自然化静态程序分析

Mohammad Mehdi Pourhashem Kallehbasti 马赞达兰科学技术大学电气与计算机工程系 邮政信箱 48518-78195，贝赫沙赫尔，伊朗 pourhashem@mazust.ac.ir

Mohammad Ghafari 克劳斯塔尔工业大学，德国 mohammad.ghafari@tu-clausthal.de

arXiv:2301.04862v1 [cs.PL] 12 Jan 2023

摘要—静态程序分析的开发并非易事，且耗时较长。我们提出了一个框架，开发者可以借助该框架用自然语言定义静态程序分析。我们展示了该框架在识别 Java 程序中的密码学误用方面的应用，并探讨了该框架如何帮助开发者开展静态程序分析开发。

索引词—静态程序分析、密码学、自然语言编程

I. 引言

静态程序分析是一门无需执行代码即可检查程序的艺术。然而，静态分析工具会产生误报，调整这些工具需要专业知识。同样，开发程序分析工具也需要深入了解编译器，或精通某种分析框架。

终端用户编程是一套技术，使终端用户能够按照适合自身实践、背景和技能的复杂程度编写程序。例如，它包括通过可视化代码块为机器人编程的可视化语言 [1]，，以及将英语句子转换为 Bash 命令的简化编程语言 [2]。我们认为，终端用户编程技术也有助于向非专业程序员隐藏编写静态程序分析任务的复杂性，并赋能他们在这一领域开展工作。

我们介绍 NASRA（自然语言静态程序分析），这是一个框架，使开发者能够用自然语言（NL）定义程序分析任务，并生成相应的查询语言（QL）查询，供 CodeQL 程序分析引擎使用。[1] 我们展示了如何将该框架用于查找 Java 程序中的密码学误用。NASRA 是开源的，可供公众使用。[2]

NASRA 的最终目标是实现“自然化”的静态程序分析开发，让开发者无需深入了解静态程序分析及特定分析框架的工作原理，就能指定自己的需求。与现有静态分析框架相比，它具有更高层次的抽象，这或许能让程序分析任务的表述更加直观。同样，它不依赖于特定编程语言的特性，因此能够为程序分析提供跨语言接口，无需学习某个程序分析框架的具体细节。本文迈出了实现上述目标的初步一步。

II. NASRA 框架

密码学是安全领域不可或缺的一部分，但它也是一个众所周知的难点，令开发者颇为头疼 [3]，[4]。定位在 Java 程序中对 `init` 方法的调用，该调用作用于 密码 对象，通常被视为分析 Java 程序中密码学代码的第一步。例如，在 CodeQL 中，需要编写如下查询来完成此任务。

```
来自 方法访问init其中 init.getMethod().getName()
 = “init” 且init.getReceiverType().getName() = “密
码”选择 init
```

我们开发了一个名为 NASRA 的框架，可通过以下形式更直观地表述上述任务：

```
一个 密码对象调用 init。
```

NASRA 是一种规则驱动的合成器。由于在这一领域缺少数据驱动方法所需的可信标注示例，我们依赖预定义规则。NASRA 接收以自然语言编写的程序分析查询，进行语义解析并生成 CodeQL 命令。输入查询应符合 Attempto受控英语（ACE）受控自然语言语法的一个子集。我们使用 Attempto 解析引擎（APE）这一工具，它接收一系列 ACE语句，并生成相应的话语表征结构（DRS），作为输入文本的语义表示。NASRA 将本节后文介绍的转换规则应用于给定的 DRS，并生成相应的 CodeQL语句。得益于 APE，NASRA语句的表述方式非常灵活，无需与 NASRA 语法完全对应。我们选择 CodeQL 作为代码分析引擎，因为它是业界领先且由社区驱动的工具，并且所有 GitHub 用户均可公开使用，无需繁琐的安装步骤。若要将 NASRA 用于新的静态分析框架，只需调整转换规则。若要支持新的应用领域，则应识别当前语法不支持的查询类型，将相应的产生式规则添加到语法中，并为其开发转换。NASRA 是开源的，目前支持针对 Java 程序中密码学误用的程序分析任务。

[1]https://codeql.github.com[2]

https://doi.org/10.5281/zenodo.7495044

A. 语法与语义

每个 NASRA 查询由一个或多个语句组成。其语法如下所示（终结符以不同颜色显示）。

$Query$ ::= $Statement\ Query$ | $StatementStatement$ ::= $BasicStatement$ | $LogicalStatement$ | $Extension$
$BasicStatement$::= Exp 是（Exp | 属于 $List$）Exp ::= $Prefix\ Exp$ | 类型 | ID | $LiteralPrefix$ ::= （（$adjective$|ε） $attribute$ 的）$LogicalStatement$ ::= $Statement$ 且 $Statement$ |$Statement$ 或 $Statement$ | 并非如此： $Statement$| 如果 $Statement$ 则 $Statement$

a) 表达式： 最小的构建单元是 表达式。它包括字面量（字符串或整数）或标识符（用户定义标识符），这些内容可直接映射为 CodeQL 表达式。表达式也可以是 CodeQL 类型，例如类、变量和方法访问，它们分别映射为 类、变量和 方法访问。

b) 前缀： 每个表达式都可以有一个可选的 前缀 ，形式为 “属性的” ，用于指明该表达式的属性。例如，名称、类型、参数和方法都是某个实体（即 表达式）的属性，在 CodeQL 中分别对应于 `getName()`、`getType()`、`getArgument()`和 `getMethod()` 方法。

例如，“`method1`的名称”是一个表达式，其中“名称”是一个属性，而`method1`是一个标识符。整个表达`method1.getName()`在 CodeQL 中表示为。

此外，属性本身可以带有可选的序数作为形容词，例如第二个，在“`init`的第二个参数”这个表达式中，它被翻译为“`init.getArgument(1)`”，其中根据从零开始编号，second 被翻译为参数序号 1。

请注意，“属性的”可以重复多次，每个属性都可以带有形容词。例如，“`init`的第二个参数的类型”这个表达式包含一个标识符（即`init`）和两个属性（即类型和参数）。

c) 基本语句：每个基本语句都是一个既可以充当布尔条件、也可以充当假设的语句。

作为一个布尔条件，基本语句可以表示两个表达式之间的等价关系，也可以表示一个表达式属于某个列表。在“表达式是表达式”结构中，等价关系两边都是表达式，且二者必须相等；而在“表达式属于列表”结构中，表达式必须等于列表中的某一项。因此，像“`arg1` 属于 [`"RSA"`、`"AES"`]。”这样的语句是一个析取表达式，可以改写为“`arg1` 是`"RSA"` 或者`arg1`是`"AES"`。”，最终翻译为“`arg1 = "RSA"`或者`arg1 ="AES"`”。

语法结构 Exp 是 Exp 还可以生成假设，当第二个 Exp 是 CodeQL 类型时。这些假设会映射到 CodeQL 查询的 `from` 部分。例如，语句“`var1` 是一个变量。”会转换为“变量 `var1`”，并属于 `from` 部分。

d）逻辑语句：一个逻辑语句可以是否定、合取、析取或蕴含。例如，“如果`arg1` 是`"RSA"`，那么`arg2` 是`"AES"`。”可翻译为

“否定`(arg1 = "RSA")` 或`arg2 = "AES"`”在 Cod- eQL 中，因为 $p \Rightarrow q$ 等价于 $\neg p \vee q$。

B. 可扩展性

可以扩展 NASRA，以涵盖辅助语句和语句模式。相应的产生式规则如下。

$Extension$ ::= $Pattern$ | $AuxiliaryStatement$

我们通过三种语句模式和一条辅助语句来引入这些特性，它们有助于涵盖使用 Java 密码学对象时的约束。

1) 模式：我们介绍三种模式，用于扩展语法中的 Pattern 非终结符。下面将逐一讨论。

a) 调用：我们使用此模式来说明某个方法由特定类的一个实例调用。它也可用于确保特定类的任何实例都不会调用某个方法。

$Pattern_1$ ::= 一个 ID (调用|不调用) ID.

第 II 节中的 NASRA 查询就是这种模式的一个示例。转换字符串分为多个步骤。首先，声明一个名称与 NASRA 语句中所用名称相同的方法访问（即`init`）。然后需要在 `where` 部分添加条件。具体而言，方法访问 `init` 的方法名应为第二行中所述的`"init"` 。最后，方法访问具有一个接收者，即调用其方法的对象。在本例中，接收者的类型名称应为"密码"，如第三行的 CodeQL 代码所示。

如果需要确保没有调用发生，则必须使用存在量词，如下所示。

```
fromwhere not (exists (方法访问 init |
init.getMethod().getName() = "init" and
init.getReceiverType().getName() = "密码"))
```

这意味着不存在满足这些条件的 方法访问`init` 。我们可以用下面的形式在 NASRA 中表述这一点。

一个 密码对象不会调用 `init`。

b) 偏序约束：此模式使我们能够对方法调用施加偏序约束。换言之，我们可以强制要求一个方法调用先于（或后于）另一个方法调用。

$Pattern_2$::=方法名 (先于|后于) 方法名。

例如，在 CodeQL 中，要声明“调用 `getInstance` 早于对`init`的调用”，需要分两步进行。首先，应指定这两个方法位于同一作用域中。接下来，前一个方法调用的行号必须小于另一个方法调用的行号。如下所示。

```
getInstance.getEnclosingCallable()                    =
init.getEnclosingCallable() 和
getInstance.getLocation().getEndLine()                <
init.getLocation().getEndLine()
```

我们可以在 NASRA 中将此查询表示如下。

```
getInstance位于 init之前。
```

c) 方法签名约束：可以使用 Pattern_3 表示方法的签名。

```
Pattern3 ::= 方法名的签名为 List.
```

方法签名可以看作一个有序的数据类型列表。此列表包含以字符串形式表示的数据类型名称，第一个字符串是第一个参数的数据类型名称，以此类推。例如，以下 NASRA 查询表明 `getInstance` 方法有两个参数，它们的类型名称分别是 `"int"` 和 `"Certificate"`。

```
getInstance 的签名是 ["int", "Certificate"]。
```

此查询被转换为以下 CodeQL 查询。

```
(count (getInstance.getAnArgument()) = 2) 并且
getInstance.getArgument(0).getType().toString()="int"
并且 getInstance.getArgument(1).getType().toString()="
Certificate"
```

首先，将参数数量设为用户定义列表的长度，然后逐一约束参数的类型。`count (method.getAnArgument())` 返回该方法的参数数量。`getArgument(i)` 返回给定方法中的第 i 个参数，`getType()` 返回给定参数的类型，最后 `toString()` 将给定的数据类型转换为字符串。

2) 辅助语句：我们的目标是找出代码中违反一项或多项强制约束的误用。例如，假设 `init` 方法的第二个参数为“私钥”时，加密算法必须为，即getInstance 方法的第二个参数必须为“RSA”；此外，如果加密算法为“AES”，那么加密模式必须为，即getInstance 方法的第一个参数必须为“CBC”。以下 NASRA 查询将找出此类违规。

```
以下情况不成立：如果init的第二个参数类型为“PrivateKey”，
那么getInstance第一个参数的算法为“RSA”；或者以下情况
不成立：如果getInstance第一个参数的算法为“AES”，那么
getInstance第一个参数的加密模式为“CBC”。
```

为了找出违反这些约束的情况，必须在查询中写明对这些约束的否定的析取。[3]

[3]例如，在“X 正在城市道路($Cond_1$)上驾驶。X 必须以低于 60 km/h 的速度驾驶($Cons_1$)。X 必须系好安全带($Cons_2$)。”中，查询需要找到一个 X：它正在城市道路上驾驶，并且车速超过 60 km/h，或者没有系安全带。如果像语句中所述的那样，为每条语句分配一个布尔变量，那么目标应是 $Cond_1 \wedge (\neg Cons_1 \vee \neg Cons_2)$，其必要性部分被转换为两个约束的否定的析取。

然而，随着约束数量增加，上述语句会变得更长，也更难理解。

我们定义了辅助语句，以便开发者更轻松地编写和理解复杂查询。具体而言，必要性语句是一种辅助语句，可使开发者用简短且独立的语句强制执行强制约束。它以“必须满足以下条件”开头，并遵循以下语法。

```
NecessityStatement                                    ::=
必须满足 Statement.
```

因此，开发者无需在一条语句中写出所有约束的否定的析取，而可以使用此结构（即 NecessiyStatement），在查询中用多条语句定义所有此类约束。相应地，之前那条冗长的语句可以改写为下面所示的两条独立语句。

```
必须满足：如果 init 的第二个参数类型是“私钥”，那么
getInstance的第一个参数中的算法是“RSA”。必须满足：
如果 getInstance的第一个参数中的算法是“AES”，那么
getInstance的第一个参数中的模式是“CBC”。
```

必要性语句的处理方式不同于其他语句。如果只有一个 NecessityStatement，其外围语句会被否定，并添加到 CodeQL 查询的 `where` 部分。如果有多个，例如 n 约束 $Cons_1$、$Cons_2$、.....、$Cons_n$，那么“(非 $TCons_1$ 或非 $TCons_2$ 或.....或非 $TCons_n$)”将被添加到 `where` 部分，其中 $TCons_i$ 是 $Cons_i$ 的翻译。

III. 示例

Cipher 是 Java 密码学中最常被误用的 API 之一 [4]。列表 1 展示了如何在 Java 中创建一个 `Cipher` 对象。我们应调用 `Cipher` 的 `getInstance` 方法。此方法接收多个参数。第一个参数是 转换字符串，这是一个由“/”分隔为三部分的字符串。这三部分依次是 算法、模式和 填充。接下来，我们应对 密码 对象调用 init 方法，并传入两个参数，以指定 密码 的操作模式，并使用密钥或证书初始化此对象。

```
密码 cipher = 密码.getInstance("AES/ECB/
        PKCS5Padding");
cipher.init(密码.ENCRYPT_模式,new SecretKeySpec(
        keyBytes, "AES"));
```

列表 1. 在 Java 中设置 Cipher 对象

在本节的其余部分，我们将介绍三种不同的程序分析任务，以确保安全地使用 Java 密码。

A. 密钥与算法

任务 1：如果密钥的类型为公钥、私钥或证书，或者加密模式为包装模式或解包模式，则转换字符串中的算法必须为“RSA”。

列表 2 展示了如何在 CodeQL 中检查这一约束。

```
来自 MethodAccessgetInstance、MethodAccess init，其中
```

列表 2. CodeQL 中的密钥与算法约束

这一约束可在 NASRA 中表示如下。

> 一个 密码 调用 init。一个 密码 调用 getInstance。必须满足：如果init的第一个参数属于 ["Cipher.WRAP_MODE"、"Cipher.UNWRAP_MODE"]，或者 init 的第二个参数类型属于 ["PublicKey"、"PrivateKey"、"Certificate"]，那么getInstance的第一个参数所指定的算法为"RSA"。

B. 算法与转换模式

任务 2：如果转换的算法是“RSA”，那么转换模式必须是“”或“ECB”。

列表 3 展示了用于在 CodeQL 中检查此约束的相应查询。我们应查找算法为“RSA”，但模式既未设置为“ECB”也未设置为空字符串的代码。

```
来自 方法访问getInstance，其中
```

列表 3。CodeQL 中的算法与转换字符串模式约束

此约束可在 NASRA 中表示如下。

> 一个 密码 对象会调用 getInstance。必须满足以下条件：如果 getInstance的第一个参数中的算法是“RSA”，那么getInstance的第一个参数中的模式属于[“”、“ECB”]。

得益于 Attempto 解析引擎（APE），NASRA 陈述不必严格遵循语法规则，因此释义时可以有一定的灵活性。例如，“getInstance 的第一个参数所对应的算法是 "RSA"”这一部分还可以用以下两种形式表达：

> （i）第一个参数的算法getInstance是“RSA”。（ii）“RSA”是getInstance的第一个参数的算法。

C. 转换与加密模式与签名的比较

任务 3：如果转换模式是“CBC”、“PCBC”、“CTR”、“CTS”、“CFB”或“OFB”之一，且加密模式不是“Cipher.ENCRYPT MODE”，那么调用的 init 方法不应具有以下任何一种签名：init(encmode, cert)、init(encmode, cert, ranGen)、init(encmode, key)、init(encmode, key, ranGen)。

清单 4 展示了如何在 CodeQL 中强制执行此约束。

```
from 方法访问 getInstance, 方法访问init where
```

清单 4. CodeQL 中转换与加密模式与签名约束的对比

以下展示了 NASRA 中该任务的实现。

> 一个 密码对象会调用 getInstance。一个 密码对象会调用 init。必须满足以下条件：如果 getInstance的第一个参数的模式属于["CBC","PCBC","CTR","CTS","CFB","OFB"] 且init的第一个参数不是 "Cipher.ENCRYPT_MODE"，那么 getInstance的签名不是["int","Certificate"]，也不是["int","Certificate","SecureRandom"]，也不是["int","Key"]，也不是["int","Key","SecureRandom"]。

D. 讨论

表 I 列出了每项分析任务所需的不同运算符和操作数的数量（即词汇量），以及运算符和操作数的总数（即长度）。[4]

[4]在 NASRA 中，我们将 init、“RSA”和 getInstance 等用户定义的终端视为操作数，并将其余语言构造计为运算符。

表 I：CodeQL 与 NASRA 对比

分析任务	词汇量		长度	
	CodeQL	NASRA	CodeQL	NASRA
密钥与算法（III-A）	32	19	179	39
算法与模式（III-B）	27	18	107	24
模式与签名（III-C）	42	26	434	56

显然，NASRA 中的查询明显短于 CodeQL 中的查询（即长度最多减少 87%），而且使用的编程构造也少得多（即词汇量最多减少 38%）。我们计算了 Halstead 复杂度度量，以估算编写这些查询所需的时间，以及编写或理解这些查询的难度 [5]。结果表明，与在 CodeQL 中完成这些任务相比，开发人员在 NASRA 中所需的精力和时间要少得多。

我们还邀请了十位开发者分享他们对 NASRA 查询的看法。他们一致认为这些查询简洁易懂，其中一人评论道："这些查询读起来就像 API 文档。"

值得注意的是，NASRA 的性能，即其检测 API 误用的能力，取决于其底层分析框架，而目前该框架是 CodeQL。换言之，NASRA 消除了定义静态程序分析所需的底层细节，但误报问题仍然值得关注。此外，尽管 NASRA 具有自然易用的特点，使用它仍然需要了解其语法。

IV. 相关工作

将自然语言陈述映射为形式化表示一直受到该领域的广泛关注，但在程序分析开发领域却鲜有关注。

Schlegel 等人为 Python 开发了一种终端用户编程范式，可将自然语言命令映射为 Python 代码 [6]。Landhauber 等人提出了一种领域无关命令解释器，它接收英文自然语言命令，并利用本体生成相关的 API 调用 [7]。Yaghmazadeh 等人开发了 SQLIZER，这是一种可从自然语言自动合成 SQL 查询的系统 [8]。Luo 等人研究了将自然语言查询转换为可视化的方法，旨在简化数据可视化的创建过程 [9]。

Heyman 等人开发了一种 Python 代码补全工具，通过添加对预期数据科学任务的自然语言描述来丰富开发者的代码 [10]。Nguyen 等人提出了一种方法，以编程任务的英文描述为输入，并合成该任务对应的 API 代码模板 [11]。Desai 等人构建了一个通用框架，用于构建程序合成器，这些合成器以自然语言为输入，并生成目标领域特定语言中的表达式 [12]。Zhai 等人提出了一种基于搜索的技术，可自动将自然语言注释转换为形式化程序规范，以描述预期的前置条件和后置条件 [13]。

本文介绍的工作也与密码学领域相关。目前已有用于发现密码学误用的工具 [14]，以及便于开发者采用密码学的库 [15]。然而，这些工具和库都没有采用自然语言方法。

V. 结论

我们介绍了 NASRA，这是一个用于以自然语言定义静态程序分析的开源框架。我们展示了如何应用该框架来发现 Java 密码学中的误用。NASRA 的最终目标是实现一种自然的静态程序分析开发方式，使主流开发者也能使用。要实现这一目标，还需要进一步研究，以确定 NASRA 在真实环境中的有效性。此外，还需要研究其查询的表达能力，以及将其扩展到其他问题领域所需的工作量。最后，无需预定义规则的自动翻译也是一个令人期待的未来研究方向。

参考文献

[1] E. Barakova, J. Gillesen, B. Huskens 和 T. Lourens，《终端用户编程架构促进机器人在社会疗法中的应用》，《机器人与自主系统》，第 61 卷第 7 期，2013 年。[2] X. V. Lin, C. Wang, L. Zettlemoyer 和 M. D. Ernst，《NL2Bash：面向 Linux 操作系统自然语言接口的语料库与语义解析器》，载于第十一届国际语言资源与评测会议论文集（LREC 2018），2018 年。[3] M. Hazhirpasand, O. Nierstrasz, M. Shabani 和 M. Ghafari，《开发者在密码学方面面临的障碍》，载于 2021 年 IEEE 国际软件维护与演化会议（ICSME），2021 年，第 659–663 页。[4] M. Hazhirpasand, M. Ghafari 和 O. Nierstrasz，《实际应用中的 Java 密码学用法》，载于第 14 届 ACM / IEEE 国际实证软件工程与度量研讨会论文集，2020 年。[5] M. H. Halstead，《软件科学要素》（操作与程序设计系统丛书）。Elsevier Science Inc.，1977 年。[6] V. Schlegel, B. Lang, S. Handschuh 和 A. Freitas，《Vajra：通过自然语言逐步编程》，载于第 24 届智能用户界面国际会议论文集，2019 年。[7] M. Landh¨auber, S. Weigelt 和 W. F. Tichy，《NLCI：自然语言命令解释器》，《自动化软件工程》，第 24 卷第 4 期，第 839–861 页，2017 年 12 月。[8] N. Yaghmazadeh, Y. Wang, I. Dillig 和 T. Dillig，《SQLIZER：从自然语言合成查询》，《ACM 程序设计语言会议论文集》，第 1 卷，OOPSLA 专刊，2017 年 10 月。[9] Y. Luo, N. Tang, G. Li, J. Tang, C. Chai 和 X. Qin，《通过神经机器翻译将自然语言转换为可视化》，《IEEE 可视化与计算机图形学汇刊》，第 28 卷第 1 期，第 217–226 页，2022 年。[10] G. Heyman, R. Huysegems, P. Justen 和 T. Van Cutsem，《自然语言引导编程》，Onward! 系列，2021 年，第 39–55 页。[11] A. T. Nguyen, P. C. Rigby, T. Nguyen, D. Palani, M. Karanfil 和 T. N. Nguyen，《将英语文本统计翻译为 API 代码模板》，载于 2018 年 IEEE 国际软件维护与演化会议（ICSME），2018 年，第 194–205 页。[12] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R 和 S. Roy，《使用自然语言进行程序合成》，载于第 38 届国际软件工程会议论文集，ICSE '16 系列，2016 年。[13] J. Zhai, Y. Shi, M. Pan, G. Zhou, Y. Liu, C. Fang, S. Ma, L. Tan 和 X. Zhang，《C2S：将自然语言注释转换为形式化程序规范》，载于第 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论文集，ESEC/FSE 2020 系列，2020 年。[14] Y. Zhang, M. M. A. Kabir, Y. Xiao, D. D. Yao 和 N. Meng，《Java 密码学 API 误用的自动检测：我们已经做到位了吗？》，《IEEE 软件工程汇刊》，2022 年。[15] S. Kafader 和 M. Ghafari，《Fluentcrypto：轻松模式下的密码学》，载于 2021 年 IEEE 国际软件维护与演化会议（ICSME），2021 年，第 402–412 页。