Patch2QL：使用自动生成的静态分析规则在开源软件供应链中发现同源缺陷

Fuwei Wang,* Yongzhi Liu, Zhiqiang Dong
腾讯云鼎实验室

摘要

在开源软件（OSS）生态系统中，存在一个复杂的软件供应链，开发者上下游广泛借用和重用代码。 这导致了反复出现的缺陷、遗漏的修复和传播问题。 这些被统称为同源缺陷，它们的规模和威胁并没有得到广泛关注和系统研究。 软件组合分析和代码克隆检测方法无法覆盖供应链场景中的各种变体问题，而代码静态分析或静态应用安全测试（SAST）技术则难以针对特定缺陷。

在本文中，我们提出了一种新颖的技术，通过自动生成SAST规则来检测OSS中的同源缺陷。 具体而言，它通过结构比较和控制流到数据流分析从补丁前后版本的代码中提取关键语法和语义信息，并生成与这些关键元素匹配的规则。

我们已经实现了一个名为 `Patch2QL`的原型工具，并将其应用于C/C++中的基本OSS。在实验中，我们发现了7个新的中度到严重的漏洞，以及大量潜在的安全问题。在分析供应链中的下游项目时，我们发现了大量代表性的同源缺陷，从而阐明了这个问题所带来的威胁。 此外，与通用的SAST和基于签名的机制相比，生成的规则在发现所有同源缺陷的方面表现更好。

1 引言

开源软件（OSS）是整个互联网生态系统的基石。 来自上游开发者的开源代码以各种形式影响着下游生态系统，例如被调用、集成、复制和部分重用，形成了一个复杂的软件供应链。

*通讯作者。 电子邮件：fullwaywang@outlook.com

chain. 根据Synopsys [29]的最新报告，17个行业中96%的扫描代码库发现包含开源组件。 随着广泛使用和审查，OSS生态系统中出现了越来越多的新漏洞和潜在的代码错误，可能会带来安全问题，我们统称为缺陷。

软件缺陷问题进一步引发了软件供应链环境中的新问题。 其中之一是同源缺陷的出现，伴随着软件供应链中的各种实践，如开源代码重用、具有类似功能的代码复制、项目克隆和二次开发、碎片化的软件分发和维护等。发现同源缺陷的目标可以分为几个任务：a）对于上游开源软件，识别与原始缺陷相似的重复缺陷，表明存在相同的编码错误；b）对于基于开源软件的分叉

项目，确认上游缺陷是否存在并影响当前代码库，以及确定是否有从原始代码库借用代码的新添加模块包含派生缺陷；c）对于任何项目，验证其是否包含与已知开源软件中的缺陷相似的缺陷，无论是在语法还是语义上。

然而，发现同源缺陷并不像看起来那么简单。 软件组件分析（SCA）机制的有效性在很大程度上依赖于显式依赖分析，而这在C/C++生态系统中不适用。

在研究领域中，发现同源缺陷被归类为代码克隆检测问题，并且近年来相关研究工作数量庞大。

主流思路是使用定制的代码指纹或签名来表示缺陷特征，并设计相应的模糊匹配算法来识别具有文本或语义相似性的目标代码。 然而，正如我们将在下面讨论的那样，语法相似性只能揭示简单相似缺陷的一小部分，并且基于相似性的方法在计算能力需求上受到极大限制。 静态应用安全测试（SAST）被广泛用于匹配特定的代码模式，有效地识别

具有相同根本原因的缺陷。 然而，不应忽视常见漏洞和曝光（CVE）与可用的SAST规则之间的显著差距，因为大多数SAST工具专注于检测常见类型的弱点，而真实世界中的大部分漏洞是由常见原则和特定逻辑的组合导致的。

我们的方法在本文中，我们提出了一种在**O**SS中发现同源缺陷的方法。其基本概念是从修补前后的代码差异中推断出缺陷的根本原因。 主要思想是通过分析缺陷的修补程序来理解其根本原因，然后自动生成一个SAST规则，而不是某种签名，该规则可以特别匹配未修补的目标代码及其控制流和数据流上下文。 这种能力使我们能够识别在语法或语义上相似的代码。

首先，我们收集历史漏洞和相应的修补程序。 我们检索未修补和修补后代码的抽象语法树（AST），并考虑到层次结构化的信息进行比较。

其次，我们过滤和重新组织所有不同的节点，将它们转换为原始的SAST规则，该规则匹配所有存在未修补节点及其上下文的代码中的出现次数，而在补丁中引入的新节点则不匹配。然后，我们的系统通过遍历这些节点周围的两个AST共享部分来建立不同AST节点的控制流和数据流上下文。 在控制流上下文方面，我们确定结构上相关的节点，指示不同节点的位置。 对于数据流上下文，我们定位在不同节点中引用的重要数据变量，并在其周围执行简单的数据流分析。 最后，通过回归测试，我们通过简化和丰富查询和条件集来完善规则，以有效匹配未修补的代码。

原型和评估我们开发了一个针对C/C++中的OSS的原型工具**Pat**ch2QL。对于底层的SAST技术，我们选择了有前景的工具C**ode**QL [13]及其类似查询的语言QL [10]。 我们从G**oogle** OSS-F**u**zz项目的列表[17]中选择了111个C语言项目作为目标，并构建了一个生成的QL规则库。 我们对8个最流行的项目进行了规则的有效性评估，结果显示这些规则的召回率可以达到71%，检测准确性明显高于其他SAST工具和基于签名的代码克隆检测方法。

我们对OSS生态系统进行了调查，以揭示OSS供应链中同源缺陷的普遍性和严重性。 我们将我们的规则应用于两种类型的目标：对于上游OSS项目，我们已经确定了与现有CVE相关的7个新的循环出现的CVE。 发现的其他问题已经报告并得到修复。 至于下游OSS项目，我们分析了流行的开源数据库项目（包括分支或疑似借用代码的项目）由云计算供应商主要开发和维护的上游OSS和三个Linux发行版内核。
我们在这些项目中检测到一批已知的缺陷或其变体，其中一些与其源代码几乎没有上下文相似性。

总之，本文提出了以下贡献。

- 我们对不同级别和原因的各种同源缺陷类型进行了全面研究，并通过发现相应的新漏洞来解释问题的当前状态。

- 我们创新地使用自动生成的SAST规则作为特征来检测同源缺陷，与通用的SAST规则集和基于自定义签名的代码克隆检测机制相比，提供了独特的检测能力。

- 我们已经开发并实现了一个专门针对C语言的原型。在不到两个月的时间里，我们将这个原型应用于分析几个基本的开源软件项目。 在这段时间内，我们成功地发现并报告了7个零日漏洞，以及其他许多问题。

本文的剩余部分结构如下：我们在第2节介绍背景和定义的威胁模型，并在第3节中说明我们系统的整体设计。 我们在第4节进一步说明了我们系统的关键技术和实现。 然后我们在第5节中呈现评估结果。 最后，我们在第6节讨论了限制，并在第7节总结了我们的工作。

2 问题的范围

2.1 开源软件供应链中的角色

供应商和消费者的角色可以分为
3个层次。 上游供应商是开源软件代码的来源。
开发人员编写和维护源代码，解决问题，并合并个人的贡献。 中游层涉及分叉和重新分发的次级开发人员。
还有一些平台，用于制造开源软件以分发给下游用户。
根据开源软件的使用方式，有三种类型的下游消费者：直接使用开源软件作为二进制文件和SDK的人；将开源软件作为完整组件明确集成到他们的项目中的人；将开源软件源代码的部分（模块、文件或片段）借用到自己的项目中的人。

已经对特定角色周围的几种威胁进行了深入研究和解决。 例如，软件组件分析（SCA）和软件组件清单（SBOM）是识别和记录下游项目导入或链接的完整组件的成熟技术。 然而，同源缺陷很少得到系统性的研究或充分解决，我们将在下文中尝试解释。

2.2 同源缺陷威胁模型

我们将同源缺陷定义为具有相同根本原因的缺陷。 在某些情况下，同源缺陷被称为不同的术语。 通过软件复用传播的错误通常被称为重复错误[15, 25]，这表明已知的和先前修复的错误在其他代码库中重新出现。 有些被称为已知漏洞的变体-性[31]，其中包括部分修复的漏洞，在修补的代码中出现的回归错误，或者在Linux的不同子系统中发现的类似错误。

根据源代码中同源缺陷的形式和相似程度，我们提供了四级分类：

L1除了微小的文本修改之外，易受攻击的代码几乎相同，而相关的代码上下文（文件、模块和函数）可能会有微小的变化，例如在两个接近的版本之间引入的变化，而没有重大的开发。

L2易受攻击的代码保持不变（有时无关的代码可能会插入其中），但上下文发生了重大变化，比如代码重构带来的修改。

L3 只重用与缺陷相关的目标代码片段，代码片段及其上下文可能已被重写或重构；或者缺陷由函数调用主导的特定语句序列模式组成。

L4 缺陷仅在功能逻辑和语义层面上被重用或共享，主要包括在利用相同算法或协议标准的代码实现中存在相同的缺陷。

通过对各种案例进行广泛分析，我们总结了开源软件供应链开发过程中同源缺陷的代表性来源和传播方式：

错误回归 由于缺乏错误跟踪，通过协作或代码重构可能错误地删除了错误修补程序。 由于冲突提交，推送到开发分支的修复提交有时也会被还原。 如果及时发现这些缺陷，它们属于L1-L2类型。

这些情况在[31]中已经广泛讨论过。

碎片化OSS的用户通常采用快照版本，并经常根据特定需求进行定制开发。 与上游开发分支相比，分歧随着时间的推移而增加，使得保持同步变得具有挑战性。 这种情况使得移植补丁变得困难。 这些未修复的缺陷属于L2类型。 我们在第5.3.2节中详细讨论的示例最能说明这种情况。

Copycat 对于协作或分叉项目，后续开发人员可能基于现有代码开发具有类似逻辑的新模块

这样做会导致缺陷在过程中的继承。 这些缺陷属于L2-L3类型。 特别是，在表3中提到的看似错误可以归类为这个类别。

常见弱点在实现类似逻辑时，常见的弱点类型可能导致相似或重复的缺陷。 此外，不清晰的接口文档可能会误导开发人员并导致他们错误使用API。这些大多属于L3-L4类型。 我们发现的与**C**VE-2020-29599同源的2个实例说明了这种情况，如表2所述。

模拟中毒 虽然没有明确讨论，但通过引入一个与先前易受攻击的代码片段相似的精心制作的代码片段，对OSS进行中毒是现实的。对于维护者来说，很难确定一个稍后出现的提交是否是先前漏洞的模仿。 这些罕见情况可能属于L4类型。

2.3 C/C++中的OSS负担

考虑上述分析，同源缺陷在开发语言生态系统中更为突出，这些生态系统具有更长的历史，其OSS供应链更加复杂和不透明。 较新的编程语言，如Golang，对基本库/包有更集中的支持，并且尚未广泛采用代码复制实践。 在像Java这样具有集中式包管理（如Maven）和具有权威性基金会（如Apache）提供大量常用组件的生态系统中，组合依赖关系相对清晰。 然而，在C/C++的情况下，由于具有悠久而基础的历史，存在许多具有类似功能的并存项目，并且缺乏通用的包管理机制，同源缺陷问题已经成为一个巨大的负担。

C/C++在基础系统和应用程序中的主导地位仍然不可动摇，并且它所面临的问题模型可能在其他生态系统中存在较小程度的存在或在不久的将来存在。 因此，我们的研究重点放在C/C++的核心生态系统上，并最终实现了用于解决C/C++问题的C语言OSS的原型工具。

然而，方法论的方法和原型工具本身可以在进行一定调整后，用于发现其他语言生态系统中的潜在问题。

3 系统概述

3.1 动机

多年来，关于漏洞的根本原因、直接原因和补丁之间的关系已经得到了广泛研究。 以前的研究[32]观察到，对于内存损坏等常见漏洞类型，补丁在输入数据处理方面存在显著差异。

```
diff --git a/crypto/x509/v3_utl.c b/crypto/x509/v3_utl.c
index 4fd1f2cd60..5c63d2d9d8 100644
--- a/crypto/x509/v3_utl.c
+++ b/crypto/x509/v3_utl.c
@@ -529,17 +529,25 @@ static int
 ↪  append_ia5(STACK_OF(OPENSSL_STRING) **sk,
     /* 首先进行一些合理性检查 */
     if (email->type != V_ASN1_IA5STRING)
         return 1;
-    if (!email->data || !email->length)
+    if (email->data == NULL || email->length == 0)
+        return 1;
+    if (memchr(email->data, 0, email->length) != NULL)
         return 1;
     if (*sk == NULL)
         *sk = sk_OPENSSL_STRING_new(sk_strcmp);
     if (*sk == NULL)
         return 0;
+
+    emtmp = OPENSSL_strndup((char *)email->data, email->length);
+    if (emtmp == NULL)
+        return 0;
+
     /* 不要添加重复项 */
-    if (sk_OPENSSL_STRING_find(*sk, (char *)email->data) != -1)
+    if (sk_OPENSSL_STRING_find(*sk, emtmp) != -1) {
+        OPENSSL_free(emtmp);
         return 1;
-    emtmp = OPENSSL_strdup((char *)email->data);
-    if (emtmp == NULL || !sk_OPENSSL_STRING_push(*sk, emtmp)) {
+    }
+    if (!sk_OPENSSL_STRING_push(*sk, emtmp)) {
         OPENSSL_free(emtmp); /* 在推送失败时释放 */
         X509_email_free(*sk);
         *sk = NULL;
```

图1：OpenSSL中CVE-2021-3712的补丁。

在不同版本之间进行操作，提供有价值的语义信息。

我们工作的动机源于对补丁的理解，补丁以及其控制流和数据流上下文包含了缺陷的根本原因的信息。 通过抽象和关联这些信息，我们可以揭示由语法差异暗示的潜在语义原因。 鉴于软件的复杂性，我们认为无法从理论上枚举和总结具有足够含义的根本原因的补丁类型和部分。 然而，即使对于无法仅通过修复来推断根本原因的漏洞，补丁仍然可以用于识别语法上相似的代码段，这些代码段可能与实际的同源缺陷重叠。

在下面的算法和系统介绍中，我们使用一个运行示例来进行清晰说明。 该示例取自OpenSSL的历史漏洞CVE-2021-3712 [12]。这是一个由于错误地将ASN.1字符串中的缓冲区视为NULL终止而导致的缓冲区溢出漏洞。 修复漏洞函数**append_ia5**的一个补丁是提交编号为986247的，如清单1所示。

3.2 架构

Patch2QL的架构如图1所示。 Patch2QL是一个基于修复安全缺陷的补丁生成SAST规则的工具。 它的主要任务是将易受攻击和修复的函数对之间的各种元素和关系转化为规则。 然后可以使用这些规则来匹配潜在的易受攻击函数。

目前，我们使用CodeQL作为底层的SAST工具和规则格式绑定。 作为一种类似SQL的查询语言，CodeQL具有很高的表达能力，并且对开发人员友好。
构建、选择和组织带有条件的查询是创建可靠规则的基础。 然而，Patch2QL的核心技术不仅限于CodeQL。

如果提供了另一个SAST工具及其规则格式，生成其特定绑定的规则就很简单。
当给定一个项目和一个安全补丁时，Patch2QL遵循五个主要步骤生成规则，具体如下。

3.2.1 处理补丁并定位函数对

漏洞的表示是在函数级别上进行的。 为了捕捉函数在应用补丁之前和之后在基本语法单元方面的差异，代码库需要转换为抽象语法树（AST）。 此外，补丁通常包含一个详细列出对易受攻击代码库进行的修改的文本列表。 通过遍历和比较修补源代码文件中所有函数的AST，我们可以定位到被修改的精确函数。

此外，有时需要对补丁进行预处理，然后再进一步处理。 一些开源软件的主要维护者经常提交大量代码提交，以解决多个错误。 在其他情况下，一个补丁可能会修复多个相同根本原因的实例。 在这种情况下，我们提取修复每个单独错误所需的最小更改。目前，这主要通过简单的文本冗余分析来实现。 它涉及分析补丁是否包含多个重复修复模式的实例，并将其拆分为单独的补丁。 例如，如果一个补丁修复了多个相同的API误用问题，每个调用点的修复将被分开。 此外，某些漏洞可能需要多个提交才能完全解决，因此需要合并这些更改以生成完整的补丁。

3.2.2 在AST之间进行结构比较

为了获得预补丁和后补丁函数的AST之间的准确差异列表，需要进行结构比较。 在我们的实现中，AST是由CodeQL通过编译生成的。 表达式、语句、字面量、函数调用被命名为AST节点。 取行`@@ -539,2 +546,4 -`的if-then分支的子树

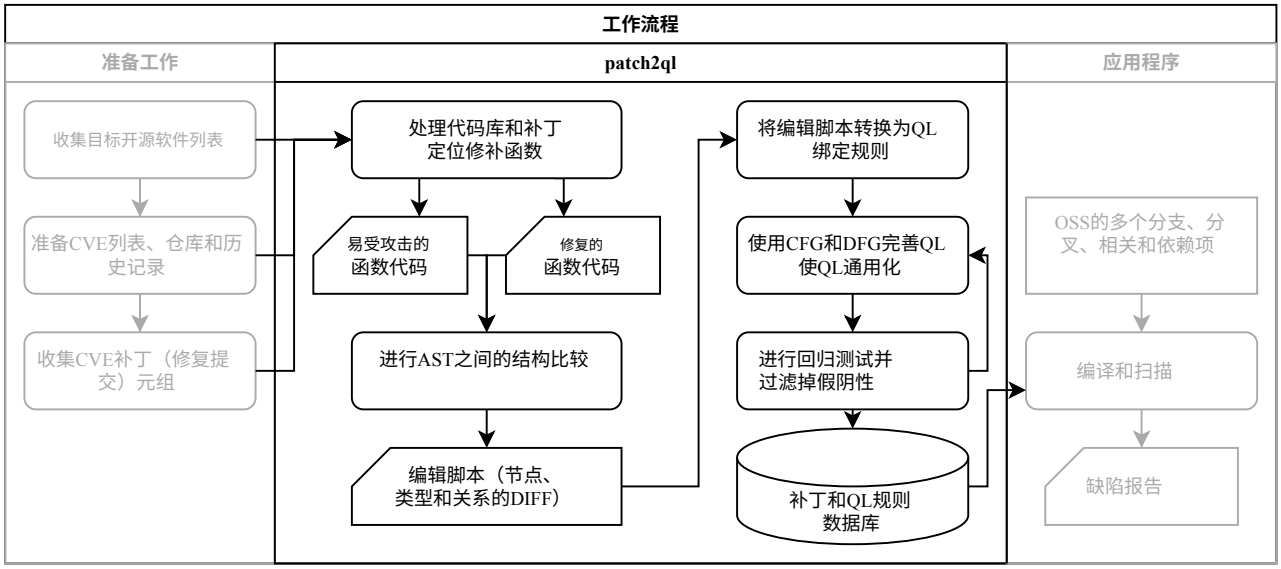

图1：Patch2QL的核心操作、先决条件和应用的示意图。

以列表1中显示的补丁为例，差异似乎是替换了函数调用的第二个参数节点。 AST的示例可以参考附录中的图2。

使用EditScript算法[6]来完成AST的比较任务，该算法被广泛用于比较结构化信息。 具体而言，我们提出了一种在算法中寻找两个AST之间匹配的定制方法，该方法在第4.1节中有详细解释。然后，我们获得一个编辑脚本，揭示了将预修补的AST转换为后修补的AST所需的编辑操作。这些操作包括四种基本类型：插入、删除、更新和移动。 在上述的if-then分支中，节点‘[VariableAccess] emtmp‘从‘call to ossl_check_OPENSSL_STRING_type‘的子树中被删除，同时插入了一个代表‘email->data‘的新子树。 插入了一个类型为BlockStmt的包装节点，并将原始的return语句的子树移动到这里作为新插入的‘call to CRYPTO_free‘子树的右兄弟。值得注意的是，我们处理两种类型的移动：将节点移动为不同父节点的子节点，以及在同一父节点下更改顺序。

```
谓词 func_0(        参数          vemail_525, FunctionCall target_0) {
  target_0.getTarget().hasName("CRYPTO_strdup")
  and not target_0.getTarget().hasName("CRYPTO_strndup")
  and target_0.getArgument(0).(PointerFieldAccess).getTarget().
  ↪  getName()="data"
  and target_0.getArgument(0).(PointerFieldAccess).getQualifier(
  ↪  ).(VariableAccess).getTarget()=vemail_525
  and target_0.getArgument(1) instanceof StringLiteral
  and target_0.getArgument(2) instanceof Literal
}
```

清单2：一个示例原始 QL 谓词。

与四种编辑操作对应的节点的查询可以表示为目标的存在或不存在。 换句话说，我们需要匹配一个函数，其中插入的元素不存在，删除的元素存在，而更新和移动的节点与未修补的AST中的属性和位置相同。为了实现这一点，我们为每个差分节点定义一个谓词，它是以类似函数的形式呈现的一组查询条件。

3.2.3 将编辑脚本转换为规则

为了在 QL绑定中生成原始规则，我们需要构建一个查询，以定位每个差异的AST 节点。 这个查询应该包括特定条件，限制节点的属性和位置，以匹配相应的缺陷。

我们使用一个通用模板和 QL语法的子集作为生成规则的协议。基本上，

要确定节点必须满足的条件，重要的是引用数据对象并描述它们与目标之间的关系。 这种关联关系是以内存对象（变量）为中心的。 换句话说，它定义了所有必要的变量对象，并描述了所有AST节点和关联变量之间的关系。 在整个规则中，通过它们的原始变量名和声明位置来识别这些关键变量，并且只使用它们的类型作为查询限定符，确保能够区分目标变量对象，而不受变量重命名或别名的影响。 通过考虑目标节点和变量之间的关系，我们

可以概述相对位置。 从节点到相关变量的路径可以从AST中提取出来。例如，可以构建一个谓词来描述在函数调用中替换函数CRYPTO_strndup为CRYPTO_strdup的更新节点，如清单2所示。

3.2.4 优化和泛化原始 QL 规则

原始规则通常不足以精确地定位到特定节点，这可能导致过度匹配和不足匹配。 为了解决这个问题，我们需要包含进一步限制AST中节点位置的附加条件。我们利用控制流图（CFG）和数据流图（DFG）信息来增强规则。

基本上，CFG描述了某些松散相关的结构化模式，例如 then块中的语句是否对 if条件表达式进行了更改。DFG描述了修改子树之外引用的数据对象的依赖关系，例如，在稍后使用或验证之前对变量的赋值。 我们将在第4.2节详细介绍。

另一方面，我们希望规则更通用，
以便能够适应具有别名或
其他微小更改的变体实例。 目前，我们使用三种基本的泛化机制：

- 使用等效或更宽松的 QL 内置谓词。 例如，对于相等表达式目标 of A==B，泛化条件是‘target.getAnOperand().getTarget()=A‘ 而不是‘target.getLeftOperand().getTarget()=A‘。

- 丢弃无意义的目标和条件。 一些代码更改可能与错误直接相关，但不总是可能对其进行预处理。 在这种情况下，我们丢弃相应的谓词。 例如，与更新‘!email->data‘为‘email->data == NULL‘相关联的谓词在清单 1 中被丢弃。

- 替换包装符号名称。 许多项目使用第三方或定制的包装函数而不是通用接口，需要进行替换。 例如，在清单 1 中，对OPENSSL_strdup的调用被替换为 strdup。

3.2.5 回归测试

最后，我们有一个经过精细调整的 QLrule，已经准备好进行测试。 然而，进一步的改进仍然是必要的，比如一个基本的rule。 某些谓词和查询条件可能是多余的，对于某些节点的CFG和DFG信息可能不足以缩小报告范围。

在这个阶段，我们利用回归测试来自动进一步改进规则。 在确保规则仍然可以匹配目标函数的同时，我们尝试添加更多的CFG和DFG依赖项，以查看是否可以消除误报。我们还结合可能多余的谓词和条件来检查是否出现了新的误报。 这个迭代过程重复进行，直到最终获得一个合理的规则。

在整个过程之后，为所有已知的漏洞准备了一组可行的规则。 以后，我们可以将这些规则应用于与原始OSS直接或间接相关的任何项目的CodeQL数据库。例如，通过使用规则扫描LibreSSL，我们发现了它自己的副本中的一个看似有缺陷的函数 append_ia5 [20]（该问题已经报告给OpenBSD）。

4个关键技术

Patch2QL的实现利用了Ed-itScript算法的定制匹配方法和简化的方法来检索CFG和DFG信息。 在本节中，我们详细介绍了这两个关键技术。

4.1 Ed-itScript算法的定制匹配方法

原始算法称为*E*dit*S*crip*t*，可用于生成适用于分层结构信息（包括*S*A*S*T）的正确编辑脚本。然而，仅仅正确性是不够的，当涉及到源代码比较时。

让我们重新审视列表1中的示例。 EditScript算法的基本过程采用自上而下的贪婪方法，平等对待所有节点。当以这种方式匹配节点时，当涉及到未修补版本中第540行的语句节点‘return 1‘时，我们遇到了一个差异。 算法将其与修补版本中第535行插入的‘return 1‘节点配对，而不是与其实际复制品（第549行）配对。 这种逻辑上的不匹配在技术上是正确的，但它导致我们忽视了整个‘if (memchr(...) != NULL return 1;‘块被插入作为强制数据验证的手段。

为此，我们提出了一种定制的匹配方法，允许更准确地配对AST节点。 在这一步中，目标是创建一组配对，其中一个AST的每个节点与另一个AST中最匹配的节点配对。任何剩余的未配对节点被视为单个节点。 这构成了EditScript算法中主要过程的基础。 该方法在算法1中呈现。

匹配方法的基本思想是在进行搜索时考虑额外的上下文。 如果一个子树与另一个子树非常相似，即使它们在值、类型或谓词上可能不同，我们也可以直接将这两个节点配对在相同的位置上。

4.2 简化的CFG&DFG分析

为了增强节点的条件查询，上下文信息是必要的，主要有两个原因。首先，

算法1：匹配2个AST的多阶段方法。

定义：
D（节点）等于节点的所有后代数量Des_N（节点）是在以节点为根的子树上进行深度优先遍历时的第N个后代

如果节点A的类型、值和子节点数量与节点B相等，则节点A等于节点B
如果节点A的类型和子节点数量与节点B相等，则节点A近似等于节点B

函数 MATCHUNIT（节点A，节点B）
  如果节点A等于节点B且节点A的父节点等于节点B的父节点且对于 $N \leq D$（节点A） Des_N（节点A）等于 Des_N（节点B）则返回 1
  否则如果节点A等于节点B且对于 $N \leq D$（节点A） Des_N（节点A）等于Des_N（节点B）则返回 2
  否则如果节点A近似等于节点B且对于 $N \leq D$（节点A） Des_N（节点A）近似等于Des_N（节点B）则返回 3
  else if $Node_a = Node_b$ and $Parent(Node_a) \simeq Parent(Node_b)$则
    返回 4
  else if $Node_a = Node_b$ 则
    返回 5
  else if $Node_a \simeq Node_b$ 则
    返回 6

$M \leftarrow \emptyset$
对于 i在范围$(1..3)$内执行
  对于 $Node_a$在DFS遍历 $Tree_a$中执行
    如果 $Node_a \notin M$且 $\exists Node_b \notin M$且 MATCHUNIT$(Node_a, Node_b)$=i 则
      对于 $N \leq D(Node_a)$ 执行
        $M \leftarrow [Des_N(Node_a), Des_N(Node_b)]$
对于 i在范围$(4..6)$ 内执行
  对于 $Node_a$在DFS遍历 $Tree_a$中执行
    如果 节点$_a \notin M$ 且 $\exists$节点$_b \notin M$ 且 MATCHUNIT(节点$_a$, 节点$_b$)=i则
      $M \leftarrow$ [节点$_a$, 节点$_b$]

它用于建立查询独立节点之间的关联，从而缩小目标范围。这在查询移动节点或复制现有节点的插入节点时特别重要，以避免误报。其次，修补的代码通常放置在实际缺陷的上游或下游，以阻止特定条件或利用路径。这些区域不包含最重要的缺陷信息。 通过追踪修补位置周围的控制和数据流，向前和向后，我们增加了识别缺陷根本原因的机会。

虽然使用CodeQL可以方便地检索CFG和DFG信息，但我们目前不会过度复杂化任务。 分析直接在AST上执行，使用更直观的方法。

4.2.1 CFG 分析

目前，我们为属于特定特征控制结构的 AST 节点提供额外信息 具体而言，已考虑以下控制流关系：

- 如果目标节点是条件表达式或其子表达式的 `If/Switch/For/While` 语句，则引用 then/do 体内的特征语句。
  相反，如果目标节点位于 then/do 体内，则引用其控制条件表达式。
- 如果目标节点是赋值或比较操作的 `lvalue`或 `rvalue`，则引用其兄弟节点。
- 如果目标节点的父节点是围绕数组的表达式，则引用其兄弟节点（数组基址或偏移量）。
- 更一般地，对于 *expression*类型的节点，引用包裹目标的最外层表达式。
- 如果需要更多上下文信息来进一步缩小目标范围，则对于 *statement*类型的节点，引用其直接父节点和兄弟语句（如果有）。

4.2.2 DFG 分析

目前，我们尚未使用精确的 DFG 分析。 相反，我们只是引用每个差分节点块中访问的关键变量的直接赋值和使用。 分析是根据访问类型执行的，如下所示：

- 如果一个变量作为 `rvalue`访问，我们会追溯控制流以找到其最近的 `lvalue`和 `rvalue`引用。 LValue 引用很可能是一个赋值操作，它确定了作用域内使用的变量的值/属性。 rvalue引用可能是一个验证操作。 如果 `lvalue`引用更接近，则忽略 `rvalue`引用节点。
- 如果一个变量作为 `rvalue`访问，我们会向前追溯控制流以找到其最近的引用。 如果引用结果是 `rvalue`，那么它就成为 DFG 上下文节点的候选项。
- 如果一个变量作为一个 `lvalue`被访问，我们会向前追踪控制流以找到其后续的 `rvalue`引用，直到遇到一个 `lvalue`引用为止。 我们将所有相关的节点标记为补充DFG节点的候选项。

5 评估和观察

接下来，我们介绍了`Patch2QL`的评估结果。 有效性可以基于覆盖率和检测进行评估。 由于规则生成阶段是离线进行的扫描阶段，并且扫描是通过调用CodeQL进行的，性能不是一个主要考虑因素。 更重要的是通过展示

通过对广泛测试的项目进行实验结果分析，特别是展示新发现的缺陷，我们可以展示该工具的创新性，并揭示开源软件中同源缺陷的情况。

5.1 准确率和召回率

我们将Patch2QL与三个可用于开源软件的工具进行比较：Coverity [28]，这是一个在商业和开源领域都得到广泛验证的SAST产品；VUDDY [16]，它利用基于签名的技术进行可扩展的代码克隆检测，并在最近的研究工作中得到了广泛的引用和比较；TRACER [15]，这是过去两年研究趋势的一个研究成果，其中利用污点分析跟踪来集成表示的漏洞代码特征。 这三个工具代表了解决同源缺陷问题的主要方法：使用通用的SAST工具和规则来发现特定模式的漏洞，使用文本和结构模糊匹配方法来发现具有相似性的代码，以及使用控制流和数据流所隐含的更丰富的漏洞特征信息来发现语义上相似的代码。

数据集为了衡量每个目标在检测同源缺陷方面的基准准确性，我们选择了8个真实的开源C语言项目。 这8个项目每个都有超过30个历史CVE漏洞，其中可以收集到漏洞补丁，并且项目代码非常复杂。 在这里，我们统一选择了2019年之后每个项目的第一个开发版本，因此应该受到2020年以来新发现的一部分漏洞的影响。

至于要匹配的目标缺陷，我们收集了2020年至2023年6月期间的所有CVE漏洞，并且只有那些通过对源代码应用补丁（而不是Makefile或配置文件）并且更改了代码后使用默认配置进行编译的漏洞。 CVE的总数由$CVE\#$表示。

对于每个工具，我们评估了检测到的总数，即阳性结果。 其中，我们尝试确认那些准确命中包含CVE漏洞并进行了修补的函数的结果，即真阳性（TP）。 特别是对于Coverity来说，它提供了大量与任何相关CVE无关的缺陷报告，我们试图过滤掉那些命中与CVE漏洞对应的易受攻击函数并且报告的原因符合漏洞类型的报告。

实验数据列在表[1]中，我们可以从三个角度解释。 由于很难确定代码库的快照版本是否受到后来的CVE的影响，统计误报率可能没有实际意义。

因为许多漏洞往往只影响新开发的功能，并且几乎没有可用于准确确定受漏洞影响的提交范围的方法或标准化数据。

重要性。

有效的正例从表中可以看出，与其他三个基准工具相比，Patch2QL生成的规则为每个目标项目实现了最准确的检测，同时保持了最低的总正例数。 因此，它实现了90%的评估准确率。 由于此基准实验仅关注前几章定义的L1和L2类型的代码克隆，忽略了由开发演化引起的代码差异和本体项目中的目标特定缺陷代码，我们可以观察到以相似性签名为基础的方法（如VUDDY）取得了良好的结果。 另一方面，以语义相似性为基础的方法（如TRACER）在检测模糊代码克隆方面存在困难。

我们还可以看到，以Coverity为代表的通用SAST工具和规则，在对目标代码进行全面分析后生成了大量报告。
然而，实际命中已知漏洞的报告比例相对较低。这反映出，在像Coverity这样持续通过工具进行分析的开源项目中，新披露的漏洞通常不被通用漏洞类型和特定编程语言规则所覆盖。 这些漏洞往往偏向于逻辑原因或涉及完整漏洞起源的因果分析，这可能超出传统SAST工具的能力边界。

召回率为了衡量Patch2QL规则的召回率，我们进行了额外的实验。 我们使用生成的原始规则扫描了每个项目在每个安全漏洞修补程序之前的版本（这些版本肯定受到漏洞的影响）。 然后，我们统计了能够成功匹配到漏洞（自2020年以来）的情况。 这些情况在表中单独定义为TP2。

只有在未修补版本中的核心易受攻击函数与其相应的规则匹配时，漏洞才被认为是被召回的。 总体而言，由Patch2QL生成的规则的召回率，通过$TP2/CVE\#$评估，达到了71.4%。 对所有111个项目进行的综合评估得出了68.0%的召回率。

如表1中的实验结果所示，所有CVE漏洞的召回率范围从47%（ImageMag-nick）到100%。 这些数据的波动在第6节中进一步讨论和解释。

假阳性我们使用了所有CVE漏洞的规则集，但不包括2020年之后的漏洞，来扫描目标项目。 假阳性包括两部分检测结果：与代码库匹配的2020年之前披露的CVE，以及与非常易受攻击函数以外的代码出现匹配的CVE。

这些FP案例的大部分可以归因于常见的根本原因。 在某些项目中，应用于函数的临时补丁可能已经重构以符合编码规范

表1：Coverity、VUDDY、Tracer和 Patch2QL在同源缺陷检测中的准确性

目标	提交	CVE#	Coverity		VUDDY		TRACER		Patch2QL		
			TP	正例	TP	正例	TP	正例	TP	正例	TP2
ImageMagick	dfd6399	44	0	157	8	12	0	0	7	23	21
curl	e24ea70	40	1	120	4	10	0	4	15	17	34
FFmpeg	7b58702	52	0	749	9	13	1	1	16	23	35
OpenSSL	12ad22d	21	0	305	4	51	1	3	4	13	15
ghostscript	d7d012b	29	0	648	6	13	运行错误		12	23	21
libtiff	0a8245b	28	0	45	1	4	0	6	10	22	23
libxml2	8f62ac9	11	0	215	1	1	0	32	11	15	11
openjpeg	024b840	9	0	25	8	16	0	0	5	9	7

样式和常见修复代码可能已经被抽象成一个单独的函数。 在这种情况下，QL可能会错误地识别重构后的函数，而没有足够的过程间分析。

此外，对于某些漏洞，它们的补丁可能没有提供足够的关于根本原因或错误的具体位置的信息。这些补丁可能只是暗示了控制流中存在错误。 其他一些补丁可能旨在阻止攻击路径，而不是直接修复错误。在这些情况下，QL可能会错误地报告其他具有类似语法但需要更多上下文信息才能更准确匹配的函数。

大多数自动生成的规则属于上述两个类别，可以进行微调以消除误报，同时仍然识别目标缺陷。

然而，也有一些无法轻易解释为误报的意外结果。 其中一部分结果实际上是同源缺陷，将在下文中进一步讨论。

5.2 发现的新同源缺陷

我们通过将历史漏洞的规则应用于各个项目的最新开发分支来进行了实验。 在对所有111个基本开源软件的扫描结果进行简要调查后，发现了一些新的同源缺陷，列在表2中。 所有这些缺陷都已及时报告给项目维护者并得到修复。 在某些情况下，我们能够创建触发这些错误的概念验证（PoC）示例，然后将其分类为漏洞并分配CVE ID。还有其他具有明确根本原因且易于识别的错误。 然而，要证明它们的可利用性需要对特定项目有广泛的专业知识。 因此，我们将其报告给维护者作为潜在的安全漏洞。 大多数这些情况被认定为常规错误，并得到适当的处理和修复。

截至6月30日，大约30天内报告的缺陷 我们仍在对其他潜在缺陷进行分析 由于可能有黑客从扫描报告中发现未公开的漏洞，我们暂时停止上传产生误报结果的QL文件到公共仓库

从这个列表中，我们得出了几个总体观察结果
首先，具有大量子模块或类似功能的项目更容易受到同源缺陷的影响 在实现子模块时，例如特定格式的解析器或设备驱动程序，开发人员经常重用来自类似模块的现有代码 这种重用可以发生在文件、函数、块和底层逻辑层面 这些情况属于L2-L3类型的同源缺陷，如第2.2节中定义的，由后续开发人员的模仿引起

其次，在新漏洞报告后，开源社区通常不会专门进行针对同源缺陷的代码审查

许多项目中存在代码克隆和派生的实例，缺乏记录，这使得维护者在跟踪此类情况时面临挑战，特别是当代码从其他作者那里复制时。

第三，普遍缺乏安全意识导致同源缺陷的出现。 例如，考虑CVE-2020-29599：在PDF格式的解析过程中，用户输入的字符串通过一个选项传递，并在连接到委托命令以执行之前进行了清理。 然而，清理程序未检查非法字符，例如常用的反引号字符用于命令注入。 在修复初始漏洞约一年后，添加了一个新功能来支持视频格式。 不幸的是，同样的错误在新模块中再次出现，导致CVE-2023-34153。同样，启用管道的功能也对用户提供的文件名进行了清理，以供稍后用作命令。 然而，清理程序再次未检查反引号的存在，导致CVE-2023-34152。这个案例明确证实了第2.2节中描述的同源缺陷由常见弱点引起的情况。

表2展示了确认的分析缺陷　　案例研究让我们来检查发现的相关缺陷

表2：已解决的同源缺陷。

OSS	标识符	严重程度	原始CVE	Fix	描述
ImageMagick	CVE-2023-34153	高	CVE-2020-29599	d31c80	一个新添加的功能中出现了一个不完整的清理问题。
ImageMagick	CVE-2023-34152	严重	CVE-2020-29599	17c485	CVE-2016-5118的修复不完整。
ImageMagick	CVE-2023-34151	中等	CVE-2022-32546	3d6d98	在不同格式的解码器中存在类似的错误类型转换问题。
opensc	CVE-2023-2977	高	CVE-2021-42782	81944d	有缺陷的代码在另一个模块中被重用。
ghostscript	CVE-2023-38559	中等	CVE-2020-16305	d81b82	在设备之间存在类似的越界问题。
ghostscript	CVE-2023-38560	中等	CVE-2017-9619	b7eb1d	未移植到 XPS 的 PCL 处理错误的补丁。
vim	CVE-2023-3896	高	CVE-2023-0512	8154e6	请参见下文。
openssl	问题 #21111	/	CVE-2015-1794	43596b	加密设置方法中存在类似的除零问题。
curl	问题 #11195	/	CVE-2022-27780	6375a6	未移植到冗余检查点的补丁。
Linux 内核	待定	/	各种各样	各种各样	一个不断增长的补丁列表 [2]

```
diff --git a/src/move.c b/src/move.c
index d3648df8b..3c50d258c 100644
--- a/src/move.c
+++ b/src/move.c
@@ -1933,6 +1933,9 @@ adjust_skipcol(void)
     return;

      int         width1 = curwin->w_width - curwin_col_off();
+     if (width1 <= 0)
+         return; // 不会显示任何文本
+
      int      width2 = width1 + curwin_col_off2();
      long     so = get_scrolloff_value();
      int      scrolloff_cols = so == 0 ? 0 : width1 + (so - 1) *
      ↪  width2;
@@ -1976,5 +1979,5 @@ adjust_skipcol(void)
     if (col > width2)
     {
         row += col / width2;
         col = col % width2;
     }
```

列表3：CVE-2023-0512的补丁，附加上下文。

更加密切地了解VIM。 最初的漏洞CVE-2023-3896是一个除零错误。当光标在VIM窗口内移动时，会重新计算其宽度，导致除零错误的发生。 新的宽度可能小于或等于零，随后被用作计算行数的除数。 解决此漏洞的补丁可以在列表3中找到。

鉴于漏洞的根本原因是变量 `width1`没有进行零检查，规则的主要方面是识别可比较的声明或赋值。 生成的规则附在附录的第9页供参考。

通过使用 QL规则，我们能够在另一个名为`scroll_cursor_bot`的函数中识别出同源缺陷。
可以在第4页的代码片段中看到易受攻击的代码。 这个函数和`adjust_skipcol`之间唯一的相似之处是更新 `width1`和 `width2`的计算。没有任何指示-

```
2592 int skip_lines = 0;
2593 int width1 = curwin->w_width - curwin_col_off();
2594 int width2 = width1 + curwin_col_off2();
2595 //在curs_columns()中使用了类似的公式
2596 if (curwin->w_skipcol > width1)
2597   skip_lines += (curwin->w_skipcol - width1) / width2 + 1;
```

列表 4：CVE-2023-3896 在`scroll_cursor_bot`中。

这些函数之间存在重复代码，使得使用基于代码相似性或基于签名的技术极其困难来发现后者函数中的漏洞。 这一发现清楚地证明了 Patch2QL 在检测 L3-L4 级别的同源缺陷方面的有效性，这些缺陷共享了最弱的相似性。

5.3 发现下游的同源缺陷

通过为 OSS 使用一组规则，可以对上游项目源自下游开源生态系统的漏洞影响进行彻底调查。 为了更清晰地说明实际影响并确定最重要的案例，我们选择了在云计算中广泛使用的两种软件类型：数据库和 Linux 内核。这些软件系统以其广泛的规模和复杂的代码库而闻名。 许多供应商经常分叉已建立的上游项目，并在自己的副本中进行重大修改，这增加了在此类下游项目中确认 CVE 存在或解决方案的复杂性。

5.3.1 数据库

关于数据库，我们选择了两个上游的开源软件项目，基于它们在不同形式的下游中的常见利用。 第一个是sqlite3 [9]，它通常

表3：从sqlite3和PostgreSQL派生的项目中发现的缺陷。

OSS	状态	上游版本	缺陷
SQLCipher	c7f9a1	3.41.2	0
libSQL	c734a2	3.43.0	0
comdb2	eb6e89	3.28.0	10
Tarantool	852664	/	7
unqlite	057067	/	1
AgensGraph	247f32	13.9	0
Greenplum	b6acf7	12.12	0
Kunlun	5a4a4d	11.5	15
postgre-xl	31dfe4	10.9	16
TBase	08b8c7	10.0	20
PolarDB	c39c0d	11.9	6
openGauss	c61c75	9.2.4	3

在源代码形式中重新设计和嵌入到第三方程序中。 第二个是PostgreSQL [14]，它作为各种商业数据库扩展的基础。

在2023年9月进行的一项简短实验中，我们收集了一些看起来是从sqlite3或PostgreSQL派生的OSS项目。我们的选择只包括具有大量星标和活跃开发的项目。 由于大多数派生项目没有明确维护分支信息，我们主要依赖特征字符串进行搜索。 分析结果见表3。

这些下游项目中的大部分由公司组织托管，我们观察到与上游开发人员相比，在对错误和漏洞报告的响应方面存在显著差异。 尽管所有缺陷都通过GitHub问题报告给项目所有者，但只有少数被确认，这使它们被归类为潜在缺陷。

一些项目在与上游开源软件同步方面采取了良好的做法。例如，SQLCipher、libSQL、AgensGraph和Greenplum等项目始终使用相对最新的代码库版本。 它们选择了或手动回溯了来自上游的补丁，即使这些补丁仅适用于最新的分支。 另一方面，一些项目存在常见问题，这些问题不仅使它们容易受到已知漏洞的影响，而且还使其在手动检查CVE存在方面具有挑战性。下面列举了一些说明此类常见问题的示例。

案例研究 **Tarantool** [30] 作为一个代表性的项目，定制并嵌入第三方开源组件的源代码形式。 在这种情况下，**sqlite**3 项目被用作 SQL 解析器。 然而，所有版权声明都被替换了，并且在函数和变量名中的所有'**sqlite**'字符串都被删除了。 因此，无法建立

```
diff --git a/src/select.c b/src/select.c
index 5f51074a0..44fb06f48 100644
--- a/src/select.c
+++ b/src/select.c
@@ -6065,5 +6065,6 @@ int sqlite3Select(
   if( (p->selFlags & (SF_Distinct|SF_Aggregate))==SF_Distinct
    && sqlite3ExprListCompare(sSort.pOrderBy, pEList, -1)==0
+   && p->pWin==0
   ){
     p->selFlags &= ~SF_Distinct;
     pGroupBy = p->pGroupBy = sqlite3ExprListDup(db, pEList, 0);
```

图5：sqlite3中CVE-2019-19244的补丁。

```
5833 if ((p->selFlags & (SF_Distinct | SF_Aggregate)) == SF_Distinct
5834     && sqlExprListCompare(sSort.pOrderBy, pEList, -1) == 0) {
5835     p->selFlags &= ~SF_Distinct;
5836     pGroupBy = sql_expr_list_dup(pEList, 0);
5837     p->pGroupBy = pGroupBy;
```

图6：Tarantool中易受攻击的函数sqlSelect。

Tarantool和sqlite3之间的明确对应关系。

缺乏映射信息给开发人员识别上游错误的存在带来了挑战。 尽管Tarantool中的嵌入式代码得到了积极维护，但仍然缺少一些补丁。 例如，在Tarantool中的相应的 `sqlSelect`函数（图6）中未找到CVE-2019-19244的补丁（图5）。 API名称的更改导致了代码混淆，但生成的 `QLrules`仍然能够匹配它们。 这个例子可以作为L2级同源缺陷的代表。 它突出了在项目的广泛代码重构和二次开发之后仍然存在残留缺陷的情况。

另一个例子是openGauss [24]，一个开源的关系数据库管理系统。它的内核起源于早期版本的PostgreSQL，该版本已不再获得上游支持。 PostgreSQL的代码库被重构为C++，并得到了社区的持续维护，包括对旧代码库的改进补丁。 然而，尽管进行了重构工作，仍然可以找到未修复的漏洞。

5.3.2 Linux发行版的内核

Linux内核生态系统构成了一个独特的软件环境。 除了上游分支的复杂分支外，每个Linux发行版都会开发一个或多个自己的内核，导致内核的碎片化程度相当高。 当涉及到维护稳定的开源软件包，包括内核时，大多数发行版（如RHEL [26]和SLES [27]）选择直接与上游内核分离，并回溯安全补丁。 然而，所有这些分发的内核都遵循上游内核的特定补丁版本，并且不会重新基于更新的版本，因为这样做需要合并更改的工作。

表4：在云环境的Linux发行版中发现的潜在内核缺陷

操作系统	提交	派生自	缺陷		
			≥高	中等	低
openEuler	03ad78	5.10.0	0	5	0
OpenAnolis	89caf8	4.19.91	12	12	1
OpenCloudOS	320119	5.4.119	12	14	1

因此，将上游补丁的回溯工作落在社区开发者身上。

2023年2月底，我们使用了中国三大云计算供应商托管的三个开源Linux内核进行了一项实验。我们利用了一组选定的QL规则，这些规则覆盖了从2021年开始的上游Linux内核中的CVE漏洞和其他安全漏洞[18]。 结果见表4。

在分析结果并检查每个发行版的git历史记录后，我们得出了三个结论。 首先，每个发行版采用的漏洞策略是确定其安全级别的重要因素。 在这三个发行版中，openEuler [22]始终合并了来自上游5.10.y分支的安全补丁。 相反，OpenAnolis [19]和OpenCloudOS [21]更倾向于在发布新版本之前一次性合并一批补丁。 其次，大多数缺陷是在过时的代码库中发现的，包括已经更新并回溯的原始代码，但保留了来自更高版本的未维护的功能。 内核的定制部分主要涉及为第三方设备和子系统添加驱动程序，而不是重写主线代码。 只要依赖的内核版本保持在上游生命周期内，这使得验证缺陷的存在相对容易。 第三，广泛的版本碎片化导致了许多复杂性。 如果一个功能从较新版本回溯，维护者经常会丢失上游的安全补丁。 此外，如果维护者在Linux社区上游之前为零日漏洞创建补丁，在后续的代码同步过程中可能会出现冲突，可能导致旧漏洞的再次发生。

案例研究 在openEuler中报告并确认的一个缺陷是CVE-2022-1015 [11]（称为openEuler-SA-2023-1253 [23]）。 原始CVE在2022年3月17日修复，最初在2022年4月29日披露。 根据报告，这个错误只影响从v5.12-rc1开始的主线内核，并且补丁已应用于5.15、5.16和5.17的LTS内核。 当时认为基于5.10上游版本的openEuler内核不受影响。 然而，结果表明漏洞模式与代码完全匹配。 随后，在2023年4月18日，openEuler维护者承认了这份报告并合并了主线修复[4]。 直到2023年6月28日才发现

```
diff --git a/arch/x86/kernel/cpu/resctrl/rdtgroup.c
↪ b/arch/x86/kernel/cpu/resctrl/rdtgroup.c
index 064e9ef44cd6..9d4e73a9b5a9 100644
--- a/arch/x86/kernel/cpu/resctrl/rdtgroup.c
+++ b/arch/x86/kernel/cpu/resctrl/rdtgroup.c
@@ -3072,7 +3072,8 @@ static int rdtgroup_rmdir(struct
↪ kernfs_node *kn)
     * 如果rdtgroup是一个mon组，并且父目录
     * 是一个有效的“mon_groups”目录，则删除mon组。
     */
-    if (rdtgrp->type == RDTCTRL_GROUP && parent_kn ==
↪ rdtgroup_default.kn) {
+    if (rdtgrp->type == RDTCTRL_GROUP && parent_kn ==
↪ rdtgroup_default.kn &&
+        rdtgrp != &rdtgroup_default) {
         if (rdtgrp->mode == RDT_MODE_PSEUDO_LOCKSETUP ||
             rdtgrp->mode == RDT_MODE_PSEUDO_LOCKED) {
                 ret = rdtgroup_ctrl_remove(kn, rdtgrp);
```

图7：Intel RDT中的错误修补程序。

```
static int resctrl_group_rmdir_ctrl(struct kernfs_node *kn,
↪ struct resctrl_group *rdtgrp,
                    cpumask_var_t tmpmask)
{
    resctrl_group_rm_ctrl(rdtgrp, tmpmask);
    kernfs_remove(rdtgrp->kn);
    return 0;
}

static int resctrl_group_rmdir(struct kernfs_node *kn)
{
    ...
    if (rdtgrp->type == RDTCTRL_GROUP && parent_kn ==
    ↪ resctrl_group_default.kn)
        ret = resctrl_group_rmdir_ctrl(kn, rdtgrp, tmpmask);
    else if (rdtgrp->type == RDTMON_GROUP &&
         is_mon_groups(parent_kn, kn->name))
        ret = resctrl_group_rmdir_mon(kn, rdtgrp, tmpmask);
    ...
}
```

清单8：在借用的 `resctrlfs`中的同源片段

上游社区将修复程序挑选到维护分支linux-5.10.y [5]，导致间隔了468天。

另一个值得注意的案例是识别出一个有问题的同源片段。 除了CVE漏洞外，我们还考虑了在GSD [3]中发现的安全漏洞用于规则生成。 在openEuler中，开发人员在*fs/resctrlfs.c*文件中实现了ARM v8 MPAM的用户界面。 这个实现部分借用了上游内核对Intel RDT的实现，正如宣布的那样。 我们发现了一个关于后者存在的错误的规则，如清单7所示，并在前者中找到了一个同源片段，如清单8所示。 经过评估，社区维护人员得出结论，这段代码并不构成一个漏洞。 然而，它仍然作为一个例子来展示在代码被重用时同源漏洞在现实世界中出现的可能性。

6 讨论

召回率 如表1所示，不同项目的召回率差异很大。 这种差异可以归因于代码库和补丁的平均复杂性。 一般来说，过于简单或过于复杂的补丁往往质量较低。

对于过于简单的补丁，只插入了一个语句或表达式，可能没有足够的特征或相关的CFG / DFG上下文来将规则转化为有意义的谓词和查询条件。 为了衡量补丁中所携带的信息丰富程度，我们计算了八个实验项目中所有目标漏洞补丁的平均变更行数（包括添加和删除的行数）。 对于ImageMagick，每个补丁的平均变更行数为40.5，每个相关函数的平均代码行数（LoC）变更为27.9，而libtiff的结果分别为143.7和46.2。这意味着在ImageMagick中有很大一部分补丁只改变了很少的代码，导致生成的规则产生了许多误报。

另一方面，过于复杂的补丁可能表示代码逻辑的复杂更改甚至完全重写。 这些补丁可能涉及太多元素，无法在生成的规则中充分表示。 在其他一些情况下，特定项目的开发人员选择推送一个巨大的提交，其中漏洞修复和其他无关的更改混在一起。在PostgreSQL的极端情况下，据计算每个补丁涉及8.2个函数和935.0行代码。所有这些都导致了将核心代码更改从补丁中分离出来并生成规则的困难。

开销在评估部分，我们没有专门评估或比较工具的执行效率。 这主要是因为扫描过程是由我们依赖的Co**de**QL实现的，并且使用P**atch**2QL基于补丁生成规则是离线完成的，因此可以忽略开销。 然而，考虑到基于签名的模糊匹配算法和SAST规则匹配都是计算密集型机制，它们的执行效率仍然值得讨论。

在实验中，我们设置了CodeQL使用8个并行线程来分析每个目标项目，并且超过98%的规则在30分钟内完成执行。 这实际上比CodeQL的平均内置规则更快，因为数据流分析仅用于规则的生成而不是执行阶段。 Coverity的扫描结果通常在上传项目的中间数据并开始扫描后的半小时内返回给用户。 至于基于自定义签名的工具，情况则大不相同。

对于VUDDY，本地签名计算阶段平均可以在10分钟内完成，上传后即刻返回分析结果给用户。 然而，对于TRACER，时间和内存消耗可能是不可取的。 在一个安装了80个CPU核心和350GB内存的服务器上，一个分析任务最多占用了一半的核心和超过60%的内存，在一个极端情况下，ghostscript的分析无法完成。

从上面的比较可以看出，使用基于签名的模糊匹配方法时，随着签名携带更多信息和项目规模的增加，计算开销可能变得无法控制。 另一方面，基于句法和语义模式匹配的SAST方法具有更好的期望值，并且SAST中的规则本身更易读。 我们还期望如果用Clang AST Matcher [7]等特定工具替代CodeQL，开销会大大降低。

可扩展性如第2.3节所解释的，由于相应的软件供应链的历史负担，我们选择了C/C++语言中的**OSS**问题作为我们的主要研究目标。 然而，我们的方法并不局限于特定的编程语言。

从技术角度来看，我们目前使用的底层工具CodeQL为所有主流语言生态系统提供了分析能力，包括AST转储和分析。Patch2QL本身采用了跨语言的通用方法进行结构化AST比较和QL查询组合。 生成特定语言的语法规则需要额外的适应和优化，例如如何执行前向和后向遍历控制流，我们相信这只需要一定的开发工作量。

从目标的角度来看，我们的初步分析针对Java和其他生态系统已经揭示出同源缺陷也很普遍，尽管它们的形式可能有所不同。
例如，在Java中，有些情况下明确引入的包与其在源代码形式中隐含引入的旧版本共存，存在隐藏的威胁。这些问题也可以使用Patch2QL的现有功能来探索。

7 结论

在本文中，我们揭示了开源软件中同源缺陷的起源和风险。我们提出了一个全面的过程，将漏洞修补程序转化为CodeQL规则，捕获关键的语义和依赖上下文信息。我们开发了一个名为Patch2QL的原型工具。在流行的C语言编写的开源软件上应用我们的系统进行评估，证明了其有效性和独特的见解。 它在历史CVE中实现了68%的召回率，并且比通用的SAST规则和基于签名的机制更有效。 它发现了7个新的漏洞，并揭示了上游项目、下游数据库和Linux内核中的其他问题，包括各种同源缺陷。

可用性

Patch2QL生成的有关开源软件供应链中历史漏洞的规则（包括Linux内核）已在存储库[1]中开源，并可供开源开发者自由使用。

参考文献

[1] Qlrules | github, 2023. https://github.com/fullwaywang/QlRules.

[2] fullway的Linux内核补丁，2024年。 https://lore.kernel.org/all/?q=fullway.

[3] 云安全联盟。全球安全数据库，2023年。https://github.com/cloudsecurityalliance/gsd-database.

[4] Pablo Neira Ayuso。修复cve-2022-1015，2022年。https://gitlab.com/linux-kernel/stable/-/commit/6e1acfa387b9ff82cfc7db8cc3b6959221a95851.

[5] Pablo Neira Ayuso. 在linux-5.10.y中选择性修复提交，2023年。https://gitlab.com/linux-kernel/stable/-/commit/9e8d927cfa564e5a00cd287bd66fac6d45f0af39.

[6] Sudarshan S Chawathe, Anand Rajaraman, HectorGarcia-Molina, and Jennifer Widom. 在分层结构化信息中进行变更检测。*Acm Sigmod**Record*, 25(2):493–504, 1996.

[7] LLVM Clang. Ast匹配器参考。https://clang.llvm.org/docs/LibASTMatchersReference.html.

[8] SQLite联盟。 Cve-2019-19244, 2019. https://nvd.nist.gov/vuln/detail/CVE-2019-19244.

[9] SQLite联盟。Sqlite, 2023.https://www.sqlite.org/index.html。

[10] Oege De Moor，Mathieu Verbaere，Elnar Hajiyev，PavelAvgustinov，Torbjorn Ekman，Neil Ongkingco，DamienSereni和Julian Tibble。主题演讲：。 用于源代码分析的ql。 在第七届IEEE国际源代码分析和操作会议（SCAM 2007）中，第3-16页。IEEE，2007年。

[11] Linux基金会。Cve-2022-1015 | Linux内核cves，2022年。https://www.linuxkernelcves.com/cves/CVE-2022-1015。

[12] OpenSSL软件基金会。 Cve-2021-3712, 2021年。 https://nvd.nist.gov/vuln/detail/cve-2021-3712。

[13] GitHub公司。Codeql，2023年。https://codeql.github.com。

[14] PostgreSQL全球开发团队。Postgresql，2023年。https://www.postgresql.org/。

[15] Wooseok Kang，Byoungho Son和Kihong Heo。Tracer：基于签名的静态分析用于检测重复漏洞。 在*2022年ACM*SIGSAC计算机与通信安全会议论文集，第1695-1708页，*2022*年。

[16] Seulbae Kim，Seunghoon Woo，Heejo Lee和Hakjoo Oh。Vuddy：一种用于易受攻击代码克隆发现的可扩展方法。 在*2017*年I*EEE*安全与隐私研讨会（S*P*）论文集，第595-614页。I*EEE*，*2017*年。

[17] Google有限责任公司。 Google oss-fuzz项目，2023年。https://github.com/google/oss-fuzz/tree/master/projects。

[18] Google有限责任公司。漏洞库 | linux | osv，2023年。https://osv.dev/list?ecosystem=Linux。

[19] OpenAnolis。anolis/cloud-kernel | gitee，2023。https://gitee.com/anolis/cloud-kernel。

[20] OpenBSD。 libressl/openbsd | github，2023。https://github.com/libressl/openbsd/blob/master/src/lib/libcrypto/x509/x509_utl.c#L725C1-L749C2。

[21] OpenCloudOS。Opencloudos/opencloudos-kernel | gitee，2023。 https://gitee.com/OpenCloudOS/OpenCloudOS-Kernel。

[22] openEuler。 openeuler/kernel | gitee，2023。 https://gitee.com/openeuler/kernel。

[23] openEuler。 安全公告 | openeuler，2023。 https://www.openeuler.org/zh/security/security-bulletins/detail/?id=openEuler-SA-2023-1253。

[24] openGauss。 opengauss | gitee，2023年。https://gitee。com/opengauss/openGauss-server。

[25] Nam H Pham，Tung Thanh Nguyen，Hoan Anh Nguyen，和Tien N Nguyen。检测到重复的软件漏洞。在I*EEE*/A*CM*国际会议上的自动化软件工程中，第447-456页，2010年。

[26] 红帽公司。 红帽企业Linux，2023年。https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux。

[27] SUSE。Suse Linux企业服务器发行版，2023年。https://www.suse.com/products/server/。

[28] Synopsys公司。Coverity扫描静态分析，2023年。https://scan.coverity.com/。

[29] Synopsys公司。开源安全和风险分析报告，2023年。https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html.

[30] Tarantool. tarantool | github, 2023. https://github.com/tarantool/tarantool.

[31] Google Project Zero. 2022年0日在野利用...到目前为止，2022年. https://googleprojectzero.blogspot.com/2022/06/2022-0-day-in-wild-exploitationso-far.html.

[32] 雷赵，云聪朱，江明，一晨张，昊天张和恒银。Patchscope：基于内存对象的补丁差异。在*2020*年*ACM SIGSAC*计算机和通信安全会议论文集, 第149-165页，*2020*年.

附录

```
import cpp

谓词 func_0(      变量       vwidth1_1935, AddExpr target_1) {
    存在(IfStmt target_0 |
    target_0.getCondition().(RelationalOperation).
    ↪  getLesserOperand().(VariableAccess).getTarget(
    ↪  )=vwidth1_1935
    和 target_0.getCondition().(RelationalOperation).
    ↪  getGreaterOperand().(Literal).getValue()="0"
    和 target_0.getCondition().(RelationalOperation).
    ↪  getLesserOperand().(VariableAccess).getLocation().
    ↪  isBefore(target_1.getAnOperand().(VariableAccess).
    ↪  getLocation()))}

谓词 func_1(      变量       vwidth1_1935, AddExpr target_1) {
    target_1.getAnOperand().(VariableAccess).getTarget(
    ↪  )=vwidth1_1935
    和 target_1.getAnOperand().(FunctionCall).getTarget().
    ↪  hasName("curwin_col_off2")}

谓词 func_2(      变量       vwidth1_1935, 初始化器 target_2) {
    vwidth1_1935.getInitializer()=target_2
    和 target_2.getExpr().(SubExpr).getLeftOperand().(
    ↪  PointerFieldAccess).getName()="w_width"
    和 target_2.getExpr().(SubExpr).getRightOperand().(
    ↪  FunctionCall).getTarget().hasName("curwin_col_off")}

来自函数 func, 变量 vwidth1_1935, AddExpr target_1,
↪  初始化器 target_2
其中
不 func_0(vwidth1_1935, target_1)
和 func_1(vwidth1_1935, target_1)
和 func_2(vwidth1_1935, target_2)
和 vwidth1_1935.getType().hasName("int")
和 vwidth1_1935.(LocalVariable).getFunction() = func
选择 func
```

列表 9：生成的 QL规则匹配 CVE-2023-0512。

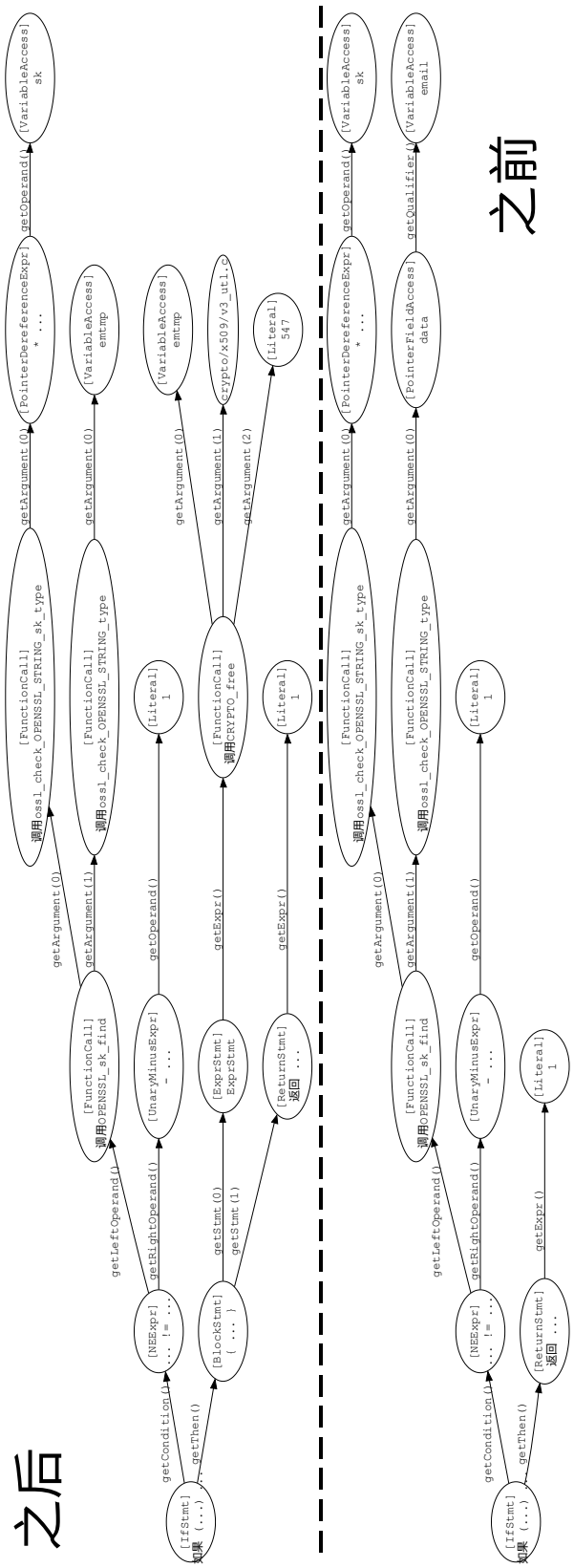

图2：预补丁和后补丁函数的AST对