Two Approaches to Fast Bytecode Frontend for Static Analysis

CHENXI LI, Nanjing University, China
HAORAN LIN, Nanjing University, China
TIAN TAN*, Nanjing University, China
YUE LI*, Nanjing University, China

In static analysis frameworks for Java, the bytecode frontend serves as a critical component, transforming complex, stack-based Java bytecode into a more analyzable register-based, typed 3-address code representation. This transformation often significantly influences the overall performance of analysis frameworks, particularly when processing large-scale Java applications, rendering the efficiency of the bytecode frontend paramount for static analysis. However, the bytecode frontends of currently dominant Java static analysis frameworks, Soot and WALA, despite being time-tested and widely adopted, exhibit limitations in efficiency, hindering their ability to offer a better user experience.

To tackle efficiency issues, we introduce a new bytecode frontend. Typically, bytecode frontends consist of two key stages: (1) translating Java bytecode to untyped 3-address code, and (2) performing type inference on this code. For 3-address code translation, we identified common patterns in bytecode that enable more efficient processing than traditional methods. For type inference, we found that traditional algorithms often include redundant computations that hinder performance. Leveraging these insights, we propose two novel approaches: *pattern-aware 3-address code translation* and *pruning-based type inference*, which together form our new frontend and lead to significant efficiency improvements. Besides, our approach can also generate SSA IR, enhancing its usability for various static analysis techniques.

We implemented our new bytecode frontend in Tai-e, a recent state-of-the-art static analysis framework for Java, and evaluated its performance across a diverse set of Java applications. Experimental results demonstrate that our frontend significantly outperforms Soot, WALA, and SootUp (an overhaul of Soot)—in terms of efficiency, being on average 14.2×, 14.5×, and 75.2× faster than Soot, WALA, and SootUp, respectively. Moreover, additional experiments reveal that our frontend exhibits superior reliability in processing Java bytecode compared to these tools, thus providing a more robust foundation for Java static analysis.

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

Additional Key Words and Phrases: Java, Bytecode, Frontend, 3-Address Code, Type Inference, Static Analysis

ACM Reference Format:
Chenxi Li, Haoran Lin, Tian Tan, and Yue Li. 2025. Two Approaches to Fast Bytecode Frontend for Static Analysis. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 303 (October 2025), 27 pages. https://doi.org/10.1145/3763081

*Corresponding authors.

Authors' Contact Information: Chenxi Li, 502022330024@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Haoran Lin, 201250184@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Tian Tan, tiantan@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Yue Li, yueli@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China.

ACM 2475-1421/2025/10-ART303
https://doi.org/10.1145/3763081

1 Introduction

Static analysis, a technique used to compute specific properties of a program without executing it, has numerous applications across various domains, including bug detection [Ayewah et al. 2008; Cai et al. 2021; Li et al. 2024, 2021], security analysis [Chow et al. 2023; Liu et al. 2024; Zhong et al. 2022], compiler optimization [Møller and Veileborg 2020; Wimmer et al. 2024], and program comprehension [Li et al. 2016; Zhang 2024]. Its significant impact is reflected in its widespread adoption in both academia [Christakis and Bird 2016; Tahaei et al. 2021] and industry [Distefano et al. 2019; Wimmer et al. 2024]. Developing static analysis tools from scratch can be both challenging and time-consuming. To streamline this process, static analysis frameworks have been developed to offer fundamental and commonly used functionalities that facilitate the creation of these tools [Ayewah et al. 2007; Bravenboer and Smaragdakis 2009; Tan and Li 2023; Vallée-Rai et al. 1999; WALA 2006]. One critical component of a static analysis framework is the frontend, which transforms input programs into an intermediate representation (IR). This IR typically takes the form of a 3-address code with type information, making it suitable for static analysis.

This work centers on Java, renowned for its extensive ecosystem. Static analysis frameworks for Java typically employ frontends that process either Java source code or Java bytecode (hereafter referred to simply as "bytecode") as input. Our research specifically targets the development of a bytecode frontend, which offers several advantages:

(1) Bytecode is the target code, as Java applications must be compiled into it before execution.
(2) Compared to the rapidly evolving Java source code, bytecode maintains greater stability, thereby enhancing the maintainability of a bytecode-focused frontend.
(3) It enables the analysis of applications and libraries for which source code is unavailable.
(4) It facilitates the analysis of other JVM-based languages, such as Scala and Kotlin.

Given these benefits, our study concentrates on developing a bytecode frontend that exclusively processes bytecode as input. To further enhance the versatility, we have integrated `javac` into our frontend, enabling the compilation of Java source code to bytecode when necessary. This integration results in a comprehensive solution capable of handling both Java bytecode and source code, thus broadening the applicability of our frontend.

Since the frontend must process the input program before analysis can be performed, its efficiency has a significant impact on the user experience and the overall effectiveness of a static analysis tool.

- In scenarios where a large number of programs need to be analyzed within a restricted time budget, such as daily scans of updated applications in large software companies, or large-scale security analysis for app stores and library repositories, a faster frontend can save substantial time and allow for more resource-intensive subsequent analyses.
- For application developers, who are the primary users of static analysis tools, time cost is a key concern [Christakis and Bird 2016]. The duration for most commonly used lightweight analyses is often brief, sometimes even shorter than the frontend runtime, underscoring the importance of frontend efficiency.
- For static analysis developers, during the implementation and testing cycle of new analyses, the frontend is frequently invoked. In these situations, the runtime of the frontend is crucial and significantly influences productivity.

Despite the substantial demand for a highly efficient bytecode frontend, the current state of the art is not satisfactory. Soot [Vallée-Rai et al. 1999] and WALA [WALA 2006] are the two most popular and well-established static analysis frameworks for Java that provide comprehensive bytecode frontends, forming the basis for numerous research papers and tools in the past decades. Recently, an overhaul of Soot, called SootUp [Karakaya et al. 2024], has been released. However, the frontends of Soot, WALA, and SootUp are relatively time-consuming. As shown in Table 1, their

frontends take, on average, 60.23 seconds, 61.34 seconds, and 318.98 seconds, respectively, to build the IR for each real-world program in our experiments (each program averages 30,268 classes). Our aim was to reduce these times to just a few seconds, and we have succeeded in doing so.

Our Method. To address the efficiency limitations, we propose a new bytecode frontend. Conventional bytecode frontends, such as those employed in Soot and WALA (note that SootUp follows a similar design and algorithms to Soot, and is therefore not detailed here for brevity), typically adopt a two-stage approach: (1) translating bytecode to untyped 3-address code, and (2) performing type inference on the translated code. We identified performance bottlenecks in both stages of existing frontends and propose two new approaches, *pattern-aware 3-address code translation* and *pruning-based type inference*, to effectively address the efficiency challenges inherent in each stage.

(1) The core of 3-address code translation lies in converting stack-based bytecode, where data flows between variables are implicit, into register-based 3-address code with explicitly expressed data flows, thereby enhancing the analyzability of the input program. To achieve this explicitness, we need to resolve def-use relations in bytecode. Through careful study, we have identified two distinct patterns of def-use relations: *stack def-use* and *local def-use*. Unlike existing methods that treat both patterns uniformly, our pattern-aware 3-address code translation (detailed in Section 3) addresses each pattern distinctly. This innovation leads to significant performance improvements over the algorithms used by Soot and WALA.
(2) The core of type inference lies in resolving type constraints, specifically *define constraints* and *use constraints*. The state-of-the-art method [Bellamy et al. 2008] (employed in Soot) first resolves all typings that meet the *define* constraints, and then utilizes the *use* constraints to determine the desired typings. However, this approach results in redundant computations because most typings that satisfy the *define* constraints ultimately fail to satisfy all type constraints. In contrast, we propose pruning-based type inference (detailed in Section 4), which applies *use* constraints earlier to avoid generating useless typings and accelerate the process. This leads to a significant improvement, achieving linear time complexity compared to the exponential worst-case time complexity of the state-of-the-art approach.

The evaluation (Section 5) demonstrates that our new approaches significantly outperforms existing methods, achieving an order of magnitude improvement in efficiency. Furthermore, our frontend generates IR in both non-SSA and SSA forms, which not only enhances the versatility of our solution but also expands its applicability across a wide range of static analysis techniques.

In summary, this work offers the following contributions:

- We introduce two novel approaches, pattern-aware 3-address code translation (Section 3) and pruning-based type inference (Section 4), forming a highly efficient bytecode frontend. The pattern-aware translation speeds up by 15.0×, 14.9×, and 11.4× over Soot, WALA, and SootUp. The pruning-based type inference offers speedups of 7.6×, 11.0× and 613.3× over Soot, WALA, and SootUp, and features proven linear time complexity, vastly outperforming the state-of-the-art, which suffers from exponential worst-case time complexity.
- We implement our innovative bytecode frontend on top of Tai-e [Tan and Li 2023], a recent developer-friendly static analysis framework for Java. Our implementation, comprising 16,034 LoC, has undergone rigorous testing to ensure reliability and correctness.
- We conduct a comprehensive evaluation (Section 5) of our frontend against state-of-the-art alternatives across a diverse set of large programs, including the latest DaCapo benchmarks, various JDKs, real-world Java programs, and bytecode compiled from five JVM-based languages beyond Java. The results demonstrate that our frontend outperforms Soot, WALA, and SootUp by an order of magnitude, achieving overall average speedups of 14.2×, 14.5×,

and 75.2×, respectively. Moreover, extensive reliability experiments show that our frontend exhibits superior robustness in processing bytecode compared to competitors.
- We will submit an artifact to the Artifact Evaluation Committee (AEC) to reproduce all experimental results and make it publicly available. Moreover, we will fully open-source our frontend implementation, contributing accessible solutions to the static analysis community.

2 Bytecode to 3-Address Code Translation: Basics and Motivations

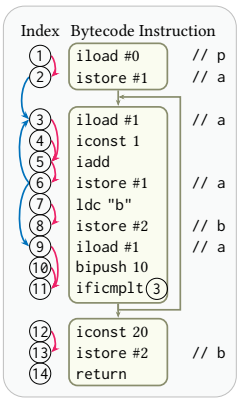
In this section, we present the foundational concepts necessary for understanding our pattern-aware translation from bytecode to 3-address code, along with the motivations behind our approach. Our detailed translation method will be introduced in Section 3.

2.1 Two Fundamental Challenges that Both Impede Bytecode Analysis and Complicating 3-Address Code Translation

To motivate our bytecode to 3-address code translation, we use a simple example in Fig. 1 to illustrate two challenges that make bytecode difficult to analyze and explain why converting to 3-address code eases analysis. These challenges not only impede bytecode analysis but also complicate the translation process itself, inspiring our innovative approach to bytecode conversion.

```
1 public void foo(int p) {
2   int a = p;
3   do {
4     a = a + 1;
5     String b = "b";
6   } while (a < 10)
7   int b = 20;
8 }
```

(a) Source code



(b) Bytecode with stack and local def-use relations marked by red and blue arrows.

```
1 void foo(int p) {
2   a = p;
3 label1:
4   a = a + 1;
5   b = "b";
6   if (a < 10) goto label1;
7   b = 20;
8   return;
9 }
```

(c) 3-address code.

Fig. 1. An example.

2.1.1 Challenge 1: Implicit Stack-Based Data Flows. As discussed in Section 1, Java bytecode's stack-based design results in data flows between variables being embedded in stack operations, making them implicit and challenging to analyze. To illustrate this concept, consider the example in Fig.1. Fig.1a shows a Java method `foo()` along with its compiled bytecode, while Fig.1c presents its corresponding 3-address code representation. Fig.1b highlights the def-use relations in the bytecode, which we will examine in detail later.

Consider how a simple assignment `a = p;` compiles into two bytecode instructions: `iload #0; istore #1;`. Here, `#0` and `#1` represent the slots for the variables `p` and `a` in bytecode, respectively. While there is a direct data flow from `p` to `a` in the source code, the bytecode implements this through sequential stack operations: first pushing `p`'s value onto the stack, then popping it into `a`. This means analyzing a single data flow requires tracking multiple instructions. This complexity compounds with more complex operations. For instance, the assignment `a = a + 1;` compiles into four bytecode instructions: `iload #1; iconst 1; iadd; istore #1;`. Here, tracing the data flow from the expression `a + 1` to `a` requires analyzing all four instructions.

In contrast, 3-address code makes these data flows explicit. Both `a = p;` and `a = a + 1;` directly express their data flows from source to destination. This clarity explains why Java static analysis frameworks typically translate bytecode to 3-address code before performing analysis.

2.1.2 Challenge 2: Reused Local Variable Slots. Bytecode frequently reuses a single local variable slot for multiple variables with non-overlapping lifespans. While this design reduces the size of the bytecode stack frame, it poses

significant challenges for 3-address code generation and analysis. In Fig.1a, there are two variables named `b`: one of type `String` in the block scope (line 5) and another of type `int` in the method scope (line 7). Despite being different variables with different types, in the compiled bytecode (Fig.1b), both share the same local variable slot #2.

Reusing local slots leads to precision loss in static analysis. For instance, if `x = new A()` and `x = new B()` appear in the code with `x` being reused, a flow-insensitive pointer analysis would incorrectly infer that variable `x` points to both objects `new A` and `new B` throughout all program points. In reality, `x` points to either `new A` or `new B` at different program points. This mismatch causes precision loss when analyzing the uses of `x` across different lifespans. This precision issue is why Java static analysis frameworks typically split reused local variable slots into distinct variables during the bytecode to 3-address code translation process.

2.1.3 Key to Challenges: Bytecode Def-Use Relations. The challenges described above not only make bytecode difficult to analyze, but also significantly complicate 3-address code translation. Bytecode's stack-based nature obscures direct variable relationships, requiring 3-address code translators to meticulously track values across multiple instructions to reconstruct these relationships in 3-address code. Additionally, the reuse of local variable slots creates tricky type inference problems. For example, in Fig. 1c, the variable `b` receives both a string assignment (`b = "b"`) and an integer assignment (`b = 20`), making it impossible to infer a single consistent type—a fundamental requirement for generating valid typed 3-address code.

We found that the key to addressing both challenges and effectively translating bytecode to 3-address code is to **resolve def-use relations in bytecode**. For handling implicit data flows, we must identify the data flows embedded in stack operations: specifically how values defined to the operand stack are subsequently used. For handling reused local variable slots, we need to determine the related definitions and uses of the same variable slot at different program points. To systematically address these issues, we introduce the concept of *bytecode def-use relations*.

Definition 2.1 (Bytecode Def-Use). A bytecode def-use relation exists between instructions i_1 and i_2 when i_1 produces a value that is subsequently consumed by i_2. In this case, we say i_1 is the definition of i_2, and i_2 is the use of i_1.

Importantly, we identify two distinct patterns in bytecode def-use relations: *stack def-use* and *local def-use*. In the remainder of this section, we first present these patterns (Section 2.2), then demonstrate how existing approaches suffer from efficiency issues due to their inadequate utilization of these patterns (Section 2.3). In Section 3, we present our pattern-aware translation method that leverages these patterns to efficiently convert bytecode to 3-address code.

2.2 Bytecode Def-Use Patterns: Stack Def-Use and Local Def-Use

Based on Definition 2.1, we observe that bytecode def-use relations fall into two distinct patterns: stack def-use and local def-use, categorized by how data flows from definition point to use point. These patterns exhibit different characteristics, and distinguishing between them is beneficial for 3-address code translation, as explained below.

In the bytecode execution model, each stack frame contains an operand stack and local variable slots [Lindholm et al. 2014] for storing program data. We identify two patterns of bytecode def-use relations: 1) A stack def-use occurs when a value is defined by being pushed onto the operand stack and later used when it is popped from the stack; 2) A local def-use occurs when a value is defined by being stored in a local variable slot and later used when it is loaded from this slot. These patterns are formally defined as follows:

Definition 2.2 (Stack Def-Use). A bytecode def-use relation exists between instructions i_1 and i_2 when i_1 produces a value on the operand stack that is subsequently consumed by i_2.

Definition 2.3 (Local Def-Use). A bytecode def-use relation exists between instructions i_1 (belonging to the `*store` family) and i_2 (belonging to the `*load` family) when i_1 stores a value in a local variable slot that is subsequently loaded by i_2 .

We have observed distinct behaviors between the two identified patterns of bytecode def-use relations. Firstly, when examining control-flow graphs (CFGs) in bytecode, we find that stack def-use relations typically have both the definition and use instructions residing within the same basic block. In contrast, local def-use relations often span across different basic blocks. For instance, in Fig. 1b, all stack def-use relations remain confined to the same basic block, whereas the local def-use relation from instruction ② to ③ extends beyond the boundaries of a single basic block.

Additionally, when considering the order of bytecode instructions within a method, we notice that for stack def-use relations, the order of definition instructions usually precedes that of the use instructions (def < use). This ordering allows them to be resolved in a single pass. However, local def-use relations frequently involve cases where the use instructions precede the definition instructions (use < def), creating cycles that necessitate revisiting basic blocks for resolution. As shown in Fig. 1b, when def < use, the arrow points downward, and all stack def-use relations follow this pattern. However, the local def-use relation from ⑥ to ③ exhibits the use < def pattern.

Due to these differences, resolving stack def-use relations is considerably easier than resolving local def-use relations. Therefore, we propose to address each pattern separately, as detailed in Section 3. In contrast, existing approaches treat both patterns uniformly, leading to inefficient 3-address code translation, as explained in Section 2.3.

2.3 Motivations: Limitations of State-of-the-Art 3-Address Code Translation

In this section, we examine major limitations in the 3-address code translation of state-of-the-art Java static analysis frameworks, Soot and WALA. We have identified that these limitations can be effectively addressed by distinguishing between and separately handling stack and local def-use relations. This distinction underpins our pattern-aware approach and highlights its advantages.

2.3.1 Inefficient Local Variable Splitting (Soot). As noted in Section 2.1.2, Soot addresses reused local variable slots by employing a *splitting* pass to separate variables sharing the same slots. For example, in the bytecode shown in Fig.1b, the slot #2 is reused for variable `b` at lines 5 and 7 in the source code. In the 3-address code generated by Soot, this single slot is split into two distinct variables `b1` and `b2`, as shown in Fig.2.

```
1 void foo(int p) {
2   a = p;
3 label1:
4   a = a + 1;
5   b1 = "b";
6   if (a < 10) goto label1;
7   b2 = 20;
8   return;
9 }
```

Fig. 2. 3-address code for Fig. 1c after splitting.

The primary issue with Soot's splitting algorithm [Vallée-Rai et al. 1999] is that it constructs def-use chains for all bytecode instructions, which is time-consuming. This problem pertains only to local def-use relations, making the computation of stack def-use relations unnecessary and inefficient. Our approach improves this process by excluding stack def-use relations from the splitting procedure, thereby enhancing efficiency.

2.3.2 Inefficient Fixed-Point Iteration (WALA). WALA translates bytecode into 3-address code via abstract interpretation. This interpretation computes symbolic representations of the operand stack and local variable slots at each program point, utilizing these representations to generate 3-address code, as further detailed in Section 3.3.1.

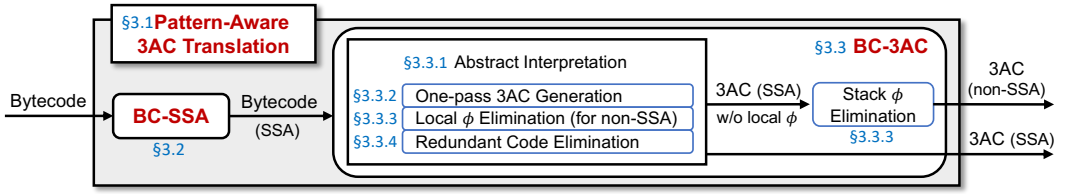


Fig. 3. Overview of our pattern-aware 3-address code translation.

As typical abstract interpretation, WALA's approach is iterative and may require revisiting certain instructions multiple times to reach a fixed point. Consider the bytecode program illustrated in Fig. 1b. Initially, WALA processes the instructions in sequence from ① to ⑪. Upon reaching instruction ⑪, it must then revisit one of its successors, specifically instruction ③, as depicted in the CFG of Fig. 1b. This revisitation is necessary due to the `istore` instructions at ⑥ and ⑧, which modify local variables. Consequently, the symbolic representation of the local variable at ③ requires updating. Once instruction ③ is revisited, all subsequent instructions from ③ to ⑪ must also be revisited. This repetitive revisitation leads to potential efficiency issues.

As shown in Fig. 1b, the need for revisiting arises from def-use relations following a use < def pattern. As discussed in Section 2.2, this pattern primarily occurs in local def-use relations. Our approach, while also based on abstract interpretation, tackles the iterative issue by distinguishing between local and stack def-use relations. By resolving local def-use relations prior to performing abstract interpretation, our approach can complete in one pass and avoid the need for revisitation.

3 Pattern-Aware 3-Address Code Translation

In this section, we present our pattern-aware method to 3-address code translation, which effectively manages both stack and local def-use relations to achieve efficient translation. Furthermore, our method can seamlessly generate both SSA and non-SSA forms of 3-address code.

3.1 Overview

Fig. 3 shows our pattern-aware translation process for converting bytecode into 3-address code, supporting both SSA and non-SSA outputs. This process comprises two phases: BC-SSA and BC-3AC. During the BC-SSA phase, the bytecode is transformed into SSA form, efficiently resolving local def-use relations. Following this, the BC-3AC phase applies abstract interpretation to generate 3-address code, addressing stack def-use relations. In this way, our pattern-aware translation applies different strategies to handle distinct def-use relations effectively. We will now provide a brief overview of BC-SSA and BC-3AC, highlighting the insights and advantages of our design. Detailed discussions of each are provided in Sections 3.2 and 3.3.

BC-SSA. We begin the translation from bytecode to 3-address code by converting it into SSA form. This design provides several benefits: (1) It effectively resolves local def-use relations by connecting each use of a local variable to its corresponding definition. (2) After transformation, each local variable—represented by a slot in bytecode—is defined exactly once, naturally addressing issues with reused local variable slots (as detailed in Section 2.1.2). (3) It establishes a solid foundation for generating SSA 3-address code if needed. Unlike traditional SSA transformation [Cooper et al. 2006], which involves virtually all 3-address code statements, BC-SSA simplifies the process by focusing solely on `*store` and `*load` bytecode instructions, thus enhancing efficiency. Overall, this efficient phase streamlines the subsequent translation to 3-address code.

BC-3AC. BC-3AC translates bytecode to 3-address code via abstract interpretation. This process resolves stack def-use relations and makes implicit stack-based data flows explicit, tackling the challenge described in Section 2.1.1. Thanks to our design, the abstract interpretation completes in *one pass*. Typically, BC-3AC need to resolve both local and stack def-use relations. While the numerous local def-use and few stack def-use relations usually require iterative resolution in abstract interpretation, our approach efficiently addresses these relations. The BC-SSA phase has already handled the local def-use relations, making it unnecessary to resolve in BC-3AC. Although few stack def-use relations require iterative resolution, we manage them using a straightforward over-approximation method, thus eliminating the need for iterations. As a result, our abstract interpretation is highly efficient, avoiding the iterations required by WALA (Section 2.3.2).

Besides, BC-3AC offers optional ϕ-function elimination to accommodate user preferences and supports both SSA and non-SSA outputs. It also employs a lightweight and effective optimization to reduce redundant code during translation, resulting in clean and compact 3-address code.

3.2 BC-SSA: Transforming Bytecode to SSA Form

In BC-SSA, we utilize a widely-used SSA transformation algorithm [Cooper et al. 2006], making minor modifications to adapt it specifically for transforming bytecode into SSA form. SSA transformation involves analyzing the definitions and uses of all variables to ensure each variable is assigned a unique version. Typically, SSA transformation applies to 3-address code, where local variables may be defined or used in virtually every kind of statement, necessitating a def-use analysis of almost all statements. In contrast, bytecode operates on a stack-based instruction set, where local variables can only be defined and used through two types of instructions: `*store` (for variable definitions) and `*load` (for variable uses). Most other instructions manipulate the operand stack rather than local variables, making these stack-manipulating instructions irrelevant to local def-use relations, and they can be ignored in BC-SSA. Specifically, BC-SSA splits a local variable v with multiple `*store` v instructions into distinct versions $v, v', v'', \ldots$, ensuring each version has at most one `*store`. ϕ-functions (analogous to standard SSA form) are inserted to connect these versions, while all `*load` v operations are updated to reference the correct version, preserving the def-use chain. Hereafter, we refer to the ϕ-functions inserted in BC-SSA as local ϕ-functions.

Therefore, we modify the algorithm from [Cooper et al. 2006] to process only `*store` and `*load` instructions, creating BC-SSA—a simpler and more efficient version than the original. Given the similarities of BC-SSA to the original algorithm, we omit a detailed description of BC-SSA here, but we will open-source our implementation with full details available.

3.3 BC-3AC: Translating Bytecode to 3-Address Code

This section introduces BC-3AC, our approach for translating bytecode into 3-address code. BC-3AC functions as an abstract interpretation, processing bytecode in SSA form (produced by BC-SSA) to generate 3-address code. As outlined in Section 3.1, BC-3AC is distinguished by several benefits stemming from its innovative design:

(1) It is highly efficient, achieving a fixed point in just *one pass*.
(2) It is capable of producing both SSA and non-SSA 3-address code.
(3) It features on-the-fly optimization to yield clean and compact 3-address code.

Although BC-3AC offers these significant advantages, its intricacy necessitates a comprehensive explanation. To facilitate understanding, this section is structured as follows:

- Section 3.3.1 introduces the basic abstract interpretation, which resolves stack def-use relations and converts bytecode in SSA form into SSA 3-address code.

$$
\begin{array}{lcll}
st = (s, \sigma) & \in & \text{State} = \text{Stack} \times \text{Locals} & \text{(Symbolic stack frame)} \\
s = [e, \dots] & \in & \text{Stack} = (\text{Exp})^* & \text{(Symbolic operand stack)} \\
\sigma = \{v \mapsto e, \dots\} & \in & \text{Locals} = \text{LVar} \to \text{Exp} & \text{(Symbolic local variable)}
\end{array}
$$

$$
\begin{array}{llll}
v & \in \text{LVar} & \text{(Local variables)} & \\
e & \in \text{Exp} & \text{(Expressions)} & \\
\texttt{instr} & \in \mathcal{I}_{bc} & \text{(Bytecode instructions)} & \\
v := e & \in \mathcal{I}_{3ac} & \text{(3-address code instructions)} &
\end{array}
\qquad
\begin{array}{lcll}
[\![\bullet]\!] & : & \text{State} \to \text{State} & \text{(Transfer function)} \\
\textsc{Gen} & : & (\mathcal{I}_{bc} \times \text{State}) \to (\mathcal{I}_{3ac})^* & \text{(Generation function)} \\
\sqcup_s & : & (\text{Stack} \times \text{Stack}) \to \text{Stack} & \text{(Stack join function)} \\
\sqcup_l & : & (\text{Locals} \times \text{Locals}) \to \text{Locals} & \text{(Local join function)}
\end{array}
$$

Fig. 4. Domains and notations of the abstract interpretation.

- Section 3.3.2 explains the methodology for ensuring that the abstract interpretation reaches a fixed point in one pass, significantly enhancing efficiency.
- Section 3.3.3 describes the elimination of ϕ-functions to produce non-SSA code if needed.
- Section 3.3.4 discusses the optimization techniques embedded in the abstract interpretation to preemptively avoid redundant code, resulting in clean and compact 3-address code.

3.3.1 Translating Bytecode to 3-Address Code via Abstract Interpretation. Translating bytecode to 3-address code involves resolving the implicit, stack-based data flows inherent in bytecode and making them explicit in the 3-address code, as discussed in Section 2.1.1. In bytecode, the execution model relies on an operand stack alongside a series of local slots to handle program data. In contrast, typical 3-address code uses local variables—also referred to as "registers"—to store all data explicitly.

To achieve this translation, temporary local variables must be introduced in the 3-address code to represent the elements in the bytecode's operand stack. Establishing the relationships (or constraints) among these variables is crucial for generating accurate 3-address code. For instance, consider the `iadd` instruction, which pops two entries from the stack and subsequently pushes their sum back onto the stack. In the corresponding 3-address code, we can introduce three variables: `t1` and `t2` represent the popped entries, while `t3` represents the pushed result. Based on the `iadd` instruction's semantics, we determine that `t3` is the sum of `t1` and `t2`. Consequently, we generate the 3-address code: `t3 = t1 + t2;`.

Inspired by [Lemerre 2023; WALA 2006], we use abstract interpretation to address the translation problem. This section details our basic abstract interpretation approach, similar to [Lemerre 2023; WALA 2006]. In Sections 3.3.2 through 3.3.4, we will present our innovative enhancements to this fundamental method that significantly improve the efficiency and effectiveness of our translation.

Domains and Notations For Abstract Interpretation. Abstract interpretation simulates the execution of each bytecode instruction to determine its abstract input and output states. Based on these states and the instruction's semantics, it then generates the corresponding 3-address code.

Fig. 4 illustrates the domains and notations used in abstract interpretation. The abstract state is defined as State = Stack × Locals, indicating that a state st is composed of two parts: a stack s and a map σ. The stack s is a list of expressions representing the symbolic values of the operand stack, denoted as $[e_1, e_2, \dots, e_n]$. Additionally, we use the notation $e : s'$ to denote a stack, where e is the top element and s' is the rest of the stack. The map σ associates each local variable, denoted by v, with its symbolic value, denoted by e. Essentially, the abstract state captures a symbolic stack frame corresponding to the concrete JVM state.

With these domains in place, we can introduce key functions for abstract interpretation. The function $[\![\bullet]\!]$ represents the transfer function in abstract interpretation. It processes a bytecode instruction, `instr`, along with an input state, st, to produce an output state, st', expressed as $[\![\texttt{instr}]\!](st) = st'$. The function Gen generates 3-address code by taking a bytecode instruction and an input state to produce the corresponding 3-address code instructions, ranging from zero

to several. During interpretation, to merge multiple incoming states at control-flow convergence points, we employ the join functions $\sqcup_s$ and $\sqcup_l$. Details of these functions are provided below.

Translating A Bytecode Instruction. To ease understanding, we begin by explaining how to translate a bytecode instruction given an input state. The translation process consists of two steps: (1) applying the *transfer function* ($[\![\bullet]\!]$) to compute the output state of the instruction from the input state, and (2) applying the *generation function* (Gen) to produce the corresponding 3-address code.

Java bytecode consists of over 200 distinct instructions, which can be grouped based on their semantics into three categories: *computation instructions*, *stack manipulation instructions*, and *local read-write instructions*. Instructions within the same category exhibit similar translation patterns, allowing us to illustrate the translation process using representative instructions from each category, rather than detailing all bytecode instructions, which would be both complex and tedious.

Computation instructions pop values from the operand stack, perform computations, and push the results back onto the stack. We use `iadd` as the representative instruction for this category. The transfer and generation functions for $\texttt{iadd}_i$ (the `iadd` instruction at index i) are defined as follows:

$$[\![\texttt{iadd}_i]\!](v : v' : s, \sigma) = (v_i : s, \sigma) \quad \textsc{Gen}(\texttt{iadd}_i, v : v' : s, \sigma) = [v_i := v' + v]$$

Here, v_i is a newly introduced variable for holding the value pushed onto the stack by instruction i. The transfer function captures the concrete behavior of `iadd`: it pops the top two values, v and v', from the stack, computes their sum, and subsequently pushes the result v_i back onto the stack. As `iadd` does not affect local variables, the local variable mapping σ remains unchanged. Additionally, the generation function produces the corresponding statement $v_i := v' + v$.

Stack manipulation instructions, such as dup, reorganize the stack's structure without computations. We define transfer and generation functions for dup as:

$$[\![\mathsf{dup}_i]\!](v : s, \sigma) = (v : v : s, \sigma) \quad \textsc{Gen}(\mathsf{dup}, v : s, \sigma) = [\,]$$

Here, the transfer function duplicates the top stack element v, while the generation function emits no 3-address code, as dup and all stack manipulation instructions involve no computation.

Local read-write instructions, specifically the `*load` family (`iload`, `aload`, etc.) and the `*store` family (`istore`, `astore`, etc.), operate on local variable slots. Given the distinct behaviors of the `*load` and `*store` instructions, `iload` and `istore` are chosen as representatives for each family. Their transfer and generation functions are defined as follows ($\sigma(v)$ denotes the value of v in σ):

$$[\![\texttt{istore}_i\ v]\!](v' : s, \sigma) = (s, \sigma\{v \mapsto v_i\}) \qquad \textsc{Gen}(\texttt{istore}_i\ v, v' : s, \sigma) = [v_i := v']$$
$$[\![\texttt{iload}_i\ v]\!](s, \sigma) = (v_i : s, \sigma) \qquad \textsc{Gen}(\texttt{iload}_i\ v, s, \sigma) = [v_i = \sigma(v)]$$

Although the transfer and generation functions are straightforward, the resulting code includes redundancies. We discuss our strategy to eliminate unnecessary code in Section 3.3.4.

Translating The Control-Flow Graph of A Method. Now, we present the translation process for an entire method. Typically, this involves building a control-flow graph (CFG) of the method to aid abstract interpretation. The translation of a CFG essentially requires repeatedly translating each bytecode instruction i within the CFG, computing the input and output states for i and generating the corresponding 3-address code, as previously described, until a fixed point is reached.

When translating a CFG, effectively managing control-flow convergence is crucial, as it requires computing the input state for the convergence instruction. To achieve this, we employ ϕ-functions from SSA form and define a join function, $\sqcup$. This function combines two expressions such that $e_1 \sqcup e_2 = e_1$ if $e_1 = e_2$; otherwise, it assigns the result to a fresh local variable, v', which holds the joined value. In scenarios where expressions e_1 and e_2 from distinct paths need convergence, the Gen function produces the statement $v' := \phi(e_1, e_2)$ to unify these expressions.

Building on the join function $\sqcup$, we define two specific join functions: $\sqcup_s$ and $\sqcup_l$. These functions serve to merge the stack and local variable slots of output states from different paths, respectively. For $\sqcup_s$, the function $\sqcup$ is applied element-wise to the stacks being merged:

$$(s_1 = e_1 : s_1') \sqcup_s (s_2 = e_2 : s_2') = e_1 \sqcup e_2 : (s_1' \sqcup_s s_2').$$

In the case of $\sqcup_l$, $\sqcup$ is applied to each variable slot existing in both σ_1 and σ_2 being joined:

$$(\sigma_1 = \{v \mapsto e_1, \dots\}) \sqcup_l (\sigma_2 = \{v \mapsto e_2, \dots\}) = \{v \mapsto e_1 \sqcup e_2, \dots\}.$$

Now, the join of two states, $st_1 = (s_1, \sigma_1)$ and $st_2 = (s_2, \sigma_2)$, can be expressed as $(s_1 \sqcup_s s_2, \sigma_1 \sqcup_l \sigma_2)$.

Utilizing the abstract interpretation defined by the transfer function $[\![\bullet]\!]$, along with the join functions $\sqcup_s$ and $\sqcup_l$, we align our approach with standard abstract interpretation frameworks. In the presence of loops within CFG, achieving a fixed point requires to merge all data flows. While WALA and [Lemerre 2023] achieve this through iterative fixed-point computation, our approach efficiently reaches the fixed point in a single pass, as detailed below.

3.3.2 Abstract Interpretation in One Pass. To enhance efficiency, our abstract interpretation is designed to reach a fixed point in a single pass, thereby avoiding iterative computations. Algorithm 1 gives the details of our one-pass 3-address code translation. It initializes σ, then traverses the CFG in reverse post-order. For each block, it computes the input stack, processes instructions, and generates IR by transfer and generation functions. Reaching a fixed point means that the stacks and locals are stable across all states in the CFG. Below, we explain how this is achieved in one pass.

For stacks processing, we traverse the CFG in reverse post-order, classifying basic blocks into two categories: normal blocks and loop headers. In normal blocks, all predecessor nodes are visited before the block itself due to the reverse post-order traversal, which allows us to directly apply $\sqcup_s$ to compute the input stack without the need for revisits. For loop headers, we adopt an over-approximation strategy for the joined input stack. Initially, each value in the input stack is assigned a fresh variable, denoted as $[v_h, v_{h-1}, \dots]$, where v_h represents the fresh variable at stack height h, similar to the approach in [Kotzmann et al. 2008]. We then emit ϕ-functions $v_h := \phi(\dots)$ for each v_h during the application of Gen to generate IRs for the loop header block. The variables in $\phi(\dots)$ are derived from all predecessors of the loop header block. This approach ensures that the joined stack corresponds to an over-approximation of input stacks in the abstract interpretation, thus negating the need to revisit the block. In practice, loop headers typically have empty input stacks; our experiments have revealed only a few instances of non-empty operand stacks at loop headers, primarily in programs compiled from JVM languages other than Java, such as Kotlin and Scala. Consequently, while this over-approximation may seem imprecise, it rarely impacts the overall efficacy in practical scenarios.

The processing of local variables is straightforward due to our design, which applies BC-SSA in advance. BC-SSA guarantees that for two local variable slots, σ_1 and σ_2, originating from different paths that converge, no variable v will satisfy $\sigma_1(v) \neq \sigma_2(v)$. Essentially, $\sigma_1(v) \neq \sigma_2(v)$ would imply multiple definitions for v, but BC-SSA preemptively resolves such conflicts by splitting them. Consequently, there is no need to join local variable slots at control-flow convergence points, as the joined locals remain identical to their pre-convergence states. This property eliminates the necessity for iterations to achieve a fixed point regarding local variables.

3.3.3 Non-SSA Generation. Our translation described in Sections 3.3.1 and 3.3.2 produces SSA 3-address code. While SSA facilitates static analysis, non-SSA code is often preferred for its readability

Algorithm 1: BC-3AC

Translate bytecode CFG to 3-address code

Input : cfg (Control flow graph of bytecode in SSA form)
Output: IR (Map from BC index to 3AC instructions)

1 $\sigma \leftarrow \{\}$, IR $\leftarrow \{\}$
2 **if** *output non-SSA IR* **then**
  /* Compute ϕ-web mapping, used by local ϕ-function elimination (Section 3.3.3) */
3 $\sigma \leftarrow$ Assign-ϕ-Webs(cfg)
4 **foreach** $bb \in$ cfg, *in reverse post order* **do**
  /* Compute input symbolic stack for bb */
5 **if** bb **is** *loop header* **then**
6 $s \leftarrow$ stack of fresh variables
7 **else**
8 **foreach** $bb' \to bb \in cfg$ **do**
9 Let s' be the output stack of bb'
10 $s \leftarrow s \sqcup_s s'$
  /* Abstract interpretation of bb */
11 **foreach** $\texttt{instr}_i \in bb$ **do**
  /* Single-use optimization, emit $v_i := e_i$ when needed */
12 **foreach** $e_j \in s$ **do**
13 **if** $e_j \in$ Invalid$(\texttt{instr}_i, s)$ **then**
14 $v_j \leftarrow$ Fallback(e_j)
15 Replace e_j with v_j in s
16 $(s, \sigma) \leftarrow [\![\texttt{instr}_i]\!](s, \sigma)$
17 IR$(i) \leftarrow$ Gen$(\texttt{instr}_i, s, \sigma)$
18 **if** bb *has two or more out edges* **then**
  /* Apply Fallback(e) to every expression e of s */
19 $s \leftarrow$ Fallback-Stack(s)
20 Store s to the output stack of bb
21 **return** IR

Algorithm 2: Fallback

Emit $v_i := e_i$ at instruction with index i

Input : e_i (Expression for instruction i)
Output: v_i (Variable for instruction i)

1 **if** e_i **is** *not* v_i **then**
2 IR$(i) \leftarrow v_i := e_i$
3 **return** v_i

$$[\![\texttt{iadd}_i]\!](v : v' : s, \sigma) = (v' + v : s, \sigma)$$
$$[\![\texttt{invokestatic}_i\ f]\!](v : s, \sigma) = (f(v) : s, \sigma)$$
$$[\![\texttt{dup}_i]\!](v : s, \sigma) = (v : v : s, \sigma)$$
$$[\![\texttt{iload}_i\ v]\!](s, \sigma) = (\sigma(v) : s, \sigma)$$
$$[\![\texttt{istore}_i\ v]\!](e : s, \sigma) = \begin{cases} (s, \sigma\{v \mapsto e\}) & v \text{ is single-def} \\ (s, \sigma) & \text{otherwise} \end{cases}$$

Gen$(\texttt{iadd}, s, \sigma) = [\,]$ Gen$(\texttt{invokestatic}, s, \sigma) = [\,]$
Gen$(\texttt{dup}, s, \sigma) = [\,]$ Gen$(\texttt{iload}\ v, s, \sigma) = [\,]$
$$\text{Gen}(\texttt{istore}\ v, e : s, \sigma) = \begin{cases} [\,] & v \text{ is single-def} \\ [\sigma(v) := e] & \text{otherwise} \end{cases}$$

Invalid$(\texttt{iadd}, e : e' : s, \sigma) = \{e, e'\}$
Invalid$(\texttt{invokestatic}\ f, e : s, \sigma)$
$= \{e\} \cup \{e \mid e \in s \wedge e \text{ has side effect}\}$
Invalid$(\texttt{dup}, e : s, \sigma) = \{e\}$
Invalid$(\texttt{iload}\ v, s, \sigma) = \emptyset$
Invalid$(\texttt{istore}\ v, e : s, \sigma) = \{\sigma(v)\}$

Fig. 5. Transfer, generation and invalidation functions used in redundant code elimination.

and closer resemblance to the source code, as it lacks ϕ-functions. To generate non-SSA code, BC-3AC needs to remove two types of ϕ-functions: (1) local ϕ-functions introduced during the BC-SSA phase, and (2) stack ϕ-functions generated during abstract interpretation.

Eliminating Local ϕ-Functions. Since local ϕ-functions are introduced during the BC-SSA phase, they can be directly eliminated during BC-3AC. Intuitively, standard ϕ-function elimination merges variables within the same ϕ-web into a single variable that receives multiple assignments. A ϕ-web [Briggs et al. 1998; Sreedhar et al. 1999] $w = \{v_1, v_2, \dots\}$ is a collection of variables connected through ϕ-functions in SSA form, conceptually representing what was originally a single variable prior to SSA conversion. In line with standard ϕ-function elimination, we first compute ϕ-webs. For each ϕ-web, we introduce a new variable v_w. This relationship is represented by symbolic local variables slot σ, where $\sigma(v) = v_w$ for all $v \in w$. The Assign-ϕ-Webs procedure in Algorithm 1 computes this σ as described [Briggs et al. 1998], and we omit its details here. We then modify the abstract interpretation rules as follows:

$$[\![\texttt{istore}_i\ v]\!](e : s, \sigma) = (s, \sigma) \qquad \text{Gen}(\texttt{istore}_i\ v, e : s, \sigma) = [\sigma(v) := e]$$
$$[\![\texttt{iload}_i\ v]\!](s, \sigma) = (v_i : s, \sigma) \qquad \text{Gen}(\texttt{iload}_i\ v, s, \sigma) = [v_i = \sigma(v)].$$

These rules directly use the σ computed by Assign-ϕ-Webs without altering it during abstract interpretation. Consequently, BC-3AC automatically eliminates local ϕ-functions during abstract interpretation, enabling the removal of local ϕ-functions once the interpretation finishes.

Eliminating Stack ϕ-Functions. To remove stack ϕ-functions, we add a pass following abstract interpretation. In this pass, for normal basic blocks, we apply the simple and efficient method from [Cytron et al. 1991] to eliminate stack ϕ-functions. For loop-header blocks, the more complex approach from [Boissinot et al. 2009] is adopted, as [Cytron et al. 1991] cannot correctly handle loop-header blocks. As mentioned in Section 3.3.2, stack ϕ-functions rarely appear in loop headers. This allows us to primarily employ the more space-efficient method [Cytron et al. 1991] for eliminating stack ϕ-functions, which results in code generation with less redundancy.

3.3.4 Redundant Code Elimination. Abstract interpretation often introduces temporary variables for intermediate results, which can lead to redundant code and variables, making the generated 3AC unnecessarily verbose. The left-most column of Fig. 6 illustrates this, where the temporary variable t holds the value of e, which is then assigned to v. We identify two redundancy cases from this example and describe their on-the-fly elimination during abstract interpretation.

Single-Def Variables. When $v := t$ is the sole definition for v, we can eliminate this statement and the variable by replacing occurrences of v with t in the code, as illustrated in the middle column of Fig. 6. In SSA output, all variables are defined once, inherently adhering to the single-def pattern due to the BC-SSA pass. For non-SSA output, we identify single-def variables by examining their ϕ-webs, noting when v is the only element in its ϕ-webs. If v has a single definition, we can optimize the translation rule for $\texttt{istore}_i\ v$ and omit the redundant $\sigma(v) := v_i$ statement, as illustrated in Fig. 5.

Redundant 3AC	Optimized (single-def)	Optimized (single-use)
$t := e$	$t := e$	
$v := t$		$v := e$
$\cdots$	$\cdots$	$\cdots$
$f(v)$	$f(t)$	$f(v)$

Fig. 6. An example for elimination.

Single-Use Variables. When $v := t$ is the sole use of t, we can optimize by inlining it with $t := e$ into $v := e$, removing t, as shown in the right-most column of Fig. 6. We employ this optimization optimistically, initially assuming all $t := e$ can be inlined, and reverting when inlining is infeasible. The Invalid function identifies scenarios where $t := e$ cannot be inlined, e.g., when t is used multiple times, or when inlining would emit incorrect 3AC. Fig. 5 gives Invalid definition for representative instructions. Upon interpreting $\texttt{instr}_i$, if $e \in \textsc{Invalid}(\texttt{instr}_i, s, \sigma)$, we use Fallback function (Algorithm 2) to emit $t := e$, falling back to non-inlined code to ensure correctness.

4 Pruning-Based Type Inference

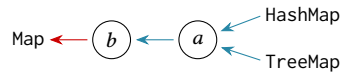
Our 3-address code translation (Section 3) produces 3-address code with untyped local variables. However, type information is vital for effective static analysis [Møller and Schwartzbach 2018]. In this section, we introduce our type inference algorithm, which surpasses current state-of-the-art methods in efficiency. We begin with a formal definition of the type inference problem (Section 4.1). Subsequently, we discuss the limitations of state-of-the-art algorithm [Bellamy et al. 2008] used by Soot (Section 4.2). We then present our innovative pruning-based type inference approach, designed to overcome these limitations and achieve linear time complexity (Section 4.3). Finally, we provide formal proofs for both the time complexity and correctness of our algorithm (Section 4.4).

4.1 The Type Inference Problem

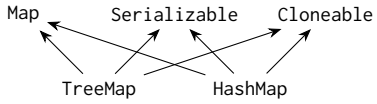
The type inference problem addressed here is essentially a constraint-solving task, aimed at finding a type mapping, or *typing*, that satisfies a set of *type constraints*. A typing $\mathcal{T}$ is a mapping from type variables to types, formally defined as $\mathcal{T} \in \text{TVar} \to \text{Type}$. Here, a type variable v ($\in$ TVar)

3-address code	Define Constraints ($\mathcal{D}$)	Use Constraints ($\mathcal{U}$)
`if (...) {`		
a := `new HashMap`	`HashMap` $\preccurlyeq a$	
`} else {`		
a := `new TreeMap`	`TreeMap` $\preccurlyeq a$	
`}`		
$b := a$	$a \preccurlyeq b$	
b.`Map#clear()`		$b \preccurlyeq$ `Map`

(a) An example of define and use constraints.



(b) Type constraint graph



(c) Type hierarchy

Fig. 7. A program in 3-address code with its type constraint and type constraint graph. `Map` is the abbreviation of `java.util.Map` and the same for `HashMap` and `TreeMap`. $a \preccurlyeq b$ represents "a is a subtype of b"

denotes the type of a local variable in the 3-address code, whereas a type T ($\in$ Type) signifies a concrete type within the program. The expression $\mathcal{T}(v) = T$ indicates that $\mathcal{T}$ assigns type T to the type variable v. To ease description, we also define $\mathcal{T}(T) = T$ for any type T.

The constraints originate from the 3-address code, and each constraint is of the form $t_1 \preccurlyeq t_2$, where t_1 and t_2 are either a type variable v or a type T. The notation $t_1 \preccurlyeq t_2$ denotes that "t_1 is a subtype of t_2". These constraints can be classified into two categories: define constraints ($\mathcal{D}$) and use constraints ($\mathcal{U}$). A define constraint for a variable v arises from a definition such as $v = \ldots$, while a use constraint is produced by a usage of v, such as in $f(v)$. For instance, an assignment statement $v_1 := v_2$ imposes the define constraint $v_2 \preccurlyeq v_1$. An invocation statement $v_1.f(v_2)$ generates two use constraints: $v_1 \preccurlyeq C$ and $v_2 \preccurlyeq A$, where C is the class type in which f is declared, and A is the parameter type of f. Fig. 7a illustrates an example of define and use constraints (middle and rightmost columns) generated from a segment of 3-address code (leftmost column).

We can now formally define type inference. A typing $\mathcal{T}$ is considered *def-valid* if it satisfies all define constraints, *use-valid* if it satisfies all use constraints, and *valid* if it is both def-valid and use-valid. Thus, the type inference problem is to determine a valid typing.

4.2 Motivation: Limitations of State-of-the-Art Type Inference

In this section, we briefly present the state-of-the-art type inference algorithm [Bellamy et al. 2008], which is also utilized by Soot, pertinent to the problem defined in Section 4.1. We also discuss its limitations, such as redundant computations and exponential worst-case complexity.

To ease the understanding of [Bellamy et al. 2008], we first introduce the core data structure of type inference: type constraint graph. The type constraint graph, denoted as G_T, is a graphical representation of type constraints. Its nodes represent type variables or types, and each edge $t_1 \longrightarrow t_2$ in G_T signifies the type constraint $t_1 \preccurlyeq t_2$. Importantly, we focus exclusively on acyclic type constraint graphs, as a cycle involving nodes $t_1, t_2, \ldots, t_n$ implies the constraints $t_1 \preccurlyeq t_2 \preccurlyeq \cdots \preccurlyeq t_n \preccurlyeq t_1$, leading to $t_1 = t_2 = \cdots = t_n$. These cycles can be collapsed, allowing attention to be directed towards the resulting acyclic graph. Fig. 7b presents an example of a type constraint graph, with define constraints shown as blue edges and use constraints as red edges.

To facilitate a clearer understanding of Soot's complex algorithm, we begin by outlining a simplified type inference algorithm, of which Soot's algorithm is a variation. In this approach, we construct the typing $\mathcal{T}$ by traversing the type constraint graph in topological order. For each type variable v, we establish its type T based on the types computed for its predecessors, and assign $\mathcal{T}(v) = T$. Consider an edge $t_i \longrightarrow v \in G_T$ as the ith incoming edge to v, with $\mathcal{T}(t_i) = T_i$. The type

T should satisfy the condition $\forall i \in [1, n], T_i \preccurlyeq T$. Accordingly, we determine T as the least common ancestor (LCA) of $T_1, \ldots, T_n$, denoted as $\mathcal{T}(v) = T = \texttt{LCA}(T_1, \ldots, T_n)$.

However, this simplistic algorithm falls short due to the fact that JVM bytecode allows interfaces to have multiple inheritance. As a result, $\texttt{LCA}(T_1, \ldots, T_n)$ may yield multiple types, making it impossible to deterministically ascertain the result type. To address this issue, Soot employs a more sophisticated approach by maintaining a set of typings $\Sigma = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n\}$ during the traversal of the type constraint graph. When processing a type variable v, Soot computes the LCA of its predecessor types for each typing $\mathcal{T}_i \in \Sigma$. If multiple LCAs exist, $\mathcal{T}_i$ is split into multiple new typings $\{\mathcal{T}_i\{v \mapsto T\} \mid T \in \texttt{LCA}(T_1, \ldots, T_n)\}$, which are then incorporated into Σ. For example, when processing the program in Fig. 7a, after processing node a, the typing set becomes $\Sigma = \{\{a \mapsto \texttt{Map}\}, \{a \mapsto \texttt{Serializable}\}, \{a \mapsto \texttt{Cloneable}\}\}$. This is because the LCA of a's predecessor types, $\{\texttt{TreeMap}, \texttt{HashMap}\}$, includes three elements, $\{\texttt{Map}, \texttt{Serializable}, \texttt{Cloneable}\}$, as per the type hierarchy depicted in Fig. 7c. The set Σ is derived by considering only define constraints, while use constraints (of the form $v \longrightarrow T$) are disregarded at this stage. Consequently, Σ computes all possible def-valid typings. Once Σ has been computed, Soot evaluates each typing $\mathcal{T} \in \Sigma$ to examine if it is use-valid. In cases where multiple use-valid typings are found, Soot non-deterministically selects one. For the case in Fig. 7, the sole use-valid typing is $\{a \mapsto \texttt{Map}, b \mapsto \texttt{Map}\}$. The remaining typings, $\{a \mapsto \texttt{Serializable}, b \mapsto \texttt{Serializable}\}$ and $\{a \mapsto \texttt{Cloneable}, b \mapsto \texttt{Cloneable}\}$, are filtered.

The limitations of Soot's state-of-the-art type inference algorithm become evident: (1) It involves numerous redundant computations, as many def-valid typings in Σ are ultimately not use-valid and are filtered out in the final step. Introducing early pruning could eliminate these use-invalid typings sooner. (2) The algorithm exhibits exponential worst-case time complexity because the size of Σ can grow exponentially with the number of variables. For example, consider a program with n variables a_i, each with $a_i :=$ `new TreeMap` and $a_i :=$ `new HashMap` assignment. When traversing each a_i, as $\texttt{LCA}(\texttt{TreeMap}, \texttt{HashMap})$ has three elements, each typing $\mathcal{T} \in \Sigma$ must be split into three. As a result, the size of Σ is tripled in each iteration. Thus, after the traversal, $|\Sigma| = 3^n$.

4.3 Our Type Inference: Pruning via Use Constraints

We now present our pruning-based type inference algorithm (Algorithm 3), which improves upon Soot's approach in two ways:

(1) **Pruning.** This is the core of our algorithm. While Soot filters out use-invalid typings after traversing the type constraint graph, we prune them early during the traversal. For instance, in Figure 7, types like `Serializable` and `Clonable` are clearly not use-valid for variables b and a. We prune these out from the `LCA` operation while traversing the graph.
(2) **Non-Deterministic Selection.** Even after performing pruning with `LCA`, multiple types may still remain, underscoring the NP-complete nature of the problem [Gagnon et al. 2000; Pratt and Tiuryn 1996]. To ensure linear time complexity and avoid inefficient exponential computations, we non-deterministically select a single type from the remaining options.

In this section, we first present our type inference algorithm, focusing on its core component: pruning via use constraints. We then explain how we guarantee that our algorithm generates valid typings for all programs, which we refer to as the algorithm's *correctness*.

Pruning Methodology in Algorithm 3. The key to effective pruning is the *use constraint set*. For each type variable v, the use constraint set of v, written as $\mathsf{C_u}(v)$, contains the type T from its use constraints $v \preccurlyeq T$. We define $\mathsf{C_u}(v)$ as the set of types T such that $v \preccurlyeq T \in \mathcal{U}$ or effectively $v \longrightarrow T \in G_T$. For example, in Figure 7, $\mathsf{C_u}(b) = \{\texttt{Map}\}$ and $\mathsf{C_u}(a) = \{\}$. Any use-valid typing $\mathcal{T}$ must satisfy $\mathcal{T}(v) \preccurlyeq T$ for all $T \in \mathsf{C_u}(v)$. Based on this, we define a "pruned `LCA`", an `LCA` with use-invalid

types excluded: $\texttt{LCA-u}(v, T_1, \ldots, T_n) = \{T \in \texttt{LCA}(T_1, \ldots, T_n) \mid \forall T' \in \mathsf{C_u}(v), T \preccurlyeq T'\}$. In Figure 7, $\texttt{LCA-u}(b, \texttt{TreeMap}, \texttt{HashMap}) = \{\texttt{Map}\}$, removing use-invalid types `Serializable` and `Clonable`.

The effectiveness of our pruning approach is directly linked to the size of $\mathsf{C_u}(v)$—the greater the number of use constraints, the more effectively we can eliminate use-invalid types. To maximize pruning capability, we introduce the *transitive use constraint set*, denoted as $\mathsf{C_u^*}(v)$, which extends $\mathsf{C_u}(v)$ by incorporating constraints from successor nodes in the type constraint graph. Specifically, if $T \in \mathsf{C_u}(v_1)$ and there is an edge $v_2 \to v_1$ in G_T, then T is included in $\mathsf{C_u^*}(v_2)$ through the transitive relationship $v_2 \preccurlyeq v_1 \preccurlyeq T$. This enhancement allows effective pruning even when direct constraints are absent. For example, while $\mathsf{C_u}(a) = \emptyset$ in our example, $\mathsf{C_u^*}(a)$ contains `Map` because of the transitive relationship $a \preccurlyeq b \preccurlyeq \texttt{Map}$.

Algorithm 3: Pruning-Based Type Inference

```
Input : G_T      (Type Constraint Graph)
Output: 𝒯        (Output Typing)
1  Merge all strong connected components of G_T.
   /* Backward propagation, build transitive use constraint set */
2  foreach v ∈ G_T, in reverse topo order do
3      C_u*(v) ← C_u(v)
4      foreach v → t ∈ G_T do
5          C_u*(v) = C_u*(v) ∪ C_u*(t)
   /* Main procedure, compute type for each node */
6  foreach v ∈ G_T, in topo order do
7      s ← ∅
8      for t → v ∈ G_T do
9          s ← s ∪ 𝒯(t)
10     c ← {T ∈ LCA(s) | ∀T' ∈ C_u*(n), T ≼ T'}
11     if c ≠ ∅ then
12         𝒯(v) ← T where T ∈ c
13     else
14         𝒯(v) ← java.lang.Object
15 return 𝒯
```

Now "pruned `LCA`" is defined as: $\texttt{LCA-u*}(v, T_1, \ldots, T_n) = \left\{T \in \texttt{LCA}(T_1, \ldots, T_n) \mid \forall T' \in \mathsf{C_u^*}(v), T \preccurlyeq T'\right\}$. This definition enables pruning when traversing a: $\texttt{LCA-u*}(a, \texttt{TreeMap}, \texttt{HashMap}) = \{\texttt{Map}\}$.

Algorithm 3 begins by constructing the transitive use constraint set through backward propagation of use constraints. It then traverses the type constraint graph forward to resolve the constraints. For each node, the algorithm computes the pruned `LCA` from the types of its predecessor nodes and selects one type from the resulting set. If no use-valid type is available, it defaults to `java.lang.Object` to ensure that the computed typing remains def-valid (as stated in Lemma 4.2).

Non-Deterministic Selection and Correctness. When multiple candidates are present in the output of the pruned `LCA` for a type variable v, Algorithm 3 (line 11) chooses one non-deterministically as the type of v. This approach can potentially introduce use-invalid types in the output typing, raising concerns about the algorithm's correctness. Essentially, these concerns are the trade-off for achieving a linear-time algorithm as it approximates an NP-complete problem. However, we assert the correctness of our algorithm through two key points: (1) In most real-world programs, non-deterministic selection does not result in use-invalid types. We demonstrate this by identifying two commonly observed bytecode patterns for variables. (2) If use-invalid types do appear in the final solution, we just insert casting instructions at use-sites to ensure the type validity of the output 3-address code, as advocated by [Bellamy et al. 2008; Gagnon et al. 2000].

To clarify why our algorithm guarantees valid typing outputs for real-world programs, we define two common bytecode patterns for variables: *definitely typed variables* and *trivially typed variables*. In the next section, we will prove Theorem 4.4, which states that if all variables in the program are either definitely or trivially typed, our algorithm will produce valid typing.

A variable v is *definitely typed* if and only if for any types $T_1, \ldots, T_n$, the set $c = \texttt{LCA-u*}(v, T_1, \ldots, T_n)$ contains at most one element. The more elements present in $\mathsf{C_u^*}(v)$, the more likely it is that v satisfies this property. Conversely, a variable v with zero or one element in $\mathsf{C_u^*}(v)$ is *trivially typed*. Intuitively, when Algorithm 3 processes a definitely typed variable v, the pruned `LCA` will yield at

most one candidate type. For a trivially typed variable v, if the pruned LCA includes two or more elements, non-deterministically selecting a type for v will not introduce any use-invalid types.

In real-world programs, most variables are either definitely or trivially typed. Although the definitely typed property might seem quite strong, it is typically satisfied by variables with multiple use constraints. This is because a non-definitely typed variable with multiple use constraints would necessitate a type hierarchy featuring multiple diamond inheritance patterns, a structure rarely encountered in practice.

4.4 Properties of Our Type Inference

In this section, we present two fundamental properties of our type inference algorithm (Algorithm 3). Theorem 4.1 addresses *efficiency*, demonstrating that our algorithm exhibits linear time complexity. Meanwhile, Theorem 4.4 proves *correctness*, showing that our algorithm produces valid typings when all variables in the program are either definitely or trivially typed.

Theorem 4.1. *Consider the type constraint graph G_T with V nodes and E edges. Let C be the cost of a single LCA query. Algorithm 3 runs in $O(V + E + C \cdot V)$ time.*

Proof. The SCC computation takes $O(V + E)$ time. During the backward propagation and main procedure, each node and edge of G_T is visited once in each phase, requiring $O(V + E)$ time. In the main procedure, the algorithm performs at most V LCA queries, taking $C \cdot V$ time. Thus, the total running time is $O(V + E + C \cdot V)$. □

Lemma 4.2. *For all input, the* TypeInference *algorithm will output a def-valid typing $\mathcal{T}$.*

Proof. The TypeInference algorithm will traverse each node of the constraint graph in topo order, yielding a traversal sequence $v_0 \dots v_n$. We prove the $\mathcal{T}_i$ is *partially* def-valid after traversing v_i. Partially def-valid means $\mathcal{T}_i$ is def-valid for the def constraints only consist the occurrence of $v_0 \dots v_i$. Trivial induction on the traversal sequence completes the proof. □

Lemma 4.3. *If a valid $\mathcal{T}$ exists, and all variable v is definitely typed, then the* TypeInference *algorithm will output a valid typing $\mathcal{T}$.*

Proof. By lemma 4.2, the $\mathcal{T}$ will be def-valid. We need to prove $\mathcal{T}$ is use-valid. We now prove the $\mathcal{T}_i$ is minimal, i.e., for all valid typing $\mathcal{T}'$, for $k \in [0, i]$, $\mathcal{T}_i(v_k) \preccurlyeq \mathcal{T}'(v_k)$. Induction on the traversal sequence, the empty case is trivial. For induction step, we need to show $\mathcal{T}_{i+1}(v_{i+1}) \preccurlyeq \mathcal{T}'(v_{i+1})$. First we prove c_{i+1} is a singleton, $c_{i+1} = \{T\}$. (1) By definitely typed property, c_{i+1} will have at most one element. (2) For all of the direct in edges v'_i of v_{i+1}, $\mathcal{T}_{i+1}(v'_i)$ is minimal, by property of LCA, for all valid typing $\mathcal{T}'$, $\mathtt{LCA}(\mathcal{T}'(v'_i)) \subseteq \mathtt{LCA}(\mathcal{T}_i(v'_i))$. It follows that $\bigcup_{\mathcal{T}'} \mathtt{LCA}(\mathcal{T}'(v'_i)) \subseteq \mathtt{LCA}(\mathcal{T}_i(v'_i))$. As $\bigcup_{\mathcal{T}'} \mathtt{LCA}(\mathcal{T}'(v'_i))$ will be all possible def-valid type set for v_{i+1}, if c_{i+1} is empty, there must be no possible type for v_{i+1} that is both def and use valid. So c_{i+1} is a singleton. As T will be the only possible def and use valid type for v_{i+1}, it follows $T = \mathcal{T}_{i+1}(v_{i+1}) \preccurlyeq \mathcal{T}'(v_{i+1})$. The definition of minimal entails the output $\mathcal{T} = \mathcal{T}_n$ is valid. □

Theorem 4.4. *If a valid $\mathcal{T}$ exists, and all variable v is either trivially typed or definitely typed, then our* TypeInference *algorithm will output a valid typing $\mathcal{T}$.*

Proof. Induction on the traversal sequence, the empty case is trivial. For induction step, we divide into 2 cases. (1) All transitive in-edge $v_i^* \to^* v_{i+1}$ of v_{i+1} is definitely typed. This case we can apply Lemma 4.3 to show $\mathcal{T}_{i+1}(v_{i+1})$ is both valid and minimal. (2) Exists at least one transitive in edge node v_i^* is trivially typed. Then v_{i+1} is trivially typed by T, by induction hyposis we have $\mathcal{T}_{i+1}(v'_i)$ is also valid, then $\forall t \in c_{i+1}, t \preccurlyeq T$. As $\mathcal{T}_{i+1}(v'_i) \preccurlyeq T$, we know c_{i+1} is not empty. Thus the algorithm will select a valid $\mathcal{T}_{i+1}(v_{i+1}) \in c_{i+1}$. □

5 Evaluation

We investigate the following research questions to evaluate our new bytecode frontend.

RQ1. How does our frontend's efficiency compare to that of state-of-the-art bytecode frontends?
RQ2. What is the efficiency of our 3-address code translation compared to state-of-the-art?
RQ3. How efficient is our type inference compared to state-of-the-art?
RQ4. How efficiently does our frontend generate 3-address code in SSA form?
RQ5. How reliable is our frontend in processing bytecode compared to state-of-the-art?
RQ6. How does our frontend affect the analysis results compared to state-of-the-art?

Implementation. Considering the usability of our new bytecode frontend, we implemented it on top of Tai-e, a state-of-the-art static analysis framework that offers a usage-friendly 3-address code IR, which facilitates the implementation of static analyses. In comparison to Soot and WALA, the APIs of Tai-e IR promote more concise code and clarify the underlying intents [Tan and Li 2023]. Our implementation encompasses bytecode to 3-address code translation (Section 3) and type inference (Section 4), amounting to 16,034 lines of thoroughly tested code. We will fully open-source our bytecode frontend and make the artifact for reproducing all experimental results publicly available, contributing our accessible solution to the static analysis community.

The Compared Bytecode Frontends. We compare our bytecode frontend to those of the two most popular and well-established static analysis frameworks for Java, Soot and WALA, as introduced in Section 1. Both frameworks are capable of converting bytecode to 3-address code for static analysis. To ensure a comprehensive evaluation, we also include SootUp, a complete overhaul of Soot and its true successor, released three years ago, in our comparison. We adopt the latest stable versions of all competitors: Soot (v4.6.0), WALA (v1.6.9), and SootUp (v1.3.0).

Benchmarks. To thoroughly evaluate the efficiency and reliability of our frontend against competitors across various conditions, we employ a large and diverse test suite with 1,032,502 classes and 8,861,806 methods, covering a wide range of Java and JVM-based bytecode. The benchmarks include: the latest DaCapo benchmarks [Blackburn et al. 2006] (its 2024 version), various JDK versions, and large-scale real-world programs written in Java and other JVM-based languages.

- **Latest DaCapo Benchmarks (2024)**: This suite includes 19 real-world applications and is widely recognized for Java program analysis benchmarking.
- **Java Development Kits (JDKs)**: Features standard libraries from latest Long-Term-Support versions Java 8, 11, 17, 21, and 6, prevalent across Java ecosystems and legacy systems.
- **Large Real-World Java Programs**: We incorporate a variety of large-scale applications, including IntelliJ IDEA (IDE), JRuby (Ruby interpreter), Apache Flink (stream processing framework), ElasticSearch (distributed system), and Ghidra (reverse engineering framework), providing coverage across diverse application domains.
- **Other JVM Languages**: We also include programs written in Scala, such as scalac (Scala compiler) and Apache Spark (data processing engine); Groovy, such as Gradle (build system); Kotlin, such as kotlinc (Kotlin compiler) and Compose (UI framework); and Clojure, such as Metabase (business intelligence). These benchmarks introduce non-Java bytecode, contributing unique patterns to examine efficiency and reliability across a wider range of scenarios.

Experiment Setup. All experiments are conducted on a machine featuring an AMD Ryzen 3.6GHz CPU, running on Java 21 and Ubuntu 24.04, with 16GB of memory allocated. This setup reflects a typical developer environment, offering practical performance insights.

5.1 RQ1: How Does Our Frontend's Efficiency Compare to State-of-the-Art Frontends?

The primary design goal of our new frontend is efficiency. To evaluate this, we compare our frontend's performance with the bytecode frontends of Soot, WALA, and SootUp by processing a comprehensive and diverse benchmark set previously introduced.

Despite being built upon different frameworks, these bytecode frontends are comparable. They share a two-stage design: first converting bytecode to untyped 3-address code and then typing this code to produce the IR. The Tai-e IR aligns directly with both Jimple (the IR of Soot and SootUp) and WALA IR, where each instruction maintains the same semantics across these representations. For instance, the `If` instruction in Tai-e IR corresponds directly to the `JIfStmt` in Jimple and the `SSAConditionalBranchInstruction` in WALA IR. Since these frontends process the same input (bytecode) and produce essentially the same output (IR), it is reasonable to compare them.

To ensure consistency in the output form, each frontend was set to its framework's default configuration. WALA outputs only SSA IR, whereas Soot and SootUp support both SSA and non-SSA forms, defaulting to the latter. Likewise, Tai-e, the foundation of our frontend, also defaults to non-SSA IR. Therefore, in comparison, our frontend, along with Soot and SootUp, all produce non-SSA IR. In Section 5.4, we will evaluate the efficiency of our frontend in generating SSA IR.

Table 1 presents the elapsed times for our frontend and others across all benchmarks in our experiments. It includes not only the total time for converting bytecode to IR but also the times for two specific stages of each frontend: 3-address code (3AC) translation and type inference. These stages in our frontend correspond to the methods approaches described in Sections 3 and 4. To ease straightforward comparison, Table 1 also features a bar chart depicting the average time taken by each frontend and its respective stages per benchmark.

As shown in Table 1, our frontend consistently records the fastest total time across all benchmarks, averaging 4.24 seconds. It outperforms the other frontends significantly, delivering notable speedups of 14.2× over Soot, 14.5× over WALA, and 75.2× over SootUp. Notably, both the 3AC translation and type inference of our frontend are also the quickest, averaging 3.78 and 0.45 seconds, respectively. Our frontend processes all benchmarks successfully, while some like SootUp encounter failures, indicated by "–" in Table 1. SootUp struggles with benchmarks that hit worst-case complexity in their typing algorithms, unable to complete even within 5 hours.

In the following research questions (Sections 5.2 and 5.3), we explore the performance of our 3AC translation (as detailed in Section 3) and type inference (as detailed in Section 4) compared to other frontends. We examine the factors contributing to the superior performance of our frontend.

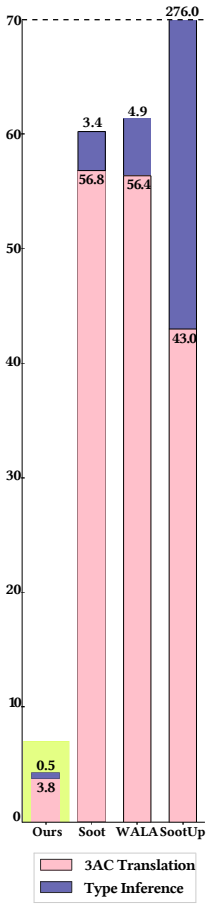
5.2 RQ2: How Does the Efficiency of Our 3-Address Code (3AC) Translation?

Table 1 illustrates that, for most frontends, the 3AC translation is the most time-consuming step in the entire IR building process, with the exception of SootUp, where type inference takes longer. In this section, we compare the efficiency of our 3AC translation (described in Section 3) with other frontends and explore the main factors contributing to the observed differences.

Our 3AC Translation vs. Soot. Our 3AC translation achieves a significant average speedup of 15.0× over Soot. Soot's translation process operates as a pipeline, starting with a basic conversion from bytecode to 3-address code, followed by several transformation passes. We identified two main factors contributing to the performance gap: First, Soot's translation involves two additional passes—local splitting and shared initializer local splitting—to split local variables for typing (see Section 2.3.1), which account for 51% of the total translation time. Second, the basic conversion produces 3-address code with redundancies, necessitating extensive subsequent optimization passes, such as copy propagation and dead code elimination, which consume 39% of the translation time.

Table 1. Efficiency results of evaluated frontends. "#Classes" represents the number of classes in each benchmark. The table details the **elapsed time** for the entire IR building process ("Total"), 3-address code translation ("3AC Translation"), and type inference ("Type Inference"), with time measured in seconds. A "–" indicates a frontend failure on a benchmark. The bar chart depicts the average time per benchmark.

Program	#Classes	Total				3AC Translation				Type Inference			
		Ours	Soot	WALA	SootUp	**Ours**	Soot	WALA	SootUp	**Ours**	Soot	WALA	SootUp
compose	24316	3.21	27.42	21.14	92.98	2.92	26.23	18.98	31.07	0.28	1.19	2.16	61.92
D-avrora	1864	0.11	2.23	0.79	3.50	0.10	2.10	0.63	0.94	0.01	0.13	0.16	2.56
D-batik	3218	0.36	6.11	2.57	14.34	0.32	5.77	2.11	4.38	0.03	0.34	0.46	9.96
D-biojava	7628	1.00	18.89	11.18	49.96	0.90	18.08	9.67	11.42	0.10	0.81	1.50	38.55
D-cassandra	53017	7.87	96.90	77.80	414.48	7.08	91.61	70.50	66.44	0.78	5.30	7.31	348.04
D-eclipse	7376	1.52	25.76	11.05	45.84	1.35	24.52	8.92	15.09	0.17	1.24	2.13	30.75
D-fop	10767	1.12	21.70	12.53	73.12	0.98	20.40	10.68	13.63	0.13	1.30	1.84	59.49
D-graphchi	12896	2.14	44.23	16.94	99.07	1.87	41.02	14.59	21.24	0.27	3.21	2.35	77.82
D-h2	4914	1.11	26.16	10.61	121.21	0.97	25.12	8.91	16.50	0.14	1.03	1.70	104.71
D-h2o	58740	9.09	115.27	102.14	833.89	8.22	108.43	92.78	133.30	0.87	6.84	9.36	700.60
D-jme	13211	1.67	29.18	16.55	69.35	1.48	27.82	14.08	16.04	0.19	1.36	2.48	53.31
D-jython	13866	1.56	26.03	14.78	105.72	1.30	24.63	12.88	14.39	0.26	1.40	1.91	91.33
D-kafka	21512	3.53	65.32	33.60	–	3.17	61.40	30.25	–	0.36	3.92	3.35	–
D-luindex	2873	0.35	6.05	2.31	11.50	0.30	5.72	1.95	3.39	0.05	0.33	0.36	8.11
D-lusearch	2875	0.31	6.16	2.37	11.79	0.27	5.82	2.00	3.36	0.04	0.34	0.37	8.43
D-pmd	4564	0.53	11.84	4.79	21.85	0.46	11.20	4.13	7.13	0.07	0.63	0.66	14.72
D-spring	30469	4.37	50.45	46.19	120.12	3.95	48.08	42.37	26.49	0.42	2.38	3.83	93.63
D-sunflow	657	0.08	2.11	0.64	4.51	0.06	1.99	0.54	1.08	0.01	0.12	0.11	3.43
D-tomcat	4110	0.79	15.19	7.09	32.33	0.68	14.42	6.07	9.02	0.10	0.77	1.02	23.31
D-xalan	2791	0.42	7.63	4.15	27.68	0.35	7.22	3.55	4.74	0.06	0.42	0.60	22.94
D-zxing	716	0.11	3.25	1.05	6.74	0.09	3.03	0.88	2.49	0.01	0.22	0.18	4.25
elastic	77232	10.90	162.49	171.63	524.69	9.84	155.43	160.71	96.18	1.06	7.07	10.92	428.51
flink	48708	7.14	96.30	72.15	234.44	6.41	86.55	64.75	48.67	0.73	9.75	7.41	185.76
frege	3471	0.52	10.06	4.87	30.33	0.47	9.60	4.14	7.16	0.05	0.46	0.72	23.17
ghidra	72733	11.25	147.72	137.20	–	10.17	139.16	126.27	–	1.08	8.56	10.93	–
gradle	80131	9.93	115.07	137.65	558.69	8.98	108.84	127.13	79.91	0.95	6.23	10.52	478.78
intellij	132977	16.52	221.16	328.80	931.84	14.71	210.50	300.26	134.31	1.82	10.67	28.53	797.53
jre11	29960	4.78	75.32	70.86	817.05	4.22	70.93	65.79	92.05	0.56	4.39	5.07	725.00
jre17	26170	5.43	70.05	67.79	835.19	4.58	65.90	63.33	88.12	0.85	4.15	4.46	747.07
jre21	27684	5.46	76.34	87.61	1049.48	4.75	71.75	81.86	105.59	0.72	4.59	5.75	943.90
jre6	18513	2.36	47.60	22.27	128.79	2.07	45.34	18.84	37.91	0.29	2.26	3.43	90.89
jre8	20792	2.59	51.29	24.52	119.76	2.27	48.92	21.31	28.08	0.32	2.36	3.21	91.68
jruby	9230	0.92	16.64	7.54	129.88	0.81	15.65	6.44	56.20	0.11	0.99	1.09	73.68
kotlin	33300	3.88	70.27	32.18	175.49	3.49	65.95	27.54	31.95	0.40	4.33	4.64	143.53
metabase	94799	11.86	165.37	222.93	1956.48	10.51	157.16	210.20	133.23	1.35	8.20	12.73	1823.24
scala3	7931	1.26	23.86	8.97	60.93	1.14	22.33	7.36	10.51	0.12	1.52	1.60	50.42
spark	124134	20.77	271.24	472.39	2089.23	18.75	253.55	444.35	239.24	2.02	17.69	28.04	1849.99
Average	29463	**4.24**	60.23	61.34	318.98	**3.78**	56.82	56.40	43.01	**0.45**	3.42	4.94	275.97



In Section 2.3.1, we explain that distinguishing between stack and local def-use shows splitting is only necessary for local def-use. Our approach applies splitting exclusively to local def-use, unlike Soot, which does so for both types. This avoids Soot's redundant computations on processing unnecessary stack def-use. In addition, as described in Section 3.3.4, unlike Soot which needs extensive optimizations due to its conversion's constraints, we adopt lightweight on-the-fly optimizations to prevent code redundancy. Together with our splitting method, these factors eliminate the need for unnecessary and time-consuming passes, making our 3AC translation significantly faster.

Our 3AC Translation vs. WALA. Our 3AC translation achieves a considerable average speedup of 14.9× over WALA. The inefficiencies associated with WALA mainly stem from its iterative approach to abstract interpretation, which necessitates repeatedly visiting bytecode instructions, as explained in Section 2.3.2. Additionally, WALA incurs overhead by attempting to optimize iteration speed, such as by ordering bytecode instructions for processing during iterative computation.

Conversely, due to our distinction between stack and local def-use, we found that only local def-use needs iterative processing. This allows us to first use a highly efficient way to preprocess local def-use, and then finish the abstract interpretation in one pass, as explained in Section 3.3.2.

Our 3AC Translation vs. SootUp. Our 3AC translation delivers a notable average speedup of 11.4× over SootUp. As described in Section 1, SootUp employs design principles similar to Soot for 3AC translation, leading to slower performance than our approach. So we omit further discussion here.

5.3 RQ3: How Does the Efficiency of Our Type Inference?

Table 1 shows that our type inference (described in Section 4) is considerably faster than other frontends. Below, we explore the primary factors contributing to this performance advantage.

Our Type Inference vs. Soot. Our type inference achieves a 7.6× speedup over Soot, primarily due to differences in algorithmic approach. Our analysis reveals that computing all def-valid typings alone accounts for 43.3% of Soot's type inference time. This process introduces significant overhead, as it generates numerous def-valid typings that require extensive filtering with use constraints, consuming an additional 26.6% of the inference time. Combined, these operations constitute about 70% of the total inference time, thus forming the primary performance bottleneck.

In contrast, our pruning-based approach enhances efficiency by preemptively eliminating use-invalid types during the typing process, thereby avoiding the need to compute all def-valid typings.

Our Type Inference vs. WALA. Our type inference achieves a 11.0× speedup over WALA. Notably, WALA's type inference is weaker than Soot's and ours, as it does not ensure the validity of inferred types. This shortcoming arises from its oversight of use constraints and its avoidance of the core challenge in type inference: handling multiple inheritance for interfaces. Instead, when performing type inference, it simply computes the least common ancestor (LCA) as the superclass of two types, defaulting to `Object` for any two interfaces regardless of their inheritance. Despite WALA's seemingly faster type inference due to its highly conservative approach, we found that it remains less efficient than ours, even when addressing significantly simpler problems. This performance disparity arises because WALA's type inference relies on its built-in general-purpose constraint solver and lacks an algorithm specifically designed for type inference tasks.

Our Type Inference vs. SootUp. Our type inference operates hundreds of times faster than SootUp's, which is exceptionally slow. Upon investigation, we found that while SootUp uses the same type inference algorithm [Bellamy et al. 2008] as Soot, its implementation is less efficient. We conducted further profiling of SootUp and identified two major performance bottlenecks. First, over 80% of the execution time is consumed by a post-processing step that replaces untyped variables with their typed counterparts after the main type inference. Second, the main type inference process itself is notably slow, being 15× slower than Soot and 120× slower than ours. A major factor contributing to this inefficiency is SootUp's implementation of subtype checking, which determines if type A is a subtype of type B. While essential for type inference, this implementation is significantly less efficient than both ours and Soot's.

5.4 RQ4: How Efficient is Our Frontend in Generating SSA 3-Address Code?

SSA form is an important IR for static analysis, offering benefits like simplifying data flow analysis and increasing analysis precision. As noted in Section 3, our frontend is able to convert bytecode into this essential IR. Here, we evaluate the efficiency of SSA generation.

Table 2 presents the elapsed time for SSA generation by our frontend, divided into total time and two stages. On average, SSA generation takes only 4.31 seconds per benchmark. This speed closely matches the non-SSA generation time of our frontend from Table 1 and is significantly faster

Table 2. Elapsed time of our frontend for generating SSA IR, measured in seconds. The "**Average**" in the bottom right corner of the table represents the average values for all the programs in our test suite.

Program	Total	3AC Tran.	Type Infer.	Program	Total	3AC Tran.	Type Infer.	Program	Total	3AC Tran.	Type Infer.	Program	Total	3AC Tran.	Type Infer.
D-avrora	0.12	0.10	0.01	D-jython	1.67	1.47	0.20	jre6	2.35	2.06	0.29	gradle	10.01	8.87	1.14
D-batik	0.30	0.26	0.04	D-kafka	3.62	3.27	0.34	jre8	2.53	2.18	0.35	intellij	16.34	14.55	1.79
D-biojava	1.06	0.92	0.14	D-luindex	0.32	0.28	0.04	jre11	5.45	4.85	0.60	jruby	0.95	0.84	0.11
D-cassandra	7.76	6.91	0.86	D-lusearch	0.33	0.28	0.05	jre17	5.42	4.83	0.59	kotlin	4.84	4.32	0.52
D-eclipse	1.48	1.29	0.19	D-pmd	0.53	0.46	0.07	jre21	5.50	4.88	0.62	metabase	12.19	10.67	1.51
D-fop	1.26	1.10	0.16	D-spring	4.13	3.51	0.62	compose	3.35	3.07	0.29	scala3	1.30	1.15	0.15
D-graphchi	2.62	2.24	0.38	D-sunflow	0.08	0.07	0.01	elastic	10.82	9.59	1.23	spark	20.79	18.52	2.26
D-h2	1.24	1.06	0.18	D-tomcat	0.85	0.71	0.13	flink	6.85	6.12	0.74				
D-h2o	8.55	7.57	0.98	D-xalan	0.38	0.32	0.05	frege	0.52	0.48	0.04				
D-jme	1.75	1.56	0.19	D-zxing	0.14	0.12	0.02	ghidra	12.06	10.35	1.71	**Average**	**4.31**	**3.81**	**0.50**

than WALA, which also produces SSA. Due to the nearly identical speed of generating non-SSA and SSA forms with our frontend, and considering that our non-SSA algorithm has previously been demonstrated to be significantly faster than those of Soot and SootUp — which require extra processing time for their SSA generations — to save space, we show only our SSA efficiency results in Table 2.

The efficiency of SSA generation by our frontend is attributed to its design. Creating SSA IR for bytecode necessitates resolving local and stack def-use relations. For local def-use relations, we perform an SSA transformation on bytecode that only needs to target `*store` and `*load` instructions, enhancing efficiency. For stack def-use relations, our frontend introduces temporary variables, which are inherently single-assignment, meeting SSA form requirements. Consequently, our frontend quickly generates SSA form with speed nearly matching that of non-SSA generation.

As discussed in Section 3, generating non-SSA IR with our frontend requires an additional pass to eliminate ϕ-functions. This prompts the question: why is SSA generation slightly slower than non-SSA generation in our frontend? We found that the primary reason is the type inference process for SSA IR, which is approximately 10% slower than for non-SSA IR. This slowdown occurs because SSA IR involves more variable processing, requiring the computation of the least common ancestor (LCA) for a greater number of variables—on average, 82,930 variables for SSA compared to 65,812 for non-SSA per benchmark. This increase is due to SSA's mechanism of splitting variables with multiple definitions into distinct variables assigned via ϕ-functions. Nevertheless, when compared to other frontends, the SSA generation of our frontend remains highly efficient.

5.5 RQ5: How Reliable is Our Frontend in Processing Bytecode?

While efficiency is the primary goal of our frontend, ensuring reliability is also crucial, particularly because a frontend serves as the initial step in a static analysis framework. Two critical factors contribute to the frontend's reliability: first, its ability to successfully process a wide range of bytecodes, and second, its ability to accurately preserve the original semantics during conversion to IR. Failure to achieve the first factor, such as through exceptions or crashes, prevents the analysis framework from obtaining the necessary IR for further analysis. Additionally, if the second factor is not met, it could compromise the correctness of subsequent analyses.

In this section, we assess the reliability of our frontend by evaluating its adherence to the two key factors. For the first factor, we conduct a comparative analysis of our frontend against others in processing bytecode from our extensive benchmark set. For the second factor, we conduct a round-trip evaluation to assess the accurate preservation of original semantics during conversion.

How Robust is Our Frontend in Processing Bytecode? We assessed the robustness of our frontend against other leading frontends by processing a comprehensive benchmark set comprising 8,861,806

Table 3. Method failure counts.

Program	Ours	Soot	WALA	SootUp	Program	Ours	Soot	WALA	SootUp	Program	Ours	Soot	WALA	SootUp
ghidra	0	1,047	0	0	D-jython	0	718	0	3	gradle	0	347	0	155
metabase	0	200	0	15	D-eclipse	0	57	0	1	intellij	0	23	0	229
kotlin	0	21	0	113	D-h2o	0	11	0	1	spark	0	9	0	5
compose	0	3	0	131	D-cassandra	0	3	0	3	D-tomcat	0	2	0	0
D-spring	0	1	1	0	elastic	0	0	0	3	flink	0	0	0	2
jre11	0	0	0	145	jre21	0	0	0	2	D-fop	0	0	0	1
D-jme	0	0	0	1	jruby	0	0	0	1	Total failed	**0**	2,142	1	817

methods as described in previous RQs. Table 3 lists only the benchmarks that include methods causing failures in any of the evaluated frontends and displays the number of these failed methods. Notably, our frontend showed the highest robustness, successfully processing all input bytecode methods. While Soot failed on 2,142 methods across 13 benchmarks, WALA failed on 1 method on 1 benchmark, and SootUp failed on 813 methods across 17 benchmarks.

In our investigation of the failures encountered by Soot and SootUp, we identified several recurring problems. Soot often struggled to accurately resolve class information, leading to errors related to inconsistent class hierarchies. It also faced difficulties in correctly constructing invocation expressions, particularly for virtual calls on interface types. In contrast, the shortcomings of SootUp were primarily due to its inability to accurately construct control flow graphs (CFGs), resulting in issues such as statements missing a predecessor. These findings highlight the superior robustness of our frontend.

Does Our Frontend Preserve Bytecode Semantics? We conduct a round-trip evaluation to assess whether our frontend accurately preserves bytecode semantics in practice. The evaluation involves three steps: 1. Convert real-world bytecode programs, denoted as P, into Tai-e IR using our frontend; 2. Transform the Tai-e IR back into bytecode form, denoted as P' (for this experiment, we implement an additional converter); 3. Execute both P and P' with identical inputs and compare their outputs. While this evaluation cannot fully verify semantic preservation—due to the inherent complexity of bytecode semantics and the substantial size of our frontend implementation (16k LoC)—it is nonetheless valuable in providing essential insights into semantic inconsistencies that may exist within our frontend.

This round-trip evaluation does impose specific requirements on the analyzed programs. In particular, the process necessitates that the programs include well-developed test suites to facilitate comparison of outputs from both the original (P) and transformed (P') versions. This requirement presents challenges for many benchmarks used in our previous research questions, as they often lack original tests and would otherwise require complex, project-specific adaptations.

To ensure a comprehensive evaluation, we selected three other projects that offer the necessary testing infrastructure: TheAlgorithms, the Clojure Compiler, and Jikes RVM. TheAlgorithms (61.1K stars on GitHub) offers a wide range of algorithm implementations leveraging core Java features. The Clojure Compiler (10.6K stars) provides test cases for compiling Clojure to bytecode, enhancing bytecode diversity. Jikes RVM includes extensive test cases for various bytecode scenarios. Collectively, these projects form a test suite comprising 158K lines of code.

Table 4. Round-trip evaluation results.

Project	Language	LoC	Passed Rate
TheAlgorithms	Java	46,604	100%
Clojure Compiler	Clojure, Java	90,171	100%
Jikes RVM	Java	21,522	100%
Total		158,299	100%

Table 4 shows that transformed programs (P', processed by our frontend) behave consistently with the original programs (P) from all three projects. The experimental results indicate that our frontend effectively preserves the semantics of bytecode.

Table 5. Comparison of live variable analysis and performance metrics between frontends.

	Total number of live variables (avg.)	LVA time (avg.)	Total time: frontend + LVA (avg.)
Tai-e (with our new frontend)	16,566,003	5.47s	9.68s
Tai-e (with Soot-based frontend)	16,764,009	5.57s	71.27s

5.6 RQ6: How Does Our Frontend Affect the Analysis Results?

In theory, the intermediate representations (IRs) generated by different frontends should be semantically equivalent. However, one might wonder how our frontend influences the analysis results and overall performance in practice compared to state-of-the-art frontend. To address this question, we conducted experiments using Live Variable Analysis (LVA), a classic and fundamental data-flow analysis. To ensure a meaningful evaluation while controlling for variables, we ran the same analysis (i.e., Tai-e's implementation of LVA) on the IRs produced by two frontends: our new frontend and the Soot's frontend. Since Tai-e does not support the Jimple IR produced by Soot, we utilized a transformer to convert Jimple IR into Tai-e IR, enabling us to conduct the experiments.

Table 5 summarizes the analysis (LVA) results and overall performance, presenting the average total number of live variables across statements, as well as the LVA time and total time (including both frontend and LVA) for each benchmark. The results demonstrate that our new frontend significantly enhances overall performance while having a negligible impact on the analysis results.

6 Related Work

The most relevant works have been compared and discussed throughout the paper. Below, we discuss additional related work that aligns with our bytecode to 3AC translation and type inference.

Bytecode to 3AC Translation. We examine the translation of bytecode to 3-address code (3AC) through three types of tools: static analysis frameworks, compilers, and verification tools.

Sawja [Demange et al. 2010; Hubert et al. 2011] is a Java analysis framework that provides a formally verified frontend for translating bytecode into 3AC, ensuring a high degree of reliability. However, Sawja primarily focuses on translation correctness and does not fully address several challenges central to our work, such as comprehensive type inference, SSA generation, and the sophisticated handling of reused local variable slots. These limitations affect its capabilities compared to our frontend.

Hotspot C1 [Kotzmann et al. 2008] and Jalapeño [Burke et al. 1999] are JIT (Just-In-Time) compilers with frontends capable of converting bytecode into 3-address code similar to ours. However, these compilers generate 3-address code predominantly for optimization purposes, whereas our frontend is designed for static analysis. Unlike our approach, they do not perform static type inference, leading to incomplete type information in their IR. Furthermore, these compilers utilize a different design for 3-address code translation compared to our frontend. The compilers' IRs are optimized for efficient code transformation, employing a pipeline-like methodology that starts with a basic bytecode-to-3AC translation followed by a series of optimization passes. In contrast, our 3-address code translation involves on-the-fly redundant code elimination during the translation.

[Gal et al. 2008] describes a bytecode verification tool that translates bytecode to SSA IR. It employs a Soot-like approach to produce non-SSA IR and then transforms it into SSA using Cytron's method [Cytron et al. 1991]. In contrast, our frontend performs an initial SSA transformation directly on the bytecode before generating IR, a design proven to be efficient in practice.

Type Inference. [Gagnon et al. 2000] introduces an algorithm to infer type information for 3-address code translated from bytecode, similar to the problem addressed by our type inference.

This algorithm treats type constraints as algebraic equations, identifying and simplifying reducible constraints through algebraic operations until no further simplification is possible. The resulting simplified constraints form the basis for the inferred types. In contrast, both [Bellamy et al. 2008] and our type inference represent type constraints using a graph-based framework, which allows for more efficient resolution. Notably, the algorithm described in [Gagnon et al. 2000] was initially used in Soot for type inference but was subsequently replaced by the more efficient graph-based approach outlined in [Bellamy et al. 2008].

[Knoblock and Rehof 2001] introduces a type inference algorithm for bytecode, which is employed by the Marmot compiler [Fitzgerald et al. 2000]. When the algorithm encounters multiple LCAs during inference, unlike our approach which strives to infer the most precise type, it uses *subtype completion* to generate a new class type that acts as the common supertype for all LCAs. However, this method disrupts the original type hierarchy, making it unsuitable for static analysis.

7 Conclusions

The bytecode frontend plays a vital role in Java static analysis frameworks, converting bytecode into typed 3-address code, which is necessary for any sophisticated static analysis. Its efficiency directly impacts user experience and the overall performance of the framework. To accelerate the frontend, this paper introduces two novel approaches: pattern-aware 3-address code translation and pruning-based type inference, effectively overcoming the performance bottlenecks in existing methods. We have implemented these approaches in our new frontend. Our experiments on an extensive benchmark set reveal that our frontend significantly outperforms established frontends used in prominent Java static analysis frameworks such as Soot, WALA, and SootUp, achieving improvements by an order of magnitude. Additionally, it demonstrates better reliability compared to other frontends. Our methods also support the generation of both non-SSA and SSA IR, thus extending their applicability across various static analysis techniques.

We will fully open-source our frontend implementation to contribute accessible solutions to the static analysis community, paving the way for more efficient Java static analysis practices.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. This work is supported in part by National Key R&D Program of China under Grant No. 2023YFB4503804 and National Natural Science Foundation of China under Grant No. 62402210. Tian Tan, the co-corresponding author, is also supported by Xiaomi Foundation.

Data-Availability Statement

We have provided an artifact [Li et al. 2025] to reproduce all experimental results presented in our evaluation (Section 5). This artifact includes all benchmarks, all the bytecode frontends used for comparison, and both the source code and executable of our new frontend implementation. The artifact is available at https://doi.org/10.5281/zenodo.16923368. To reproduce the results, please refer to the instructions provided in the accompanying `README.pdf` document within the artifact.

References

Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (2008), 22–29. doi:10.1109/MS.2008.130

Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. 2007. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, San Diego California USA, 1–8. doi:10.1145/1251535.1251536

Ben Bellamy, Pavel Avgustinov, Oege de Moor, and Damien Sereni. 2008. Efficient local type inference. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA '08)*. Association for Computing Machinery, New York, NY, USA, 475–492. doi:10.1145/1449764.1449802

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. doi:10.1145/1167473.1167488

Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, and Christophe Guillon. 2009. Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09)*. IEEE Computer Society, USA, 114–125. doi:10.1109/CGO.2009.19

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. *ACM SIGPLAN Notices* 44, 10 (Oct. 2009), 243–262. doi:10.1145/1639949.1640108

Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. 1998. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.* 28, 8 (jul 1998), 859–881.

Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. 1999. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM 1999 Conference on Java Grande* (San Francisco, California, USA) *(JAVA '99)*. Association for Computing Machinery, New York, NY, USA, 129–141. doi:10.1145/304065.304113

Yuandao Cai, Peisen Yao, and Charles Zhang. 2021. Canary: practical static detection of inter-thread value-flow bugs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1126–1140. doi:10.1145/3453483.3454099

Yiu Wai Chow, Max Schäfer, and Michael Pradel. 2023. Beware of the Unexpected: Bimodal Taint Analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 211–222. doi:10.1145/3597926.3598050

Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*. Association for Computing Machinery, 332–343. doi:10.1145/2970276.2970347 Place: New York, NY, USA.

Keith Cooper, Timothy Harvey, and Ken Kennedy. 2006. *A Simple, Fast Dominance Algorithm.* Journal Abbreviation: Rice University, CS Technical Report 06-33870 Publication Title: Rice University, CS Technical Report 06-33870.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. doi:10.1145/115372.115320

Delphine Demange, Thomas Jensen, and David Pichardie. 2010. A Provably Correct Stackless Intermediate Representation for Java Bytecode. In *Programming Languages and Systems*, Kazunori Ueda (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 97–113.

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

Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. 2000. Marmot: an optimizing compiler for Java. *Software: Practice and Experience* 30, 3 (2000), 199–232. doi:10.1002/(SICI)1097-024X(200003)30:3<199::AID-SPE296>3.0.CO;2-2

Etienne M. Gagnon, Laurie J. Hendren, and Guillaume Marceau. 2000. Efficient Inference of Static Types for Java Bytecode. In *Static Analysis*, Jens Palsberg (Ed.). Springer, Berlin, Heidelberg, 199–219. doi:10.1007/978-3-540-45099-3_11

Andreas Gal, Christian W. Probst, and Michael Franz. 2008. Java bytecode verification via static single assignment form. *ACM Trans. Program. Lang. Syst.* 30, 4, Article 21 (aug 2008), 21 pages. doi:10.1145/1377492.1377496

Laurent Hubert, Nicolas Barré, Frédéric Besson, Delphine Demange, Thomas Jensen, Vincent Monfort, David Pichardie, and Tiphaine Turpin. 2011. Sawja: Static Analysis Workshop for Java. In *Formal Verification of Object-Oriented Software*, Bernhard Beckert and Claude Marché (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 92–106.

Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. 2024. SootUp: A Redesign of the Soot Static Analysis Framework. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer Nature Switzerland, 229–247. doi:10.1007/978-3-031-57246-3_13 Place: Cham.

Todd B. Knoblock and Jakob Rehof. 2001. Type elaboration and subtype completion for Java bytecode. *ACM Trans. Program. Lang. Syst.* 23, 2 (March 2001), 243–272. doi:10.1145/383043.383045

Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot™ client compiler for Java 6. *ACM Trans. Archit. Code Optim.* 5, 1, Article 7 (may 2008), 32 pages. doi:10.1145/1369396.1370017

Matthieu Lemerre. 2023. SSA Translation Is an Abstract Interpretation. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 65:1895–65:1924. doi:10.1145/3571258

Chenxi Li, Haoran Lin, Tian Tan, and Yue Li. 2025. *Two Approaches to Fast Bytecode Frontend for Static Analysis (Artifacts).* doi:10.5281/zenodo.16923368

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 111 (April 2024), 26 pages. doi:10.1145/3649828

Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program Tailoring: Slicing by Sequential Criteria. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:27. doi:10.4230/LIPIcs.ECOOP.2016.15

Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2021. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2183–2196. doi:10.1145/3460120.3484541

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

Tong Liu, Zizhuang Deng, Guozhu Meng, Yuekang Li, and Kai Chen. 2024. Demystifying RCE Vulnerabilities in LLM-Integrated Apps. 15 pages. doi:10.1145/3658644.3690338

Anders Møller and Michael I. Schwartzbach. 2018. Static Program Analysis. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

Anders Møller and Oskar Haarklou Veileborg. 2020. Eliminating abstraction overhead of Java stream pipelines using ahead-of-time program optimization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 168 (Nov. 2020), 29 pages. doi:10.1145/3428236

Vaughan Pratt and Jerzy Tiuryn. 1996. Satisfiability of Inequalities in a Poset. *Fundam. Inf.* 28, 1,2 (apr 1996), 165–182.

Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. 1999. Translating Out of Static Single Assignment Form. In *Static Analysis*, Agostino Cortesi and Gilberto Filé (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 194–210.

Mohammad Tahaei, Kami Vaniea, Konstantin (Kosta) Beznosov, and Maria K Wolters. 2021. Security Notifications in Static Analysis Tools: Developers' Attitudes, Comprehension, and Ability to Act on Them. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 691, 17 pages. doi:10.1145/3411764.3445616

Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, 1093–1105. doi:10.1145/3597926.3598120 Place: New York, NY, USA.

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research (CASCON '99)*. IBM Press, 13. Place: Mississauga, Ontario, Canada.

WALA. 2006. Watson Libraries for Analysis.

Christian Wimmer, Codrut Stancu, David Kozak, and Thomas Würthinger. 2024. Scaling Type-Based Points-to Analysis with Saturation. *Proc. ACM Program. Lang.* 8, PLDI, Article 187 (June 2024), 24 pages. doi:10.1145/3656417

Yichi Zhang. 2024. Detecting Code Comment Inconsistencies using LLM and Program Analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 683–685. doi:10.1145/3663529.3664458

Zexin Zhong, Jiangchao Liu, Diyu Wu, Peng Di, Yulei Sui, and Alex X. Liu. 2022. Field-based static taint analysis for industrial microservices. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 149–150. doi:10.1145/3510457.3513075

Received 2025-03-25; accepted 2025-08-12