# Symmaries：面向 Java 程序的形式化安全摘要自动推断

NARGES KHAKPOUR，英国纽卡斯尔大学NICOLAS BERTHIER，法国 OcamlPro 和法国利物浦大学

我们提出了一种可扩展、模块化且可靠的方法，可自动为 Java 字节码程序构建形式化安全规范，并以方法摘要的形式呈现。摘要提供了方法安全行为的抽象表示，包括安全调用该方法所需满足的条件，以及对信息流和别名更新的规范。这类摘要可供静态代码分析工具使用，也有助于开发者理解代码片段（例如库）的行为，从而评估其在应用程序中被重用时可能带来的安全影响。我们将此方法实现为名为 Symmaries 的工具，用于自动生成安全摘要。我们将Symmaries应用于 Java API 库，以提取其安全规范；同时也将其用于大型真实世界应用程序，以评估其可扩展性。结果表明，该工具能够扩展至分析拥有数十万行代码的应用程序，并且 Symmaries 能够达到良好的精确率，具体取决于所使用的堆模型。我们证明了该方法的可靠性，即能够保证终止不敏感非干扰。

**ACM 引用格式：**
Narges Khakpour 和 Nicolas Berthier。2025 年。Symmaries：面向 Java 程序的形式化安全摘要的自动推断。1，1（2025年12月），28 页。https://doi.org/10.1145/1122445.1122456

arXiv:2512.20396v1 [cs.CR] 23 Dec 2025

## 1 引言

由于开源软件开发已成为一种广泛采用的成熟实践，漏洞可能会迅速传播，尤其是通过许多开发者直接或间接使用的常见第三方软件包传播。确保开源代码安全的方法侧重于多个方面，包括检测不安全代码 [12, 23, 52–54], 隔离恶意代码 [44, 57], 通过软件成分分析识别被重复使用的代码实例（无论其已过时还是经过修改） [53], 以及开发可信包管理器，以确保所上传软件包的完整性 [24, 48]。这些漏洞识别方法通常依赖于分析代码模式，并通过先进的静态分析技术检测潜在的安全问题，例如将代码与已报告的已知漏洞进行匹配。尽管这些工作已经促成了可扩展的方法，有助于防止使用不安全的开源代码，但仍无法检测出所有漏洞。这一局限源于缺乏提供可靠安全保证所必需的形式化基础。

缺乏形式化基础，因此需要能够正确描述代码行为的方法，使使用者能够在特定情境下严谨地评估并推理安全性。由于安全性是相对于特定要求定义的，在一种环境中安全的代码段，在另一种环境中可能并不安全。例如，图 1 所示的 javax.imageio.stream 类中的 FileImageOutputStream 方法，会将一系列字节写入流中当前位置的

作者联系方式：Narges Khakpour，纽卡斯尔大学，英国，narges.khakpour@ncl.ac.uk；Nicolas Berthier，OcamlPro，法国及利物浦大学，法国，nicolas.berthier@ocamlpro.com。

允许免费制作本作品全部或部分内容的数字或纸质副本，供个人或课堂使用，但不得为营利或商业利益而制作或分发副本，且副本首页须附有本声明及完整引文。本作品中第三方组件的版权必须予以尊重。如需用于其他用途，请联系所有者／作者。© 2024年 版权归所有者／作者所有。ACM XXXX-XXXX/2025/12-ART
https://doi.org/10.1145/1122445.1122456

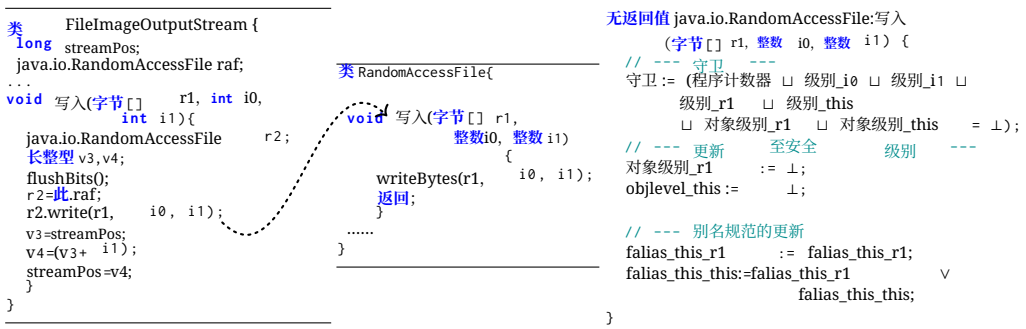



**图1**。 一个 Java 字节码示例（左）以及 `RandomAccessFile.write` 的安全摘要（右）

位置。它在内部调用 java.io.RandomAccessFile 中的 write，后者又通过本地方法 writeBytes 写入数据。该方法的安全性取决于所写数据的敏感程度（r1），以及它访问的流索引（i0 和 i1）；如果在这些信息中任意一项高度敏感的情境下使用，就可能导致信息泄漏。因此，提供关于方法行为的形式化且正确的规范，对于理解其安全影响至关重要。手动构建正确且完整的规范既困难又容易出错 [36]。为此，已有多种方法被提出，用于从代码中自动推断规范 [2, 10, 13, 22, 45, 47, 50]。然而，现有方法至少面临 C1–C4 中的一项挑战，限制了它们在安全分析中的有效性。第一个挑战是，只有少数方法专注于推断与安全相关的规范，主要是污点规范 [45, 47], ，而且这些方法并不可靠（C2）；但可靠性对于生成能够正确刻画代码安全行为的形式化规范至关重要。此外，可靠的形式化方法在应用于现实的大型代码库时，往往面临可扩展性问题（C3）。此外，现有规范推断技术的输出大多是为静态分析工具设计的，通常缺乏简单、清晰且易于使用的条件，因而开发者难以手动评估第三方组件是否安全、能否复用，或评估自身代码的安全性（C4）。

为弥补这一空白，我们提出了一种方法，用于在字节码级别自动构建 Java 方法的形式化规范，该规范涵盖两个主要方面。第一方面涉及方法在信息流控制（IFC）方面与安全相关的行为。 [42], 信息流控制是一种成熟的应用程序安全分析技术，可防止敏感数据遭到未经授权的披露或修改。信息流控制通常通过基于非干扰的性质进行形式化， [14],这些性质确保攻击者的观察不会受到高敏感度数据的影响。我们的规范通过堆跟踪信息流，并同时考虑显式流和隐式流。第二方面涉及方法的别名规范，用于描述方法调用如何修改堆，其形式是引用之间的关系，例如跟踪哪些对象可能指向同一内存位置（可能别名关系），以及执行方法后这些关系如何发生变化。通过捕获堆及其别名结构的更新，我们的方法能够正确理解信息如何通过堆流动，并有助于理解和推理副作用、潜在的信息泄露以及方法的总体安全影响。

我们开发了 Symmaries，这是一款可扩展的工具，采用形式化的基于摘要的方法，为 Java 方法构建可靠的安全摘要。我们将程序语义建模为符号控制流图，为变量和引用赋予安全级别，并将堆表示为引用之间的一组别名关系。该语义捕捉了这些安全级别如何

以及堆别名关系如何随方法行为而演变。方法摘要由两个主要部分组成：（i）可靠的守卫，以简洁的逻辑表达式指定安全调用该方法所需满足的条件；（ii）更新，描述方法调用对程序状态产生的效应，以一组赋值表示。守卫和更新基于堆中的别名关系，以及方法参数和 this 的安全级别定义。图 1 展示了为 write 构建的安全摘要示例，其中变量 falias_r1_r2 表示两个典型引用 r1 和 r2 之间的字段别名关系（即 r1 可能通过一系列字段访问与 r2 形成别名关系），而 objectLevel_r 和 level_r 表示引用 r 的安全级别（更多详情请参见第 2 节）。该摘要的守卫指出，在任何其参数或*this* 均不包含高敏感信息的上下文中，都可以安全地调用此方法。我们认为，我们的安全摘要能够对方法的安全行为提供有意义、可靠且易于理解的抽象。静态分析工具或代码使用者可以利用这些摘要，在无需检查实现细节的情况下推断代码的安全性。

贡献总结。我们提出了一种可扩展、精确且可靠的方法，用于构造 Java 字节码程序的安全摘要。我们的贡献如下：

• 我们提出了一种构造 Java（字节码）程序安全摘要的新方法，可对其安全行为进行直观易懂的抽象。我们的方法以模块化方式分析程序以提取摘要，通过堆捕获信息流，并同时考虑显式流和隐式流。

• 我们实现了 Symmaries，这是支持该方法的一套完整工具链，并通过大量实验评估其可扩展性。实验表明，我们的基于摘要的方法能够扩展至大型真实世界开源应用程序。该工具输出一组方法契约，我们对其进行了设计，使用户能够轻松理解。

• 得益于其符号化设计，Symmaries 无需手动注释（例如，入口点识别），因此在大型应用程序中实用且轻量；我们的评估也证实了这一点。

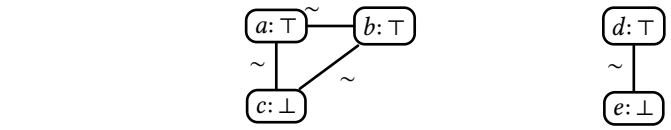
• 推断出的安全摘要的精确率取决于多个敏感性维度，包括上下文、对象、流和字段。我们的方法具有流敏感、上下文敏感和对象敏感的特性，而字段敏感性则取决于所选的堆模型。Symmaries提供多种堆模型，在这些模型中，堆以不同的别名关系表示 [7]。因此，结果的精确率取决于所选的堆模型。我们将Symmaries 应用于部分 Java JDK，以评估精确率。结果表明，该工具能够为方法推断出精确的规范。

• 我们从形式上证明了我们的方法在终止不敏感的非干扰性方面是可靠的。

## 2 基本概念

安全域。我们采用基于类型的方法来形式化安全性，将安全类型赋予程序变量和对象。我们考虑一个经典的两级安全域，它定义为有界格 $\mathbb{L}, \sqsupseteq$，其中 $\mathbb{L} \supseteq \{\bot, \top\}$ 是安全级别的有限集合，$\sqsubseteq$ 是定义在 $\mathbb{L}$ 上的偏序，$\bot$ 和 $\top$ 分别表示最低（低）和最高（高）安全级别。我们通常用变量 $x$ 表示与程序变量 $x$ 相关联的安全级别（在图1中记作 level_x）。接下来，我们着重于执行机密性策略：高敏感（$\top$）信息不应影响低敏感（$\bot$）输出。

输入程序。我们将输入程序视为一组用于操作基本数据和对象引用的方法。为使我们的方法尽可能易于使用，我们专注于




$$\mathrm{h} = (\vec{a} = \top) \wedge (\vec{b} = \top) \wedge (\vec{c} = \bot) \wedge (\vec{d} = \top) \wedge (\vec{e} = \bot) \wedge \tilde{p}_{a,b} \wedge \tilde{p}_{a,c} \wedge \tilde{p}_{b,c} \wedge \tilde{p}_{d,e}$$

**图 2.** 堆模型及其形式化表示 h

一种低级语言，其中大多数语句都对应一类Java 字节码，方式类似于Jimple [49]。方法方法体 $m$ 是由分号分隔的非空有限语句序列，并根据 $\mathbb{S}$以下文法构建：

$$\begin{aligned}\mathbb{S} &::= [l:]\ a; \mathbb{S} \mid \surd \\ a &::= \mathbb{S}' \mid [v =] r.m'(w) \mid \mathtt{output}_l(y) \mid \mathtt{return}\ [v \mid r] \\ \mathbb{S}' &::= v = e \mid v = r.f \mid r.f = e \mid r = r \mid r = \mathsf{new}\ c \mid \\ &\quad r = \mathtt{null} \mid [\mathtt{if}\ (e)]\ \mathtt{goto}\ l\end{aligned}$$

其中，方括号表示可选结构，$\surd$ 表示空语句序列，而$l$是唯一标识某条语句的标签。 $r.m'(w)$ 表示对对象的方法 $m'$的调用，该对象由 $r$引用，并带有映射 $w$，为 $m'$的每个形式参数关联一个值 。我们的语言还包含语句 output$_l(y)$，用于将 $y$ 输出到公共通道。 $\mathbb{S}'$ 展示了过程内语句的语法，例如赋值、分支和构造函数，其中 $e$是表达式， $v$ 是变量， $f$是字段。

堆模型。 我们采用 [7],中介绍的方法，通过堆抽象 h 来捕获流入和流出堆对象的信息流。该方法维护

- 一个**堆类型环境**，用于为可通过任意引用 $r \in R$ 访问的堆部分关联一个安全级别上界，其中 $R$ 是一组引用。由 $r$引用的对象的安全级别记为 $\vec{r}$。
- （**过度近似** 的）**堆相关关系**，这些关系存在于 $R$ 中的引用变量之间，和/或存在于从这些引用变量可达的对象之间。堆关系的一个例子是**可能别名** 关系：如果两个引用变量都指向同一个对象，则它们之间存在这种关系。

我们将堆形式化为一个谓词，该谓词定义在一组变量上，这些变量表示堆类型环境和堆相关关系。关系在形式上被指定为一个定义在一组布尔变量上的谓词，每个布尔变量都表示两个引用是否存在该关系（即，我们使用谓词抽象，其中一个布尔变量表示两个引用之间是否存在某种关系）。图2展示了一个堆及其形式化表示的示例。节点表示引用变量及其安全类型，两个引用之间的弧表示它们存在**可能共享**堆相关关系：对于给定的 $\{r, s\} \subseteq R$, $r$和 $s$，当且仅当可能存在一个可通过 $r$和 $s$（传递地）到达的对象时，它们才可能共享。在这一形式化表示中，变量 $\vec{r}$（即对应于图1中的变量objlevel_r）包含可通过 $r \in R$到达的任何对象的安全级别上界；当 $r$和 $s$存在可能共享关系时，命题 $[\frown]rs$成立。该公式表明，引用 $a$、 $b$和 $d$包含高敏感信息（例如， $\vec{a} = \top$），而 $e$和 $c$则是低敏感的（例如， $\vec{e} = \bot$）。此外，与这些边相关联的引用可能彼此存在别名关系；例如， $a$和 $c$可能彼此存在别名关系（例如，命题变量 $a \frown c$成立）。请注意，可以使用不同类型的关系来表示堆。例如，

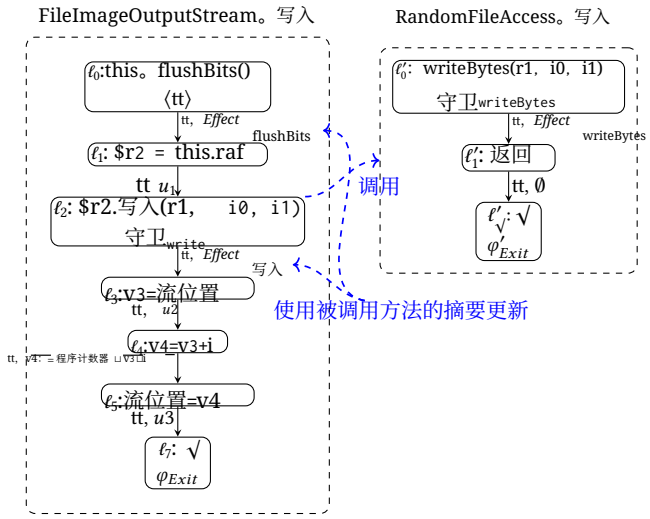



**图3.** 我们基于摘要的方法

$_{*}$，我们使用 $\overset{\hookrightarrow}{p}_{r,\ s}$ 表示字段别名关系（即图1中的变量falias_x_y）。有关可能关系的一些示例，请参见 [7]。

符号控制流图（SCFG）。我们用于描述程序语义的转换系统，是传统的带标签转换系统，并分别通过状态变量集和输入变量集进行扩展，记为 $X$ 和 $I$。输入变量的值可视为来自系统环境。图3展示了一个 SCFG，用于描述 FileImageOutputStream.write 方法的语义。赋值中的谓词和右侧表达式由传统逻辑连接符构成（即 $\neg$、 $\vee$、 $\wedge$ 和 $\twoheadrightarrow$，并结合三元条件结构“if · then · else ·”，其含义显而易见。我们使用的符号变量通常取自安全域 $\mathbb{L}$，或布尔值集合 $\mathbb{B}$ def= {ff, tt}。我们使用合并操作 u $\sqcup$ 来合并变量赋值。此操作通过取并集实现；当多个表达式被赋给同一个变量 $v$ 时，使用某种连接符 $\sqcup_v$ 将它们合并。连接符取决于各变量的语义：由于在编码中变量仅用于保存过度近似，我们对布尔值使用析取 $\vee$，对安全级别使用最小上界 $\sqcup$。例如，$\{a{:=}\mathrm{tt}\ b{:=}\mathrm{ff}\}$ u $\sqcup\{b{:=}\mathrm{tt}\} = \{a{:=}\mathrm{tt}\ b{:=}\mathrm{ff}\ \vee \mathrm{tt}\}$。 $S$的语义是一个有限状态自动机 $[\![S]\!] = \mathrm{Q}, \mathrm{I}, \rightarrow, \mathrm{Q}_0$，其状态空间 Q 是位置集合 $\Lambda$ 与状态变量所有可能赋值集合 Val $(X)$ 的笛卡尔积。$\mathrm{Q}_0 \subseteq \mathrm{Q}$ 是满足 $X_0$的状态变量初始赋值集合。每当 $[\![S]\!]$ 接收到所有输入变量 I 的一个赋值（即 Val $(I)$ 中的一个元素）时，它就从 $\rightarrow$ 执行一次转换。在任何位置，都恰有一个转换的守卫满足所有变量的赋值。执行该转换时，其赋值会应用于更新状态变量。SCFG $S$ 的不变量$\varphi$ 是一个从位置映射到状态变量谓词的函数。$S$ 满足 $\varphi$，当且仅当每个位置为 $\ell$ 且可由 S 到达的状态 $q$ 都满足 $q \models \varphi(\ell)$。

**3 方法概述**

我们设计了一种可靠且可扩展的方法，以模块化方式构建方法的信息流摘要，其中每个方法都单独分析。我们为变量赋予安全级别

以及方法操作的堆对象指定安全级别，并使用符号控制流图来描述程序语义。程序语义的状态变量包括：(i) 安全类型环境（即变量的安全级别，以及用程序计数器（pc，意为“程序上下文”）表示的上下文级别）；以及 (ii) 用于捕获流向堆对象和来自堆对象的信息流的堆模型。我们遵循 [7] 中介绍的方法来描述非方法调用语句的语义。为定义方法调用的语义，我们采用基于摘要的方法，并重用被调用方法的摘要，而不是将其方法体的语义内联到调用方的语义中。方法的摘要是一种契约，由以下内容构成

• 一个守卫，用于指定方法调用安全的调用条件，即方法中不存在非法信息流。守卫描述为对方法参数的安全级别和堆结构的约束。

• 一种效应，用于表示其对安全级别以及堆对象之间别名关系的副作用；原则上，它是一组赋值，用于描述该方法如何更新别名关系和安全标签（示例见图1）。

图3展示了一个 SCFG，它描述了图1所示程序的简化语义，其中位置对应于语句，而转换会同时更新安全级别和别名关系。每个位置标签表示一条语句及其可能具有的不变量（如果有），该不变量展示了该位置必须满足的规范。每个转换都带有一个守卫和一个更新函数。为了定义方法的语义，我们用被调用方法的摘要更新来代替其调用。例如，从 $\ell_0$ FileImageOutputStream.write 出发的转换会根据 flushBits 的效应更新状态变量，即效应$_{\texttt{flushBits}}$。我们还将被调用方法的摘要守卫用作调用位置的安全属性（称为不变量），以防止任何非法信息流，例如位置 $\ell_2$ 处的 $\text{Guard}_{\texttt{write}}$，或位置 $\ell_0$处的 true。此外，u1 – u3 是一组根据字段加载和存储语句的语义定义的更新。

为构建方法摘要，我们通过共可达性分析，找出所有能够避免到达特定状态的路径条件。为构建摘要守卫，我们找出所有不会导致不安全状态的路径条件；也就是说，如果某个状态是泄露高安全级别敏感信息的汇点语句，或者是直接或间接到达不安全状态的方法调用，那么该状态就是不安全的。例如，图 3 中的 write 方法：(i) 不应在与位置 $\ell_7$相关联的汇点语句中泄露任何敏感信息；(ii) 对方法 flushBits 和 RandomFileAccess. write 的调用不应导致到达不安全状态。为确保满足情况 (i)，我们要求不变量 $\varphi_{Exit}$ 在位置 $\ell_4$成立；通俗地说，这表示参数、*this*以及用于建模堆的变量的安全级别保持不变。通过这种编码，我们能够识别违反该不变量的条件，并据此提取更新。为确保满足情况 (ii)，我们要求调用 flushBits 和 RandomFileAccess.write 时，其摘要守卫成立（分别在 $\ell_0$ 中定义为真不变量 tt，在 $\ell_2$中定义为 $\text{Guard}_{\texttt{write}}$）；也就是说，通过调用这些方法不会到达不安全状态。

为了构建摘要效应，我们找出导致堆或参数安全级别发生更新的所有路径条件。在我们的示例中，我们存储初始堆 $\mathfrak{h}$ 于 $\mathfrak{h}^r$ 方法入口处，并且在执行过程中从不修改 $\mathfrak{h}^r$。然后，我们使用 $\varphi_{Exit}$ 中的不变量 $\ell_7$，其中 $\mathfrak{h}$ 是方法退出点处经方法更新后的堆，用它来表示退出时引用之间安全类型或堆关系的符号上界。接着，我们的共可达性分析会识别所有 $\phi$ 会更新堆的路径条件，即违反 $\varphi_{Exit}$中符号界限的条件；同时也会识别导致不安全状态的路径条件，即违反其他不变量的条件。我们使用一种三角化 技术 [20] 将 $\phi$ 转换为一个守卫和一组赋值，构成该方法的摘要。我们证明了使用我们的方法构造的摘要是可靠的（见第 5.3 节），因此可以用这些摘要替代方法调用。摘要的表达能力取决于

所选择的堆相关关系（参见 [7]，其中给出了采用不同堆相关关系的堆抽象示例）。下文对这组关系基本不作具体规定，以便在阐述中略去不必要的细节；在不失一般性的前提下，我们将重点讨论二元关系。

对于过程间分析，我们采用模块化基于摘要的方法，分别分析每个方法以构造其摘要，从而显著提升分析的可扩展性。如果待分析的程序不包含（潜在的）递归方法（即其调用图 $G$ 是无环的），则可以通过对 $G$ 进行简单的拓扑遍历来计算所有摘要。然而，如果 $G$ 是循环的，即包含递归方法，这种方案便不再适用。为解决这一问题，我们以一种能够采用混沌迭代策略 [8]的方式定义摘要，从而找到一个不动点，得到程序的所有摘要集合。

## 4 摘要构造

本节首先对摘要的概念进行形式化定义，然后计算一个不包含任何方法调用的方法的摘要。

### 4.1 摘要的定义

考虑一个方法 $m$，其方法体为 $\mathbb{S}_m$，参数为$Args_m$；对于非静态方法，还包括引用变量`this`。令 $Refs_m$ 表示 $m$中使用的所有引用变量的集合，其中包括引用参数。我们假设，$m$ 的引用参数绝不会出现在 $\mathbb{S}_m$中的任何赋值语句左侧——这一初步要求可通过简单的程序转换轻松满足。我们将方法 $m$ 的支持集定义为由以下变量组成的集合：参数的安全类型、程序上下文变量程序计数器，以及变量集合 $\mathbb{V}_{\mathfrak{h}}$，该集合用于编码方法调用时堆中可达的部分，即：

$$\mathrm{Supp}_m \stackrel{\text{def}}{=} \bigcup_{x \in Args_m} \{\bar{x}\} \cup \{pc\} \cup \mathbb{V}_{\mathfrak{h}} m.$$

该足迹的 $m$ 定义方式类似，其中包括 $\pi$ 返回值的安全级别 $\pi$（如果有）¯，以及用于编码 $\mathbb{V}_{\mathfrak{h}^r}$ 调用终止时堆的可达部分的变量集合，即 $\mathrm{Foot}_m \stackrel{\text{def}}{=} \pi \cup \mathbb{V}_{\mathfrak{h}^r}$.

摘要守卫$Guard_m$ 仅涉及支持集 $\mathrm{Supp}_m$ 中的变量；若 $m$ 的任何执行都不会导致其语义中的任何不变量被违反，则该条件成立。相应地，Effect$Effect_m$ 包含足迹 $\mathrm{Foot}_m$中每个变量对应的一条赋值语句，即为安全类型环境的每个分量赋予一个表达式，该表达式给出上界，用于表示终止时 $m$对应的安全级别；对于任何过度近似的堆相关关系，处理方式也类似。

### 4.2 计算摘要

我们在算法 1 中概述了计算摘要的整体过程，并将在本节中详细阐述。

*4.2.1* 语义规则与不变量。为了计算方法 $m$的摘要，我们算法中的 Semantics（$m$）会构造一个 SCFG $S_m$ 来指定其语义，并为每个位置构建一个不变量映射 $\varphi_m$。图4介绍了过程间语句的语义和不变量。

为确保机密性，我们为每条会触发 Sink 规则应用的汇点语句关联一个客观不变式 $\varphi_{\text{SINK}}$。该属性通过确保输出到公共通道的任何标量数据或被引用对象的安全级别不为高敏感，避免机密性遭到破坏。此外，该属性还通过断言 $pc \sqsubseteq \bot$，确保输出只能在低敏感上下文中执行，从而避免通过上下文造成泄漏。例如，$\varphi_{\ell_4}$ 在图3中

$$\textsc{Sink}\ \frac{}{(\langle\!\langle \mathsf{output}_l(x)\rangle\!\rangle;\mathbb{S}) \xrightarrow{\mathsf{tt},\varnothing} (\mathbb{S},\mathsf{njb})} \qquad \textsc{Exit}\ \frac{}{(\surd) \xrightarrow{\mathsf{tt},\varnothing} \ell}$$

$$\textsc{Return}\ \frac{T = [\mathit{ret\text{-}var} \coloneqq \mathrm{Ret}_x] \ \text{ if } a = \langle\!\langle \mathtt{return}\ x\rangle\!\rangle, \varnothing \text{ otherwise}}{(\langle\!\langle \mathtt{return}\ [_]\rangle\!\rangle;\mathbb{S}) \xrightarrow{\mathsf{tt},T} (\surd,\mathsf{njb})}$$

$$\varphi_{\textsc{Sink}} = pc \sqsubseteq \bot \wedge \overline{x} \sqsubseteq \bot \wedge (\overline{\vec{x}} \sqsubseteq \bot \text{ if } x \in \mathit{Refs}_m, \mathsf{tt} \text{ otherwise}) \qquad (\varphi\text{-}\textsc{Sink})$$

$$\varphi_{\textsc{Exit}} \overset{\text{def}}{=} \varphi_x \wedge \varphi_{\mathfrak{h}} \qquad (\varphi\text{-}\textsc{Exit})$$

$$\varphi_x \overset{\text{def}}{=} \bigwedge_{\langle\!\langle \mathtt{return}\ x\rangle\!\rangle \in \mathbb{S}_m} (\mathit{ret\text{-}var} = \mathrm{Ret}_x \Rightarrow pc \sqcup \overline{x} \sqsubseteq \overline{\pi}^r) \qquad (\varphi\text{-}\textsc{RetVal})$$

$$\varphi_{\mathfrak{h}} \overset{\text{def}}{=} \bigwedge_{\langle\!\langle \mathtt{return}\ r\rangle\!\rangle \in \mathbb{S}_m} (\mathit{ret\text{-}var} = \mathrm{Ret}_r \Rightarrow \mathfrak{h}[\pi \leftarrow r] \sqsubseteq_{\mathsf{H}} \mathfrak{h}^r) \qquad (\varphi\text{-}\textsc{Heap})$$
$$\wedge\ (\mathfrak{h} \sqsubseteq_{\mathsf{H}} \mathfrak{h}^r \text{ if } \nexists\langle\!\langle \mathtt{return}\ r\rangle\!\rangle \in \mathbb{S}_m, \mathsf{tt} \text{ otherwise})$$

**图4.** 推断摘要的规则。

**算法 1：** InferSummary

**输入：** 要分析的方法 $m$
**结果：** 多态 IF 摘要 $Sum_m = \mathit{Effect}m$，守卫$m$
{ 将 $m$ 的语义编码为 *SCFG* $S_m$，其中所有与调用上下文相关的状态变量均未初始化，并且将安全要求表示为谓词 $\varphi_m(\ell)$，作用于 $S_m$ 的每个位置 $\ell$ 上的状态变量： }
1 $(S_m, \varphi_m) \leftarrow$ 语义（$m$）
{ 定义所有已知的不安全状态，包括每个违反安全属性的状态（$\Lambda_m$ 是 $S_m$ 的位置）： }
2 $\mathrm{B}_0 \leftarrow \{\ell \mapsto \neg\varphi_m(\ell) \mid \ell \in \Lambda_m\}$
{ 计算所有可到达 $\mathcal{B}_0$ 的状态： }
3 $\mathrm{B}_\infty \leftarrow$ 可达前驱 $(S_m, \mathrm{B}_0)$
{ 将不属于调用上下文的状态变量提取出来（因此 $G$ 仅涉及来自 $\mathrm{Supp}_m \cup$ 足迹$m$）： }
4 $G \leftarrow$ 余因子 $(\neg\mathrm{B}_\infty(\ell_0(S_m))\ X_0(S_m))$
5 $\langle \mathit{Effect}m, \text{守卫}m\rangle \leftarrow$ 三角化 $(G, \text{足迹}m, \varnothing)$

展示了为输出 s 的汇点位置 $\ell_4$ 定义的不变式。该属性要求 s 的安全级别和上下文均为低。

Return 规则定义了返回语句的语义，其基本作用是将所返回值（若有）的变量记录在 *ret-var*中，然后跳转到方法退出点；也就是说，*ret-var* 的取值数量可以与该方法中不同返回语句的数量相同。我们将位置 $\ell_\surd$ 关联到退出点。为了构造摘要效应，我们为该位置定义一个不变量 $\varphi_{\textsc{exit}}$，以确保足迹 $\mathrm{Foot}_m$中的每个变量都有一个下界。在图4中，谓词 $\varphi_x$ 表示：如果该方法返回值或引用 $x$，即 $\mathit{ret\text{-}var} = \mathrm{Ret}_x$，那么它与$pc$ 对应的上界应被返回值的初始级别所涵盖，记为 $\pi^r$。共可达性算法沿每条可能路径将此谓词向后传播到初始状态时，上界 $\pi^r$ 将计入对返回值的更新。 —

同样，堆抽象 $\mathfrak{h}^r$ 是方法终止时实际堆抽象的一个上近似$\mathfrak{h}$，我们定义不变量 $\varphi_{\mathfrak{h}}$ 以获得方法对堆模型的更新。根据方法是否返回一个引用，$\mathfrak{h}[\pi \leftarrow r]$表示堆抽象 $\mathfrak{h}$ 扩展后加入一个引用 $\pi$，该引用是 $r$的别名。此外，$\mathfrak{h} \sqsubseteq_{\mathsf{H}} \mathfrak{h}^r$ 表示一个偏序：当抽象值 $\mathfrak{h}$限制在属于 $R' \subseteq R$ 的引用集合上时，$\mathfrak{h}^r$对一组堆配置构成上近似；该组堆配置大于

或等于由 $\mathfrak{h}^r$表示的堆配置集合。这相当于对安全级别进行两两比较，即必须满足 $\mathfrak{h} \sqsubseteq_{\mathsf{H}} \mathfrak{h}^r \Rightarrow (\forall r \in R',\ \overrightarrow{r} \sqsubseteq \overrightarrow{r})$ (对任何关于堆的上近似事实也同样如此，例如可能别名关系)。

例如，图3中为退出位置 $\ell_{\surd}$ 定义的不变量$\varphi_{\ell_{\surd}}$ 要求 e 的级别被返回值 $\pi$的初始级别所蕴含。请注意，此方法只有一个返回语句，并且 *ret-var* $= \mathrm{Ret}_e$ 始终成立。此外，引用 *this* 和 r 不应在返回时被升级；如果 *this* 和 r 最初彼此为别名（即$\mathfrak{h} \sqsubseteq_{\mathsf{H}} \mathfrak{h}^r$），则它们仍保持别名关系。

*4.2.2* 通过共可达性构造摘要。我们的方法采用共可达性分析，以计算方法 $m$ 不会到达非期望状态的初始状态。我们将状态空间表示为从位置到状态变量上的谓词的映射。我们用 $\mathrm{B}_0$ 表示所有初始已知的非期望状态的集合；这些状态违反语义中定义的不变量（例如图4中所示的 $\varphi_{\textsc{Sink}}$ 和$\varphi_{\textsc{Exit}}$），因而应予以避免。随后，通过标准的共可达性分析（由 Coreach 实现）对非期望状态集合进行反向传播，从而找出所有能够到达状态集合 $\mathrm{B}_0$ 的状态。通常使用不动点 [33, 35]求解这一问题。对于类似 $S_m$的符号有限状态系统，这一计算总能终止，并且传统上使用最小不动点（lfp）执行。

$$\mathcal{B}_\infty \stackrel{\text{def}}{=} \mathrm{lfp}\ \lambda \mathcal{B}_i.\mathcal{B}_0 \cup \mathrm{pre}(\mathcal{B}_i), \tag{1}$$

其中，pre(B) 给出 B 的所有前驱状态。映射 $\mathrm{B}_\infty$ 将每个位置$\ell$ 与一个谓词关联起来：对于从 $\ell$ in $S_m$ 出发的所有后续路径，该谓词都必须不 成立，才能表示期望执行。谓词 $\mathrm{B}_\infty(\ell_0)$ 描述的是能够到达非期望状态 $\mathrm{B}_0$ 的初始状态集合。因此，它的补集 $\neg\mathrm{B}_\infty(\ell_0)$ 就是所有能够导向期望执行的路径条件集合。

要获得该方法的守卫和 Effect，我们需要从 $\neg\mathrm{B}_\infty(\ell_0)$ 中消除所有与其调用上下文无关的符号变量。换言之，我们要消除安全类型环境和堆抽象中与局部 基本类型变量或引用变量相关的部分。在 $S_m$的初始状态（记作 $X_0$）中，局部 基本类型变量和引用变量的安全级别设为低敏感，且局部引用在 null 堆抽象中初始化为 $\mathfrak{h}$。相反， $X_0$ 中每个形式参数和$pc$ 的安全级别均保持不受约束 。因此，要执行上述消除操作，我们可以根据初始化谓词 $X_0$，对 $\mathrm{B}_\infty(\ell_0)$ 的补集进行部分求值。具体来说，使用 cofactor($\neg\mathrm{B}_\infty(\ell_0)$ $X_0$) 完成此操作：它会查找所有满足 $X_0 \Rightarrow (x = c)$ 的等式 $x = c$，其中 $x$ 是变量， $c$ 是常量，然后在 $\neg\mathrm{B}_\infty(\ell_0)$ 中以 $c$ 替换 $x$。由此得到的谓词 $G$ 由方法支持集和方法足迹中的变量构成，即 $\mathrm{Supp}_m \cup \mathrm{Foot}_m$。随后，我们使用三角化 技术 [20] 将 $G$ 转换为 $m$的摘要。

*4.2.3* 三角化。 算法 1 中的余因子分解会得到一个谓词 $G$，该谓词由方法支持集和方法足迹中的变量表示，即 $\mathrm{Supp}_m \cup \mathrm{Foot}_m$。随后，我们使用一种三角化 技术 [20]，将 $G$ 转换为 $m$的摘要。通过使用该算法，我们逐步计算赋值集合$Effect_m$，具体过程依次为：(i) 找出一个赋值 $v := e$，使得 $(v = e) \Rightarrow G$，其中 $v$是一个足迹变量；(ii) 在 $G$中用 $e$替换 $v$。对足迹中的每个变量完成上述操作后， $G$ 便只涉及支持集 $\mathrm{Supp}_m$中的变量，并构成该方法的守卫$Guard_m$。函数 Triang ($G, V, E$),

$$\textsc{Call}\ \frac{\begin{array}{c}(a = \langle\!\langle [_ =] r.m(w) \rangle\!\rangle) \qquad T'_{\mathfrak{h}} = \mathsf{Upgrade}(\mathit{Effect}_m, \lfloor \top \rfloor_\Upsilon, \mathfrak{h}) \\ T_{\mathfrak{h}} = T'_{\mathfrak{h}}[\pi \to r'] \text{ if } a = \langle\!\langle r' = r.m(w) \rangle\!\rangle, T'_{\mathfrak{h}} \text{ otherwise} \\ T_\pi = [\overline{x} :=_\Upsilon \mathit{Effect}_m(\overline{\pi})] \text{ if } a = \langle\!\langle x = r.m(w) \rangle\!\rangle, \varnothing \text{ otherwise}\end{array}}{\ell = a; \mathbb{S} \xrightarrow{\mathsf{tt}, T_\pi \between T_{\mathfrak{h}}} (\mathbb{S}, \mathsf{njb})}$$

where

$$\lfloor l \rfloor_\Upsilon \stackrel{\text{def}}{=} (\text{if } \Upsilon \text{ then } \bot \text{ else } l) \sqcup pc \tag{2}$$

$$\overline{x} :=_\Upsilon l \stackrel{\text{def}}{=} \overline{x} := (\text{if } \Upsilon \text{ then } \overline{x} \text{ else } l) \sqcup pc \tag{3}$$

$$\varphi_{\textsc{Call}} \stackrel{\text{def}}{=} \mathit{Guard}'_m \tag{$\varphi$-Call}$$

**图 5.** 方法调用的翻译规则和客观不变式。

该函数根据 $G$ 构造一个摘要（即守卫和效应组成的对），定义如下 s:

$$\text{三角化}\,(G, V, E) \stackrel{\text{定义}}{=} \begin{cases} \textsc{Triang}\,(G\,[v \mapsto e]\,, V \setminus \{v\}\,, E \cup [v := e]) & v \in V, \\ \langle E, G \rangle & \text{if } V = \varnothing \end{cases}$$

其中 $e = \mathsf{Minimize}\,(G, v)$ 是一个符号表达式 $e$，它仅涉及 $G$ 中的变量，并在满足 $G$ 的条件下最小化 $v$；也就是说，显然有 $(v = e) \Rightarrow G$，且不存在一个与 $e$ 不等价的表达式 $e'$，使得：(i) $(v = e') \Rightarrow G$，并且 (ii) $e' \sqsubset e$ 可满足（若 $v$ 是层级变量）。因此，在三角化中寻找“最小”表达式是获得摘要中精确 效应的关键因素。例如，将 $\top$ 赋给足迹的层级变量的更新，尽管是安全的上近似，却不能构成对方法效应足够精确的描述。

然而请注意，从算法 1 中获得的单个谓词 $G$ 可能推导出多个摘要。事实上，最终效应会受到在 Triang 中选择足迹中各变量的顺序影响。此外，足迹变量的域可能存在多个最小元素，而在 Minimize 中选择有效符号表达式 $e$ 也可能取决于某种任意选择。在我们的实现中，我们为全部变量及其各自的域定义了任意但确定且无歧义的优先级。特别是，对于每个变量 $v \in \mathsf{Foot}_m$，三角化都以一个全序 $(\prec_v)$ 序列作为参数。

## 5 使用摘要进行过程间安全分析

为了实现完整的过程间分析，我们扩展了语义，用摘要替代方法调用，并详细介绍了模块化计算摘要的技术。我们将在后续章节说明如何处理可能出现的递归。

### 5.1 调用图构建

我们首先计算整个应用程序调用图的过度近似。为此，我们需要确定在任何调用点都可能被调用的方法 $x = r.m(w)$。

令二元组 $\mathbb{T}$，$\prec$ 表示整个应用程序的类型层次结构，其中 $\mathbb{T}$ 是类型集合，即类、接口和基本类型的集合；$\prec$ 是 $\mathbb{T}$ 上的偏序，用于表示对象类型之间的继承关系。对于类型 $t$，$t' \in \mathbb{T}$，若满足以下条件之一，则关系 $t \prec t'$ 成立：(i) $t$ 和 $t'$ 是类，且 $t$ 通过扩展 $t'$ 继承自它；或 (ii) 类 $t$ 实现接口 $t'$。我们进一步用 $V = \mathsf{Typ}_{\mathsf{recv}}$，名称，$\mathsf{Typ}_{a,\,1}$，…，$\mathsf{Typ}_{a,\,n}$ 表示方法签名

其中 $\mathrm{Typ}_{\mathrm{recv}}$ 是接收者引用的类型，名称 是方法名，而 $\mathrm{Typ}_{a,\,i}$ 表示第 $i$个参数的类型。

签名为 $\mathrm{Typ}_{\mathrm{recv}}$，$m$，$\mathrm{Typ}_{a,\,1}$，……，$\mathrm{Typ}_{a,\,n}$ 的方法 $m$ 是方法调用$x = r.m(w)$ 的潜在目标，当且仅当满足以下条件：

- $\mathrm{Typ}(r) \prec \mathrm{Typrecv}$,
- $|w| = n$，且
- 类型$(w_i) \prec$ 类型$_{a,\,i}$ 对于 $0 \le i \le n$

对于静态调用（即 $r$ 是类名），$\mathrm{Typ}(r) = r$。我们分析整个应用程序，并构建一个过近似调用图，其边表示方法调用的潜在目标方法。

### 5.2 重用方法摘要

我们首先计算整个应用程序调用图的过度近似。

我们将为某个调用点已计算出的可重用摘要集合结合起来，以编码方法调用。根据调用图，我们可以获得在任意调用点可能成为目标的所有方法集合（的过度近似） $r.m(w)$。因此，我们可以获得相应的摘要。

令 $Sums_m$ 为方法调用 $r.m(w)$ 中所有可能被分派的方法的摘要集合。我们首先取这个摘要集合，并将形式参数替换为对应的实际参数 $r$和 $w$；这样会得到一组支持集和足迹完全相同的摘要。然后，为了得到一个可用于替代 $r.m(w)$ 的单一摘要，我们将这些摘要的守卫条件和效应结合起来。组合后的守卫条件应避免任何可能的目标方法所对应的不安全状态，而组合后的效应应包含所有可能目标方法的效应。因此，我们将组合后的守卫条件定义为所有可能目标方法的摘要守卫的合取，并将组合后的效应定义为它们效应的过度近似并集。因此，用于替代 $r.m(w)$ 的组合摘要定义如下

$$\langle Guard_m, Effect_m \rangle \stackrel{\text{def}}{=} \langle \bigwedge_{\langle Guard, _ \rangle \in Sums_m} Guard,\ \bigsqcup^{u}_{\langle _, Effect \rangle \in Sums_m} Effect \rangle.$$

为解释虚方法分派根据目标对象的实际类所引发的隐式分支行为，我们进一步通过在守卫和效应中进行如下替换来升级上下文级别 $pc$：

$$Guard'_m \stackrel{\text{def}}{=} \begin{cases} Guard_m[pc \mapsto pc \sqcup \vec{r}] & \text{if } |Sums_m| > 1 \\ Guard_m & \text{otherwise} \end{cases}$$

其中，$\vec{r}$是接收者对象（由 $r$指向）的安全级别，$e\,[v \mapsto l]$ 表示在 $e$中用安全级别表达式 $l$ 替换变量 $v$。在效应中进行的替换会作用于赋值右侧的每个表达式。

为捕获隐式流，我们考虑由状态变量 $\Upsilon$ 确定的两种执行模式：(i) 在名义模式（$\Upsilon = \mathrm{ff}$）下，对安全级别的更新反映显式信息流；(ii) 在升级分析模式（$\Upsilon = \mathrm{tt}$）下，捕获从高安全级执行上下文 pc 流向所有相关分支的信息流。图 5 中给出的 Call 规则编码了语义位置 $\ell$ 中方法调用的语义，其中不变量 $\varphi$-Call 确保方法 $m$不存在非法信息流，$T_\pi$将任何被赋值的返回值或引用 $x$的安全类型更新为 Effect $\pi$ 中的返回值安全类型，并且$m$将方法效应应用于调用方的堆模型。我们应用函数 Upgrade(Effect $T_{\mathfrak{h}}$,$m$, $\lfloor \top \rfloor_\Upsilon$) 将隐式流纳入堆中，其中 Upgrade(eff $\mathfrak{h}$, $l$) 构造一个新的更新函数，将更新函数 eff 应用于堆模型 $\mathfrak{h}$，使得 eff 进行的每次安全级别升级都以上界安全级别表达式 $\mathfrak{h}$为限。这一构造的正确性源于以下事实：效应编码的、针对可能发生变异的对象安全级别的更新始终小于或

等于上下文级别 $pc$——除非守卫在高安全级上下文中不成立，在这种情况下，方法调用会因强制不变量而变得不可达。换言之，$pc$ 是由 $m$的效应分配给可能发生变异的堆的任意部分所关联安全级别的上界。因此，由于在升级分析模式下 $\lfloor\top\rfloor_\Upsilon = pc$——参见公式 (2)——，堆类型环境中对应于可能发生变异的堆部分的每个组件，都会根据上下文级别进行弱升级。然而，在名义模式下，由于 $\lfloor\top\rfloor_\Upsilon = \top$，对类型环境的赋值会直接执行。

请注意，构成所应用 Effect 足迹的堆抽象域，对应于调用方方法中所有引用变量的一个子集 $m'$。因此，封顶应用操作需要采取保守策略，将每次级别升级以及堆相关关系的变化传播到可能指向堆中某些区域的引用变量；这些区域可通过 $w$中的引用参数（间接）到达。规则 Call 为得到 $T_\mathfrak{h}$ 而执行的其余转换，实质上是通过重命名变量，将返回符号 $\pi$ 替换为任意一个引用变量 $r$，该变量在 $m$终止时被赋值。

### 5.3 基于 FullSummary 的程序分析

我们将 Summary $(m, D)$ 记作上述过程，该过程根据一组摘要 $D$ 计算已实现方法 $m$ 的摘要，其中这组摘要是针对该方法可能调用的每个方法计算得出的。于是，为程序中的每个已实现方法计算正确摘要，归结为求解以下方程组：

$$\begin{cases} Sum_{m_1} = \textsc{Summary}\,(m_1, \{Sum_m \mid m \in \text{Deps}\,(m_1)\}) \\ \vdots \\ Sum_{m_n} = \textsc{Summary}\,(m_n, \{Sum_m \mid m \in \text{Deps}\,(m_n)\}) \end{cases} \tag{4}$$

其中，Deps $(m)$ 给出可能被 $m$调用的方法集合。式（4）中的每个未知变量都是该程序中某个方法的摘要，而该方程组自然导出一个依赖图 $G$，其中节点为方法；$G$对应于调用图的过度近似。如果待分析程序中不存在（潜在的）递归方法（即 $G$ 是无环的），则可以通过对 $G$ 进行简单的拓扑遍历来计算每个摘要。然而，如果 $G$ 是循环的，即包含递归方法，这种方法便不适用。为解决这一问题，我们采用抽象解释领域中成熟的一种技术：遵循混沌迭代策略 [8]，求解式（4）的不动点解。

如果对于任意固定的 $m$，Summary $(m, \cdot)$ 都是单调的，那么该解决方案就是可靠的。我们为每个方法 $m$ 计算的摘要属于一个域，该域还可以配备一个最小元素和一个偏序，从而构成一个摘要格 $\mathbf{Z}_m$，$\sqsubseteq_{\mathcal{Z}_m}$。我们非正式地假定，$m$ 的可能摘要按许可程度递减的顺序排列，因此最小元素对应于最宽松的摘要。该格的底元素对应于这样一个摘要：其守卫始终成立（即 tt），且其 Effect（如果有）会返回一个低级别的值，并且不会执行任何对象变更。我们证明，过程 Summary $(m, .)$ 关于格 $\mathbf{Z}_m$，$\sqsubseteq_{\mathcal{Z}_m}$ 是单调的。请注意，采用混沌迭代策略可能还需要定义一个合适的拓宽算子，以合并摘要，从而加速向不动点的稳定（或者在 $\mathbf{Z}_m$ 为无限集时强制收敛）。不过请注意，当前的实现并未采用上述拓宽步骤来加速收敛，而是依赖于格 $\mathbf{Z}_m$ 的有限性。我们将在第 7 节看到，实证实验也证实了这一选择是合理的。

在我们的运行示例中，假设 flushBits 可能调用 FileImageOutputStream.write。那么，计算 write 的摘要的过程大致包括：假设一个最宽松的摘要，例如 摘要$^0$ `flushBits`，用于 flushBits；然后使用该摘要计算 write 的摘要，

使用 摘要$^0$ $_{\texttt{flushBits}}$，然后依次为每个方法计算摘要，直到达到稳定状态：

(1) 为 flushBits 生成最宽松的摘要$Sum^0$ $_{\texttt{flushBits}}$；(2) 计算 $Sum^0_{\texttt{write}}$ = 摘要 $\Big($写入 $\{Sum^0$ $_{\texttt{flushBits}}, Sum^0_{\texttt{RandomAccessFile.write}}\}\Big)$；(3) 计算 $Sum^{i+1}_{\texttt{flushBits}}$ = 摘要 (flushBits $\{Sum^i_{\texttt{write}}\}$)，直到稳定，即 $Sum^{i+1}_{\texttt{flushBits}} = Sum^i_{\texttt{flushBits}}$；(4) 计算 $Sum^{i+1}_{\texttt{write}}$ = 摘要 (写入 $\{Sum^{i+1}_{\texttt{flushBits}}, Sum^{i+1}_{\texttt{RandomAccessFile.write}}\}$)，然后从步骤 (3) 重新开始，直到稳定，即 $Sum^{i+1}_{\texttt{write}} = Sum^i_{\texttt{write}}$。

为了证明我们的摘要计算方法是可靠的，我们需要证明过程 Summary $(m, .)$ 关于 $\mathbb{Z}_m$, $\sqsubseteq_{\mathcal{Z}_m}$ 是单调的。设 $Sum$ = 守卫, $Effect$ 和 $Sum'$ = 守卫′, $Effect'$ 是两个摘要。我们规定 $Sum$ $\not\sqsubseteq_{\mathcal{Z}_m} Sum'$ 当且仅当 守卫′ 比 守卫 更宽松，即对于所有状态 $s$，守卫′$(s)$ $\Longrightarrow$ 守卫 $(s)$。

性质 1. 给定一个方法 $m$，摘要 $(m, \cdot)$ 相对于 $\langle \mathcal{Z}_m, \sqsubseteq_{\mathcal{Z}_m} \rangle$是单调的。

证明概要：. 参见附录 A。 □

**6 形式化正确性**

我们证明，任何由其信息流摘要守卫进行守卫的方法都能保证终止不敏感非干扰 [42]。这一概念指出，对于任意初始状态 $s$ 和 $s'$，只要它们的秘密部分可能不同，程序分别从状态 $s$ 和 $s'$ 运行所产生的观测序列就要么相同，要么其中一个是另一个的前缀。后一种情况的原因在于，这一概念是终止不敏感性质。除了属于每个被调用方法语义的符号变量（即堆模型和安全类型环境）外，我们还需要考虑执行中使用的扩展堆模型，以及包含局部变量值的调用栈。令 S 表示程序的（符号）完整语义，令 $[\![S]\!] = \mathrm{Q}, \mathrm{I} \rightarrow, \mathrm{Q}_0$ 表示描述其扩展语义的自动机，其中 Q 是状态集，I 是输入集，$\rightarrow \subseteq \mathrm{Q} \times \mathrm{Q}$ 是转换集，$\mathrm{Q}_0$ 是初始状态集。程序状态$q \in \mathrm{Q}$ 定义为 $\mathrm{f}_m :: \eta$，其中 $\mathrm{f}_m$ 是当前正在执行的方法的栈帧，而 $\eta$ 是调用上下文的栈。方法栈帧是一个元组 $\mathrm{Prm}, \mathrm{h}\ \ell, \mathrm{X},$，其中 $\ell$ 是当前位置，Prm 是方法原始变量 $P$的赋值，h 是扩展堆变量 $\mathbb{V}_\mathfrak{h}$ 的赋值，而 X 是语义中状态变量的当前赋值。X 由以下部分组成：(i)Ω: $P \cup R \rightarrow \mathbb{L}$，用于表示原始变量 $P$和引用 $R$的安全类型环境；(ii) $\Gamma: R \rightarrow \mathbb{L}$，用于表示对象安全类型环境。随后，我们基于两个状态 $q_1$ 和 $q_2$之间的低等价关系定义非干扰，该关系规定状态的公开部分不可区分。

我们称两个原始变量的赋值 $\mathrm{Prm}_1$ 和 $\mathrm{Prm}_2$ 是不可区分的，记作 $\mathrm{Prm}_1 =_\perp \mathrm{Prm}_2,$，当且仅当对所有 $v \in P$（其中 $v = \perp$），$\mathrm{Prm}_1(v) = \mathrm{Prm}_2(v)$。我们将这一概念扩展到由引用集合可达的堆的部分。若两个扩展堆的低敏感引用的堆相关关系和原始字段相同，即 (i) 对应于它们堆的低敏感部分的引用图同构，且 (ii) 它们低敏感引用的原始字段的赋值相同，则称这两个扩展堆不可区分。引用图展示了引用之间的堆关系。

我们使用记号 $q(a)$ 表示状态 $q$ 中 $a$ 的值。此外，当且仅当对于所有 $r \in R$，$q_1(\vec{r}) \sqsubseteq q_2(\vec{r})$ 时，我们称 $q_1(\Gamma) \sqsubseteq q_2(\Gamma)$。我们称两个状态 $q_i = \mathrm{Prm}_i, \mathrm{h}_i, \ell_i, \mathrm{X}_i, :: \eta_i$ $i \in \{1, 2\}$ 是兼容的，记作 $q_1 \approx q_2$，当且仅当 $q_1(\ell) = q_2(\ell)$、$q_1(\mathrm{Prm}) =_\perp q_2(\mathrm{Prm})$、$q_1(\mathrm{h}) =_\perp q_2(\mathrm{h})$、$q_1(\Omega) = q_2(\Omega)$、$q_1(\Gamma) = q_2(\Gamma)$，且 $q_1(\eta) = q_2(\eta)$。设 $q \xrightarrow{\beta}_* q'$ 是完整语义的一次非零长度执行（即具体转移关系 $\rightarrow$ 的自反传递闭包），从

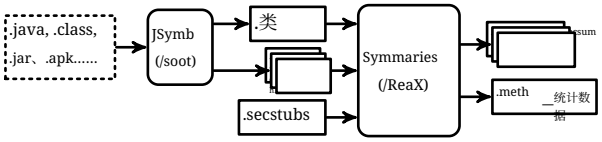



**图 6.** Symmaries 的工作流程。

状态 $q$ 到状态 $q'$，其中 $\beta \in \{o,\ \bot\}$。此执行要么以执行一条在某个通道上输出的语句结束（即 $\beta = o$），要么继续执行而不产生观察（即 $\beta = \bot$）。我们将永远不会到达观察点的执行记作 $q \xrightarrow{\bot}_{*}$。我们将完整语义中两个状态之间的低等价关系定义如下：

定义 *6.1*（低双模拟）。若两个状态 $q_1$ 和 $q_2$ 低双模拟，记作 $q_1 \sim_{low} q_2$，当且仅当 $q_1 \approx q_2$，并且如果 $q_1 \xrightarrow{o}_{*} q_1'$，那么要么 (a) 存在 $q_2'$，使得 $q_2 \xrightarrow{o}_{*} q_2'$ 和 $q_1' \sim_{low} q_2'$；要么 (b)$q_2 \xrightarrow{\bot}_{*}$，反之亦然。

对于程序 $c$ 的任意两个初始状态 $q_0$ 和 $q_0'$，若 $q_0 \approx q_0'$，且$q_0 \sim_{low} q_0'$成立，则称 $c$ 保证非干扰。

定理 6.2（非干扰）。对于任意具有安全摘要 ⟨守卫$_m$、$Effect_m$⟩的方法 $m$，以及任意满足 $q_1 \approx q_2$ 和 $q_i \models Guard_m$、$i \in \{1,2\}$的初始状态 $q_1$ 和 $q_2$，有 $q_1 \sim_{low} q_2$。

证明。见附录 D。 □

**7 实现与评估**

为支持所提出的方法，我们实现了一套新工具链，其架构如图 6 所示。JSymb 模块接收一个应用程序，并使用soot [49] 生成其 Jimple 格式的中间 Java 字节码表示，随后将其转换为Symmaries的输入格式。输入包括数据类型定义（即类、字段和继承关系，存储在.classes 文件中）、待分析的方法列表，以及对排除方法行为的可选规范（例如本机代码，或任何其他库方法或应用程序方法，存储在.secstubs 文件中）。JSymb 使用Java 实现，代码约有 5K 行（LOC）；而Symmaries使用OCaml 编写，代码约有 18K 行。

为应对可能计算密集的分析，我们为Symmaries配置了可自定义的超时和内存限制选项，以中断运行时间过长或占用内存过多的分析；以这种方式跳过的方法随后会被视为第三方方法，这意味着我们假定它们满足所需的信息流控制属性，但它们只能在低上下文中被调用，并且只能接收低敏感输出。不过请注意，跳过某个方法并不意味着它完全无法分析；相反，这通常表明我们需要在不同配置下进行分析。

我们的评估围绕以下研究问题展开：
(1) **RQ1：** Symmaries 在为 Java 程序合成安全摘要方面有多精确？(2) **RQ2：** Symmaries 处理实际应用的可扩展性如何？(3) **RQ3：** Symmaries的易用性如何？(4) **RQ4：** 它与类似工具相比如何？

**RQ1：精确率**

我们将Symmaries 应用于 Java API 库（OpenJDK 8），以提取其安全规范。Java API 库是我们分析的合适基准，因为它是使用最广泛且最成熟的标准库之一

。几乎每个 Java 程序都依赖其核心软件包，因此它是提取安全规范的相关目标。在实验中，我们重点分析了两个主要软件包：java.io 和 java.util，它们涵盖输入/输出通信和数据集合。我们提取了它们的所有依赖项，包括来自其他软件包的依赖项，以确保分析的完整性。不过，我们排除了涉及并发和反射的方法，因为我们的工具对这些特性的支持不如对本机代码的支持完善。对于所有排除的方法，我们通过查阅 Java API 文档和源代码，半手动地提供了安全规范，以捕获其行为。我们还使用 AI 生成注解语言中的初始规范草案，然后手动验证并修正这些规范，因为其中包含若干错误。为定义汇点和源点集合，我们提取了直接向外部源写入或从外部源读取的方法，这些操作要么通过本地方法完成，要么通过内存写入完成。在我们的方法中，源是符号性的，即其安全标签未知，并且根据上下文可能是高敏感或低敏感。例如，方法 java.io.FileInputStream.readByte 是一个从文件输入流中读取数据的本地方法，它是一个符号源，根据所读取文件的敏感性，可以充当低敏感源或高敏感源。因此，调用 readByte 的方法 read 也在其摘要（objlevel_r1）中包含从 p0 到 r1 的流，其中 p0 表示 readByte 所读取数据的安全级别。换句话说，如果 readByte 返回高敏感数据（即 p0 为高敏感），那么 objlevel_r1 也将变为高敏感：

int java.io. FileInputStream :read (byte [] r1) { // --- 守卫条件 --- guard :=tt; // --- 安全级别更新 --- level_ret := pc ⊔ level_r1 ⊔ level_this ⊔ objlevel_r1 ⊔ objlevel_this ; objlevel_r1 := p0 ⊔ pc ⊔ objlevel_r1 ; .... }

我们评估了 *Symmaries* 的精确率，其定义为具有正确规范的方法所占的比例。精确率可以从三个方面进行评估：安全守卫的精确率、为每个方法提取的信息流的正确性，以及推断出的别名关系的精确率。然而，由于需要人工验证的方法数量庞大，我们仅根据安全守卫的精确率来衡量精确率。我们认为，这能够合理地反映工具的整体精确率，因为不精确性的主要来源是共可达性算法引入的过度近似，而该算法仅应用于完整的程序语义一次。随后，所得输出被分解为上述三个相应组成部分。因此，分析复杂性导致的任何过度近似通常都会影响这三个维度，我们相信，研究其中一个方面可以对整体精确率提供相当可靠的估计。

Symmaries 的精度取决于分析所采用的堆模型。因此，我们使用两种不同的堆模型进行了实验：一种在建模别名关系时精度较低的模型LPrec，以及另一种精度较高、具有字段敏感性的模型HPrec。请注意，对于这两种堆模型，我们的分析仍然是流敏感、对象敏感和上下文敏感的。表1展示了针对已分析的 Java 包，使用两种堆模型HPrec 和LPrec所得的精度结果。列 #meth 表示在每个软件包及其子包中检查的方法总数；#skipd 统计工具因内存不足或超时而未能分析的方法数；#strict 表示具有严格守卫条件的方法数；#aliasing 表示会更新别名关系的方法数；prec1 则表示具有精确守卫的方法所占的百分比。我们以半自动方式检查了守卫条件的精度。我们预期，所有无法到达汇方法的方法都应

**表1.** 精度结果。

| 软件包 | #方法 | HPrec 结果 | | | | | LPrec 结果 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | #skipd | #严格匹配 | #别名关系 | prec1 (%) | prec2 (%) | #skipd | #严格匹配 | 精确度1 (%) | 精确度2(%) |
| java.util | 1460 | 201 | 251 | 523 | 82.8 | 99.8 | 1 | 31 | 98 | 100 |
| java.io | 546 | 27 | 77 | 208 | 85.9 | 99.8 | 0 | 5 | 99.1 | 99.4 |
| java.lang | 278 | 9 | 39 | 88 | 86 | 100 | 0 | 0 | 100 | 100 |
| java.nio | 140 | 12 | 37 | 65 | 73.6 | 89.4 | 0 | 0 | 100 | 100 |
| java.text | 42 | 2 | 4 | 20 | 90.5 | 100 | 0 | 0 | 100 | – |
| java.security | 24 | 6 | 3 | 11 | 87.5 | 100 | 0 | 1 | 95.8 | 100 |
| javax.security | 12 | 1 | 3 | 2 | 75 | 50 | 0 | 0 | 100 | 100 |
| **总计** | 2502 | 258 | 414 | 919 | 83.5 | 99 | 1 | 37 | 98.5 | 99.9 |

**表 2**。开源项目列表

| Name | 描述 |
|---|---|
| Apache BookKeeper | BookKeeper 是一款可扩展的存储服务，专为实时工作负载设计，可将记录条目按序列存储这些序列称为账本，并会在多台服务器之间复制，以确保可靠性。 |
| Apache RocketMQ | 一个分布式消息与流处理平台。 |
| Apache Kerby | 一种常见的、基于 Java 的 Kerberos V5 协议实现。 |
| AWS Secrets Manager | 一种用于管理凭据、API 密钥等敏感信息的服务。 |
| EventBus | 一个用于在 Android 和 Java 应用程序中进行基于事件的发布/订阅通信的开源库。 |
| Keyloack | 一个用于身份和访问管理的开源项目，可创建具有自定义角色和群组的用户数据库。 |
| Snappy Java | 一款快速的压缩器／解压缩器。 |
| ZXing | ZXing（“斑马线”）是一个用 Java 实现的开源多格式条码图像处理库。 |

具有真守卫条件。如果某个方法无法到达汇方法，但其守卫条件不为真，我们就将其归类为严格匹配。对于能够到达汇方法的方法，我们手动检查了守卫条件是否严格。

HPrec 的精确率为 83.5%，而 LPrec 则达到 98.5%。尽管人们可能会认为 HPrec能够得到更精确的结果，但它使用更多变量来建模堆，这会导致分析更加复杂，也使其在传播守卫条件时更容易产生过度近似。此外，该堆模型的精度对别名规范的影响更大。若要全面比较这两个域，还需要检查它们推断出的别名规范。不过，我们的目标并非直接比较这些堆模型，而是展示在不同堆假设下整个算法的精确率。此外，我们观察到，能够到达被跳过方法的方法往往具有严格守卫，因为工具会将此类方法视为行为悲观的第三方代码。这种不精确性会传播到所有直接或间接调用被跳过方法的方法。换言之，我们的精度结果受到可扩展性限制的影响，从而降低了整体精度。移除被跳过的方法及所有依赖于它们的调用方后，精度会有所提高，如 prec2 所示。我们认为这些精度结果非常令人鼓舞。即使工具无法分析某些方法，用户也可以手动为该方法提供摘要或将其排除，同时仍能分析应用程序的其余部分。

**RQ2：可扩展性评估**

本部分评估考察了我们方法的可扩展性。我们从 GitHub 收集了多个热门的真实世界开源项目并对其进行分析，以推断其摘要规范。表 2 列出了所选项目及其描述，项目规模从 2K 行到约 220K 行 Java 字节码不等。所有实验均在一台运行 Ubuntu Linux 的台式电脑上进行，该电脑配备 32 GB 内存和 3.5 GHz 四核英特尔酷睿 i7 CPU。为获得合理的计时结果，我们将超时设为 5 分钟，并限制

**表 3.** 选定开源项目的分析结果

| 应用名称 | 大小 | | 性能统计 | | | | 摘要统计 | |
|---|---|---|---|---|---|---|---|---|
| | 字节码指令 | #meth | 已处理 (%) | $T_{tot}$ (s) | $T_{wc}$ (s) | $T_{meth}$ (ms) | #守卫 | #流程 |
| EventBus | 2.9k | 177 | 100 | 5.25 | 9.0 | 30 | 10 | 37 |
| Snappy Java | 5.6k | 193 | 100 | 10.06 | 14.5 | 52 | 13 | 30 |
| AWS Secrets Manager | 35.4k | 3970 | 100 | 129.6 | 457 | 33 | 0 | 188 |
| Keyloack | 43.2k | 4441 | 100 | 123.474 | 307 | 28 | 22 | 441 |
| ZXing | 54.4k | 1764 | 100 | 137.4 | 236 | 78 | 1 | 245 |
| Apache Kerby | 62k | 4637 | 100 | 162.6 | 596 | 35 | 148 | 711 |
| Apache BookKeeper | 143.7k | 6527 | 99.9 | 204 | 1265 | 26 | 313 | 1006 |
| Apache RocketMQ | 221.5k | 13694 | 99.99 | 1504 | 5396 | 110 | 144 | 1376 |

内存用量最高为 8 GB。超过任一限制的方法摘要任务都会被跳过，并被视为具有悲观规范的第三方方法。

我们的实验表明，Symmaries 能够有效扩展，以处理大型实际应用，如表 3 所示。这些结果是使用精度最低的堆域获得的。分析应用程序所需的时间和复杂度取决于多个因素，包括分析类型（污点检查或隐式流分析）、方法数量、方法控制流图的复杂度、生成的符号控制流图中的状态变量数量等。在六个应用程序中，所有方法均已成功处理（“已处理”列）；而在RocketMQ 和 Apache BookKeeper 中，超过 99% 的方法已处理。我们的实验总共分析了 ~35,000 个方法，Symmaries 成功为其中的**99.99%** 计算出了摘要。

$T_{tot}$ 在表 3 中显示了Symmaries分析应用程序所耗费的总有效时间，$T_{wc}$ 表示相应的墙钟时间，$T_{meth}$ 显示分析一个方法的平均耗时。尽管构建某些方法的摘要耗时较长，但大多数方法都能在 2 毫秒内完成。分析一个方法所需的时间从不到 1 毫秒到约 9 秒不等，平均为 26 毫秒。分析应用程序的有效时间从 5 秒到 25 分钟不等，具体取决于应用程序的规模和已分析方法的数量。然而，准确性较低的墙钟时间则从 9 秒到约 90 分钟不等。我们的结果证明，Symmaries 能够扩展至分析大型实际应用程序。

表 3 还展示了推断出的摘要结果。#guarded 列显示守卫规范不是重言式的方法数量，即这些方法可能会根据调用上下文导致信息泄漏。此外，Symmaries 发现了若干带有错误守卫的方法。这些情况主要出现在控制流结构复杂的方法中，过度近似会导致过于严格的结果。此外，#flows 列报告了其参数与返回值之间存在信息流的方法数量。

**RQ3：可用性**

尽管我们没有开展实验性用户研究，但Symmaries的若干方面展现了它在实践中的可用性。首先，该工具集成了soot——一个广泛使用的 Java 字节码分析框架，并且只需用户进行极少的配置：待分析的应用程序或库（以 .jar 文件形式提供）、汇点和源点列表（以 .xml 文件形式提供），以及在方法定义缺失（例如本地方法）或程序的某些部分过于难以分析时，为排除的方法提供可选的安全假设（.secstubs 文件）。得益于基于摘要的分析以及符号源的概念，Symmaries 无需大量代码注释或识别入口点，而这些通常是静态代码分析所必需的。这种方法使我们能够以极少的人工工作量分析真实世界应用程序。

其次，该工具的输出以简洁、易于理解的摘要语言呈现。这种设计使用户能够结合自身情境解读并手动验证结果，也有助于用户在重复使用第三方组件或评估自身代码的安全性时做出明智的决策。论文中提供了输出示例，例如图1。

第三，Symmaries 支持一种混合工作流，在自动化与用户控制之间取得平衡。该工具旨在自动推断方法的安全摘要，但用户可以通过 .secstubs 文件覆盖这些摘要，以编写更精确的规范。得益于这种灵活性，即使某些大型或复杂的方法无法处理，用户也可以将其排除，并提供替代规范（例如，将其视为第三方库方法，或采用更简单的、对流不敏感的分析——这留待未来工作），同时仍然分析应用程序的其余部分。

最后，我们在真实世界的 Java API 库和应用程序上进行的实验，为可用性提供了间接证据。Symmaries 能够在合理的时间和内存限制内，成功处理代码行数多达数十万行的项目。处理如此大型系统的能力表明，该工具不仅在理论上可靠，而且在实践中也适用。综上，这些特性证明Symmaries 在实践中具有可用性。

**RQ3：与最先进方法的比较**

已有多种工具被提出用于从代码中进行规范推断，包括为库推断别名规范 [6, 13] 或为方法推断类似于 Symmaries 的污点规范 [10, 45, 47]，（见第 8 节）。然而，其中许多工具要么针对 Java 以外的语言，要么生成不同类型的规范，这使得与我们的摘要进行直接比较颇具挑战。即使是在少数支持 Java 的工具中，规范格式也各不相同。例如，[6] 将结果表示为路径规范，[13] 使用专用谓词，而 [27] 则综合生成源、汇和净化器。相比之下，Symmaries 将安全守卫推断为逻辑表达式，同时生成别名规范和以赋值集合表示的信息流，而这些内容无法直接映射到上述格式。最接近的相关方法是 StubDroid [2]，，它提取 Android 应用程序中源与汇之间的数据流。StubDroid 自动为 Android 框架方法生成摘要，用于污点分析，以提升 FlowDroid [3]的性能。然而，它并不健全，不支持隐式流，并且仅限于显式数据流提取。相比之下，Symmaries 同时捕获隐式流和别名信息。因此，要与现有方法进行有意义的比较，就需要它们针对 Java 方法生成与我们的规范具有可比性的规范，而目前没有任何方法能够做到这一点。因此，无法对输出结果与先前工作的结果进行详细比较。

**结果有效性的威胁**

我们的精度结果有效性面临的一个威胁是，实验仅评估了守卫条件的精确率。更全面的评估还应纳入别名关系，以及信息流，从而更完整地呈现精确率。此外，我们假设用户提供的安全存根是正确的。由于不正确的规范可能直接导致工具输出错误，这一假设是依赖用户提供的注解或输入的形式化分析工具所面临的常见威胁。

我们使用精确度较低的堆模型 LPrec开展了可扩展性实验。与精确度更高的堆域相比，该模型性能更好，理论上代价是精确率较低。然而，我们最初将该工具应用于实际应用时观察到，对于控制流结构复杂的方法，精确度更高的堆模型会生成限制更多的守卫条件，从而导致更高的假阳性率。要恰当地调查这一说法，需要投入大量精力，手动检查数千个方法的守卫条件；这项工作留待

未来开展。因此，我们认为，整体而言，LPrec 在实践中或许能提供更好的可扩展性和精确率，用它来运行Symmaries 不失为一种合理的选择。

**8 相关工作**

基于摘要的分析。 基于摘要的方法为程序分析提供了一种解决方案，主要用于解决可扩展性问题，其目标是进行模块化分析、增量分析 [18], ，甚至循环摘要。根据分析目的，摘要可以捕获各种类型的信息，例如指向关系 [32], 、数据流 [51], 、加锁行为 [4]。UBITect [55] 采用基于摘要的方法来检测 Linux 内核中的 UBI 漏洞，其方法摘要与我们的方法一样分为两部分：方法调用安全时对输入的要求，以及相对于输入对输出进行的更新。Bai 等人提出了一种基于摘要的数据流分析方法，用于检测 Linux 内核中的锁 [4]。 [25] 中的作者采用基于摘要的方法来检测多语言程序中的漏洞：他们从以客户语言编写的程序中提取语义摘要，并将其用于宿主语言的分析。这些方法侧重于检测特定漏洞，而非推断安全规范，并且都不健全。

现有的、用于安全分析的基于摘要的方法为数不多，且都侧重于污点分析。Wei 等人 [51] 提出了一种基于摘要的污点分析方法，用于识别包含本机代码的 Android 应用程序中的跨语言数据流。Stievenart 等人 [46] 针对 WebAssembly 程序提出了一种基于摘要的污点分析方法，其中函数摘要描述了执行后从参数和全局变量到返回值及全局变量的信息流。Zhang 等人 [56] 使用 DR. CHECKER [29], （一个用于检测 Linux 驱动程序中错误的框架），为入口函数构建污点流，以检测高阶污点。上述方法侧重于污点检测，而 Staicu 等人 [45] 的目标与我们类似，他们提出了一种基于摘要的方法，用于提取 JavaScript 库的污点规范。这项工作通过动态污点分析和测试来构建摘要；与我们的摘要不同，他们的摘要既不描述过程如何影响堆，也不描述何时可以使用此方法。Merlin [27] 通过概率推断和过程间数据流分析提取信息流规范，并识别方法中的潜在源、汇和净化器。然而，它不会生成描述方法行为的规范。与上述方法相比，我们的方法具有可靠性，且摘要的表达能力更强：它们将守卫条件推断为逻辑表达式，并以赋值的形式表示信息流规范和别名规范。现有方法或许具有更好的可扩展性，但这是以牺牲可靠性为代价的。

基于摘要的方法已被用于解决形式化验证的可扩展性问题，例如软件模型检查 [43], 、通过抽象解释 [11], 验证程序中的数值属性，以及通过关系抽象解释 [9]构造过程摘要。这些方法构造的摘要是关系，用于关联输入和输出的值。相比之下，我们计算的摘要是契约，我们认为它们在信息流控制分析的语境中更易于理解。除了提供效应之外，我们还提供一个守卫，说明何时可以安全地使用此方法。Gulavani 等人 [16] 的形状分析（用于发现被操作数据结构的结构属性）也使用基于逻辑的摘要，但依赖于分离逻辑 [41]的某些片段和／或扩展。据我们所知，目前尚无研究使用基于摘要的形式化分析来解决安全问题，尤其是在信息流分析的语境中。

库规范推断。 目前已有多种推断 API 规范的方法。USpec [13] 使用无监督学习推断 API 中的别名关系，并据此增强指向分析。Atlas [6] 通过

动态执行自动推断指向规范。DAInfer [50] 利用自然语言处理和优化技术，从 API 文档中提取别名关系。Spectre [22] 利用模糊测试技术为库 API 生成别名规范。 [13] 中的方法通过分析 API 使用模式来学习规范，无需手动注释、访问源代码或执行 API。Ramanathan 等人 [36] 通过谓词挖掘和过程间数据流分析推断 API 调用的前置条件。尽管其中一些方法与我们的工作相关，因为它们推断别名关系，但与我们的方法不同，它们不会提取安全摘要，也缺乏可靠性。Clapp 等人 [10] 从具体执行中挖掘库的显式信息流，Tileria 等人 [47] 则从 API 文档中提取污点规范。然而，与我们的方法不同，这些方法并不可靠，无法捕获隐式流，也不会在提取的规范中考虑别名关系。

信息流控制验证。 确保非干扰的静态分析方法已在研究领域得到广泛研究。绝大多数已提出的信息流控制方案都侧重于类型系统 [5, 34, 42], ，并且已有多种面向实际编程语言的工具被开发出来，用于验证此类属性，例如 JFlow/JIF [31] 和 KeY [1]。其他方法则依赖更通用的静态分析技术，例如过程间数据流分析 [40] 和程序切片 [21]。此类方法中值得关注的框架包括 JOANA [17], 、DroidSafe [15], 和 FlowDroid [3]。此外，还有专为 Web 应用程序 [19, 30]和 Android 应用 [26]量身打造的实用信息流控制方案。然而，这些方案大多无法处理隐式流，或缺乏可靠性。在要求可靠性的情况下，往往会牺牲实用性，例如拒绝任何在高安全级上下文中包含方法调用的程序 [28], ，或像 [1, 31]那样要求大量手动注释。这些方法旨在通过静态分析检测漏洞，而我们的重点在于推断安全相关行为的抽象表示。 在这些方法中，只有 JFlow/JIF [31] 会推断变量的安全标签。然而，我们的方法引入了表达能力更强的守卫条件，也可以包含有关堆结构的条件。此外，与Symmaries相比，JIF 不会计算方法更新，并且严重依赖手动注释。

开源软件安全。 开源代码中的静态安全分析和漏洞检测旨在确定程序是否符合特定的安全要求。它们根据给定的安全策略将代码归类为安全或不安全，但该策略可能与使用者的策略不同。然而，这些方法不会针对任意安全策略分析代码的安全行为，也不会说明代码被复用时可能产生的安全影响。软件复用策略 [37]可分为两类：白盒方法，即开发者可以访问组件内容并对其进行修改；以及黑盒方法，即代码按“原样”使用，开发者无法访问代码。软件开发依赖包管理器和代码仓库，使开发者能够共享代码并复用第三方代码。识别开源代码中漏洞的方法可分为两类：基于版本控制的方法，检查代码克隆的特定版本是否包含已报告的漏洞 [12, 53], ；以及基于代码的方法，通过与已知的易受攻击代码进行匹配，检测恶意或易受攻击的代码 [23, 54]。V1SCAN [52] 是一种混合方法，结合基于版本和基于代码的方法，利用分类技术检测 C/C++ 程序中的漏洞。有多种方法可用于识别修改过的代码；例如， [53] 提出了 Centris 方法，用于识别经过修改的开源软件复用； [38] 则开发了工具 VDiOS，用于查找原始项目中已修补的漏洞。此外，旨在确保供应链完整性的现有工作，要么侧重于检测并隔离恶意软件包 [44, 57], ，要么侧重于设计可信包管理器 [24, 48]。软件成分分析（SCA）是识别程序中开源代码的过程。我们的方法与上述方法互为补充：上述方法侧重于检测易受攻击的代码是如何传播的，而我们则采取

不同的方法，提供程序安全行为的抽象表示，以帮助开发者选择要使用的安全开源代码。此外，现有的安全开源代码开发方法通常借助包管理器，使开发者能够安全地共享和复用第三方代码。现有工作主要侧重于检测并隔离恶意软件包 [44, 57]，或设计可信包管理器以确保供应链完整性 [24, 48]。然而，这些方法往往缺乏提供可靠安全保证所需的形式化基础，也无法像我们的方法那样解释代码的安全行为。

## 9 结论

我们提出了一种可扩展的方法，用于为 Java 字节码方法构造可靠的安全摘要。每个摘要由一个守卫和一个更新组成：守卫指定方法调用安全的条件，更新则描述别名关系和信息流的变化。我们借助工具支持的技术，利用共可达性分析计算这些摘要。对 Symmaries 的评估表明，该工具有效、精确且可扩展，能够成功分析大型真实世界应用程序。

作为正在进行的工作，我们正在预测难以分析的方法，并将其排除在自动分析之外，同时提供手动摘要，以提高我们方法的精确率和可扩展性（即，通过避免分析那些不太可能成功完成分析的方法来节省时间）。Symmaries 目前采用一种简单的基于名称的解析过程来构造调用图的过度近似，并求解公式 (4)。在这方面，我们可以采用其他可靠但更精确的调用图构造算法 [39]，从而提高工具在实际应用中的精确率。此外，自底向上的程序分析通常非常适合并行实现，我们计划以此提高 Symmaries 的运行时性能。另一项工作是扩展我们的方法，以生成用于控制信息流的运行时监视器。具体而言，我们将通过检查点语句扩展输入语言，并为其合成安全条件，在运行时检查这些条件以确保安全。

## 引用

[1] Wolfgang Ahrendt、Thomas Baar、Bernhard Beckert、Richard Bubel、Martin Giese、Reiner Hähnle、Wolfram Menzel、Wojciech Mostowski、Andreas Roth、Steffen Schlager 和 Peter H. Schmitt。2005。KeY 工具。软件系统建模4, 1（2005年2月），32–54。doi:10.1007/s10270-004-0058-x[2] Steven Arzt 和 Eric Bodden。2016。StubDroid：自动推断 Android 框架的精确数据流摘要。载于第*38*届国际软件工程会议论文集。725–735。[3] Steven Arzt、Siegfried Rasthofer、Christian Fritz、Eric Bodden、Alexandre Bartel、Jacques Klein、Yves Le Traon、Damien Octeau 和 Patrick McDaniel。2014。FlowDroid：面向 Android 应用的精确上下文敏感、流敏感、字段敏感、对象敏感且生命周期感知的污点分析。载于第*35*届 *ACM SIGPLAN* 编程语言设计与实现会议论文集（英国爱丁堡）（*PLDI '14*）。ACM，纽约，美国纽约州，259–269。doi:10.1145/ 2594291.2594299[4] Jia-Ju Bai、Tuo Li 和 Shi-Min Hu。2022。DLOS：有效检测操作系统内核中的死锁。载于*2022 USENIX* 年度技术会议，*USENIX ATC 2022*，美国加利福尼亚州卡尔斯巴德，*2022*年*7*月*11*日至*13*日，Jiri Schindler 和 Noa Zilberman（编）。USENIX 协会，367–382。https://www.usenix.org/conference/atc22/presentation/bai [5] Gilles Barthe、David Pichardie 和 Tamara Rezk。2007。一种经过认证的轻量级非干扰 Java 字节码验证器。载于第*16*届欧洲程序设计研讨会论文集 （葡萄牙布拉加）（*ESOP'07*）。Springer 出版社，柏林，海德堡，

125 140 =1762174 1762189 − 。 2018

[6] http://dl.acm.org/citation.cfm?id 。Osbert Bastani、Rahul Sharma、Alex Aiken 和 Percy Liang。。指向规范的主动学习。载于第*39*届 *ACM SIGPLAN* 编程语言设计与实现会议论文集。678–692。[7] Nicolas Berthier 和 Narges Khakpour。2023。用于多态信息流守卫推断的符号化抽象堆。载于第*24*届验证、模型检测与抽象解释国际会议论文集（*VMCAI*）。LNCS，美国波士顿。

[8] François Bourdoncle。1993。采用扩宽的高效混沌迭代策略。载于《程序设计及其应用中的形式化方法》，Dines Bjørner、Manfred Broy 和 Igor V. Pottosin（编）。Springer Berlin Heidelberg，柏林，海德堡，128–141。[9] Rémy Boutonnet 和 Nicolas Halbwachs。2019。用于过程间程序分析的析取关系抽象解释。载于《验证、模型检测与抽象解释》（VMCAI '19），Constantin Enea 和 Ruzica Piskac（编）。Springer International Publishing，Cham，136–159。[10] Lazaro Clapp、Saswat Anand 和 Alex Aiken。2014。从具体执行中挖掘显式信息流规范。载于 FSE，第5卷。24–82。[11] Patrick Cousot 和 Radhia Cousot。1977。抽象解释：用于通过构造或近似不动点进行程序静态分析的统一格模型。载于第4届 ACM SIGACT-SIGPLAN 编程语言原理研讨会论文集（美国加利福尼亚州洛杉矶）（POPL '77）。ACM，纽约，美国纽约州，238–252。doi:10.1145/512950.512973[12] Ruian Duan、Ashish Bijlani、Meng Xu、Taesoo Kim 和 Wenke Lee。2017。大规模识别开源许可证违规和零日安全风险。载于2017年 ACM SIGSAC计算机与通信安全会议论文集，CCS 2017，美国得克萨斯州达拉斯，2017年10月30日至11月3日，Bhavani Thuraisingham、David Evans、Tal Malkin 和 Dongyan Xu（编）。ACM，2169 2185 10 1145 3133956 3134048 – 。doi: . / 。Jan Eberhardt、Samuel [13] Steffen、Veselin Raychev 和 Martin Vechev。2019。API 别名规范的无监督学习。载于第40届 ACM SIGPLAN 编程语言设计与实现会议论文集。745–759。[14] J. A. Goguen 和 J. Meseguer。1982。安全策略与安全模型。载于1982年 IEEE 安全与隐私研讨会。IEEE 计算机学会，洛斯阿拉米托斯，美国加利福尼亚州，11 20 10 1109 1982 10014 – 。doi: . /SP. 。Michael I. [15] Gordon、Deokhwan Kim、Jeff H. Perkins、Limei Gilham、Nguyen Nguyen 和 Martin C. Rinard。2015。DroidSafe 中 Android 应用程序的信息流分析。载于第22届网络与分布式系统安全研讨会，NDSS 2015，美国加利福尼亚州圣迭戈，2015年2月8日至11日。互联网协会。https://www.ndss-symposium.org/ndss2015/information-flow-analysis-android-applications-droidsafe [16] Bhargav S. Gulavani、Supratik Chakraborty、Ganesan Ramalingam 和 Aditya V. Nori。2009。自底向上形状分析。载于第16届静态分析国际研讨会论文集（美国加利福尼亚州洛杉矶）（SAS '09）。Springer 出版社，柏林，海德堡，188 204 10 1007 978 3 642 03237 0_14 – 。doi: . / - - - - Christian Hammer 和 Gregor [17] Snelting。。基于程序依赖图的流敏感、上下文敏感和对象敏感信息流控制。《国际信息安全杂志》2009 8, 6 2009 399 422 10 1007 10207 009 0086 1（10 [18] 月），– 。doi: . /s - - - Fei He、Qianshan Yu 和 Liming Cai。。用于软件回归验证的高效摘要复用。《IEEE 软件工程汇刊》2022 48, 4 2022 1417 1431 10 1109 2020 3021477（），– 。doi: . /TSE. 。Daniel Hedin、 [19] Arnar Birgisson、Luciano Bello 和 Andrei Sabelfeld。2014。JSFlow：跟踪 JavaScript 及其 API 中的信息流。载于第29届 ACM 应用计算研讨会论文集。ACM，1663–1671。[20] Yann Hietter、Jean-Marc Roussel 和 Jean-Jacques Lesage。2008。状态模型转移条件的代数综合。载于第9届离散事件系统国际研讨会（WODES '08）。IEEE，187–192。[21] John B. Kam 和 Jeffrey D. Ullman。1977。单调数据流分析框架。《Acta Inf.》7, 3（1977年9月），305–317。doi:10.1007/BF00290339[22] Shuangxiang Kan、Yuekang Li、Weigang He、Zhenchang Xing、Liming Zhu 和 Yulei Sui。2025。Spectre：通过模糊测试为库 API 自动生成别名规范。《ACM 软件工程与方法学汇刊》（2025）。[23] Seulbae Kim、Seunghoon Woo、Heejo Lee 和 Hakjoo Oh。2017。VUDDY：一种可扩展的易受攻击代码克隆发现方法。载于2017年 IEEE 安全与隐私研讨会，SP 2017，美国加利福尼亚州圣何塞，2017年5月22日至26日。IEEE 计算机学会，595 614 10 1109 2017 62 – 。doi: . /SP. 。Trishank [24] Karthik Kuppusamy、Vladimir Diaz 和 Justin Cappos。2017。Mercury：针对社区代码库回滚攻击的带宽高效防护。载于2017 USENIX 年度技术会议，USENIX ATC 2017，美国加利福尼亚州圣克拉拉，2017年7月12日至14日，Dilma Da Silva 和 Bryan Ford（编）。USENIX 协会，673–688。https://www.usenix.org/conference/atc17/technical-sessions/presentation/kuppusamy [25] Sungho Lee、Hyogun Lee 和 Sukyoung Ryu。2020。拓展多语言静态分析的视野：从 C 代码中提取语义摘要以进行 JNI 程序分析。载于第35届 IEEE/ACM 自动化软件工程国际会议，ASE 2020，澳大利亚墨尔本，2020年9月21日至25日。IEEE，127–137。doi:10.1145/3324884.3416558[26] Li Li、Tegawendé F Bissyandé、Mike Papadakis、Siegfried Rasthofer、Alexandre Bartel、Damien Octeau、Jacques Klein 和 Le Traon。2017。Android 应用静态分析：系统性文献综述。《信息与软件技术》88（2017），67–95。[27] Benjamin Livshits、Aditya V Nori、Sriram K Rajamani 和 Anindya Banerjee。2009。Merlin：显式信息流问题的规范推断。《ACM SIGPLAN 通知》44, 6（2009），75–86。

[28] Steffen Lortz、Heiko Mantel、Artem Starostin、Timo Bähr、David Schneider 和 Alexandra Weber。2014。Cassandra：迈向面向 Android 的可认证应用商店。载于第4届 ACM 智能手机与移动设备安全和隐私研讨会论文集（美国亚利桑那州斯科茨代尔）（SPSM '14）。ACM，美国纽约州纽约，93–104。doi:10.1145/ 2666620.2666631[29] Aravind Machiry、Chad Spensky、Jake Corina、Nick Stephens、Christopher Kruegel 和 Giovanni Vigna。2017。DR. CHECKER：针对 Linux 内核驱动程序的可靠性分析。载于第26届 USENIX安全研讨会，USENIX Security 2017，加拿大不列颠哥伦比亚省温哥华，2017年8月16日至18日，Engin Kirda 和 Thomas Ristenpart（编）。USENIX 协会，1007–1024。https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry [30] Anders Møller 和 Mathias Schwarz。2014。自动检测客户端状态操纵漏洞。ACM 软件工程与方法论汇刊。 23, 4, 29 2014 30 10 1145 2531921 文章（9月），页码。doi：。/ [31] Andrew C. Myers。。1999 JFlow：实用的、主要为静态的信息流控制。载于第届 ACM 编程语言原理研讨会（POPL）。228–241。http://www.cs.cornell.edu/andru/papers/popl99/popl99.pdf [32] Erik M. Nystrom、Hong-Seok Kim 和 Wen-mei W. Hwu。2004。自底向上与自顶向下的上下文敏感、基于摘要的指针分析。载于《静态分析》，第11届国际研讨会，SAS 2004，意大利维罗纳，2004年8月26日至28日，论文集（《计算机科学讲义》，第3148卷），Roberto Giacobazzi（编）。Springer，165–180。doi:10.1007/978-3-540-27864-1 14 [33] Amir Pnueli 和 Roni Rosner。1989。反应式模块的综合。载于第16届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集（美国得克萨斯州奥斯汀）（POPL '89）。计算机协会，美国纽约州纽约， 179 190 10 1145 75277 75293 –。doi：。/。François Pottier 和 Vincent Simonet。。[34] 2003 ML 的信息流推断。ACM 程序设计语言与系统汇刊 25，1（1月 2003 117 158 10 1145 596980 596983 ），–。doi：。/。Peter J. G. Ramadge 和 W. Murray Wonham。1989。[35] 离散事件系统的控制。IEEE 会刊；离散事件系统动力学特刊 77, 1 1989 81 98 10 1109 5 21072 （），–。doi：。/。Murali Krishna Ramanathan、Ananth Grama 和 Suresh Jagannathan。。[36] 2007 使用谓词挖掘进行静态规范推断。ACM SIGPLAN 通知 42，6（2007），123–134。[37] Thiagarajan Ravichandran 和 Marcus A. Rothenberger。2003。软件复用策略与组件市场。ACM 通讯 46, 8 2003 109 114 10 1145 859670 859678 （），–。doi：。/。David Reid、Mahmoud Jahanshahi [38] 和 Audris Mockus。。2022 开源软件中代码复用所导致的孤儿漏洞规模。载于第44届 IEEE/ACM 第44届软件工程国际会议，ICSE 2022，美国宾夕法尼亚州匹兹堡，5月 25 27, 2022 2104 2115 10 1145 3510003 3510216 –。ACM，–。doi：。/。Michael Reif、Florian Kübler、Michael Eichberg、Dominik Helm 和 Mira Mezini。2019。Judge：识别、理解和评估调用图中的不健全性来源。载于第28届 ACM SIGSOFT 软件测试与分析国际研讨会论文集（中国北京）（ISSTA 2019）。ACM，美国纽约州纽约，251–261。doi:10.1145/3293882.3330555[40] Thomas Reps、Susan Horwitz 和 Mooly Sagiv。1995。通过图可达性进行精确的过程间数据流分析。载于第22届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集（美国加利福尼亚州旧金山）（POPL ' 95 49 61 10 1145 199448 199462 ）。ACM，美国纽约州纽约，–。[41] doi：。/。John C. Reynolds。。2002 分离逻辑：用于共享可变数据结构的逻辑。载于第17届 IEEE 计算机科学逻辑年度研讨会论文集（LICS '02）。IEEE 计算机学会，美国华盛顿特区，55–74。http://dl.acm.org/citation.cfm?id =645683 664578 。A. Sabelfeld 和 A. C. Myers。。基于语言的信息流安全。IEEE 通信领域[42] 精选期刊 21, 1 2003 2003 5 19 10 1109 2002 806121 （1月），–。doi：。/JSAC.。Ondrej Sery、Grigory Fedyukovich [43] 和 Natasha Sharygina。。2012 有界模型检测中的基于插值的函数摘要。载于第7届海法硬件与软件验证及测试国际会议论文集（以色列海法）（HVC'11）。Springer 出版社，德国柏林、海德堡，160–175。doi:10.1007/978-3-642- 34188-5 15 [44] Cristian-Alexandru Staicu、Michael Pradel 和 Benjamin Livshits。2018。SYNODE：理解并自动防止针对 NODE.JS 的注入攻击。载于第25届网络与分布式系统安全年度研讨会，NDSS 2018，美国加利福尼亚州圣迭戈，2018年2月18日至21日。互联网协会。http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_07A-2_Staicu_paper.pdf [45] Cristian-Alexandru Staicu、Martin Toldam Torp、Max Schäfer、Anders Møller 和 Michael Pradel。2020。提取 JavaScript 库的污点规范。载于 ICSE '20：第42届软件工程国际会议，韩国首尔， 27 19 2020, 198 209 10 1145 3377811 3380390 6月至7月，[46] Gregg Rothermel 和 Doo-Hwan Bae（编）。2020 。ACM，–。doi：。/。Quentin Stiévenart 和 Coen De Roover。。面向 WebAssembly 程序的组合式信息流分析。载于第20届 IEEE 源代码分析与操作国际工作会议，SCAM 2020，澳大利亚阿德莱德，2020年9月28日至10月2日。IEEE，13–24。doi:10.1109/SCAM51674.2020.00007

[47] Marcos Tileria、Jorge Blasco 和 Santanu Kumar Dash。2024。DocFlow：从软件文档中提取污点规范。载于 *IEEE/ACM* 软件工程国际会议第*46*届论文集。1–12。[48] Santiago Torres-Arias、Hammad Afzali、Trishank Karthik Kuppusamy、Reza Curtmola 和 Justin Cappos。2019。in-toto：为比特和字节提供从农场到餐桌的保障。载于 第*28*届 *USENIX*安全研讨会，*USENIX Security 2019*，美国加利福尼亚州圣克拉拉，*2019*年*8*月*14*日至*16*日，Nadia Heninger 和 Patrick Traynor（编）。USENIX 协会，1393–1410。
https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias [49] Raja Vallée-Rai、Phong Co、Etienne Gagnon、Laurie Hendren、Patrick Lam 和 Vijay Sundaresan。2010。Soot：Java 字节码优化框架。载于 *CASCON*首个十年高影响力论文。214–224。[50] Chengpeng Wang、Jipeng Zhang、Rongxin Wu 和 Charles Zhang。2024。Dainfer：通过神经符号优化从库文档中推断 API 别名规范。*ACM*软件工程论文集 1，FSE（2024），2469–2492。[51] Fengguo Wei、Xingwei Lin、Xinming Ou、Ting Chen 和 Xiaosong Zhang。2018。JN-SAF：用于对含本机代码的 Android 应用程序进行安全审查的精确高效、具备NDK/JNI感知能力的跨语言静态分析框架。载于 *2018*年 *ACM SIGSAC*计算机与通信安全会议论文集，*CCS 2018*，加拿大安大略省多伦多，*2018*年*10*月*15*日至*19*日，David Lie、Mohammad Mannan、Michael Backes 和 XiaoFeng Wang（编）。ACM，1137–1150。doi:10.1145/3243734.3243835[52] Seunghoon Woo、Eunjin Choi、Heejo Lee 和 Hakjoo Oh。2023。V1SCAN：使用代码分类技术发现复用 C/C++ 开源软件组件中的1日漏洞。载于 第*32*届 *USENIX*安全研讨会，*USENIX Security 2023*，美国加利福尼亚州阿纳海姆，*2023*年*8*月*9*日至*11*日，Joseph A. Calandrino 和 Carmela Troncoso（编）。USENIX 协会，6541–6556。
https://www.usenix.org/conference/usenixsecurity23/presentation/woo [53] Seunghoon Woo、Sunghan Park、Seulbae Kim、Heejo Lee 和 Hakjoo Oh。2021。Centris：一种用于识别经过修改的开源软件复用的精确且可扩展方法。载于 第*43*届 *IEEE/ACM* 软件工程国际会议，*ICSE 2021*，西班牙马德里，*2021*年*5*月*22*日至*30*日。IEEE，860–872。doi:10.1109/ICSE43902.2021.00083[54] Yang Xiao、Bihuan Chen、Chendong Yu、Zhengzi Xu、Zimu Yuan、Feng Li、Binghong Liu、Yang Liu、Wei Huo、Wei Zou 和 Wenchang Shi。2020。MVP：使用补丁增强型漏洞特征检测漏洞。载于 第*29*届 *USENIX*安全研讨会，*USENIX Security 2020*，*2020*年*8*月*12*日至*14*日，Srdjan Capkun 和 Franziska Roesner（编）。USENIX 协会，1165–1182。https://www.usenix.org/conference/usenixsecurity20/presentation/xiao [55] Yizhuo Zhai、Yu Hao、Hang Zhang、Daimeng Wang、Chengyu Song、Zhiyun Qian、Mohsen Lesani、Srikanth V. Kr- ishnamurthy 和 Paul L. Yu。2020。UBITect：一种用于检测 Linux 内核中使用前未初始化错误的精确且可扩展方法。载于 *ESEC/FSE '20*：第*28*届 *ACM* 欧洲软件工程会议与软件工程基础研讨会联合会议，线上活动，美国，*2020*年*11*月*8*日至*13*日，Prem Devanbu、Myra B. Cohen 和 Thomas Zimmermann（编）。ACM，221–232。
doi:10.1145/3368089.3409686[56] Hang Zhang、Weiteng Chen、Yu Hao、Guoren Li、Yizhuo Zhai、Xiaochen Zou 和 Zhiyun Qian。2021。在操作系统内核中静态发现高阶污点类漏洞。载于 *CCS '21*：*2021*年 *ACM SIGSAC*计算机与通信安全会议，线上活动，韩国，*2021*年*11*月*15*日至*19*日，Yongdae Kim、Jong Kim、Giovanni Vigna 和 Elaine Shi（编）。ACM，811 824 10 1145 3460120 3484798 –。doi：。/。Markus [57] Zimmermann、Cristian-Alexandru Staicu、Cam Tenny 和 Michael Pradel。2019。高风险的小世界：npm生态系统安全威胁研究。载于 第*28*届 *USENIX*安全研讨会，*USENIX Security 2019*，美国加利福尼亚州圣克拉拉，*2019*年*8*月*14*日至*16*日，Nadia Heninger 和 Patrick Traynor（编）。USENIX 协会，995–1010。
https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman

**摘要计算的可靠性证明**

令 $\Psi_0\,\Psi_1 \ldots \Psi_n$ 为用于计算方法 $M$ 的 IF 摘要的过程 Summary (., .) 的一系列执行，其中 $\Psi_i = (m_i, Sums_i)$，$m_i \in M$是步骤 $i$ 中计算 IF 摘要的方法，而 $Sums_i$ 是迄今为止为所有方法 $m_i, 0 \le i \le n$ 计算出的全部 IF 摘要的集合。为证明 Summary (., .) 关于 $\mathsf{Z}_m, \sqsubseteq_{\mathcal{Z}_m}$ 的单调性，我们应证明对任意方法 $m$（其中 $0 \le i < j \le n$），$Sums_i(m) \sqsubseteq_{\mathcal{Z}_m} Sums_j(m)$ 均成立。为此，我们对 $n$ 使用归纳法。基例显然成立。假设该性质对索引不超过 $k$ 的任意一对均成立，即 $\{0 < i, j \le k\}$。要证明该性质对 $n = k + 1$ 成立，可能有两种情况。如果不存在$\Psi_i = (m_{k+1}, Sums_i)$ $i < k + 1$，则该性质显然成立。否则，设 $m_{k+1}$ 在步骤 $i < k + 1$中被处理。根据归纳步骤，对由 $m$ 调用的任意方法 $m'$（其中 $0 \le i < j \le k$），$Sums_i(m') \sqsubseteq_{\mathcal{Z}_m} Sums_j(m')$ 均成立。设 $Sums_j(m) \not\sqsubseteq_{\mathcal{Z}_m} Sums_{k+1}(m)$，即步骤 $k + 1$ 中为 $m$ 合成的守卫 $\mathrm{Guard}_{m,k+1}$ 比在 $j$ 中计算的 $\mathrm{Guard}_{m,j}$ 更宽松，并且对于某个状态 $\gamma$，$\gamma \not\models \mathrm{Guard}_{m,j}$ 和 $\gamma \models \mathrm{Guard}_{m,k+1}$。Call 规则中的不变量限制更严格

根据归纳步骤假设，步骤 $k+1$ 中对所有状态 $\gamma$， $\gamma \models \mathrm{Guard}_{m',j} \implies \gamma \models \mathrm{Guard}_{m',k+1}$。如果对由 $m$ 调用的所有 $m' \in M$， $Sums_j(m') = Sums_{k+1}(m')$ 均成立，则显然可推出 $Sums_j(m) = Sums_{k+1}(m)$，这与我们的假设矛盾。如果存在一个被调用方法 $m'$，其中 $Sums_j(m') \sqsubset_{\mathcal{Z}_m} Sums_{k+1}(m')$，则合成守卫 $\mathrm{Guard}_{m,k+1}$ 应避开会导致到达调用 $m'$ 所在状态的一些状态；也就是说，存在状态 $\gamma$，从该状态执行 $m$ 将导致 $m$ 的新状态 $\gamma'$，这些状态受不变量 $\mathrm{Guard}_{m',k+1}$ 限制，同时 $\gamma' \models \mathrm{Guard}_{m',j}$。这意味着，为避免坏状态 $\gamma'$，应满足 $\gamma \not\models \mathrm{Guard}_{m,k+1}$，这与我们的假设矛盾，并由此证明 $Sums_j(m) \sqsubseteq_{\mathcal{Z}_m} Sums_{k+1}(m)$。

**B 过程内安全语义**

图7展示了赋值和分支语句的简化安全语义。 $f_p$ 和 $f_r$分别表示原始字段和引用字段。函数 updHeap(s) 会更新堆结构，以及受 $s$执行影响的对象的安全级别。变量$\omega$ 抽象掉了实际的分支条件。函数 Brch(CDR($a$)) 用于升级分析，以处理隐式流。有关过程内语句安全语义的详细说明，请参阅 [7]。

$$\textsc{Goto}\ \frac{}{(\langle\!\langle \text{Goto语句 } l \rangle\!\rangle; \mathbb{S})\ \mathrm{tt} \xrightarrow{\varnothing} (\text{目标}(l), \mathrm{njb})}$$

$$T_a = \begin{cases} [\vec{v} :=_\Upsilon \vec{e}] & \text{若} a = \langle\!\langle v = e \rangle\!\rangle \\ \left[\vec{v} :=_\Upsilon \vec{r} \sqcup \vec{r}\right] \sqcup \left(\!\left| r = s.f_p \right|\!\right) & \text{若} a = \langle\!\langle v = r.f_p \rangle\!\rangle \\ \left[\vec{r} :=_\Upsilon \vec{s} \sqcup \vec{s}\right] \sqcup \left(\!\left| r = s.f_r \right|\!\right) & \text{若} a = \langle\!\langle r = s.f_r \rangle\!\rangle \\ [\vec{r} :=_\Upsilon \vec{s}] \sqcup \left(\!\left| r = s \right|\!\right) & \text{如果 } a = \langle\!\langle r = s \rangle\!\rangle \\ [\vec{r} :=_\Upsilon \bot] \sqcup \left(\!\left| r = \text{空值} \right|\!\right) & \text{如果 } a = \langle\!\langle r = \text{空值} \rangle\!\rangle \\ [\vec{r} :=_\Upsilon \bot] \sqcup \left(\!\left| r = \text{新建} \uparrow \text{程序计数器} \right|\!\right) & \text{如果 } a = \langle\!\langle r = \text{新建 } c \rangle\!\rangle \\ \left(\!\left| r.f_p \uparrow \lfloor e \rfloor_\Upsilon \right|\!\right) & \text{如果 } a = \langle\!\langle r.f_p = e \rangle\!\rangle \\ \left(\!\left| r.f_r = s, \uparrow \lfloor \vec{s} \sqcup \vec{s} \rfloor_\Upsilon \right|\!\right) & \text{如果 } a = \langle\!\langle r.f_r = s \rangle\!\rangle \end{cases}$$

$$\text{赋值}\ \frac{}{(a; \mathbb{S})\ \mathrm{tt} \xrightarrow{T_a} (\mathbb{S},\ \mathrm{njb})}$$

$$\text{分支}\ \frac{\ell = \langle\!\langle \text{如果 } (e) \text{ 转到 } l \rangle\!\rangle; \mathbb{S}}{\ell \xrightarrow{\omega \wedge \neg\Upsilon \wedge e \sqsubseteq \text{程序计数器}} (\mathrm{Brch}(\mathrm{CDR}(a)), \mathrm{target}(l), \mathrm{njb}) \quad \ell \xrightarrow{\omega \wedge \neg\Upsilon \wedge e \sqsubseteq \bar{\vee}\ \omega \wedge \Upsilon \varnothing} (\text{程序计数器}, \mathrm{target}(l), \mathrm{njb})}$$

$$\ell \xrightarrow{\neg\omega \wedge \neg\Upsilon \wedge e \sqsubseteq \text{程序计数器}, \mathrm{Brch}(\mathrm{CDR}(a))} (\mathbb{S},\ \mathrm{njb}) \qquad \ell \xrightarrow{\neg\omega \wedge \neg\Upsilon \wedge e \sqsubseteq\ \bar{\vee} \neg\omega \wedge \Upsilon \varnothing} \text{程序计数器}, (\mathbb{S},\ \mathrm{njb})$$

**图7.** 过程内语句的安全语义。

**C 完整语义**

过程内语句的执行语义根据 [7] 中介绍的语义定义。本文仅讨论过程间相关语句的定义。在完整语义中，配置 定义为非空栈 $\mathrm{f}_{m_0} :: \mathrm{f}_{m_1} :: \ldots :: \eta$，由栈帧 组成，其中每个栈帧表示一次方法调用的局部状态。对于对方法 $m$ 的未终止调用，我们将栈帧 定义为元组 $\mathrm{f}_m \stackrel{\text{def}}{=} (\ell_m, \mathrm{Prm}_m, \mathbf{h}_m)$，其中 $\ell_m$ 表示其基本语义中的语义位置，$\mathrm{Prm}_m$ 是 $m$的方法体中所有原始变量的赋值 ，$\mathbf{h}_m$ 是具体堆值。我们在图8中给出完整语义，其中 $\xrightarrow{\phi\, T}$ ,和 $\Longrightarrow$ 分别是基本语义和完整语义的转换。请注意，基本语义所做的更新由转换更新捕获，而执行语义所做的更新

$$\textsc{m-Stm}\ \frac{(a;\mathbb{S}) \xrightarrow{\phi,T} (\mathbb{S}') \quad a \in \{\langle\!\langle \mathtt{return} \rangle\!\rangle, \langle\!\langle \mathtt{return}\ x \rangle\!\rangle, \langle\!\langle \mathtt{output}_l(_) \rangle\!\rangle\} \quad (\mathtt{Prm}', \mathbf{h}') = \mathsf{Apply}_{(a,\mathtt{Prm},\mathbf{h})}}{(a;\mathbb{S},\mathtt{Prm},\mathbf{h}) :: \eta \xrightarrow{\boxed{\neg\Upsilon \wedge \phi}\ \boxed{T}} (\mathbb{S}',\mathtt{Prm}',\mathbf{h}') :: \eta}$$

$$\textsc{m-Call}\ \frac{\begin{array}{c} a = \langle\!\langle [_ =]\, r.m'(w) \rangle\!\rangle \quad \ell_m = (a;\mathbb{S}) \quad \ell_m \xrightarrow{\mathtt{tt},T} \mathbb{S}' \\ \mathtt{f}_m = (\ell_m,\mathtt{Prm},\mathbf{h}) \quad \mathtt{f}_{m'} = \mathrm{NewFrame}\,(a,\mathtt{Prm},\mathbf{h}) \end{array}}{\mathtt{f}_m :: \eta \xrightarrow{\boxed{\neg\Upsilon}\ \boxed{\emptyset}} \mathtt{f}_{m'} :: \mathtt{f}_m :: \eta}$$

$$\text{m-Exit}\ \frac{a = \langle\!\langle [_ =]\, r.m'(w) \rangle\!\rangle \quad \ell_m = (a;\ \mathbb{S})\ (\mathtt{Prm}'_m, \mathbf{h}'_m) = \text{应用}^*(\mathrm{footUpd}, \mathtt{Prm}_m, \mathbf{h}_m)\ a;\ \mathbb{S}\ \mathtt{tt} \xrightarrow{T} (\mathbb{S}, \mathrm{njb})}{(\surd, \mathtt{Prm}_{m'}, \mathbf{h}_{m'}) :: (\ell_m, \mathtt{Prm}_m, \mathbf{h}_m) :: \eta \xrightarrow{\boxed{\mathtt{tt}}\ \boxed{T}} (\mathbb{S}, \mathtt{Prm}'_m, \mathbf{h}'_m) :: \eta}$$

其中

$$\mathrm{footUpd} = \{r = this\} \cup \{w^{-1}(p) = p \mid p \in Args_m \cap Refs_m\} \cup \begin{cases} x = \pi & \text{if } a = \langle\!\langle x = r.m(w) \rangle\!\rangle \\ \emptyset & \text{if } a = \langle\!\langle r.m(w) \rangle\!\rangle \end{cases}$$

$$\mathsf{Apply}_{(a,\mathtt{Prm},\mathbf{h})} \stackrel{\text{def}}{=} \begin{cases} (\mathtt{Prm}[\pi \mapsto \mathtt{Prm}(v)], \mathbf{h}) & \text{if } a = \{\mathtt{return}\ v\} \\ (\mathtt{Prm}, \mathbf{h}[\pi \mapsto r]) & \text{if } a = \{\mathtt{return}\ r\} \\ (\mathtt{Prm}, \mathbf{h}) & \text{if } a \in \{\mathtt{output}_(_), \mathtt{return}\} \end{cases}$$

and

$$\mathsf{Apply}^*(A, \mathtt{Prm}, \mathbf{h}) \stackrel{\text{def}}{=} \begin{cases} (\mathtt{Prm}, \mathbf{h}) & \text{if } A = \emptyset \\ \mathsf{Apply}^*(A \backslash \{a\}, \mathtt{Prm}', \mathbf{h}') & a \in A \text{ and } (\mathtt{Prm}', \mathbf{h}') = \mathsf{Apply}_{(a,\mathtt{Prm},\mathbf{h})} \end{cases}$$

**图8.** 过程间语句的完整语义。

会在目标位置中明确显示。完整语义的转换使用源配置中栈顶栈帧所关联的 SCFG 中的变量。我们将使用上述语义规则构建、用于指定某个方法完整语义的 SCFG $m$ 称为 $\mathrm{G}_m$。m-Stm 规则处理返回语句和汇点语句：它在基本语义的更新之外，还将更新扩展到方法内存 Prm 和具体堆$\mathbf{h}$（我们使用专用的返回变量 $\pi$ 保存返回值，直到到达汇点位置）。调用方法时，会借助 NewFrame 函数创建用于执行被调用方法 $m'$ 的新上下文，并将其压入栈中（m-CALL）；只有当栈顶的方法以名义模式运行时（即 $\Upsilon = \mathrm{ff}$），才会发生这种情况。该栈帧基本上由一个元组 $\mathtt{f}_{m'} = (\mathbb{S}_{m'}, \mathtt{Prm}_{m'}, \mathbf{h}_{m'})$ 构成，其中 $\mathbb{S}_{m'}$ 是目标方法 $m'$ 的方法体，其确定依据是 $r$所引用对象的类，即根据动态分派；$\mathtt{Prm}_{m'}$ 是 $m'$的局部变量和参数的赋值，并根据有效参数 $w$进行适当初始化；而$\mathbf{h}_{m'}$ 则是初始化后的具体堆。创建 $\mathtt{f}_{m'}$时，会实例化一个 SCFG $\mathrm{G}_{m'}$，其初始位置为 $(\mathbb{S}_{m'})$。其状态变量按照基本语义的初始化方程进行初始化，此外还需添加一项约束：(i) 将$Args_{m'}$ 中的每个符号状态变量赋值为 $\mathrm{G}_m$中对应状态变量的当前赋值；(ii) 针对有效参数 $w$初始化$m'$ 的符号堆抽象；(iii) 将调用上下文的安全级别$pc$ 复制到被调用方法的安全级别。只有到达被调用方法的汇点位置时，栈帧 $\mathtt{f}_m$ 的语义位置才会改变（m-Exit 规则）；此时栈顶帧会从栈中弹出，并通过 methUpd 函数更新基引用 $r$、引用参数以及返回值（如果有），从而将控制权交还给调用方方法。注意，$w^{-1}$ 是该映射的逆映射

函数 $w$ 将被调用方法中的变量映射到调用方变量。它还会根据被调用方法的摘要更新来升级调用方的类型环境（参见 Call 规则）。

**D 非干扰性证明**

定义 *D.1*（兼容状态）。我们称两个状态 $q_1$ 和 $q_2$ 兼容，记作 $q_1 \approx q_2$，当且仅当

(1) $q_1(\ell) = q_2(\ell)$，（2）
$q_1(\Omega) = q_2(\Omega)$，（3） $q_1$
$(\Gamma) = q_2(\Gamma)$，（4） $q_1$
$(\mathrm{Prm}) =_\perp q_2(\mathrm{Prm})$，（5）
$q_1(\mathbf{h}) =_\perp q_2(\mathbf{h})$，以及
（6） $q_1(\eta) = q_2(\eta)$

证明。为证明定理 1，我们需要找到一个见证双模拟关系 $\beta$，用于见证$q_1 \sim_{low} q_2$。我们将 $\beta$ 定义如下，并证明它是一个双模拟关系：

$$\beta = \{\langle q, q' \rangle \mid q \approx q' \wedge \beta(q, q') \}$$

其中

$$\beta(q, q') = \begin{array}{l} q(\Upsilon) = q'(\Upsilon) = \mathrm{ff} \wedge \\ q(pc) = q'(pc) = \perp. \end{array} \tag{5}$$

根据定义 1（低级双模拟），初始状态 $q_0$ 和 $q_0'$ 显然处于关系 $\beta$ 中，$q_0 \models x_0\text{-}all$ 和 $q_0' \models x_0\text{-}all$。令 $q$，$q' \in \beta$。根据低级双模拟的定义，我们应证明，若 $q \xrightarrow{o}_* t$，则以下两种情况之一成立：(a) 存在 $t'$，使得 $q' \xrightarrow{o}_* t'$ 和$t \beta t'$；或 (b) $q' \xrightarrow{\perp}_*$，反之亦然。如果没有从 $q'$进行观察，则情况 (b) 显然成立。如果在 $q'$之后进行了观察，则由引理 D.2 可得结论，该引理指出 $t$ 和 $t'$ 也处于关系 $\beta$ 中。□

引理 D.2（单次观察）。设 $q_1 \beta q_2$。若 $q_1 \xrightarrow{o}_* q_1'$ 且 $q_2 \xrightarrow{o}_* q_2'$，则 $q_1' \beta q_2'$。

证明。我们通过对 $q_1 \xrightarrow{o}_* q_1'$的长度进行归纳法来证明这一点。由于执行在 $q_1$中不终止，这意味着 $q_1(\ell) \neq \surd$并且 $q_1(\ell) = a$；𝕊。

**基本情形** 基本情形（即长度为 1）发生在 $a$ 是一个输出语句，例如输出$_l$ ($y$)。

对于输出$_l$($y$)，根据规则 Sink 中的 $\varphi_\ell$，$y$不能是高敏感的。因此，根据定义 D.1 中的条件（4）和（2），两种状态中的观察应相同。由于它不会更新状态，除了符号位置之外，目标状态也将保持兼容。

**归纳步骤** 假设对于长度为 $m$ 或更短的所有执行 $q_1 \xrightarrow{o}_* q_1'$，结论均成立。我们需要证明该引理对于长度为 $m + 1$的执行成立。我们对$a$进行分类讨论。过程内语句的证明与 [7]中相应语句的证明相同。我们仅证明过程间语句。

(i) 设$a$是一次方法调用 $r.m'(w)$。规则 Call 中的不变量 $\varphi$-Call 成立，否则这些状态早已被合成守卫排除。因此，根据 M-Call 规则，会创建并初始化一个新的调用上下文。初始化包括执行一系列非干扰赋值。我们可以证明，根据引理 D.3，分别应用每项赋值后得到的两个状态仍保持关系。由于初始化步骤中的赋值互不干扰，因此很容易证明，所有赋值整体带来的更新会保持这种关系。

这一结论是通过对栈帧初始化步骤的最终状态应用归纳步骤假设得出的。

(ii) 设 $a$ 是一个输出语句$\texttt{output}\iota(y)$。由于这些状态并未被安全守卫排除，这意味着规则 $\varphi$-Sink 中的不变量 Sink 在 $q_i, i \in \{1,2\}$中成立。然而，根据 $\xrightarrow{o}_*$的定义，观察应当在此次执行的最后一次状态转换中产生。由于这次状态转换并非最后一次，这意味着 $a$ 无法产生任何观察，即不变量 $\varphi$-Sink 不成立，因此 $a$ 不可能是输出语句。

(iii) 设 $a$ 为返回 [x]。设 $q_1 \to t_1$ 且 $q_2 \to t_2$。返回变量 *ret-var* 在两种状态下都更新为 $x$，即新状态仍保持关系。

(iv) 设$a$ 为一个退出点，即，

$$q_i = (\sqrt{}, \mathtt{Prm}_m, \mathbf{h}_m) :: ([a =]r.m'(w), \mathtt{Prm}_{m'}, \mathbf{h}_{m'}) :: \eta$$

对于 $i \in \{1,2\}$。根据 $q_1 \beta q_2$ 以及定义 D.1 中的条件 (6)，可知两个状态中的栈相同，即 $q_1(\eta) = q_2(\eta)$。因此，结合这一点以及栈顶帧会从栈中弹出且不会对其作任何进一步更改这一事实，我们可以得出 $q_1'(\eta) = q_2'(\eta)$ 也成立 (I)。

根据规则 m-Exit，该规则会更新引用参数 $w^{-1}(p)$ 和可能的赋值目标$a$。调用方法的类型环境也会根据被调用方法的 Effect 进行升级。这意味着将执行一系列非干扰赋值。与引理 D.3 的证明类似，我们可以证明，对于这对 $(q_1', q_2')$ (II)，式 5 和定义 D.1 中的条件 (1)-(5) 均成立。

因此，由 (I) 和 (II) 可得$q_1' \beta q_2'$。

□

引理 D.3（赋值保持关系）。设 $q_1 \beta q_2$，其中 $q_i(\ell) = a$；$\mathbb{S}, i \in \{1, 2\}$，且 $a$ 是一个赋值。如果 $q_1 \to q_1'$ 且 $q_2 \to q_2'$，那么 $q_1' \beta q_2'$。

证明。见 [7]。 □