# Linux安全模块框架授权钩子函数放置的一致性分析

TRENT JAEGER IBM T. J. Watson Research Center，ANTONY EDWARDS  Symbian Ltd.  and  XIAOLAN ZHANG IBM T. J. Watson Research Center

我们提出了一种一致性分析方法来帮助Linux社区验证Linux安全模块（LSM）框架中授权钩子放置的正确性。LSM框架由一组插入到Linux内核中的授权钩子组成，用于执行额外的授权（例如，用于强制访问控制）。与系统调用插入相比，内核中的授权更具有安全性和性能上的优势，但更难的是验证LSM钩子的放置是否能够确保所有内核的安全敏感操作都得到授权。静态分析以前用于验证这些中介（即，某些钩子中介对安全敏感操作的访问），但该工作无法确定是否检查了所有必要的授权集。在本文中，我们开发了一种方法来测试安全敏感操作和LSM钩子之间关系的一致性。其思想是，每当安全敏感操作作为可指定事件的一部分执行时，一定有一组特定的LSM钩子中介了该操作。这项工作表明，影响一致性的事件数量是可管理的，一致性的概念对于验证正确性很有用。我们描述了执行验证的一致性方法、实现此方法的运行时工具的实现、在打上了LSM补丁的Linux 2.4.16内核中发现的异常情况，以及此方法的静态分析版本的实现。

## 1 介绍

Linux安全模块（LSM）项目旨在提供一个通用框架，在这个框架之下可以使得各种授权机制和策略得到强制执行。这样的框架将使开发人员能够实现他们为Linux内核选择的授权模块。然后，系统管理员可以选择最能实施其系统安全策略的模块。通常，其目标是除了传统的Unix自由选择策略之外，强制执行一个访问控制策略，以允许包含受损的系统服务。从2.6版开始，LSM框架已经以SELinux和Linux功能LSMs的形式实现，并且代码被合并到主分支Linux内核（www.kernel.org）中。

LSM框架通过在Linux内核中插入一组必要的授权钩子来实现引用监控器接口[Anderson 1972]。这些钩子定义了它们应该插入的位置以及对应模块可以实施的授权类型。将钩子放在内核本身而不是系统调用边界更具有安全性和性能上的优势。主要问题是，在一些系统中，是调用对象的名称，而不是其引用，从用户级进程传递到内核的（例如，open）。首先，系统调用行为必须将对象名解析为对象引用来对其进行授权。因为内核也会执行这个解析，所以两次解析就会造成不必要的性能开销。第二，更重要的是，对象名称和对象引用之间的映射可能在授权和内核解析之间发生更改，从而导致未经授权的访问。因此，系统调用插入据说容易受到时间检查到使用时间（Toctou）(time-of-check-to-time-of-use)攻击的影响[Bishop和Dilger，1996年]，在这种攻击情况下，在授权后，另一个非授权对象会被替换为授权对象。­

引用监控器接口的一个关键点是，它确保所有受控制的操作（即那些为了保证安全性而必须受到控制的操作）在运行之前都得到授权。虽然在内核中放置LSM引用监控器的授权钩子可以提高安全性，但是更难的是确定钩子是否有真的对所有受控操作进行了介入和授权。系统调用接口是一个很好的中介点，因为所有内核控制的操作（即访问安全敏感数据的操作）最终都必须通过这个接口。在内核内部，系统调用接口没有明显的模拟。任何内核函数都可以实现对一个或多个安全敏感数据结构的访问。因此，任何媒体接口都处于较低的抽象级别（例如，inode成员访问）。除了中介之外，还必须确保为每个安全敏感操作强制执行正确的访问控制策略（例如，写入数据）。如果执行的策略与根据该策略执行的受控制操作不匹配，则会使得执行未经授权的操作成为可能。我们认为手动验证低级中介接口的正确授权是不现实的。

近年来，许多工作已经证明了静态源代码分析对于发现安全漏洞[Engler等人2000年；Larochelle和Evans 2001年；Shankar等人2001年]，甚至在一些合理假设范围内对一些安全属性的验证【Chen和Wagner，2002年；Zhang等人。2002】可能具有一定的效果。

在其他工作中，我们开发了一种静态分析方法，能够全面验证安全敏感操作中使用的变量是否已获得授权[张等人2002]。但是，更难的是确定安全敏感操作中使用的所有变量在执行所有必要的操作时是否是确定被授权的。首先，我们需要一个模型来帮助我们预测什么时候我们已经确定了必要的授权。第二，我们需要一种分析方法来验证这个模型。我们发现一个有用的见解是，许多LSM授权钩子如果都被正确放置了，那么授权需求的不一致性通常表示这之中存在问题。我们发现的另一个想法是，授权的一致性依赖于上下文，因此我们需要某种方式来表达和测试我们期望一致授权的上下文。我们发现使用运行时分析工具更容易探索出可能的分析选项，因此我们在这里描述了这种工具的性质。然而，我们最终发现，静态分析方法也可以利用这种方法，因此我们简要描述了这个原型。因此，我们提出了通过运行时工具开发分析以及检查它们在静态分析工具中的应用的想法。­

本文提出了一种一致性分析方法，来帮助LSM社区和Linux内核开发人员验证LSM授权钩子是否能够对访问进行完全的授权。我们还介绍了这种方法的实现，使用动态和静态的分析技术。在这两种情况下，实现都由两部分组成：（1）生成包含与测量一致性相关事件的系统日志的数据收集工具；（2）识别受控操作和LSM钩子之间一致性的一致性分析工具。系统日志的生成要么通过Linux内核的运行时工具，要么通过对Linux内核源代码的静态分析来完成。运行时收集是准确的，但缺少许多可能的执行路径，因此我们实现了一个静态分析收集机制，该机制生成兼容的日志。一致性分析通过从收集的数据中识别与期望一致性授权的不一致的内容，从中发现钩子放置的错误。我们设计了一种过滤语言来描述预期的一致授权的上下文。我们的分析工具生成了两种不同的表示，从这种表示中我们来发现其中的不一致性：（1）显示受控操作在执行与其授权之间一致性的授权图，以及（2）显示对于授权一致性敏感的受控操作的属性的敏感类列表。

使用这种方法，我们在文件系统中发现了三个lsm钩子放置的bug，这些bug现在已经被修复了，以及另一个导致重大讨论的异常。虽然我们目前使用的方法还不完整（例如，可能会遗漏一些错误），但我们对使用这些工具发现错误的能力感到鼓舞。我们将在打上了LSM补丁的Linux内核版本2.4.16上演示这些工具的使用。

论文的其余部分结构如下。在第2节中，我们定义了一般的钩子放置问题。在第3节中，我们开发了一种解决一般钩子放置问题的方法。在第4节中，我们概述了运行时数据收集和一致性分析工具的实现，并讨论了所执行的分析及其结果。在第5节中，我们描述了静态分析如何用于日志收集。在第6节中，我们描述了与使用这些工具相关的问题，例如回归测试。在第7节中，我们总结并描述了未来的工作。

## 2 一般性的钩子放置问题

### 2.1 基本概念

我们在构建授权框架时确定了以下关键概念：

-安全敏感操作：这些操作会影响系统的安全性。

-受控操作：安全敏感操作的一个子集，可访问所有其他安全敏感操作。这些操作定义一个中介接口。

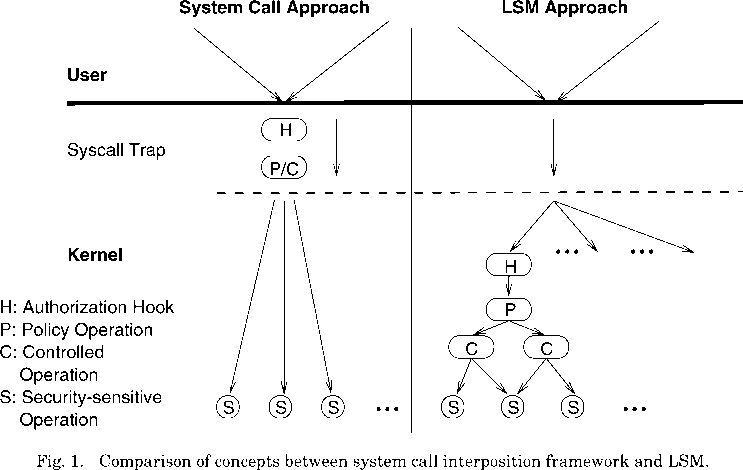
-授权钩子：系统中所具有的授权检查功能（例如，打上了LSM补丁的Linux内核）。

-策略操作：授权钩子授权的概念操作。

放置正确的授权钩子必须确保授权钩子授权了所有安全敏感操作。这些授权测试系统的授权策略是否允许请求主体执行特定的安全敏感操作。在更高级别（例如，文件读或写）表达授权策略更为方便，因此我们授权概念操作（我们称之为策略操作）而不是授权单个安全敏感操作。此外，由于安全敏感操作的数量可能很大，因此最好在调解所有安全敏感操作的接口上对它们进行一次授权。受控操作集定义了这样一个中介接口。因此，我们定义了我们的问题，用LSM授权钩子验证所有受控操作是否被授权用于预期的策略操作。­

与以往的系统调用中介机制相比，LSM的内核中介更难识别受控操作。如图1所示，系统调用接口以提供系统调用中所有安全敏感操作的中介而闻名。因此，系统调用接口既可以用作控制操作，也可以用作策略操作

当在内核中插入授权钩子时，中介接口不那么明显了，因此受控操作及其到策略操作的映射不再容易识别。例如写文件时，不是在系统调用接口上验证文件是否是写访问权限打开，而是在执行这些操作时比如目录（exec）、链接（follow link）或者最终写文件（write）的时候进行LSM授权。这种方法的好处是消除了对toctou（time-of-check-to-time-of-use）攻击的敏感性[Bishop和Dilger，1996]和冗余授权处理，但为了验证钩子的放置是否正确，需要更多的工作来识别受控操作以及它们对应的策略操作，并验证授权钩子是否正确的对它们进行了授权。­



### 2.2 需要验证的关系

图2显示了概念之间的关系。

（1）识别受控操作：找到定义中介接口的一组操作，通过该接口可以访问所有安全敏感操作。

（2）确定授权要求：对于每个受控操作，确定必须由LSM钩子授权的授权要求（即策略）。

（3）验证完整授权：对于每个受控操作，验证LSM钩子是否授权了正确的授权要求。

（4）验证钩子放置的清晰性：实施策略操作的受控操作应易于从其授权中识别钩子。否则，即使对源代码进行细微的更改，也可能导致钩子无法生效。

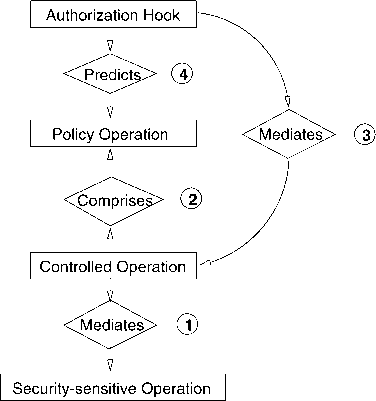


图2的授权概念之间的关系。需要验证的问题是：（1）确定控制操作; （2）确定授权的要求; （3）验证是否完全授权;（4）验证钩子放置清晰度。­

其基本思想是识别受控操作及其授权需求，然后验证授权钩子是否正确地介入了这些受控操作。首先，我们需要识别内核中具有代表性的受控操作。其次，由于受控操作的级别低于策略操作（即授权要求），我们需要一种方法来确定每个受控操作的授权要求。第三，我们需要将LSM 钩子的授权与预期的授权需求进行比较。这些任务对于内核内授权来说很复杂，因此很明显需要将任务自动化。­­­­

受控操作到授权需求的映射不一定是静态的。例如，可以对打开进行读取的文件执行许多相同的操作，就像对打开进行写入的文件执行一样。因此，上下文也是将受控操作映射到授权需求的决定因素。我们的方法必须能够有效地管理上下文依赖性，以便测试受控操作和授权需求之间的预期关系。­

### 2.3 相关工作

近日，静态分析来验证安全性已显示出希望。首先，现有的程序分析工具已被用于查找常见的安全错误，如缓冲区溢出和printf漏洞。其次，许多用于寻找安全漏洞的专业工具已经被开发出来，如xgcc，ITS4/RATS，MOPS，MAGIC等。­

静态分析工具基于编程语言的形式属性，因此它们可用于完整的分析（no false negatives，无错误否定）。但是，静态程序验证的计算成本高昂，所以常常对分析模型做一些简化。这些简化可导致更保守的分析（即，more false positives）或某些特性的抽象（即，false negatives）。此外，静态分析工具可能需要大量的代码注释工作，这是构建所需分析模型所必需的。

专业分析工具专注于特定类型的错误。 恩格勒等人。 允许扩展GCC（称为xgcc）来进行源分析，它们称之为元编译。 称为metal的规则语言用于在更高级语言中表达必要的分析注释。 metal规范不是直接注释代码，而是定义引导分析引擎的有限状态自动机。 由于规则匹配多个语句，因此减少了注释工作量。 此工具已发现各种软件错误，包括安全漏洞。

大多数专业分析工具缺乏完整性（即可能导致漏报），但MOPS专门致力于简化分析的规范和完整性。 使用MOPS，安全属性表示为有限状态自动机，程序表示为下推自动机。 未表示数据流，因此忽略别名和值关系。 但是，对于许多分析，仍然可以找到有用的bugs，并且通常可以通过其他方式显示许多数据流关系不存在。­­­

在另一项努力，我们使用一个程序分析工具，CQual ，一种静态查找LSM钩子放置错误的方法。使用GCC分析自动化CQual注释，然后，我们可以执行CQual分析，验证所有受控操作都由至少一个LSM钩子作为中介。 通常，我们还希望验证只有在检查了所需的授权钩子时才运行受控操作。CQual提供了一个可用于定义预期授权的类型点阵，尽管从概念上讲它很复杂。此外，受控操作和授权要求之间关系的上下文依赖性超出了CQual可以处理的范围。

有一个Java静态分析工具，叫 JaBA，已被用来收集对Java受控操作的实际授权。对于我们而言，这种方法有两个缺点：（1）它不分析Linux内核的C代码和（2）它没有提供有关所做的授权是否正确指导。我们实际上定义了从C到JaBA分析概念的转换，并构建了一个原型实现。关于第二点，JaBA确实提供了上下文敏感的控制流图和上下文敏感的数据流图，可用于任何分析。 因此，我们将在第5节中研究这些图形在生成分析日志中的用法。

由于使用这些方法的复杂性，我们发现运行时数据收集帮助我们快速获得准确的数据，这样我们就可以探索可能的分析选项。通过检查收集的数据，我们开发了一种一致性分析方法，我们在本文中描述了该方法，使我们能够确定对受控操作是否进行了适当授权钩子的检查。最终，该方法与我们是否对运行时收集的数据进行一致性分析或通过代码的静态分析无关。 在本文中，我们研究了两种数据收集方法。­­

另一个相关问题是系统认证。从历史上看，橙皮书被用于构建安全操作系统的指导，但现在这已被普通标准所取代。但是，认证任务是临时和费力的，并且通常不能成功地改进常用操作系统的安全性。 Gutmann在他的论文中指出，认证方法，包括形式验证工具，注定要失败，除非它们表示源代码层面的概念。 Gutmann还提倡静态和运行时分析的组合。我们使用的方法与认证的不同之处在于它检查特定错误而不是提供自上而下的保证，即整个系统满足其要求。一个有趣的研究问题是，这种检查的广度和深度是否能提供与认证相当的可信度。与认证不同，随着源代码的发展，可以保持这种信心。

## 3. 解决方案说明

我们在LSM框架运行时分析中依赖的关键是LSM授权挂钩放置在很大程度上是正确的，因此与规范不一致的情况可能表示错误。 例如，如果特定受控操作对同一系统调用的不同（不同时间或者不同上下文？）运行具有不同的授权要求，则将被视为异常。 虽然这种见解并不能保证我们找到所有LSM钩子放置错误（参见第6节），但它使我们能够识别一些错误，并且已经成为工具开发的有价值的指南。

在下面的所有讨论中，我们使用以下假设。 首先，我们相信一个假设，即受控操作中的对象在内核中以类型安全的方式处理。 这不会使我们发现的任何错误无效，但也可能存在其他错误。 其次，我们假设对授权数据类型对象的访问定义了受控操作集（即，中介接口）。 这些数据类型对应于系统调用概念（例如，文件，inode，套接字，skbuffs，ipc消息队列等）。 访问内核数据需要通过这些数据结构。 虽然我们尚未明确验证这一点，但我们已在其他地方进行了更详细的分析。­­

### 3.1 授权一致性

我们首先定义一个受控操作和一组授权要求之间的一致性。­

定义 1 （授权一致性）。 如果在执行受控操作时，受控操作和一组授权要求（即，被授权的策略操作）之间的关系是一致的，则调用与这些授权要求相关联的授权钩子。

我们发现这种形式的一致性并非绝对必要。 受控操作的执行可以在具有不同授权的不同系统调用的上下文中发生。 显然，在这种情况下，授权要求的满足情况将是不同的。­

因此，有必要定义期望一致性的上下文。 一般来说，上下文可以是任意的，但我们的经验是三种类型的上下文很重要：⑴系统调用;⑵具有特定输入的系统调用（例如，标志）; （3）一组特定的受控操作。 在第一种情况下，授权用于整个系统调用（例如，fcntl）。 此类授权适用于系统调用中的所有受控操作。 在第二种情况下，授权依赖于系统调用的一些参数，通常是标志（例如，打开以供读取）。 因此，一些系统调用属于一种情况，一些系统调用属于另一种。 在第三种情况下，受控操作集的出现与它们出现的系统调用无关，需要特定的授权（例如，与访问集所有者字段相关的操作）。 在这些情况下，一致性会忽略系统调用信息。

定义2（执行上下文）。 执行上下文定义了一组执行路径。 执行上下文可以通过（1）系统调用（所有执行）; （2）具有特定参数值（或值的范围）的系统调用; （3）一组受控操作（包括它们的所有路径）来定义。 其他上下文定义是可能的。

我们的解决方案必须支持我们期望持久授权的上下文的描述。 通常，静态分析中的上下文相关数据流指的是区分相同功能的不同输入。 在这种情况下，上下文敏感度要窄得多（仅取决于系统调用），或者可能忽略剩余上下文的大部分（对于受控操作集）。 这种分析需要对静态分析器进行大量注释，并且可能依赖于静态分析器之外的变量（例如，用户提供的标志）。 例如，JaBA完全忽略了原始类型的值，但显然那些可以影响分析。

### 3.2 授权一致性水平

执行上下文通常由许多受控操作组成，因此以相同方式聚合受控操作是有帮助的。

表一，授权一致性水平：名称和授权影响

|  |  |
| --- | --- |
| 级别 | 授权 |
| 系统调用 | 在系统调用的所有控制操作 |
| 系统调用输入 | 在同一个系统调用相同的输入所有控制操作 |
| 数据类型 | 相同的数据类型的对象，所有控制操作 |
| 对象 | 同一对象上的所有控制操作 |
| 成员 | 上相同的数据类型的所有控制操作，访问同一部件，用相同的操作 |
| 函数 | 在相同功能的所有成员一样控制操作 |
| Intrafunction | 同样的受控操作实例 |
| 路径 | 相同的执行路径相同的控制操作实例 |

例如，如果上下文中的所有受控操作具有相同的授权，那么我们可以查看相对于上下文的一致性而不是单个操作。

我们发现，我们可以描述每个受控操作与在特定上下文中通过一组离散值（称为一致性级别）执行时调用的授权钩子之间的一致性。 此外，一致性级别形成如下的总顺序。

定义 3 （一致性级别总顺序）。如果两个不同的受控操作对于级别i的相同值是授权一致的，则它们对于级别j的任何值是授权一致的，其中在一致性级别总顺序中i> j（参见表I）。

如果对同一对象执行了两个不同的受控操作，但它们具有一致的授权，那么这些操作的成员和访问的值不会影响一致性。 例如，如果对特定对象的所有受控操作具有相同的授权要求，则成员访问权限无关紧要。 表I列出了离散一致性级别。 我们将这组级别统称为授权一致性级别。 这些级别包括受控操作执行的各个方面，包括执行它的上下文（系统调用，系统调用输入，函数，函数中的位置，受控操作的路径），执行它的对象（数据类型） 和对象），以及执行的操作（成员和访问）。

一致性级别将受控操作聚合到一致性类中，其中所有受控操作都具有在给定当前放置时调用的相同授权钩子。

定义 4 （一致性类）。两个不同的受控操作属于执行上下文的相同一致性类，如果它们在每次在该上下文中执行时都具有相同的授权钩子。

### 3.3 授权一致性影响

通过授权一致性对受控操作进行分类将受控操作分为两类：⑴已知异常和（2）授权要求需要验证的一致性类。在第一种情况下，我们认为某些授权一致性级别是非法的。我们在下面为这些情况定义不变量。在第二种情况下，我们必须确定执行上下文中每个受控操作的最大一致性级别是否表示可接受的授权要求。例如，如果受控操作属于对象一致性级别的一组受控操作，则表示对象上的所有受控操作都检查了相同的授权。然后，确定这是否正确是手动任务。但是，一致性聚合的数量表示将受控操作划分为具有相同授权的最大大小的类。这些类可以验证授权要求和识别异常分类。

#### 3.1.1 反常现象

表I中最后两行，内部函数和路径，级别的授权一致性始终被认为是异常的。 这些类型的灵敏度意味着函数（函数内）中的执行路径（路径）或位置决定了同一成员的特定受控操作的授权要求。

以下不变量正式表达了我们的路径不一致性。

定义5（路径不一致不变）。

Vc1，c2 e C, e1,e2 e E, (c1 = c2)八（e1 = e2) — ^(c1, e1) = E(c2, e2) (1)

该不变量表明相同的受控操作（c1 = c2）在同一事件中运行（e1 =由系统调用及其输入定义的e2）必须具有相同的授权要求（由函数R定义）。 也就是说，事件内的执行路径不会影响受控操作的授权要求。­

*同样，我们为函数内不一致定义了一个不变量。*

定义6（Intrafunction Inconsistency Invariant，功能不一致不变量）。

Vc1,c2 e C, e1,e2 e E, (F(d) = F(c2))八

(M(c1) = M(c2))八（e1 = e2) — K(c1,e1) = E(c2,e2) (2)

在这种情况下，同一函数中的两个受控操作（由函数F计算）并且进行相同的成员访问（由函数M计算）必须具有相同的授权要求R.

#### 3.3.2 授权一致性类。

授权一致性类。 对于其他情况，我们不能轻易将它们识别为错误。 相反，我们将受控操作划分为其授权一致性类，并确定其授权要求是否正确。

授权一致性类计算如下。 对于从最高（系统调用）开始的每个一致性级别，我们将受控操作划分为一致性类，其中所有受控操作对于一致性级别具有相同的值，然后我们测试该类是否也具有相同的授权。 如果没有，那么我们尝试下一个较低级别并根据两个级别进行分区并再次测试。 这种方法重复进行，直到我们将每个受控操作分配给一致性类。

分类由一致性级别定义。 对于系统调用级别，系统调用的所有受控操作都在一个类中。 对于系统调用输入，将汇总同一系统调用和相同类型输入的所有受控操作（请参阅第3.4节）。 对于数据类型级别，受控操作按系统调用，输入和操作对象的数据类型进行分类。 因此，在分析的每个步骤中创建连续更精细的分区。

如果分类是该类中所有受控操作具有相同授权的第一级，则分类成功（即，x是一致的，其中x是级别）。 请注意，具有相同授权一致性的其他类将被聚合以形成最大大小的类。 创建类后，验证每个类的授权是否正确是一个手动过程。 对于文件系统，类的数量足够小，手动验证是实用的。

例如，考虑读取系统调用。 文件操作是数据类型一致的，因为文件对象上的所有受控操作都有权读取。 手动验证涉及检查文件的读取权限是否足够。 由于读取授权也适用于文件的inode，因此我们将文件的inode标记为已授权读取。 但是，在分类之后，未授权一个inode的受控操作。 它位于不同的对象上，因此inode操作可能是对象一致的。 这是对文件的目录inode的操作，以确定是否应该发送一个读取目录的信号。

除了完全缺失授权之外，识别错误的最常见方式是找到执行重要共同操作的两个分类（即，具有不同授权的两个聚合）。 这种情况发生在fcntl中，其中两个不同的分类（基于不同的系统调用输入）在同一f\_owner字段上运行（参见第4.2.4节）。

### 3.4 必要的数据采集

通过记录系统调用入口/出口/参数，函数入口/出口，受控操作（即对象，数据类型，成员和操作）和授权，我们收集一致性级别的所有必要值。 可以轻松记录所有信息，但需要进一步分析有意义的对象标识符和系统调用输入更改。

在执行期间，通过函数指针引用对象，但这不一定是对象的充分标识。 例如，inode具有在授权中使用的持久标识符（即，设备，inode号）。 因此，对于每种数据类型，我们定义了用于计算其对象标识符的特定方法。这些标识符用于确定对象上的所有操作和授权。

表二。日志记录类型

|  |  |
| --- | --- |
| 记录类型 | 数据 |
| 受控操作  授权  函数入口  函数退出 | 上下文ID控制操作ID OID  上下文ID授权ID OID  上下文ID指令地址  上下文ID |

在整个系统调用实例中, 我们假定在同一变量中使用的对象具有相同的授权要求。为了模拟这一点, 我们使用了第一个受控操作, 其中对象显示为标识符。如果首先在同一受控操作中访问两个对象, 则必须将其分配给同一变量 (因为该变量在两个受控操作中是相同的)。但是, 不同的执行路径可能会导致首先在不同的受控操作中使用相同的变量。但是, 具有相同授权要求的类的聚合将合并这些情况, 因此这一假设已被证明是有效的。

系统调用参数几乎在每次调用时都会更改, 但只有少数参数真正影响授权 (例如, 打开时的访问标志)。因此, 我们收集参数, 但只使用我们发现有影响的授权要求来进行分区的参数。我们检查过的系统调用中只有少数基于其输入参数的授权, 如开放、ioctl 和 fcntl。由于根据不同的输入使用不同的授权, 因此这些系统调用更为复杂, 因此更容易出错。

## 4.运行时实现

完整授权的运行时分析包括两个步骤: 内核执行日志的生成及其离线分析以确保一致性。本节介绍创建执行日志的工具的实现、用于准备和显示分析数据的日志筛选工具的实现以及我们的分析结果。

### 4.1 收集运行时信息

#### 4.1.1 日志内容

表二显示了运行时分析期间收集的信息。受控制的操作由元组标识 (指令指针、对象类型、成员、访问)。将受控操作 id 分配给每个唯一组合。授权由 (lsm 钩子、策略操作) 唯一标识。与受控操作一样, 为每个操作分配了唯一的授权 id。函数入口和出口也会被记录。函数项地址唯一标识函数。

对于执行的每个受控操作或授权, 日志必须包含所涉及对象的标识 (例如 inode)。对象实体 (oid) 是根据对象类型定义的, 例如, inodes 是由设备 id 标识的,节点内的编号, 而任务则由进程 id 标识。oid 在上下文中只需要是唯一的。

我们使用上下文的概念来表示内核事件的处理 (例如, 系统调用)。授权显然仅在执行它们的上下文中有效, 因此, 日志条目还必须包括受控操作和授权的上下文。

#### 4.1.2 集合概述

图3显示了该工具的概述。日志的创建涉及三个阶段: 必须生成所需的信息, 必须收集这些信息, 并且必须将其写入日志。

信息是以三种不同的方式生成的。首先, 授权信息由 lsm 挂钩生成。其次, 通过使用 gcc 的修改版本编译内核来生成受控操作的控制操作详细信息, 并在所有此类操作之前使用对处理程序函数的调用来处理内核。通过在编译时检测内核, 也会生成控制流信息。第三, 上下文信息是通过在内核中放置断点生成的。以下各节将更详细地讨论这三种方法。

第四, 加载内核模块以接收图3所示的信息。这些模块执行粗粒度筛选, 并将信息排列为正确的格式, 然后再将记录传递到日志记录模块。日志记录模块将上下文 id 分配给传入记录, 并将信息写入缓冲区。

#### 4.1.3 授权信息

lsm 修补程序已经提供了日志授权信息, 因此几乎不需要额外的实现。授权筛选器只是一个 lsm 模块, 它为每个授权添加一个日志条目。这些日志条目标识已执行的授权（例如，RMDIR PARENT、RMDIR TARGET）和授权的对象。

#### 4.1.4受控操作

为了记录受控操作，我们首先必须在内核中定位受控操作，然后提供用于检测这些操作的执行的机制。

识别内核中的受控操作需要源分析。我们选择通过分析GCC的中间树表示来识别受控操作，而不是直接进行源代码分析（这很困难）。由于Linux依赖于GCC扩展，所以源代码分析需要使用GCC解析器，因此利用它已经构建的树似乎是合乎逻辑的。为了识别受控操作，我们遍历树寻找访问中介数据类型的成员的表达式。2当检测到受控操作时，我们在表达式存在的语句之前插入对包含对象、类型、成员和访问的function\_control\_op的调用。如果表达式是循环的条件语句，则在循环之前和每次迭代结束时插入调用。此调用包含标识受控操作并允许处理程序提取对象标识所需的所有信息。

一些访问会导致此方法的问题。首先，可以通过获取成员的地址、将其存储到指针并通过指针更改成员来修改结构成员。由于初始访问是对指针变量的读取，因此我们可能会错过后续的写入。我们不是执行更广泛的源分析来识别这些情况，而是简单地检测何时出现混淆。第二，当受控数据结构被转换为非受控t时，我们也可能错过对受控数据结构的访问。YPE。这也被检测到。我们的初步分析表明，这些情况以少量（虽然是第一次，但次数很多）的方式发生，因此它们可以作为特殊情况处理。

#### 4.1.5控制流程

控制流信息是通过使用GCC-3.0提供的finstrument-functions开关编译内核生成的。此选项使编译器在每个函数的入口和出口插入对处理程序函数的调用。这些处理程序函数然后将信息传递给适当的模块。

#### 4.1.6上下文信息

由于内核中随时可能存在多个执行上下文，所以所有日志条目都必须包含上下文ID，因此分析可以判断哪些条目彼此相关。不幸的是，没有可用的唯一标识单个执行上下文的密钥，我们必须选择非唯一密钥，并设计一种方法来区分具有相同密钥的上下文。

我们选择当前内核堆栈的底部作为非唯一密钥，因为我们需要一个至少是并发活动执行中唯一的密钥，并且该属性对于堆栈来说似乎不可能被违反。尽管内核堆栈在并发活动执行中是唯一的，但是由于三个原因，每个上下文内核堆栈都不是唯一的：来自同一进程的所有系统调用都使用相同的内核堆栈，一旦进程死亡，内核堆栈可能重新分配给新进程，并且用它们中断的进程的内核堆栈来执行中断。这里的关键特性是，尽管上下文键不是唯一的，但是具有相同键的上下文从不交错。因此，通过记录上下文的开始和结束（以及相关的键），我们可以明确地向上下文分配日志条目。

幸运的是，只有少数几个点可以开始上下文（全部位于entry.S中），并且上下文可以结束的位置数量大致相等。退出系统调用是一个例外情况，因为它从不返回，因此，do\_exit()中的schedule()调用也被标识为上下文退出点。由于入口/出口点的数量是可管理的，因此我们为每个内核版本手动标识它们的位置，并在运行时自动插入它们。为了在运行时收集此信息，上下文过滤器在所有入口和出口点将断点指令插入内核的（内存映像）中。当执行断点时，上下文过滤器创建包含上下文键的日志条目，以及这是上下文的开始还是结束。

#### 4.1.7性能

我们做了一个简单的性能检查以确定仪表化内核中的性能下降。在未修改的Linux内核上，LMBench用于“快速基准测试”花费了3分钟4秒。仪器内核直径3min24μm，退化率略高于10%。我们认为，这种开销对于深入分析来说是可以接受的。回想一下，内核用于分析，并且以后使用内核不需要挂钩。

除此之外，我们在20个系统调用中抽取了1个。这是为了保持日志增长率低于磁盘吞吐率。由于这些基准测试多次执行相同的系统调用，因此我们没有注意到“丢失”任何与安全相关的信息。如果必要，可以设计用于确定何时删除日志条目的策略。

### 4.2日志分析

我们还构建了一个工具，能够进行日志分析，以识别授权需求中的一致性，如3.2节所述。该工具能够指定用于提取所需日志条目的规则，称为日志-过滤规则，并根据提取的条目计算授权一致性。我们以两种方式显示一致性结果：（1）表示每个授权和控制操作之间的一致性的授权图和（2）表示通过授权和一致性级别聚合的受控操作的一致性类列表。

#### 4.2.1 日志筛选规则

日志过滤工具以执行日志和一组过滤规则作为输入，并输出匹配规则的日志条目。规则语言目前水平相当低，因为我们更关心的是证明其可行性，以及创建良好的高级规则语言。然而，我们演示了规则语言，以了解可能的分析类型。

规则库由一组定义匹配需求的规则定义。规则包括（1）索引；（2）依赖项规范；（3）一组语句。索引标识规则库中的规则。依赖关系通过索引声明与其他规则的关系。我们可以声明一个规则只能匹配另一个规则（D，i）也匹配的条目，其中i是另一个规则的索引。此外，我们可以声明一个规则不包括由另一个规则i匹配的条目的依赖性，如(N，i)。最后，语句描述条目的匹配条件。通过识别条目类型（id类型），然后匹配类型-specic级别，可以对这些进行指定。条目类型包括：事件（CONTEXT）、授权（SEC\_CHK）、函数（FUNC）和受控操作（CNTL\_OP）。

图4显示了一些示例规则。路径一致性规则在运行指定地址的受控操作时，将读取系统调用上下文中的所有授权。rst行收集读取系统调用（即系统调用的开始）的所有上下文条目。第二行收集指定位置上受控操作的所有条目。(D,1)表示该语句依赖于语句1，因此将只收集读取系统调用上下文中的条目。第三行收集读取系统调用上下文中的所有授权。在这种情况下，此受控操作的每次执行都应该具有相同的授权，或者存在违反路径不一致性不变量的情况，该路径不一致性不变量禁止受控操作具有多组合法授权。

函数一致性规则在读系统调用上下文中收集“读 节点成员”的所有授权和受控操作。在第二行上指定(D,1)意味着将提取读系统调用中此类型的所有受控操作。如果与此受控操作关联的授权不相同，则成员访问与其位置一致。

系统调用输入一致规则收集每个打开的系统调用中的所有日志条目，以便进行只读访问。打开的系统调用的授权依赖于文件被打开的访问，所以打开是系统调用输入一致的。此外，我们还在这个规则中显示了一个负面过滤器，它消除了path\_walk函数范围内的所有条目。文件查找的授权，包括任何链接遍历，可以与授权打开该文件的授权分开。这种过滤功能使我们能够灵活地选择分析范围。

所面临的挑战是编写日志过滤规则，以表示可以执行有意义的分析的情况。通常，我们希望日志过滤规则描述这样一种情况:结果符合我们的期望的一致性类。我们发现，一个乐观的一致性初始假设，然后进行细化，对于设计这样的规则是有效的。

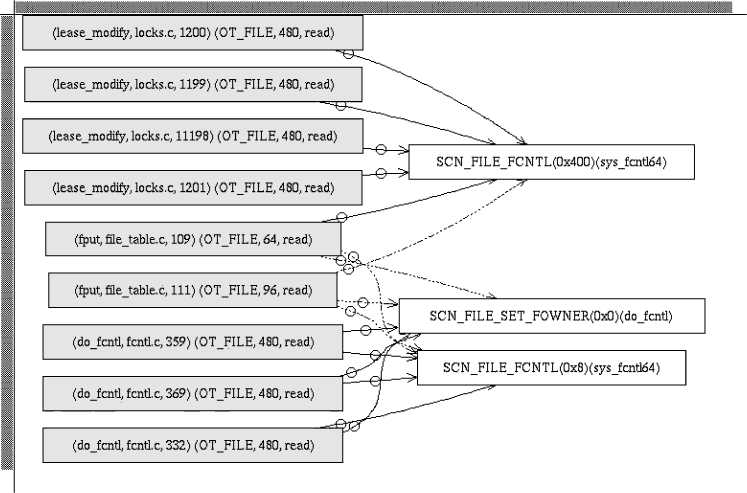
我们从假设所有受控操作的最高级别一致性开始，系统调用一致性。如果系统调用执行中的所有受控操作具有相同的授权(即，则我们只需验证授权是否正确即可。如果没有，我们将检查是否可以在下一个一致性级别(系统调用输入)上编写规则。如果输入没有可识别的效果，或者无法解决与我们期望的一致性(即。然后我们进入下一个层次，数据类型一致性，等等。通常，在不更改规则的情况下，有效的分析可以降低一致性级别。我们只需要一种方法来查看具有相同一致性级别的聚合，例如在数据类型一致的情况下，将每种数据类型映射到其授权。我们使用一致性类列表来查看多个一致性类(参见4.2.3节)。

#### 4.2.2 图形化日志分析

分析工具还可以生成图形，以支持对经过过滤的数据进行可视化分析。使用这些图，可以通过检查验证授权一致性，如下所述。授权图由过滤后的日志中的两组节点组成:(1)受控操作和(2)所做的授权。从每个受控制的操作到运行时已满足的授权，都要绘制边缘。有两种类型的边:(1)总是边表示每次运行受控操作时都满足关联授权;(2)有时边表示在运行受控操作时至少满足一次关联授权。

始终边缘(以及缺少边缘)意味着授权与低层不一致。有时边表示不一致。如果缺少预期的边缘，则表示缺少授权。

图5显示了一个授权图示例。示例图使用dotty图形可视化工具[Koutsofios和North]显示。在本例中，授权图显示了两种类型的fcntl调用的受控操作和授权:(1)fcntl(fd, F\_SETOWN, pid\_owner)和



(2) fcnt1 (fd, F-SETLEASE, F-UNLCK)。受控操作节点包括位置(函数名、文件名、行号)和操作(数据类型、成员偏移量、操作类型)信息。授权节点包括authorization、命令和包含授权的函数。总是用实线表示边，有时用虚线表示边。如果受控操作和授权之间不存在边界，则永远不会为该操作执行该授权。

通过形象化的分析，我们可以分辨出当前图中是否包含在3.3节中描述的常量。这样，这样的话，有时候，fput和它的授权之间的关系可能表示存在问题。针对相同域的不同授权集合也可能表示出现了问题。这就需要人工检查来分辨到底是系统出了错误，还是一个合法的一致。

#### 4.2.3 一致性等级列表

一致性等级列表展示了一种通过一致性等级及其关联的权限要求的划分。这种划分是通过3.3节介绍的算法实现的。与授权图相比，授权等级列表为授权结果提供了一种新的视角。授权图显示了每个受控制操作与授权之间的关系，一致性等级列表使得数据中存在的不同授权情况变得更加明显。而且，由于它是文本性的，所以更容易进行回归测试[Jaeger等人的一致性类列表是更容易回归测试使用。2002]。

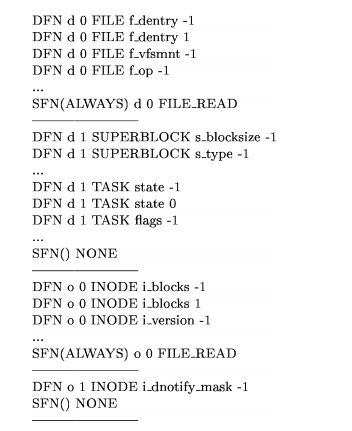
图6显示了如何把系统读调用的受控操作进行一致性等级划分，这个划分在3.3节中已经举过例子了。像那里描述的那样，一致性等级列表中数据类型的两个等级：一个是没有授权的任务和超级块，另一个是对文件的读权限。在对象一致性方面，有两个等级：一个是对inode的读操作，另一个没有读目录的权限。基本上，我们希望如果给与当前任务，文件的目录或者文件的超级块授予读权限，那么将会导致所有的受控操作拥有相同的权限（也就是系统调用一致性）。

图6.一致性类列表用于读系统调用具有以下字段：（1）入口类型（DFN或SFN）; （2）稠度（d为数据类型以及o为对象）; （3）类号码; （4）数据类型; （5）部件; （6）访问标识符。

我们大多数经验都是与文件系统相关的，同时我们也测试了任务相关的授权。由于大多数对象只有一个或没有权限，所以一致性等级列表不会太过复杂。取消链接系统调用是为数不多的一个对象含有多个权限的个例之一。使用一致性等级列表很容易看出目录inode拥有三个授权（执行、写和取消链接目录），而正在被删除的inode有一个（取消链接文件），因为它们是对象一致的且被放在不同的等级中。因此，对我们已经测试过的文件系统和任务操作来说，授权图和一致性等级列表已经足够去验证授权了。

* fcntl函数的F-SETOWN只有场f\_owner

1. =（+，ID\_TYPE，CONTEXT）（+，di\_cfm\_eax，的fcntl）（+，co\_ecx，F\_SETOWN）
2. （d，1）=（+，ID\_TYPE，SEC-CHK）
3. （d，1）=（+，ID\_TYPE，CNTL\_OP）（+，di\_dfm\_member，f.owner）

* 的fcntl适用于F ^ SETLEASE只有场f.owner

1. =（+，ID\_TYPE，CONTEXT）（+，二cfnueax，的fcntl）

（+，co\_ecx，F\_SETLEASE）（+，co\_edx，F\_UNLCK）

1. （d，4）=（+，ID\_TYPE，SEC\_CHK）
2. （d，4）= C +，ID\_TYPE＃CNTL\_OP）（+，di\_dfm\_member，F-所有者）

图7.规则异常发现歼所有者。

#### 4.2.4 样本分析

我们简要演示了对我们发现异常的样本分析。虽然发现异常的方法是与实际发现异常同时开发的，但是我们使用的方法与描述的方法大致相同，尽管有些方法不是自动的。这种异常发生在fcntl系统调用中。fcntl的一致性等级列表表明，fcntl的一致性是系统调用输入一致的。cmd和arg参数对fcntl的值可以更改所需的授权。我们使用授权图来查看不同输入下的授权，因为更容易看到粗粒度的问题（很多边缘问题会发生）。

图7包含两套规则：(1)一个收集所有授权和可控操作的文件结构字段f\_owner fcntl (fd, FSETOWN pid\_owner)的系统调用,和(2)一个收集所有授权信息与可控操作字段f\_ownerin fcntl (fd, F-SETLEASE F UNLCK)的系统调用。注意，这与用于生成图5中的图形使用相同的规则（fput 控制操作除外）。­

在图5中，我们看到，一些控制操作的被授权为 的fcntl 和 set\_fowner 授权和一些只授权 的fcntl。尽管这是一个事实，即控制操作访问相同的领域，f\_owner （偏移480）。鉴于这种异常，我们检查了内核源代码，以确定是否可以利用这种异常。我们将在下一节讨论这种分析的结果。

### 4.3 结果

我们将2001年12月10日的LSM补丁应用到Linux 2.4.16源代码中，并使用我们修改过的GCC-3.0.3版本编译内核，以创建要分析的执行日志，我们在LMBench的三个并行实例中执行。SAINT漏洞工具(www.wwdsi.com/saint/)，这是一个内核编译器，以及我们在猜测并检查异常时编写的一些测试程序。由于运行时分析的有效性依赖于运行足够多的代码，因此必须开发涵盖足够多有趣路径的基准测试。例如，LMBench只运行大约20%的内核代码。­

我们进行内核测试，以收集对主要内核数据结构(文件、inode、超块、任务、套接字和skbuffs)的受控操作。到目前为止，我们只对文件系统授权做了详细的分析。以及对任务授权的初步分析。由于对文件系统的理解相当透彻，所以我们没有预期会出现大量异常，但是我们还是发现了一些异常。

**-****成员一致（多系统调用）**：etgroups16是几个向后兼容ABI的16位任务操作之一，例如setuid16和setchown16。这些操作通常将其16位值转换为32位值，并调用包含授权的当前版本。但是，由于setgroups16设置了一个数组，因此不进行数组转换就更容易，所以不调用当前版本(包含一个钩子)。我们发现setgroups16函数中没有授权钩子，所以我们可以在没有授权的情况下重置任务组。注意，在libc的当前版本中没有setgroups16调用，因此我们不得不编写一个汇编程序来执行这个漏洞。­­

-成员一致（单系统调用）： 成员一致(单个系统调用):f\_owner的pid成员告诉内核关于这个文件上的IO向哪个进程发送信号。如果用户试图通过fcntl(fd, F-SETOWN, pid\_owner)函数直接设置该字段，则该字段由file\_ops->set\_fowner授权。但是，如果用户通过fcnt1(fd、F- setlease、F UNLCK)函数删除文件的租约，则所有者在未执行授权的情况下被设置为0。此外，进程可以在没有执行授权的情况下将通用TUN设备(驱动程序/网络/tun.c)的所有者设置为自己。为了实现这一点，流程在一个开放的、附加的TUN设备一致性(单个系统调用)上调用ioct1(fd、F-SETFL、FASYNC)函数。

-成员一致（单系统调用）： 在我们进行上述filp.f\_owner的一致性检测时，我们发现，获得的filp.f\_owner.signum（应该IO完成时发送的信号）可以在没有授权的情况下进行设置。

-系统调用一致（多系统调用）： 系统调用一致(多个系统调用):在每次读系统调用开始时执行读授权。此授权是必需的，因为由于进程更改其安全级别、文件更改其安全级别、新进程使用的文件或安全策略中的更改，在最初打开文件时执行的授权可能不再有效。但是，在内存映射文件的页面错误期间，不会执行此授权。因此，一旦进程拥有内存映射文件，它就可以继续读取该文件，而不必考虑对安全级别或安全策略的更改。

我们与LSM社区进行了讨论，除了读取内存映射文件的异常外，我们对所有异常都进行了修补。社区决定，需要读授权的文件不能被内存映射。我们很受鼓舞，因为我们能够帮助发现和修复钩子的放置问题。

## 5 研究静态分析

我们现在已经使用JaBA statid分析工具实现了一个静态分析原型，该工具收集Linux ker nel中相关执行路径的日志。对于静态分析，可以比运行时分析更直接地评估完整的内核执行路径覆盖率。不幸的是，ker nel执行路径的数量通常是指数级的，因此模型必须设计成尽可能多地删除不相关路径，并进行优化，以便尽可能多地评估路径。静态生成的日志可以作为运行时生成的日志输入到4.2节中描述的分析工具中。­­­

### 5.1 JaBA分析工具

JaBA是基于指针的数据流分析的流敏感静态分析工具[Koved等，2002]。JaBA具有处理Java类字节码的基本函数，用于为变量和对象字段构建过程内基本块图、过程间控制流图和数据流图。我们将此基本分析信息作为输入，为日志收集构建进一步的静态分析。这些分析是过程性的，而不是声明性分析语言，如MC [Engler et al. 20001]。然后我们按照章节4.2的描述处理这些日志。­

JaBA中的上下文敏感控制流意味着可以区分进入同一功能的不同路径，从而可以区分进入同一功能的不同输入对象，并准确跟踪它们与授权的关系。

JaBA中的上下文敏感数据流跟踪程序中分配站点的数据流。对于Linux内核，大多数对象(例如文件和inode)使用相同的分配站点，因此在上下文不敏感的数据流中，该类型的所有操作都将使用相同的对象。JaBA通过调用站点将对象区分为创建对象(或任意级别)的函数，从而支持上下文敏感性的定义。但是，创建的对象越多，分析的成本就越高。

由于Linux内核显然不是Java的，所以我们编写了从C到Java字节码的转换[Zhang et al.]。转换只保留与JaBA分析相关的Linux内核语义。例如，指向C结构的指针被转换为大小为1的数组，数组中包含相应类型的Java对象。另外，非类型安全指针置换被转换为wildPtr对象。

### 5.2 日志生成过程

JaBA日志分析过程使用两个输入:(1)根控制流图节点，通常是系统调用(例如sys\_open)的函数;(2)正在分析其控制操作的结构的数据类型。所有JaBA分析仅限于函数(即…可从根节点访问的CFG节点)。日志生成返回由运行时数据收集工具收集的表单的日志。

使用JaBA生成日志的过程如下：

1. 转储ID命名映射功能，授权，并通过日志分析工具的使用来控制操作。
2. 收集指定数据类型对象的所有受控操作。这些数据类型被指定为分析过程的输入，并且集合的范围受根函数的限制。
3. 收集指定的数据类型的对象的所有授权。
4. 为每个路径定义一个一致性上下文，当前定义为系统调用中使用的每个单独对象实例中的每个路径。由于系统调用标志值的关系，此上下文还可能包含条件分支限制，来模拟转换工作。­
5. 使用路径标识节点，这些路径将导致受控操作或授权。所有其他节点都与分析无关。
6. 生成所有相关的过程内路径(见下文)。
7. 从生成过程内的路径序列过程间的路径。
8. 重复步骤4，直到没有新的上下文需要检查。

第一步通过提供从数值表示(如函数CFG节点编号)到名称(如函数名称)的映射，使日志分析工具能够正确显示来自JaBA生成的日志的结果。­

第二步收集感兴趣的数据类型的所有受控操作，正如在运行时情况中所做的那样。在这个概念验证中，我们到目前为止只收集写访问:需要收集读访问来查找一些错误(例如从f-owner字段读取以更新其成员)，因此需要扩展到读访问。虽然增加更多的访问只会增加分析的复杂性，但是我们认为额外的复杂性并不像第6节中讨论的那样重要。­

第三步收集所感兴趣的数据类型的授权。在运行时系统中，我们必须手动列出所有被认为是由每个LSM钩子授权的对象。在这里，我们希望使用数据流更准确地描述第6节中讨论的授权对象。但目前只收集直接授权。我们定义了一种从授权中推断授权对象的机制，但是这个步骤目前没有运行时分析中广泛。。

第四步定义了一组分析上下文。在运行时工具中，系统调用的执行定义了上下文。在静态分析中，系统调用定义了许多路径，因此我们需要标识值得收集的路径上下文。我们已经确定了一些方法来减少值得考虑的程序性路径的数量(见5.3节)，但是我们仍然发现需要进一步减少以便及时分析。我们遵循xgcc项目的领导，其中分析自动机处理是依赖于变量的[Hallem et al. 2002]。

因为JaBA跟踪对象而不是变量，所以对象定义上下文。也就是说，对于每个系统调用，将标识一个对象，并且只有对该对象的操作和授权才定义上下文。该对象的路径分析完成后，系统调用中的另一个对象被选中，所有相关上下文(即对象的路径)被记录。这样重复直到所有相关的对象(即分析每个系统调用中的目标数据类型的对象，并收集它们的日志。­

第五步进行上下文描述，确定值得分析的CFG节点。如果一个节点被移除，那么它就值得分析。删除它的任何后代将导致丢失受控操作或授权日志条目。因此，调用任何具有与当前上下文相关的受控操作或授权的函数的函数都值得日志记录。。

第六步为值得分析的节点创建所有相关的过程内路径。这是分析中的关键步骤，将在5.3节中详细讨论。。

第七步将过程内路径合并到单个日志中。从逻辑上讲，它按执行顺序列举了过程内日志的所有组合。实际上，日志生成是为了从为下一个组合提供新的过程内路径的函数构建新日志而优化的，而不是从头构建每个路径。日志是通过为那些在下一次传递中将在这些路径中调用的函数收集一系列过程内路径来构建的，选择日志中具有多个过程内路径的最后一个函数来定义下一个组合。到那一点的日志保持不变，因此日志生成只在新路径开始时进行。­

4.2节中描述的日志分析之前执行步骤(7)中生成的日志。唯一不同的日志条目有:(1)JaBA种标识符用于对象而不是特定对象标识符定义运行时和(2)不被JaBA原始参数,那么系统调用标记标识符必须由一个条件分支,我们存储在系统调用的参数位置第一目前四个条件。

由于JABA跟踪对象，而不是变量但是对象定义上下文。也就是说，每个系统调用，一个对象被定义，只有在该对象上的操作和授权决定上下文。对这个对象和调用这个对象的调用的路径进行分析。这重复进行，直到所有相关对象（即，目标数据类型的对象）在每个系统调用进行了分析和他们的日志收集。­

第五步骤在所述上下文描述，并确定是值得分析的CFG节点。如果删除节点会导致节点的操作和授权受损那么这就是有价值的节点。因此，调用具有受控的相对于当前上下文对象的操作或授权的任何函数的功能是值得记录的。

第六步创建的节点值得分析的所有相关过程内路径。这是在分析的关键步骤，并且其详细描述在第5.3节中讨论。

第七步结合过程内的路径成单独的日志。从逻辑上讲，它列举ofintraprocedural在执行顺序记录所有组合。在实际中，日志生成被优化，以从提供新过程内路径的下一个组合，而不是从头开始构建每个路径中的功能建立新的日志。日志由收集的过程内路径序列为那些会在这些路径中调用的函数建。在接下来的通过，即具有多个过程内路径的日志中的最后一个函数被选择为定义下一个组合。日志到这一点是一样的，所以日志生成由新路径的开始只是做了。­

在第4.2节中描述的日志分析如之前在步骤（7）中生成的日志进行。在日志条目的唯一区别是：（1）JABA对象标识符用于对象，而不是特定于对象的标识符作为在运行时和（2）原语参数不被JABA捕获定义的，所以系统调用标记必须表示由我们在目前的前四个条件句的系统调用的参数的位置存储条件分支标识符。

### 5.3 过程内分析

过程内分析定义每个节点相关的路径的数量。由于节点的路径数是指数级的所以我们要进可能的减少。我们同时执行节点级和条件语句的分析，以尽量消除冗余路径。只有linux文件系统需要3条路径，其他只需要一条路径。

我们正在寻找的主要事情是操作和授权之间的关系。因此，节点中具有相同关系所有路径可以被组合成一个。因为授权的数量比控制操作的数目小得多，所以我们以授权为主：如果在一个节点的所有路径具有相同的授权，那么的控制操作可以组合成一个路径。由于授权控制的操作之前，通常会出现，这种优化是非常有效的。

在功能不同的路径是由它的条件语句的条件规定，每一个条件语句的分支都会生成一个路径。如果条件语句生成的路径并不不影响控制操作与授权之间的关系，那么这些路径不必单独考虑。我们称这个过程*合并。* 在大多数函数中，这导致在路径数一个显著减少。

1. 节点以拓扑顺序进行排序与CFG保持第一。
2. 确定节点调用有任何授权的节点。如果没有，压缩控制操作为一个路径。
3. 收集节点的基本块到一个路径。如果找到一个条件语句，分析条件语句生成的路径是否可以合并。如果不是，选择第一路径，将其他路径压入栈中继续进行分析。
4. 在进行路径分析的过程中保存函数调用的位置。
5. 当一个节点的路径分析完毕时，退栈分析其他路径，直到全部路径都被分析。

在Linux虚拟文件系统的唯一有困难的函数是path\_walk，其中有28条相关的路径。首先，函数中有很多很复杂的条件语句。然而，我们认为，这个函数调用，实际上是包含的很多语义不同的子程序，所以JABA数据流分析不能捕获正确的语义。许多子程序没有分析的必要。path\_walk函数路径分析需要3分钟，很多其他函数仅需几秒或者几十秒，因为大多数其他节点只具有1或2个路径。­­

Linux的虚拟文件系统的全部处理时间大约需要1小时。我们乐观地认为，静态分析性能可以显著改善。目前，我们的JABA系统的架构方式，我们必须重新计算path\_walk函数的路径分析，因为很多系统调用中使用它。除去该多余的计算减少大概一半的处理时间。另外，我们可以通过启动第一次看到节点的分析以进一步提高性能。

### 5.4 静态一致性结果

我们的目标是使用JABA一致性分析，发现无论是4.3节的漏洞还是TOCTTOU漏洞，我们使用静态分析工具CQUAL发现。这两者都可以被归类为一致性错误，但后者是很难找到一个运行时分析，因为它需要每次成功的攻击造成结果不一致。但如果是我们已经知道的漏洞，我们就不需要工具。

回想一下，file\_ops-> set\_fowner错误，因为在该领域（读取在这种情况下操作）的受控操作由对于除lease\_modify所有功能的set\_fowner授权。静态日志包括fcntl\_setlease项，这表明，在某些日志set\_fowner授权并在某些日志事实并非如此。有趣的是，日志不会收集日志条目lease\_modify，因为它总是与相同的授权调用，只有的fcntl。因为关系是固定的，我们不需要考虑功能区内的路径。不过，我们也必须考虑到一个特定的控制操作可以使用可识别的一致性问题，比如这里的情况。在这种情况下，我们想知道的是，日志可以调用lease\_modify等功能，并执行其控制操作。由于控制运营和授权之间的关系是静态的，我们可以聚集所有的控制操作（即，将它们合并成一个总的路径）。这样做将使瓦利分析工具，以确定缺乏lease\_modify授权的。­­­­

该TOCTTOU漏洞Zhang等人发现，因为文件指针被授权sys\_fcntl，而是一个新的文件指针是基于fcntl\_getlk用户提供的文件描述符提取。由于用户可以控制文件描述符和它引用的文件之间的映射，竞争条件导致攻击者可以利用来进行未经授权的fcntl操作。JABA提供跟踪程序中的变量和字段的可能值的数据流图。在这种情况下，我们需要根据其来源来区分文件指针变量（即从文件描述符fget）。生成的静态记录表明，文件指针对象是sys\_fcntl被授权在所有情况下，而不同的文件指针对象在被从未授权fcntl\_getlk使用。­

因此，JABA分析工具可以覆盖我们以前使用的静态分析，以及更广泛的一致性分析。

## 6 讨论

在本节中，我们简要分析一下在使用LSM一致性分析的三个问题：（1）静态分析的有效性; （2）LSM钩放置的验证; 和（3）用于回归测试。­

### 6.1静态分析的问题

在一般情况下，我们发现，我们可以使用通过两种动态或静态方法收集的日志数据所描述的一致性分析。目前证明概念的静态分析日志收集与构建和执行动态的日志收集大约需要同样多的时间。并且可以使用任何一种工具来执行相同的日志分析。

静态分析具有路径覆盖和准确性的优点。首先，它能够找到执行几乎所有相关的路径。基本上，为了执行正确的路径代码的静态分析是需要必要的基准。其次，以前的CQual静态分析表明，TOCTTOU错误可以发现，但是无法找到

使用动态分析。查找使用动态分析这些错误的要求有一个主动的攻击者攻击TOCTTOU漏洞，所以我们已经知道攻击的存在。

有效利用静态分析依赖于处理所有必要的控制操作和使用数据流量来识别授权的对象。读操作扩展分析会延长日志的长度，并可能增加相关路径的数量。相关的路径数量取决于控制操作和授权之间的新关系，但由于每个系统调用的授权的数量较少，所以可以进一步细化上下文的概念，以限制日志生成的范围。

静态分析的一个重要用途是判断通过LSM钩子授权的对象。该LSM钩子没有明确描述他们授权的对象，因此必须确定。使用手动指定的运行时分析这样的特性有，但推断静态分析可以使得这样的性质。在一般情况下，传递到挂钩对象的授权。另外，也可以认为是被授权的对象通过固定字段引用的对象进行授权。这些场是否是不可改变的取决于是否有一个系统调用，可以得到一个别名相同的对象，并且可以修改字段。创建对象一般要求不符合，因为他们创造一个新的对象，所以他们不能修改现有对象。虽然文件的文件描述符关系可以通过系统调用进行修改，文件对象和其索引节点之间的关系一旦创建了文件对象很可能不会被修改。我们正在研究的数据流分析会使这种特征得到验证。­

### 6.2 验证

大多数分析工作的目的是找bug。尽管一些分析方法是完整的，这些工具还没有被用来证明系统任何显著安全性能。有了足够的分析支持，我们希望能够真正验证系统的安全性。我们要考虑在完成这项任务时起到的作用。

我们的分析使验证，即一个特定的一组 *中介业务­* （例如，LSM钩子授权）被一致地执行，当一组的 *使用操作*（例如，内核控制操作）被执行时。如果中介操作可以在事先确定（即，我们可以找出所需的授权）就可以关联到操作，那么这个分析就可以完成。

在LSM钩子分析，事先未指定用于控制操作的授权，因此一致性分析所述来确定所需操作。有三个问题，使这个假设是困难的：⑴ 有可能是LSM的钩子只是完全缺失，因此不存在不一致的问题;⑵上下文描述可能不包括所有相关路径; （3）的设定由LSM钩子标识的对象可以不包括某些已经被授权的对象。在第一种情况下，如果是一个钩子在任何情况下不丢失并不存在一致性问题。这个想法是，授权控制的操作关系的聚集使这个手动验证。然而，在人工验证错误的可能性。在第二种情况下，上下文规范必须保证一致性分析涵盖的相关日志路径。为了实现这一目标，每个控制操作必须属于一个范围内声明的授权关系是一致的，对失踪钩子进行验证。三种类型的上下文的简单性（即召回系统调用，系统调用的标志，和操作集）我们的目标是使用JABA数据流分析，以验证该属性。

因此，如果我们能保证所有的内核路径，这种影响的一致性，保证他们的授权的所有对象的全覆盖，并可靠地验证授权要求，那么我们就可以进行有效的验证。我们预计，我们可以验证，如对象标识（使用前标签），对象初始化，审计和对象重用性。例如，审计业务应与安全决策匹配。这些都是很容易识别，所以这个验证应该是简单的推断，虽然受影响的对象依然是一个问题。对于标签，我们可以识别标签的操作，并确定它们是否对前安全检查的所有对象运行。然而，对象可以被重新标记。确定是否发生重新贴在必要时需要重新标记定义的要求。指示重新标签和表示使用一个重新标记对象的受控操作必须被确定。因此，我们可以验证初始标记，但正确的重新贴标签的验证更加困难。­­­­­­

### 6.3回归

一旦安全性能得到验证，必须确保当系统更新后此属性保持不变。该过程通常被称为回归测试。由于很多人使用Linux系统因此回归测试比较简单。特别是，我们想最大限度保证一个简单的变化将不会导致回归的数据没有手动检查的可能性。

对于LSM钩子验证，使用稠度类提供针对改变一些弹性。只要一致性类列表不改变，那么输出基本上是相同的。由于一致性的变化是指示性的安置问题，在回归测试一致的一致，表明有效安置。为了让这个假设持有，我们还必须考虑的事实是必要的挂钩可以在任何情况下丢失。因此，只要没有新的控制操作，则此回归方法的工作原理。对于增加新的控制操作，需要做更多的工作。­­

但存在潜在的问题才会对受控操作进行测试。一般而言，较大的不同集合的授权给该受控操作在一致性列表相关联的数是从特定的授权其独立的一个量度。这种情况将被视为­需要一个新的LSM钩子。在另一方面，如果控制操作是一组定义的上下文控制操作的一部分，则有必要对一个LSM钩的可能性要高得多。我们可以汇总基于这些种类的标准来判断我们需要的控制操作。

## 7 结论

在本文中，我们提出了一致性分析用于验证LSM框架的正确性。LSM框架由一组放置在内核内部授权挂钩子所组成，所以它更难以识别完整的调试点。我们利用的事实是大多数LSM钩子会被放到正确的地方，并通过这一事实判断错误的钩子。我们使用对主要内核数据结构作为中介界面结构件业务，并收集这些操作的授权。通过分析运行时测试的输出，判断它的授权是否是不一致的操作。我们分析了文件系统和一些任务操作，发现一些可能被利用的异常。与LSM社区沟通，这些问题已经被修复。

最终，我们发现一致性分析验证，认定与规范不一致的可能是错误的方式非常有用。然而，运行时间日志收集在路径覆盖的局限性和会错过一些重要的错误，比如TOCTTOU，这个漏洞依赖于异常的输入数据或上下文。我们已经发现，这种分析方法也可以应用于通过静态分析收集的执行日志。我们演示了如何使用JABA分析工具收集这些日志。此外，我们还讨论了改进，可以使用JABA的数据流分析进行了全面的分析。我们正在积极寻找完整的，易于使用的验证的LSM授权钩子和检查等安全性能的方法。­­­