Jitk: A Trustworthy In-Kernel Interpreter Infrastructure

阅读报告

江梦　P14206009

本文提出了一种新的构建内核解释器的基础架构——Jitk，能够保证应用程序中用户指定过滤器翻译为本地机器码过程中的正确性和安全性，并进行了形式化验证证明。作者从概述、设计、实现以及评估等几个方面向我们详细介绍了 Jitk 的概念、方法、设计目标、架构以及系统实现，并通过一系列测试证明了 Jitk 不仅拥有媲美现在的内核解释器的良好性能，而且还提供了正确性和安全性的保证，是一种值得信赖并有广阔前景的内核解释器架构。

许多操作系统允许用户空间应用程序通过加载用户指定的代码到内核从而定义新功能或系统策略。一个著名的例子就是 BSD 包过滤器（BPF）架构，用户可以通过将过滤器加载到内核来筛选应用程序需要的包。出于安全性和可移植性考虑，内核使用一个解释器将用户代码转换为本地机器码后执行。这样一来，解释器和用户指定代码的正确性成为影响系统安全性的关键。由于解释器驻留于内核空间从而拥有完整权限，一旦解释器出现错误可能使攻击者控制整个系统。然而，现有技术很难保证内核解释器和用户代码没有错误。在这样的背景下，作者设计了 Jitk 解决上述问题，并通过形式验证证明了正确性和安全性。Jitk 能保证解释器生成的本地机器码完整继承了用户空间提交的代码的语义，而且排除了除 0 或越界内存访问等破坏性操作。同时 Jitk 还引入了一种新型高层规范语言 SCPL（System Call Policy Language）来减少用户代码出现错误的可能，实现并证明了从 SCPL 到 BPF 编译的功能正确性。

作者介绍了一些相关研究工作。包括关于构建可信的形式验证的软件系统的一些创举，如 SeL4，CompCert，MinVisor，VCC 和 Myreen’s x86 JIT compiler。隔离失效内核的技术有很多，如扩展 OS 设计的 microkernels 和 exokernels，使用类型安全语言的 SPIN (Modula-3)、Singularity (C#) 、Mirage (OCaml)，基于软件的故障隔离方法 BGI、LXFI、XFI、VINO 和 SVA，作者指出这些技术通过将用户指定代码与内核其余部分隔离开以保障内存安全和内核完整性，但无法保证下载代码的功能正确性。还有一些与内核安全性相关的测试工具如 EXE，KLEE 和 Trinity syscall fuzzer，能有效发现内核代码中的 bug，但对消除 bug 的产生没有作用。

在设计 Jitk 之前，作者以 BPF 为例对内核解释器中出现的 bug 进行了分析总结，并相应讨论了实现正确性的困难。内核解释器可能产生的 bug 主要有：（1）控制流错误，如跳转中

的差一错误、偏移量溢出；（2）算术错误：除 0 和优化的算术操作；（3）内存错误：越界内存访问；（4）信息泄露：过滤器访问未初始化的寄存器或暂存得到之前应用程序的敏感信息。另外，由于程序员可能忽略无效输入或编写错误代码，在用户空间应用程序也可能出现 bug。

本文 Jitk 的应用环境是 Linux 中使用了 Jitk/BPF JIT 的 Seccomp 子系统，普通的 Seccomp 采用 BPF 来解释进程的系统调用策略，其架构如图 1 所示，与之相比，改进后使用 Jitk/BPF 的系统架构（如图 2）有三个重要区别：（1）SCPL，使程序开发员可以更容易的编写系统调用策略，而不再是手工编写 BPF 代码，提供了 SCPL 编译器可以将 SCPL 规则翻译为 BPF 过滤器；（2）共享后端的 JIT 编译器，Jitk 中 BPF 过滤器提交到内核后就即时翻译为本机码，之后的每个系统调用只需根据本机码来决定是否允许调用，不需要再依次调用 BPF JIT，节省了系统开销。Jitk 还包含了一个独立于 BPF，能被编译器共享的复用 CompCert 的后端；（3）SCPL 编译器和 BPF JIT 都增加了形式验证。

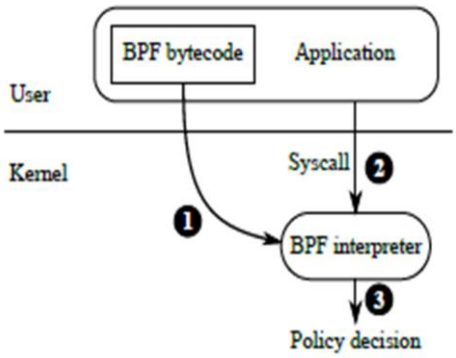

图 1：Linux 中 Seccomp 系统架构

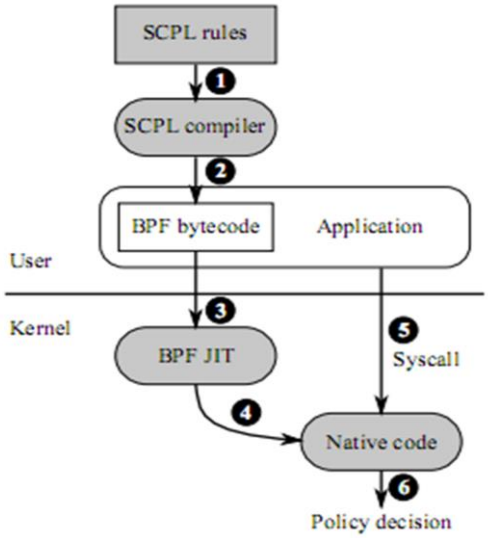

图 2：Jitk/BPF 系统架构

Jitk 的设计目标是将易于理解的高级策略正确翻译为能在内核中安全执行的底层本机代码。目标主要是实现两个方面：一、正确性目标，即在内核中应能正确运行行为良好的应用程序的过滤器；二、安全性目标，任何情况下攻击者都不可能通过滥用 Jitk 来控制内核。作者用一系列定理和引理来定义这两个目标，以便进行形式验证的证明。正确性主要是语义保持，定理表示即 **Theorem1** (End-to-end correctness)——对于任何用 SCPL 编写的系统调用策略 p，若 Jitk 接收它，则整个系统都坚持执行 p 的语义。Jitk 中系统调用策略从用户空间加载到内核去执行需要经历三个步骤：（1）SCPL 编译器将 SCPL 规则翻译为 BPF 过滤器；（2）BPF 过滤器在用户空间和内核中都是内存表示的程序，但要以字节码表示形式越过两个空间之间的边界，所以需要先在用户空间编码为字节码，到内核后再解码；（3）内核中 BPF JIT 将

过滤器翻译为本地机器码。为了实现正确性目标，上述三个过程都要保证前后语义一致，对此 Jitk 分别用三个引理来表示：**Lemma 2** (SCPL-to-BPF semantic preservation)、**Lemma 3** (User-kernel representation equivalence)、**Lemma 4** (BPF-to-native semantic preservation)。另一方面，Jitk 还保证了应用程序不会滥用 Jitk 来独占 CPU 时间或者污染内核内存，即安全性目标。Theorem1 保证了 SCPL 规则到本机码的正确性，但没有考虑到本机码中可能存在无限循环和堆栈溢出对安全性的影响。为此，Jitk 增加了两个定理：**Theorem 5** (Termination)，保证了生成的本机码一定会在有限步骤内停止下来（没有无限循环）且不会有使执行卡住的未明确定义的行为（如除 0 和越界内存访问）；**Theorem 6** (Bounded stack usage)，保证了生成的本机码不会溢出内核栈。除了 BPF JIT 生成的本机码要能安全执行外，JIT 本身的安全性由 Coq 确保，因为 JIT 用 Coq 编写，而 Coq 编写的程序都能保证内存安全并且能够终止。

为了确保 Jitk 是一个可信赖的内核解释器，必须要证明它的实现满足正确性和安全性目标，因此 Jitk 在 CompCert 基础上利用 Coq 验证助手进行开发的，开发流程如图 3 所示。Jitk 的实现代码（包括 SCPL 编译器和 BPF JIT）和证明都是用 Coq 编写的，其中每个组件的 Coq 源码包括三个主要部分：规范、实现、证明。

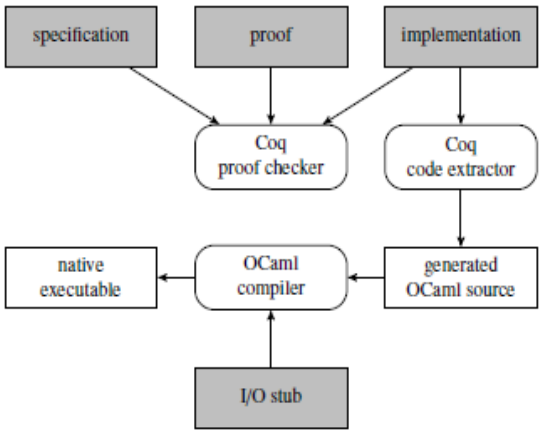

图 3：利用 Coq 开发 Jitk 的流程

将 Coq 实现代码提取为 OCaml 代码并与与 I/O stub 和 OCaml 运行环境一起链接成一个本地可执行

Jitk 的详细设计和工作流程如图 4 所示，在给出设计的同时作者还通过形式验证证明

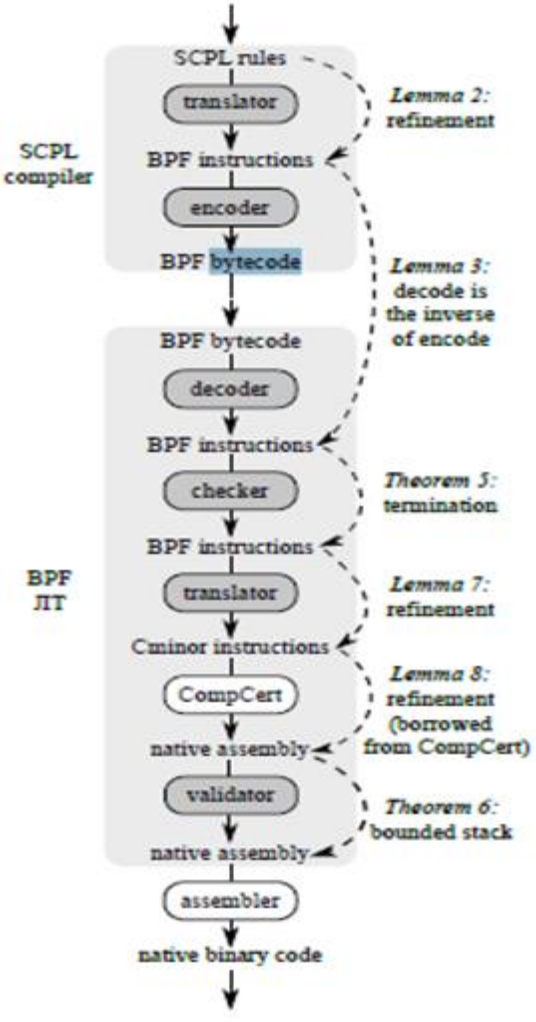

图 4：Jitk 中 SCPL 编译器和 BPF JIT 的细节描述

了 Jitk 的正确性。主要包括：

1、 内核中的 BPF JIT： BPF JIT 的实现应该满足 BPF 指令翻译为本机码时语义完整且能保证生成的本机码的安全性，即满足 **Lemma 4**、**Theorem 5** 和 **Theorem 6**。为此，在 Jitk 中设计了三种组件：翻译器、检查器和验证器。翻译器将定义良好的 BPF 过滤器翻译为本机码并保证了语义一致，Jitk 选择了一种 CompCert 的中间语言 Cminor 作为 BPF 到本机码的过渡，可以保留高层语言的构图以便进行证明。作者通过证明翻译器满足两条性质——**Lemma 7** (BPF-to-Cminor semantic preservation)和 **Lemma 8** (Cminor-to-native semantic preservation)证明了翻译器的正确性。检查器是为了实现 **Theorem 5** (Termination)提出的终止目标，它会拒绝所有未明确定义的输入 BPF 过滤器，并且保证通过的过滤器都能终止。由于 CompCert 中没有提供在编译过程中计算栈边界的方法，Jitk 设计了验证器来检查它的输入——本地汇编码中函数入口处的堆栈分配指令指定的栈大小，并与栈空间的最大值 S 进行比较，若超过最大值则拒绝，这样就保证了对堆栈的有界限使用。

2、 SCPL：SCPL 是一种新的高层规范语言，应用程序开发人员可以利用 SCPL 直观地编写规则来指定所需的系统调用策略，减少出错的可能。

```
{ default_action = Kill;
  rules = [
    { action = Errno EACCES; syscall = SYS_open };
    { action = Allow; syscall = SYS_getpid };
    { action = Allow; syscall = SYS_gettimeofday };
    ...
  ] }
```

图 5：SCPL 举例

最后作者对 Jitk 的实现原型进行了评估。首先是构建 Jitk 花费的工作量如图 6 所示，其中大部分精力用于构造证明。作者还将 Jitk 用于 Linux 内核的 INET-DIAG 解释器，以确定 Jitk 可以用于多种字节码语言。其次，通过人工验证 Jitk 的定理，表明了 Jitk 能避免目前解释器中发现的错误。然后为了评价生成的本机码的质量，作者比较了在不同架构上 Jitk 与两种内核 Linux 和 FreeBSD 的 BPF JIT 生成本机码的规模，结果表明 Jitk 生成的本机码质量不逊色于现有的 JIT。此外，作者还通过实际测试证明了 SCPL 的可用性。性能方面，通过测试不同情况下的用户认证时间，如在 OpenSSH 两种不同配置：手工编写的 BPF 过滤器和 SCPL 生成的 BPF 过滤器，以及两种内核配置：普通 Linux 和使用 Jitk BPF JIT 的 Linux 内核，可以得到 SCPL 生成的过滤器与手写 BPF 过滤器有同样表现。Jitk 的 BPF JIT 较于传统解释器的一个优势是，在 BPF 过滤器加载到内核并被翻译为本机码开始执行后，对之后到来的系统调用都只需执行已生成的本机码，不用再每次都调用 BPF JIT，节省了系统开销。

总之，Jitk 是一种构建内核解释器的新架构，并使用形式验证来确保功能正确性。Jitk 保证从用户空间应用程序的高层策略规则，经过用户—内核空间交界，到底层 BPF，再到内核

中的本机码，每一步的正确性，也确保了内核中生成的本机码执行时能够终止和有界的栈使用。通过对已知的解释器漏洞的分析，论证了 Jitk 避免了所有这些安全漏洞的出现。而且实践评估表明 SCPL 规则对现在的应用程序有很好的适用性，且 Jitk 具有灵活性、安全性以及良好性能。

本文主要贡献如下：

（1） 提出了一种用于构建形式验证化的内核解释器新的方法和基础架构——Jitk；

（2） 研究了现实世界中操作系统的 BPF 解释器目前已发现的几类漏洞，并通过形式验证证明了 Jitk 能有效避免这些已发现的各种漏洞；

（3） 明确了在内核中执行用户指定策略所要达到的目标——正确性和安全性目标，并通过形式化方法给出了相关定理和引理，以便对 Jitk 进行形式验证证明；

（4） 提出一种新的用于指定系统调用策略的高级语言——SCPL，SCPL 能使应用程序开发人员更方便直观地编写系统调用策略，减少用户代码中出现错误的可能，并证明了 SCPL 到 BPF 编译的正确性；

（5） 开创地将形式验证方法应用于内核解释器，证明了 Jitk 能够保证从用户指定代码到本地机器码过程的功能正确性；

（6） 提出了 Jitk 实现原型，实践验证了 Jitk 可以应用于 BPF 和 INET-DIAG 两种即时编译解释器，且对 Jitk 的安全性和性能进行了评估。

本文主要是提出了一种新的构建内核解释器的方法——Jitk。内核解释器能使用户在应用程序中指定的系统调用策略在内核中被翻译为本地机器码执行，在这个过程中，无论是翻译过程或用户代码都或多或少会出现错误，而现有的内核解释器不能有效避免这些错误，就可能导致攻击者利用这些漏洞控制内核从而对整个系统造成巨大危害。

因此，作者提出了一种能够确保从用户指定代码翻译为本地机器码过程的功能正确性的构建内核解释器的架构——Jitk，同时还提出了新的高级语言——SCPL 能更简洁明了地编写系统调用策略从而减少了用户代码中的错误。避免漏洞要求内核解释器能实现正确性和安全性目标，作者通过定义一系列的定理引理来描述正确性和安全性目标，并以形式验证方法证明 Jitk 能满足这些定理，由此说明 Jitk 能有效避免内核解释器可能产生的漏洞。

但是 Jitk 也有一定的局限性，例如有些固有的错误无法避免，如 BPF 和 SCPL 规范本身存在错误，JIT 喷射造成内存崩溃，编码器/解码器的自洽错误。Jitk 利用了基于 CompCert 框架的 Coq 证明助手开发的，无论是规范、实现代码还是证明都是用 Coq 编写的，对于整个 Jitk 架构中的这些部分如 Coq proof checker、Coq extraction system、OCaml compiler 没有进行

形式验证，只是根据 Coq 能提供强力的可信保证而假定其正确性，若 Coq 本身存在错误，那么 Jitk 不可避免的也会发生错误。所以我认为下一步的研究工作可以进一步完善 Jitk 的正确性和安全性。

另外，作者也对 Jitk 的性能进行了测试评估，结果表明 Jitk 能良好运行于多种环境（x86、ARM、PowerPC）上的 Linux 和 FreeBSD 操作系统内核，并且能与 BPF 和 INET-DIAG 解释器集成。但是与传统的解释器相比会多出一部分额外延迟，这是由于 Jitk 将 Coq 代码提取为 OCaml 代码并编译为本地可执行文件才能运行，需要调用一个进程设置 OCaml 运行环境和 OCaml 汇编器，增加了系统开销。如今 Jitk 只是与已有的内核解释器相当，且还多出了一部分额外延迟和花销，下一步研究方向可以提升改善 Jitk 的性能使其优于现有内核解释器。