such as worker threads and file operations (in Java libraries). Sanitizers include functions that are not shipped in production, such as tests, debug utilities (e.g., logging) and Java preconditions. The second property aims to find call chains where the source and sink has incompatible threading annotations (e.g., worker thread as source calling main thread as sink).

Table 1. Number of annotations and regular expressions with their coverage over WhatsApp Android's functions.

Prop.	Annotations			Regexps			Coverage		
	src	snk	san	src	snk	san	src	snk	san
#1	1	2	2	2	3	3	0.356%	2.689%	0.198%
#2	2	1	0	0	3	0	16.267%	0.002%	0.000%

Specifications partially reused existing annotations (e.g., `@WorkerThread`), but some annotations were added in-code just for the purpose of this checker (e.g., known hot spots related to chat loading). One disadvantage of using regular expressions in Infer's configuration file is that it can get out of sync with the codebase. However, we mostly used

regular expressions for well-established Android/Java library functions that we don't expect to change often. Table 1 shows the percentage of WhatsApp Android's functions that are covered as source, sink or sanitizer by the two properties. The second property has a significantly higher coverage due to reusing more of the threading annotations already present in the code. Note that at the scale of WhatsApp Android's codebase, even small percentages amount to a non-trivial number of functions.

4.2 Deployment

Infer has two kinds of deployments in general: continuous and diff-time. *Continuous* scans analyze the latest revision of a repository periodically (typically a few times every day). With a few exceptions where tasks are filed and triaged to code owners [8, 11], these runs are only visible on internal dashboards for Infer developers to get an idea on the baseline of pre-existing issues and to experiment with new checkers. *Diff-time* deployments, on the other hand, analyze every code change (*diff*) and report newly introduced issues inlined as comments during the code review process (along with feedback from other automated tools and human reviewers). Developers can then submit a new version of the diff, triggering a new run of Infer. We measure *fix rate* by checking if issues reported in an intermediate version of the diff disappear in the final version that gets merged to the main branch. Note that fix rate is only a proxy to estimate true positive rate, but in practice it works well because (1) it can be measured automatically, and (2) it indicates the actionability of the checker from a developer's perspective.

Continuous analysis. We first deployed continuous runs of the reachability analysis, allowing us to get an idea on the baseline of pre-existing issues and to iteratively add more annotations, fine tune the checker, and make improvements to Infer. Overall, most of the issues reported by Infer were technically true positives (a path from source to sink indeed exists). However, without path minimization, Infer reported around[7] 12500 issues, which was deemed overwhelming, even if they were all true positives. Applying source and sink minimization individually reduced the number of issues to approximately 10600 and 1600 respectively. Applying both resulted in roughly 1100 issues. We concluded that sink minimization is sufficient as it already reduces the issues to a manageable volume. In addition, sources are usually closer to the business logic of code owners, whereas sinks are typically in common code or lower level libraries. Therefore, not minimizing on the source side allows engineers to fix the issues in (or closer to) the code they own. While this might not fix all related issues at once, developers are usually more confident in changing the code they own. Engineers also added further sanitizers to suppress certain reports that were technically true positives, but still acceptable because of known mitigating factors.

Diff-time analysis. Once we were satisfied with the quality and volume of reported pre-existing issues, we started rolling out the analysis on diffs. We first enabled a so-called *shadow mode* where the analysis was running for all code changes, but issues only showed up on an internal dashboard for us and the app health team to assess. Then we gradually enabled reporting for a few dozen engineers who opted in as early adopters. We set up channels to gather feedback and monitored fix rate. After a few weeks, we rolled out the analysis to all WhatsApp Android engineers.

4.3 Results

Pre-existing issues. Interestingly, as engineers examined the reports from continuous scans, they have found various pre-existing issues that were likely to have a high impact on performance. For such issues, they created tasks, first manually, then later in a semi-automated way, and triaged them to the appropriate code owners to consider fixing. As of writing the paper, 59 tasks have been filed with 7 fixed already (closed with a diff attached). One notable example was directly linked to ANRs (app not responding) from production logs. Infer's call trace provided insight to why the ANRs are happening. The fix reduced ANRs by 0.56% in certain regions where the feature was enabled and also resulted in an end-user measurable 1.25% speedup in chat loading speed globally.[8]

Diff-time reports. Over a period of 3 months, Infer reported 174 issues on code changes in total, with 92 being fixed (53% fix rate). Our analyis is on par with other Infer deployments, typically having a fix rate between 40% and 60%. We looked at some of the unfixed issues and categorized them in the following main groups.

There are diffs that added further source/sink annotations with the intention to be able to see pre-existing issues (without fixing them) and to prevent newly introduced issues. Such diffs caused Infer to find new issues by design.

We have also seen diffs that converted certain modules from Java to Kotlin. Infer has some mechanisms to identify if an issue moves around (e.g., lines shifting, or class renamings), however, moving to a new language was beyond the capabilities of these heuristics.

We encountered false positives due to pre-existing issues being reported as if they were newly introduced. One notable category was related to non-deterministic analysis in case of mutually recursive function calls. As an example, consider a static callgraph of the form $src \rightarrow f \rightleftarrows g \rightarrow snk$. As mentioned before, Infer analyzes recursive calls to a depth of one, which can intuitively be thought of as having to "cut"

[7] Due to the codebase evolving, and internal timeouts, there is a small fluctuation in the number of issues reported on the latest revision.

[8] Measured by a controlled experiment where one group had the fix enabled and the other group did not have it.

edges from the callgraph until it becomes acyclic. It is clear that if $f \leftarrow g$ is cut, Infer still sees the issue (path from source to sink exists), but if $f \rightarrow g$ is cut, then the issue is not found. The edge to be "cut" depends on the order in which functions are scheduled to be analyzed, which can have some non-determinism, especially if the surrounding code changes. On each diff, Infer runs twice: once for the current revision and once for the parent, and computes the difference of issues to only report newly introduced issues. Our observations suggest that a different scheduling order for the current/parent revision can cause an issue to be only seen in one of the runs. As a quick workaround, we mitigated this by switching to a more deterministic scheduler (the issue is either found or missed, but it is consistent across runs). However, if this causes too many missed issues, we can consider bringing back Infer's fixed-point computation (the issue is found regardless of scheduler ordering).

Finally, we have seen false positives related to flow- and path-insensitivity. Two simplified examples are presented in Figure 4: a call to a sink is guarded by a condition that prevents it from happening in production, and a pair of special function calls "enclose" a block that should be treated as a sanitizer. These could be mitigated by making the analysis flow- and path-sensitive, but that would come with its own downsides (e.g., path explosion).

Note that we did not explicitly count false positives, we only sampled unfixed issues and use the fix rate as a measure of checker quality instead.

```
void source() { // Needs path-sensitivity
  if (is_debug()) sink();
}
void source() { // Needs flow-sensitivity
  beginSanitizing();
  sink();
  endSanitizing();
}
```

Figure 4. Example false positives.

Discussion. Currently, we mostly rely on engineers' domain knowledge to identify which issues have a truly significant impact on performance. Post-fix analysis is possible via experiments, as described earlier with the example directly linked to ANRs. We have also not yet measured whether loop highlighting contributes to higher rate of fixes. In the future we plan to improve actionability and prioritization by using runtime tracing data to identify and flag if a reported call chain is on a hot path (executed often).

Analyzer performance. It is not straightforward to independently assess the performance of the analyzer, because production deployments of Infer (1) rely heavily on caching when compiling the source code into the intermediate representation, and (2) have multiple checkers running in parallel (not just annotation reachability). Overall, the p90 compilation and analysis times for continuous runs are 33 and 53 minutes, respectively, including all checkers and the full WhatsApp Android repository, consisting of millions of lines of Java and Kotlin code. Annotation reachability alone (without compilation) takes around 15 minutes. On diffs, the p90 total execution time of Infer is 32 minutes, including compilation and analysis for parent and current revisions, with all checkers, but limited to the changed files[9] (and their dependencies). The key takeaway from these numbers is that the analysis is fast enough (1) to run multiple times a day in continuous mode, (2) to provide timely feedback on developers' diffs, and (3) annotation reachability is not a bottleneck.

5 Related Work

Samhi et al. [14] survey various Android static analysis tools that compute a callgraph. They show that tools fail to cover at least 40% of the calls that can happen in runtime. Infer – including our analyzer – targets bug finding and not soundness, and even with our static approach we are finding interesting bugs. Nevertheless, it would be interesting to compute metrics on coverage and missed bugs.

There are various static taint analysis tools for Android, such as FlowDroid [1] or Difuzer [15]. While these tools employ a callgraph in the background, the primary purpose is to track data flow. ACID [10] relies on callgraphs to detect API invocation and callback incompatibility issues. ARPDROID [5] and NatiDroid [9] focus on security properties (permissions for API invocations). NatiDroid also extends control-flow analysis to native libraries.

Yang et al. [17] proposes a method to enhance the control flow graph with edges corresponding to events and callbacks, which are traditionally missing from static graphs. In our use case, event handlers are usually the entry points of a path (sources) so we don't have to track where they are invoked from. Midtgaard and Jensen [12] approximate interprocedural control-flow using abstract interpretation. Their main focus is on first class functions and tail call optimization, which are not prevalent at the moment in our use case.

6 Conclusions

We reported on our experience on applying static callgraph reachability analysis on WhatsApp Android's codebase to improve app health and performance. Just within its first 3 months, the analysis prevented 92 issues from being introduced, and resulted in fixes for 7 pre-existing issues, including an example with end-user measurable impact.

[9] As described in Section 3.2, Infer's checkers are modular and compositional: they can start from an arbitrary set of functions – for example, functions in the files touched by the code change – and transitively find and analyze their dependencies as needed.

References

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *SIGPLAN Not.* 49, 6 (2014), 259–269. doi:10.1145/2666356.2594299

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects (Lecture Notes in Computer Science, Vol. 4111)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 115–137. doi:10.1007/11804192_6

[3] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 459–465. doi:10.1007/978-3-642-20398-5_33

[4] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods (Lecture Notes in Computer Science, Vol. 9058)*, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 3–11. doi:10.1007/978-3-319-17524-9_1

[5] Malinda Dilhara, Haipeng Cai, and John Jenkins. 2018. Automated detection and repair of incompatible uses of runtime permissions in Android apps. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM, 67–71. doi:10.1145/3197231.3197255

[6] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. doi:10.1145/3338112

[7] Robert W. Floyd. 1962. Algorithm 97: Shortest path. *Commun. ACM* 5, 6 (1962), 345. doi:10.1145/367766.368168

[8] Ákos Hajdu, Matteo Marescotti, Thibault Suzanne, Ke Mao, Radu Grigore, Per Gustafsson, and Dino Distefano. 2022. InfERL: Scalable and Extensible Erlang Static Analysis. In *Proceedings of the 21st ACM SIGPLAN International Workshop on Erlang*. ACM, 33–39. doi:10.1145/3546186.3549929

[9] Chaoran Li, Xiao Chen, Ruoxi Sun, Minhui Xue, Sheng Wen, Muhammad Ejaz Ahmed, Seyit Camtepe, and Yang Xiang. 2022. Cross-language Android permission specification. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 772–783. doi:10.1145/3540250.3549142

[10] Tarek Mahmud, Meiru Che, and Guowei Yang. 2022. ACID: An API Compatibility Issue Detector for Android Apps. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. ACM, 1–5. doi:10.1145/3510454.3516854

[11] Ke Mao, Cons T Åhs, Sopot Cela, Dino Distefano, Nick Gardner, Radu Grigore, Per Gustafsson, Ákos Hajdu, Timotej Kapus, Matteo Marescotti, Gabriela Cunha Sampaio, and Thibault Suzanne. 2024. PrivacyCAT: Privacy-Aware Code Analysis at Scale. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 106–117. doi:10.1145/3639477.3639742

[12] Jan Midtgaard and Thomas P. Jensen. 2009. Control-flow analysis of function calls and returns by abstract interpretation. *SIGPLAN Not.* 44, 9 (2009), 287–298. doi:10.1145/1631687.1596592

[13] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification*. Lecture Notes in Computer Science, Vol. 12225. Springer, 225–252. doi:10.1007/978-3-030-53291-8_14

[14] Jordan Samhi, René Just, Tegawendé F. Bissyandé, Michael D. Ernst, and Jacques Klein. 2024. Call Graph Soundness in Android Static Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 945–957. doi:10.1145/3650212.3680333

[15] Jordan Samhi, Li Li, Tegawendé F. Bissyandé, and Jacques Klein. 2022. Difuzer: uncovering suspicious hidden sensitive operations in Android apps. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, 723–735. doi:10.1145/3510003.3510135

[16] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. doi:10.1137/0201010

[17] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static control-flow analysis of user-driven callbacks in Android applications. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE, 89–99. doi:10.5555/2818754.2818768

Received 2025-03-04; accepted 2025-04-25

Towards Bit-Level Dominance Preserving Quantization of Neural Classifiers

Dorra Ben Khalifa
Fédération ENAC ISAE-SUPAERO ONERA
Université de Toulouse
Toulouse, France
dorra.ben-khalifa@enac.fr

Matthieu Martel*
LAMPS Laboratory
Université de Perpignan Via Domitia
Perpignan, France
matthieu.martel@univ-perp.fr

Abstract

Quantization consists of replacing the original data types used to represent the weights of neural networks with less resource-intensive data types. While considerable research has focused on quantization, most existing methods that offer theoretical guarantees do so by providing error bounds on the difference between the original and reduced precision models. In this article, we introduce a new quantization technique that, rather than focusing on bounding errors, determines the minimum precision necessary to preserve class dominance, independent of any specific set of numerical formats. In other words, regardless of the exact scores for each class, our method guarantees that the class predicted by the original network remains unchanged after quantization.Our method is static and the proposed quantization holds for all the inputs. Technically, we leverage existing theorems that provide error bounds for dot products and formulate an optimization problem whose solution yields the required reduced precision. We also present experimental results to validate the effectiveness of our method.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**; **Artificial intelligence**; • **Software and its engineering** → **Compilers**; **Source code generation**; • **Mathematics of computing** → **Solvers**; • **Hardware** → **Power and energy**; **Power estimation and optimization**; • **Computer systems organization** → **Embedded and cyber-physical systems**.

Keywords: Neural Networks, Constraint Systems, Computer Arithmetic, Numerical Precision, Embedded Systems

ACM Reference Format:
Dorra Ben Khalifa and Matthieu Martel. 2025. Towards Bit-Level Dominance Preserving Quantization of Neural Classifiers. In *Proceedings of the 14th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '25), June 16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3735544.3735584

*Also with Numalis, Bureaux du Polygone, 265 Av. des États du Languedoc, Montpellier, France.

This work is licensed under a Creative Commons Attribution 4.0 International License.

SOAP '25, Seoul, Republic of Korea
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1922-6/25/06
https://doi.org/10.1145/3735544.3735584

1 Introduction

Quantization is a widely used technique for memory and computation reduction in neural networks (NNs) which consists of replacing the original data types used to represent the weights and perform the computations with less resource-intensive data types [13]. For example, one can use single precision floating-point numbers instead of double precision ones [1], or half precision instead of single precision numbers or fixed-point numbers instead of floating-point numbers, etc. In any case, the challenge is to reduce the memory and computational footprint of inference without compromising the quality of the classification.

Many studies have been carried out on quantization, including those that provide guarantees on the impact of data size reduction on the results produced by the networks, based on static [2, 3, 10] or dynamic [5, 8, 9] methods. However, existing methods that provide guarantees simply give error bounds on the difference between the score of each class in the original and reduced precision. This guarantee has at least two limitations: Firstly, it does not ensure that the dominant class (the class with the higher score) is preserved and, secondly, it does not ensure that this constraint allows one to obtain the best gains in terms of precision reduction.

In this article, we propose a new quantization technique that, rather than providing an error bound on the difference between the original and reduced precision network outputs, guarantees the preservation of dominance. Our claim is that dominance is the property of interest that must be maintained by quantization, regardless of the scores obtained for each class. In other words, it does not matter what the scores of each class are, as long as the dominant class remains the same. Note that we could refine this criterion by considering the n most dominant classes, or by adding an additional constraint on the minimal gap between the dominant and other classes.

We consider a post-training, uniform quantization (the precision is the same for all layers) in floating-point arithmetic. We take as input a trained neural network operating at some floating-point precision and find the minimum precision required to maintain dominance. The computed precision is

independent of the IEEE754 formats [1] and corresponds to the minimum number of bits required for the mantissa, assuming no overflow. This information makes it possible to determine the best format among those available in the machine.

Technically, we use results from [15] to bound the error on a dot product when we reduce the precision. We derive an optimization problem which we solve using a solver based on the interior point algorithm [4]. The solution of this problem gives the new precision. Experimental results on classical neural networks show that our method saves between 35% and 60% of memory while preserving dominance.

The rest of this article is organized as follows. Background material on error bounds in floating-point computations is introduced in Section 2. The quantization of dense layers is introduced in Section 3 and the resulting system of constraints is presented in Section 4. Experimental results are given in Section 5 and the quantization of other layers is discussed in Section 6. Finally, Section 7 concludes.

2 Notations and Background

This section outlines how errors introduced by reduced numerical precision propagate through computations. Let $\mathbb{Z}$ denote the set of integers, $\mathbb{R}$ the set of real numbers, and $\mathbb{F}_P$ the set of floating-point numbers in precision P as defined by the IEEE754 Standard [1, 12]. Here, the precision refers to the size of the significand, i.e. a floating-point number $x \in \mathbb{F}_P$ is a number of the form

$$x = \frac{s}{2^{P-1}} \cdot 2^e \ , \tag{1}$$

where $s \in \mathbb{Z}$ is the significand, P is the precision, i.e. the number of digits in the significand and e is the exponent. In this article, we focus on the optimization of the precision P and we do not specify the range of the exponent, assuming that no overflow may arise during the computations. For simplicity, denormalized numbers are omitted from consideration in this work.

All along this article, we use floating-point numbers in two different precisions, the original precision P_o and the reduced precision P_r, assuming that $P_r \leq P_o$. Intuitively, P_o represents the precision used during the training of the neural network, while P_r denotes the lower precision we aim to use during inference in place of P_o. We will also write P instead of P_o or P_r when equations apply to both precisions, or when the specific choice of precision is irrelevant. The round down (resp. round up) function is denoted $\llcorner . \lrcorner : \mathbb{F}_P \to \mathbb{Z}$ (resp. $\ulcorner . \urcorner : \mathbb{F}_P \to \mathbb{Z}$.) First, we recall the notion of unit in the last place [11].

Definition 1 (Unit in the last place)**.** *The* ulp *of a floating-point number $x \in \mathbb{F}_P$ is defined by*

$$\mathsf{ulp}_P(x) = \llcorner \log_2(x) \lrcorner - P + 1 \ . \tag{2}$$

Let $x,\ y \in \mathbb{F}_{P_o}$ be two floating-point numbers in the original precision and let u be the unit in the last place of 1 in the reduced precision P_r. Using rounding to the nearest we have

$$u = \llcorner \frac{1}{2} \cdot \mathsf{ulp}_{P_r}(1) \lrcorner \ . \tag{3}$$

The ulp of 1 in precision P is also called the machine epsilon in precision P. The numbers x and y of $\mathbb{F}_{P_o}$ are approximated in precision P_r. Following the standard model of floating-point arithmetic [6], this means that there exists $\delta, \delta' \in \mathbb{R}$, $|\delta| < u$ and $|\delta'| < u$ such that

$$\hat{x} = x \cdot (1 + \delta) \quad \text{and} \quad \hat{y} = y \cdot (1 + \delta') \ . \tag{4}$$

Note that the standard model of floating-point arithmetic has been designed to bound errors between exact computations in $\mathbb{R}$ and computations with floating-point numbers in some set $\mathbb{F}_P$. However $\mathbb{F}_P \subseteq \mathbb{R}$ and the model holds for errors between $\mathbb{F}_{P_o}$ and $\mathbb{F}_{P_r}$. In the next sections, we will consider trained neural networks that are supposed to work correctly in $\mathbb{F}_{P_o}$ which will be considered as exact or, in other words, as our arithmetic of reference, and the errors will be introduced by passing to $\mathbb{F}_{P_r}$. Next, for any operation $\diamond \in \{+, -, \times\}$, there exists δ'', with $|\delta''| < u$, such that

$$x \mathbin{\hat{\diamond}} y = (x \ \diamond \ y) \cdot (1 + \delta'') \ , \tag{5}$$

and, by combining equations (4) and (5), we can state that, for some δ, $|\delta| < u$,

$$\hat{x} \mathbin{\hat{\diamond}} \hat{y} = \big(\big(x \cdot (1 + \delta)\big) \ \diamond \ \big(y \cdot (1 + \delta')\big)\big) \cdot (1 + \delta'') \ . \tag{6}$$

Using equations (5) and (6) we obtain the following proposition, introduced by Rump [7, 15].

Proposition 1 (Error in dot product [15])**.** *Let $\mathbf{x},\ \mathbf{y} \in \mathbb{F}_P^n$ be two n-dimensional vectors and let $\mu = 2^u$. Assuming that $n \cdot \mu < 1$, the error on the dot product $\mathbf{x}^T \cdot \mathbf{y}$ can be bound by*

$$|\hat{\mathbf{x}}^T \mathbin{\hat{\cdot}} \hat{\mathbf{y}} \ - \ \mathbf{x}^T \cdot \mathbf{y}| \ \leq \ \frac{n \cdot \mu}{1 - n \cdot \mu} \cdot |\mathbf{x}^T| \cdot |\mathbf{y}|, \ . \tag{7}$$

Proposition 1 is used in the following sections to compute the errors due to the quantization of neural networks.

3 Quantization of Dense Layers

In this section, we formalize the concept of dominance-preserving quantization and demonstrate how to minimize inference-time precision while ensuring that dominance is preserved. We restrict our analysis to the case of single-layer, fully connected (also known as dense) neural networks without activation functions. Extensions to more complex architectures are discussed in Section 6.

Let $\mathbf{x} \in \mathbb{F}_{P_o}^n$ be a vector corresponding to the input of the fully connected (FC) layer, $\mathbf{x} = (x_1, \dots x_n)$, let $W \in \mathbb{F}_{P_o}^{m \times n}$ be a matrix of size $n \times m$ corresponding to the weights of the FC layer and whose coefficients are denoted w_{ij}, $1 \leq i \leq m$, $1 \leq j \leq n$, and let $\mathbf{y} \in \mathbb{F}_{P_o}^m$, $\mathbf{y} = (y_1, \dots y_m)$, be the vector corresponding to the result of the inference,

$$\mathbf{y} = W \cdot \mathbf{x} \ . \tag{8}$$

In particular, for all i, $1 \leq i \leq m$, we have

$$y_i = w_{i1} \cdot x_1 + w_{i2} \cdot x_2 + \ldots + w_{in} \cdot x_n \ . \tag{9}$$

Definition 2 (Dominance). *Following the notations of equations (8) and (9), a class i is dominant for some i, $1 \leq i \leq m$, and for some input x if and only if*

$$\forall j,\ 1 \leq j \leq m,\ j \neq i,\ y_i \geq y_j \ . \tag{10}$$

Our goal is to quantize fully connected layers. Let $\delta,\ \varepsilon \in \mathbb{R}$ be real numbers representing the errors introduced by the quantization of W and $\mathbf{x}$, respectively, and such that $|\delta| < u$ and $|\varepsilon| < u$, where u is the value defined in Equation (3). Following the standard model of floating-point arithmetic [6], the vector $\mathbf{x}$ and matrix W are approximated by $\hat{\mathbf{x}}$ and $\hat{W}$ such that

$$\hat{x}_i = x_i \cdot (1+\delta),\ 1 \leq i \leq n \ , \tag{11}$$

and

$$\hat{w}_{ij} = w_{ij} \cdot (1+\varepsilon),\ 1 \leq i \leq m,\ 1 \leq j \leq n \ , \tag{12}$$

and the quantized network computes

$$\hat{\mathbf{y}} = \hat{W} \cdot \hat{\mathbf{x}} \ . \tag{13}$$

Clearly, dominance is preserved under quantization if, for any input $\mathbf{x}$, the dominant class predicted by the quantized neural network matches that of the original network.

Definition 3 (Dominance preservation). *Following the notations introduced in equations (8) to (13), the dominance is preserved by quantization if and only if, for all input $\mathbf{x}$, the output $\mathbf{y}$ and the quantized output $\hat{\mathbf{y}}$ satisfy*

$$\forall j,\ 1 \leq j \leq m,\ j \neq i\ :\ y_i \geq y_j \iff \hat{y}_i \geq \hat{y}_j \ . \tag{14}$$

In the following, we introduce the main contribution of this article. This enables one to determine the minimum precision required for quantization in order to ensure the preservation of dominance.

Proposition 2 (Dominance preservation). *Let $W \in \mathbb{F}_{P_o}^{m \times n}$ be a matrix of size $n \times m$ corresponding to the weights of a fully connected layer, let $\mathbf{x} \in \mathbb{F}_{P_o}^{n}$ be a vector corresponding to the input, let $u = \mathsf{ulp}_{P_r}(1)$ be the machine epsilon and let $\mu = 2^u$. Finally, let*

$$\begin{gathered} y_i = \textstyle\sum_{k=1}^{n} W_{ik} \cdot x_k, \quad y_j = \sum_{k=1}^{n} W_{jk} \cdot x_k, \\ y'_i = \textstyle\sum_{k=1}^{n} |W_{ik}| \cdot |x_k|, \quad y'_j = \sum_{k=1}^{n} |W_{jk}| \cdot |x_k| \ . \end{gathered} \tag{15}$$

The dominance is preserved if for all j, $1 \leq j \leq m$, $j \neq i$,

$$\mu \ \leq \ \frac{1}{n} \cdot \frac{y_j - y_i}{y_j - y_i + y'_i + y'_j} \ . \tag{16}$$

Proof. First of all, we introduce the following notations.

$$\Gamma_i = \frac{n \cdot \mu}{1 - n \cdot \mu} \cdot y'_i, \quad \Gamma_j = \frac{n \cdot \mu}{1 - n \cdot \mu} \cdot y'_j \ . \tag{17}$$

We want to have $\hat{y}_j \leq \hat{y}_i$. Accordingly to Equation (7), we know that

$$-\Gamma_i \leq \hat{y}_i - y_i \leq \Gamma_i \quad \text{and} \quad -\Gamma_j \leq \hat{y}_j - y_j \leq \Gamma_j \ . \tag{18}$$

So, in particular, using equations (10) and (18), we must have

$$y_j - \Gamma_j \ \leq \ \hat{y}_j \ \leq \ \hat{y}_i \ \leq \ y_i + \Gamma_i \ , \tag{19}$$

and we deduce that we must have $y_j - y_i \ \leq \ \Gamma_i + \Gamma_j$. Using the notations of equations (15) and (17), this means that the following equation must hold.

$$y_j - y_i \ \leq \ \frac{n \cdot \mu}{1 - n \cdot \mu} \cdot \left(y'_i + y'_j\right) \ , \tag{20}$$

or, equivalently, with the assumption $n \cdot \mu < 1$ of Proposition 1,

$$(y_j - y_i) \cdot (1 - n \cdot \mu) \ \leq \ (n \cdot \mu) \cdot (y'_i + y'_j) \ . \tag{21}$$

In turn, Equation (21) can be rewritten as

$$y_j - y_i \leq \ (n \cdot \mu) \cdot (y_j - y_i + y'_i + y'_j) \ . \tag{22}$$

Next, since $y_j - y_i \leq 0$,

$$\frac{1}{\mu} \geq n \cdot \frac{y_j - y_i + y'_i + y'_j}{y_j - y_i} \ , \tag{23}$$

and, by a last rewriting, we obtain Equation (16). □

4 Constraint Generation

In this section, we use Proposition 2 to formalize the dominance preserving quantization problem as a nonlinear optimization problem. We want to minimize the precision P_r while preserving the dominant class. By equations (2) and (3), the machine epsilon in precision P_r is

$$u = \llcorner \frac{-P_r + 1}{2} \lrcorner \ . \tag{24}$$

Then, u being contravariant in function of P_r, we have to maximize u, or, equivalently $\mu = 2^u$ (μ being covariant in function of u), under the constraint that the dominance is preserved. According to Proposition 2, and by developing Equation (16), if class i is the dominant one, we must ensure that, for all j, $1 \leq j \leq m$, $j \neq i$, the following condition holds

$$\mu \ \leq \ \frac{1}{n} \cdot \frac{\sum_{k=1}^{n} W_{jk} \cdot x_k - \sum_{k=1}^{n} W_{ik} \cdot x_k}{\sum_{k=1}^{n} W_{jk} \cdot x_k - \sum_{k=1}^{n} W_{ik} \cdot x_k + \sum_{k=1}^{n} |W_{ik}| \cdot |x_k| + \sum_{k=1}^{n} |W_{jk}| \cdot |x_k|} \ . \tag{25}$$

The optimization problem is then formalized as follows.

- The unknown is μ which we want to maximize. This value μ is used to compute the reduced precision

$$P_r = 1 - \llcorner \log_2 \mu \lrcorner \ , \tag{26}$$

- The dimension of the input vector is n which is statically known, i.e. n is a constant for the constraint system,
- The W_{ik} and W_{jk}, $1 \leq i, j \leq m$, $1 \leq k \leq n$ are the weights of the FC layer which are statically known. Again, these are constants for the constraint system,

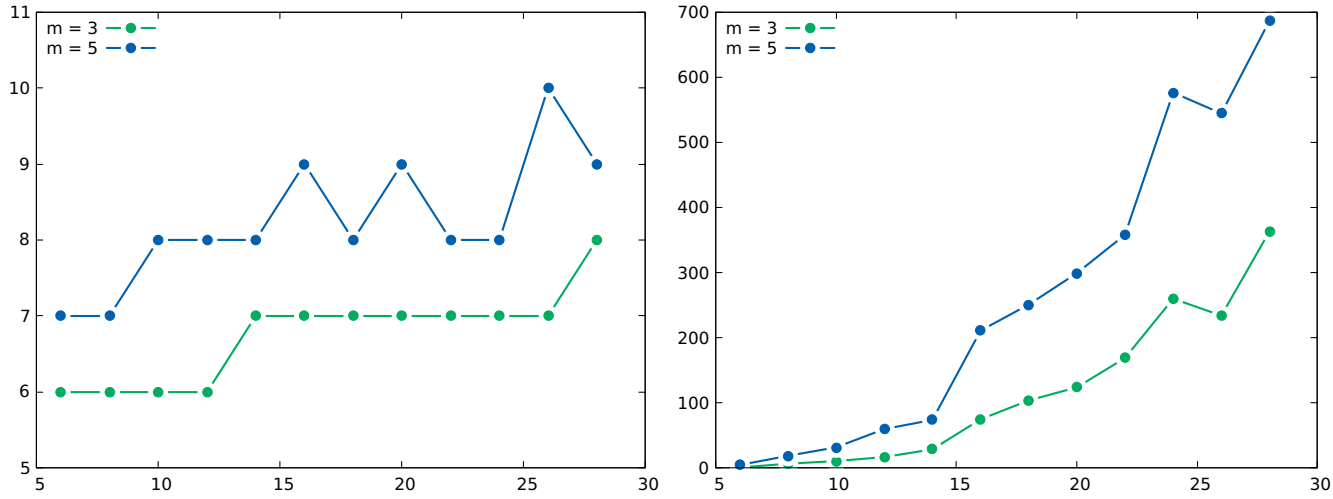


Figure 1. Left: Minimal dominance preserving precision P_r found by our algorithm for a FC layer with $n \times n$ inputs (n on the x-axis) and m outputs. **Right:** Time in seconds needed to solve the optimization problem in function of m and n.

- x_k, $1 \leq k \leq n$ are the elements of the input vector assumed to belong to a range $R = [\underline{x}, \overline{x}]$ given by the user, i.e. for all k, $1 \leq k \leq n$, we have the constraint $\underline{x} \leq x_k \leq \overline{x}$. Typically, for images, $R = [0, 255]$ or $R = [0, 1]$.

Then, we solve the following problem.

$$\mu_{opt} = \max_{\substack{1 \leq i \leq m \\ 1 \leq j \leq m,\ j \neq i \\ \underline{x} \leq x_k \leq \overline{x},\ 1 \leq k \leq n}} \frac{1}{n} \cdot \frac{\sum_{k=1}^{n} W_{jk} \cdot x_k - \sum_{k=1}^{n} W_{ik} \cdot x_k}{\begin{array}{r} \sum_{k=1}^{n} W_{jk} \cdot x_k \\ - \sum_{k=1}^{n} W_{ik} \cdot x_k \\ + \sum_{k=1}^{n} |W_{ik}| \cdot |x_k| \\ + \sum_{k=1}^{n} |W_{jk}| \cdot |x_k| \end{array}} \tag{27}$$

Note that by allowing $\mathbf{x}$ to vary in the range R of the inputs, we solve the problem for all possible inputs of the network. Note also that the range of $\mathbf{x}$ can be reduced to some $R' \subseteq R$ if the user intends to quantize the network for a specific subset of inputs.

5 Experimental Results

In this section, we present experimental results on the quantization of common neural classifiers. To solve the optimization problem of Equation (27), we employ a trust-region method [14, 16], namely the `trust-constr` minimization method of `scipy`.

Our first experiment evaluates how our method performs in quantizing common neural networks. We considered the following datasets for this purpose:

- MNIST: A dataset of 28×28 grayscale images containing handwritten digits,
- DIABETES: A dataset from the National Institute of Diabetes and Digestive and Kidney Diseases, used for predicting whether a patient has diabetes based on medical measurements,
- CIFAR10: A standard benchmark for $32 \times 32 \times 3$ image classification tasks,
- CARDS: A dataset containing $224 \times 224 \times 3$ card playing images (53 classes),
- RICE: A dataset containing $224 \times 224 \times 3$ images of five rice varieties,
- WASTE: A dataset containing $224 \times 224 \times 3$ images of recyclable or non-recyclable waste.

Table 1. Reduced precision P_r found by our method for various neural network classifiers and saved memory.

Network	Layer Shape	P_r	Saved Memory	
MNIST	64×10	**9**	1 120 Bytes	60.9%
DIABETES	8×32	**12**	352 Bytes	47.8%
CIFAR10	64×10	**13**	800 Bytes	43.4%
CARDS	16×53	**15**	848 Bytes	34.7%
RICE	256×5	**13**	1 600 Bytes	43.4%
WASTE	64×16	**15**	1024 Bytes	34.7%

Note that, for all the networks, the input images are normalized between 0 and 1 before being passed to the networks (i.e. $\underline{x} = 0$ and $\overline{x} = 1$ using the notations of Section 4.) For each of the pre-trained networks used in this experiment, we quantize the last (dense) layer with the exception of DIABETES and WASTE, for which we have taken the penultimate layer, the last being used to calculate a score.

The results are shown in Table 1. For each network, we report the shape of the dense layer and the reduced precision P_r computed by our method. We also indicate how many bytes are saved compared to the original network in single precision ($P_o = 23$) and the percentage of saved memory.

Across all tested networks, the reduced precision P_r ranges between 9 and 15 bits, with memory savings consistently falling between 34.7% and 60.9%. For instance, the MNIST classifier, with a dense layer of shape 64×10, achieves a reduced precision of $P_r = 9$, resulting in a memory saving of 1120 bytes, equivalent to a 60.9% reduction. Similarly, the DIABETES model, though smaller in size with a layer shape of 8×32, benefits from a reduction to $P_r = 12$, saving 352 bytes, or 47.8%.

Our second experiment investigates the scalability of our method. We trained neural networks on the MNIST dataset with $m = 3$ and $m = 5$ different classes and for images with sizes ranging from 5×5 to 28×28. The results are shown in Figure 1. The graph on the left shows the reduced precision P_r obtained for each set of parameters. We observe that the gains are important, compared to the original precision $P_o = 23$[1]. The graph on the right of Figure 1 illustrates the execution time of the method as a function of m and n. Note that the size of the inputs is quadratic in n. In all cases, the method completes in just a few minutes.

6 Quantization of More General Networks

In this section, we discuss how to extend the technique introduced in Section 4 to other types of layers beyond dense layers. While further work is required to address the most common layer types encountered in neural classifiers, we strongly believe that our approach is generalizable. In particular, we highlight the following elements:

- **Activation function:** Most activation functions are monotone. For example, `ReLU`, `tanh`, `sigmoid`, `Leaky ReLU`, `ELU` (Exponential Linear Unit) are all increasing. As a result, they do not change the dominance and can simply be ignored by our method as they do not need to be quantized.
- **Monotonic layers:** Several layers also are monotonic. For example, the `max pool`, `mean pool` and `up sampling` layers are increasing. As for the monotonic activation functions, they can be left as they are in the quantified network.
- **Convolutional Layers:** Convolutional layers perform a sequence of operation of the form

$$y_{ij} = \sum_{m=0}^{M} \sum_{n=0}^{n} K_{m,n} \cdot x_{i+m,j+n} \ , \tag{28}$$

for some kernel K of size $M \times N$. This corresponds to a sequence of multiplications and additions that can be bound as in Equation (7). We strongly believe that convolutional layers can be treated in a similar way to dense layers in sections 3 and 4.

- **Multi-layer networks:** The composition of many layers also needs to be addressed. Nevertheless, the composition of monotone functions is monotone and this should help us a lot in solving the problem. Then, if we omit monotonic functions, the main difficulty is the composition of convolutional and dense layers. We believe that this case reduces to a sum of elementary products, i.e. a computation of the form

$$y = \sum_{i \in I} \alpha_i \cdot \beta_i \ , \tag{29}$$

for some set I of indexes and some terms α_i and β_i coming from the entries and parameters of the network. Again, this kind of computation can be bound as in Equation (7) and treated in a similar way to dense layers in sections 3 and 4.

7 Conclusion

In this article, we introduced a novel quantization method that ensures the dominant class is preserved in the quantized network compared to the original network. Our method is static, relying on the network's weights rather than specific inputs. Experimental results demonstrate that we can achieve significant precision reductions (over 50%) while guaranteeing that the dominance is preserved.

We use a theorem of Rump [15] that provides bounds on the round-off errors in dot products, leading to a nonlinear optimization problem that we solve using a trust-region method [14]. Currently, our method is applicable to dense layers; however, as discussed in Section 6, it appears feasible to extend it to various other layer types.

This article lays the foundations for a new method, and much work remains to be done to extend it to real-world use cases. More layers need to be addressed, as discussed in section 6, and the case of multi-layer networks must also be considered . We also aim at improving the resolution of the constraint system. First, we could explore alternative solvers to identify the most efficient one. Second, we might relax the system of constraints to make it solvable using simpler and faster methods.

[1] Single precision numbers have 23 bits of significand.

References

[1] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

[2] Dorra Ben Khalifa and Matthieu Martel. 2024. Efficient Implementation of Neural Networks Usual Layers on Fixed-Point Architectures. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES*. ACM, 12–22. https://doi.org/10.1145/3652032.3657578

[3] Hanane Benmaghnia, Matthieu Martel, and Yassamine Seladji. 2024. Code Generation for Neural Networks Based on Fixed-point Arithmetic. *ACM Trans. Embed. Comput. Syst.* 23, 5 (2024), 68:1–68:28. https://doi.org/10.1145/3563945

[4] Richard H. Byrd, Mary E. Hribar, and Jorge Nocedal. 1999. An interior point algorithm for large-scale nonlinear programming. *SIAM Journal on Optimization* 9, 4 (Sept. 1999), 877–900. https://doi.org/10.1137/S1052623497325107

[5] Quentin Ferro, Stef Graillat, Thibault Hilaire, Fabienne Jézéquel, and Basile Lewandowski. 2022. Neural Network Precision Tuning Using Stochastic Arithmetic. In *Software Verification and Formal Methods for ML-Enabled Autonomous Systems - NSV (Lecture Notes in Computer Science, Vol. 13466)*. Springer, 164–186. https://doi.org/10.1007/978-3-031-21222-2_10

[6] Nicholas J. Higham. 2002. *Accuracy and stability of numerical algorithms, Second Edition*. SIAM. https://doi.org/10.1137/1.9780898718027

[7] Claude-Pierre Jeannerod and Siegfried M. Rump. 2013. Improved Error Bounds for Inner Products in Floating-Point Arithmetic. *SIAM J. Matrix Anal. Appl.* 34, 2 (2013), 338–344. https://doi.org/10.1137/120894488

[8] Christoph Quirin Lauter and Anastasia Volkova. 2020. A Framework for Semi-Automatic Precision and Accuracy Analysis for Fast and Rigorous Deep Learning. In *27th IEEE Symposium on Computer Arithmetic, ARITH*. IEEE, 103–110. https://doi.org/10.1109/ARITH48897.2020.00023

[9] Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2023. Training with Mixed-Precision Floating-Point Assignments. *Trans. Mach. Learn. Res.* 2023 (2023). https://openreview.net/forum?id=ZoXi7n54OB

[10] Debasmita Lohar, Clothilde Jeangoudoux, Anastasia Volkova, and Eva Darulova. 2023. Sound Mixed Fixed-Point Quantization of Neural Networks. *ACM Trans. Embed. Comput. Syst.* 22, 5s (2023), 136:1–136:26. https://doi.org/10.1145/3609118

[11] Jean-Michel Muller. 2005. *On the definition of ulp(x)*. Research Report RR-5504, LIP RR-2005-09. INRIA, LIP. 16 pages. https://inria.hal.science/inria-00070503

[12] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. 2010. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston. https://doi.org/10.1007/978-0-8176-4705-6

[13] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. 2021. A White Paper on Neural Network Quantization. *CoRR* abs/2106.08295 (2021). arXiv:2106.08295 https://arxiv.org/abs/2106.08295

[14] Andrew R. Conn, Nicholas I. M. Gould, and Philippe L. Toint. 2000. *Trust Region Methods*. SIAM. https://doi.org/10.1137/1.9780898719857

[15] Siegfried M. Rump. 2012. Error estimation of floating-point summation and dot product. *BIT* 52, 1 (March 2012), 201–220. https://doi.org/10.1007/s10543-011-0342-4

[16] Ya-Xiang Yuan. 2015. Recent advances in trust region algorithms. *Math. Program.* 151, 1 (June 2015), 249–281. https://doi.org/10.1007/S10107-015-0893-2

Received 2025-03-03; accepted 2025-04-25

Universal High-Performance CFL-Reachability via Matrix Multiplication

Ilia Muravev
i.v.muraviev@spbu.ru
Saint Petersburg State University
St. Petersburg, Russia

Semyon Grigorev
s.v.grigoriev@spbu.ru
Saint Petersburg State University
St. Petersburg, Russia

Abstract

Context-free language (CFL) reachability is a fundamental computational framework for formulating key static analyses (e.g., alias analysis, value-flow analysis, and points-to analysis) as well as some other graph analysis problems. Achieving high performance in universal CFL-reachability solvers remains a significant challenge. Specialized tools such as Pearl and Gigascale are optimized for specific CFLs but lack general applicability, whereas existing universal CFL-reachability solvers often do not scale well in important cases. In particular, prior efforts to leverage high-performance linear algebra operations in universal CFL-reachability solvers produced a matrix-based solver, MatrixCFPQ, that excels at performing common navigational queries on RDF graphs (which are unrelated to program analysis) but is inefficient when it comes to modeling static analyses.

In this work, we introduce FastMatrixCFPQ, a universal matrix-based CFL-reachability solver that overcomes the limitations of MatrixCFPQ by leveraging the properties of the CFL-semiring, common patterns in context-free grammars, and the features of the SuiteSparse:GraphBLAS sparse linear algebra library. Our experimental results demonstrate that FastMatrixCFPQ outperforms the state-of-the-art universal CFL-reachability solvers across five client analyses—often by orders of magnitude—and, in many cases, even surpasses the speed of specialized solvers designed for specific CFLs.

CCS Concepts: • **Software and its engineering** → *Software verification and validation*; *Automated static analysis*; • **Theory of computation** → *Theory and algorithms for application domains.*

Keywords: CFL-Reachability, Static Analysis, GraphBLAS

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOAP '25, Seoul, Republic of Korea
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1922-6/25/06
https://doi.org/10.1145/3735544.3735585

ACM Reference Format:
Ilia Muravev and Semyon Grigorev. 2025. Universal High-Performance CFL-Reachability via Matrix Multiplication. In *Proceedings of the 14th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '25), June 16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3735544.3735585

1 Introduction

Static analyses (i.e., analyses performed without program execution) are commonly used to infer program properties such as security vulnerabilities, possible bugs, and potential optimizations. These analyses typically rely on constructing a program model in the form of an edge-labeled graph G, where nodes represent source code entities (e.g., variables or variable–instruction pairs) and edges capture direct relationships between these entities (e.g., control-flow or direct data-dependence). Questions about the program are then reduced to graph analysis problems.

Language reachability. As shown by Thomas Reps [20], one prominent graph analysis problem that many static analyses reduce to is the language reachability problem. Essentially, given an edge-labeled graph G and a language $\mathcal{L}$, the problem is to find all pairs of vertices $\langle u, v \rangle$ that are connected by a path whose concatenated edge labels form a string in $\mathcal{L}$. The language $\mathcal{L}$ serves to filter out paths that cannot occur during normal program execution.

Context-free language reachability. A common class of languages in language-reachability analyses is context-free languages (CFLs). The ability of CFLs to model recursive structures has various applications in static analyses.

For example, CFL-reachability is used to model *calling-context-sensitive*[1] interprocedural analyses [22] (a.k.a. whole-program analyses), where a CFL of properly balanced parenthesis strings (see Fig. 1) ensures that function calls (represented by opening parentheses) and function returns (represented by closing parentheses) match in a valid execution path (see Fig. 2 for an example). Similarly, in *field-sensitive* analyses [27], another CFL ensures that field writes and field reads match when modeling the information flow through composite object fields.

[1] It might sound paradoxical that a context-free language makes the analysis context-sensitive, but this is just a naming collision. "Context sensitivity" refers to calling contexts, and the CFL simulates the call stack.

$$S \to S\ S \mid (_i\ S\ )_i \mid \varepsilon$$

$$\begin{aligned} S &\to S\ S \mid C_i\ SR_i \mid \varepsilon \\ SR_i &\to S\ R_i \\ C_i &\to (_i \\ R_i &\to)_i \end{aligned}$$

Figure 1. *(Left):* A context-free grammar (CFG) for context-sensitive value-flow analysis. S is the start non-terminal. $(_i$ and $)_i$ denote, respectively, the call and the return operations for the i-th call site. Production rules involving index i are repeated for all $i \in \{1, \ldots, k\}$, where k is the number of call sites. *(Right):* An equivalent CFG in weak Chomsky normal form (WCNF). See Section 2 for definitions.

```
fun id(v1)
  return v1

fun f(v2, v3)
  v4 ← id(v2) // call site 1
  id(v3)      // call site 2
```

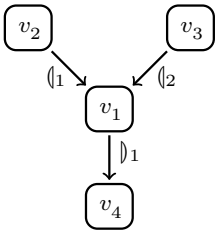


Figure 2. *(Left):* An analyzed program. *(Right):* A graph for context-sensitive value-flow analysis. The path $v_2 \rightsquigarrow v_4$ has a properly balanced label $(_1)_1$, witnessing a true value flow from v_2 to v_4. In contrast, the path $v_3 \rightsquigarrow v_4$ has an imbalanced label $(_2)_1$ and hence does not correspond to any interprocedurally valid execution.

Moreover, CFL-reachability is also used as a building block in various algorithms [4, 10, 26] that approximate *interleaved Dyck reachability*—a well-established undecidable formulation of static analyses that seek to be both context-sensitive and field-sensitive [21].

CFL-reachability solvers. Existing CFL-reachability solvers can be categorized as either specialized or universal. Specialized solvers, such as Gigascale [8] and Pearl [25], are tailored for specific CFLs. Gigascale is designed for Java field-sensitive points-to analysis [27], while Pearl is implemented for context-sensitive value-flow analysis [28] and field-sensitive alias analysis [34] for C/C++ programs. These tools achieve high performance by exploiting the characteristics of their targeted CFLs.

In contrast, universal CFL-reachability solvers, such as Graspan [32] and MatrixCFPQ [2], aim to provide a general solution applicable to any CFL. While this generality is appealing for flexibility and broad applicability, the existing universal solvers often suffer from significant performance limitations.

Linear algebra. One promising approach to accelerating CFL-reachability solvers is to leverage linear algebra operations on sparse matrices, which are highly optimized for modern hardware [12] and can express a wide range of graph algorithms [5, 6, 11]. In particular, the CFL-reachability problem has been reduced to linear algebra operations by Rustam Azimov in the MatrixCFPQ algorithm [2]. Although MatrixCFPQ performs well on RDF graph analysis problems [33] (a CFL-reachability application distinct from program analysis), it becomes impractical when the analyzed graph contains paths with moderately deep *derivation trees* or when the context-free grammar (CFG) is large. These limitations are especially problematic for field-sensitive (resp., context-sensitive) static analyses, which require CFGs with distinct production rules for every field (resp., call site).

Our contribution. We address the poor scalability of MatrixCFPQ with respect to the depth of derivation trees by rewriting the algorithm to operate on matrices that capture only the most recent graph updates, adapting matrix multiplications and additions to efficiently handle these sparse graph update matrices, and eliminating trivial operations. Additionally, we improve scalability with respect to grammar size by utilizing block matrices. Finally, we enhance performance for certain CFLs by transforming their CFGs.

An experimental evaluation of our FastMatrixCFPQ solver on five client analyses demonstrates that it is significantly faster than the state-of-the-art universal CFL-reachability solvers and, in many cases, even outperforms specialized solvers implemented for specific CFLs.

2 Preliminaries

CFL representation. A *context-free grammar* (CFG) is a tuple $\mathcal{G} = \langle \mathcal{N}, \Sigma, \mathcal{P}, \mathcal{S} \rangle$ where $\mathcal{N}$ is a finite set of non-terminals, Σ is a finite set of terminals, $\mathcal{P}$ is a finite set of production rules of the form $A \to \alpha$ (with $A \in \mathcal{N}$ and $\alpha \in (\mathcal{N} \cup \Sigma)^*$), and $\mathcal{S} \in \mathcal{N}$ is the start symbol.

Given a CFG $\mathcal{G} = \langle \mathcal{N}, \Sigma, \mathcal{P}, \mathcal{S} \rangle$, we say that the string $\omega_1 \in (\mathcal{N} \cup \Sigma)^*$ *directly yields* the string $\omega_2 \in (\mathcal{N} \cup \Sigma)^*$ and write $\omega_1 \Rightarrow \omega_2$ if there is a production $(A \to \alpha) \in \mathcal{P}$ and strings $\beta, \gamma \in (\mathcal{N} \cup \Sigma)^*$ such that $\omega_1 = \beta A \gamma$ and $\omega_2 = \beta \alpha \gamma$.

If there exists a finite sequence of strings $\omega_1, \omega_2, \ldots, \omega_k \in (\mathcal{N} \cup \Sigma)^*$ with $k \geq 1$ such that $\omega_1 \Rightarrow \omega_2, \ldots, \omega_{k-1} \Rightarrow \omega_k$, then we say that ω_k is *derived* from ω_1 and write $\omega_1 \overset{*}{\Rightarrow} \omega_k$.

The *context-free language* (CFL) generated by the CFG $\mathcal{G}$ is defined as

$$\mathcal{L}(\mathcal{G}) = \{\omega \in \Sigma^* \mid \mathcal{S} \overset{*}{\Rightarrow} \omega\}.$$

Language reachability. Consider an edge-labeled directed graph $G = \langle V, E, \Sigma \rangle$, where V is the set of nodes, Σ is the set of edge labels, and $E \subseteq V \times \Sigma \times V$ is the set of edges. A path $\pi : a \rightsquigarrow b$ in G is a sequence of edges $\langle u_1, l_1, v_1 \rangle, \ldots, \langle u_k, l_k, v_k \rangle \in E$ such that $u_1 = a$, $v_k = b$, and $\forall i < k : v_i = u_{i+1}$. The label of π is defined as $\lambda(\pi) = l_1 l_2 \ldots l_k$.

Given a language $\mathcal{L}$ and a graph G, the all-pairs relation semantics $\mathcal{L}$-reachability problem (CFL-reachability when $\mathcal{L}$ is a CFL) seeks to find the *$\mathcal{L}$-reachability relation* $\mathcal{R}$ defined

as follows:

$$\mathcal{R} = \{\langle a, b\rangle \in V^2 \mid \exists \pi : a \rightsquigarrow b, \lambda(\pi) \in \mathcal{L}\}.$$

CFL-semiring. A CFG $\mathcal{G} = \langle \mathcal{N}, \Sigma, \mathcal{P}, \mathcal{S}\rangle$ is said to be in *weak Chomsky normal form* (WCNF) if $\mathcal{P} \subseteq \{A \to b \mid A \in \mathcal{N}, b \in \Sigma\} \cup \{A \to \varepsilon \mid A \in \mathcal{N}\} \cup \{A \to B\ C \mid A, B, C \in \mathcal{N}\}$, where ε denotes the empty string (see Fig. 1 for an example).

Given a CFG $\mathcal{G} = \langle \mathcal{N}, \Sigma, \mathcal{P}, \mathcal{S}\rangle$ in WCNF, the *CFL-semiring* is a non-associative semiring $R_{\mathcal{G}} = \langle 2^{\mathcal{N}}, \cup, \otimes_{\mathcal{G}}, \varnothing\rangle$, where $2^{\mathcal{N}}$ is the domain, the empty set $\varnothing$ acts as zero, set union $\cup$ acts as addition, and $\otimes_{\mathcal{G}}$ acts as multiplication defined as follows:

$$X \otimes_{\mathcal{G}} Y = \{A \in \mathcal{N} \mid \exists (A \to B\ C) \in \mathcal{P} : B \in X, C \in Y\}.$$

3 Baseline Implementation

Our work builds upon the MatrixCFPQ algorithm (Algorithm 1) devised by Azimov [2]. This CFL-reachability algorithm incrementally constructs a matrix $\mathbf{T} \in R_{\mathcal{G}}^{|V|\times|V|}$ over the CFL-semiring such that

$$\mathbf{T}_{i,j} = \{A \in \mathcal{N} \mid \exists \pi : v_i \rightsquigarrow v_j, A \overset{*}{\Rightarrow} \lambda(\pi)\}.$$

Algorithm 1 first converts the input CFG $\mathcal{G}$ to an equivalent CFG $\mathcal{G}'$ in WCNF and then computes a transitive closure similarly to the classic Floyd-Warshall algorithm for unconstrained reachability, with the key difference that the CFL-semiring $R_{\mathcal{G}'}$ is used for the matrix operations.

For example, consider the CFG $\mathcal{G}$ shown in Fig. 1 and the graph G shown in Fig. 2, where $\mathcal{G}$ generates a CFL $\mathcal{L}(\mathcal{G})$ of well-nested labeled parenthesis strings. Let $\mathbf{T}^{(p)}$ denote the state of the matrix $\mathbf{T}$ immediately before the p-th iteration of the **while** loop. After the initialization (lines 1–9) is completed, we have:

$$\mathbf{T}^{(1)} = \begin{bmatrix} \{S\} & & & \{R_1\} \\ \{C_1\} & \{S\} & & \\ \{C_2\} & & \{S\} & \\ & & & \{S\} \end{bmatrix}.$$

On the first iteration of the **while** loop, the value of $\mathbf{T}_{1,4}$ is updated to include SR_1, since $S \in \mathbf{T}_{1,1}$, $R_1 \in \mathbf{T}_{1,4}$, and the production rule $(SR_i \to S\ R_i)$ is in $\mathcal{P}$. Thus,

$$\mathbf{T}^{(2)} = \begin{bmatrix} \{S\} & & & \{SR_1, R_1\} \\ \{C_1\} & \{S\} & & \\ \{C_2\} & & \{S\} & \\ & & & \{S\} \end{bmatrix}.$$

On the second iteration of the **while** loop, the value of $\mathbf{T}_{2,4}$ is updated to include S, since $C_1 \in \mathbf{T}_{2,1}$, $SR_1 \in \mathbf{T}_{1,4}$, and the production rule $(S \to C_i\ SR_i)$ is in $\mathcal{P}$. Thus,

$$\mathbf{T}^{(3)} = \begin{bmatrix} \{S\} & & & \{SR_1, R_1\} \\ \{C_1\} & \{S\} & & \{S\} \\ \{C_2\} & & \{S\} & \\ & & & \{S\} \end{bmatrix}.$$

On the third iteration, no further changes occur (i.e., a fixed point is reached), so $\mathbf{T} = \mathbf{T}^{(3)}$. Notice that $S \in \mathbf{T}_{2,4}$ because

Algorithm 1: Original MatrixCFPQ algorithm [2].

Require: $\mathcal{G}$ is a CFG, $G = \langle V, E, \Sigma\rangle$ is a graph with vertices enumerated as $V = \{v_1, \ldots, v_n\}$.
Ensure: $\mathcal{R}$ is the $\mathcal{L}(\mathcal{G})$-reachability relation.

1 $\mathcal{G}' \leftarrow$ CFG in WCNF equivalent to CFG $\mathcal{G}$;
2 $\langle \mathcal{N}, \Sigma, \mathcal{P}, \mathcal{S}\rangle \leftarrow \mathcal{G}'$;
3 $\mathbf{T} \leftarrow \varnothing^{n\times n}$; `// zero matrix over` $R_{\mathcal{G}'}$
4 **foreach** $(v_i, x, v_j) \in E$ **do**
5 $\quad \mathbf{T}_{i,j} \leftarrow \mathbf{T}_{i,j} \cup \{A \mid (A \to x) \in \mathcal{P}\}$;
6 **end**
7 **foreach** $v_i \in V$ **do**
8 $\quad \mathbf{T}_{i,i} \leftarrow \mathbf{T}_{i,i} \cup \{A \mid (A \to \varepsilon) \in \mathcal{P}\}$;
9 **end**
10 **while** *matrix* $\mathbf{T}$ *is changing* **do**
11 $\quad$ `// matrix operations over` $R_{\mathcal{G}'}$ `semiring`
12 $\quad \mathbf{T} \leftarrow \mathbf{T} \cup (\mathbf{T} \cdot \mathbf{T})$;
13 **end**
14 $\mathcal{R} \leftarrow \{\langle v_i, v_j\rangle \in V^2 \mid \mathcal{S} \in \mathbf{T}_{i,j}\}$;

there exists a path $\pi : v_2 \rightsquigarrow v_4$ such that $\lambda(\pi) = (\!|_1|\!)_1 \in \mathcal{L}(\mathcal{G})$, whereas $S \notin \mathbf{T}_{3,4}$ because there is no path $\pi : v_3 \rightsquigarrow v_4$ such that $\lambda(\pi) \in \mathcal{L}(\mathcal{G})$.

To improve performance, implementations of Algorithm 1 do not store the matrix $\mathbf{T}$ directly. Instead, they decompose it into $|\mathcal{N}|$ Boolean matrices. For every non-terminal $A \in \mathcal{N}$, a Boolean matrix $\mathbf{T}^A \in \{0,1\}^{|V|\times|V|}$ is maintained such that $\mathbf{T}^A_{i,j}$ is true if $A \in \mathbf{T}_{i,j}$. This representation allows one to compute the matrix product $\mathbf{T} \cdot \mathbf{T}$ over the CFL-semiring by performing $|\mathcal{P}|$ sparse Boolean matrix multiplications, as shown by Leslie Valiant [31]:

$$\forall A \in \mathcal{N} : (\mathbf{T} \cdot \mathbf{T})^A = \bigvee \left\{\mathbf{T}^B \cdot \mathbf{T}^C \mid (A \to B\ C) \in \mathcal{P}\right\}. \quad (1)$$

4 Our Approach

In this section, we describe the modifications we made to Algorithm 1 to improve performance, along with the motivation behind each enhancement.

Handling deep derivations. As noted by Azimov [2], the most intensive computation in Algorithm 1 is the matrix multiplication on line 12 executed in a **while** loop. When Algorithm 1 is applied to RDF graph analysis problems on which it was originally evaluated, the **while** loop typically converges after only a few iterations. However, in the case of program analysis tasks, tens or even hundreds of iterations are often required for convergence. This is because, unlike RDF graphs, program graphs typically contain at least one valid path π whose label $\lambda(\pi)$ necessitates a deep derivation.

To avoid repeating expensive computations in every iteration, we leverage the distributivity of $\otimes_{\mathcal{G}}$ over $\cup$, which allows us to replace $\mathbf{T} \leftarrow \mathbf{T} \cup (\mathbf{T} \cdot \mathbf{T})$ on line 12 with:

$$\mathbf{T} \leftarrow \mathbf{T} \cup (\mathbf{T}_{old} \cdot \mathbf{T}_{old}) \cup (\mathbf{T}_{old} \cdot \Delta\mathbf{T}) \cup (\Delta\mathbf{T} \cdot \mathbf{T}_{old}) \cup (\Delta\mathbf{T} \cdot \Delta\mathbf{T}),$$

where $\Delta\mathbf{T} = \mathbf{T} \setminus \mathbf{T}_{old}$ and $\mathbf{T}_{old}$ represents the value of $\mathbf{T}$ during the previous iteration (on the first iteration, $\mathbf{T}_{old} = \varnothing^{n \times n}$). Observe that by construction $(\mathbf{T}_{old} \cdot \mathbf{T}_{old}) \subseteq \mathbf{T}$. We can thus omit $(\mathbf{T}_{old} \cdot \mathbf{T}_{old})$ from the element-wise union and finally rewrite the update $\mathbf{T} \leftarrow \mathbf{T} \cup (\mathbf{T} \cdot \mathbf{T})$ as follows:

$$\Delta\mathbf{T} \leftarrow ((\mathbf{T}_{old} \cdot \Delta\mathbf{T}) \cup (\Delta\mathbf{T} \cdot \mathbf{T}_{old}) \cup (\Delta\mathbf{T} \cdot \Delta\mathbf{T})) \setminus \mathbf{T};$$
$$\mathbf{T} \leftarrow \mathbf{T} \cup \Delta\mathbf{T}.$$

This modification considerably speeds up the algorithm by ensuring that in every multiplication at least one operand is $\Delta\mathbf{T}$, which is usually much sparser than $\mathbf{T}$.

Skipping trivial operations. The sparse linear algebra library SuiteSparse:GraphBLAS [7], which is used to implement MatrixCFPQ and FastMatrixCFPQ, spends a non-negligible amount of time on trivial operations involving zero Boolean matrices. Our solver circumvents this overhead by adding shortcuts for the computation of $\mathbf{T}^A \vee 0^{n \times n} = \mathbf{T}^A$, $\mathbf{T}^B \cdot 0^{n \times n} = 0^{n \times n}$, and $0^{n \times n} \cdot \mathbf{T}^C = 0^{n \times n}$, where $0^{n \times n}$ is a zero Boolean matrix.

Speeding up imbalanced multiplications. The introduction of the graph update matrix $\Delta\mathbf{T}$ makes the operands of some matrix multiplication operations imbalanced in terms of their sparsity. In particular, in $\mathbf{T}_{old} \cdot \Delta\mathbf{T}$ and $\Delta\mathbf{T} \cdot \mathbf{T}_{old}$, the matrix $\Delta\mathbf{T}$ is usually highly sparse, while the matrix $\mathbf{T}_{old}$ is less sparse. When a considerably sparser matrix is on the left (resp., right), it is more efficient to use the row-major (resp., column-major) matrix storage format, as noted in the SuiteSparse:GraphBLAS documentation [7]. To exploit this, we maintain a copy of $\mathbf{T}$ in each of the two formats and dynamically choose the best format for every multiplication operation based on operand sparsity.

In practice, to reduce memory consumption, in our implementation the row-major (resp., column-major) copy of $\mathbf{T}$ does not store non-terminals that do not appear as C (resp., B) in production rules of the form $A \to B\ C$.

Speeding up additions. Surprisingly, the next bottleneck is not matrix multiplication but rather matrix addition, namely $\mathbf{T} \leftarrow \mathbf{T} \cup \Delta\mathbf{T}$. This operation is slow because adding a highly sparse matrix $\Delta\mathbf{T}$ to a less sparse matrix $\mathbf{T}$ causes a complete reconstruction of $\mathbf{T}$. We address this bottleneck by avoiding the summation of matrices that differ drastically in their sparsity. In particular, we fix a constant factor $b > 1$ in the implementation[2] and represent $\mathbf{T}$ as a set of matrices $\widetilde{T} \subseteq \mathcal{R}^{n \times n}_{\mathcal{G}'}$ such that $\mathbf{T} = \bigcup \widetilde{T}$ and for any two distinct matrices $\mathbf{T}_1, \mathbf{T}_2 \in \widetilde{T}$, one matrix's number of non-zero elements (denoted $nnz(\cdot)$) is at least b times larger than the other's. Formally, the following invariant holds:

$$\max\{nnz(\mathbf{T}_1), nnz(\mathbf{T}_2)\} \geq b \cdot \min\{nnz(\mathbf{T}_1), nnz(\mathbf{T}_2)\}. \quad (2)$$

We perform all operations involving $\mathbf{T}$ without fully materializing $\bigcup \widetilde{T}$. For instance, $\Delta\mathbf{T} \cdot \mathbf{T}$ is computed as $\bigcup_{\mathbf{T}_i \in \widetilde{T}} \Delta\mathbf{T} \cdot \mathbf{T}_i$, and $\mathbf{T} \leftarrow \mathbf{T} \cup \Delta\mathbf{T}$ is computed by first adding $\Delta\mathbf{T}$ to $\widetilde{T}$ as a set element, and then repeatedly replacing $\widetilde{T}$ with $\widetilde{T} \setminus \{\mathbf{T}_1, \mathbf{T}_2\} \cup \{\mathbf{T}_1 \cup \mathbf{T}_2\}$ as long as there exist distinct matrices $\mathbf{T}_1, \mathbf{T}_2 \in \widetilde{T}$ that violate invariant (2).

Handling large CFGs. In many static analyses, the context-free grammars contain a large number of productions—often proportional to the number of call sites or fields (see Fig. 1 and Fig. 3 for examples). This high production count makes the standard approach based on Valiant's formula (1) inefficient because it requires a separate matrix multiplication for each production.

For example, consider a group of productions of the form $SR_i \to S\ R_i$ for $i \in \{1, \ldots, k\}$. Using Valiant's formula (1), we would compute each $(\mathbf{T} \cdot \mathbf{T})^{SR_i}$ via a separate multiplication $\mathbf{T}^S \cdot \mathbf{T}^{R_i}$, potentially iterating k times over the $\mathbf{T}^S$ matrix. Instead, we can aggregate the matrices $\mathbf{T}^{R_i}$ into a single block row matrix $\begin{bmatrix}\mathbf{T}^{R_1} & \mathbf{T}^{R_2} & \cdots & \mathbf{T}^{R_k}\end{bmatrix}$, and compute all corresponding products in one go:

$$\begin{bmatrix}\mathbf{T}^{SR_1} & \mathbf{T}^{SR_2} & \cdots & \mathbf{T}^{SR_k}\end{bmatrix} = \mathbf{T}^S \cdot \begin{bmatrix}\mathbf{T}^{R_1} & \mathbf{T}^{R_2} & \cdots & \mathbf{T}^{R_k}\end{bmatrix}.$$

This approach applies similarly to other groups of production rules where the only difference among the individual rules is in the indices (subscripts) of non-terminals.

It is worth noting that stacking matrices into a block matrix also takes some time. To avoid this overhead, we use block matrices for indexed non-terminals from the start and perform all operations exclusively in the block matrix form.

Transforming CFGs. For two equivalent CFGs that generate the same CFL, the performance of Algorithm 1 can vary significantly. As our last purely empirical enhancement, we manually fuzzed CFGs from the CFPQ_Data dataset [13] and for two client static analyses obtained the equivalent CFGs shown in Fig. 3 and Fig. 4. As will be demonstrated in Fig. 6, replacing the original CFGs from the CFPQ_Data dataset with our equivalent CFGs substantially accelerates our solver.

5 Experiments

To evaluate our enhancements, we implemented the proposed modifications in our FastMatrixCFPQ CFL-reachability solver [19] and compared its performance against the baseline MatrixCFPQ implementation [19] and other state-of-the-art CFL-reachability solvers: Pearl [24], Pocr [15], Graspan [3], and Gigascale [9].

We used three datasets—CPU 17[3] [14], CFPQ_Data [13], and CFPQ_JavaGraph [17]—which together comprise over 50 graphs corresponding to five client analyses. Specifically, CPU 17 provides program graphs for field-sensitive C/C++

[2] In our implementation, $b = 10$.

[3] From the CPU 17 dataset, we used the SVFG and PEG graphs located in the "simplified-interdyck" folders. These graphs were preprocessed using the procedure outlined in Pocr [16], which involves cycle elimination [29] and variable substitution [23]. Our evaluation does not include the preprocessing time.

$$
\begin{aligned}
PT &\to PTH\ alloc \\
PTH &\to \varepsilon \mid assign\ PTH \\
PTH &\to load_i\ Al\ store_i\ PTH \\
FT &\to \overline{alloc}\ FTH \\
FTH &\to \varepsilon \mid \overline{assign}\ FTH \\
FTH &\to \overline{store_i}\ Al\ \overline{load_i}\ FTH \\
Al &\to PT\ FT
\end{aligned}
\qquad
\begin{aligned}
PT &\to alloc \mid assign\ PT \\
PT &\to LPFS_i\ PT \\
FT &\to \overline{alloc} \mid FT\ \overline{assign} \\
FT &\to FT\ SPFL_i \\
LPFS_i &\to LP_i\ FS_i \\
LP_i &\to load_i\ PT \\
FS_i &\to FT\ store_i \\
SPFL_i &\to SP_i\ FL_i \\
SP_i &\to \overline{store_i}\ PT \\
FL_i &\to FT\ \overline{load_i}
\end{aligned}
$$

Figure 3. CFGs for field-sensitive Java points-to analysis. PT is the start non-terminal. Production rules involving index i are repeated for all $i \in \{1, \ldots, k\}$, where k is the number of fields. *(Left):* A CFG taken from CFPQ_Data [13]. *(Right):* An equivalent CFG whose use instead of the original CFG leads to improved performance of our solver.

$$
\begin{aligned}
M &\to \overline{d}\ V\ d \\
V &\to V_1\ V_2\ V_3 \\
V_1 &\to \varepsilon \mid V_2\ \overline{a}\ V_1 \\
V_2 &\to \varepsilon \mid M \\
V_3 &\to \varepsilon \mid a\ V_2\ V_3
\end{aligned}
\qquad
\begin{aligned}
M &\to N_1\ N_3 \mid N_2\ N_3 \\
N_1 &\to \overline{d} \mid N_1\ \overline{a} \mid N_2\ \overline{a} \\
N_2 &\to N_1\ M \\
N_3 &\to d \mid a\ N_3 \mid AM\ N_3 \\
AM &\to a\ M
\end{aligned}
$$

Figure 4. CFGs for field-insensitive C/C++ memory alias analysis. M is the start non-terminal. *(Left):* A CFG taken from CFPQ_Data [13]. *(Right):* An equivalent CFG whose use instead of the original CFG leads to improved performance of our solver.

alias analysis (FSCA) [34] and context-sensitive C/C++ value-flow analysis (CSCVF) [28]; CFPQ_JavaGraph provides graphs exclusively for field-sensitive Java points-to analysis (FSJPT) [8]; and CFPQ_Data includes program graphs for field-insensitive C/C++ memory alias analysis (FICA) [34] and field-sensitive Java points-to analysis (FSJPT) [8] as well as RDF graphs for same layer of hierarchy analysis (SLH) [33]. For each analysis, the CFG was taken from the respective cited work. Characteristics of representative graphs are shown in Table 1.

All experiments were conducted on a machine with a 12-core Ryzen 9 7900X @ 4.7 GHz CPU and 128 GB of DDR5 RAM, running Ubuntu 20.04. We set a timeout of 10,000 seconds for every individual solver run.

Correctness. In all cases, the results obtained by our FastMatrixCFPQ solver matched those of other correctly working solvers.

Table 1. Characteristics of some graphs: number of nodes ($|V|$), number of edges ($|E|$), and the size of the $\mathcal{L}$-reachability relation for the corresponding analysis ($|\mathcal{R}|$).

Graph (analysis)	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\mathcal{R}\rvert$
pmd (FSJPT) [13]	54,444	118,658	60,518
tradeb (FSJPT) [13]	439,693	933,938	696,316
junit5 (FSJPT) [17]	59,818	149,370	129,598
guava (FSJPT) [17]	129,562	336,232	26,384,496
taxon_h (SLH) [13]	2,112,625	65,752,578	5,351,657
postgre (FICA) [13]	5,203,419	9,357,086	90,661,446
perlb (FSCA) [14]	38,091	110,874	851,737,865
parest (CSCVF) [14]	233,900	307,850	1,342,540
perlb (CSCVF) [14]	605,864	1,114,892	297,504,186

Overall performance. We ran all CFL-reachability solvers five times on every graph and the corresponding CFG from the considered datasets. For each combination of a client analysis and a dataset, we identified the graph on which FastMatrixCFPQ exhibited its worst performance in terms of analysis time. Fig. 5 presents the average analysis time and peak memory usage of all solvers on these most demanding graphs.[4]

Our experiments show that for all considered program analyses FastMatrixCFPQ is drastically faster than the state-of-the-art universal CFL-reachability solvers such as MatrixCFPQ [2], Pocr [16], and Graspan [32]. For example, no other universal solver was able to analyze the "guava (FSJPT)" graph in under 10,000 seconds, while FastMatrixCFPQ completed the analysis in 7.7 seconds. On the other hand, for RDF graphs performance gains are more moderate (up to 1.4× speedup relative to the baseline MatrixCFPQ).

Comparing the execution time of FastMatrixCFPQ with that of the specialized solvers Pearl and Gigascale yields mixed results. In many cases, such as "perlb (CSCVF)" and "tradeb (FSJPT)", FastMatrixCFPQ is significantly faster than these solvers (up to 9.2× faster). However, in some cases, such as "guava (FSJPT)", it is up to 6.4× slower than Gigascale, which is specifically designed for FSJPT analysis.

In terms of memory efficiency, our experiments indicate that FastMatrixCFPQ usually exhibits the smallest memory footprint among all CFL-reachability solvers. However, as shown in Fig. 5, there are exceptions. For instance, when analyzing "taxon_h (SLH)", Graspan achieved the lowest peak memory usage, albeit running almost 500 times slower than FastMatrixCFPQ.

Enhancements analysis. To understand the impact of each enhancement, we performed an ablation study, incrementally implementing and evaluating each optimization on top of the baseline MatrixCFPQ algorithm. Fig. 6 shows how the

[4] For brevity, we only report results on the most demanding graphs. Similar performance trends were observed across nearly all tested graphs [18].

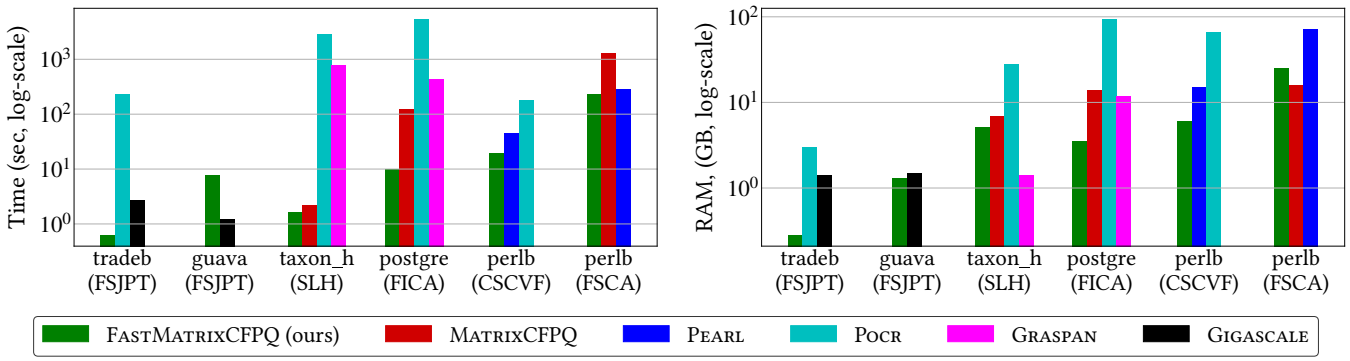


Figure 5. Performance of CFL-reachability solvers. Mean values over five runs are shown with sample standard deviations below 7% in all cases. If a solver's bar is missing, it indicates that the solver either exceeded the 10,000-second time limit, exceeded the 128 GB memory limit, or is incompatible with the analysis. *(Left):* Execution time. *(Right):* Peak memory usage.

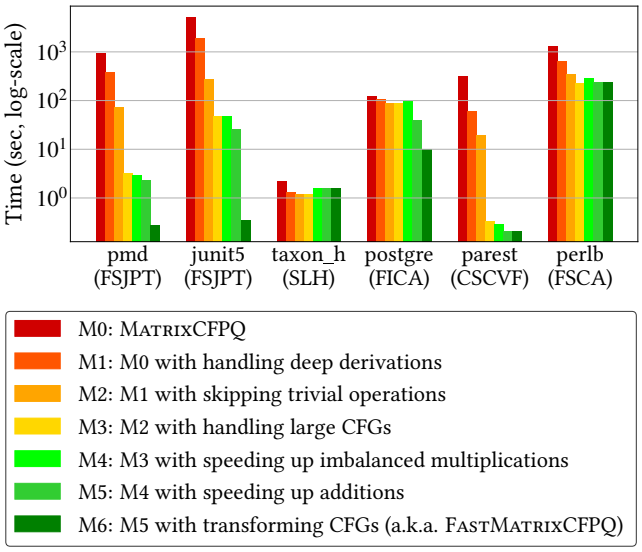


Figure 6. Execution time of MatrixCFPQ with various numbers of enhancements. Mean times over five runs are shown with sample standard deviations below 8% in all cases. "Mn" denotes MatrixCFPQ with n enhancements.

execution time changes as each subsequent enhancement is introduced. Note that the subset of graphs presented in Fig. 6 differs from those in Fig. 5 because we replaced the graphs on which the baseline MatrixCFPQ timed out.

The experimental results show significant speedups from handling deep derivations (up to 5.3×), skipping trivial operations (up to 7×), handling large CFGs (up to 58×), speeding up matrix additions (up to 2.4×), and transforming CFGs (up to 74×). In contrast, the enhancement for speeding up imbalanced multiplications is less effective due to the overhead of maintaining an additional copy of **T**, and often even causes slowdowns. The combined effect of all enhancements yields a cumulative speedup of up to 14,800× relative to the baseline MatrixCFPQ.

6 Conclusion

We have presented FastMatrixCFPQ, a universal CFL-reachability solver that is well suited for handling a wide range of context-free languages in realistic program analysis settings. The solver leverages high-performance matrix operations and overcomes the scalability limitations of MatrixCFPQ with respect to both grammar size and derivation depth. Our experimental evaluation across five client analyses—field-insensitive memory alias analysis, field-sensitive alias analysis, context-sensitive value-flow analysis, field-sensitive points-to analysis, and same layer of hierarchy analysis—demonstrates that FastMatrixCFPQ consistently outperforms both the baseline MatrixCFPQ and the other state-of-the-art universal solvers, often achieving orders-of-magnitude speedups. Furthermore, in many cases, FastMatrixCFPQ even exceeds the performance of specialized tools.

7 Future Work

Building on these promising results, we plan to generalize our enhancements to the multiple-source variant [30] and the all-paths variant [1] of MatrixCFPQ. Unlike the all-pairs version, the multiple-source version can be used for on-demand analysis, while the all-paths version can detect "contributing edges" [10] and serve as a high-performance base for a mutual-refinement algorithm [4, 10] that approximates interleaved Dyck reachability, thereby enabling analyses that are both context-sensitive and field-sensitive.

We also plan to further boost the performance of FastMatrixCFPQ by reducing redundant derivations (using techniques similar to those in Pearl [25]) and by offloading matrix computations to GPUs.

Data Availability Statement

The materials for our evaluation are publicly available [18] and can be used to reproduce the experimental data. The code is hosted in the project repository [19].

References

[1] Rustam Azimov, Ilya Epelbaum, and Semyon Grigorev. 2021. Context-free path querying with all-path semantics by matrix multiplication. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Virtual Event, China) *(GRADES-NDA '21)*. Association for Computing Machinery, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/3461837.3464513

[2] Rustam Azimov and Semyon Grigorev. 2018. Context-Free Path Querying by Matrix Multiplication. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Houston, Texas) *(GRADES-NDA '18)*. Association for Computing Machinery, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/3210259.3210264

[3] Pan Chuang and Zhiqiang Zuo. 2020. Graspan-C: A Disk-based Highly Parallel Interprocedural Static Analysis Engine. https://github.com/Graspan/Graspan-C

[4] Giovanna Kobus Conrado and Andreas Pavlogiannis. 2024. A Better Approximation for Interleaved Dyck Reachability. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (Copenhagen, Denmark) *(SOAP 2024)*. Association for Computing Machinery, New York, NY, USA, 18–25. https://doi.org/10.1145/3652588.3663318

[5] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4, Article 44 (Dec. 2019), 25 pages. https://doi.org/10.1145/3322125

[6] Timothy A. Davis. 2023. Algorithmnbsp;1037: SuiteSparse:GraphBLAS: Parallel Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 49, 3, Article 28 (Sept. 2023), 30 pages. https://doi.org/10.1145/3577195

[7] Timothy A. Davis, Corey J. Nolet, Joe Eaton, Christoph Grüninger, Gabor Szarnyas, Markus Mützel, and Erik Welch. 2024. SuiteSparse:GraphBLAS, the official SuiteSparse library. https://github.com/DrTimothyAldenDavis/GraphBLAS

[8] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. 2015. Giga-Scale Exhaustive Points-to Analysis for Java in under a Minute. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 535–551. https://doi.org/10.1145/2814270.2814307

[9] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. 2015. Implementation of Giga-Scale Exhaustive Points-To Analysis for Java in Under a Minute. https://bitbucket.org/jensdietrich/gigascale-pointsto-oopsla2015/src/master/

[10] Shuo Ding and Qirun Zhang. 2023. Mutual Refinements of Context-Free Language Reachability. In *Static Analysis*, Manuel V. Hermenegildo and José F. Morales (Eds.). Springer Nature Switzerland, Cham, 231–258. https://doi.org/10.1007/978-3-031-44245-2_12

[11] Márton Elekes, Attila Nagy, Dávid Sándor, János Benjamin Antal, Timothy A. Davis, and Gábor Szárnyas. 2020. A GraphBLAS solution to the SIGMOD 2014 Programming Contest using multi-source BFS. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE Press, USA, 1–7. https://doi.org/10.1109/HPEC43674.2020.9286186

[12] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. 2023. A Systematic Survey of General Sparse Matrix-Matrix Multiplication. *ACM Comput. Surv.* 55, 12, Article 244 (mar 2023), 36 pages. https://doi.org/10.1145/3571157

[13] Nikita Kovalev and Vadim Abzalov. 2022. CFPQ_Data: Graphs and Grammars for Context-Free Path Querying Algorithms evaluation. https://github.com/FormalLanguageConstrainedPathQuerying/CFPQ_Data

[14] Yuxiang Lei. 2024. CPU17-graphs: SPEC 2017 C/C++ Program Graphs for Context-Free Language Reachability (CFL-r) Evaluation. https://github.com/kisslune/CPU17-graphs

[15] Yuxiang Lei. 2024. POCR: CFL-reachability tool. https://github.com/kisslune/POCR

[16] Yuxiang Lei, Yulei Sui, Shuo Ding, and Qirun Zhang. 2022. Taming Transitive Redundancy for Context-Free Language Reachability. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 180 (oct 2022), 27 pages. https://doi.org/10.1145/3563343

[17] Ilia Muravev. 2024. CFPQ_JavaGraphMiner: Java Graph Miner And Dataset for CFL-reachability-based analyses. https://github.com/FormalLanguageConstrainedPathQuerying/CFPQ_JavaGraphMiner

[18] Ilia Muravev and Semyon Grigorev. 2025. Artifact of "Universal High Performance CFL-reachability via Matrix Multiplication". https://doi.org/10.5281/zenodo.15324265

[19] Ilia Muravev, Semyon Grigorev, Vladimir Kutuev, and Rustam Azimov. 2024. CFPQ_PyAlgo: The collection of Context-Free Path Querying algorithms. https://github.com/FormalLanguageConstrainedPathQuerying/CFPQ_PyAlgo/tree/fast-matrix-cfpq

[20] Thomas Reps. 1998. Program analysis via graph reachability. *Information and Software Technology* 40, 11 (1998), 701–726. https://doi.org/10.1016/S0950-5849(98)00093-7

[21] Thomas Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 162–186. https://doi.org/10.1145/345099.345137

[22] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

[23] Atanas Rountev and Satish Chandra. 2000. Off-Line Variable Substitution for Scaling Points-to Analysis. *SIGPLAN Not.* 35, 5 (may 2000), 47–56. https://doi.org/10.1145/358438.349310

[24] Chenghang Shi, Haofeng Li, Yulei Sui, Jie Lu, Lian Li, and Jingling Xue. 2024. Artifact of "Pearl: A Multi-Derivation Approach to Efficient CFL-Reachability Solving". https://doi.org/10.6084/m9.figshare.23702271

[25] Chenghang Shi, Haofeng Li, Yulei Sui, Jie Lu, Lian Li, and Jingling Xue. 2024. Pearl: A Multi-Derivation Approach to Efficient CFL-Reachability Solving. *IEEE Trans. Softw. Eng.* 50, 9 (Sept. 2024), 2379–2397. https://doi.org/10.1109/TSE.2024.3437684

[26] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. *Proc. ACM Program. Lang.* 3, POPL, Article 48 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290361

[27] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. *SIGPLAN Not.* 40, 10 (Oct. 2005), 59–76. https://doi.org/10.1145/1103845.1094817

[28] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering* 40, 2 (2014), 107–122. https://doi.org/10.1109/TSE.2014.2302311

[29] Robert Tarjan. 1971. Depth-First Search and Linear Graph Algorithms. In *Proceedings of the 12th Annual Symposium on Switching and Automata Theory (Swat 1971) (SWAT '71)*. IEEE Computer Society, USA, 114–121. https://doi.org/10.1109/SWAT.1971.10

[30] Arseniy Terekhov, Vlada Pogozhelskaya, Vadim Abzalov, Timur Zinnatulin, and Semyon V. Grigorev. 2021. Multiple-Source Context-Free Path Querying in Terms of Linear Algebra. In *International Conference on Extending Database Technology*. OpenProceedings, Konstanz, Germany, 487–492. https://doi.org/10.5441/002%2Fedbt.2021.56

[31] Leslie G. Valiant. 1975. General context-free recognition in less than cubic time. *J. Comput. System Sci.* 10, 2 (1975), 308–315. https://doi.

org/10.1016/S0022-0000(75)80046-8

[32] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-Machine Disk-Based Graph System for Interprocedural Static Analyses of Large-Scale Systems Code. *SIGPLAN Not.* 52, 4 (apr 2017), 389–404. https://doi.org/10.1145/3093336.3037744

[33] Xiaowang Zhang, Zhiyong Feng, Xin Wang, Guozheng Rao, and Wenrui Wu. 2016. Context-Free Path Queries on RDF Graphs. In *The Semantic Web – ISWC 2016*, Paul Groth, Elena Simperl, Alasdair Gray, Marta Sabou, Markus Krötzsch, Freddy Lecue, Fabian Flöck, and Yolanda Gil (Eds.). Springer International Publishing, Cham, 632–648. https://doi.org/10.1007/978-3-319-46523-4_38

[34] Xin Zheng and Radu Rugina. 2008. Demand-Driven Alias Analysis for C. *SIGPLAN Not.* 43, 1 (jan 2008), 197–208. https://doi.org/10.1145/1328897.1328464

Received 2025-03-04; accepted 2025-04-25

Scalable Language Agnostic Taint Tracking using Explicit Data Dependencies

Sedick David Baker Effendi
Stellenbosch University
Stellenbosch, South Africa
dbe@sun.ac.za

Xavier Pinho
StackGen
San Ramon, USA

Andrei Michael Dreyer
Whirly Labs
Cape Town, South Africa

Fabian Yamaguchi
Whirly Labs
Cape Town, South Africa

Abstract

Taint analysis using explicit whole-program data-dependence graphs is powerful for vulnerability discovery but faces two major challenges. First, accurately modeling taint propagation through calls to external library procedures requires extensive manual annotations, which becomes impractical for large ecosystems. Second, the sheer size of whole-program graph representations leads to serious scalability and performance issues, particularly when quick analysis is needed in continuous development pipelines.

This paper presents the design and implementation of a system for a language-agnostic data-dependence representation. The system accommodates missing annotations describing the behavior of library procedures by over-approximating data flows, allowing annotations to be added later without recalculation. We contribute this data-flow analysis system to the open-source code analysis platform Joern, making it available to the community.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Formal software verification.*

Keywords: static analysis, taint analysis, code property graph, data flow

ACM Reference Format:
Sedick David Baker Effendi, Xavier Pinho, Andrei Michael Dreyer, and Fabian Yamaguchi. 2025. Scalable Language Agnostic Taint Tracking using Explicit Data Dependencies. In *Proceedings of the 14th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '25), June 16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3735544.3735586

This work is licensed under a Creative Commons Attribution 4.0 International License.
SOAP '25, Seoul, Republic of Korea
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1922-6/25/06
https://doi.org/10.1145/3735544.3735586

1 Introduction

Continuous integration and deployment [9] are now standard in many organizations [12], but achieving continuous vulnerability detection without slowing down the release process remains an ambitious goal. Vulnerability discovery techniques such as symbolic execution [e.g. 18, 22] or fuzz testing [e.g. 5, 19] fall short in this environment, as they assume a relatively static target. Expensive state exploration, however, stands in direct conflict with the need for quick feedback in modern pipelines.

Researchers have explored using graph databases to store and process whole-program representations of code [24]. In this context, explicit data-dependence representations have proven particularly useful in vulnerability discovery [17], as they can model a wide range of taint-style vulnerabilities, including command injections, file inclusion vulnerabilities, and cross-site scripting (XSS) vulnerabilities. These representations facilitate combining taint propagation information with syntactic and control-flow information to identify vulnerable code [29] and enable automated processing using graph-based machine learning algorithms [6]. However, their accuracy hinges on knowing the taint propagation semantics of all methods.

We present and implement a taint-tracking strategy based on a whole-program data-dependence representation that can be incrementally updated as knowledge about the semantics of external libraries becomes available, avoiding full recomputation when adding new annotations. We contribute our resulting work to an existing open-source code analysis platform, Joern [14], and make it available to the research community. We evaluate the efficiency of our analysis on Java, Python, and JavaScript programs.

2 Background

Consider a simple example of data flow that spans external calls to motivate our approach. Listing 1 defines two methods, `foo` and `bar`. The `foo` method obtains an object `u` from an external source (`Source.getValue`), creates a new object `v`, and calls `u.transform(v)` to produce `result`, which is then passed to `bar(result, v)`. The `bar` method calls

an external method (`Sink.addValue`) on both its parameters. We want to know whether data from the source, i.e., the return of `Source.getValue`, can reach the first argument of `Sink.addValue`. This means checking if a call to `Source.getValue` *defines* a value w that is later *used* as w' in a call to `Sink.addValue(`w'`)`. In other words, is w' obtained through a series of transformations from w?

```
public class Example {
  public static void foo() {
    Obj u = Source.getValue();
    Obj v = new Obj();
    if (Config.isPrivileged()) {
      Obj result = u.transform(v);
      bar(result, v); // internal
    }
  }
  static void bar(Obj x, Obj y) {
    Sink.addValue(x); // sink
    Sink.addValue(y); // sink
  }
}
```

Listing 1. Sample Java code with methods `foo` and `bar` that call external methods `getValue`, `isPrivileged`, `addValue`, and `transform`.

This perspective on code as operations on variables for which arguments are *used*, *defined*, or *used* and *defined* within a method can be expressed via a *data dependence graph*. Originally developed for program slicing [8], this graph contains edges from nodes describing operations that define a variable to those that use it without prior redefinitions.

When no information is available about external methods, an analyzer has two options: assume the calls have no effect or assume they taint everything. With the former approach, we obtain an under-approximated graph where no data dependencies are established between the external method calls. With the latter approach, we obtain an over-approximated graph with spurious data dependency paths that may not reflect actual taint propagation.

Compared to the under-approximated graph, the over-approximated graph offers the advantage that the possible data dependency between `result` at the call to `bar` and its occurrence at the call to `Sink.addValue` is indicated by a path in the data dependence graph. Similarly, the potential re-definition of `u` or `v` introduced by the call to `transform` is visible due to the inability to traverse from the node of `bar` to that of `Sink.addValue` without passing through that of `transform`. This is an example of a transitive data dependency, where a dependency from one variable to another is due to a chain of intermediate steps or functions [13].

To deal with these transitive dependencies, Horwitz et al. [13] proposes an elegant extension of intraprocedural data dependence graphs, which they refer to as *system dependence graphs*. In the system dependence graph, separate nodes for input and output arguments are introduced, and transitive dependencies are encoded via direct edges from input to output nodes. This compresses transitive dependencies onto a representation that only perceives local (non-transitive) dependencies.

Nonetheless, the idea of maintaining a representation of data dependencies that is independent of transitive data dependencies – and that therefore does not need to be recalculated as new information about external methods becomes available – forms the intellectual basis for our approach. This fits our scenario well, where the behavior of external methods may be characterized in greater detail by the user over time.

3 Design

The Joern [14] code analysis platform is extended with a data-flow engine. Joern's language frontends and standard stages generate a unified abstract syntax tree (AST), control flow graph (CFG), and control dependence graph (CDG), forming a near-complete *code property graph (CPG)* [29]. The data-flow engine provides the necessary primitives to construct the intermediate data dependence graph (DDG) for a full CPG. It includes a querying engine to determine flows on the fly for specified sources and sinks. This scheme is a *may* analysis that identifies flows from sources to sinks, considering user-provided semantics of external methods. The following sections describe the design and implementation in greater detail.

3.1 Data-Dependence Representation

The data-dependence representation is based on program dependence graphs (PDGs) [8] (see Figure 1).

A problem with this approach is that a method's data-dependence representation is only precise if the semantics of all its transitive callees are known [13]. As [13] furthermore shows, summary edges can encode these semantics and be calculated in polynomial time for callee methods available at analysis time. However, for external library methods (with code unavailable at compile time), the user must provide semantics, or they are assumed to be unknown. In this case, it is assumed that all input parameters may taint all output parameters to safely overapproximate the data flow. Palepu et al. [21] finds success in dynamically generating program summaries for external library code as data and control dependencies between inputs and outputs of calls to external procedures. The authors acknowledge that these summaries can introduce unsoundness and imprecision; however, the performance gains may outweigh these costs. Toman and Grossman [27] explores how library code may bring along many transitive dependencies, and a resulting summary for a method may require referencing indirect flows to other functions. The related efforts in summarising external code suggest difficulties in how granular these could be in a language-agnostic approach, and one must accept the

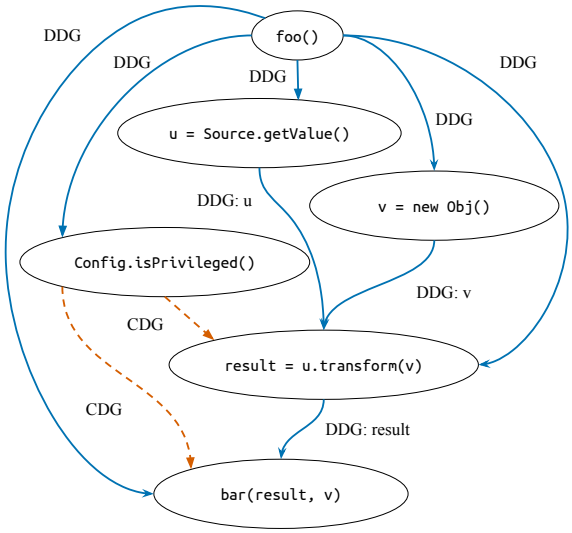


Figure 1. The program dependence graph of the code in Listing 1. Edges are labelled as belonging to either the control dependence graph (CDG) or the data dependence graph (DDG).

inherent imprecision and unsoundness introduced by using such an approach.

To address these challenges, our data-flow engine maintains a stable data-dependence representation as users refine method semantics. It achieves this by treating all callees as external with unknown semantics, over-approximating data dependencies at each call site. Unlike the exploded supergraph in the classical IFDS framework [23], this approach does not rely on hard-coded semantics. While this introduces invalid paths, i.e., paths that are not valid in any runtime execution, they are discarded at query time. As a result, adding such summaries is not required to discover additional flows but helps eliminate false positives. In Figure 1, we note that a call to `transform` is crucial in determining which paths from the `Source` to `Sink` classes are valid. As shown in Listing 2, we can describe the valid flows for `transform` to have several outcomes. To define a semantic, one must supply the method's full name, followed by a list of flows between arguments annotated by their positional index and/or argument name, for languages that support named arguments.

Certain positional indexes denote special cases. Such special cases are the return of a call, as index "`-1`", and the receiver as index "`0`", which denotes the object to which the method is bound. Any unspecified flows will be interpreted as *killed* or *sanitized*, i.e., no flow exists between the input and output node. Thus, we need to be explicit where flows are not killed, e.g., `0->0`. As this may become tedious for methods with many arguments, several special flow objects in the programmatic API provide shorthand ways to define common cases.

To explain how one can define these semantics, we detail the examples from Listing 2. The first parameter is sanitized (and thus omitted), while the receiver is not and propagates to the return value, defining flows `0->0` and `0->-1`. Next, we modify the semantics so that the receiver now taints the non-sanitized first parameter, resulting in flows `1->1` and `0->1`.

```
/* E.g.1: Argument 1 is sanitized, receiver flow
 propagates to the return value */
"Obj.transform:Obj(Obj)" 0->0 0->-1

/* E.g.2: Receiver taints argument 1 */
"Obj.transform:Obj(Obj)" 0->0 1->1 0->1
```

Listing 2. An example of user-supplied semantics for a call to `transform`.

Semantics can be written manually or programmatically. One can use heuristics, data-driven approaches, or the data-flow engine to programmatically generate and load new semantics on the fly until one needs to run a data-flow query.

3.2 Identifying Data-Flows

With the data-dependence representation in place, the next step is determining data flows based on user-provided queries. As is true for many other taint analysis systems [e.g., 4, 10, 25], our query consists of a set of sources and a set of sinks, and it is our goal to determine all source-sink pairs for which a flow from source to sink exists, along with a sample flow. However, as the data-dependence representation does not have hard-coded semantics, a query also includes a set of semantics for external library methods, as we allow the semantics of library methods to change.

Given such a query, the goal now is to calculate data flows in an algorithmically efficient manner that effectively uses multicore CPUs. To this end, an approach similar to Duesterwald et al. [7] is chosen. They answer queries incrementally, translating queries into tasks and deriving new tasks from the results of prior tasks at method boundaries. Using this approach, each task operates only within the boundaries of a method, such as `foo` or `bar` shown in Listing 1, and can be calculated independently and concurrently.

Taint analysis can be performed in forward and backwards modes: either traverses data-dependence edges from sources along the edge direction towards sinks or from sinks against the edge direction towards sources. In the following, only the taint analysis in the backwards direction is described, but the forward direction can be implemented analogously.

A (backwards) task is defined to be given by a start node, an already-known path from the start node to a sink node, the set of source nodes, and the set of semantics. Moreover, a positive integer that indicates analysis depth is stored, referring to the call-chain depth that the analysis explores.

To simplify notation, tasks are described only by pairing start nodes and paths from the start node to the sink; the

result table and the analysis depth are assumed to be available. As the sources, sinks, and semantics remain constant throughout the processing of a query, it is assumed that they are available for reading via a globally shared object. With these simplifications in mind, for a given set of sinks $\mathcal{D}$, the initial set of tasks is given by $\{(d, [], 0) \mid d \in \mathcal{D}\}$, where [] denotes an empty path, and 0 is the initial call depth.

These tasks are submitted to a work queue, with resulting paths pushed to the output queue. A result can either be *complete*, meaning it describes a flow from a source in $\mathcal{S}$ to a sink in $\mathcal{D}$, or it can be *partial*, meaning that it is a flow that may be part of a complete flow from a source to a sink. We fetch these results from the output queue, record complete results, and derive new tasks from each result. We note that tasks must also be created from complete results, as they may describe sub-flows of a larger complete flow. This procedure is carried out until all tasks have been evaluated and no more new tasks need to be submitted. At this point, all recorded results are returned.

A result is given by a path $p = ([(v_1, r_1), \ldots, (v_N, r_N)], k)$ where N is the path length, and for all i from 1 to N, v_i is a node, r_i is a Boolean, and k is the current call depth. For nodes that are arguments in method calls, the Boolean r_i indicates whether the associated method has been resolved in the process of generating the result (`true`) or whether resolving it has been deferred to a future task (`false`).

Translating results to new tasks. From a result p, new tasks are generated according to the following rules, shown by Algorithm 1. First, new tasks are only created if the new call depth is no larger than the maximum depth (line 3). If so, do not generate new tasks (line 4), resulting in partial tasks with no new dependent tasks. In this case, flows will be over-approximated for dependent callers of this result. This early termination is a form of widening to ensure termination, analogous to k-limiting [15]. Second, if the path begins with a parameter (line 7), we look up the set of corresponding arguments $\mathcal{A}$ (line 8) and generate the tasks $\{(a, p) \mid a \in \mathcal{A}\}$ (line 9). These corresponding arguments include positional or named arguments at call sites and the receiver of call sites referring to the parameter's method as a higher-order function. Finally, for the given path, all unresolved arguments are determined (line 11). For each unresolved argument, the tasks $\{(o, p) \mid o \in O\}$ (line 12) are generated from the set of associated formal output parameters O (line 11). If the argument is the actual return value of a call, the task (r, p) is also generated, where r denotes the corresponding formal return parameter. If the argument is a method reference, such as a closure, then the closure's return statement becomes a task (r_c, p), where r_c denotes the return statement of the closure.

Solving tasks. Each task (s, p, k) is solved by a separate worker thread. Results are calculated by inspecting s alone and then determining results for all *valid parents*, that is, nodes with an outgoing data-dependence edge to s that is *valid* according to the semantics S.

Algorithm 1 Given a partial result p, generates new tasks from parameters and unresolved arguments using the call graph.

```
1:  procedure CreateTasksFromResult(p)
2:      (x, k) ← p        ▷ Extract the path x and call depth k
3:      if k + 1 >= k_max then
4:          return ∅
5:      end if
6:      (v, r) ← x[0]   ▷ Extract head node v and Boolean r
7:      if IsParameter(v) then
8:          A ← GetArgsFromCallers(v)
9:          return [(a, p, k + 1) for a ∈ A]
10:     else if IsArgument(v) and r is false then
11:         O ← GetUnresolvedOutArgsAndReturns(v)
12:         return [(o, p, k + 1) for o ∈ O]
13:     else
14:         return ∅
15:     end if
16: end procedure
```

The result for s is determined as follows. If the head of p is a source, the result is a complete path $(s, \text{false}) : p$, where ":" denotes an append operation. An additional partial path result is pushed if the source is a method parameter. This additional result allows Algorithm 1 to create a new task from this result and possibly find additional sources later. If the head of p is not from the source set but a method parameter, then $(s, \text{false}) : p$ is returned as the path for a partial result.

To determine edge validity, the edges from actual returns of method calls are discarded if the semantic value states that the call does not define the return argument. If s is not an argument, we return the remaining list of parents. If s is an argument, incoming edges from parent nodes that are not arguments are valid. In these cases, return a partial result and mark the result as unresolved. These cases either reflect an incomplete call graph or that the task depends on a partial task that was discarded for exceeding the maximum call depth. In either case, the outcome will be that the result is over-approximated, i.e., it is assumed that all of its arguments are both used and defined by a call to the method.

Validity of parents based on semantics. A parent node s_0 is connected directly to s via an outgoing data-dependence edge, but not all edges are valid according to the semantics. For parent nodes that are arguments, if s and s_0 are arguments of the same call site and the parent node is used while s is defined according to the semantics, the edge is valid. The edge is also valid if s and s_0 are arguments of different call sites, but s is used according to the semantics; otherwise, the edge is invalid. The data flows are over-approximated for methods without defined semantics.

Computing results for valid parents. In the following, we refer to Algorithm 2. For each valid parent (lines 3–4), whether a result exists in the table is determined (line 5), and if so, it is used to compute the result by determining the sub-path from the current parent node to the sink and appending p, followed by the parent s_0. Otherwise, the result is computed recursively; that is, we compute the results for $s_0 : p$. Upon collecting results for all parents and the head node, deduplicate and return (line 13).

Algorithm 2 Given a task (s, p, k), determine valid results for parents of s using the semantics and data-dependence representation.

```
1:  procedure ComputeResultsForParents(s, p, k)
2:      R* ← ∅
3:      for all s0 in Out(s) do          ▷ Traverse DDG edges
4:          if IsValidEdge(s, s0) then
5:              if s0 ∈ R* then          ▷ Prepend known path
6:                  R* ← R* ∪ (R*[s0] : p, k)
7:              else
8:                  r0 ← IsOutputArg(s0)
9:                  R* ← R* ∪ ((s0, r0) : p, k)
10:             end if
11:         end if
12:     end for
13:     return Deduplicate(R*)
14: end procedure
```

Finally, the union of the results for p and its valid parents is returned. This result is stored in the result table as a cache.

4 Limitations

Operators such as assignments, arithmetic, and field accesses are modeled as ordinary call nodes with a default set of semantics. Consequently, aliasing and the heap of data structures are not tracked. In the case of aliases, assignments will propagate flow, but only via weak updates. For data structures that use index accesses for arbitrary keys, such as index values or keys in maps, the data-flow engine tracks these as "containers": if an internal member is tainted, then by the semantic definition, the container is tainted.

This leaves future work to make this analysis alias and object-sensitive. However, this imprecision may be attributed to the analysis's low overhead.

5 Evaluation

This evaluation aims to answer the following research questions: (**RQ1**) Is the system able to detect taint-style vulnerabilities effectively for multiple programming languages, and (**RQ2**) without analyzing library code? Finally, an essential property for data-flow analysis in the context of modern programs is (**RQ3**), i.e., how scalable is our analysis?

5.1 Method

We compare the precision of the data flow engine of Section 3 against two static analysis tools that support multiple languages, Semgrep [25] and CodeQL [10]. The primary considerations for related work are that they are widely adopted, support multi-language taint analysis, and allow partial program analysis. Each tool is run on the same Java, JavaScript, and Python benchmarks, where partial and whole program analysis techniques are compared. We measure precision and recall using the F1-score and Youden's J index [30] (rewards higher specificity). We also recorded each tool's analysis runtime and memory usage to assess scalability. All experiments [3] were performed on a platform with a 6-core x86 CPU (3.4 GHz), 32 GB memory, and running Java 21.0.2.

5.2 Dataset

Choosing a *well-suited* dataset for taint analysis is non-trivial, where we define well-suited as publicly available and providing a sink, source, and the outcome for any given test. Securibench Micro [20] meets these criteria for Java.

While Guarnieri et al. [11] mentions developing a benchmark akin to Securibench Micro but for JavaScript, the associated link is dead at the time of writing. To this end, and as a contribution, we develop *securibench-micro.js* [1] as a JavaScript equivalent to Securibench Micro. For Python, such a benchmark is not readily available; however, an incomplete synthetic benchmark similar in spirit to Securibench Micro exists. As another contribution, this benchmark is completed and dubbed "Thorat" [2] after its original author [26].

When measuring scalability, however, none of the programs in the datasets above compares in magnitude to an industry-sized program. To address this shortcoming, we use Defects4j [16] and include a Python-inspired variant, namely BugsInPy [28]. While not intended for measuring taint analysis, they include real-world programs that test the scalability of a static analysis tool. The latest versions of each program of these datasets are obtained at the time of writing.

5.3 Determining a Suitable Analysis Depth

To justify a suitable value for k to be used by the Joern-based data-flow analysis, one must explore how different values for k affect the results. The results' figures for finding an appropriate value for k have been omitted here for brevity but can be found within the supplementary material.

When measuring against the taint analysis benchmarks, each experiment runs for 10 iterations with user-defined semantics enabled. A significant variation in the J index and F score appears between $k \in [0, 3]$, followed by a slight increase in precision when $k = 8$ in Securibench Micro and securibench-micro.js. When observing the taint analysis wall-clock times for Defects4j and BugsInPy, for a subset of programs, the beginning of exponential complexity for

Table 1. Benchmark results on the partial program static taint analysis for Joern, Joern$_{SEM}$, Semgrep, and CodeQL.

Benchmark	Tool	TP	TN	FP	FN	J Index	F1 Score	Runtime (s)	Memory (GB)
Securibench Micro	Joern	119	17	36	17	0.196	0.818	**1.48±0.54**	0.33 ± 0.03
	Joern$_{SEM}$	118	36	17	18	**0.547**	**0.871**	1.74 ± 0.59	0.29 ± 0.02
	Semgrep	100	39	14	36	0.471	0.800	16.73 ± 0.57	**0.14±0.01**
	CodeQL	93	37	16	43	0.382	0.759	79.57 ± 1.05	1.26 ± 0.05
Thorat	Joern	29	22	12	11	0.372	**0.716**	0.91 ± 0.34	0.26 ± 0.02
	Joern$_{SEM}$	29	22	12	11	0.372	**0.716**	**0.77±0.25**	0.25 ± 0.02
	Semgrep	15	24	10	25	0.081	0.462	15.01 ± 0.41	**0.14±0.01**
	CodeQL	23	29	5	17	**0.423**	0.676	44.76 ± 0.69	0.97 ± 0.04
securibench-micro.js	Joern	93	18	26	24	0.204	0.788	6.94 ± 0.49	0.29 ± 0.02
	Joern$_{SEM}$	93	19	25	24	0.227	**0.791**	**6.88±0.47**	0.28 ± 0.02
	Semgrep	5	43	1	112	0.020	0.081	14.64 ± 0.57	**0.14±0.01**
	CodeQL	85	31	13	32	**0.431**	**0.791**	93.99 ± 1.98	1.31 ± 0.05

runtime is observed from $k = 6$. Thus, to strike a balance between precision and recall while remaining practical, we determine that $k = 5$ is a safe value for k.

5.4 Taint Analysis

This section outlines Joern's performance with the presented data-flow engine, using a max call depth $k = 5$, for the three benchmarks against Semgrep and CodeQL.

Table 1 presents the results of partial taint analysis for each evaluated tool. Joern is assessed in two configurations: without user-defined semantics (Joern) and with manually specified semantics (Joern$_{SEM}$). Both configurations include operator semantics, but the latter incorporates additional, manually curated semantics for external procedure calls, thereby mitigating unnecessary false positives. Whole program analysis is also considered, where the datasets used by Table 1 are appended with their external dependencies, including transitive ones. The results of whole-program analysis are omitted here but can be found within the supplementary material.

5.5 Discussion

Semantic annotations reduce false positives in Securibench Micro, none in Thorat, and one in securibench-micro.js. This is likely due to Java being a more verbose language than Python and JavaScript, leading to the overtainting of more data flows if calls are left unconstrained. Similarly, operations on data structures in JavaScript and Python often use syntactic sugar present as operators, such as index or property accesses.

CodeQL is a reliable choice for analyzing dynamic languages when considering precision, and Semgrep generally falls short. From the evaluation, the Joern-based data-flow analysis can identify the most vulnerabilities; however, this comes at the cost of additional false positives. While the user-defined semantics have been shown to reduce false positives without needing to rerun the analysis, the lack of precision for dynamic languages leaves room for future work.

For whole-program analysis, the Joern results reported fewer false negatives and, in some cases, fewer true positives. The cost of the whole analysis scaled the worst compared to the other candidates. However, it still ended up being the fastest tool in most cases. Compared to the partial-program analysis, a significant cost is incurred for a small precision gain, thus suggesting that the benefits of partial-program analysis outweigh the imprecision.

While memory is not only a direct result of the data-flow analysis, beyond the discrepancy of Joern performing worse in Java, the Joern-based approaches have a memory footprint far closer to that of Semgrep while being closer to CodeQL in precision. However, this figure suggests that CodeQL may scale better than Joern on sufficiently large programs.

Joern's precision for partial program analysis is serviceable and falls somewhere between Semgrep and CodeQL. The individual success of these tools supports the applicability of our work in real-world applications. The results, interpreted through the constraints of **RQ1** and **RQ2**, indicate that our tool effectively and efficiently performs partial-program static taint analysis across multiple programming languages. The results in all categories suggest that the answer to **RQ3** is that Joern is scalable enough for partial-program analysis of modern programs.

6 Conclusion

In response to the growing demand for performant vulnerability discovery in large systems, this paper presented a system capable of language-agnostic static taint analysis without direct access to external dependencies. By using simple annotations to model these dependencies, this system can answer taint analysis queries written with a high-level query language without having to re-analyze the dependencies.

References

[1] Sedick David Baker Effendi. 2025. *securibench-micro.js Dataset.* Stellenbosch University. doi:10.5281/zenodo.15396620

[2] Sedick David Baker Effendi. 2025. *Thorat Dataset.* Stellenbosch University. doi:10.5281/zenodo.15396362

[3] Sedick David Baker Effendi and Andrei Michael Dreyer. 2025. *Joern Benchmarks.* Joern Open-Source Community. doi:10.5281/zenodo.15396731

[4] Eric Bodden. 2012. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the International Workshop on State of the Art in Java Program analysis.* Association for Computing Machinery, New York, NY, USA, 3–8. doi:10.1145/2259051.2259052

[5] Ella Bounimova, Patrice Godefroid, and David Molnar. 2013. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proc. of the International Conference on Software Engineering (ICSE).* IEEE Press, San Francisco, CA, USA, 122–131. doi:10.1109/ICSE.2013.6606558

[6] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* 48, 09 (2021), 3280–3296. doi:10.1109/TSE.2021.3087402

[7] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1997. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 6 (1997), 992–1030. doi:10.1145/267959.269970

[8] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349. doi:10.1145/24039.24041

[9] Martin Fowler, Jim Highsmith, et al. 2001. The agile manifesto. *Software development* 9, 8 (2001), 28–35.

[10] GitHub, Inc. 2024. CodeQL (Version 2.19.2). https://codeql.github.com. Retrieved June 2024.

[11] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. 2011. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada). Association for Computing Machinery, New York, NY, USA, 177–187. doi:10.1145/2001420.2001442

[12] Bill Holz and Mike West. 2019. Results Summary: Agile in the Enterprise (Updated). https://circle.gartner.com/Portals/.../Summary%20(updated).pdf. Retrieved July 2021.

[13] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 1 (1990), 26–60. doi:10.1145/77606.77608

[14] Joern Community. 2024. Joern (Version 4.0.119). https://github.com/joernio/joern. Retrieved October 2024.

[15] Neil D Jones and Steven S Muchnick. 1979. Flow analysis and optimization of LISP-like structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* Association for Computing Machinery, San Antonio, Texas, 244–256. doi:10.1145/567752.567776

[16] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis.* Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055

[17] Soheil Khodayari and Giancarlo Pellegrino. 2021. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In *Proc. of USENIX Security Symposium.* USENIX Association, Vancouver, B.C., 2525–2542.

[18] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394. doi:10.1145/360248.360252

[19] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. 2018. Fuzz testing in practice: Obstacles and solutions. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE Computer Society, Los Alamitos, CA, USA, 562–566. doi:10.1109/SANER.2018.8330260

[20] Benjamin Livshits. 2006. Securibench Micro. https://github.com/too4words/securibench-micro. Retrieved May 2024.

[21] Vijay Krishna Palepu, Guoqing Xu, and James A Jones. 2017. Dynamic dependence summaries. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 4 (2017), 1–41. doi:10.1145/2968444

[22] Corina S Păsăreanu and Willem Visser. 2009. A survey of new trends in symbolic execution for software testing and analysis. *International journal on software tools for technology transfer* 11, 4 (2009), 339–353. doi:10.1007/s10009-009-0118-1

[23] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proc. of the Symposium on Principles of programming languages (POPL).* Association for Computing Machinery, New York, NY, USA, 49–61. doi:10.1145/199448.199462

[24] Oscar Rodriguez-Prieto, Alan Mycroft, and Francisco Ortin. 2020. An efficient and scalable platform for java source code analysis using overlaid graph representations. *IEEE Access* 8 (2020), 72239–72260. doi:10.1109/ACCESS.2020.2987631

[25] Semgrep, Inc. 2024. Semgrep (Version 1.95.0). https://semgrep.dev. Retrieved May 2024.

[26] Rajiv Thorat. 2022. Benchmark For Taint Analysis Tools Python. https://github.com/rajiv-thorat/benchmark-for-taint-analysis-tools-for-python. Retrieved May 2024.

[27] John Toman and Dan Grossman. 2017. Taming the static analysis beast. In *2nd Summit on Advances in Programming Languages (SNAPL 2017).* Schloss-Dagstuhl-Leibniz Zentrum für Informatik, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 18:1–18:14. doi:10.4230/LIPIcs.SNAPL.2017.18

[28] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering.* Association for Computing Machinery, New York, NY, USA, 1556–1560. doi:10.1145/3368089.3417943

[29] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proc. of IEEE Symposium on Security and Privacy.* IEEE Computer Society, Los Alamitos, CA, USA, 590–604. doi:10.1109/SP.2014.44

[30] William J Youden. 1950. Index for rating diagnostic tests. *Cancer* 3, 1 (1950), 32–35. doi:10.1002/1097-0142(1950)3:1<32::aid-cncr2820030106>3.0.co;2-3

Received 2025-03-02; accepted 2025-04-25

Pick Your Call Graphs Well: On Scaling IFDS-Based Data-Flow Analyses

Kadiray Karakaya
Heinz Nixdorf Institute
Paderborn University
Paderborn, Germany
kadiray.karakaya@upb.de

Palaniappan Muthuraman
Heinz Nixdorf Institute
Paderborn University
Paderborn, Germany
palaniappan.muthuraman@upb.de

Eric Bodden
Heinz Nixdorf Institute
Paderborn University & Fraunhofer IEM
Paderborn, Germany
eric.bodden@upb.de

Abstract

Recent works on scaling IFDS-based analyses propose sophisticated techniques ranging from sparsification and disk-assisted computing to intelligent garbage collection. Yet, they choose a fixed call graph, thereby disregarding its implications on scalability.

This work presents an empirical evaluation of call graph precision's impact on the precision and scalability of the IFDS framework. To this end, we build QCG, a call graph generation tool for Android that extends the Qilin pointer analysis framework, and integrate it with FlowDroid, a state-of-the-art IFDS-based taint analysis solver. We assess the precision of 27 call graphs built with QCG and 4 default call graphs in FlowDroid, on the TaintBench benchmark of Android malware. We then evaluate how increasing the call-graph precision impacts FlowDroid's runtime performance and memory consumption on real-world apps.

We report that the time invested in building precise context-sensitive call graphs pays off: They significantly reduce IFDS analyses' runtimes while also improving their precision. However, there appears to be a sweet spot in the trade-off between the call graph construction time and the reduction in total analysis runtime.

CCS Concepts: • **Software and its engineering → Automated static analysis**; *Software testing and debugging*.

Keywords: IFDS, Program Analysis, Data-flow Analysis, Call Graphs

ACM Reference Format:
Kadiray Karakaya, Palaniappan Muthuraman, and Eric Bodden. 2025. Pick Your Call Graphs Well: On Scaling IFDS-Based Data-Flow Analyses. In *Proceedings of the 14th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '25), June 16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3735544.3735587

This work is licensed under a Creative Commons Attribution 4.0 International License.

SOAP '25, Seoul, Republic of Korea
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1922-6/25/06
https://doi.org/10.1145/3735544.3735587

1 Introduction

IFDS (Interprocedural Finite Distributive Subset) [28] is a framework for solving flow- and context-sensitive interprocedural data-flow analysis problems, whose flow functions distribute over the meet operator in a finite domain. Many interesting static data-flow analysis problems can be represented as IFDS problems, including taint [2, 39], typestate [7, 32], and pointer analysis [33]. The IFDS framework reduces data-flow analysis problems to graph reachability on an exploded supergraph. This graph's nodes correspond to pairs of program statements and data-flow facts, and its edges correspond to statements' effects on the reachability of the data-flow facts. Data-flow facts are considered to hold at statements, if and only if they are reachable at those statements.

As previous works [3, 14] have shown, IFDS-based analyses can easily become unscalable when analyzing complex real-world programs. The scalability of the IFDS framework is bounded by (1) the runtime cost of propagating individual path edges and (2) the memory requirements to accommodate these edges for later reuse. Scalable IFDS extensions aim to either improve how these edges are computed [11, 16, 17], or how they are stored in the memory [1, 10, 20, 41]. They, however, often choose a fixed call-graph algorithm without considering its implications on how these edges are computed, and thus, how they are stored later on.

Precision and scalability are historically known to be competing objectives. Previous works on call graphs [8, 9, 26, 37] have shown that increased call-graph precision comes with an increased runtime cost. Others [5] have seen a trend where cheaper-to-compute call graphs waste time and precision in the later phases of the analysis, but so far there has been *no empirical evidence* that confirms this observation.

Figure 1 shows a motivating example of an IFDS-based taint analysis on an exploded supergraph. Here having a precise call graph not only reduces the workload of the IFDS algorithm but also improves its precision. Given the method `foo()` in Figure 1.a, `s` is being tainted by `source()` and passed to method `bar()`. Assuming the class hierarchy, where `C <: B <: A`, the method `bar()` in class `C` should be executed, which kills the taint, and therefore, there should not be any leaks. Using an imprecise call-graph algorithm,

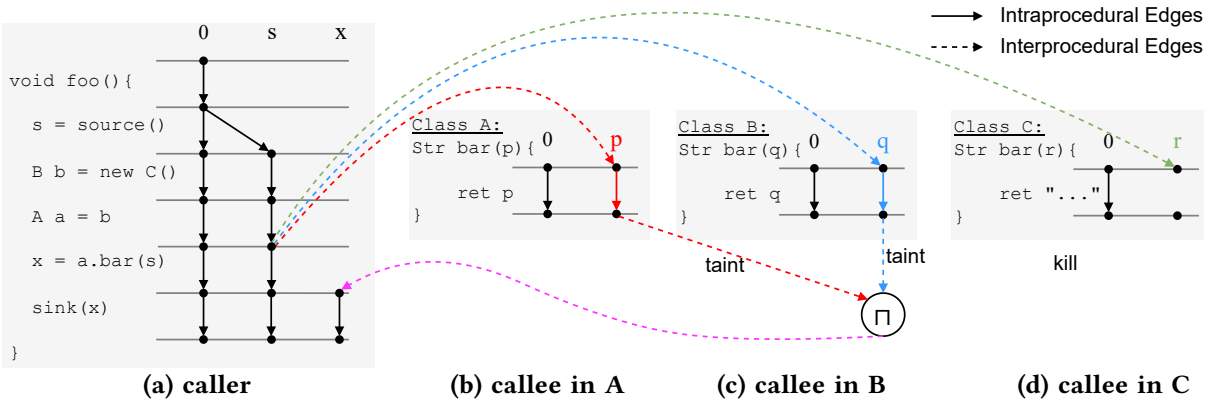


Figure 1. Call-graph precision's theoretical implications on the computation of the IFDS algorithm.[1]

for instance, CHA (Class Hierarchy Analysis) [6], would resolve to method `bar()` in all three classes. Consequently, the IFDS algorithm will map the data-flow fact `s` into all three methods and union their effects on `s`. This results in both an increased number of methods to process and an imprecise finding, as the taint analysis would *falsely* report a leak at `sink(x)`. We are therefore motivated to assess whether this theoretical implication holds in practice.

This work presents an empirical evaluation of call-graph precision's impact on the precision and scalability of the IFDS framework. To perform the experiments, we use FlowDroid [2], a state-of-the-art Android taint analysis, as a reference IFDS implementation. To obtain call graphs with varying degrees of precision, we extend Qilin [13], a state-of-the-art Java pointer analysis framework, in a tool called QCG. QCG enables leveraging Qilin's pointer analysis to obtain Android-compatible call graphs.

First, we sort call-graphs by their degrees of precision. To do so, we assess how they impact the precision and recall of FlowDroid on the TaintBench [24] benchmark of Android malware. Second, we evaluate how increasing the call-graph precision impacts FlowDroid's runtime performance and memory on a set of popular real-world apps.

To summarize, we present the following contributions:

- QCG, a Qilin-based call-graph generation tool for Android, which is open-sourced at Github,
- precision and recall of FlowDroid when using its 4 call graphs and 27 call graphs obtained by QCG, and
- an evaluation of the call graphs' impact on FlowDroid's scalability.

The remainder of the paper is organized as follows: Section 2 presents the related work. Section 3 introduces the empirical study design. Section 4 shows the evaluation results. Section 5 presents the available artifacts and Section 6 concludes this work.

[1]Interprocedural edges for the tautological fact 0 are omitted for brevity.

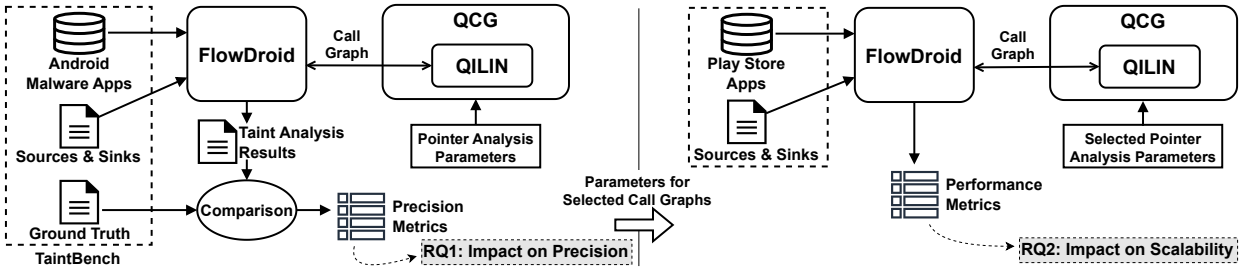
2 Related Work

Call graphs are indispensable data structures for interprocedural program analyses [8, 38]. CHA [6] and RTA [4] are classical examples of call-graph algorithms that do not rely on pointer information. Sundaresan et al. [34] show how pointer information can aid with virtual method resolution at polymorphic call sites. The SPARK framework [19] of Soot [40] provides cheap context-insensitive pointer information, which can be used for call-graph construction. Neupane and Thakur [27] measured the effect of FlowDroid's default call-graph algorithms on its precision. To the best of our knowledge, there is not an empirical study on call-graph precision's impact on the IFDS-based analyses.

Scaling IFDS-based analyses is an open challenge [3, 14]. Recently, many techniques were introduced to address this challenge. SparseDroid [11] sparsifies IFDS analyses, where intraprocedural edges are created not for every statement in a method but instead only for statements where data-flow facts are being used. Sparse IDE [17] generalizes this approach to the IDE framework [29]. DiskDroid [20] stores IFDS analysis data on disk when memory consumption reaches a threshold. DStream [41] scales IFDS analyses through a parallel streaming-based computation model. CleanDroid [1] improves the memory footprint of IFDS analyses by safely garbage-collecting obsolete intraprocedural edges. Interestingly, none of the scalable IFDS extensions mention which call graphs they use under the hood, except for CleanDroid, which uses FlowDroid's default context-insensitive call-graph based on SPARK [19]. We argue that all of these techniques would benefit from employing a more precise call graph which has an orthogonal impact on scalability.

3 Study Design

In this work, we measure the impact of call-graph precision on the precision and scalability of the IFDS analyses. To do so, as a preliminary step, we measure the precision of all the call graphs that we obtain by using various pointer



(a) Phase I: Collecting precision metrics **(b)** Phase II: Collecting performance metrics

Figure 2. Experimental Setup Overview

analysis techniques and FlowDroid's default call graphs. To summarize, our empirical study focuses on the following concrete research questions:

- RQ1: How does call-graph precision impact the precision of IFDS analyses?
- RQ2: How does call-graph precision impact the performance of IFDS analyses in terms of runtime and memory?

3.1 Experimental Setup

Figure 2 shows the overview of the experimental setup. The experiments are conducted in two phases. In *phase I*, we aim to answer *RQ1* and select a set of call graphs with highest precision. We then compare these precise call graphs with the default call graph options in FlowDroid in *phase II* to answer *RQ2*.

FlowDroid. FlowDroid [2] is one of the most widely used IFDS-based taint analyses. It soundly handles many Android-specific language features, as well as many precision dimensions: flow- and context-sensitivities are inherent properties of the IFDS framework. FlowDroid uses access paths as data-flow fact abstraction to be field-sensitive, and it resolves aliasing with its integrated on-demand alias analysis.

QCG and Qilin. We have built QCG to extend the Qilin pointer analysis framework for Java [13] to obtain Android compatible call graphs. Qilin builds on top of a parameterized pointer analysis kernel that can be configured with context-sensitivity parameters to use a set of state-of-the-art pointer analysis techniques. QCG ensures that Qilin-generated call graphs contain necessary methods for FlowDroid to soundly perform the taint analysis. We configure Qilin in each run with the parameters for a specific pointer analysis technique shown in Table 1, leading to 27 different call graphs. Additionally, we use FlowDroid's 4 internal Soot-based call-graph options for comparison. We conduct our experiments 31 different call graphs in total.

Table 1. Pointer Analysis Techniques in Qilin

Context-Sensitivity	Techniques
Method-Level	k-callsite-sensitivity (kCFA) [30], k-object-sensitivity (kOBJ) [25], k-type-sensitivity (kTYPE) [31], hybrid k-object-sensitivity H-kOBJ [18]
Fine-Grained	Bean [35], Mahjong [36], Zipper [21], Eagle [23], Turner [12], Data-Driven [15]

3.1.1 Phase I.

In this phase (Figure 2a), we evaluate the precision of FlowDroid when using different call graphs with varying degrees of precision. We use TaintBench as the analysis target. TaintBench contains 39 real-world Android malware applications with custom *sources & sinks* and a well-documented *ground truth*. In taint analysis, a source is a method that generates a taint. When a taint flows into a sink the analysis raises a warning, typically as a security vulnerability or a leak. The ground truth defines expected *taint-flows* that might occur between the sources and the sinks. We run FlowDroid with each of the 31 call-graph algorithms on TaintBench's 39 apps, in total we conduct $31x39$ runs. We compare the taint analysis results against the ground truth from TaintBench and compute the precision of FlowDroid with each call graph. In the end, we select a set of call-graph algorithms that lead to the most effective taint analysis results.

3.1.2 Phase II.

In this phase (Figure 2b), we use 20 popular Android applications from the Play Store (listed in Table 3) and FlowDroid's default source & sink definitions. The performance evaluations are conducted on real-world apps because, TaintBench apps are rather cheap to analyze and may not represent analysis performance on large-scale code bases.

We evaluate the call graphs' impact on the scalability of FlowDroid in terms of runtime and memory consumption. We run FlowDroid with 8 different call-graph algorithms, where 4 of them are the default algorithms in FlowDroid

and 4 of them are selected as the most precise algorithms in *Phase I*. The call graphs in FlowDroid represent a set of commonly used (cheap and rather imprecise) call graphs in literature, namely CHA (class hierarchy analysis) [6], RTA (rapid type analysis) [4], VTA (variable type analysis) [34], and SPARK [19]. Qilin-based call graphs represent a set of precise call graphs that utilize context-sensitive pointer analyses. In *Phase 2*, in total, we conduct 20*x*8 distinct runs and repeat each run 3 times to account for the noise on runtime and memory measurements. We perform the evaluations on a Linux virtual machine with 22 CPUs and 256 GB memory. Each run was given a budget of 5-hours and 220 GB memory.

4 Experimental Results

We next present the experimental results. We start with measuring the precision (RQ1) of the call graphs and their impact on IFDS' precision. Then we measure the call graphs' impact on IFDS' scalability (RQ2).

4.1 RQ1: Call-Graph's Impact on Precision

Table 2 shows the precision measurements of the call graphs and taint analysis measurements when using each call graph. Column 1 shows the call graphs. Column 2 shows the total number of call edges in each call graph created for all the apps in TaintBench. Column 3 shows the percentage of call edges in each call graph in comparison to those of the CHA-based, the least precise, call graph. Columns 4 and 5 show taint analysis findings as true positives and false positives respectively. Columns 6 and 7 are the taint analysis precision and recall respectively, and Column 8 is the F1 score.

Call-Graph Precision. We use the prefix *k* to refer to all the parameterized instantiations of a call graph. We use number of call edges as the precision metric for the call graphs. Our findings are in line with the experiments presented in previous work [12, 13, 21–23]. Eagle-based [23] call graphs (E-*k*obj) preserve the number of call-graph edges, compared to those of corresponding *k*obj-based call graphs. Mahjong [36] (M-*k*obj and M-*k*cfa) and Zipper [21] (Z-*k*obj and Z-*k*cfa) sometimes lose precision, i.e., contain more call-graph edges than other *k*obj- and *k*cfa-based call graphs. Turner-based call graphs (T-*k*obj) [12] contain either the same or slightly less number of call edges, compared to the Z-*k*obj-based call graphs. Selective context sensitivity-based call graphs (s-*k*cfa) [22] always contain either less or the same number of call edges, compared to the Z-*k*cfa-based call graphs.

Taint Analysis Precision. Considering the precision of the taint analysis, we observe that, generally, a more precise call graph *does* lead to a more precise taint analysis. Table 2 is first sorted by *precision* and then by *recall* of the taint analysis. The most precise call graphs by the number of call edges are underlined. Each of them helps taint analysis to yield more precise results. However, when considering the recall of the taint analysis, 2CFA- and M-2CFA-based call graphs cause a slight drop in recall. It appears that a more precise call graph does not always lead to more effective taint analysis results, one must also consider their impact on recall. Based on these results, we select the following Qilin-based call graphs, s-2cfa, Z-2obj, M-2obj, T-1obj, to be compared against the default call graphs in FlowDroid, i.e. CHA, RTA, VTA, SPARK in phase II.

A more precise call graph leads to a more precise taint analysis, but it does not guarantee better recall.

Table 2. Call graphs and their impact on FlowDroid's findings, sorted by descending precision and recall. The most precise call graphs are underlined. The call graphs that are selected for the evaluation in phase II are indicated in **bold**.

Call Graph	#Call Edges	Comp. to CHA	Taint Analysis				
			TP	FP	Precision	Recall	F1
s-2cfa	36 543	7.47%	45	8	0.85	0.22	0.35
Z-2obj	36 621	7.49%	45	8	0.85	0.22	0.35
M-2obj	36 660	7.50%	44	8	0.85	0.22	0.35
T-1obj	36 660	7.50%	43	8	0.84	0.21	0.34
2hyb	36 621	7.49%	43	8	0.84	0.21	0.34
2obj	36 621	7.49%	41	8	0.84	0.2	0.32
2type	36 621	7.49%	41	8	0.84	0.2	0.32
1cfa	36 660	7.50%	41	8	0.84	0.2	0.32
1hyb	36 660	7.50%	41	8	0.84	0.2	0.32
1hyb-type	36 699	7.51%	41	8	0.84	0.2	0.32
1obj	36 660	7.50%	41	8	0.84	0.2	0.32
1type	36 660	7.50%	41	8	0.84	0.2	0.32
2cfa	36 543	7.47%	41	8	0.84	0.2	0.32
2hyb-type	36 699	7.51%	41	8	0.84	0.2	0.32
D-2cfa	36 699	7.51%	41	8	0.84	0.2	0.32
D-2obj	36 699	7.51%	41	8	0.84	0.2	0.32
E-1obj	36 816	7.53%	41	8	0.84	0.2	0.32
E-2obj	36 738	7.51%	41	8	0.84	0.2	0.32
M-1cfa	36 660	7.50%	41	8	0.84	0.2	0.32
M-1obj	36 660	7.50%	41	8	0.84	0.2	0.32
M-2cfa	36 543	7.47%	41	8	0.84	0.2	0.32
B-2obj	36 621	7.49%	41	8	0.84	0.2	0.32
Z-1cfa	36 660	7.50%	41	8	0.84	0.2	0.32
Z-1obj	36 699	7.51%	41	8	0.84	0.2	0.32
Z-2cfa	36 582	7.48%	41	8	0.84	0.2	0.32
T-2obj	36 621	7.49%	41	8	0.84	0.2	0.32
s-1cfa	36 660	7.50%	41	8	0.84	0.2	0.32
SPARK	66 729	13.65%	42	10	0.81	0.21	0.33
VTA	137 904	28.20%	42	10	0.81	0.21	0.33
RTA	271 245	55.48%	42	10	0.81	0.21	0.33
CHA	488 943	100%	43	10	0.81	0.21	0.33

4.2 RQ2: Call-Graph's Impact on Scalability

Figure 3 shows average analysis runtime and memory consumption, when running FlowDroid with each call graph on 20 real-world apps.

Runtime. We observe that FlowDroid's runtime tends to *decrease* with increased call-graph precision. The least precise call-graph algorithm, CHA, leads to an average runtime of 3410 seconds, whereas the most precise algorithm, s-2cfa, leads to an average runtime of 2106 seconds (1.6x speedup). We observe, in particular, that all the algorithms with fine-grained context sensitivity, compared to FlowDroid's default call-graph algorithms, lead to a smaller analysis runtime. Yet, a more precise call graph does not always reduce the total analysis runtime: Figure 3 shows that although RTA, VTA and SPARK are more precise than CHA, they lead to a longer runtime than CHA. Similarly, M-2obj and Z-2obj lead to a slightly longer runtime than T-1obj.

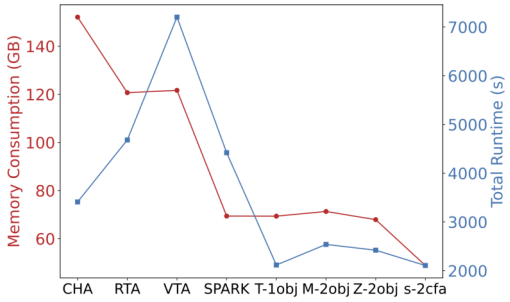


Figure 3. Average analysis runtime and memory consumption

Memory. FlowDroid's memory consumption tends to *decrease* with increased call-graph precision. The least precise call-graph algorithm, CHA, leads to an average memory consumption of 152 GB, whereas the most precise algorithm, s-2cfa, leads to an average of 48 GB (69% reduction). We observe a slight increase in memory consumption when switching from context-insensitive SPARK to algorithms with fine-grained context sensitivity. Call-graphs with context sensitivity are expected to increase the memory consumption, because they store each method context per a context identifier. However, surprisingly, overall memory consumption when using context-sensitive call graphs decrease substantially. We attribute this to the reduction in number of call-edges, which reduces IFDS computation and the space it requires to store the method summaries.

Trade-off. We observe a trade-off between the time invested in constructing a more precise call graph and the time saved during the downstream taint analysis. Figure 4 shows the relative time spent on constructing call graphs, and on taint analysis. We see that precise call-graphs are expensive to compute. For instance, the relative times spent on call-graph construction by RTA, VTA and SPARK are larger than

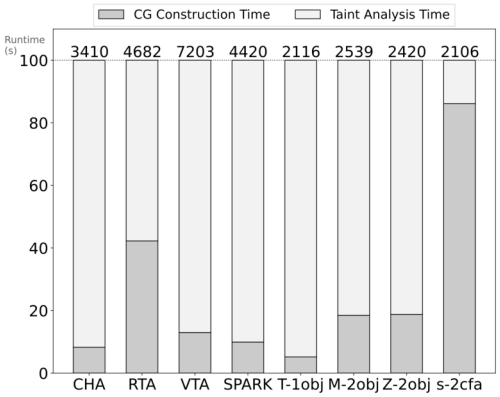


Figure 4. Relative time spent on taint analysis and CG construction (in %)

that of CHA. Similarly, the relative times spent on call-graph construction by M-2obj, Z-2obj and s-2cfa are larger than those of CHA, VTA and SPARK, and T-1obj has the least relative call-graph construction time. Among FlowDroid's context-insensitive call-graph algorithms, CHA has the least precision. Despite this, counterintuitively, a large number of interprocedural edges does not result in a blow-up in the IFDS analysis runtime. Among the call graphs based on fine-grained context-sensitive pointer analyses, T-1obj is the cheapest to compute but still precise enough that the total analysis time when using T-1obj is still less than the total analysis times when using M-2obj and Z-2obj, respectively. Figure 4 clearly shows that this is because of the larger portion of total time being spent on call-graph construction by M-2obj and Z-2obj. s-2cfa, on the other hand, shows the exemplary case, where despite a larger portion of time spent on call-graph construction, the speedup in the IFDS analysis runtime is so much that it pays off for the total analysis duration.

> *Time invested in building precise context-sensitive call graphs pays off, but a more precise call graph does not always improve scalability, as it might take longer to build.*

4.3 Discussion

Table 3 presents the full set of absolute numbers for runtime and propagation metrics collected in Phase II. It shows the number of data-flow fact propagations (*#P*), the number of interprocedural edges (*#IE*), and the analysis runtimes (as seconds) with each call-graph algorithm and on each app. Note that, due to time (5 hours) and memory (220GB) budget, some runs resulted in time-out exceptions (TOE) or out-of-memory errors (OOM). The runtimes of the runs with TOE are still included in the final results (as 5 hours), but the ones with OOM errors are not included in the final results, to prevent rewarding the early terminations with a shorter runtime.

Table 3. Runtime and propagation metrics of FlowDroid on 20 real-world apps, with each call graph

	CHA			RTA			VTA			SPARK			T-1obj			M-2obj			Z-2obj			s-2cfa		
APK	#P	#IE	time	#P	#IE	time	#P	#IE	time	#P	#IE	time	#P	#IE	time	#P	#IE	time	#P	#IE	time	#P	#IE	time
gkeyboard	5.0M	592K	3149	0	0	7957	3M	1.8K	3464	9M	16M	693	0	0	3	0	0	3	0	0	3	0	0	5
ucmobile	294M	2.7B	2372	-	-	17957	-	-	17957	58M	36M	1390	518M	106M	7315	546M	113M	14801	535M	111M	640	519M	106M	4993
gclock	340M	219M	324	367M	206M	428	353M	82M	237	351M	20M	366	282M	19M	346	286M	19M	353	312M	91M	12706	271M	18M	350
whatsapp	313M	2.4B	1189	322M	977M	1715	-	-	RE	12M	6M	494	11M	6M	483	12M	6M	528	11M	6M	487	10M	5M	538
garena	338M	3.3M	1317	376M	2B	2386	-	-	17957	-	-	OOM	88K	7.7K	9	97K	11K	10	90K	8.4K	9	95K	11K	210
shareme	293M	7B	1174	287M	5.8B	4871	287M	5.8B	4871	297M	59M	1305	47K	10K	29	47K	10K	30	3M	326K	34	47K	9.8K	553
mxplayer	289M	1.7B	2167	-	-	RE	-	-	RE	588M	151M	3192	455M	158M	3019	445M	141M	3118	461M	155M	3082	437M	136M	1064
shareit	286M	114M	3307	-	-	RE	-	-	OOM	-	-	17947	165M	35M	6522	169M	36M	7796	172M	37M	6142	166M	36M	1906
msword	294M	6.3B	1379	-	-	RE	-	-	RE	-	-	17949	3K	757	14	3K	760	25	3K	754	13	3K	754	231
tiktok	247M	808M	11242	-	-	OOM	-	-	OOM	377M	280M	5928	584M	337M	3689	724M	300M	8408	623M	329M	4309	-	-	17940
viber	-	-	17940	-	-	OOM	-	-	ER	333M	580M	1930	332M	510M	1963	361M	556M	2255	361M	556M	2255	-	-	17940
gfiles	249M	1.3B	17937	234M	1B	17937	-	-	17939	-	-	17941	-	-	17941	-	-	17941	-	-	17939	-	-	17940
webview	481M	4.6B	1644	31M	338M	389	28K	8.9K	14	343	57	7	343	51	4	343	51	4	343	51	4	343	51	20
netflix	333M	4.8B	1661	326M	2.8B	13540	359M	718M	1708	11M	3M	368	4M	1M	109	5M	1M	552	7M	2M	169	7M	2M	2152
mivideo	40M	7M	2608	46M	7M	2487	41M	7M	2993	30M	36M	1037	26M	17M	479	29M	19M	817	27M	19M	606	19M	16M	8509
zoho-show	585M	223M	290	375M	1.8B	4439	333M	1B	2089	415M	476M	1649	405M	484M	1500	387M	385M	1601	381M	375M	1456	400M	462M	1528
excel	299M	5.7B	1387	-	-	RE	-	-	RE	-	-	17941	3K	757	12	3K	758	21	3K	754	11	3K	757	179
fblite	11M	4M	65	11M	3M	67	1K	440	3	1K	412	2	1K	384	3	1K	383	2	1K	392	3	1K	384	4
candycrush	295M	5.5B	1193	307M	3.4B	2892	-	-	17940	6M	1M	342	1K	232	5	1K	231	5	1K	230	6	1K	231	22
gsearchlite	259M	1.6B	1337	239M	1B	6397	240M	1.1M	6752	229M	283M	1827	4M	37M	1058	4M	43M	1871	4M	42M	4746	4M	43M	2314

Out-of-memory errors are indicated with an OOM, runtime exceptions are indicated with a RE and time-out-errors are indicated in blue. For brevity, we use K for thousand, M for million and B for billion.

Also on some apps, analyses terminated prematurely with a runtime exception (RE). We observe that although CHA is imprecise, the taint analysis runs that use CHA almost always terminate, except for the app Viber. RTA terminates with a runtime exception when analyzing 4 apps and VTA terminates with a runtime exception on 5 apps. We find this behavior surprising because, in theory, they should prune an initial CHA-based call graph. These exceptions appear to be implementation bugs in the underlying Soot framework. Therefore, when using Soot-based call-graphs, the least precise CHA still seams to be a sound option. The analysis runs that use context-sensitive call-graph algorithms, i.e. T-1obj, M-2obj, Z-2obj and s-2cfa never terminated due to OOM, while only in few cases they terminated due to TOE.

A Call graph can theoreticallly impact the scalability of the IFDS framework primarily through the number of interprocedural edges it provides. Table 3 shows that, in general, when the number of interprocedural edges (*#IE*) increase, analysis runtimes increase as well. In the IFDS framework, by definition, an increased number of interprocedural edges will lead to an increased number of data-flow fact propagations, i.e. the tainted variables will be propagated through an increased number of method contexts. We observe that when the number of interprocedural edges (*#IE*) increase (i.e. when using a less precise call graph), the number of data-flow fact propagations (*#P*) increase as well. However, it is hard to conclude a general rule for the correlation between the number of interprocedural edges and propagations. The number of program statements, through which the data-flow facts are propagated, are unlikely be the same accross all different methods. Although we observe a clear correlation, it is hard to predict a call graph's exact impact on the scalability solely on the number of its call edges.

5 Artifact Availability

We have fully open-sourced QCG at https://github.com/secure-software-engineering/QCG. We have also prepared an artifact to run the complete evaluation pipeline presented in this work, which is available at https://doi.org/10.5281/zenodo.14898269.

6 Conclusion

This work presents an empirical study on call-graph precision's impact on the precision and scalability of IFDS-based data-flow analyses. Interprocedural data-flow analysis frameworks, such as IFDS, rely on call graphs to propagate data-flow facts across methods. We show, for the specific case of IFDS, that fewer call edges lead to an increase in data-flow analysis precision but they may also lead to a slight decrease in analysis recall. We also show that, in general, an increased call-graph precision pays off for the IFDS analyses in terms of memory consumption. In terms of runtime, there appears to be a sweet spot in the trade-off between the precision, and thus the construction time, of a call graph and the total analysis runtime. We, nonetheless, observe that fine-grained context-sensitive call graphs lead to significantly better IFDS analysis runtimes while also improving the analysis precision.

References

[1] Steven Arzt. 2021. Sustainable Solving: Reducing The Memory Footprint of IFDS-Based Data Flow Analyses Using Intelligent Garbage Collection. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1098–1110.

doi:10.1109/ICSE43902.2021.00102

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. doi:10.1145/2594291.2594299

[3] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining Apps for Abnormal Usage of Sensitive Data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 426–436. doi:10.1109/ICSE.2015.61

[4] David F. Bacon and Peter F. Sweeney. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Jose, California, USA) *(OOPSLA '96)*. Association for Computing Machinery, New York, NY, USA, 324–341. doi:10.1145/236337.236371

[5] Eric Bodden. 2018. The secret sauce in efficient and precise static analysis: the beauty of distributive, summary-based static analyses (and how to master them). In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, Netherlands) *(ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 85–93. doi:10.1145/3236454.3236500

[6] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 952)*, Walter G. Olthoff (Ed.). Springer, 77–101. doi:10.1007/3-540-49538-X_5

[7] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2, Article 9 (may 2008), 34 pages. doi:10.1145/1348250.1348255

[8] David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.* 23, 6 (2001), 685–746. doi:10.1145/506315.506316

[9] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call Graph Construction in Object-Oriented Languages. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1997, Atlanta, Georgia, October 5-9, 1997*, Mary E. S. Loomis, Toby Bloom, and A. Michael Berman (Eds.). ACM, 108–124. doi:10.1145/263698.264352

[10] Dongjie He, Yujiang Gui, Yaoqing Gao, and Jingling Xue. 2023. Reducing the Memory Footprint of IFDS-Based Data-Flow Analyses using Fine-Grained Garbage Collection. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 101–113. doi:10.1145/3597926.3598041

[11] Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. 2019. Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 267–279. doi:10.1109/ASE.2019.00034

[12] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. 2021. Accelerating Object-Sensitive Pointer Analysis by Exploiting Object Containment and Reachability. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:31. doi:10.4230/LIPICS.ECOOP.2021.16

[13] Dongjie He, Jingbo Lu, and Jingling Xue. 2022. Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:29. doi:10.4230/LIPIcs.ECOOP.2022.30

[14] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 106–117. doi:10.1145/2771783.2771803

[15] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 100 (Oct. 2017), 28 pages. doi:10.1145/3133924

[16] Kadiray Karakaya and Eric Bodden. 2023. Two Sparsification Strategies for Accelerating Demand-Driven Pointer Analysis. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, April 16-20, 2023*. IEEE, 305–316. doi:10.1109/ICST57152.2023.00036

[17] Kadiray Karakaya and Eric Bodden. 2024. Symbol-Specific Sparsification of Interprocedural Distributive Environment Problems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 104:1–104:12. doi:10.1145/3597503.3639092

[18] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 423–434. doi:10.1145/2491956.2462191

[19] Ondrej Lhoták. 2003. Spark: A flexible points-to analysis framework for Java. (2003).

[20] Haofeng Li, Haining Meng, Hengjie Zheng, Liqing Cao, Jie Lu, Lian Li, and Lin Gao. 2021. Scaling Up the IFDS Algorithm with Efficient Disk-Assisted Computing. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 236–247. doi:10.1109/CGO51591.2021.9370311

[21] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 141:1–141:29. doi:10.1145/3276511

[22] Jingbo Lu, Dongjie He, and Jingling Xue. 2021. Selective Context-Sensitivity for k-CFA with CFL-Reachability. In *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12913)*, Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi (Eds.). Springer, 261–285. doi:10.1007/978-3-030-88806-0_13

[23] Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 148:1–148:29. doi:10.1145/3360574

[24] Linghui Luo, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, and Fabio Massacci. 2022. TaintBench: Automatic real-world malware benchmarking of Android taint analyses. *Empir. Softw. Eng.* 27, 1 (2022), 16. doi:10.1007/S10664-021-10013-5

[25] Ana L. Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, Phyllis G. Frankl (Ed.). ACM, 1–11. doi:10.1145/566172.566174

[26] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S.-C. Lan. 1998. An Empirical Study of Static Call Graph Extractors. *ACM Trans. Softw. Eng. Methodol.* 7, 2 (1998), 158–191. doi:10.1145/279310.279314

[27] Prakash Neupane and Manas Thakur. 2023. Variational Study of the Impact of Call Graphs on Precision of Android Taint Analysis. In

Proceedings of the 16th Innovations in Software Engineering Conference (Allahabad, India) *(ISEC '23)*. Association for Computing Machinery, New York, NY, USA, Article 21, 5 pages. doi:10.1145/3578527.3578545

[28] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 49–61. doi:10.1145/199448.199462

[29] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.* 167, 1–2 (Oct. 1996), 131–170. doi:10.1016/0304-3975(96)00072-2

[30] Micha Sharir, Amir Pnueli, et al. 1978. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences ....

[31] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 17–30. doi:10.1145/1926385.1926390

[32] Johannes Späth, Karim Ali, and Eric Bodden. 2017. IDE^{al}: efficient and precise alias-aware dataflow analysis. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 99:1–99:27. doi:10.1145/3133923

[33] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:26. doi:10.4230/LIPICS.ECOOP.2016.22

[34] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical virtual method call resolution for Java. *SIGPLAN Not.* 35, 10 (Oct. 2000), 264–280. doi:10.1145/354222.353189

[35] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9837)*, Xavier Rival (Ed.). Springer, 489–510. doi:10.1007/978-3-662-53413-7_24

[36] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. *SIGPLAN Not.* 52, 6 (June 2017), 278–291. doi:10.1145/3140587.3062360

[37] Frank Tip and Jens Palsberg. 2000. Scalable propagation-based call graph construction algorithms. *SIGPLAN Not.* 35, 10 (oct 2000), 281–293. doi:10.1145/354222.353190

[38] Frank Tip and Jens Palsberg. 2000. Scalable propagation-based call graph construction algorithms. *SIGPLAN Not.* 35, 10 (Oct. 2000), 281–293. doi:10.1145/354222.353190

[39] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 87–97. doi:10.1145/1542476.1542486

[40] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: a Java bytecode optimization framework. In *CASCON First Decade High Impact Papers* (Toronto, Ontario, Canada) *(CASCON '10)*. IBM Corp., USA, 214–224. doi:10.1145/1925805.1925818

[41] Xizao Wang, Zhiqiang Zuo, Lei Bu, and Jianhua Zhao. 2023. DStream: A Streaming-Based Highly Parallel IFDS Framework. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2488–2500. doi:10.1109/ICSE48619.2023.00208

Received 2025-03-05; accepted 2025-04-25

Author Index