UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
BACHARELADO EM SISTEMAS DE INFORMAÇÕES

Henrique Gomes de Oliveira

Detecção de Conflitos Semânticos usando Infer

Recife
2025

Henrique Gomes de Oliveira

Detecção de Conflitos Semânticos usando Infer

Trabalho apresentado ao Programa de Graduação em Sistemas de Informações do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informações.

Área de Concentração: Engenharia de Software

Orientador (a): Paulo Borba

Coorientador (a): Gabriela Sampaio

Recife

2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Oliveira, Henrique Gomes de.
Detecção de Conflitos Semânticos usando Infer / Henrique Gomes de Oliveira. - Recife, 2025.
47 p. : il., tab.

Orientador(a): Paulo Borba
Cooorientador(a): Gabriela Sampaio
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Sistemas de Informação - Bacharelado, 2025.
Inclui referências.

1. Engenharia de Software. 2. Conflitos de Integração de Código. 3. Dataflow. 4. Infer. 5. Análise Estática. 6. Conflitos Semânticos. I. Borba, Paulo. (Orientação). II. Sampaio, Gabriela. (Coorientação). IV. Título.

000 CDD (22.ed.)

Henrique Gomes de Oliveira

Detecção de Conflitos Semânticos usando Infer

Trabalho apresentado ao Programa de Graduação em Sistemas de Informações do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informações.

Aprovado em: 03/04/2025.

BANCA EXAMINADORA

_____

Dr. Paulo Borba (Orientador)

Universidade Federal de Pernambuco - UFPE

_____

Drª. Gabriela Sampaio (Coorientadora)

Meta

_____

Dr. André Luis (Examinador Interno)

Universidade Federal de Pernambuco - UFPE

AGRADECIMENTOS

Primeiramente, agradeço a Deus, meu braço forte e minha fortaleza, por ter sustentado cada passo da minha jornada até aqui. Sem Sua graça e direção, nada disso teria sido possível.

À minha esposa Maria Eduarda, meu amor, minha amiga, minha parceira de vida. Sua gentileza, paciência e presença nos momentos mais difíceis foram fundamentais para que eu não desistisse. Obrigado por estar sempre ao meu lado.

Aos meus pais, Ricardo e Verônica, minha base e inspiração. Obrigado por acreditarem em mim, investirem na minha educação e sempre me incentivarem a ir além. Um agradecimento especial ao meu pai, exemplo de homem trabalhador, cujo esforço e suor tornaram possível que eu tivesse acesso a uma educação de qualidade.

Aos meus irmãos, Riquinho, que me apresentou ao mundo da tecnologia e sempre teve uma palavra sábia para me orientar, e Gustavo, meu amigo e parceiro de infância, com quem compartilhei tantos momentos importantes da vida.

Gostaria de agradecer também aos professores do CIN, pela excelência no ensino e pela dedicação à formação de cada aluno. Em especial, ao Professor Paulo, por toda orientação, paciência e apoio durante este trabalho, e à Gabi, que como coorientadora, foi essencial para o sucesso deste projeto, contribuindo com seu conhecimento técnico.

Agradeço ainda ao meu chefe, Carluci, por toda a flexibilidade e apoio que me proporcionou, para que eu pudesse conciliar trabalho e estudos ao longo dessa jornada.

Por fim, deixo um agradecimento aos meus amigos. Sem os momentos de descontração, conversas, jogatinas e saídas, eu não teria suportado tanta pressão nesses últimos períodos de faculdade. Obrigado por estarem comigo, trazendo leveza e alegria mesmo nos momentos mais difíceis.

A todos vocês, minha eterna gratidão. Esse trabalho é também de vocês.

RESUMO

No desenvolvimento colaborativo de software, a integração de código entre diferentes desenvolvedores frequentemente leva a conflitos. Enquanto os conflitos textuais são bem tratados por sistemas de controle de versão como o Git, os conflitos semânticos — onde as alterações interferem no comportamento do sistema — continuam sendo um grande desafio. Esses conflitos podem ser estáticos, detectados em tempo de compilação, ou dinâmicos, emergindo apenas durante a execução. Este trabalho foca na detecção de conflitos semânticos dinâmicos usando o Infer, uma ferramenta de análise estática desenvolvida pela Meta.

As abordagens atuais para detectar esses conflitos dependem da geração automatizada de testes, bem como de análises dinâmicas e estáticas, que enfrentam problemas de escalabilidade e precisão. Para melhorar a eficiência, este estudo propõe a adaptação da Análise de Taint do Infer para uma abordagem baseada em fluxo de dados na detecção de conflitos semânticos. Esse método busca aprimorar a eficiência computacional, garantindo ao mesmo tempo a identificação confiável de interferências.

A técnica proposta envolve a transformação de código Java utilizando o Java Development Tools (JDT) para marcar sistematicamente modificações feitas por diferentes desenvolvedores. Essa transformação permite que o Infer rastreie o fluxo de dados entre alterações conflitantes, possibilitando a detecção de interferências. A metodologia inclui a travessia automatizada da AST, a marcação de instruções sensíveis ao fluxo de dados e a análise interprocedimental para propagar a detecção de conflitos através de chamadas de métodos e instâncias de objetos.

Os resultados da abordagem adaptada baseada no Infer indicam seu potencial para detectar interferências semânticas em código mesclado. Ao equilibrar precisão e desempenho, este trabalho contribui para o avanço das estratégias de detecção de conflitos no desenvolvimento colaborativo de software, minimizando interrupções comportamentais indesejadas em bases de código integradas.

Palavras-chave: Conflitos Semânticos, Análise Estática, Infer, Fluxo de Dados, Merge.

ABSTRACT

In collaborative software development, code integration between different developers often leads to conflicts. While textual conflicts are well addressed by version control systems like Git, semantic conflicts—where changes interfere with system behavior—remain a significant challenge. These conflicts can be static, detected at compile-time, or dynamic, emerging only during execution. This work focuses on detecting dynamic semantic conflicts using Infer, a static analysis tool developed by Meta.

Current approaches to detecting such conflicts rely on automated test generation, dynamic and static analysis, which face scalability and accuracy issues. To improve efficiency, this study proposes adapting Infer's Taint Analysis to a data flow-based approach for semantic conflict detection. This method enhances computational efficiency while ensuring reliable interference identification.

The proposed technique involves transforming Java code using the Java Development Tools (JDT) to systematically mark modifications from different developers. This transformation enables Infer to track data flow between conflicting changes, allowing interference detection. The methodology includes automated AST traversal, marking of data flow-sensitive statements, and interprocedural analysis to propagate conflict detection across method calls and object instances.

Results from the adapted Infer-based approach indicate its potential to detect semantic interference in merged code. By balancing precision and performance, this work contributes to advancing conflict detection strategies in collaborative software development, minimizing unintended behavioral disruptions in integrated codebases.

Keywords: Semantic Conflicts, Static Analysis, Infer, Data Flow, Merge.

SUMÁRIO

1 Introdução

No contexto do desenvolvimento de software colaborativo, o processo de integração de código entre diferentes desenvolvedores é desafiador e frequentemente suscetível a conflitos. Enquanto conflitos textuais são amplamente abordados por ferramentas como o Git, eles representam apenas uma parte dos problemas que podem surgir. Há também os conflitos semânticos, que ocorrem quando as alterações de dois desenvolvedores, embora integradas com sucesso no nível textual, interferem no comportamento ou no estado esperado do sistema durante a execução.

Os conflitos semânticos podem ser classificados como estáticos ou dinâmicos. Conflitos semânticos estáticos são identificados no momento da compilação e geralmente se manifestam como erros relacionados à estrutura ou às dependências do código, como referências inexistentes [1]. Já os conflitos semânticos dinâmicos, foco deste trabalho, são aqueles que emergem somente durante a execução do programa. Um exemplo típico acontece quando as alterações feitas por um desenvolvedor afetam um elemento de estado que é acessado pelo código alterado por outro desenvolvedor, que assumiu uma invariante de estado que não é mais válida após o merge. Isso pode levar a comportamentos inesperados, pois a segunda alteração depende de uma premissa que não é mais satisfeita.

Para detectar conflitos semânticos dinâmicos, diferentes abordagens foram propostas. Métodos baseados em geração automática de testes, utilizam casos de teste como especificações parciais das alterações que devem ser integradas, identificando comportamentos indesejados gerados durante o processo de merge [2]. De outra forma há o uso de análise dinâmica, destacando sua capacidade de detectar conflitos envolvendo operações de escrita no mesmo elemento de estado durante a execução, sem a necessidade de especificações explícitas do código em análise [3]. Embora essas abordagens sejam eficazes, enfrentam limitações relacionadas à performance e escalabilidade.

Complementando essas técnicas, a detecção de conflitos semânticos pode ser feita com análises estáticas, como System Dependence Graphs (SDGs) [11], embora tenham alto custo computacional. Para melhorar o desempenho, técnicas mais leves foram exploradas, como Interprocedural Data Flow (DF), Interprocedural Confluence, Interprocedural Override Assignment (OA) e Program Dependence

Graph (PDG) [12, 13]. Entre essas, a análise de Data Flow se destaca pela precisão na identificação de interferências em fluxos de dados entre alterações, mas apresenta desafios devido ao alto custo computacional associado à sua execução. No trabalho de Galileu [1], esse custo é reduzido pela imposição de um limite de profundidade de 5, o que restringe a análise a um número máximo de interações entre funções e torna o processo mais eficiente. No entanto, de forma geral, a análise de Data Flow ainda é computacionalmente cara. Isso gera a necessidade de explorar ferramentas industriais, como o Infer, que, embora menos precisas, oferecem uma solução mais eficiente para a detecção de conflitos semânticos.

Análises de taint [14] são baseadas em fluxos de dados e frequentemente utilizadas em contextos de segurança ou privacidade para identificar a propagação de dados maliciosos ou sensíveis durante a execução de um programa. De modo geral, essas análises recebem como entrada *sources* e *sinks*, que denotam respectivamente o dado sensível que se deseja rastrear na análise e o ponto no programa onde esse dado *tainted* (contaminado) pode levar a uma vulnerabilidade.

Existem inúmeras ferramentas [15, 16, 17, 24] que implementam análises de *taint*. Um exemplo clássico é a ferramenta Infer [4], desenvolvida pela Meta, que provê análises estáticas de *taint* para diversas linguagens de programação. Por analisar código industrial em larga escala na Meta, Infer tem alta escalabilidade e implementa heurísticas para lidar com situações onde o código a ser analisado tem alta complexidade, tornando-a robusta quando comparada a outras ferramentas utilizadas em projetos de menor escala.

Este trabalho se propõe a explorar a utilização da ferramenta Infer na detecção de conflitos semânticos dinâmicos. Para isto, adaptamos a análise de taint do Infer para o contexto de conflitos semânticos. Consideramos que as mudanças feitas por dois desenvolvedores representam, respectivamente, *sources* e o *sinks* da análise. A abordagem utilizada consiste das seguintes etapas: (1) Identificação e coleta de dados de conflitos semânticos em projetos open source: utilizamos funções do Mining [8] para identificar os arquivos e as linhas alteradas pelos desenvolvedores durante os merges, permitindo a coleta dos dados necessários para a análise; (2) Marcação do código alterado pelos dois desenvolvedores: aplicamos a API do Eclipse JDT para marcar os nós alterados na AST, encapsulando-os com chamadas de métodos de uma classe wrapper. Esses métodos são configurados como source e sink na análise do Infer, permitindo o

rastreamento correto dos fluxos de dados; (3) Execução da análise Pulse com a ferramenta Infer: rodamos a análise Pulse, fornecendo o código modificado pelos dois desenvolvedores como entrada, e geramos o report com os possíveis conflitos semânticos detectados.

A avaliação do software desenvolvido demonstrou tanto a consistência na transformação do código quanto a precisão na detecção de conflitos semânticos. Embora a compilação não tenha sido bem-sucedida em todos os casos, a ferramenta conseguiu aplicar corretamente as transformações em 17 dos 18 commits testados, sem gerar erros de sintaxe. A análise também destacou a eficácia do Infer na detecção de conflitos semânticos. Nos commits que foram compilados com sucesso, a ferramenta identificou corretamente os conflitos, além de detectar outros tipos de erros, como Null Dereference e Resource Leak, reforçando sua capacidade de identificar problemas no código.

E por fim entre as principais contribuições deste trabalho, destacam-se: a adaptação da Análise de Taint do Infer para a detecção de conflitos semânticos em fluxos de dados; a automatização do processo de marcação de código utilizando a API JDT; e a validação empírica da abordagem em um conjunto diversificado de projetos.

A pesquisa está organizada da seguinte forma: na Seção 2, são apresentados os conceitos fundamentais de interferência e como ela pode ocorrer por meio de dataflow, além de um exemplo motivador para o trabalho. Na Seção 3, é detalhado o fluxo de execução do Infer e como o código fonte pode ser adaptado para suportar a ferramenta na detecção de conflitos. A Seção 4 descreve o funcionamento central da ferramenta, abordando o processo de transformação do código fonte, as visitações e as alterações nos nós da AST. Na Seção 5, são discutidos os resultados da avaliação e experimentação da ferramenta. As limitações encontradas são apresentadas na Seção 6, enquanto trabalhos relacionados são explorados na Seção 7. Por fim, a conclusão é apresentada na Seção 8.

2 Interferência por meio de dataflow

Um conflito semântico ocorre quando, durante o processo de merge[1], há uma interferência indesejada entre as alterações feitas por diferentes desenvolvedores [5]. Esse tipo de conflito vai além dos conflitos sintáticos detectados por sistemas tradicionais de controle de versão, pois afeta o comportamento do programa após a integração.

Para entender como essas interferências acontecem, considere um repositório de código onde um commit base *B* é modificado por dois desenvolvedores simultaneamente. O desenvolvedor à esquerda cria a versão *L* e o desenvolvedor à direita cria a versão *R*. O merge *M* é a junção dessas versões *L* e *R* sobre *B*.

Formalmente dizemos que as alterações separadas *L* e *R* em um programa base *B* interferem quando as alterações integradas não preservam o comportamento alterado de *L* ou *R*, ou o comportamento inalterado de *B* [5].

No contexto de análise de fluxo de dados, uma interferência ocorre quando há uma relação de def-use entre as versões *L* e *R* e uma dependência entre as alterações. Essa relação é caracterizada por uma variável que tem seu valor definido em um ponto do programa (*def*) e lido posteriormente (*use*). Se *L* altera o estado de uma variável e *R* lê esse estado alterado, ou vice-versa, surge uma interferência indesejada [1].

Figura 1 – Motivating Example

```
1. public class MotivatingExample {
2.     private String text;
3.
4.     public void cleanText() {
5.         normalizeWhiteSpace(); //left
6.         removeComments();
7.         removeDuplicateWords(); //right
8.     }
9. }
```

Fonte: SILVA et al. [18]

[1] O comando git merge integra mudanças de diferentes branches em um só.

Considere a Figura 1. O método `cleanText` tem a finalidade de remover espaços extras, comentários e palavras duplicadas de um atributo `text` dessa classe. A chamada do método `normalizeWhiteSpace()`, introduzida na versão comentada no código como left, altera `text` removendo espaços consecutivos. Já `removeDuplicateWords()`, adicionada na versão right, também modifica `text`, mas antes de fazê-lo, lê o valor modificado por `normalizeWhiteSpace()`. Isso gera uma relação de fluxo de dados, mais especificamente uma em que é estabelecida uma dependência entre as duas alterações, onde uma modifica um valor que a outra assume como referência, resultando em uma interferência semântica.

Considerando a entrada "the_the__dog", utilizaremos “_” para representar um espaço em branco. A execução na versão *L*, que apenas aplica `normalizeWhiteSpace()`, resulta em "the_the_dog". Já a execução na versão *R*, onde apenas `removeDuplicateWords()` é aplicado, gera "the___dog". No entanto, ao realizar o merge das duas versões sem resolver o conflito, o resultado final será "the___dog". Esse comportamento ocorre porque `removeDuplicateWords()` remove a segunda ocorrência de "the" sem eliminar corretamente os espaços extras, contrariando a intenção original da modificação em *L*. Assim, no *merge* das duas versões apenas o comportamento esperado de *R* é preservado, dessa forma *R* interfere em *L*, caracterizando um conflito semântico decorrente de interferência no fluxo de dados [1].

Detectar conflitos semânticos é desafiador, pois revisores de código, testes do projeto e pipelines de integração contínua (CI) nem sempre conseguem identificar essas falhas antes da execução [19, 20]. Isso reforça a necessidade de ferramentas especializadas, incluindo análises estáticas, que podem antecipar esses problemas. Dentre elas, ferramentas eficientes como o Infer se destacam por equilibrar desempenho e precisão na detecção de conflitos.

3 Infer e detecção de conflitos

Infer [4] é uma ferramenta de análise estática desenvolvida pela Meta com o objetivo de detectar potenciais bugs antes que o código seja enviado aos usuários, ajudando a evitar falhas que possam comprometer a estabilidade e o desempenho das aplicações. A ferramenta é amplamente utilizada na análise de código para identificar problemas em programas escritos em diversas linguagens, como Java, Android, C, C++, Objective-C e Erlang. Em particular, temos interesse em seu uso nas linguagens Java e Android, que permite identificar erros relacionados a de ponteiro nulo, vazamentos de recursos, acessibilidade de anotações, ausência de bloqueios necessários e até mesmo segurança e privacidade.

Por analisar código industrial em larga escala na Meta, Infer possui alta escalabilidade e robustez. Sendo assim, este trabalho propõe a utilização da análise de Taint do Infer, inicialmente focada em aspectos de segurança e privacidade, adaptando-a para uma análise de fluxo de dados voltada à detecção de conflitos semânticos. No entanto, essa adaptação pode reduzir a precisão da detecção, pois o Infer adota uma abordagem conservadora para evitar falsos positivos. Na análise de *taint*, por exemplo, a ferramenta não propaga *taint* em certos operadores de Java, limitando a detecção de fluxos de dados em alguns cenários. Além disso, como toda análise estática, o Infer não tem acesso ao *runtime* do programa, o que restringe sua capacidade de capturar certos comportamentos dinâmicos.

3.1 Infer workflow

O Infer opera em duas fases principais: captura e análise. Na fase de captura, ele intercepta os comandos de compilação para traduzir os arquivos de código-fonte em uma linguagem intermediária própria. Durante esse processo, a ferramenta computa o grafo de dependências dos procedimentos a serem analisados. Já na fase seguinte, de análise, o Infer analisa os procedimentos utilizando uma técnica composicional, *biabduction*, baseada em lógica de separação [21, 22]. Isso permite que cada procedimento seja analisado independentemente, onde o Infer tentará inferir os respectivos sumários (pré e pós condições). Então, o Infer atualiza os sumários inferidos levando em consideração o contexto das chamadas dos procedimentos, que pode ser relevante, por exemplo, devido a informações

adicionais de tipos dinâmicos. Durante a análise, caso o Infer encontre algum possível caminho de execução que leve a algum erro, esse erro é reportado para o usuário [4].

3.2 Análise Pulse e limitações

Esse estudo utilizará a análise Pulse [23] do Infer, com foco na detecção de erros classificados como Sensitive Data Flow, que são falhas do tipo Taint. A análise Pulse será explorada para entender como a ferramenta pode ser adaptada para identificar fluxos de dados sensíveis em cenários de conflitos semânticos. Ferramentas como Interprocedural Dataflow e JOANA [1, 18] permitem identificação de *sensitive data flow* com base em linhas alteradas. Ou seja, dadas contribuições de dois desenvolvedores, essas ferramentas tem como entrada linhas modificadas por cada desenvolvedor e dão como saída informações de fluxos de com base em pares de linhas modificadas.

O Infer opera de maneira diferente. Ao invés de ter linhas modificadas como entrada, precisamos expressar as modificações dos desenvolvedores em termos de métodos, atributos e variáveis. Essa abordagem não é completa e não abrange certos tipos de alterações como a ferramenta de Galileu e Leuson [1, 18]. Por este motivo, desenvolvemos nesse trabalho um mecanismo de marcação, que encapsula as alterações de código por meio de métodos que servem de *input* para o Infer. O programa atribui as marcações de forma que as alterações de *left* sejam identificadas como *source* e as de *right* como *sink*. Após a execução da primeira análise, os papéis são invertidos e uma nova análise é realizada para identificar possíveis interferências entre as duas linhas de fluxos.

3.3 Configurações

Dado que a configuração padrão do Infer não permite a marcação direta de statements, foi necessária uma adaptação para viabilizar a análise proposta. Para isso, foi desenvolvida uma classe denominada `InferWrapper`, cuja função é encapsular valores e sinalizar ao Infer a presença de dados sensíveis. Essa classe contém os métodos `l` e `r` com parâmetro para um valor genérico e retorno de

mesmo tipo. Esses metódos são construídos para encapsular os nós alterados a fim de que o infer identifique a marcação.

Figura 2 – Classe InferWrapper

```
1. public class InferWrapper {
2.     public static <T> T l(T value) {return value;}
3.     public static <T> T r(T value) {return value;}
4. }
```

Fonte: retirada do programa de [6].

A abordagem adotada consiste em configurar o Infer para reconhecer os métodos da classe `InferWrapper` como *source* e *sink* de dados sensíveis. O método `l` representa um valor modificado por *left*, enquanto o método `r` indica uma modificação por *right*. Essa configuração é definida em dois arquivos estáticos *infer_config_left_to_right.json* e *infer_config_right_to_left.json*, que estabelece os métodos da classe `InferWrapper` como pontos de interesse para a análise de fluxo de dados do Infer.

Figura 3 – infer_config_left_to_right.json

```
 1. {
 2.   "pulse-taint-sources": [
 3.     {
 4.       "class_names": ["infer.InferWrapper"],
 5.       "method_names": ["l"],
 6.       "taint_target": ["ArgumentPositions", [0]]
 7.     }
 8.   ],
 9.   "pulse-taint-sinks": [
10.     {
11.       "class_names": ["infer.InferWrapper"],
12.       "method_names": ["r"]
13.     }
14.   ]
15. }
```

Fonte: retirada do programa de [6].

No arquivo *infer_config_left_to_right.json*, o método `l` da classe `InferWrapper` é configurado como *source*, representando as alterações realizadas na versão *left*. Além disso, um *taint_target* é definido para indicar que o primeiro argumento passado para o método `l` deve ser tratado como um dado sensível. Por outro lado, o método `r` é configurado como *sink*, sinalizando os pontos de consumo das modificações oriundas da versão *right*. No arquivo *infer_config_right_to_left.json*, a configuração é invertida: o método `r` passa a ser identificado como *source*, com seu respectivo *taint_target* configurado, enquanto o método `l` é definido como *sink*. Essa inversão permite que a análise seja realizada em ambas as direções, possibilitando a detecção de interferências entre as versões do código a partir de diferentes perspectivas. Automatizando esse empacotamento de nós, é possível alcançar um comportamento semelhante ao da ferramenta de Galileu [1], mas utilizando uma análise estática industrial de alta eficiência, como a do Infer.

4 Solução

Com a configuração adequada do Infer, é possível realizar a análise de conflitos semânticos manualmente, marcando variáveis, atributos, objetos e quaisquer estruturas de dados relevantes, como ilustrado na seção anterior. Para isso, basta encapsular os valores modificados nas versões *left* e *right* através de chamadas aos métodos da classe `InferWrapper`, garantindo que o Infer identifique corretamente o fluxo de dados entre as diferentes versões do código.

Figura 4 – Exemplo de empacotamento de modificações

Original Code

```
1.  String s = "tained"; //left
2.  //... base
3.  foo(s); //right
```

Transformed Code

```
1.  String s = InferWrapper.l("tained");
2.  //... base
3.  foo(InferWrapper.r(s));
```

Fonte: Imagem elaborada pelo autor (2025).

Considere o programa da Figura 4. A transformação aplicada insere chamadas aos métodos `l` e `r` da classe `InferWrapper` para explicitar o fluxo de dados ao Infer. Especificamente, o valor inicial da variável `s`, modificado na versão *left*, é encapsulado pelo método `l`, configurado como *source* na análise (Figura 3). Da mesma forma, o uso subsequente dessa variável na chamada `foo(s)`, presente na versão *right*, é envolvido pela chamada do método `r`, definido como *sink*. Essa abordagem permite que o Infer identifique corretamente a propagação de informações entre as duas versões e, potencialmente, conflitos semânticos decorrentes dessa interação.

Essa transformação foi escolhida porque respeita a relação entre *source* e *sink* definida na configuração do Infer, garantindo que qualquer dado modificado em uma versão seja devidamente rastreado até seu consumo na outra. Mais adiante,

detalharemos como essa estratégia é generalizada para diferentes cenários, apresentando as regras e o algoritmo de marcação adotados.

A fim de automatizar esse processo e garantir consistência na aplicação das marcações, a transformação de código será realizada utilizando o Java Development Tools (JDT) do Eclipse [7]. O JDT fornece uma API robusta para análise e manipulação do código-fonte Java, permitindo a identificação das estruturas alteradas e a inserção das chamadas aos métodos da classe `InferWrapper`.

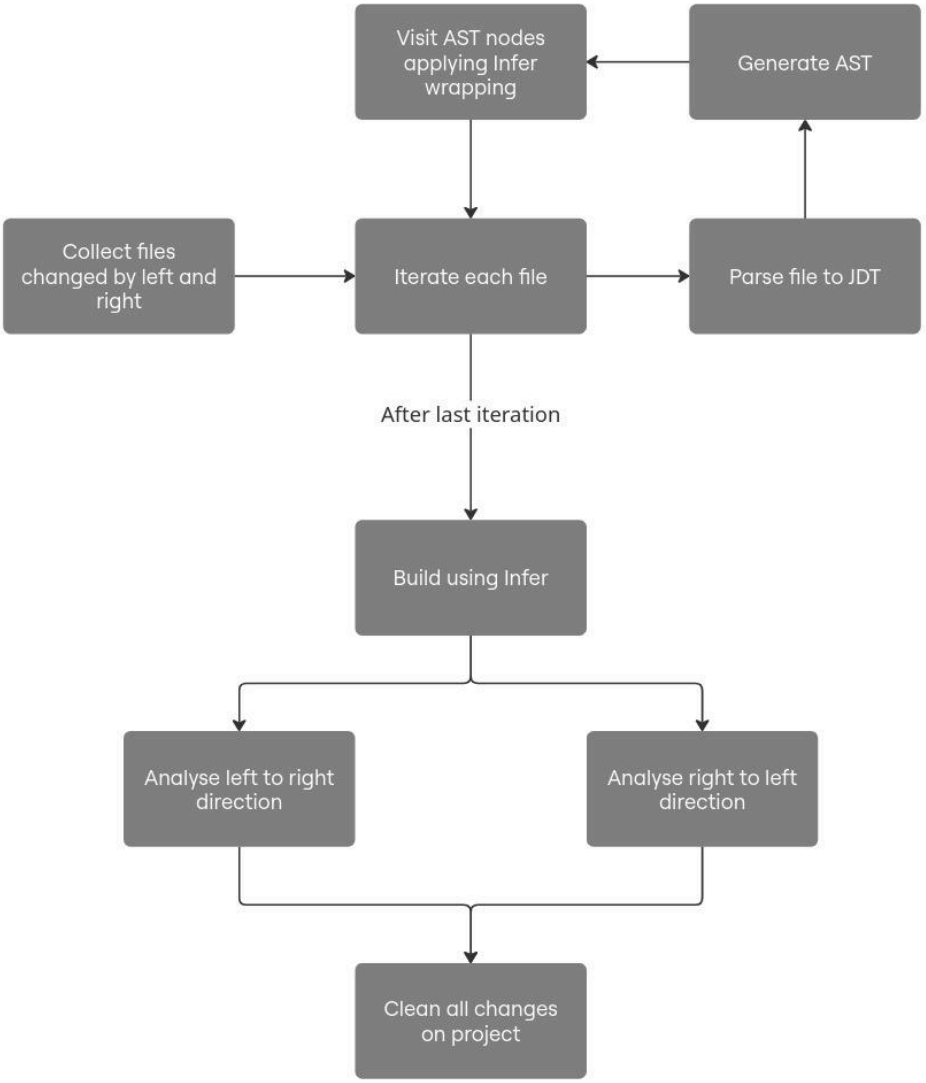
4.1 Fluxo geral da solução

O fluxo geral da solução como demonstrado no fluxograma 1 se inicia com a coleta dos arquivos modificados por left e right, que representam as versões a serem comparadas. Em seguida, cada arquivo é processado individualmente, onde é feito o parsing para o JDT, gerando uma Abstract Syntax Tree (AST). Durante esse processo, os nós da AST são percorridos e a transformação de Infer *wrapping* é aplicada nas expressões relevantes, garantindo que o fluxo de dados possa ser rastreado pela ferramenta.

Após essa etapa, o projeto é compilado utilizando o Infer, permitindo que a análise de conflitos seja realizada. A análise ocorre em duas direções: primeiro, os efeitos das alterações de *left* sobre *right* são analisados, considerando uma configuração do Infer em que os métodos encapsulados por `l` são *source*, e `r` *sink*; depois, a direção oposta é verificada, por uma segunda configuração que possui o inverso das referências de *source* e *sink* identificando possíveis interações indesejadas entre as modificações.

Por fim, todas as alterações temporárias aplicadas ao projeto durante o processo são removidas, garantindo que o estado original seja restaurado sem impactar a base de código.

Fluxograma 1 – Solução geral



Fonte: Fluxograma elaborado pelo autor (2025).

4.2 Transformação de código

O processo de transformação do código segue uma abordagem sistemática para identificar e modificar as estruturas necessárias à detecção de conflitos semânticos. Primeiramente, organizamos e separamos os arquivos alterados no merge, identificando as linhas do código que foram adicionadas por left e aquelas que foram adicionadas por right. Com essas informações, iteramos sobre os arquivos modificados para realizar a transformação automática do código.

Durante essa iteração, cada arquivo é processado utilizando o JDT, que nos permite gerar uma AST do código. Essa AST representa a estrutura hierárquica do programa e será utilizada para identificar os nós que devem ser transformados. Para isso, a AST é passada a um Visitor, que percorre a árvore e determina quais elementos precisam ser modificados.

Ao visitar cada statement, verificamos se ele pertence a uma linha marcada como alterada por left ou right. Caso o *statement* não esteja em uma dessas categorias, ele é ignorado, e seus nós filhos não são visitados, ou seja, no código resultante o *statement* aparece como no código original, não sofre nenhuma transformação. Isso garante que apenas as alterações relevantes sejam analisadas, otimizando o processo de transformação.

Para ilustrar o funcionamento do algoritmo, considere a seguinte demonstração, onde `W` representa a classe `InferWrapper` e, para simplificar a explicação, omitiremos as chamadas dos métodos `l` e `r`. Segue a tabela 1 com as regras de transformações:

Tabela 1 – Transformação de Código

Visitors	**Wrapping**
VariableDeclarationFragment	`T id = exp → T id = W(exp)`
	`exp ∉ F ∪ C`, onde `F` representa o conjunto de chamada de método e `C` o conjunto de instância de classe.
Assignment	`id = exp → id = W(exp)`
	`exp.f = exp' → W(exp).f = W(exp')`
	`exp.f o= exp', onde o = {+, -, *, /, %} → exp.f = exp.f o exp' → W(exp).f = W(exp.f o exp')`

Visitors	Wrapping
	`exp ∉ F ∪ C`, onde `F` representa o conjunto de chamada de método e `C` o conjunto de instância de classe.
PrefixExpression	`o(exp), onde o = {+, -, ++, --, ~, !} → W(o(exp))`
PostfixExpression	`exp(o), onde o = {++,--} → W(exp(o))`
InfixExpression	`exp → exp [replace: id → W(id)]`
IfStatement	`if(id) body → if(W(id)) body`
WhileStatement	`while(id) body → while(W(id)) body`
ForStatement	`for(exp1; id; exp3) body → for(exp1; W(id); exp3) body`
ReturnStatement	`return exp → return W(exp)`
ClassInstanceCreation	`new C() → W(new C())`
	`new C(exp1, …, expn) → W(new C(W.l(exp1), W(...), W(expn)))`
	aplica o visitor ao construtor
MethodInvocation	`exp.m(exp1, …, expn) → W(exp).m(W(exp1), W(...), W(expn))`

Visitors	Wrapping
	aplica o visitor a declaração do método

Fonte: Tabela elaborada pelo autor (2025).

4.2.1 VariableDeclarationFragment Visit

O *VariableDeclarationFragment* é usado em declarações de campo, declarações de variáveis locais, inicializadores *ForStatement* e parâmetros *LambdaExpression* [7]. Esse nó é gerado quando uma variável é declarada e recebe um valor no momento da inicialização. A transformação é aplicada apenas se o inicializador do *VariableDeclarationFragment* não for `null`, ou seja, se houver de fato uma expressão à direita da atribuição. A transformação ocorre apenas no lado direito (*rhs* – *right-hand side*) da expressão porque é nesse ponto que o valor inicial da variável é definido. Em um *VariableDeclarationFragment*, a variável declarada ainda não possui um valor prévio, portanto, apenas a inicialização importa para a análise, pois é ela que contém o dado potencialmente transmitido. Alterar o lado esquerdo (*lhs* – *left-hand side*) não faria sentido, já que a variável em si não carrega informações até ser inicializada.

Por exemplo, em uma declaração simples com `int x = 10;`, a transformação será aplicada ao *rhs*, que é o valor literal `10`. Nesse caso, o valor será envolvido pela função `W()`, resultando na forma `int x = W(10);`. Isso permite que o Infer rastreie a propagação de dados sensíveis durante a execução do programa. Em uma declaração de campo como `private int id = obj.f;`, a transformação será aplicada ao *rhs*, ou seja, o valor do campo `obj.f`, sendo marcado como `W(obj.f)`.

Se o *rhs* é uma invocação de método (*MethodInvocation*) ou a criação de uma instância de classe (*ClassInstanceCreation*), a transformação não é aplicada nesse momento. Nessas situações, o tratamento da marcação ocorrerá durante os respectivos *visits* dessas operações na AST, garantindo que a análise do fluxo de dados seja realizada de forma consistente e precisa durante a execução.

4.2.2 Assignment Visit

Diferente do *VariableDeclarationFragment*, que ocorre no momento da inicialização de uma variável, o *Assignment* refere-se a atribuições subsequentes, onde o valor de uma variável ou campo é modificado em algum ponto posterior do código. Essa distinção é crucial, pois um *Assignment* pode representar tanto a substituição direta de um valor quanto operações compostas, como adição (`+=`), subtração (`-=`), multiplicação (`*=`), entre outras.

Na transformação aplicada, tanto o lado esquerdo (*lhs*) quanto o lado direito (*rhs*) de um *Assignment* são marcados. No caso de uma atribuição direta `id = exp`, a transformação resulta em `id = W(exp)`, garantindo que o Infer rastreie corretamente a propagação de dados sensíveis.

Para atribuições em campos de objetos, como `obj.f = exp`, a transformação ocorre como `W(obj).f = W(exp)`. Nesse caso, marcamos apenas a referência do objeto (`obj`) no *lhs*, pois, no Infer, quando um campo recebe um valor *tainted*, a referência daquele objeto já se torna *tainted* automaticamente. Assim, se `W` for um *wrapper* do tipo *source*, o Infer entenderá que o objeto foi contaminado e rastreará seu uso subsequente. Se `W` for um *wrapper* do tipo *sink*, o Infer verificará se o objeto já está *tainted* e, caso esteja, reportará um `TAINT_ERROR`, indicando um fluxo inseguro de dados. Dessa forma, a transformação garante que a análise de *taint* funcione corretamente sem redundâncias.

Quando a atribuição envolve operadores compostos, como `obj.f += exp`, a expressão é expandida para `obj.f = obj.f + exp`, e em seguida, a marcação é aplicada, resultando em `W(obj).f = W(obj.f + exp)`. Isso garante que tanto a leitura quanto a escrita do campo sejam analisadas pelo Infer.

Por fim, se o lado direito da atribuição for uma chamada de método (*MethodInvocation*) ou uma criação de objeto (*ClassInstanceCreation*), a marcação direta não ocorre nesse momento. Em vez disso, a transformação desses elementos

acontece durante seus respectivos visits dentro da AST, garantindo uma análise consistente e precisa.

4.2.3 PrefixExpression e PostfixExpression Visit

Na análise e transformação do código, é necessário lidar com operadores unários que modificam diretamente o valor de uma variável, como os operadores de incremento (`++`) e decremento (`--`). Essas operações podem ocorrer em duas formas: prefixada (*PrefixExpression*) e posfixada (*PostfixExpression*), ambas exigindo marcação para garantir que o Infer consiga rastrear corretamente a propagação dos dados.

No caso de um *PrefixExpression*, como `++(exp)` ou `--(exp)`, a operação ocorre antes da utilização do valor da variável. Para manter a rastreabilidade, a transformação aplicada encapsula a variável dentro do *wrapper*, resultando em `W(++(exp))` ou `W(--(exp))`. Os operadores relativos ao *PrefixExpression* são os operadores unários (`+`, `-`, `++`, `--`, `~`, `!`) [7,9]. Já no *PostfixExpression*, como `(exp)++` ou `(exp)--`, a variável é utilizada antes da modificação do seu valor. No entanto, a marcação segue a mesma lógica do prefixo, sendo transformada para `W((exp)++)` ou `W((exp)--)`. E os operadores relativos ao *PostfixExpression* são de incremento e decremento apenas (`++`, `--`) [7].

Essa marcação é essencial para garantir que todas as modificações no estado das variáveis sejam corretamente detectadas pela análise de fluxo de dados do Infer. Por exemplo, em um laço de repetição como `for (int i = 0; i < n; i++)`, a transformação aplicada garantirá que a iteração seja rastreada corretamente, resultando em *`for (int i = 0; i < n; W(i++))`*. Isso assegura que a variável de controle do laço continue sendo monitorada dentro da análise semântica do código.

4.2.4 InfixExpression Visit

O nó *InfixExpression* representa operações binárias dentro da AST, como somas (`+`), subtrações (`-`), comparações (`<`, `>`, `==`), entre outras [7]. No contexto da análise de conflitos semânticos, apenas expressões booleanas serão marcadas.

O objetivo desse *visit* é garantir que os identificadores utilizados nas condições lógicas de estruturas de controle sejam corretamente rastreados pelo Infer.

A transformação ocorre substituindo identificadores individuais dentro da expressão pela versão encapsulada no wrapper: `exp→exp[replace: id→W(id)]`. Por exemplo, em uma comparação simples entre duas variáveis `a > b`, a transformação aplicada resultará em `W(a) > W(b)`. Da mesma forma, em expressões mais complexas, como `x == y && z != u`, os identificadores são marcados individualmente, ficando `W(x) == W(y) && W(z) != W(u)`.

Visando apenas marcação de estruturas capazes de propagar dados, apenas identificadores (variáveis) são marcados, ou seja, valores constantes são ignorados. A marcação ocorre individualmente em cada identificador presente na expressão lógica. Essa transformação é aplicada dentro de expressões condicionais de *IfStatement*, *WhileStatement* e *ForStatement*, já que normalmente o nó de condição desses statements são *InfixExpressions*. Assim, um condicional simples como `while (count > 0)` será marcada como `while (W(count) > 0)`. Essa abordagem assegura que o Infer possa rastrear corretamente os identificadores usados ou lidos das expressões condicionais.

4.2.5 IfStatement, WhileStatement e ForStatement Visit

A transformação das estruturas de controle *IfStatement*, *WhileStatement* e *ForStatement* tem como objetivo garantir que o Infer possa rastrear corretamente os identificadores utilizados como condições diretas nesses blocos. No entanto, essa transformação só ocorre quando a condição é um identificador isolado, ou seja, não faz parte de uma expressão booleana composta. Isso acontece porque, em expressões booleanas mais complexas, a marcação já é realizada pelo *visit* de *InfixExpression*.

No caso do *IfStatement*, quando a condição do bloco for representada apenas por um identificador, ele será envolvido pelo wrapper para permitir que o Infer identifique o fluxo de dados sensível. Por exemplo, a estrutura `if (id) {...}` será transformada em `if (W(id)) {...}`, garantindo que a variável de controle seja rastreada corretamente. Da mesma forma, para o *WhileStatement*, se a condição for apenas um identificador, como em `while (id) {...}`, a

transformação resultará em `while (W(id)) {...}`, permitindo o monitoramento adequado das mudanças no estado da variável dentro do laço de repetição.

O ForStatement apresenta uma estrutura mais complexa, composta por quatro elementos: inicializadores, expressão condicional, atualizadores e corpo do laço que é um *Statement* [7,9]. Os inicializadores são tratados pelo *VariableDeclarationFragment Visit*, os atualizadores são contemplados pelo *Assignment Visit*, o corpo do laço não passa por nenhuma transformação específica nesse contexto porque a regra de transformação já abrange as expressões e *statements* dentro dele conforme a árvore sintática é percorrida. Como a classe `Visitor` continua a visitar os nós filhos do corpo, eventuais expressões ou atribuições relevantes já serão marcadas pelas visitas correspondentes, como *Assignment Visit* ou outras transformações aplicáveis. Dessa forma, a transformação ocorre apenas na expressão condicional se ela for um identificador isolado. Um exemplo disso é a estrutura `for (int i = 0; id; i++) {...}`, que será transformada em `for (int i = 0; W(id); i++) {...}`, assegurando que o Infer reconheça corretamente o fluxo de dados durante a execução do laço.

4.2.6 ReturnStatement Visit

A transformação do *ReturnStatement* garante que valores retornados sejam rastreados pelo Infer. Sempre que uma expressão for retornada, ela será envolvida pelo *wrapper* correspondente. Assim, um retorno como `return exp;` será transformado em `return W(exp);`, permitindo a identificação correta do fluxo de dados sensíveis.

4.2.7 ClassInstanceCreation Visit

A transformação do *ClassInstanceCreation* garante que instâncias de classes criadas em trechos alterados por *left* ou *right* sejam rastreadas pelo Infer. A marcação sempre é aplicada, mas o wrapping da instanciação só ocorre quando o nó pai é um *VariableDeclarationFragment* ou um *Assignment*, pois nesses casos o objeto instanciado está sendo armazenado em uma variável.

Quando uma classe é instanciada, se o nó pai for um *VariableDeclarationFragment* ou um *Assignment*, o objeto resultante será envolvido pelo *wrapper*. Assim, uma criação simples como `new C()` será transformada em `W(new C())`. Caso a instanciação possua argumentos, todos eles também são marcados, independentemente do nó pai: `new C(exp1, exp2)` se torna `new C(W(exp1), W(exp2))`.

Além disso, ocorre uma visita interprocedural ao construtor da classe instanciada, o que corresponde ao nó *MethodDeclaration* [7], garantindo a continuidade da marcação dentro da classe. Essa visita acontece independentemente do nó pai da instanciação, assegurando que os fluxos de dados sensíveis sejam corretamente propagados. Para evitar marcações duplicadas, verifica-se se o construtor já foi marcado por *left* ou *right*; se já tiver sido marcado por left, apenas novas marcações por right serão permitidas, e vice-versa.

Durante a visita interprocedural, o algoritmo é executado recursivamente, e a marcação é atribuída a quem originou a alteração. Dessa forma, todos os nós internos ao método são marcados conforme a origem da chamada. Se a alteração foi realizada por *left*, todos os nós são marcados como *left*, se por *right* de igual modo. A profundidade da visita é passada como parâmetro, permitindo o controle sobre até onde a análise interprocedural deve se estender.

4.2.8 MethodInvocation Visit

O comportamento dessa transformação é semelhante ao do *ClassInstanceCreation*. O *wrapping* da chamada de método ocorre de forma diferente dependendo do nó pai. Se o nó pai for um *VariableDeclarationFragment* ou um *Assignment*, significa que o valor retornado pelo método está sendo armazenado, então a transformação encapsula toda a chamada: `exp.m(exp1, exp2)→W(W(exp).m(W(exp1),W(exp2)))`. Caso contrário, quando a chamada não possui um destinatário para o retorno, a marcação ocorre apenas nos elementos internos da invocação, sem o encapsulamento externo: `exp.m(exp1, exp2)→W(exp).m(W(exp1),W(exp2))`.

Além disso, cada chamada de método dispara uma visita interprocedural ao seu *MethodDeclaration*, garantindo que a marcação se propague dentro do método

chamado. Antes de realizar essa visita, verifica-se se o *MethodDeclaration* já foi visitado por *left* ou *right* para evitar marcações duplicadas. Se já houver sido marcado por *left*, apenas novas marcações por *right* serão permitidas, e vice-versa.

Para chamadas encadeadas (*method chaining*), a marcação ocorre apenas na chamada externa caso o nó pai da cadeia seja um *VariableDeclarationFragment* ou um *Assignment*. Considerando a chamada encadeada: `obj.m1().m2().m3()`. Se o nó pai for uma atribuição ou declaração de variável, apenas a chamada externa recebe o *wrapping*: `W(W(obj).m1().m2().m3())`. Isso ocorre porque, dentro da estrutura de *method chaining*, o nó pai de `obj.m1().m2()` é `obj.m1().m2().m3()`, que por sua vez também é um *MethodInvocation*. Portanto, a marcação externa ocorre apenas no nível mais alto da cadeia.

Porém, se a chamada encadeada não estiver associada a uma atribuição ou declaração de variável, a marcação será aplicada apenas internamente aos argumentos e à instância: `W(obj).m1().m2().m3()`.

4.3 Arquitetura

O processo de detecção de conflitos semânticos pode ser dividido em três fases principais: captura, transformação e análise. Na fase de captura, identificamos as linhas adicionadas por *left* e *right* a partir do histórico de commits, associando-as aos arquivos modificados. Na fase de transformação, os arquivos passam por ajustes na AST para que o Infer consiga analisar corretamente a propagação de dados, aplicando regras específicas de *wrapping* para criar uma a estrutura do código que servirá de entrada para as análises. Por fim, na fase de análise, o Infer executa a detecção de conflitos semânticos, gerando relatórios para avaliação.

4.3.1 Fase de Captura

O processo se inicia na classe `StaticAnalysisMerge` uma classe que foi adaptada do SSM (Static Semantic Merge) [25]. Essa classe utiliza uma função da classe `ModifiedLinesManager`, pertencente ao Mining [8], para obter a relação de linhas alteradas por *left* e *right*, com base nos commits base, *left*, *right* e *merge*. A função foi adaptada para retornar as linhas adicionadas por L ou R, agrupadas por

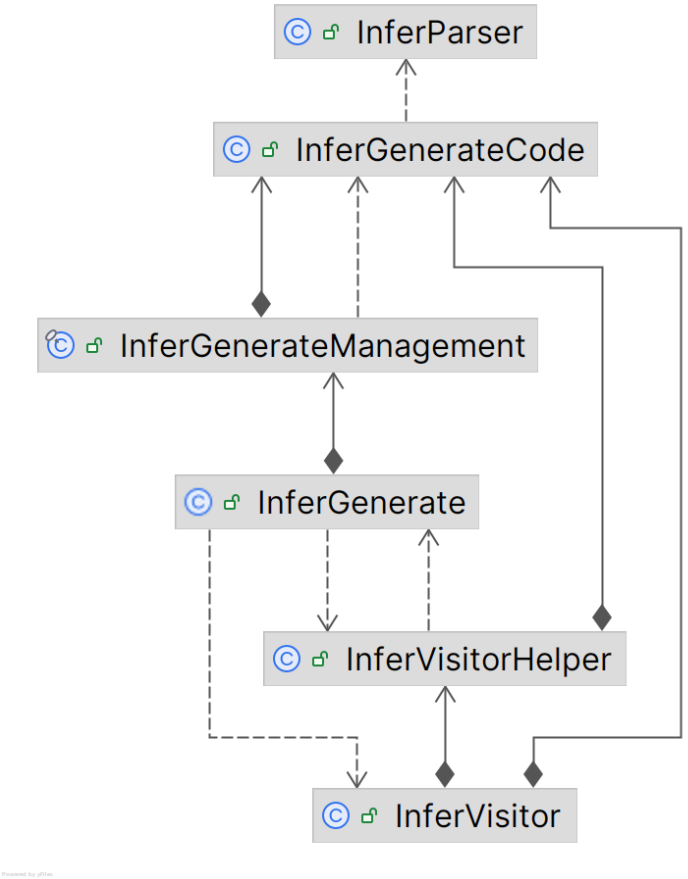
arquivos. Como resultado, obtém-se uma lista onde cada elemento contém o caminho do arquivo modificado e a respectiva relação de linhas adicionadas por cada lado. Esse output serve como entrada para o módulo de transformação.

4.3.2 Fase de Transformação

A fase de transformação tem como principal objetivo modificar a AST dos arquivos identificados no processo anterior, garantindo que as alterações sejam aplicadas corretamente de acordo com regras pré-definidas. Para isso, sua arquitetura é composta por diversas classes que trabalham de forma integrada para realizar essa transformação de maneira estruturada.

O processo se inicia na classe `InferGenerate`, que recebe como entrada a lista de arquivos e suas respectivas linhas modificadas, identificadas na etapa anterior. Essa classe percorre através do método `generateInferCodeForEachCollectedMergeData` a lista de arquivos e, para cada um deles, verifica se já existe uma instância de `InferGenerateCode` associada ao arquivo em questão. Caso a instância ainda não tenha sido criada, ela é gerada no momento da iteração. Esse controle é feito pela classe `InferGenerateManagement`, um gerenciador implementado como um *singleton*. O propósito desse gerenciador é evitar que um mesmo arquivo seja transformado mais de uma vez dentro do processo, o que poderia resultar em sobrescrita de alterações previamente realizadas. Dessa forma, o `InferGenerateManagement` assegura que, se um arquivo já tiver sido processado, a mesma instância de `InferGenerateCode` seja reutilizada.

Figura 5 - Diagrama de classe do Transformador



Fonte: Imagem elaborada pelo autor (2025).

A classe `InferGenerateCode` é um dos elementos centrais do transformador, pois é responsável pelo gerenciamento da AST do código-fonte. Para cumprir esse papel, ela contém dois atributos fundamentais: (1) `CompilationUnit` [26] Representa a AST do código e possibilita a análise estrutural do programa . Esse atributo permite percorrer a estrutura sintática do código-fonte, identificar seus componentes e mapear as linhas às quais cada nó pertence. Dessa forma, é possível verificar se um determinado nó faz parte de uma linha modificada por *left* ou *right*, garantindo que as transformações sejam aplicadas corretamente; (2) `ASTRewrite` [27] que é responsável pelo gerenciamento das modificações feitas nos nós da AST. Esse atributo mantém um histórico de eventos que registram todas as transformações aplicadas ao código, garantindo que as mudanças sejam organizadas e processadas corretamente antes de serem gravadas no arquivo.

Após obter a instância de `InferGenerateCode`, o `InferGenerate` inicializa as classes `InferVisitorHelper` e `InferVisitor`. Essas classes

percorrem a árvore sintática do programa e aplicam as regras de transformação definidas. O `InferVisitor` é o responsável por visitar cada nó da AST, analisando sua estrutura e determinando se alguma modificação precisa ser aplicada. Quando uma alteração é necessária, a transformação é realizada conforme as regras estabelecidas na Tabela 2, garantindo que o código seja corretamente instrumentado.

Além da transformação direta nos arquivos modificados por *left* e *right*, o processo também realiza uma transformação interprocedural para garantir que alterações propagadas por chamadas de métodos e instâncias de classes sejam devidamente analisadas. Durante o *visiting*, ao encontrar um nó que representa uma chamada de método ou a instância de uma classe, é realizado um *binding* utilizando o Eclipse JDT para localizar o arquivo onde o método ou construtor está declarado. A partir dessa informação, uma nova função do `InferGenerate` é iniciada, reduzindo a profundidade da análise. Esse processo é semelhante ao que ocorre no método `generateInferCodeForEachCollectedMergeData`, mas com uma diferença fundamental: os nós que pertencem ao corpo da declaração são tratados pelo `InferVisitorHelper` como se tivessem sido todos modificados pelo responsável da chamada do método ou instanciação da classe. Assim, ao percorrer esses nós na AST, as regras de transformação da Tabela 2 são aplicadas considerando que todos elementos analisados foram modificados por *left* ou *right*, garantindo consistência na propagação das transformações ao longo da estrutura do código.

Para evitar que a análise interprocedural se torne infinita ou alcance partes do código irrelevantes, dois critérios de parada são adotados nessa função recursiva: a profundidade da análise não pode ser menor que zero e o caminho do arquivo identificado deve existir. Se qualquer um desses critérios não forem atendidos, a função encerra a busca e evita a propagação desnecessária da transformação.

4.3.3 Fase de Análise

A fase de análise é realizada pela classe `InferAnalysis` e tem como objetivo executar o Infer para identificar possíveis conflitos semânticos nos arquivos

transformados. Essa etapa é conduzida por meio da execução de comandos utilizando a classe ProcessBuilder [28].

O primeiro comando executado é responsável pela compilação do projeto, configurando o Infer para operar na fase de captura. Durante essa etapa, o Infer analisa o código-fonte e gera uma representação interna que servirá como base para as próximas fases da análise.

Em seguida, são realizadas duas execuções do Infer, cada uma utilizando um arquivo de configuração específico. O segundo comando executa a análise considerando o arquivo *infer_config_left_to_right.json*, que direciona a detecção de fluxos de dados e possíveis conflitos da esquerda para a direita. O terceiro comando segue a mesma lógica, mas utilizando o arquivo *infer_config_right_to_left.json*, analisando os fluxos na direção oposta.

Por fim, os *reports* gerados pelo Infer e toda a saída da análise são coletados e armazenados em um diretório interno da ferramenta, garantindo que os resultados possam ser consultados posteriormente. Após essa coleta, todas as modificações feitas no código durante a transformação são revertidas, restaurando o projeto ao seu estado original.

5 Resultados

O software desenvolvido foi avaliado em dois aspectos principais: (1) a consistência na transformação do código e (2) a precisão na detecção de conflitos semânticos. Para demonstrar esses pontos, utilizamos o exemplo motivacional do projeto (Figura 1) e sua correspondente transformação, conforme apresentado na Figura 6.

Figura 6 – Motivating Example Transformed

```
1.  public class MotivatingExample {
2.      private String text;
3.
4.      public void cleanText(){
5.          normalizeWhiteSpace(); //left
6.          removeComments();
7.          removeDuplicateWords(); //right
8.      }
9.
10.     private void normalizeWhiteSpace(){
11.         text = InferWrapper.l(text.trim().replaceAll(InferWrapper.l(" +"), InferWrapper.l(" "));
12.     }
13.
14.     private void removeComments(){...}
15.
16.     private void removeDuplicateWords(){
17.         String[] words = InferWrapper.r(text.split(InferWrapper.r("\\s+")));
18.         Stack<String> st = InferWrapper.r(new Stack<>());
19.         for (int i = InferWrapper.r(0); InferWrapper.r(i) < InferWrapper.r(words.length);
    InferWrapper.r(i++)) {
20.             if (InferWrapper.r(st).isEmpty() ||
    InferWrapper.r(!st.peek().equals(InferWrapper.r(words[i])))) {
21.                 InferWrapper.r(st).push(InferWrapper.r(words[i]));
22.             } else {
23.                 InferWrapper.r(st).push(InferWrapper.r(" "));
24.             }
25.         }
26.         text = InferWrapper.r(String.join(InferWrapper.r(" "), InferWrapper.r(st)));
27.     }
28. }
```

Fonte: Imagem elaborada pelo autor (2025).

Primeiramente identificamos as diferenças entre as versões *left* e *right* do código utilizando funções específicas do Mining [8]. Esse processo permitirá extrair os arquivos modificados em cada versão, bem como identificar as linhas dos *statements* alterados. A partir dessas informações, será possível determinar quais nós do código precisam ser transformados A transformação dos nós na AST ocorre de forma precisa, com todos os elementos relevantes sendo corretamente marcados pelo `InferWrapper`. Essa consistência garante que o fluxo de dados sensíveis seja rastreado em diferentes partes do programa.

No exemplo apresentado, o Infer identifica com sucesso um fluxo de dado sensível (`TAINT_ERROR`) partindo da chamada do método `normalizeWhiteSpace` em direção ao método `removeDuplicateWords`. Esse erro ocorre porque a ferramenta identifica que o campo `text` foi marcado como *tained* e foi propagado ao longo do código até atingir um ponto de *sink* configurado no Infer.

A lógica por trás dessa detecção pode ser explicada da seguinte maneira. Durante a etapa de transformação de código, a ferramenta identifica que a chamada ao método `normalizeWhiteSpace()` foi adicionada na versão *left*, enquanto `removeDuplicateWords()` foi introduzido na versão *right*. Como resultado, os nós do corpo do método `normalizeWhiteSpace()` são encapsulados com `InferWrapper.l()`, e os de `removeDuplicateWords()` são empacotados com `InferWrapper.r()`, conforme ilustrado na Figura 5.

O campo `text` é inicialmente marcado na linha 11 pela transformação aplicada em *left*, utilizando `InferWrapper.l()`. No arquivo *infer_config_left_to_right.json*, esse wrapper é configurado como um *source*, o que significa que qualquer valor passado para ele será tratado pelo Infer como um dado sensível que precisa ser rastreado ao longo da execução do programa.

Posteriormente, `text` é utilizado dentro do método `removeDuplicateWords()`, onde ocorre a chamada `text.split(...)` na linha 16. De acordo com a transformação aplicada, essa expressão é convertida para `InferWrapper.r(text.split(InferWrapper.r(" ")))`. Como `InferWrapper.r()` foi configurado como um *sink* na configuração de análise, o Infer verifica se os dados que chegam a ele estão marcados como sensíveis. Dado que `text` foi previamente empacotado como source em *left*, ao atingir o *sink* na primeira execução, o Infer detecta um fluxo de dados sensível não sanitizado e, consequentemente, reporta um `TAINT_ERROR` como mostrado na Figura 7.

Figura 7 – Infer error report

```
#0
src/main/java/org/example/MotivatingExample.java:7: error: Taint
Error
  Built-in Simple taint kind, matching any Simple source with any
Simple sink except if any Simple sanitizer is in the way.
```

```
`this->text` is tainted by value passed as argument `#0` to `Object
InferWrapper.l(Object)` with kind `Simple` and flows to value
passed as argument `#0` to `Object InferWrapper.r(Object)` with
kind `Simple`.
   5.           normalizeWhiteSpace(); //left
   6.           removeComments();
   7. >         removeDuplicateWords(); //right
   8.       }
   9.

Found 1 issue
  Issue Type(ISSUED_TYPE_ID): #
    Taint Error(TAINT_ERROR): 1
```

Fonte: Imagem elaborada pelo autor (2025).

A análise evidencia que o processo de transformação aplicado pelo software não altera a lógica original do programa. a preservação da semântica ocorre porque os wrappers adicionados na transformação não modificam o comportamento do programa. Eles apenas encapsulam as expressões sem alterar seus valores, pois simplesmente retornam o parâmetro recebido. Dessa forma, a lógica original do código permanece inalterada, garantindo que a execução produza os mesmos resultados enquanto permite ao Infer detectar com precisão interações críticas entre as alterações provenientes de diferentes branches.

A avaliação da ferramenta foi realizada utilizando o conjunto de cenários de merge de projetos Java catalogados pelo Software Productivity Group (SPG) [10]. Foram selecionados 18 commits de merge provenientes de 9 projetos distintos. Destes, apenas 6 commits foram compilados com sucesso. A compilação foi realizada utilizando as ferramentas Gradle ou Maven, sendo importante destacar que o Infer não realiza a compilação diretamente; em vez disso, ele intercepta e coleta os fluxos de dados durante o processo para posterior análise.

Os 12 commits que falharam na compilação apresentaram erros independentes das transformações realizadas pela ferramenta, sendo majoritariamente causados por problemas com dependências externas ou configurações específicas do ambiente de construção dos projetos.

Embora a compilação não tenha sido bem-sucedida em todos os casos, foi possível validar a consistência da transformação de código (ou *wrapping*) por meio

da análise estática da IDE IntelliJ. Consideramos uma transformação bem-sucedida quando, dentre os arquivos alterados do projeto, não havia nenhuma sinalização de erro de sintaxe. Esse critério foi adotado especialmente porque parte dos projetos não pôde ser compilada. Em 17 dos 18 commits, as transformações foram aplicadas corretamente, sem gerar erros de sintaxe detectados pela IDE. O único caso em que a transformação falhou ocorreu devido a um problema de compatibilidade de tipos em uma asserção. Especificamente, no projeto okhttp na linha: `assertEquals(InferWrapper.r('a'), buffer.getByte(InferWrapper.r(0)));`, gerou o erro mostrado na Figura 8.

Figura 8 – Erro de assertion

```
reference to assertEquals is ambiguous both method
assertEquals(long,long) in Assert and method
assertEquals(Object,Object) in Assert match
```

Fonte: Imagem elaborada pelo autor (2025).

Esse erro ocorre porque o compilador interpreta `InferWrapper.r('a')` como um objeto (`Object`), enquanto `buffer.getByte(InferWrapper.r(0))` retorna um valor do tipo `byte`. O Java possui sobrecargas do método `assertEquals` tanto para tipos primitivos (`long, long`) quanto para objetos (`Object, Object`). No entanto, ao encapsular o caractere `'a'` com `InferWrapper.r()`, o tipo primitivo char é convertido em `Object`, tornando a chamada ambígua, pois o compilador não consegue decidir entre as duas versões do método assertEquals. Esse problema ilustra uma limitação da transformação, onde a introdução do wrapper interfere na inferência de tipos do Java, resultando em incompatibilidades que não existiriam no código original.

A quantidade de *reports* do tipo `TAINT_ERROR` gerados pelo Infer pode variar significativamente devido a diversos fatores. Um dos principais fatores é a quantidade de marcações realizadas, que está diretamente relacionada à profundidade da análise interprocedural. No contexto da ferramenta desenvolvida, a profundidade da análise foi configurada para um limite de 5. Quanto maior a profundidade e a quantidade de chamada de métodos ou instâncias de classes analisados, maior será o número de nós transformados. Consequentemente, mais fluxos sensíveis podem ser identificados e reportados pelo Infer. Esse

comportamento é esperado, já que a análise interprocedural expande o alcance da detecção para além do método diretamente envolvido, permitindo rastrear fluxos de dados que atravessam diferentes partes do programa.

Outro fator que contribui para a variação no número de *reports* é a interpolação de marcações, que ocorre quando tanto *left* quanto *right* adicionam uma mesma chamada de método ou instanciam uma mesma classe em diferentes partes do código. Nesse cenário, a transformação interprocedural é aplicada a todas as declarações de método ou construtor associadas a essas operações, gerando uma sobreposição de wrappings. Essa sobreposição causa a multiplicação dos *reports* do tipo `TAINT_ERROR`, uma vez que o Infer interpreta cada fluxo interpolado como um possível risco individual.

Um exemplo claro desse comportamento ocorre no projeto Jsoup, no commit `a44e18aa3c`. Apesar de esse commit envolver um número relativamente pequeno de arquivos alterados no merge, ele gerou um volume elevado de *reports* de `TAINT_ERROR`. A causa principal desse aumento foi a combinação da análise interprocedural aprofundada com a interpolação de marcações em pontos críticos do código. Esse caso evidencia como a quantidade de *reports* pode ser inflacionada quando múltiplas transformações ocorrem em métodos compartilhados entre as duas ramificações do merge.

Um efeito colateral interessante da abordagem proposta é que, além da detecção de `TAINT_ERROR`, o Infer também foi capaz de identificar outros possíveis erros, como Null Dereference e Resource Leak, como pode ser observado na Tabela 2.

Tabela 2 – Avaliação da ferramenta em agluns cenários do merge-dataset

Project	Commit	Transformation	compiled	Report
jsoup	3f7d2c71db	yes	no	-
jsoup	a44e18aa3c	yes	yes	Left -> Right Taint Error(TAINT_ERROR): 1146 Null Dereference(NULLPTR_DEREFERENCE): 1 Right -> Left Taint Error(TAINT_ERROR): 1368 Null Dereference(NULLPTR_DEREFERENCE): 1
jsoup	a8b6982de9	yes	no	-
jsoup	fee4762322	yes	no	-

elasticsearch	0404db65e3	yes	no	-
elasticsearch	36884807b3	yes	no	-
elasticsearch	3764b3ff80	yes	no	-
elasticsearch	59cb67c7bd	yes	no	-
okhttp	1151c9853c	no	no	-
retrofit	2b6c719c66	yes	yes	Left -> Right Taint Error(TAINT_ERROR): 4 Right -> Left Taint Error(TAINT_ERROR): 2
retrofit	71f622ce51	yes	yes	Left -> Right Taint Error(TAINT_ERROR): 42 Right -> Left Taint Error(TAINT_ERROR): 38
vavr	204635d915	yes	no	-
crawler4j	6fdb8f27b5	yes	no	-
cloud-slang	20bac30d9b	yes	no	-
Storm	ad2be67883	yes	yes	Right -> Left Taint Error(TAINT_ERROR): 2
titan	04edd7f0e7	yes	yes	Left -> Right Taint Error(TAINT_ERROR): 288 Resource Leak(PULSE_RESOURCE_LEAK): 1 Null Dereference(NULLPTR_DEREFERENCE): 1 Right -> Left Taint Error(TAINT_ERROR): 215 Resource Leak(PULSE_RESOURCE_LEAK): 1 Null Dereference(NULLPTR_DEREFERENCE): 1
titan	387c16ea05	yes	yes	Left -> Right Taint Error(TAINT_ERROR): 6 Resource Leak(PULSE_RESOURCE_LEAK): 1 Null Dereference(NULLPTR_DEREFERENCE): 1 Right -> Left Taint Error(TAINT_ERROR): 2 Resource Leak(PULSE_RESOURCE_LEAK): 1 Null Dereference(NULLPTR_DEREFERENCE): 1

Fonte: Tabela elaborada pelo autor (2025).

6 Limitações

Um ponto crítico observado durante o desenvolvimento é a aplicação das transformações nos nós da AST. As alterações são registradas como eventos pelo ASTRewriter, ele coleta descrições de modificações em nós e traduz essas descrições em edições de texto que podem ser aplicadas à fonte original. O ponto principal é que tudo isso é feito sem realmente modificar o AST original [7]. e somente em um estágio posterior essas modificações são efetivamente persistidas no arquivo analisado. No entanto, esse mecanismo pode resultar na sobrescrita de alterações, pois um nó pai pode substituir modificações feitas em seus nós filhos durante a reescrita da AST. Para mitigar esse problema, adotamos um fluxo de transformação *bottom-up* para *MethodInvocation* e *ClassInstanceCreation*, onde as modificações começam pelos nós mais internos da árvore. Antes de modificar um nó pai, recuperamos as propriedades alteradas nos nós filhos, garantindo que essas modificações sejam preservadas ao atualizar a estrutura hierárquica da AST. Essa estratégia é especialmente importante para preservar marcações dos argumentos mais internos ao mais externos do nó, que poderiam ser perdidos caso a transformação seguisse um fluxo *top-down*.

O Infer adota uma abordagem conservadora na propagação de *tainted objects* para reduzir falsos positivos. Essa decisão de design afeta diversas situações, como chamadas a métodos desconhecidos e operações aritméticas.

Quando o Infer não consegue interpretar o código-fonte de um método, como no caso de bibliotecas externas ou métodos resolvidos indiretamente (como em chamadas via ponteiro de função), ele o classifica como *unknown_method*. Isso ocorre porque o Infer não tem acesso ao código-fonte completo, seja devido a métodos de terceiros, onde apenas a assinatura é conhecida, ou devido à resolução indireta do método. Nesses casos, objetos sensíveis passados como argumentos para esses métodos não são propagados, interrompendo o rastreamento do fluxo de dados sensíveis. Essa abordagem conservadora ajuda a evitar falsos positivos, pois o Infer assume que não se pode propagar dados sensíveis dentro desses métodos. No entanto, isso também pode gerar falsos negativos, já que o fluxo de dados sensíveis não é rastreado [23]. Para contornar essa limitação nas experimentações, esse comportamento foi manualmente desativado no código-fonte do Infer,

permitindo a propagação mesmo em métodos desconhecidos, garantindo uma análise mais precisa do fluxo de dados sensíveis.

Além disso, o Infer não mantém a marcação de *taint* em expressões aritméticas. Por exemplo, no código `int x = W.l(10); int y = W.r(x + 5);`, não será reportado um `TAINT_ERROR` entre as versões. No entanto, em casos de concatenação de strings, a ferramenta realiza a propagação corretamente e gera o *report* adequado. Esse comportamento limita a detecção de conflitos quando modificações aritméticas estão envolvidas.

Além disso, a estratégia de *wrapping* utilizada na transformação consiste em encapsular valores por meio de uma classe auxiliar (`InferWrapper`) com dois métodos de argumento e retorno genérico (`l` e `r`). Embora essa abordagem seja simples, a classe precisa estar presente em cada projeto analisado para que as transformações sejam válidas. Para facilitar essa integração, a classe foi manualmente importada nos projetos-alvo utilizando os arquivos de configuração do Maven e do Gradle.

Por fim, após a aplicação de todas as transformações necessárias no código, ainda é preciso compilar o projeto utilizando o Infer para viabilizar a execução da análise [4]. Esse processo adiciona um custo computacional adicional, pois a ferramenta precisa processar os arquivos transformados juntamente com o Infer. Essa etapa final reforça a necessidade de otimizações para minimizar o impacto do tempo de processamento e garantir que a solução seja viável em cenários reais.

7 Trabalhos relacionados

A detecção de conflitos semânticos tem sido abordada na literatura por meio de diferentes estratégias, incluindo análise dinâmica, geração automática de testes e análise estática. Enquanto algumas abordagens priorizam a observação de interações em tempo de execução, outras exploram técnicas de análise estática para identificar interferências semânticas sem necessidade de execução do código.

Moraes et al. [3] propõem uma estratégia baseada em análise dinâmica para identificar conflitos semânticos em código JavaScript. Utilizando o framework Jalangi2, a abordagem rastreia operações de escrita em elementos de estado compartilhados durante a execução do código integrado, permitindo detectar interferências semânticas sem a necessidade de testes unitários explícitos. Essa técnica se destaca pela eficiência e pela baixa taxa de falsos positivos, pois os conflitos são identificados com base em acessos reais ao estado do programa.

Já Maciel et al. [2] exploram a detecção de conflitos semânticos em múltiplos métodos e classes por meio da ferramenta SMAT, que se baseia na geração automática de testes unitários. Considerando quatro versões do código-base, esquerda, direita e *merge*, a ferramenta executa os testes gerados para cada versão e compara seus resultados, evidenciando mudanças comportamentais inesperadas. Essa abordagem permite detectar conflitos de maneira estruturada, mas pode ser limitada pela cobertura dos testes gerados automaticamente.

No contexto da análise estática, o trabalho de Santos de Jesus et al. [1] é particularmente relevante, pois emprega técnicas como Interprocedural Data Flow (DF) para detectar interferências sem a necessidade de executar o código. Essa estratégia possibilita identificar conflitos semânticos de forma antecipada, antes da execução, apresentando um bom equilíbrio entre recall e precisão. No entanto, a abordagem enfrenta desafios relacionados ao custo computacional da análise, especialmente em projetos de grande porte.

O presente trabalho inspira-se na abordagem de análise estática, mas propõe uma adaptação do Infer para rastrear fluxo de dados. Ao encapsular as modificações utilizando métodos específicos, a solução busca uma forma mais eficiente de identificar conflitos semânticos, aproveitando a capacidade do Infer para realizar análises precisas antes da execução do código.

8 Conclusão

O objetivo principal deste trabalho foi investigar e aplicar técnicas para a detecção de conflitos semânticos em programas de software, com ênfase em *dataflow*. Ao longo do trabalho, exploramos a utilização do Infer, uma ferramenta de análise estática, para identificar e rastrear a propagação de dados sensíveis, propondo transformações no código-fonte para permitir uma análise mais eficaz. Desenvolvemos um mecanismo automatizado de transformação de código utilizando a API JDT do Eclipse, permitindo marcar sistematicamente as alterações realizadas por diferentes desenvolvedores. Com essa adaptação, o Infer foi capaz de rastrear e identificar fluxos de dados sensíveis que indicam potenciais conflitos semânticos.

Os resultados experimentais demonstraram a viabilidade da solução em identificar interferências em cenários reais de merges em projetos de código aberto. A avaliação realizada em múltiplos commits de diferentes projetos evidenciou a consistência da transformação do código e a capacidade do Infer em detectar fluxos de dados sensíveis, gerando relatórios detalhados de possíveis conflitos. Embora a adaptação tenha mostrado eficácia, algumas limitações foram observadas, como a dificuldade na propagação de dados em operações aritméticas e a necessidade de ajustes manuais em ambientes de compilação complexos.

Entre as principais contribuições deste trabalho, destacam-se: a adaptação da Análise de Taint do Infer para a detecção de conflitos semânticos em fluxos de dados; a automatização do processo de marcação de código utilizando a API JDT; e a validação empírica da abordagem em um conjunto diversificado de projetos. Essas contribuições ampliam o escopo de utilização do Infer em análises estáticas avançadas, oferecendo uma alternativa eficiente para a detecção de interferências em merges de código.

Como trabalhos futuros, sugere-se: procurar maneiras de suprir as limitações identificadas na abordagem proposta; integrar a ferramenta em pipelines de integração contínua para automatizar a detecção de conflitos semânticos em fluxos de desenvolvimento ágeis; e avaliar a ferramenta em termos de desempenho, analisando seu impacto em projetos de diferentes tamanhos e complexidades.

Em síntese, este estudo avançou na detecção automatizada de conflitos semânticos dinâmicos, oferecendo uma solução adaptável e eficiente para ambientes de desenvolvimento colaborativo.

REFERÊNCIAS

[1] DE JESUS, Galileu Santos; BORBA, Paulo; BONIFÁCIO, Rodrigo; OLIVEIRA, Matheus Barbosa de. Detecting semantic conflicts using static analysis. 2023. Disponível em: https://doi.org/10.48550/arXiv.2310.04269. Acesso em: 3 mar. 2025.

[2] MACIEL, Toni; BORBA, Paulo; DA SILVA, Léuson; BURITY, Thaís. Explorando a detecção de conflitos semânticos nas integrações de código em múltiplos métodos. 2024. Disponível em: https://doi.org/10.5753/sbes.2024.3348. Acesso em: 3 mar. 2025.

[3] MORAES, Amanda; BORBA, Paulo; DA SILVA, Léuson. Semantic conflict detection via dynamic analysis. 2024. Disponível em: https://doi.org/10.5753/sblp.2024.3471. Acesso em: 3 mar. 2025.

[4] META. Infer: A static analyzer for Java, C, C++, and Objective-C. Disponível em: https://fbinfer.com/. Acesso em: 3 mar. 2025.

[5] Susan Horwitz, Jan Prins, and Thomas Reps. 1989. Integrating noninterfering versions of programs. ACM Transactions on Programming Languages and Systems (TOPLAS) 11, 3 (1989), 345–387. https://doi.org/10.1145/65979.65980

[6] DE OLIVEIRA, Henrique Gomes. infer-data-flow. 2025. Disponível em: https://github.com/riqueGo/infer-data-flow. Acesso em: 3 mar. 2025.

[7] ECLIPSE FOUNDATION. Eclipse JDT. Disponível em: https://projects.eclipse.org/projects/eclipse.jdt. Acesso em: 3 mar. 2025.

[8] SPGROUP. MiningFramework. 2025. Disponível em: https://github.com/spgroup/miningframework. Acesso em: 3 mar. 2025.

[9] GOSLING, James; JOY, Bill; STEELE, Guy; BRACHA, Gilad; BUCKLEY, Alex. The Java® Language Specification, Java SE 8 Edition. Oracle, 2015. Disponível em: https://docs.oracle.com/javase/specs/jls/se8/html/index.html. Acesso em: 3 mar. 2025.

[10] SPGROUP. Sample Semantic Conflicts – mergedataset. 2025. Disponível em: https://github.com/spgroup/mergedataset/blob/master/semantic-conflicts/sample-semantic-conflicts.csv. Acesso em: 3 mar. 2025.

[11] BINKLEY, David; HORWITZ, Susan; REPS, Thomas. Program integration for languages with procedure calls. ACM Transactions on Software Engineering and Methodology, New York, v. 4, n. 1, p. 3–35, jan. 1995. Disponível em: https://doi.org/10.1145/201055.201056. Acesso em: 3 mar. 2025.

[12] DE JESUS, Galileu Santos; BORBA, Paulo; BONIFÁCIO, Rodrigo; BARBOSA, Matheus. Lightweight semantic conflict detection with static analysis. ICSE Companion – Proceedings of the 46th International Conference on Software Engineering (ICSE 2024), 2024, p. 340–343.

[13] BARBOSA, Matheus; BORBA, Paulo; BONIFÁCIO, Rodrigo; SANTOS, Galileu. Semantic conflict detection with overriding assignment analysis. Proceedings of the XXXVI Brazilian Symposium on Software Engineering, 2022, p. 435–445.

[14] DAMASCENO, Alexandre; ROCHA, Thiago; SOUTO, Eduardo. TaintJSec: Um Método de Análise Estática de Marcação em Código JavaScript para Detecção de Vazamento de Dados Sensíveis. In: SIMPÓSIO BRASILEIRO DE SEGURANÇA DA INFORMAÇÃO E DE SISTEMAS COMPUTACIONAIS (SBSEG), 18. , 2018, Natal. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2018 . p. 196-209. DOI: https://doi.org/10.5753/sbseg.2018.4253.

[15] ENCK, William; GILBERT, Peter; HAN, Seungyeop; TENDULKAR, Vasant; CHUN, Byung-Gon; COX, Landon P.; JUNG, Jaeyeon; MCDANIEL, Patrick; SHETH, Anmol N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computational Systems, New York, v. 32, n. 2, art. 5, jun. 2014. Disponível em: https://doi.org/10.1145/2619091. Acesso em: 3 mar. 2025.

[16] JETBRAINS. Taint analysis. 2025. Disponível em: https://www.jetbrains.com/help/qodana/taint-analysis.html. Acesso em: 26 mar. 2025.

[17] R2C. Semgrep: Taint mode. 2025. Disponível em: https://semgrep.dev/docs/writing-rules/data-flow/taint-mode. Acesso em: 26 mar. 2025.

[18] SILVA, Leuson Da; BORBA, Paulo; MAHMOOD, Wardah; BERGER, Thorsten; MOISAKIS, João. Detecting semantic conflicts via automated behavior change detection. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), 2020. Proceedings [...]. IEEE, 2020. p. 174–184. Disponível em: https://doi.org/10.1109/ICSME46990.2020.00026. Acesso em: 28 mar. 2025.

[19] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. 2013. Early detection of collaboration conflicts and risks. IEEE Transactions on Software Engineering 39, 10 (2013), 1358–1375. https://doi.org/10.1109/TSE.2013.28. Acesso em: 29 mar. 2025.

[20] GUIMARÃES, Mário Luís; SILVA, António Rito. Improving early detection of software merge conflicts. In: 2012 34th international conference on software engineering (icse). IEEE, 2012. p. 342-352. 10.1109/ICSE.2012.6227180. Acesso em: 29 mar. 2025.

[21] META. Separation Logic and Bi-Abduction. 2025. Disponível em: https://fbinfer.com/docs/separation-logic-and-bi-abduction. Acesso em: 29 mar. 2025.

[22] CALCAGNO, Cristiano et al. Compositional shape analysis by means of bi-abduction. In: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2009. p. 289-300.

[23] META. Pulse Checker. 2025. Disponível em: https://fbinfer.com/docs/checker-pulse. Acesso em: 29 mar. 2025

[24] ARZT, Steven et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. ACM sigplan notices, v. 49, n. 6, p. 259-269, 2014.

[25] SPGROUP. Static Semantic Merge. 2025. Disponível em: https://github.com/spgroup/static-semantic-merge. Acesso em: 31 mar. 2025

[26] ECLIPSE FOUNDATION. CompilationUnit (JDT API). 2025. Disponível em: https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FCompilationUnit.html. Acesso em: 31 mar. 2025.

[27] ECLIPSE FOUNDATION. ASTRewrite (JDT API). 2025. Disponível em: https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2Frewrite%2FASTRewrite.html. Acesso em: 31 mar. 2025.

[28] ORACLE. ProcessBuilder (Java Platform SE 8). 2025. Disponível em: https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html. Acesso em: 31 mar. 2025