



中国科学院大学  
University of Chinese Academy of Sciences

## 硕士学位论文

针对 Intel SGX SDK 的安全增强框架技术研究实现

作者姓名：\_\_\_\_\_陈力恒\_\_\_\_\_

指导教师：\_\_\_\_\_马恒太 副研究员\_\_\_\_\_

\_\_\_\_\_中国科学院软件研究所\_\_\_\_\_

学位类别：\_\_\_\_\_工学硕士\_\_\_\_\_

学科专业：\_\_\_\_\_计算机软件与理论\_\_\_\_\_

培养单位：\_\_\_\_\_中国科学院软件研究所\_\_\_\_\_

2021 年 6 月



**Research and Implementation of Security Enhancement**  
**Framework Technology for Intel SGX SDK**

A thesis submitted to  
University of Chinese Academy of Sciences  
in partial fulfillment of the requirement  
for the degree of  
Master of Science in Engineering  
in Computer Software and Theory

By  
CHEN Liheng

Supervisor: Associate Professor MA Hengtai

Institute of Software Chinese Academy of Sciences

June 2021



**中国科学院大学**  
**研究生学位论文原创性声明**

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。

作者签名：

日 期：

**中国科学院大学**  
**学位论文授权使用声明**

本人完全了解并同意遵守中国科学院有关保存和使用学位论文的规定，即中国科学院有权保留送交学位论文的副本，允许该论文被查阅，可以按照学术研究公开原则和保护知识产权的原则公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：

日 期：

导师签名：

日 期：



## 摘 要

云计算技术不断发展，吸引越来越多的用户将应用程序部署到云平台。云平台上大量应用程序对安全敏感，如何防止攻击者窃取应用程序中敏感内容是云平台的热难点问题。

Intel SGX 所提供的用户态可信执行环境 Enclave 能有效解决云计算等场景中的远程计算安全问题，保护云平台上应用程序敏感内容。Intel SGX 主要通过密码学方法和访问控制保障应用程序代码数据的机密性和完整性，抵御 Ring0 权限攻击。但最近研究表明攻击者能通过 Enclave 接口和共享资源等攻击面开展攻击，破坏 SGX 安全性。SGX Enclave 的安全不仅依赖于强隔离机制，还依赖于安全可靠的软件架构以确保安全的接口调用，开发者如果忽视接口安全，SGX Enclave 将丧失安全保障。

本文对 SGX 安全增强技术展开研究，旨在抵御 SGX 攻击并提升 SGX 安全性。本文主要贡献包括：

1. 现有 SGX 攻击向量能够破坏 SGX 安全性，但已有防御方案针对性地防御特定 SGX 攻击向量或攻击面，各个方案遵循的安全准则不统一，至今没有一个方案对 SGX 安全关键点进行全面分析，导致现有 SGX 架构难以有效兼容这些防御方案。针对上述问题，本文以 SGX 软件栈为主刻画了 SGX 架构及其关键执行路径，总结了其中的安全关键点。
2. 针对利用 SGX SDK 无法管控 Enclave 线程执行序及无法审计安全/非安全上下文切换事件等安全问题透过 Enclave 接口及共享资源等攻击面的多种 SGX 攻击向量，本文设计实现了针对 Intel SGX SDK 的安全增强框架（SGX-SEF），审计 SGX 关键执行路径，兼容已有安全准则防御多种攻击向量。
3. 针对恶意篡改 Enclave 代码调用顺序的调用排序和并发调用攻击向量，本文提出了“调用顺序白名单”安全策略，基于此策略审计检查关键执行路径，判断调用顺序是否符合安全策略。
4. 实验验证了 SGX-SEF 能有效审计 SGX 关键执行路径，评估了 SGX-SEF

审计功能引起的性能开销，分析了引起性能开销的原因。审计功能开销随日志容量增大而增大，开销包括日志使用维护开销及缓存未命中开销。当日志大小 100 项时，审计功能对各种调用方式引起的开销增幅不明显，100 项大小的日志足够记录最近的安全事件用于审计检查功能。日志容量为 100 项时，开销增幅在最优情况(Ordinary 调用方式)下为 13%~18%。实验验证了防御者可以在 SGX-SEF 中灵活部署安全策略以抵御多类攻击向量。

**关键词：**Intel SGX，安全增强框架，关键执行路径，安全关键点



## Abstract

The continuous development of cloud computing technology attracts more and more users to deploy applications on cloud platforms. A large number of applications on cloud platforms are sensitive to security. How to prevent attackers from stealing sensitive content in applications is a hot and difficult problem for cloud platforms.

The user-mode trusted execution environment "Enclave" provided by Intel SGX can effectively solve remote computing security issues in cloud computing and other scenarios, and protect sensitive content of applications on the cloud platform. Intel SGX mainly protects the confidentiality and integrity of application code data through cryptographic methods and access control, and resists Ring0 permission attacks. However, recent studies have shown that attackers can attack through the Enclave interface, shared resources and other attack surfaces to undermine the security of SGX. The security of SGX Enclave not only relies on a strong isolation mechanism, but also relies on a secure and reliable software architecture to ensure secure interface calls. If developers ignore interface security, SGX Enclave will lose security guarantees.

This paper conducts research on SGX security enhancement technology, which aims to resist SGX attacks and improve SGX security. The main contributions of this paper include:

1. Existing SGX attack vectors can undermine the security of SGX, but there are existing defense schemes to specifically defend against specific SGX attack vectors or attack surfaces. The security principles followed by each scheme are not uniform. So far, there is no scheme to conduct a comprehensive analysis of SGX security critical points, which makes it difficult for the existing SGX architecture to effectively compatible with these defense schemes. In response to the above problems, this paper analyses the SGX architecture and its critical execution paths mainly about SGX software stack, and summarizes the security critical points.

2. In response to security issues such as SGX SDK's inability to control Enclave thread execution order and the inability to audit secure/non-secure context switching events, which are leveraged by various SGX attack vectors to attack through the attack surfaces of Enclave interfaces and shared resources, this paper designs and implements Security Enhancement Framework for Intel SGX SDK (SGX-SEF), audits the SGX critical execution path, and is compatible with existing security principles to defend against multiple attack vectors.
3. Aiming at the Call Permutation and Concurrent Calls attack vectors that maliciously tamper with the calling sequence/timing of Enclave code, this paper proposes the "calling sequence/timing white-list" security strategy, auditing and checking the critical execution paths, and judging whether the calling sequence/timing complies with this security strategy.
4. The experiments verify that SGX-SEF can effectively audit the critical execution path of SGX, evaluates the performance overhead caused by the SGX-SEF audit function, and analyzes the reasons for the performance overhead. The audit function overhead increases as the log capacity increases, and the overhead includes log maintenance overhead and cache miss overhead. When the log size is 100 items, the audit function will not cause significant overhead for various calling methods. A log size of 100 items is enough to record the most recent security events for the audit check function. When the log capacity is 100 items, the overhead increase is 13%-18% under the optimal situation (Ordinary call method). The experiments also verify that the defender can flexibly deploy security strategies in SGX-SEF to resist multiple attack vectors.

**Key Words:** Intel SGX, Security Enhancement Framework, Critical Execution Path, Security Critical Point

## 目 录

第一章 引言.....	1
1.1 选题背景和意义.....	1
1.2 相关研究概述.....	2
1.3 主要研究内容和贡献.....	3
1.4 论文结构.....	5
第二章 国内外相关研究现状.....	7
2.1 Intel SGX.....	7
2.1.1 Intel SGX 编程模型.....	8
2.1.2 Intel SGX 软硬件架构.....	8
2.1.3 SGX Enclave 安全机制.....	10
2.2 SGX 攻击现状.....	11
2.2.1 源于 Enclave 开发中的安全问题.....	12
2.2.2 源于 SGX 设计存在安全缺陷.....	13
2.2.3 代表性攻击.....	14
2.3 传统系统防御方案.....	19
2.4 SGX 防御现状.....	20
2.4.1 Enclave 接口消毒.....	20
2.4.2 Enclave 内外资源隔离、痕迹防泄露及攻击痕迹检测.....	23
2.4.3 其他防御方案.....	24
2.5 本章小结.....	24
第三章 Intel SGX 关键执行路径及安全关键点分析.....	25
3.1 Enclave 接口关键执行路径.....	25
3.1.1 ECALL 进入 Enclave.....	25
3.1.2 ECALL 结束退出 Enclave.....	27
3.1.3 OCALL 离开 Enclave 及 OCALL 结束返回 Enclave.....	27
3.1.4 AEX 离开 Enclave 及异常处理完成返回 Enclave.....	28
3.2 其他关键执行路径.....	29

3.3	SGX 线程模型.....	32
3.4	本章小结.....	35
第四章 针对 Intel SGX SDK 的安全增强框架设计.....		37
4.1	针对 Intel SGX SDK 的安全增强框架总体设计.....	37
4.1.1	审计检查点插桩方式.....	37
4.1.2	审计检查点插桩位置.....	38
4.1.3	安全事件审计.....	39
4.1.4	安全事件检查.....	39
4.2	对 SGX 原生软件栈的修改.....	41
4.3	本章小结.....	42
第五章 针对 Intel SGX SDK 的安全增强框架验证.....		43
5.1	实验环境.....	43
5.2	审计功能效果验证及性能分析.....	43
5.2.1	审计功能效果验证.....	44
5.2.2	审计功能性能开销分析.....	44
5.3	防御（检查功能）效果验证.....	46
5.3.1	针对“调用排序”攻击向量的防御效果验证.....	46
5.3.2	针对“并发调用”攻击向量的防御效果验证.....	49
5.3.3	针对“恶意 Enclave 线程调度”攻击向量的防御效果验证.....	52
5.3.4	针对“时间侧信道”的防御效果验证.....	56
5.4	本章小结.....	57
第六章 总结与展望.....		59
6.1	总结.....	59
6.2	展望.....	60
参考文献.....		61
致 谢.....		65
作者简历及攻读学位期间发表的学术论文与研究成果.....		67

## 图目录

图 1.1	云平台上部署的服务.....	1
图 1.2	Intel SGX 保护云平台上服务程序安全.....	2
图 2.1	Enclave 应用生命周期 <sup>[30]</sup> .....	8
图 2.2	SGX 软硬件架构.....	9
图 2.3	SGX Enclave 安全问题、攻击面和攻击向量.....	12
图 2.4	调用排序攻击示意图.....	15
图 2.5	AsyncShock 攻击示意图.....	17
图 2.6	Haven 组件和接口 <sup>[26]</sup> .....	21
图 2.7	Graphene-SGX 架构图 <sup>[27]</sup> .....	22
图 2.8	SCONE 架构图 <sup>[28]</sup> .....	22
图 3.1	Ordinary ECALL 代码流程图.....	25
图 3.2	Switchless ECALL 的代码流程图.....	26
图 3.3	AEX 及异常处理流程图.....	29
图 3.4	受保护文件系统布局 <sup>[4]</sup> .....	30
图 3.5	本地认证（使用 DH 密钥交换库） <sup>[4]</sup> .....	31
图 3.6	安全信道的建立（使用 DH 密钥交换库） <sup>[4]</sup> .....	32
图 3.7	SGX 应用中 Ordinary/Switchless ECALL/OCALL 线程模型.....	33
图 4.1	插桩方式对比.....	37
图 4.2	OCALL Stub 函数示意图.....	38
图 4.3	审计检查点内部架构.....	40
图 4.4	SGX-SEF 对 SGX 软件栈的修改.....	41
图 5.1	审计功能的效果.....	44
图 5.2	调用排序攻击之前程序执行结果图.....	46
图 5.3	调用排序攻击之后程序执行结果图.....	47
图 5.4	调用排序攻击检测结果.....	47
图 5.5	调用排序攻击前 SGX_SQLite 执行情况图.....	48
图 5.6	调用排序攻击后 SGX_SQLite 执行情况图.....	48
图 5.7	SGX_SQLite 中调用排序攻击检测结果.....	49
图 5.8	并发调用攻击之前程序执行结果.....	49

图 5.9	恶意篡改并发调用.....	50
图 5.10	并发调用攻击之后程序执行结果.....	50
图 5.11	并发调用攻击检测结果.....	51
图 5.12	并发调用攻击前 SGX_SQLite 执行情况图.....	51
图 5.13	并发调用攻击后 SGX_SQLite 执行情况图.....	52
图 5.14	SGX_SQLite 中并发调用攻击检测结果.....	52
图 5.15	恶意线程调度攻击前程序执行结果.....	53
图 5.16	恶意中断 ecall_free_obj1 函数.....	54
图 5.17	恶意线程调度攻击后程序执行结果.....	54
图 5.18	使用 Kprobe 的页错误型 Enclave 线程暂停原语示意图.....	55
图 5.19	使用 SIGSEGV 的页错误型 Enclave 线程暂停原语示意图.....	55
图 5.20	恶意线程调度攻击检测结果.....	56
图 5.21	SGX-Step/Nemesis 源码分析流程图.....	56

## 表目录

表 2.1	“Enclave 接口” 攻击表.....	18
表 2.2	“微架构共享资源” 攻击表.....	18
表 2.3	“部分功能依赖内核” 攻击表.....	19
表 2.4	“可观测的物理信息” 攻击表.....	19
表 2.5	“密封文件” 攻击表.....	19
表 5.1	基于 Switchless 的审计功能性能开销测量表.....	45
表 5.2	基于 SampleEnclave 的审计功能性能开销测量表.....	46





## 第一章 引言

### 1.1 选题背景和意义

云计算不断发展，吸引越来越多的用户将应用程序部署到云平台。根据 Gartner 2019 年的报告，全球云计算市场（IaaS 部分）份额达到 445 亿美元，市场份额前三名的云平台分别为亚马逊 AWS、微软 Azure 和阿里云。部署在云平台上的应用程序（如图 1.1 所示）包括 Facebook（即时通信服务程序）、23andMe（基因技术服务程序）及 PayPal（网络支付服务程序）等。



图 1.1 云平台上部署的服务

Figure1.1 Services deployed on the cloud platform

云平台上的大量应用程序对安全敏感。尽管云平台已经部署了 Web 应用防火墙和 DDOS 防御系统等，针对云平台上应用程序的攻击事件依然频发。黑客能够构建提权攻击，云平台管理员能够直接访问云用户内容，使得云用户无法完全信任云平台。如何保护远程计算（如云平台上程序的计算过程及计算内容）安全性的问题被论文<sup>[1]</sup>总结为远程计算安全问题。Intel Software Guard Extensions（简称 SGX）能够有效解决远程计算安全问题，保护远程计算安全性。SGX 构

建了用户态可信执行环境 Enclave，通过密码学方法和访问控制保护 Enclave 免受 Ring0 权限攻击。云平台上的应用程序可以使用 SGX 保护其安全（图 1.2）。

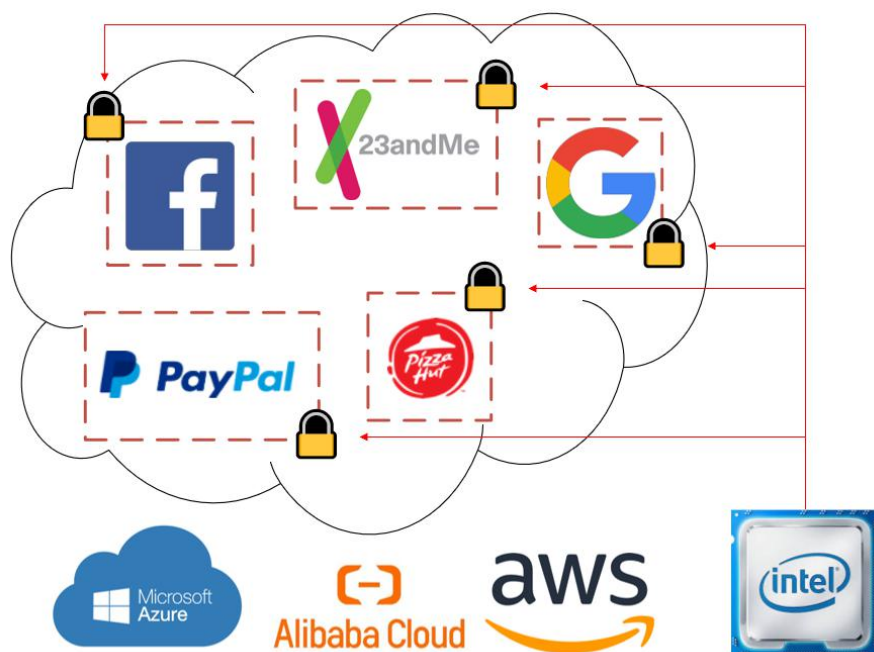


图 1.2 Intel SGX 保护云平台上服务程序安全

Figure1.2 Intel SGX protects the security of service programs on the cloud platform

阿里云研发了“基于 SGX2.0 和 TPM 的虚拟化实例”和“系统可信解决方案”，标志着阿里云具备提供完整云可信产品的能力。其它云平台（如亚马逊 AWS 和微软 Azure）也应用 Intel SGX 增强云安全能力，达到企业及国家法规要求（如中国国标等保 2.0）。

2013 年 SGX 概念被提出，2015 年 SGX 在 Intel Skylake 架构 CPU 中得到部署。此后，SGX 在云安全、网络通信和本地应用中被大量应用，多方计算、机器学习、区块链、人工智能和生物识别技术保护等场景逐渐使用 SGX 解决问题<sup>[2]</sup>。但最近研究表明，Enclave 开发过程存在诸多安全问题，SGX 设计存在安全缺陷，攻击者针对这些安全问题透过 SGX 攻击面利用各种攻击向量破坏 SGX 安全性。对此，本文旨在提出一种框架性的防御方案全面加固 SGX。

## 1.2 相关研究概述

Intel SGX<sup>[3-5]</sup> 旨在提供用户态的可信执行环境（Trusted Execution

Environment, 缩写为 TEE), 保护并证明计算过程及计算内容可信。TEE 技术是可信计算技术的发展, 基于隔离技术保护程序敏感内容。TEE 技术实现上可分为基于软件的和基于硬件的, 基于 CPU 硬件的除 Intel SGX 外还包括 ARM TrustZone、RISC-V Keystone、AMD SEV, 基于 GPU 硬件的包括论文<sup>[6,7]</sup>等, 基于软件的包括 Virtual Ghost<sup>[8]</sup>等。基于软件的 TEE 能快速形成原型系统, 基于硬件的 TEE 性能得到提升。TEE 技术根据架构可分为全栈式 TEE 和用户态 TEE(常称为 Enclave), 前者将计算资源划分为安全世界和非安全世界, 两者均包含用户态和内核态架构但彼此隔离, SGX 等提供的用户态可信执行环境更着重于保护用户态应用程序的敏感内容。

本文关注的 SGX 主要通过密码学方法和访问控制保护用户态可信执行环境 Enclave 免受 Ring0 权限攻击<sup>[1,9,10]</sup>。但最近研究表明, 开发者开发 Enclave 过程中未严格遵循 SGX 开发规范滋生安全问题, SGX 设计中对安全场景考虑不充分存在安全缺陷, 攻击者针对安全问题利用“缓存侧信道”、“页表攻击”、“影子分支攻击”及“调用排序攻击”等攻击向量开展攻击<sup>[11-25]</sup>, 严重破坏 SGX Enclave 的安全性。这些攻击向量主要透过 Enclave 接口和共享资源攻击面。Enclave 接口是指 Enclave 向外暴露的接口, 攻击者能够轻易操控 Enclave 接口攻击面展开攻击。而共享资源是指 Enclave 内外软硬件层面的共享资源, 攻击者观察共享资源上的信息变化推测 Enclave 内的执行流或数据流, 但共享资源攻击面难以被发现且难以被有效利用。

针对 SGX 攻击, 有若干防御方案被提出<sup>[26-38]</sup>, 它们在 Enclave 代码中对 Enclave 接口进行消毒, 将 Enclave 内外共享资源用更彻底的隔离机制进行隔离避免 Enclave 内将状态等信息泄露到 Enclave 内外共享资源, 检测攻击代码在 Enclave 内留下的痕迹。但分析发现各个防御方案所遵循的安全准则不统一, 难以被 SGX 架构同时兼容。至今没有防御方案深入刻画 SGX 架构, 全面加固 SGX。

### 1.3 主要研究内容和贡献

SGX 安全问题深刻影响 SGX 实际应用, 本文为了系统分析 SGX 架构, 增强 SGX 安全性, 展开了如下研究:

1. Intel SGX 关键执行路径及安全关键点分析。归纳现有 SGX 攻击所针对的安全问题、所利用的攻击向量及所涉及的攻击面, 分析 SGX 架构关键

执行路径及安全关键点位，加固 SGX 安全关键点以提升 SGX 自身安全性。

2. 针对 Intel SGX SDK 的安全增强框架设计。本文基于 SGX 原生架构设计针对 Intel SGX SDK 的安全增强框架（简称 SGX-SEF），以审计检查关键执行路径及安全关键点，记录安全事件，兼容多种安全策略防御多种攻击向量。
3. 针对 Intel SGX SDK 的安全增强框架验证。本文实验验证 SGX-SEF 对关键执行路径和安全关键点的审计效果，评估 SGX-SEF 审计功能引起的性能开销，分析引起性能开销的原因，实验验证 SGX-SEF 兼容多种安全策略防御多种攻击向量的能力。

本文主要贡献包括：

1. 现有 SGX 攻击向量能够破坏 SGX 安全性，但已有防御方案针对性地防御特定 SGX 攻击向量或攻击面，各个方案遵循的安全准则不统一，至今没有一个方案对 SGX 安全关键点进行全面分析，导致现有 SGX 架构难以有效兼容这些防御方案。针对上述问题，本文以 SGX 软件栈为主刻画了 SGX 架构及其关键执行路径，总结了其中的安全关键点。
2. 针对利用 SGX SDK 无法管控 Enclave 线程执行序及无法审计安全/非安全上下文切换事件等安全问题透过 Enclave 接口及共享资源等攻击面的多种 SGX 攻击向量，本文设计实现了针对 Intel SGX SDK 的安全增强框架（SGX-SEF），审计 SGX 关键执行路径，兼容已有安全准则防御多种攻击向量。
3. 针对恶意篡改 Enclave 代码调用顺序的调用排序和并发调用攻击向量，本文提出了“调用顺序白名单”安全策略，基于此策略审计检查关键执行路径，判断调用顺序是否符合安全策略。
4. 实验验证了 SGX-SEF 能有效审计 SGX 关键执行路径，评估了 SGX-SEF 审计功能引起的性能开销，分析了引起性能开销的原因。审计功能开销随日志容量增大而增大，开销包括日志使用维护开销及缓存未命中开销。当日志大小 100 项时，审计功能对各种调用方式引起的开销增幅不明显，100 项大小的日志足够记录最近的安全事件用于审计检查功能。日志容

量为 100 项时,开销增幅在最优情况(Ordinary 调用方式)下为 13%~18%。实验验证了防御者可以在 SGX-SEF 中灵活部署安全策略以抵御多类攻击向量。

## 1.4 论文结构

本论文共分为六章,各章内容组织如下:

第一章是引言。该章阐述了选题的出发点及意义,总结了相关研究现状,提出了本研究课题的主要研究内容及贡献。

第二章是国内外相关研究现状。该章分析了 SGX 的编程模型、软硬件架构及安全机制,总结了 SGX 的安全问题、攻击向量及攻击面,借鉴传统防御方案思路,分析 SGX 防御的不足。

第三章是 Intel SGX 关键执行路径及安全关键点分析。该章分析了 SGX 架构、关键执行路径,总结了 SGX 安全关键点位。

第四章是针对 Intel SGX SDK 的安全增强框架设计。该章设计了针对安全关键点的审计检查代码插桩方式及插桩位置,设计实现了安全事件审计检查功能,全面提升 SGX 架构安全性。

第五章是针对 Intel SGX SDK 的安全增强框架验证。该章实验验证了 SGX-SEF 审计关键执行路径及安全关键点的效果,通过实验评估了 SGX-SEF 审计功能的性能开销,分析了引起性能开销的原因,验证了 SGX-SEF 兼容多种安全策略防御多种攻击向量的能力。

第六章是总结与展望。该章总结了本文的主要内容及成果,指出了后续可研究的方向。



## 第二章 国内外相关研究现状

本研究课题围绕 SGX 安全性展开研究,本章针对 SGX 阐述其编程模型、软硬件架构和安全机制并总结 SGX 特点,分析现有 SGX 攻击并指出 SGX 安全问题、攻击面和攻击向量,分析传统防御方案及现有 SGX 防御方案,对比传统防御方案思路指出现有 SGX 防御方案不足。

### 2.1 Intel SGX

Intel SGX<sup>[3-5]</sup>全称为 Intel Software Guard Extension, SGX 提供了用户态的可信执行环境 (TEE), 通过密码学方法和访问控制保护并证明计算过程及计算内容可信。基于 CPU 实现的可信执行环境技术除 Intel SGX 外还包括 ARM TrustZone、RISC-V Keystone、AMD SEV (Secure Encrypted Virtualization), 基于 GPU 实现的可信执行环境技术包括论文<sup>[6,7]</sup>等, 基于软件实现的可信执行环境技术包括 Virtual Ghost<sup>[8]</sup>等。研究人员最早使用高权限或运行时检查技术基于软件构建可信执行环境, 软件 TEE 的优势在于能够快速生成原型系统, 但与部分硬件架构不兼容且性能较差。随后 CPU 厂商基于硬件实现可信执行环境, TEE 性能得到提升, 但仅能保护 CPU 中的计算过程及计算内容。人工智能运用 GPU 及硬件加速器加速模型训练过程, 研究人员为了保护 GPU 中的计算过程及计算内容, 提出了若干种 GPU TEE 技术, 并结合 CPU TEE 技术保护 CPU 和 GPU 中的计算过程及计算内容。

可信执行环境可分为全栈式可信执行环境和用户态可信执行环境。全栈式可信执行环境将计算资源划分为安全世界和非安全世界, 安全世界及非安全世界各自均包含用户态和内核态架构等, 但两者彼此隔离。而 SGX 等技术提供的用户态可信执行环境只在用户态构建可信执行环境保护用户态应用程序的敏感内容。可信执行环境技术是可信计算技术的发展, 可信计算技术 (如 TPM) 中通过单独的硬件模块保护 BIOS、操作系统等上层架构可信, 而可信执行环境技术保护程序执行过程中的敏感代码数据, 更加适合保护应用程序级敏感内容。

### 2.1.1 Intel SGX 编程模型

Intel SGX 建议仅将用户态应用程序中敏感内容放到用户态可信执行环境 Enclave 中，以减少 Enclave 内的可信计算基（Trusted Computing Base，缩写为 TCB）大小。Enclave 中过大的 TCB 会导致 Enclave 漏洞多，安全性差。

SGX 编程模型中，SGX 应用被分为不可信部分和 Enclave。Enclave 通过密码学方式和访问控制抵御 Ring0 权限攻击（包括来自操作系统和虚拟机监控程序等的攻击，如图 2.1 下方所示）。简化的 SGX 应用执行流程及 Enclave 生命周期：

（1）不可信部分显式创建（及销毁）Enclave，（2）不可信部分调用 ECALL（全称为 Enclave Call，即可信函数的入口）进入 Enclave，（3）执行可信函数，（4）可信函数调用 OCALL（全称为 Outside Call）暂退到不可信部分并执行 Enclave 内无法实现的功能，（5）可信函数执行完成返回到不可信部分。

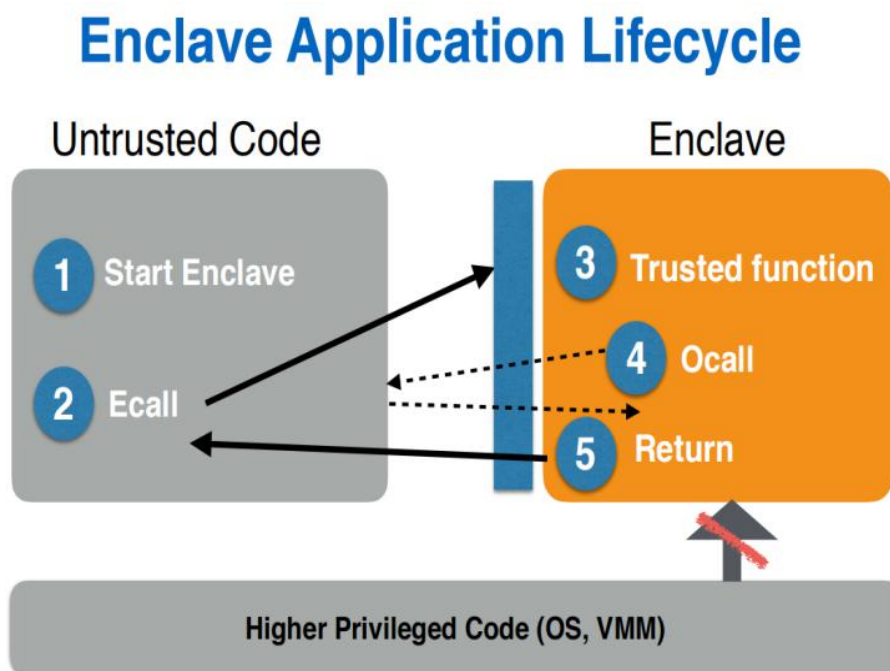


图 2.1 Enclave 应用生命周期<sup>[30]</sup>

Figure2.1 Enclave application lifecycle<sup>[30]</sup>

### 2.1.2 Intel SGX 软硬件架构

Intel SGX 狭义上是 Intel CPU 提供的一套扩展指令集，包括 Ring0 和 Ring3 权限指令。广义上还包括 SGX 软件栈，方便开发者使用硬件功能，SGX 软件栈包括 Intel SGX PSW（Platform Software）、Intel SGX SDK（Software Development



Kit) 和 Intel SGX Driver。Intel SGX 软硬件架构 (包括 SGX 硬件及 SGX 软件栈) 的架构如图 2.2 所示。

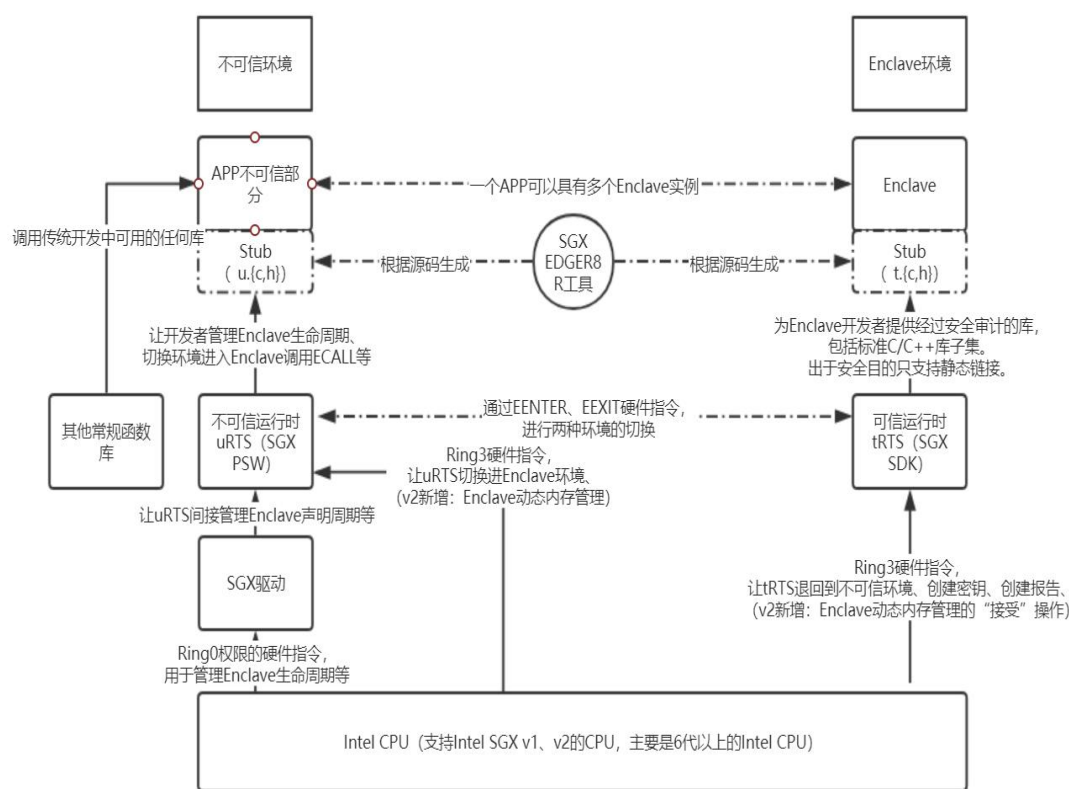


图 2.2 SGX 软硬件架构

Figure 2.2 SGX software and hardware architecture

Intel SGX 提供了 Ring0 指令 ENCLS (S 指 Supervisor 模式), 对 ENCLS 指令指定不同的 EAX 寄存器值调用不同的叶功能 (Leaf Function, SGX 手册中也将叶功能直接称为指令)。ENCLS 指令被内核态的 (不可信) SGX 驱动调用, 以提供 Enclave 实例生命周期管理等功能。如 ECREATE 指令 (ENCLS.00H, EAX 值为 00H) 主要用于创建 Enclave 实例的 SGX Enclave 控制结构体 (SGX Enclave Control Structure, 缩写为 SECS, 每个 SECS 唯一标识一个 Enclave 实例)。

Intel SGX 还提供了 Ring3 指令 ENCLU (U 指 User 模式), 提供 CPU 模式 (普通模式或 Enclave 模式) 切换等功能。ENCLU 的叶功能可划分为两部分:

1. 用户态的 SGX 应用不可信代码和 uRTS (untrusted Run-Time System) 调用的叶功能。如 EENTER 指令 (ENCLU.02H) 将 CPU 模式从普通模式切换成 Enclave 模式 (CR\_ENCLAVE\_MODE 寄存器置 1), 标识 CPU

能访问指定 Enclave 实例的资源（CR\_ACTIVE\_SECS 置为目标 Enclave 实例的 SECS）。

2. 用户态的 Enclave 代码和 tRTS (trusted Run-Time System) 调用的叶功能。如 EEXIT 指令 (ENCLU.04H) 将 CPU 模式从 Enclave 模式切换到普通模式 (CR\_ENCLAVE\_MODE 寄存器置 0), CPU 不能再访问任何 Enclave 实例的资源。

SGX SDK (也称为 tRTS) 向 Enclave 开发人员提供经过安全审计的静态链接库 (如 ENCLU 部分叶功能的封装函数、经过安全审计的 C/C++ 库子集等), 方便开发人员编写 Enclave 代码。静态链接的目的是确保 Enclave 代码均存储于安全内存中。

通过 SGX PSW (也称为 uRTS), SGX 应用不可信部分可以调用部分 ENCLU 指令, 并利用 IOCTL 通告 SGX 驱动调用 ENCLS 指令。SGX 应用不可信部分还可以调用其他常规函数库。

SGX 应用开发方式比较特殊。开发过程中, 开发人员需要将敏感代码数据统一编写成 Enclave。SGX 应用不可信部分与 Enclave 边界处的代码 (简称边界代码, 也称为 Stub) 由 SGX EDGER8R 工具依据开发者编写的 EDL (Enclave Definition Language) 文件自动生成。

SGX 假设除 Enclave 外均是不可信的。SGX 应用不可信部分可以被攻击者攻占, 并以任意顺序调用 Enclave 接口。

### 2.1.3 SGX Enclave 安全机制

SGX 通过密码学方法和访问控制等安全机制保护 Enclave 的机密性、完整性、新鲜度 (Freshness) 和真实性 (Authenticity) 等, 抵御 Ring0 攻击<sup>[1,9,10]</sup>。SGX 1 代提供的安全机制包括 EPC (全称为 Enclave Page Cache) 访问控制机制、EPC 机密性保护机制、EPC 完整性保护机制、EPC 新鲜度保障机制、Enclave 真实性保障机制和 Enclave 实例可信建立机制。EPC 是处理器保留内存 (Processor Reserved Memory, 缩写为 PRM, 位于 DRAM 起始位置, 无法被系统软件及外设 DMA 直接访问) 中的 64M/128M/256M 大小的物理内存。Enclave 实例的虚拟内存 (即 ELRANGE) 以动态库形式位于 SGX 应用的进程地址空间中, ELRANGE 的虚拟页映射 EPC 中的物理页。

EPC 访问控制机制确保 CPU 在普通模式下无法直接访问 EPC，被 EENTER 指令切换为 Enclave 模式后才能访问特定 Enclave 实例的资源，包括：

1. 内核段页机制中的访问控制。
2. EPC 无法被系统软件及外设 DMA 直接访问。
3. Page Miss Handler 硬件查询 SGX 独自维护的 EPCM（EPC Map），在代码非法访问 EPC 时触发中止页面语义。

EPC 机密性保护机制确保 Enclave 内容在 CPU 中明文存在，在物理内存（包括 EPC 部分和非 EPC 部分）和硬盘中密文存在，包括：

1. 内存加密引擎（Memory Encryption Engine，缩写为 MEE）使用 SGX 在计算机启动时生成的密钥将 CPU 中的明文加密存储于 EPC。
2. 将 EPC 内容改用 Enclave 实例特定密钥加密存储到普通内存。
3. 文件密封：Enclave 内操作的文件加密存储于不可信外存中。

EPC 完整性保护机制中，MEE 构建哈希树保护 EPC 完整性。哈希树维护开销大，只能保护较小尺寸（64M/128M/256M）的 EPC。

EPC 新鲜度保障机制。为了抵御针对 EPC 的重放攻击（Replay Attack），MEE 在哈希树中加入 Nonce，在加密数据时使用包含地址时间信息的 Version Nonce。

Enclave 真实性保障机制包括本地认证和远程认证等。

Enclave 实例可信建立机制。Enclave 实例建立过程中，EINIT 指令验证 Enclave 文件签名及 Enclave 实例建立过程度量值的有效性。

## 2.2 SGX 攻击现状

现有攻击表明，由于 Enclave 开发过程存在安全问题，SGX 设计存在安全缺陷，2.1.3 节中的安全保护机制不足以全面保护 SGX。攻击者针对安全问题利用各类攻击向量（指攻击者为了触发攻击所具体采用的方法）如缓存侧信道、内存安全问题、弱缓解技术、EDL 中未初始化的填充数据、页表攻击、影子分支攻击、针对 SGX 的 Row-hammer、并发调用攻击、调用排序攻击、恶意输入和嵌套调用攻击<sup>[11,15]</sup>，透过 Enclave 接口、Enclave 内外共享资源等攻击面（指系统中可以被攻击者输入或提取数据造成攻击的点位）破坏 Enclave 安全性。（如图 2.3 所示）

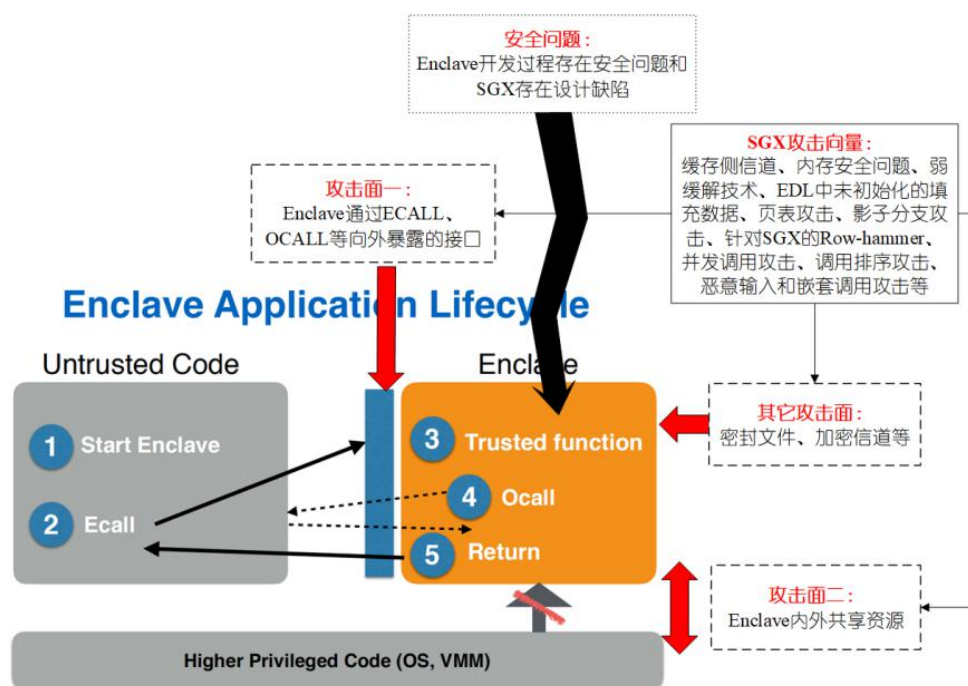


图 2.3 SGX Enclave 安全问题、攻击面和攻击向量

Figure 2.3 SGX Enclave security issues, attack surfaces and attack vectors

### 2.2.1 源于 Enclave 开发中的安全问题

SGX 假设除 Enclave 外均是不可信的，SGX 应用不可信部分可以被攻击者攻占，并以任意顺序调用 Enclave 可信函数。但开发者在开发过程中会忽视这个情况，产生的安全问题被攻击者频繁利用，SGX 未对此提供安全服务功能。

Coin Attack<sup>[15]</sup>归纳了并发调用攻击、调用排序攻击和嵌套调用攻击三种攻击向量。这些攻击向量主要源于开发者忽视 SGX 开发特点，未限制不可信代码对 Enclave 的调用逻辑。它们主要透过 Enclave 接口攻击面展开攻击，具体描述如下：

1. 调用排序攻击（Call Permutation）。SGX 编程模型中，不可信代码可以任意顺序调用 Enclave 函数，但是 Enclave 开发人员未做出限制，使得攻击者能够恶意顺序调用 Enclave 函数绕过特定接口检查。
2. 并发调用攻击（Concurrent Calls）。不可信代码可以并发调用多个 Enclave 函数，触发 Enclave 并发错误，如 Enclave 内的 Race Condition。
3. 嵌套调用攻击（Nested Calls）。嵌套 ECALL 和 OCALL 构建非预期的控制流，进而形成攻击。

SGX 要求开发者基于先验知识针对 Enclave 接口进行安全检查，SGX 只向开发者提供基础性的接口安全检查功能。开发者编写 Enclave 代码时，对 Enclave 接口检查不充分，产生安全问题被多类攻击向量利用。针对此安全问题的攻击向量包括：

1. 恶意篡改输入（Input Manipulation）。攻击者在 Enclave 接口处输入非预期值，影响 Enclave 内状态，形成攻击。SGX tRTS（trusted Run-Time System）无法先验地知道输入数据的处理方法，只能提供安全能力有限的接口消毒功能（如 EDL 关键字）。
2. EDL 中未重新初始化的填充数据（Uninitialized Padding In EDL）。传入传出 Enclave 的数据结构内存在用于保证数据结构正确对齐的填充数据，填充数据可能携带残留的敏感信息，使得 Enclave 敏感信息泄露，Enclave 外非法输入恶意数据影响 Enclave 状态。

### 2.2.2 源于 SGX 设计存在安全缺陷

研究表明 SGX 提供的安全机制存在缺陷。CPU 资源隔离机制中对资源隔离不充分，形成 Enclave 内外共享资源攻击面，被大量侧信道攻击利用<sup>[1,10]</sup>。Enclave 内外共享资源包括：

1. 微架构共享资源。Enclave 内外存在微架构层面的共享资源，如缓存、分支预测器和 TLB。
2. 部分功能依赖不可信内核。用户态 Enclave 部分功能依赖于不可信内核，如页错误处理和线程调度。
3. 可观测的物理信息。Enclave 内的物理信息如执行时间可被 Enclave 外感知。

针对此安全问题的攻击向量包括：

1. 接口侧信道。攻击者观察 Enclave 在接口处所泄露的侧信道信息构建攻击。
2. 缓存侧信道。攻击者观察 Enclave 在 Enclave 内外共享缓存（L1、L2 和 LLC）中泄露的信息，推测 Enclave 内容。
3. 分支预测器侧信道。攻击者观察 Enclave 在 Enclave 内外共享分支预测器（Branch Predictor）中泄露的信息，推测 Enclave 内容。

4. 影子分支攻击（Branch Shadowing Attack）。该攻击向量类似于分支预测器侧信道，但它使用影子分支提取被泄露的信息，传统分支预测器侧信道使用缓存侧信道提取被泄露的信息。
5. 页表攻击。攻击者观察 Enclave 在内核页表中泄露的信息（如页错误时会泄露页地址），推测 Enclave 内容。
6. 恶意线程调度。攻击者利用 Ring0 权限恶意调度（暂停及恢复）Enclave 线程执行，触发非预期并发状态并形成攻击。
7. 时间侧信道。攻击者观察 Enclave 执行过程中不同粒度的时延信息，推测 Enclave 内容。

密封文件等安全机制对安全场景考虑不充分，存在安全缺陷，向攻击者暴露攻击面。相关攻击向量如文件回滚攻击：当 Enclave 操作密封文件时，不可信环境向 Enclave 内提供旧文件，将 Enclave 状态回滚到旧状态，形成攻击。

### 2.2.3 代表性攻击

本小节阐述代表性攻击，指出每个代表性攻击所针对的安全问题、所利用的攻击向量及所涉及的攻击面。

Iago attacks<sup>[12]</sup>中，攻击者恶意构造系统调用 OCALL（Enclave 无法提供系统调用）返回值影响 Enclave 状态，形成攻击。针对“Enclave 开发中的安全问题”，用到“恶意篡改输入”攻击向量，涉及“Enclave 接口”攻击面。

Lee S 等人的工作<sup>[13]</sup>中，攻击者观察 Enclave 内传出的数据结构中未初始化的填充数据，获得残留的敏感数据。针对“Enclave 开发中的安全问题”，用到“EDL 中未初始化填充数据”攻击向量，涉及“Enclave 接口”攻击面。

A Tale of Two Worlds<sup>[14]</sup>总结了针对 SGX 软件栈的 ABI 和 API 攻击，实现了新型的 ABI 攻击（通过修改标志寄存器）及 API 攻击（对 SGX 帮助性函数进行时间侧信道）。针对“Enclave 开发中的安全问题”和“SGX 设计存在安全缺陷”，用到“恶意篡改输入”和“时间侧信道”等攻击向量，涉及“Enclave 接口”和“可观测的物理信息”等攻击面。

COIN Attacks<sup>[15]</sup>归纳了并发调用攻击、调用排序攻击、恶意篡改输入和嵌套调用攻击这四种透过 Enclave 接口攻击面的攻击向量，基于此开发了 SGX 应用漏洞自动挖掘工具。这些攻击向量针对“Enclave 开发中的安全问题”，涉及

“Enclave 接口”攻击面。“调用排序”攻击向量如图 2.4 所示，攻击者攻占 SGX 应用不可信部分，恶意顺序调用 ECALL 以绕过特定接口检查，触发非预期的控制流。如攻击者利用 ECALL C 中的状态设置操作绕过 ECALL B 的状态检查。

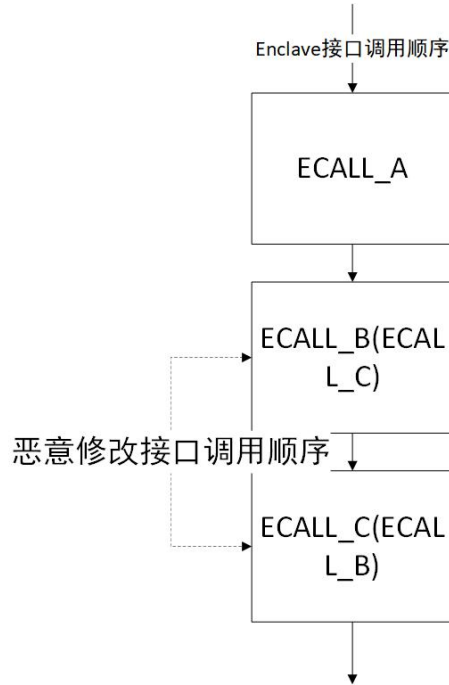


图 2.4 调用排序攻击示意图

Figure 2.4 Diagram of Call Permutation Attack

Wang J 等人的工作中<sup>[16]</sup>中，攻击者通过侧信道手段观察受害者 Enclave 的接口调用顺序、参数及时延，推测受害者 Enclave 的控制流。针对“SGX 设计存在安全缺陷”，用到“接口侧信道”攻击向量，涉及“Enclave 接口”和“可观测的物理信息”等攻击面。

Foreshadow<sup>[17]</sup>中，攻击者手动构造页错误绕过 SGX 中止页面语义，瞬态执行未被授权执行的代码，将 Enclave 敏感信息泄露到 L1 缓存，窃取泄露信息。针对“SGX 设计存在安全缺陷”，用到“缓存侧信道”攻击向量，涉及“微架构共享资源”攻击面。

SgxPectre<sup>[18]</sup>是 Spectre 攻击在 SGX 下的复现，攻击者恶意训练分支预测器，受害者 Enclave 在执行过程中分支预测错误，瞬态执行非预期分支，将敏感信息泄露到缓存，攻击者窃取泄露信息。针对“SGX 设计存在安全缺陷”，用到“分支预测器侧信道”和“缓存侧信道”等攻击向量，涉及“微架构共享资源”攻击

面。

Branch Shadowing<sup>[19]</sup>中，由于 Enclave 内外共享分支预测器及 Enclave 在退出时不刷新分支预测器，攻击者构建地址碰撞的影子分支（Shadow Branch），感知受害 Enclave 分支选择情况。针对“SGX 设计存在安全缺陷”，用到“影子分支攻击”攻击向量，涉及“微架构共享资源”攻击面。

Controlled-channel attacks<sup>[20]</sup>中，攻击者清除受害 Enclave 的“页存在（Present）位”，使受害 Enclave 执行过程中触发页错误，泄露错误页地址，攻击者观察错误页地址推测 Enclave 内页粒度执行流。针对“SGX 设计存在安全缺陷”，用到“页表攻击”攻击向量，涉及“部分功能依赖不可信内核”攻击面。

SGX-PTE<sup>[21]</sup>中，攻击者利用 A/D 通道【利用页访问/脏（Access/Dirty）位构建侧信道】及针对 PTE 的缓存侧信道，观察页地址访问情况，推测 Enclave 内页粒度控制流。针对“SGX 设计存在安全缺陷”，用到“页表攻击”和“缓存侧信道”攻击向量，涉及“部分功能依赖不可信内核”和“微架构共享资源”攻击面。

AsyncShock<sup>[22]</sup>中，攻击者利用 Ring0 权限恶意调度受害 Enclave 多个线程，触发非预期并发控制流，构建 UAF（Use after Free）和 TOCTOU（Time to Check to Time to Use）攻击。AsyncShock 如图 2.5 所示，线程 2 释放一个指针后，需要将其置 NULL 避免悬空指针（Dangling Pointer）存在，但攻击者可通过“Enclave 线程暂停原语”（采用页错误、APIC 时钟中断和 Linux Signal 等方法）将线程 2 暂停在释放指针后及置 NULL 前，调度线程 1 执行 ECALL 使用悬空指针，构建 UAF 攻击。主要针对“SGX 设计存在安全缺陷”，用到了“恶意线程调度”攻击向量，涉及“部分功能依赖不可信内核”攻击面。



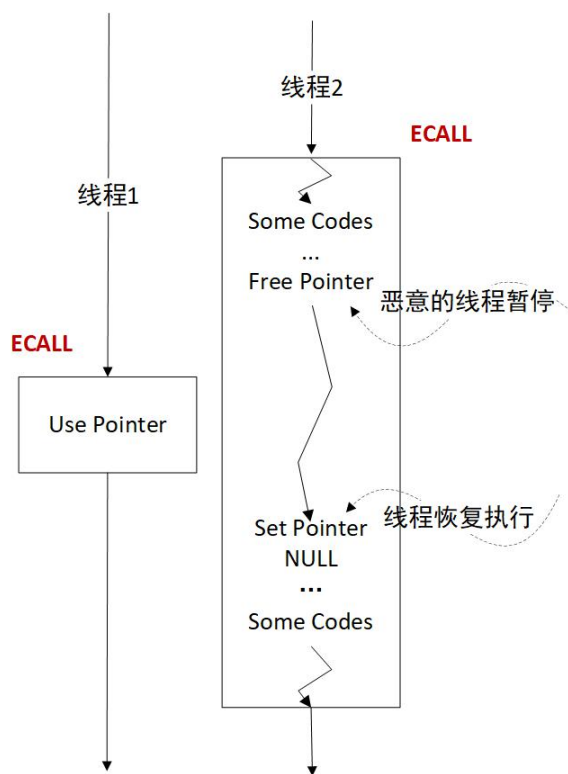


图 2.5 AsyncShock 攻击示意图

Figure 2.5 Diagram of AsyncShock Attack

Game of Threads<sup>[23]</sup>中，攻击者利用“页错误型线程暂停原语”恶意调度受害 Enclave 多个线程，对 ASGD（异步随机梯度下降）场景进行攻击。针对“SGX 设计存在安全缺陷”，用到“恶意线程调度”攻击向量，涉及“部分功能依赖不可信内核”攻击面。

Nemesis<sup>[24]</sup>中，攻击者利用 APIC 时钟中断按一定的时间间隔单步中断 Enclave 汇编代码执行，测量每条汇编代码执行时长构建时延序列，将时延序列与先验的 Enclave 各分支时延序列对比，推测 Enclave 分支选择情况。针对“SGX 设计存在安全缺陷”，用到“时间侧信道”攻击向量，涉及“可观测的物理信息”攻击面。

RIDL<sup>[25]</sup>中，攻击者利用受害者在微架构 LFB（Line Fill Buffer）泄露的信息实现侧信道攻击。针对“SGX 设计存在安全缺陷”，用到“LFB 侧信道”攻击向量，涉及“微架构共享资源”攻击面。

本文将 Enclave “恶意线程调度”攻击向量中恶意暂停线程执行的方法命名为“Enclave 线程暂停原语”，包括：

1. 清除目标 Enclave 页存在位/访问位等触发页错误，暂停 Enclave 线程执行。
2. 设置 APIC 时钟中断，暂停 Enclave 线程执行。
3. 利用 Linux Signal（如 SIGSEGV、SIGUSR1 和 SIGUSR2）暂停 Enclave 线程执行。

本文将代表性 SGX 攻击归纳成表 2.1 至 2.5。

**表 2.1** “Enclave 接口” 攻击表

**Table2.1** Table of attacks on "Enclave interface"

攻击向量（安全问题）	“Enclave 接口” 攻击
并发调用（Enclave 开发中的安全问题）	COIN Attacks
调用排序（Enclave 开发中的安全问题）	COIN Attacks
恶意篡改输入（Enclave 开发中的安全问题）	Iago attacks、A Tale of Two Worlds、COIN Attacks
嵌套调用（Enclave 开发中的安全问题）	COIN Attacks
EDL 中未初始化填充数据（Enclave 开发中的安全问题）	Leaking Uninitialized Secure Enclave Memory via Structure Padding
接口侧信道（SGX 设计存在安全缺陷）	Interface-based side channel attack against intel SGX

**表 2.2** “微架构共享资源” 攻击表

**Table2.2** Table of attacks on "shared resources of micro-architecture"

攻击向量（安全问题）	“微架构共享资源” 攻击
缓存侧信道（SGX 设计存在安全缺陷）	Foreshadow 、 SgxPectre 、 SGX-PTE
分支预测器侧信道和影子分支攻击（SGX 设计存在安全缺陷）	SgxPectre、Branch Shadowing
LFB 侧信道（SGX 设计存在安全缺陷）	RIDL

表 2.3 “部分功能依赖内核”攻击表

Table 2.3 Table of attacks on "partial functions depend on kernel"

攻击向量（安全问题）	“部分功能依赖内核”攻击
页表攻击（SGX 设计存在安全缺陷）	Controlled-channel attacks、SGX-PTE
恶意线程调度（SGX 设计存在安全缺陷）	AsyncShock、Game of Threads

表 2.4 “可观测的物理信息”攻击表

Table 2.4 Table of attacks on "observable physical information"

攻击向量（安全问题）	“可观测的物理信息”攻击
接口侧信道（SGX 设计存在安全缺陷）	Interface-based side channel attack against intel SGX
时间侧信道（SGX 设计存在安全缺陷）	A Tale of Two Worlds、Nemesis

表 2.5 “密封文件”攻击表

Table 2.5 Table of attacks on "sealed file"

攻击向量（安全问题）	“密封文件”攻击
文件回滚攻击（SGX 设计存在安全缺陷）	文件回滚攻击 <sup>[1]</sup>

### 2.3 传统系统防御方案

针对 SGX 安全问题、攻击向量及攻击面，可以借鉴传统系统防御方案的思路。传统系统防御方案中，有针对特定攻击类型以补丁形式防御的方案。如 Dangling Pointer Tagging 和 Red-zone Inserting 缓解 UAF 和栈溢出等内存安全问题，CFI 和 DFI 增强目标应用程序控制流/数据流的完整性缓解控制流/数据流劫持，ASLR 和 KASLR 将程序和内核加载地址随机化增加攻击者定位漏洞难度，SMEP、SMAP 和 KPTI 单向或双向隔绝用户态和内核态代码数据抵御针对内核的攻击。但这些防御方案安全准则不统一，难以被系统同时支持。

还有防御方案深入分析系统，加固系统自身安全。如访问控制领域中，Matetic S 等人对 Linux 系统的访问控制模型及关键访问控制点位深入分析，提出 Linux Security Module<sup>[39]</sup>（缩写为 LSM）访问控制框架，将安全准则与实施框架分离，

高效兼容多种安全准则。SELinux<sup>[40]</sup>（Security-Enhanced Linux）基于 LSM 进行了具体实现。LSM 使 Linux 同时支持了 Domain and Type Enforcement 和 Linux capabilities 等多种访问控制模型，启发了 AppArmor、TOMOYO 和 YAMA 等安全增强模块产生。实现上，LSM 在 Linux 关键执行路径中安插了许多 Hook 点。基于 LSM 实现的访问控制模型选用 Hook 点并自定义安全策略来实施主客体访问控制。Hook 点上获取的主客体访问控制信息会交给策略引擎（Policy Engine）决断，确定当前访问是否符合安全策略。

框架性的防御方案深入分析系统，加固系统自身安全，相较于针对性的防御方案更加原生，兼容多种安全准则的能力更强，提供更高安全性，启发新的安全准则。

## 2.4 SGX 防御现状

针对 SGX 攻击，学术界及工业界均提出了防御方案，用到的防御方法包括 Enclave 接口消毒、Enclave 内外资源隔离、痕迹防泄露和攻击痕迹检测等。尚无防御方案系统分析 SGX 架构及安全关键点，加固 SGX 架构自身安全。

### 2.4.1 Enclave 接口消毒

Enclave 接口消毒指对 Enclave 接口进行安全检查，避免攻击者从 Enclave 接口输入恶意数据危害 Enclave。代表性防御方案包括 SGX 原生软件栈提供的和学术界提出的防御方案。SGX 原生软件栈提供了 EDL 关键字及 SGX 帮助性函数，由于无法先验地知道 Enclave 接口具体用法，因此只能够简单消毒 Enclave 接口。

EDL 关键字。描述 Enclave 边界的 EDL 文件中，Enclave 开发者可以设置 EDL 关键字，对跨越 Enclave 内外的函数和传参进行简单限定。如 ECALL 中“`In`”关键字可以修饰一个原指向 Enclave 外缓冲区的指针，SGX 软件栈会将指针指向的缓冲区内容拷贝进 Enclave，Enclave 内使用指向 Enclave 内缓冲区的指针。“`In`”关键字防止 Enclave 执行时 Enclave 外通过修改缓冲区直接影响 Enclave 内部状态。但 EDL 关键字缺乏针对函数和传参的先验场景知识，接口消毒能力有限，复杂接口消毒功能需要 Enclave 开发者实现。

SGX 帮助性函数（主要指 `sgx_is_within_enclave` 和 `sgx_is_outside_enclave` 函数）是 SGX 软件栈提供的一类函数，帮助 Enclave 判断变量位于 Enclave 内部

(`sgx_is_within_enclave`) 还是 Enclave 外部 (`sgx_is_outside_enclave`)，方便开发者进行接口检查。

学术界代表防御方案包括 Haven<sup>[26]</sup>、Graphene-SGX<sup>[27]</sup>、SCONE<sup>[28]</sup>、Ryoan<sup>[29]</sup> 和 Glamdaring<sup>[30]</sup>等。

Haven<sup>[26]</sup>(如图 2.6 所示)基于 Drawbridge 在 Enclave 中内嵌 LibOS,使 Enclave 内能二进制级别兼容非 SGX 应用, Shield 模块通过 20 多个接口向 Host 内核请求 Enclave 内无法实现的功能。针对 Iago Attack 等攻击透过 Enclave 接口攻击面展开攻击, Haven 在 Shield 模块中基于先验 LibOS 场景检查接口传参安全。

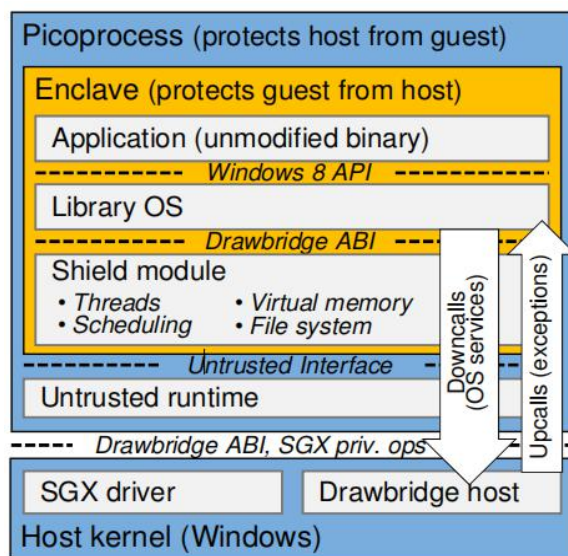


图 2.6 Haven 组件和接口<sup>[26]</sup>

Figure 2.6 Components and interfaces of Haven<sup>[26]</sup>

Graphene-SGX<sup>[27]</sup> (如图 2.7 所示)类似于 Haven, 在 Linux 系统下的 SGX Enclave 中内嵌 LibOS, 二进制级别兼容非 SGX 应用和 Docker, 基于先验 LibOS 场景检查 28 个接口安全。文件读取(通过文件校验码)和进程间协作(通过认证和加密)等 18 个接口被全面检查。

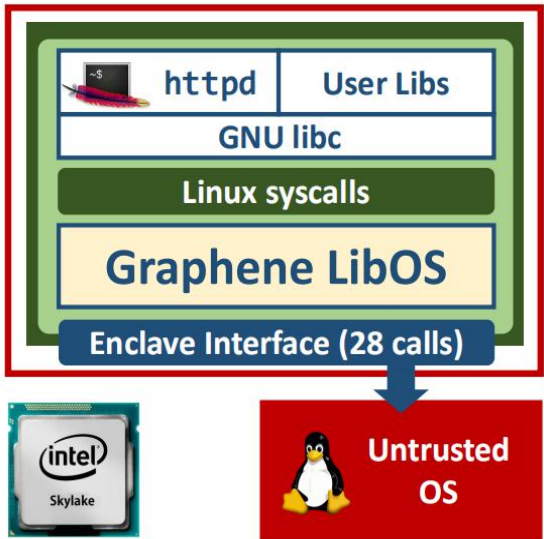


图 2.7 Graphene-SGX 架构图<sup>[27]</sup>

Figure2.7 Graphene-SGX architecture diagram<sup>[27]</sup>

SCONE<sup>[28]</sup>（如图 2.8 所示）源码级别兼容非 SGX 应用（需要重新编译）。将 Graphene-SGX 和 Haven 中 LibOS 及 C 库移出 Enclave, 通过 Shim 层（SCONE C Library）间接调用 Enclave 外 C 库，使 Enclave 保持较小 TCB。Shield 层向下调用 HostOS 功能，向上提供文件系统等功能，基于先验文件系统等功能设置接口消毒。

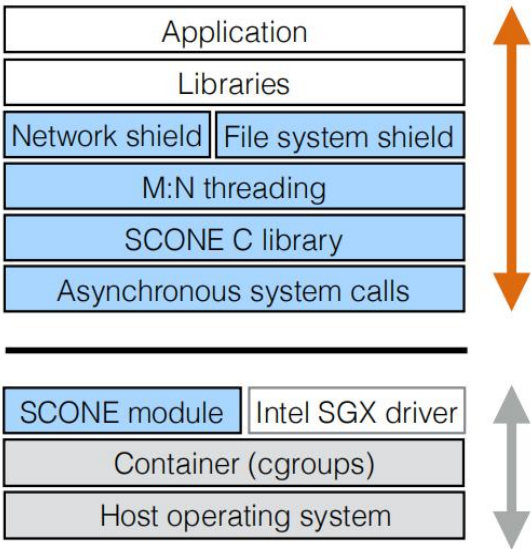


图 2.8 SCONE 架构图<sup>[28]</sup>

Figure2.8 SCONE architecture diagram<sup>[28]</sup>

Ryooan<sup>[29]</sup>构建 Enclave 内沙箱并检查 Enclave 接口,防止服务提供商的 Enclave 恶意向 Enclave 外传递用户隐私数据。

Glamdaring<sup>[30]</sup>自动划分非 SGX 应用的可信部分和非可信部分,组装成 SGX 应用,它通过减少 Enclave 接口数量缓解调用排序攻击向量。

#### 2.4.2 Enclave 内外资源隔离、痕迹防泄露及攻击痕迹检测

为避免侧信道等攻击利用“Enclave 内外共享资源”攻击面,有防御方案基于资源隔离、执行痕迹防泄露和攻击痕迹检测等思路展开防御。代表性防御方案包括学术界提出的及 Intel 提出的防御方案。学术界的防御方案包括 T-SGX<sup>[31]</sup>和 Racing in Hyperspace<sup>[32]</sup>等。

T-SGX<sup>[31]</sup>利用 Intel TSX 事务代码出错回滚特性,将页错误泄露的页地址信息限制到跳板页地址。针对会触发大量 AEX 的 Controlled-channel Attack,它在跳板页记录 AEX 频率并预警高频 AEX (即检测攻击痕迹)。

Racing in Hyperspace<sup>[32]</sup>针对超线程侧信道攻击,度量受害者两个线程间通讯时长,判断是否在兄弟核上(同一个物理核的两个逻辑核),若通讯时长超出阈值,那么受害者两个线程不在兄弟核上,即可能有攻击者抢占受害者的兄弟核并发起超线程攻击。

OBFUSCURO<sup>[33]</sup>将代码数据访问模式、分支模式及代码块执行时长固定化,只留下相同的访问痕迹,避免攻击者观察访问模式差别推测 Enclave 控制流。

Eleos<sup>[34]</sup>实验证明进出 Enclave 时切换上下文开销巨大,通过类 RPC 机制及 SUVM (Secure User Virtual Memory, 即 Enclave 内页表机制)避免频繁进出 Enclave。SUVM 机制还能避免页错误地址信息泄露到 Enclave 外,缓解页表攻击。

CoSMIX<sup>[35]</sup>使用 LLVM Pass 插桩 Enclave 源码,实现了多层 Memory Store (基于底层物理内存、内核虚拟内存或自定义虚拟内存构建新一层自定义虚拟内存)栈式堆叠,满足多种(安全)需求,如 ORAM 和 Enclave 内页表机制的双重堆叠。

Autarky<sup>[36]</sup>针对 Controlled-Channel Attack,修改软硬件使 Enclave 内页错误处理优先于 OS 页错误处理,Enclave 内页错误处理中,Enclave 结合内部记录判断当前异常是攻击者伪造的还是正常触发的。Autarky 中还可以编写策略检测攻击。

Gruss D 等人的工作<sup>[37]</sup>针对 TAA (TSX Asynchronous Abort 利用 TSX 回滚事务时不回滚微架构的缺陷) 攻击向量, 在封装敏感代码的事务入口处加载冗余缓存, 增加攻击噪声。

Intel 公司通过微码补丁和硅上改 (Silicon-based Change) 对危害严重的攻击进行防御。

Intel 修改 SGX 硬件指令微码流 (SGX 指令是由若干微码组成的微码流, 执行时被翻译成微码并被发送到执行单元执行) 实施防御措施。如针对缓存侧信道, 在进出 Enclave 所用指令微码流中增加 L1D 缓存刷新操作。

Intel 通过硅上改重新设计 Intel CPU 微架构, 缓解某些侧信道。如修改乱序执行中异常提交和状态回滚相关机制。

### 2.4.3 其他防御方案

有研究人员提出防御方案, 缓解透过“密封文件”攻击面的文件回滚攻击。

ROTE<sup>[38]</sup>基于拜占庭容错系统思路, 将标识文件新旧的版本号存储于分布式主机群中, 当 Enclave 读取密封文件时, 比对密封文件的版本号和主机群中读出的版本号, 避免使用旧版本文件引起文件回滚攻击。

SGX Counter。Intel 公司在支持 SGX 的 CPU 中添加了 SGX Counter (单调递增计数器), 记录每个密封文件的最新状态。当不可信环境提供旧文件时, Enclave 能检测到旧文件标识号与 SGX Counter 值不一致。

## 2.5 本章小结

本章阐述了 SGX 编程模型、软硬件架构和原生安全机制, 指出 SGX 的特殊之处。归纳了 SGX 攻击所针对的安全问题包括“Enclave 开发中的安全问题”和“SGX 设计存在安全缺陷”, 所透过的攻击面包括“Enclave 接口”和“共享资源”等, 所利用的攻击向量包括调用排序、并发调用等。阐述了代表性攻击及其相关的安全问题、攻击面和攻击向量。分析了传统系统防御方案的思路, 指出框架性防御方案的优势。归纳现有 SGX 防御方法包括接口消毒、资源隔离、痕迹防泄露和攻击痕迹检测等。指出尚无防御方案系统分析 SGX 架构及安全关键点, 加固 SGX 架构自身安全。



### 第三章 Intel SGX 关键执行路径及安全关键点分析

本章刻画 SGX 架构，分析 SGX 关键执行路径及安全关键点，总结 SGX 线程模型，为 SGX 架构安全增强提供着力点。主要围绕“Enclave 开发中的安全问题”分析 SGX 软件栈关键执行路径及安全关键点，分析方法也适用于其他类型安全问题。本文将安全关键点中记录的安全信息称为安全事件或事件。

#### 3.1 Enclave 接口关键执行路径

分析 Enclave 接口关键执行路径，能够有效定位 SGX 架构中不可信环境与可信环境交界面上的安全关键点，为 SGX 架构安全增强提供着力点。

##### 3.1.1 ECALL 进入 Enclave

不可信环境线程通过 ECALL 进入 Enclave 访问 Enclave 内容，ECALL 实现方式包括 Ordinary 方式和 Switchless 方式。

Ordinary ECALL 代码流程如图 3.1 所示。位于 SGX 应用不可信部分的普通线程调用 ECALL Stub 函数进入 uRTS，调用 EENTER 指令（ENCLU.02H）切换上下文进入 tRTS，变成 Enclave 线程，最终真正执行开发者编写的 ECALL 代码。

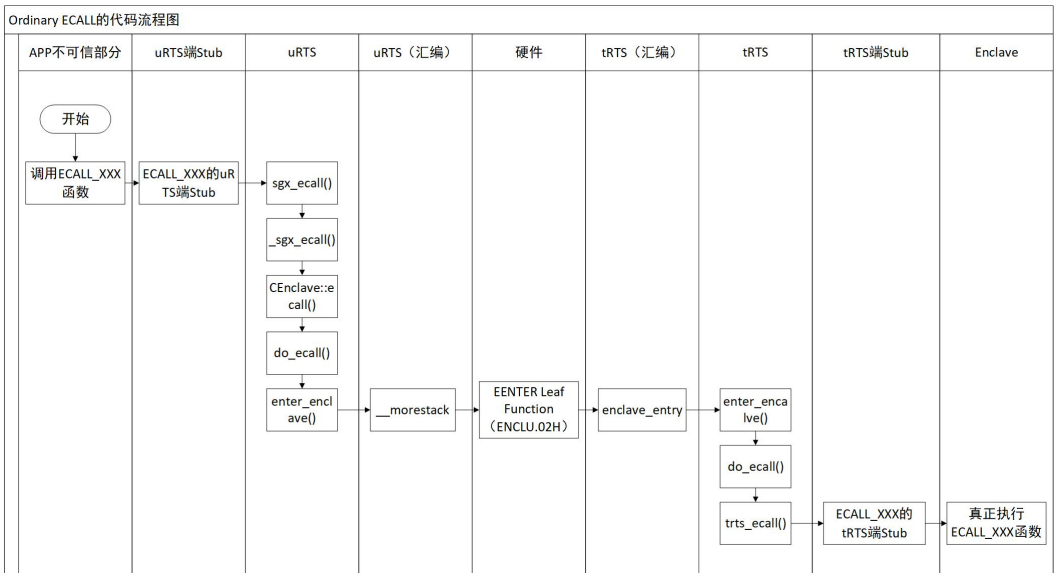


图 3.1 Ordinary ECALL 代码流程图

Figure3.1 Flowchart of Ordinary ECALL

Ordinary ECALL 的关键在于线程调用 EENTER 指令切换上下文进入 Enclave。在切换上下文时，线程（物理上对应一个 CPU 核）通过元数据寄存器标记它从普通模式转变为 Enclave 模式，将不可信上下文的部分内容备份，并将线程上下文完全替换成 Enclave 内的上下文。

Ordinary 方式是最早提出的 ECALL 方式。为避免 Ordinary 方式切换上下文带来巨大开销，SGX 软件栈新增 Switchless 方式<sup>[41,42]</sup>。

Switchless ECALL 代码流程如图 3.2 所示。APP 不可信部分的普通线程执行 Switchless ECALL 时，将 ECALL 封装成任务放到 ECALL 任务池中，通过信号线(Signal Line)通知 tRTS 端的 TWorker(可信工人线程)提取并代理执行 ECALL 任务，等待 TWorker 执行完 ECALL 任务。如果 Switchless ECALL 调用者不等待 TWorker 执行完 ECALL 任务并继续执行剩余代码，那么程序执行流将不符合开发者预期。

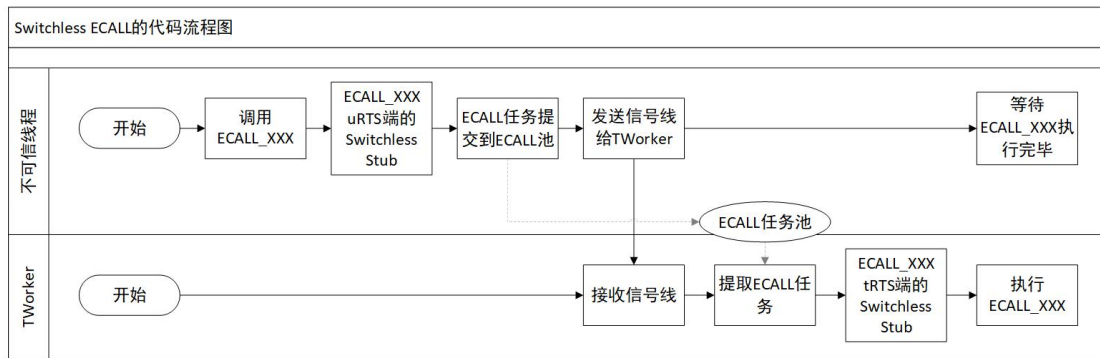


图 3.2 Switchless ECALL 的代码流程图

Figure3.2 Flowchart of Switchless ECALL

Switchless ECALL 的关键在于有一些预先切换上下文进入 Enclave 内的 TWorkers 帮助 Switchless ECALL 调用者代理完成 ECALL 任务。TWorkers 自身的一次上下文切换，可以避免 ECALL 调用者 N 次的上下文切换。根据论文<sup>[42]</sup>的描述，当 ECALL/OCALL 函数代码量较少且被频繁调用时，占用工人线程带来的性能开销低于上下文切换的性能开销，Switchless 方式在性能上会优于 Ordinary 方式。如公式 1 所示，当可信代码执行时长与不可信代码执行时长之和等于上下文切换时长时，两种方式的性能相同。

$$Time(Trust edCode) + Time(UntrustedCode) = Time(ContextSwitch)$$

$$\Leftrightarrow Performance(Ordinary) = Performance(Switchless) \quad (1)$$

“Switchless/Ordinary ECALL”中攻击者能通过恶意传参等方法攻击 Enclave。“Switchless/Ordinary ECALL”是一个安全关键点，对此在其 tRTS 端（确保环境位置可信）插桩代码审计检查“ECALL 进入 Enclave”相关事件，审计检查的安全事件信息包括寄存器、ECALL 名、ECALL 序号、ECALL 方向、Ordinary/Switchless 和传参等。

### 3.1.2 ECALL 结束退出 Enclave

ECALL 执行完后，控制流会回到 ECALL 调用者的下一行代码。Ordinary ECALL 和 Switchless ECALL 的结束过程描述如下。

Ordinary ECALL 结束过程。ECALL 执行完后，线程从 tRTS 端 Stub 逐级返回并清理上下文信息，调用 EEXIT 指令切换上下文返回 uRTS 端，将控制流归还给 ECALL 调用者。

Switchless ECALL 结束过程。TWorker 执行完 ECALL 后，将函数执行结果放回到 ECALL 任务池中，通知 Switchless ECALL 调用者 ECALL 任务已执行完。Switchless ECALL 调用者停止等待并执行后续代码。TWorker 空转等待 Switchless ECALL 调用者提交新任务，若空转次数超过（默认 20000 次）阈值（即一定时间内没有新的 ECALL 任务）就返回 tRTS，睡眠等待新的 ECALL 任务。当有新的 ECALL 任务时，TWorker 重新进入 Enclave 并代理执行 ECALL 任务。

“Switchless/Ordinary ECALL 返回”中，攻击者能恶意泄露 Enclave 内敏感内容及构建非法调用顺序等。“Switchless/Ordinary ECALL 返回”是一个安全关键点，对此在其 tRTS 端插桩代码审计检查“ECALL 结束退出 Enclave”相关事件，可审计检查的安全事件信息包括寄存器、ECALL 名、ECALL 序号、ECALL 方向、Ordinary/Switchless 和返回值等。

### 3.1.3 OCALL 离开 Enclave 及 OCALL 结束返回 Enclave

Enclave 内部无法实现部分功能（系统调用和 CPUID 指令等），但能通过 OCALL 调用 Enclave 外提供的功能。

“OCALL 离开 Enclave”在形式上与“ECALL 进入 Enclave”类似，但方向和目的相反，OCALL 方式包括 Ordinary 方式和 Switchless 方式。Ordinary OCALL

中, Enclave 内线程调用 EEXIT 指令切换上下文前往 Enclave 外执行不可信函数。Switchless OCALL 中, Enclave 内线程将 OCALL 任务交给 Enclave 外的不可信工人线程 UWorker 代理执行。

由于 ECALL 内可以调用 OCALL 及 OCALL 内可以调用 ECALL, Enclave 线程栈中会出现 ECALL 栈帧和 OCALL 栈帧堆叠的情况 (即嵌套调用)。

“OCALL 结束返回 Enclave”在形式上与“ECALL 结束退出 Enclave”类似,但方向和目的相反。由于 OCALL 返回值能被不可信环境篡改, Enclave 需要基于先验知识检查不可信的 OCALL 返回值。

在“OCALL 离开 Enclave”及“OCALL 结束返回 Enclave”中,攻击者能恶意输入数据、恶意泄露 Enclave 内容及构建非法调用顺序等。“OCALL 离开 Enclave”及“OCALL 结束返回 Enclave”是安全关键点,对此在其 tRTS 端插桩代码审计检查“OCALL 离开 Enclave”及“OCALL 结束返回 Enclave”相关事件,可审计检查的安全事件信息包括寄存器、OCALL 名、OCALL 序号、OCALL 方向、Ordinary/Switchless、传参和返回值等。

#### 3.1.4 AEX 离开 Enclave 及异常处理完成返回 Enclave

Enclave 异常处理流程与传统异常处理流程不同。非 SGX 应用中出现异常时, CPU 立即让 OS 完成异常处理。而 Enclave 中出现异常时, CPU 首先执行 AEX (Asynchronous Enclave Exit, 异步 Enclave 退出), AEX 将 Enclave 上下文保存到 SSA (State Save Area, 状态保存区域, 异常处理完成返回 Enclave 时从 SSA 中恢复上下文), 切换上下文退出 Enclave, 进入内核态执行内核态异常处理句柄。AEX 中包含 Enclave 内到 Enclave 外的切换及 Ring3 到 Ring0 的切换。AEX 保护 Enclave 信息在中断时不被泄露。AEX 及异常处理流程如下图 3.3 所示。

1. AEX 及进入内核异常处理句柄。
2. 若内核无法处理异常, 会发出 Signal 告知用户态。位于用户态的 uRTS 针对 SIGSEGV、SIGFPE、SIGILL、SIGBUS 和 SIGTRAP 五种信号注册了信号处理句柄。
3. 信号处理句柄中, 线程以 ECMD\_EXCEPT 标记调用 EENTER 指令进入 tRTS, 线程栈包含两个 ECALL 栈帧, 普通 ECALL 栈帧和 ECMD\_EXCEPT ECALL 栈帧。

4. 线程在 tRTS 端处理完异常后调用 EEXIT 指令退回到信号处理句柄。
5. 信号处理句柄完成相关操作后返回到内核错误句柄。
6. 内核错误句柄通过 IRET (Interrupt Return) 返回到 AEP。返回到 AEP 的原因是 AEX 时将 AEP 赋值给内核栈中的 RIP。
7. AEP 调用 ERESUME 指令回到 Enclave，从中断处继续执行代码。

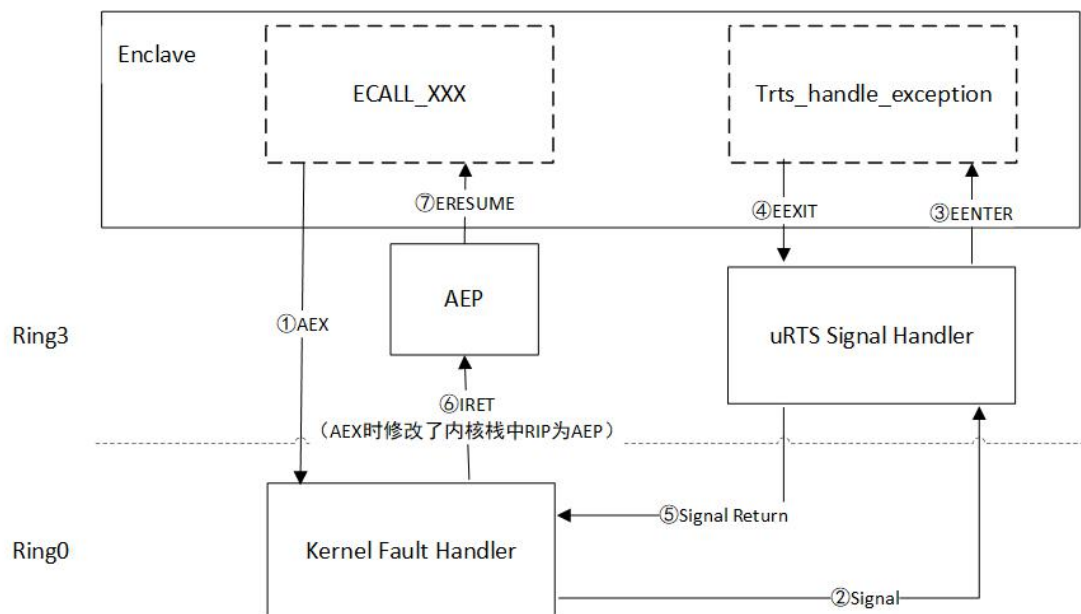


图 3.3 AEX 及异常处理流程图

Figure3.3 Flowchart of AEX and exception handling

“AEX 离开 Enclave”及“异常处理完成返回 Enclave”是重要的安全关键点，但现有防御方案并未深入分析。对此，在 AEX 触发处和 tRTS 异常处理句柄处插桩代码审计检查“AEX 离开 Enclave”及“异常处理完成返回 Enclave”相关事件，可审计检查的安全事件信息包括寄存器、异常相关信息和 AEX 方向等。

## 3.2 其他关键执行路径

除 Enclave 接口外对其它关键执行路径进行分析，提取其中的安全关键点，进一步增强 SGX 架构安全。例如，密封文件、加密信道和认证过程等是 tRTS 向 Enclave 提供的安全服务机制，它们的关键执行路径中存在安全关键点，需要安全加固。

tRTS 中受保护文件系统库向 Enclave 提供密封文件功能。如图 3.4 所示，不

可信线程进入 Enclave 后，通过受保护文件系统库完成文件密封操作。tRTS 将文件加密，通过 OCALL 让 uRTS 将加密文件存储到不可信外存中。

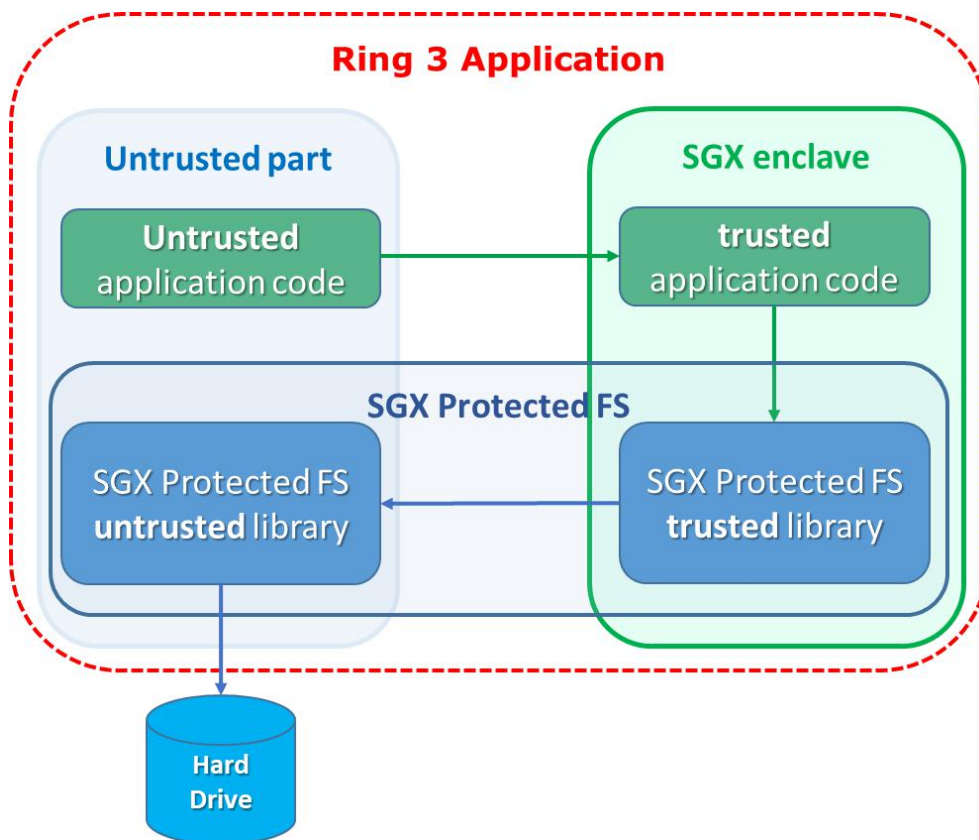


图 3.4 受保护文件系统布局<sup>[4]</sup>

Figure3.4 Protected File System Layout<sup>[4]</sup>

文件密封机制中进出 Enclave 及加解密文件处是安全关键点，对此插桩代码审计检查相关安全事件，防范恶意密封文件等。可审计检查的安全事件信息包括加密文件、明文文件、文件元数据和操作方向等。

本地认证过程如图 3.5 所示，两个 Enclave 间通过 uRTS 不断交换信息，验证对方本地认证报告，完成 Diffie-Hellman 密钥交换。

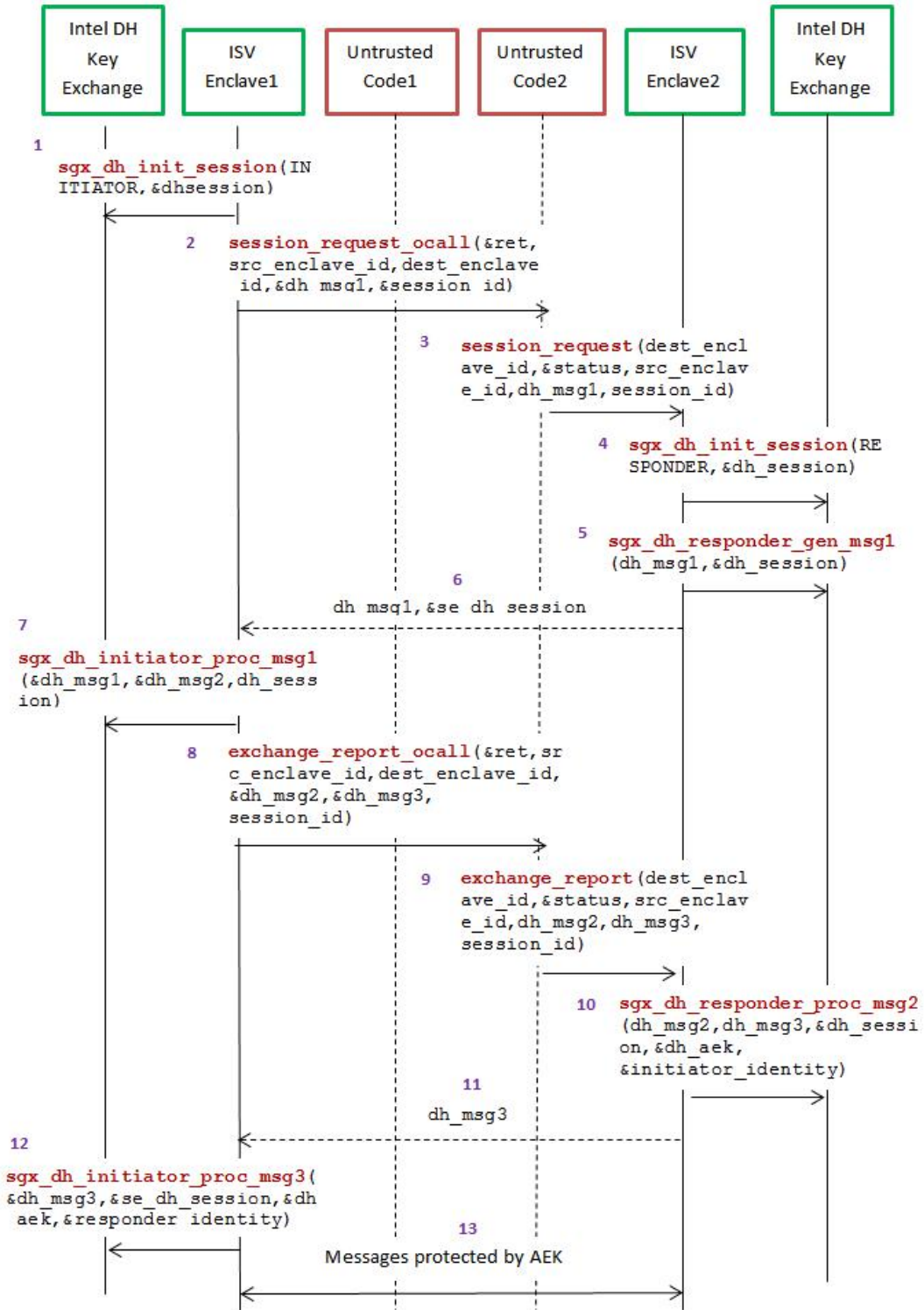


图 3.5 本地认证（使用 DH 密钥交换库）<sup>[4]</sup>

Figure3.5 Local Attestation Flow with the DH Key Exchange Library<sup>[4]</sup>

安全信道建立过程如图 3.6 所示，两个 Enclave 间通过 uRTS 不断交换信息，完成 Diffie-Hellman 密钥交换，使用协商的密钥构建安全信道。



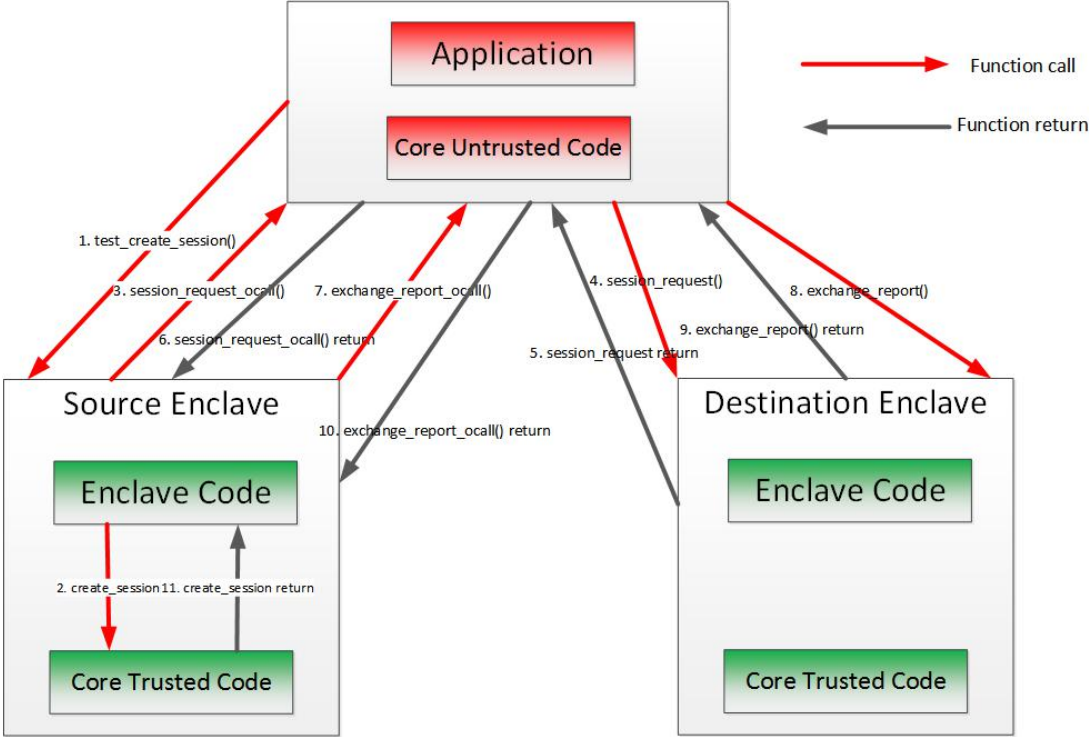


图 3.6 安全信道的建立（使用 DH 密钥交换库）<sup>[4]</sup>

Figure3.6 Secure Channel Establishment Flow with the DH Key Exchange<sup>[4]</sup>

这些安全机制中进出 Enclave 及加解密数据等处是安全关键点，对此在其 tRTS 端插桩代码审计检查相关安全事件。

3.3 SGX 线程模型

基于对 SGX 关键执行路径及安全关键点的分析，归纳总结 Ordinary/Switchless ECALL/OCALL 中线程模型，如图 3.7 所示。



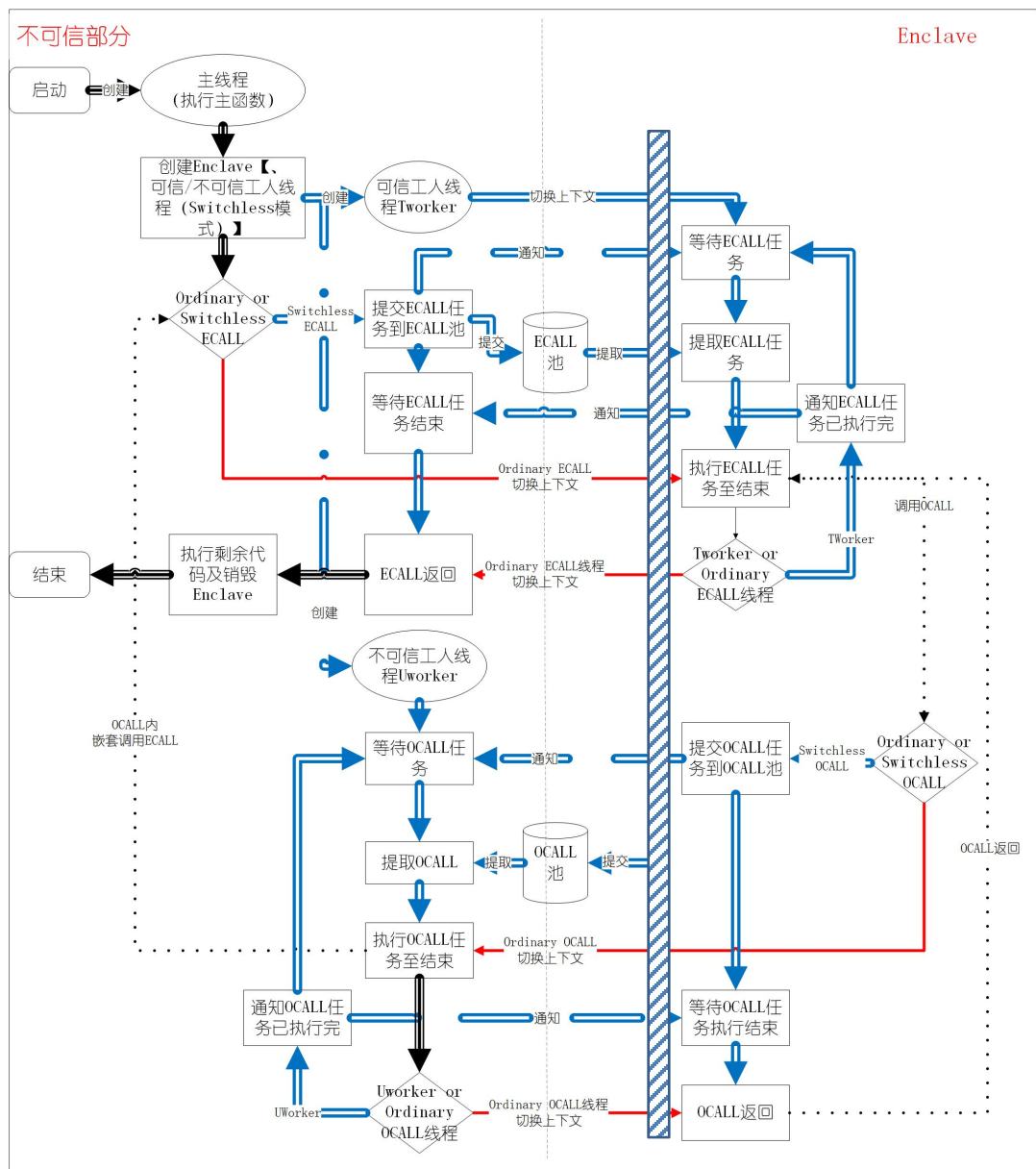


图 3.7 SGX 应用中 Ordinary/Switchless ECALL/OCALL 线程模型

Figure 3.7 Thread model of Ordinary/Switchless ECALL/OCALL in SGX application

基于图 3.7 及 SampleEnclave/Switchless 示例代码（属于 linux-sgx Github 仓库）分析 SGX 线程模型。

Ordinary ECALL/OCALL 线程模型：（基于 SampleEnclave 示例代码，图中红色单条细直线及黑色三条粗直线）

1. 主线程创建 Enclave 实例。
2. 主线程 Ordinary ECALL 切换上下文进入 Enclave: 主线程(物理上是 CPU

核) 占用一个已注册的 TCS (Thread Control Structure, 线程控制结构体) 调用 EENTER 指令进入 tRTS 【将 CR\_ENCALVE\_MODE 置 1 标识 CPU 处于 Enclave 模式, 将不可信栈帧寄存器 (u\_RSP 和 u\_RBP) 等上下文存储到 Enclave 内的 SSA (State Save Area, 状态存储区域)】, tRTS 清理通用寄存器, 设置 Enclave 线程栈为 Root ECALL 栈帧 (Enclave 线程栈独立于不可信线程栈)。

3. 被切换上下文的主线程执行 ECALL 内容。
4. ECALL 中, 线程调用 Ordinary OCALL 切换上下文到 Enclave 外: 线程在 tRTS 中将上下文封装成 OCALL 栈帧保存到 Enclave 线程栈中, 未来 OCALL 返回时从 Enclave 线程栈中恢复上下文 (Enclave 线程栈变成 “Root ECALL 栈帧+OCALL 栈帧”), 清理通用寄存器, 调用 EEXIT 指令 (恢复不可信栈帧, 将 CR\_ENCLAVE\_MODE 置 0) 退到 Enclave 外 uRTS 并执行 OCALL。
5. 线程执行 OCALL 函数。如果 OCALL 中嵌套调用 ECALL, Enclave 线程栈变成 “Root ECALL 栈帧+OCALL 栈帧+ECALL 栈帧”。
6. OCALL 执行结束, 线程返回 Enclave。线程调用 EENTER 指令返回到 Enclave, 从 OCALL 栈帧恢复上下文 (Enclave 线程栈变成 “Root ECALL 栈帧”)。
7. 线程执行 ECALL 剩下代码。当 ECALL 执行完毕时, 线程清理通用寄存器, 调用 EEXIT 指令 (恢复不可信栈帧寄存器) 退到 Enclave 外 uRTS。
8. 线程执行主函数剩余代码, 销毁 Enclave。

Enclave 外主线程可以创建若干线程并发进入 Enclave, 每个线程执行如上单线程场景。

Switchless ECALL/OCALL 线程模型: (基于 Switchless 示例代码, 图中蓝色双条粗直线加黑色三条粗直线。)

1. 主线程创建 Enclave 实例, 初始化 Switchless 模式。主线程在 Enclave 外创建若干线程, 一部分切换上下文进入 Enclave 成为可信工人线程 TWorker, 另一部分留在 Enclave 外成为不可信工人线程 UWorker。
2. 主线程调用 Switchless ECALL: 主线程将 ECALL 任务提交到 ECALL 任

务池并通知 TWorker，然后等待 ECALL 执行完。TWorker 收到通知，从 ECALL 任务池提取任务并执行。若无空闲 TWorker，Switchless ECALL 任务会退化，被主线程以 Ordinary 方式执行。

3. Switchless ECALL 中执行 Switchless OCALL：TWorker 将 OCALL 任务提交到 OCALL 任务池并通知 UWorker，然后等待 OCALL 执行完。UWorker 收到通知，从 OCALL 任务池提取任务并执行。若无空闲 UWorker，Switchless OCALL 任务会退化，被当前 TWorker 以 Ordinary 方式执行。
4. UWorker 通知 TWorker Switchless OCALL 任务已执行完，向 TWorker 传输函数返回值。TWorker 解除等待。
5. TWorker 执行 ECALL 剩余代码，直至 Switchless ECALL 任务执行完。TWorker 通知主线程 Switchless ECALL 任务已执行完，向主线程传输函数返回值。主线程解除等待。
6. 主线程执行主函数剩余代码，销毁 Enclave。

SGX 线程模型中，线程或者切换上下文执行目标代码，或者委托预先切换上下文的工人线程代理执行目标代码。线程切换上下文过程中，备份切换前的上下文，设置 CPU 内元数据寄存器值并转变模式，清理通用寄存器，启用新的上下文。

“Switchless/Ordinary ECALL/OCALL 及其结束”时，攻击者能够透过这些攻击面恶意输入输出信息。对此，在其 tRTS 端插桩代码审计检查相关安全事件（如图 3.7 斜纹竖长条柱所示）。

### 3.4 本章小结

本章分析了 Enclave 接口关键执行路径及其他关键执行路径，描述了其代码流程，提取其安全关键点。以 SampleEnclave 和 Switchless 示例代码描述 SGX 线程模型，归纳总结 Ordinary/Switchless ECALL/OCALL 中的线程模型。SGX 关键执行路径及安全关键点分析能指导审计检查代码插桩方式及位置、审计检查功能实施和已有安全准则兼容，还能启发新的安全准则。



## 第四章 针对 Intel SGX SDK 的安全增强框架设计

基于关键执行路径及安全关键点分析，选择针对安全关键点的审计检查代码插桩方式及插桩位置，设计针对安全事件的审计检查功能，提出针对 Intel SGX SDK 的安全增强框架（简称 SGX-SEF），加固 SGX 架构自身安全。本文将安全关键点上部署的审计检查代码称为审计检查点。

### 4.1 针对 Intel SGX SDK 的安全增强框架总体设计

#### 4.1.1 审计检查点插桩方式

为了审计检查 SGX 关键执行路径中的安全事件，需要对安全关键点插桩审计检查代码。本节对比分析了三种审计检查点插桩方式（如图 4.1 所示）：

1. 使用编译器插桩技术在 Enclave 代码插桩审计检查点。LLVM-Pass 等编译器插桩技术适合插桩模式固定的场景，操作简单且自动化，但需要先验的插桩模式。本文旨在对关键执行路径插桩，没有先验的固定模式。Enclave 代码不如 tRTS 能获取更多的安全事件信息。
2. 中间层（Shield 框架和沙箱）内嵌审计检查点。中间层丰富扩充了 Enclave 功能，但也引入大量 TCB。中间层不如 tRTS 能获取更多安全事件信息。
3. SGX 软件栈插桩。tRTS 中能获取更丰富的安全事件信息，实施更丰富的安全策略，但 SGX 软件栈插桩方式要求深入理解 SGX 软件栈。

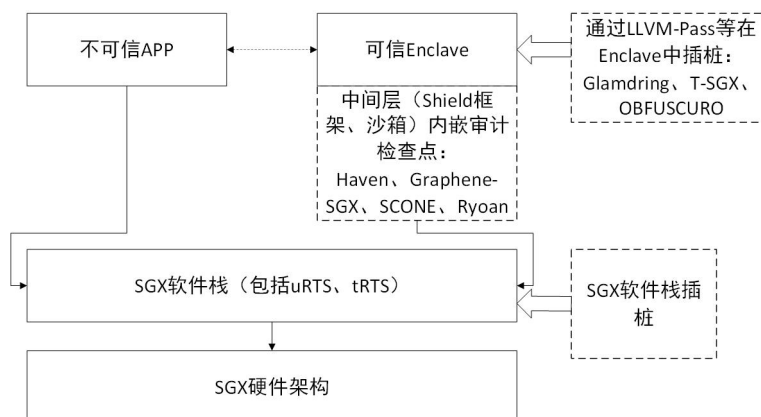


图 4.1 插桩方式对比

Figure4.1 Comparison of instrumentation methods

本文设计 SGX-SEF 时,针对关键执行路径上的安全关键点采用 SGX 软件栈插桩方式插桩审计检查代码。

#### 4.1.2 审计检查点插桩位置

一个安全关键点中存在若干代码点位可供插桩审计检查代码,但有些代码点位存在相关冲突,不适合插桩审计检查点。例如针对 OCALL,如果在 tRTS 端 OCALL 流程中插桩审计检查点,审计检查代码内嵌子 OCALL (如用于记录日志的文件操作和维护并发的互斥量/读写锁等功能需要通过 OCALL 调用系统调用),那么就可能出现非预期的错误。错误原因的关键点在于(如图 4.2 所示):

1. 带参数 OCALL 的 tRTS 端 Stub 函数中使用 `sgx_ocalloc` 函数将存储于 Enclave 内的参数复制到 Enclave 外,供 OCALL 后续使用。
2. 由于当前环境为 tRTS,不可信栈帧的 `u_RBP` 和 `u_RSP` 值(64 位寄存器, `u` 代表不可信)已被备份到 SSA 中,因此 `sgx_ocalloc` 会调整 SSA 中的 `u_RBP` 及 `u_RSP` 值将拷贝到 Enclave 外的参数包裹到不可信栈帧中。
3. 但是如果父 OCALL 的 `sgx_ocalloc` 与 `sgx_ocfree` 之间内嵌了子 OCALL (含 `sgx_ocalloc`),那么父 OCALL 对 `u_RSP` 的调整操作将丢失,导致父 OCALL 代码出错。

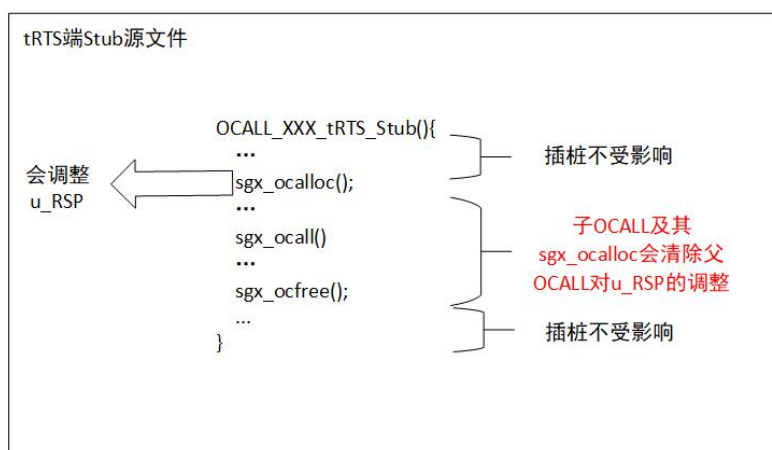


图 4.2 OCALL Stub 函数示意图

Figure4.2 Diagram of OCALL Stub function

因此, `sgx_ocalloc` 和 `sgx_ocfree` 之间不适合内嵌(包含 `sgx_ocalloc` 的)子 OCALL。SGX-SEF 将针对 OCALL 的审计检查点位定在了 tRTS 端 Stub 函数中

及 `sgx_ocalloc-sgx_ocfree` 外。

针对“Ordinary/Switchless ECALL/OCALL 及其结束”，修改 SGX Edger8r 工具将审计检查点自动部署到 tRTS 端各个 Stub 函数内。针对 AEX，在 tRTS 端异常处理函数（`trts_handle_exception`）中插桩审计检查点，但该审计检查点能被不可信内核绕过。针对 tRTS 提供的受保护文件系统库等，在其关键执行路径上插桩审计检查点，如在 `sgx_fread` 插桩审计检查点避免读取的文件内容存在恶意信息。

### 4.1.3 安全事件审计

为了对 SGX 安全关键点进行安全加固，本节设计了针对安全事件的审计功能，旨在将安全关键点上的安全事件逐一记录，以便防御者度量安全事件并分析攻击痕迹。

**审计功能：**当代码执行到审计检查点时，将当前安全事件的环境信息【包括当前事件类型（ECALL/OCALL/AEX/其他）、函数名、函数索引值（索引值唯一对应了其地址）和参数信息等】传入审计检查点，审计检查点将安全事件记录到 Enclave 内的全局日志中。针对全局日志，目前实现了两种存储方式：Enclave 内存方式和密封文件方式。

**Enclave 内存方式：**全局日志以数据结构形式存储在 Enclave 内存中。本文采用有限长度的双端队列数据结构，删除最久远的日志记录避免占用过大的 Enclave 内存，避免程序崩溃。由于 EPC 物理内存大小有限（如 64M/128M/256M），使得每个 Enclave 实例的内存大小有限，当内存使用超出限制时会导致程序崩溃。

**密封文件方式：**将日志以密封文件方式存储到外存。这能减少对 Enclave 内存的消耗，但由于密封文件能被攻击者恶意删除，使得全局日志的可用性无法得到保障。密封文件方式开销巨大（涉及加解密和 OCALL 等）使其在应用中导致程序执行时间过长。

SGX-SEF 设计实现了这两种日志存储方式，默认使用 Enclave 内存方式记录全局日志，不建议使用开销巨大的密封文件方式。

### 4.1.4 安全事件检查

设计安全事件检查功能，分析审计检查点获取的安全事件信息根据安全策略检测攻击。审计检查点内部架构如图 4.3 所示。Enclave 内从不可信环境（依靠

安全建立等技术)可信读取策略文件及符号表。当代码执行到审计检查点时,审计检查点会记录安全事件信息到全局日志中(即审计功能),根据安全策略进行安全事件检查操作(能够查询全局日志实现联动检查),若符合策略,代码继续执行,若不符合策略,审计检查点发出警告或终止代码运行等。

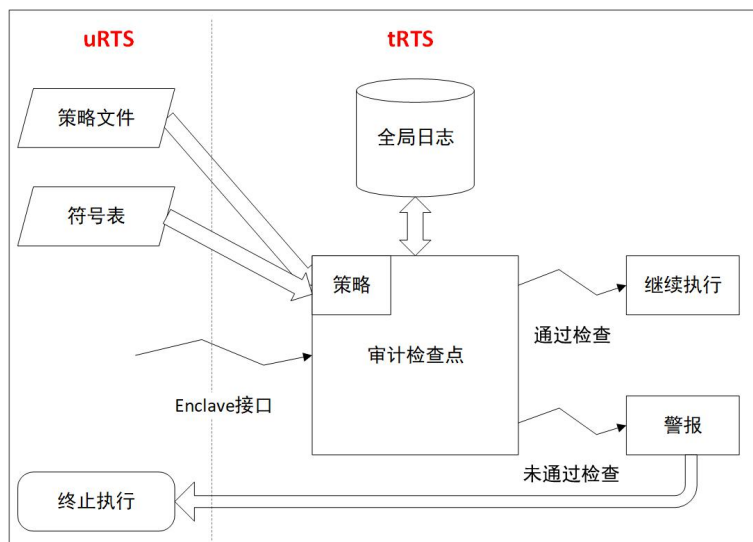


图 4.3 审计检查点内部架构

Figure4.3 Internal architecture of audit-checkpoints

审计检查点主要包括事件获取、日志记录、安全策略、符号表和策略匹配五个关键部分。

**事件获取。**当代码执行到审计检查点时,将安全事件环境信息传给审计检查点。安全事件信息包括事件类型(ECALL/OCALL/AEX/其他)、函数符号名、函数索引值(索引值唯一对应其地址)和参数信息等。

**日志记录。**本文维护了全局日志以记录所有安全事件,检查功能中可以查询全局日志实现联动检查。针对全局日志,设计了 Enclave 内存方式及密封文件方式,默认采用 Enclave 内存方式,不推荐开销巨大的密封文件方式。

**安全策略。**防御者基于 SGX-SEF 制定安全策略,达到所要求的安全准则。tRTS 端的审计检查点需要可信读取存储在不可信硬盘上的安全策略文件。本文修改 SGX Edger8r 以读取安全策略文件,将安全策略重新组织成 tRTS 端 Stub 内的数据结构,使安全策略与 Stub 一起编译到 Enclave 镜像文件中(Enclave 镜像文件受签名保护),利用 Enclave 实例安全建立机制检测安全策略是否被篡改。

**符号表。**安全策略以字符形式存储于策略文件中,但实际检查时只能使用函



数和参数的地址信息（非 Debug 模式下去除了大部分符号名），审计检查点无法直接比对字符名与地址。对此修改 SGX Edger8r，在 SGX Edger8r 读取 EDL 文件时，在 Stub 文件中额外生成多个符号表（包含了地址与符号名的对应关系），帮助审计检查点完成检查操作。

策略匹配。本文针对调用排序、并发调用、恶意线程调度和时间侧信道攻击向量提供了安全策略文件及策略匹配算法，以检测当前程序运行是否符合安全策略，如果符合安全策略，程序继续执行，否则审计检查点返回错误码并终止程序执行。

## 4.2 对 SGX 原生软件栈的修改

SGX-SEF 在 SGX 软件栈关键执行路径中插桩若干审计检查点，审计检查安全事件，加固安全关键点。SGX-SEF 对 SGX 原生软件栈所做修改如图 4.4 所示（图中斜纹形状）：

1. 修改 SGX Edger8r 将安全策略、符号表和审计检查点入口等固化到 tRTS 端 Stub，使新增内容受到 Enclave 实例安全建立机制的保护。
2. 在 tRTS 中增加了全局日志记录及策略检查等功能。

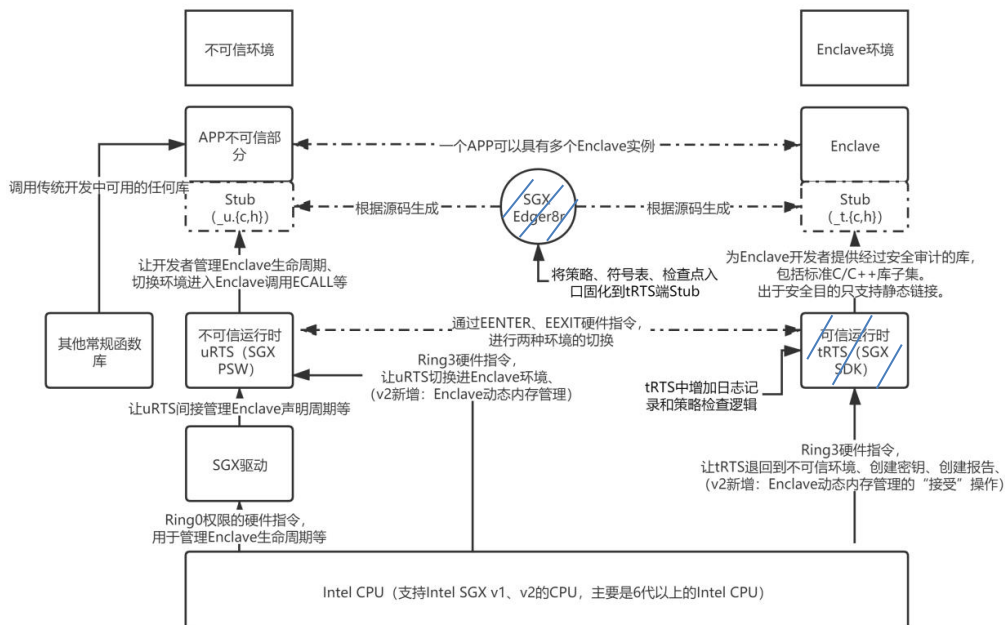


图 4.4 SGX-SEF 对 SGX 软件栈的修改

Figure 4.4 Modification of SGX software stack by SGX-SEF

### 4.3 本章小结

为了审计检查 SGX 关键执行路径中的安全事件，本章对比分析了多种审计检查点插桩方式，选择 SGX 软件栈插桩方式。分析了审计检查代码冲突点位及冲突原因，针对各个安全关键点选择审计检查点位。设计了审计功能，阐述了用于记录全局日志的 Enclave 内存方式和密封文件方式。设计了检查功能，阐述了检查功能中五个关键部分。形成了针对 Intel SGX SDK 的安全增强框架（SGX-SEF），指出了 SGX-SEF 对 SGX 原生软件栈所做的修改，对 SGX 原生 Enclave 实例安全建立机制的利用。

## 第五章 针对 Intel SGX SDK 的安全增强框架验证

本章描述实验过程中的实验环境。展示 SGX-SEF 的审计效果，评估 SGX-SEF 审计功能的性能开销，分析引起性能开销的原因。针对调用排序、并发调用、恶意线程调度和时间侧信道攻击向量编写安全策略验证 SGX-SEF 防御（检查功能）效果。

### 5.1 实验环境

本文的实验环境为笔记本，CPU 型号为 i5-9300H，内存大小为 16GB，操作系统为 Ubuntu20.04，Linux 内核版本为 5.4，linux-sgx（Intel SGX PSW 及 SDK）版本为 2.11，linux-sgx-driver（Intel SGX 驱动）版本为 2.11，CPU 及 BIOS 均支持 SGX。

SkyLake 及更新架构的 CPU 均支持 Intel SGX 特性，但部分架构（如 TigerLake）不支持 Intel SGX，具体内容可以前往 Intel 官网查询。除需要 CPU 支持 SGX 特性外，还需要确保 BIOS 中开启了 SGX。有些旧版 BIOS 不支持 SGX，但部分电脑厂商在后续 BIOS 更新中增加 SGX 开启选项，对此可以升级 BIOS 以获得 SGX 开启选项。

建议使用真机（Bare Metal）形式安装 Ubuntu 操作系统，VMWare Workstation 等大部分虚拟机管理程序中安装的 Ubuntu 虚拟机不支持 SGX 特性（即使 CPU 及 BIOS 均支持 SGX，但 VMWare Workstation 自身不能向虚拟机提供 Intel SGX 特性）。KVM-SGX 及 QEMU-SGX 能够帮助 SGX 用户在 QEMU 虚拟机中开启 SGX 特性。

可以通过 `sgx-software-enable` Github 仓库或参照 SGX 手册使用 `CPUID` 硬件指令检测当前平台是否成功开启了 SGX。

### 5.2 审计功能效果验证及性能分析

本节展示审计功能效果，评估审计功能开销，分析引起开销的原因。

### 5.2.1 审计功能效果验证

审计功能的实现效果如图 5.1 所示。对 SampleEnclave 中的安全事件进行审计，安全事件信息包含接口类型、接口名称、接口索引和参数（实现中可以捕获参数，但由于参数数量各异，先用 0 值填充）等。图中的演示效果是通过调用 ECALL 打印（printf）位于 Enclave 内的全局日志。由于 printf 需要通过 OCALL 实现（Enclave 内不支持），因此要求审计检查点在审计时过滤 printf 所用的 OCALL，避免无限次嵌套导致程序崩溃。

```
leone@leone-Inspiron-7590:~/文档/linux-sgx/SampleCode/SampleEnclave$ ./app
Checksum(0x0x7ffd2514a770, 100) = 0xffffd4143
Info: executing thread synchronization, please wait...
=====LOG MEM MODE=====
[CP_INFO] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_array_user_check(19), FuncParam: 0x0
[CP_INFO] InterfaceType: INTERFACE_ECALL_RET(3), FuncName: ecall_array_user_check(19), FuncParam: 0x0
[CP_INFO] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_array_in(20), FuncParam: 0x0
[CP_INFO] InterfaceType: INTERFACE_ECALL_RET(3), FuncName: ecall_array_in(20), FuncParam: 0x0
[CP_INFO] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_array_out(21), FuncParam: 0x0
[CP_INFO] InterfaceType: INTERFACE_ECALL_RET(3), FuncName: ecall_array_out(21), FuncParam: 0x0
[CP_INFO] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_array_in_out(22), FuncParam: 0x0
[CP_INFO] InterfaceType: INTERFACE_ECALL_RET(3), FuncName: ecall_array_in_out(22), FuncParam: 0x0
[CP_INFO] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_array_isary(23), FuncParam: 0x0
[CP_INFO] InterfaceType: INTERFACE_ECALL_RET(3), FuncName: ecall_array_isary(23), FuncParam: 0x0
[CP_INFO] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_pointer_user_check(9), FuncParam: 0x0
[CP_INFO] InterfaceType: INTERFACE_OCALL(6), FuncName: ocall_print_string(0), FuncParam: 0x0
[CP_INFO] InterfaceType: INTERFACE_OCALL_RET(7), FuncName: ocall_print_string(0), FuncParam: 0x0
[CP_INFO] InterfaceType: INTERFACE_ECALL_RET(3), FuncName: ecall_pointer_user_check(9), FuncParam: 0x0
[CP_INFO] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_pointer_in(10), FuncParam: 0x0
[CP_INFO] InterfaceType: INTERFACE_ECALL_RET(3), FuncName: ecall_pointer_in(10), FuncParam: 0x0
```

图 5.1 审计功能的效果

Figure5.1 Effect of Audit function

### 5.2.2 审计功能性能开销分析

SGX-SEF 审计功能实现中，采用密封文件方式记录全局日志会导致性能开销过大，原因在于加解密和 OCALL 等会造成巨大开销，建议采用 Enclave 内存方式记录全局日志。本小节将对采用 Enclave 内存方式记录全局日志的审计功能进行性能分析。基于 linux-sgx Github 仓库的 SampleEnclave 和 Switchless 示例代码，评估审计功能（在 Switchless/Ordinary ECALL/OCALL 上）的性能开销，并分析其原因。本文所测时间值以秒为单位，小数点精确到微秒，所有时间值均已被测量三次取平均。

基于 Switchless 示例代码评估 SGX-SEF 审计功能性能开销。Switchless 代码中重复 50000 次 Switchless 和 Ordinary 调用，所调用函数中没有任何代码（即空负载），因此该评估工作旨在测量 Switchless 和 Ordinary 调用的固有开销。由于

EPC 物理内存大小有限，因此审计功能采用有限长度的（删除最久之前的记录）双端队列以内存形式记录全局日志。双端队列大小为 10000、1000、100、10 项以及不使用 SGX-SEF 时的时间开销如下表 5.1 所示。

**表 5.1** 基于 Switchless 的审计功能性能开销测量表

**Table 5.1** Table of audit performance overhead measurement based on Switchless

日志记录项数	10000 项	1000 项	100 项	10 项	No SGX-SEF
Switchless OCALL 时长（秒）	3.149677	0.346707	0.081700	0.057652	0.016433
Ordinary OCALL 时长（秒）	3.494144	0.527837	0.263078	0.238200	0.174782
Switchless ECALL 时长（秒）	3.254656	0.363362	0.096939	0.067975	0.021817
Ordinary ECALL 时长（秒）	3.587716	0.722888	0.452293	0.429314	0.382114

当日志记录项数增大时，审计功能对各种调用方式产生的开销增大，原因在于日志使用维护开销增大，缓存命中率降低进一步导致开销增大。当日志项数过大时，占用内存超出 Enclave 实例可用 EPC 大小，Enclave 实例崩溃。使用 SGX-SEF 审计功能相较于不使用 SGX-SEF 时，Switchless 调用方式开销增幅明显，Ordinary 调用方式开销增幅较小，原因在于 Switchless 调用方式中多个 Enclave 工人线程并发记录日志所用的同步机制及 OCALL 大幅增加性能开销，记录日志代码流程待优化。依据实验中接触的 Enclave 应用场景及反复实验得出的经验，日志大小为 100 项的审计功能对各种调用方式引起的开销增幅不明显，但又足够记录最近的安全事件用于审计检查功能。日志大小 100 项的 SGX-SEF 审计功能最优情况（Ordinary 调用方式）下开销增幅仅为 18%。

基于 SampleEnclave 示例代码评估 SGX-SEF 审计功能性能开销。SampleEnclave 代码流程以 Ordinary 调用为主，调用函数并非空负载，该评估工作旨在评估审计功能在 SGX 实际应用中产生的性能开销。测量过程使用日志大小 100 项 Enclave 内存形式日志的 SGX-SEF 审计功能，结果如表 5.2 所示，使用 SGX-SEF 审计功能时，SampleEnclave 总时间开销为 0.072994 秒，而不使用 SGX-SEF 时的时间开销为 0.064461 秒，开销增幅为 13%。

**表 5.2** 基于 SampleEnclave 的审计功能性能开销测量表

**Table5.2** Table of audit performance overhead measurement based on SampleEnclave

日志记录项数	100 项	No SGX-SEF
SampleEnclave 执行时长（秒）	0.072994	0.064461

综上所述，SGX-SEF 审计功能最优情况下产生 13%~18%的开销增幅。

### 5.3 防御（检查功能）效果验证

本节针对调用排序、并发调用、恶意线程调度和时间侧信道攻击向量编写安全策略验证防御（检查功能）效果。

#### 5.3.1 针对“调用排序”攻击向量的防御效果验证

实验过程中构造实现了“调用排序”攻击向量的 POC。攻击之前，ECALL 调用顺序先后为 `ecall_malloc_obj1` 函数、`ecall_malloc_obj2_use_obj1` 函数和 `ecall_free_obj1` 函数，`ecall_malloc_obj1` 函数在 Enclave 堆上创建值为“hello”的对象 1（即所指内容为“hello”的对象 1 指针，下文不再对此特别说明），`ecall_malloc_obj2_use_obj1` 函数在 Enclave 堆上创建值为“uaf”的对象 2 并使用对象 1 的值“hello”，`ecall_free_obj1` 函数释放对象 1 但未将对象 1 置 NULL（对象 1 成为悬空指针，悬空指针即所指对象已被释放但自身未被置 NULL 的指针）。程序结果如图 5.2 所示



```

leone@leone-Inspiron-7590: ~/文档/Call Permutation
leone@leone-Inspiron-7590:~/文档/Call Permutation$ ./app
Malloc obj1(0x7faba203d0b0)=hello
Malloc obj2(0x7faba203d0d0)=uaf
Use obj1(0x7faba203d0b0)=hello
Free obj1
leone@leone-Inspiron-7590:~/文档/Call Permutation$ 

```

**图 5.2** 调用排序攻击之前程序执行结果图

**Figure5.2** Diagram of program execution result before Calling Permutation Attack

恶意将 `ecall_malloc_obj1` 函数、`ecall_malloc_obj2_use_obj1` 函数和 `ecall_free_obj1` 函数的先后顺序修改为 `ecall_malloc_obj1` 函数、`ecall_free_obj1` 函数和 `ecall_malloc_obj2_use_obj1` 函数，使得 `ecall_free_obj1` 中的悬空指针（即对象 1 指针）被 `ecall_malloc_obj2_use_obj1` 使用，触发 UAF。程序执行结果如图

5.3 所示，`ecall_malloc_obj2_use_obj1` 函数中，对象 2 指针被分配了在与悬空指针（对象 1 指针）相同的地址，使用悬空指针（对象 1 指针）所指内容等同于使用对象 2 的值“uaf”。



```

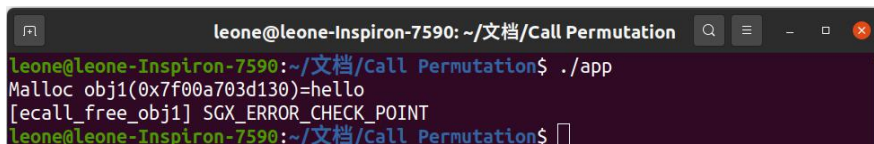
leone@leone-Inspiron-7590: ~/文档/Call Permutation
leone@leone-Inspiron-7590:~/文档/Call Permutation$ ./app
Malloc obj1(0x7f9b5203d0b0)=hello
Free obj1
Malloc obj2(0x7f9b5203d0b0)=uaf
Use obj1(0x7f9b5203d0b0)=uaf
leone@leone-Inspiron-7590:~/文档/Call Permutation$

```

图 5.3 调用排序攻击之后程序执行结果图

Figure5.3 Diagram of program execution result after Calling Permutation Attack

为了对抗“调用排序”攻击，Enclave 开发者能够依据先验的预期执行情况编写策略文件，在 SGX-SEF 中设置“ECALL 调用顺序白名单”安全策略，要求检测到的 ECALL 调用顺序先后分别为 `ecall_malloc_obj1` 函数、`ecall_malloc_obj2_use_obj1` 函数和 `ecall_free_obj1` 函数，否则 SGX-SEF 将向 Enclave 代码返回错误码 `SGX_ERROR_CHECK_POINT` 并终止代码运行。如图 5.4 所示，在 SGX-SEF 中设置“ECALL 调用顺序白名单”策略能成功检测非预期的调用顺序。



```

leone@leone-Inspiron-7590: ~/文档/Call Permutation
leone@leone-Inspiron-7590:~/文档/Call Permutation$ ./app
Malloc obj1(0x7f00a703d130)=hello
[ecall_free_obj1] SGX_ERROR_CHECK_POINT
leone@leone-Inspiron-7590:~/文档/Call Permutation$

```

图 5.4 调用排序攻击检测结果

Figure5.4 Detection results of Call Permutation Attack

Github 上开源的 `SGX_SQLite` 中存在数据库句柄释放后未置 NULL 成为悬空指针的情况，能够被“调用排序”攻击。“调用排序”攻击前 `SGX_SQLite` 程序执行情况如图 5.5 所示，即打开数据库句柄（`ecall_opendb`），使用数据库句柄执行 SQL 语句（`ecall_execute_sql`），关闭数据库句柄（`ecall_closedb`）。



```

leone@leone-Inspiron-7590: ~/文档/SGX-Attack-POC
leone@leone-Inspiron-7590:~/文档/SGX-Attack-POC$ ./app
Info: SQLite SGX enclave successfully created.
Enclave: Created database connection to a.db
Enter SQL statement to execute or 'quit' to exit:
> select *
SQLite error: no tables specified
> quit
Enclave: Closed database connection
=====LOG MEM MODE=====
[SGX_SQLite Call Permutation] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_opendb(0), FuncParam: 0x0
[SGX_SQLite Call Permutation] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_execute_sql(1), FuncParam: 0x0
[SGX_SQLite Call Permutation] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_closedb(2), FuncParam: 0x0
[SGX_SQLite Call Permutation] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_show_log(3), FuncParam: 0x0
=====LOG MEM MODE END=====
Info: SQLite SGX enclave successfully returned.
leone@leone-Inspiron-7590:~/文档/SGX-Attack-POC$

```

图 5.5 调用排序攻击前 SGX\_SQLite 执行情况图

Figure5.5 Diagram of SGX\_SQLite execution before Call Permutation attack

恶意篡改 SGX\_SQLite 中 ECALL 函数的调用顺序，使得数据库句柄关闭（ecall\_closedb）后未置 NULL 形成的悬空指针被继续使用（ecall\_execute\_sql），形成安全隐患。程序执行情况如图 5.6 所示。

```

leone@leone-Inspiron-7590: ~/文档/SGX-Attack-POC
leone@leone-Inspiron-7590:~/文档/SGX-Attack-POC$ ./app
Info: SQLite SGX enclave successfully created.
Enclave: Created database connection to a.db
Enclave: Closed database connection
Enter SQL statement to execute or 'quit' to exit:
> select *
> quit
=====LOG MEM MODE=====
[SGX_SQLite Call Permutation] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_opendb(0), FuncParam: 0x0
[SGX_SQLite Call Permutation] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_closedb(2), FuncParam: 0x0
[SGX_SQLite Call Permutation] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_execute_sql(1), FuncParam: 0x0
[SGX_SQLite Call Permutation] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_show_log(3), FuncParam: 0x0
=====LOG MEM MODE END=====
Info: SQLite SGX enclave successfully returned.
leone@leone-Inspiron-7590:~/文档/SGX-Attack-POC$

```

图 5.6 调用排序攻击后 SGX\_SQLite 执行情况图

Figure5.6 Diagram of SGX\_SQLite execution after Call Permutation attack

为了缓解“调用排序”攻击，Enclave 开发者能够依据先验的预期执行情况编写策略文件，在 SGX-SEF 上设置“ECALL 调用顺序白名单”策略，要求检测到的 ECALL 调用顺序先后分别为 ecall\_open\_db 函数、ecall\_execute\_sql 函数和 ecall\_close\_db 函数。如果执行情况违反策略，SGX-SEF 将返回错误码 SGX\_ERROR\_CHECK\_POINT（0x9001）并终止程序运行。如图 5.7 所示，在 SGX-SEF 中设置“ECALL 调用顺序白名单”策略能成功检测非预期的调用顺序。





```

leone@leone-Inspiron-7590: ~/文档/SGX-Attack-POC
leone@leone-Inspiron-7590:~/文档/SGX-Attack-POC$ ./app
Info: SQLite SGX enclave successfully created.
Enclave: Created database connection to a.db
[ecall_closedb] Error: 0x9001
leone@leone-Inspiron-7590:~/文档/SGX-Attack-POC$

```

图 5.7 SGX\_SQLite 中调用排序攻击检测结果

Figure5.7 Detection results of Call Permutation Attack in SGX\_SQLite

除 Glamdring 外尚无防御方案针对 SGX 调用排序攻击向量进行有效防御。Glamdring 减少 ECALL 数量以避免调用排序攻击，该方法效果有限，因为有些 ECALL 函数无法被裁剪。

### 5.3.2 针对“并发调用”攻击向量的防御效果验证

实验构造实现了“并发调用”攻击向量 POC。并发调用攻击之前，ECALL 调用顺序先后分别为 `ecall_malloc_obj1` 函数、`ecall_malloc_obj2_use_obj1` 函数和 `ecall_free_obj1` 函数。程序执行结果如图 5.8 所示，主线程调用 `ecall_malloc_obj1` 函数在 Enclave 堆上创建对象 1 并等待线程 1 使用对象 1，线程 1 在 Enclave 堆上创建对象 2 并使用对象 1 的值“hello”，线程 1 结束后主线程释放对象 1 但未将对象 1 置 NULL（对象 1 指针成为悬空指针）。



```

leone@leone-Inspiron-7590: ~/文档/Concurrent Calls
leone@leone-Inspiron-7590:~/文档/Concurrent Calls$ ./app
Malloc obj1(0x7fb19f03d0b0)=hello
===Thread1 Running===
Malloc obj2(0x7fb19f03d0d0)=uaf
Use obj1(0x7fb19f03d0b0)=hello
===Thread1 Exit===
Free obj1
leone@leone-Inspiron-7590:~/文档/Concurrent Calls$

```

图 5.8 并发调用攻击之前程序执行结果

Figure5.8 Program execution result before Concurrent Calls Attack

恶意暂停线程 1 执行 `ecall_malloc_obj2_use_obj1` 函数，将 `ecall_malloc_obj1` 函数、`ecall_malloc_obj2_use_obj1` 函数和 `ecall_free_obj1` 函数的 ECALL 调用顺序篡改改为 `ecall_malloc_obj1` 函数、`ecall_free_obj1` 函数和 `ecall_malloc_obj2_use_obj1` 函数的调用顺序（如图 5.9 所示），使得 `ecall_free_obj1` 中的悬空指针（对象 1 指针）被 `ecall_malloc_obj2_use_obj1` 使用，触发 UAF。

程序执行结果如图 5.10 所示，`ecall_malloc_obj2_use_obj1` 函数中，对象 2 被分配在与悬空指针（对象 1 指针）相同的地址，使用悬空指针（对象 1 指针）所指内容等同于使用对象 2 的值“uaf”。

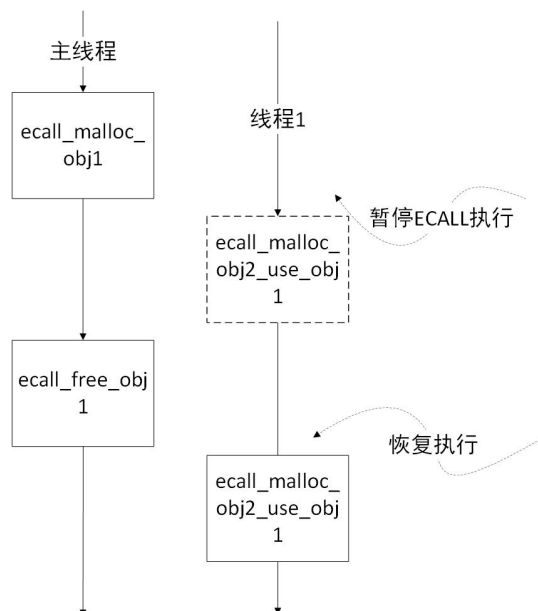


图 5.9 恶意篡改并发调用

Figure5.9 Maliciously tampers with the concurrent calls

```

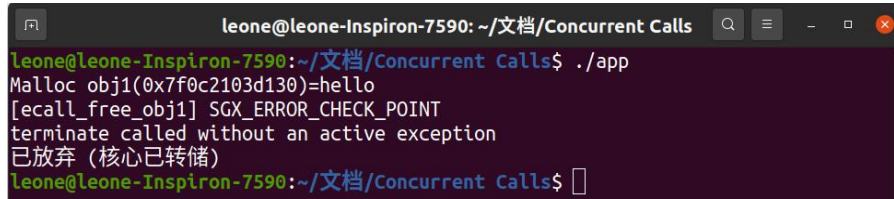
leone@leone-Inspiron-7590: ~/文档/Concurrent Calls
leone@leone-Inspiron-7590:~/文档/Concurrent Calls$ ./app
Malloc obj1(0x7fbe2603d0b0)=hello
Free obj1
===Thread1 Running===
Malloc obj2(0x7fbe2603d0b0)=uaf
Use obj1(0x7fbe2603d0b0)=uaf
===Thread1 Exit===
leone@leone-Inspiron-7590:~/文档/Concurrent Calls$
  
```

图 5.10 并发调用攻击之后程序执行结果

Figure5.10 Program execution result after Concurrent Calls Attack

对此，Enclave 开发者依据先验的预期执行情况编写策略文件，基于 SGX-SEF 设置“ECALL 调用顺序白名单”安全策略，要求检测到的 ECALL 调用顺序满足 `ecall_malloc_obj1` 函数、`ecall_malloc_obj2_use_obj1` 函数和 `ecall_free_obj1` 函数的先后顺序，否则 SGX-SEF 返回错误码 `SGX_ERROR_CHECK_POINT` 并终止代码运行。如图 5.11 所示，在 SGX-SEF 中设置“ECALL 调用顺序白名单”策略能成功检测非预期的调用顺序，核心已转储的原因是主线程调用

ecall\_free\_obj1 不符合安全策略被终止，线程 1 随主线程的终止而终止并发出核心已转储等信息。



```

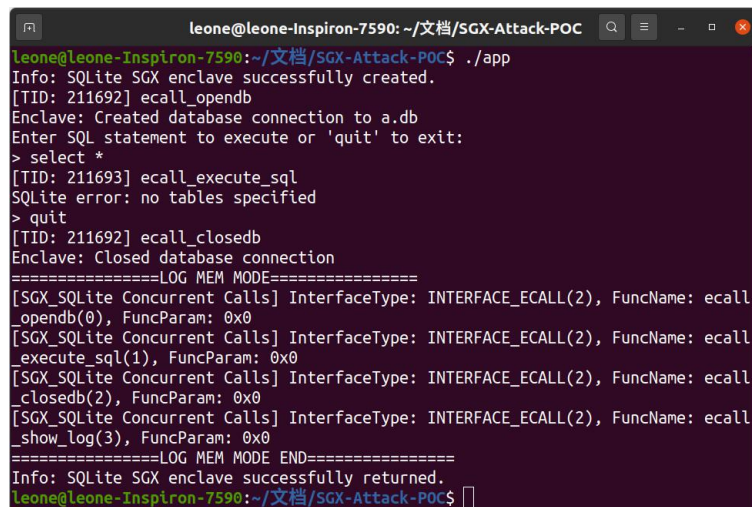
leone@leone-Inspiron-7590: ~/文档/Concurrent Calls
leone@leone-Inspiron-7590:~/文档/Concurrent Calls$ ./app
Malloc obj1(0x7f0c2103d130)=hello
[ecall_free_obj1] SGX_ERROR_CHECK_POINT
terminate called without an active exception
已放弃 (核心已转储)
leone@leone-Inspiron-7590:~/文档/Concurrent Calls$ 

```

图 5.11 并发调用攻击检测结果

Figure5.11 Detection result of Concurrent Calls Attack

SGX\_SQLite 中的悬空指针的情况，也能被“并发调用”攻击。攻击前 SGX\_SQLite 程序执行情况如图 5.12 所示，即主线程打开数据库句柄（ecall\_opendb），线程 1 使用数据库句柄执行 SQL 语句（ecall\_execute\_sql），主线程关闭数据库句柄（ecall\_closedb）。



```

leone@leone-Inspiron-7590: ~/文档/SGX-Attack-POC
leone@leone-Inspiron-7590:~/文档/SGX-Attack-POC$ ./app
Info: SQLite SGX enclave successfully created.
[TID: 211692] ecall_opendb
Enclave: Created database connection to a.db
Enter SQL statement to execute or 'quit' to exit:
> select *
[TID: 211693] ecall_execute_sql
SQLite error: no tables specified
> quit
[TID: 211692] ecall_closedb
Enclave: Closed database connection
=====LOG MEM MODE=====
[SGX_SQLite Concurrent Calls] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_opendb(0), FuncParam: 0x0
[SGX_SQLite Concurrent Calls] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_execute_sql(1), FuncParam: 0x0
[SGX_SQLite Concurrent Calls] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_closedb(2), FuncParam: 0x0
[SGX_SQLite Concurrent Calls] InterfaceType: INTERFACE_ECALL(2), FuncName: ecall_show_log(3), FuncParam: 0x0
=====LOG MEM MODE END=====
Info: SQLite SGX enclave successfully returned.
leone@leone-Inspiron-7590:~/文档/SGX-Attack-POC$ 

```

图 5.12 并发调用攻击前 SGX\_SQLite 执行情况图

Figure5.12 Diagram of SGX\_SQLite execution before Concurrent Calls attack

恶意拖延线程 1 执行 SQL 语句的时机，使得主线程中数据库句柄关闭（ecall\_closedb）后未置 NULL 形成的悬空指针被继续使用（ecall\_execute\_sql），形成安全隐患。程序执行情况如图 5.13 所示。

```

leone@leone-Inspiron-7590: ~/文档/SGX-Attack-POC
leone@leone-Inspiron-7590:~/文档/SGX-Attack-POC$ ./app
Info: SQLite SGX enclave successfully created.
[TID: 212162] ecall_opendb
Enclave: Created database connection to a.db
[TID: 212162] ecall_closedb
Enclave: Closed database connection
Enter SQL statement to execute or 'quit' to exit:
> select *
[TID: 212163] ecall_execute_sql
> quit
=====LOG MEM MODE=====
[SGX_SQLite Concurrent Calls] InterfaceType: INTERFACE_ECALL(2), FuncName: ecal
l_opendb(0), FuncParam: 0x0
[SGX_SQLite Concurrent Calls] InterfaceType: INTERFACE_ECALL(2), FuncName: ecal
l_closedb(2), FuncParam: 0x0
[SGX_SQLite Concurrent Calls] InterfaceType: INTERFACE_ECALL(2), FuncName: ecal
l_execute_sql(1), FuncParam: 0x0
[SGX_SQLite Concurrent Calls] InterfaceType: INTERFACE_ECALL(2), FuncName: ecal
l_show_log(3), FuncParam: 0x0
=====LOG MEM MODE END=====
Info: SQLite SGX enclave successfully returned.
leone@leone-Inspiron-7590:~/文档/SGX-Attack-POC$

```

图 5.13 并发调用攻击后 SGX\_SQLite 执行情况图

Figure5.13 Diagram of SGX\_SQLite execution after Concurrent Calls attack

为了缓解“并发调用”攻击，Enclave 开发者依据先验的预期执行情况编写策略文件，在 SGX-SEF 上设置“ECALL 调用顺序白名单”策略。如果执行情况违反策略，SGX-SEF 将返回错误码 SGX\_ERROR\_CHECK\_POINT (0x9001) 并终止程序运行。如图 5.14 所示，在 SGX-SEF 中设置“ECALL 调用顺序白名单”策略能成功检测非预期的调用顺序，核心已转储的原因是主线程调用 ecall\_close\_db 时不符合安全策略被终止，线程 1 随主线程的终止而终止并发出核心已转储等信息。

```

leone@leone-Inspiron-7590: ~/文档/SGX-Attack-POC
leone@leone-Inspiron-7590:~/文档/SGX-Attack-POC$ ./app
Info: SQLite SGX enclave successfully created.
[TID: 211868] ecall_opendb
Enclave: Created database connection to a.db
[TID: 211868] ecall_closedb
[ecall_closedb] Error: 0x9001
[sgx_destroy_enclave]Error: 0x9001
terminate called without an active exception
已放弃 (核心已转储)
leone@leone-Inspiron-7590:~/文档/SGX-Attack-POC$

```

图 5.14 SGX\_SQLite 中并发调用攻击检测结果

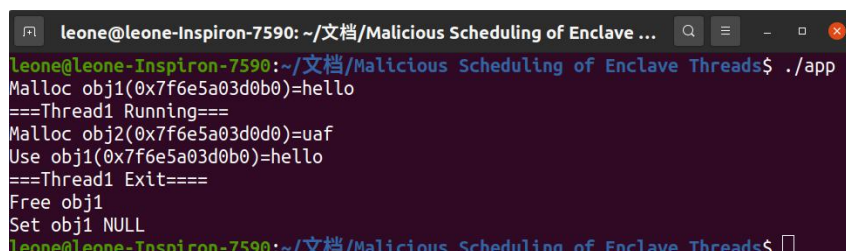
Figure5.14 Detection results of Concurrent Calls Attack in SGX\_SQLite

目前尚未发现其它针对 SGX 并发调用攻击向量的防御方案。

### 5.3.3 针对“恶意 Enclave 线程调度”攻击向量的防御效果验证

利用“线程暂停原语”恶意调度 Enclave 线程，构造实现“恶意 Enclave 线

程调度”攻击向量 POC。攻击之前，ECALL 调用顺序为 `ecall_malloc_obj1` 函数、`ecall_malloc_obj2_use_obj1` 函数和 `ecall_free_obj1` 函数(悬空指针已通过置 NULL 修复)。程序执行结果如图 5.15 所示，主线程调用 `ecall_malloc_obj1` 函数在 Enclave 堆上创建值为“hello”的对象 1 并等待线程 1 执行，线程 1 调用 `ecall_malloc_obj2_use_obj1` 函数在 Enclave 堆上创建值为“uaf”的对象 2 并使用值为“hello”的对象 1，线程 1 结束后主线程将对象 1 释放并置 NULL。



```

leone@leone-Inspiron-7590: ~/文档/Malicious Scheduling of Enclave ...
leone@leone-Inspiron-7590:~/文档/Malicious Scheduling of Enclave Threads$ ./app
Malloc obj1(0x7f6e5a03d0b0)=hello
===Thread1 Running===
Malloc obj2(0x7f6e5a03d0d0)=uaf
Use obj1(0x7f6e5a03d0b0)=hello
===Thread1 Exit===
Free obj1
Set obj1 NULL
leone@leone-Inspiron-7590:~/文档/Malicious Scheduling of Enclave Threads$

```

图 5.15 恶意线程调度攻击前程序执行结果

Figure 5.15 Program execution result before Malicious Scheduling of Threads Attack

通过“线程暂停原语”将线程 1 的执行流停在 `ecall_free_obj1` 中释放指针后以及设置指针 NULL 前，形成 `ecall_malloc_obj1` 函数、`ecall_free_obj1` 函数（释放对象 1 指针，指针成为悬空指针）、`ecall_malloc_obj2_use_obj1` 函数和 `ecall_free_obj1` 函数(对象 1 指针置 NULL)的调用顺序（如图 5.16 所示），使得 `ecall_free_obj1` 的悬空指针（对象 1 指针）置 NULL 之前被 `ecall_malloc_obj2_use_obj1` 使用，触发 UAF。程序执行结果如图 5.17 所示，`ecall_free_obj1` 函数中的释放过程和置 NULL 过程被中断，`ecall_malloc_obj2_use_obj1` 函数中，对象 2 被分配了在与悬空指针(对象 1 指针)相同的地址，使用悬空指针(对象 1 指针)所指内容等同于使用对象 2 的值“uaf”。



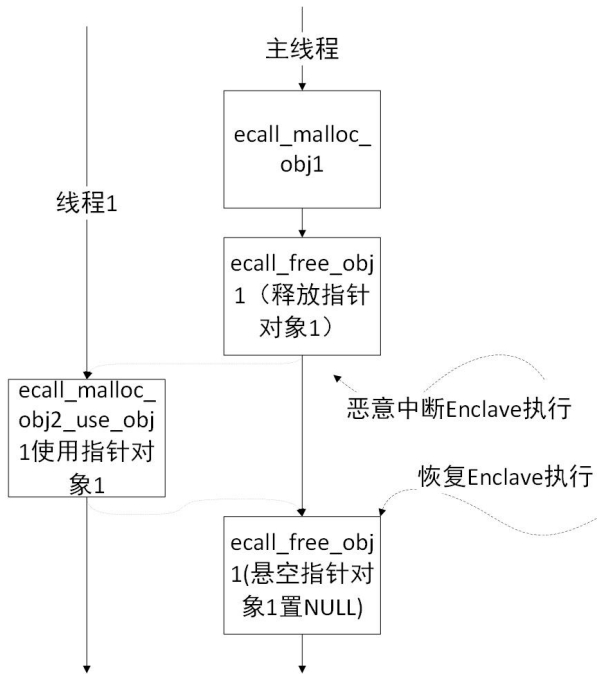


图 5.16 恶意中断 ecall\_free\_obj1 函数

Figure5.16 Malicious interruption of ecall\_free\_obj1 function

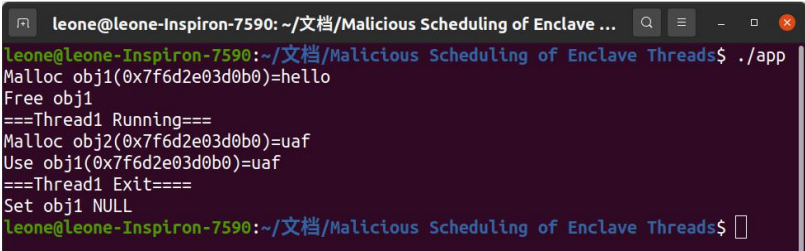


图 5.17 恶意线程调度攻击后程序执行结果

Figure5.17 Program execution results after Malicious Scheduling of Threads Attack

实验中使用的“Enclave 线程暂停原语”包括：

1. 使用 Kprobe 的页错误型 Enclave 线程暂停原语。如图 5.18 所示，攻击者清理目标页存在（Present）位，利用 Kprobe 拦截所有页错误并从中过滤出目标页错误，最后暂停目标错误页所对应的线程。

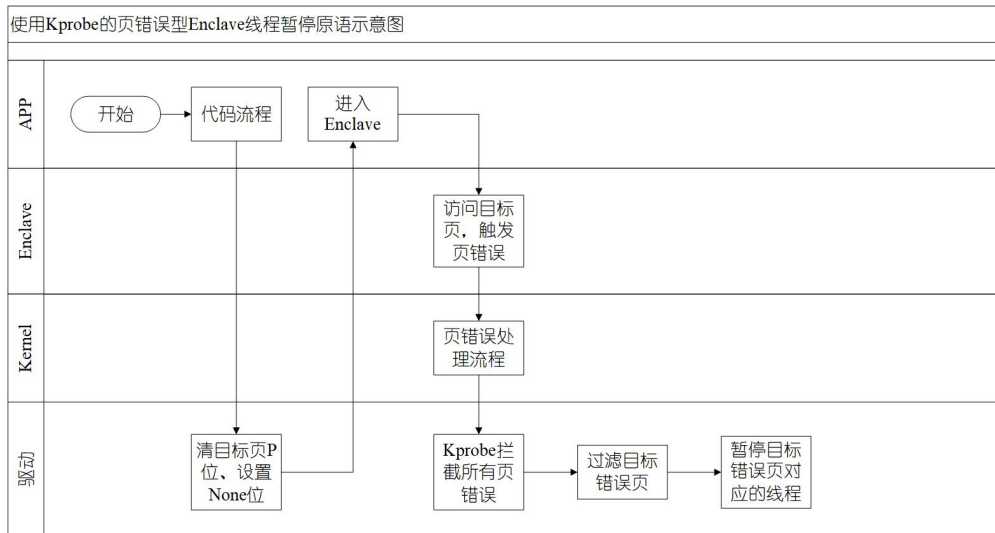


图 5.18 使用 Kprobe 的页错误型 Enclave 线程暂停原语示意图

Figure5.18 Diagram of Enclave thread suspension primitive using page fault and Kprobe

2. 使用 SIGSEGV 的页错误型 Enclave 线程暂停原语。如图 5.19 所示，攻击者预先清理目标页 Present 位，受害者 Enclave 线程访问此页时会触发页错误并进入攻击者事先注册的 SIGSEGV 处理句柄，受害者 Enclave 在 SIGSEGV 处理句柄中被暂停执行。由于 SGX 软件栈中的 uRTS 会直接报告 Enclave 处于崩溃状态，因此实现该线程暂停原语时需要修改 uRTS 端代码。

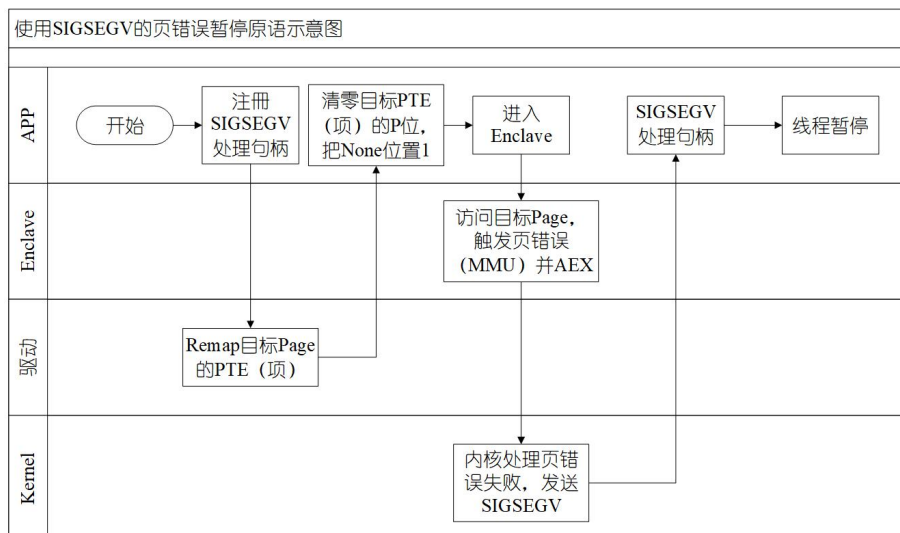


图 5.19 使用 SIGSEGV 的页错误型 Enclave 线程暂停原语示意图

Figure5.19 Diagram of Enclave thread suspend primitive using page fault and SIGSEGV

对此，Enclave 开发者编写策略文件，基于 SGX-SEF 设置“检测到 AEX 就报警”安全策略，要求程序执行过程中不能出现“Enclave 线程暂停原语”会用到的 AEX，否则 SGX-SEF 返回错误码 SGX\_ERROR\_CHECK\_POINT 并终止代码运行。检测结果如图 5.20 所示。

```

leone@leone-Inspiron-7590: ~/文档/Malicious Scheduling of Enclave ...
leone@leone-Inspiron-7590:~/文档/Malicious Scheduling of Enclave Threads$ ./app
Malloc obj1(0x7ffa6203d0b0)=hello
Free obj1
AEX!
terminate called without an active exception
已放弃（核心已转储）
leone@leone-Inspiron-7590:~/文档/Malicious Scheduling of Enclave Threads$
    
```

图 5.20 恶意线程调度攻击检测结果

Figure5.20 Detection results of Malicious Scheduling of Threads Attack

但“检测到 AEX 就报警”策略过于简单，容易导致误报，需要进一步研究更合理的安全策略及策略匹配算法。目前尚未发现其它针对 SGX 恶意线程调度攻击向量的防御方案。

### 5.3.4 针对“时间侧信道”的防御效果验证

本文分析并复现了 SGX-Step<sup>[43]</sup>/Nemesis 攻击（使用时间侧信道攻击向量），其源码流程如图 5.21 所示。攻击者会频繁使用 APIC 时钟中断（时钟中断间隔为五十多个 CPU 周期，接近一条二进制代码执行时长），以中断 Enclave 代码执行，在 AEX 处理过程中计算一条二进制代码执行时延。Nemesis 进一步利用该时延构建时延侧信道。

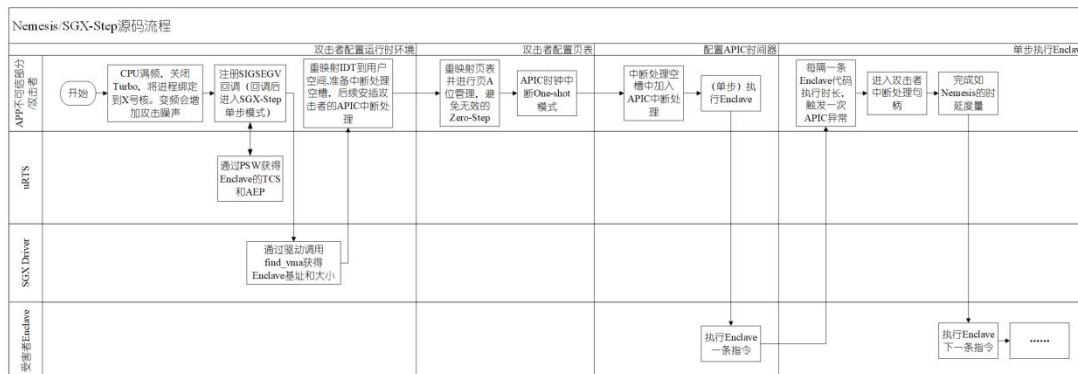


图 5.21 SGX-Step/Nemesis 源码分析流程图

Figure5.21 Flowchart of SGX-Step/Nemesis



基于 SGX-SEF 设置“AEX 超过 10 次即为攻击”安全策略。实验中, SGX-SEF 能够成功检测该攻击, 返回错误码并终止程序运行。由于所使用的攻击 POC 代码量较少, 因此使用较小的阈值, 实际应用中可以参考 Varys<sup>[44]</sup>设置约 5000 次/秒的阈值来检测攻击。

#### 5.4 本章小结

本章描述了实验环境。展示了审计功能的效果, 评估了审计功能性能开销增幅在最优情况下为 13%~18%, 分析了引起性能开销的原因。针对调用排序、并发调用、恶意线程调度和时间侧信道攻击向量制定了安全策略验证 SGX-SEF 防御(检查功能)效果, 指出 SGX-SEF 具有良好的防御效果, 能灵活地兼容多种安全准则。

SGX 关键执行路径及安全关键点分析启发“调用顺序白名单”安全策略以抵御调用排序和并发调用攻击向量, 此前尚无防御方案能够有效防御这两类攻击向量。



## 第六章 总结与展望

### 6.1 总结

现有 SGX 攻击针对 Enclave 开发中的安全问题和 SGX 设计存在安全缺陷，透过 Enclave 接口及共享资源等攻击面，构建各类攻击向量，威胁 SGX 安全性。现有防御方案安全准则不统一，尚无防御方案分析 SGX 安全关键点并从 SGX 架构自身进行安全增强。本文提出了针对 Intel SGX SDK 的安全增强框架（简称 SGX-SEF），旨在从 SGX 架构自身出发，全面加固 SGX 架构中的安全关键点，主要成果包括：

1. 产出 SGX 关键执行路径及安全关键点分析结果。本文以 SGX 软件栈为主刻画了 SGX 架构及关键执行路径，总结了其中的安全关键点，指导了对已有防御方案安全准则的兼容，启发了新的安全准则。
2. 设计实现了针对 Intel SGX SDK 的安全增强框架（SGX-SEF）。SGX-SEF 审计了 SGX 关键执行路径，记录了上下文切换等安全事件，提供了函数名及参数等信息用于检测恶意行为，加固了安全关键点，将安全框架与安全策略分离，依据多种安全策略防御多种攻击向量，解决现有 SGX 架构难以有效兼容安全准则不统一的防御方案的问题。
3. 提出了“调用顺序白名单”安全策略。针对调用排序和并发调用攻击向量会恶意篡改调用逻辑违背开发者预期的情况，Enclave 开发者能够基于 SGX-SEF 在 SGX 关键执行路径的安全关键点中插桩审计检查代码并部署“调用顺序白名单”安全策略以要求调用情况符合预期，从而抵御调用排序和并发调用攻击向量。针对“调用排序”和“并发调用”攻击向量的 SGX-SEF 防御效果验证中，依据预期执行情况设计的 ECALL 调用顺序白名单策略能很好地检测非预期的 ECALL 调用。调用顺序白名单策略还能进一步包含对 OCALL 调用的限定，使程序执行情况充分符合预期。
4. 针对 Intel SGX SDK 的安全增强框架（SGX-SEF）效果验证及性能评估。通过实验验证了 SGX-SEF 能够有效审计 SGX 架构的关键执行路径。评

估了审计功能性能开销，分析了引起性能开销的原因。审计功能开销随日志容量增大而增大，开销包括日志使用维护开销及缓存未命中开销。当日志大小 100 项时，审计功能对各种调用方式引起的开销增幅不明显，100 项大小的日志足够记录最近的安全事件用于审计检查功能。日志容量为 100 项时，开销增幅在最优情况(Ordinary 调用方式)下为 13%~18%。通过调用排序、并发调用、恶意线程调度和时间侧信道攻击向量这四个例子实验验证了防御者能在 SGX-SEF 中灵活部署多种安全策略以抵御多类攻击向量。

## 6.2 展望

结合当前可信执行环境的发展趋势，在本文工作基础上，还有以下几个研究点值得进一步探索。

1. 分析内核可信部分并联合 SGX-SEF 形成综合性安全增强框架。SGX 设计中仅保护用户态程序敏感部分，将内核等模块排除在 TEE 之外并认为其不可信。SGX Enclave 只具备用户态能力，为保证其能正常运行，Enclave 严重依赖于内核提供的进程机制和段页表机制等。许多 SGX 攻击正是利用 Enclave 对内核的依赖发起攻击。在实际应用中完全假设内核不可信可能过于严苛，后续研究可深入分析内核可信部分和不可信部分，通过可信度量及挑战应答等手段从 Enclave 出发扩张可信域，更好地权衡可信域的 TCB 尺寸和安全性。当假设内核的部分模块可信时，可进一步将 LSM 为例的内核安全增强模块和 SGX-SEF 结合，实现综合性安全增强框架，保护 Enclave 安全。
2. 针对其他 TEE 技术设计安全增强框架。除 Intel SGX 外的其他 TEE 技术如 ARM TrustZone 也存在着可信环境功能依赖不可信环境及可信环境接口消毒不充分等问题。对此，后续研究可以借鉴本文的方案，深入刻画其它 TEE 架构并分析关键执行路径和安全关键点，设计实现对应的安全增强框架，以审计关键执行路径，加固安全关键点，全面提升安全性。此外，也非常需要进一步研究如何针对具体应用场景更好地权衡 TEE 技术的 TCB 尺寸、功能性及安全性等。

## 参考文献

- [1] Costan V, Devadas S. Intel SGX Explained[J]. IACR Cryptol. ePrint Arch., 2016, 2016(86): 1-118.
- [2] 董春涛,沈晴霓,罗武,吴鹏飞,吴中海.SGX 应用支持技术研究进展[J].软件学报,2021,32(01):137-166.
- [3] Intel Corporation. Intel® Software Guard Extensions (Intel® SGX) Developer Guide, 2020. Revision Number 2.10.
- [4] Intel Corporation. Intel® Software Guard Extensions (Intel® SGX) SDK for Linux\* OS Developer Reference, 2020. Revision Number 2.10.
- [5] Intel Corporation. Intel64 and IA32 Architectures Software Developers Manual, 2020. Reference no. 325384-072US.
- [6] Jang I, Tang A, Kim T, et al. Heterogeneous isolated execution for commodity gpus[C]//Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. 2019: 455-468.
- [7] Volos S, Vaswani K, Bruno R. Graviton: Trusted execution environments on gpus[C]//13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). 2018: 681-696.
- [8] Criswell J, Dautenhahn N, Adve V. Virtual ghost: Protecting applications from hostile operating systems[J]. ACM SIGARCH Computer Architecture News, 2014, 42(1): 81-96.
- [9] Gueron S. A memory encryption engine suitable for general purpose processors[J]. IACR Cryptol. ePrint Arch., 2016, 2016: 204.
- [10] 王鹏,樊成阳,程越强,等. SGX 技术的分析和研究[J]. Journal of Software, 2018, 9: 2778-2798.
- [11] Kim T. Security Issues on Intel SGX. ACM CCS 2017. <https://www.youtube.com/watch?v=0TXR2JNsFBk&t=4s>
- [12] Checkoway S, Shacham H. Iago attacks: why the system call API is a bad untrusted RPC interface[C]//Proceedings of the eighteenth international conference on Architectural support

- p>for programming languages and operating systems. 2013: 253-264.
- [13] Lee S, Kim T. Leaking uninitialized secure enclave memory via structure padding[J]. arXiv preprint arXiv:1710.09061, 2017.
  - [14] Van Bulck J, Oswald D, Marin E, et al. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes[C]//Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019: 1741-1758.
  - [15] Khandaker M R, Cheng Y, Wang Z, et al. COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX[C]//Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 2020: 971-985.
  - [16] Wang J, Cheng Y, Li Q, et al. Interface-based side channel attack against intel SGX[J]. arXiv preprint arXiv:1811.05378, 2018.
  - [17] Van Bulck J, Minkin M, Weisse O, et al. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution[C]//27th {USENIX} Security Symposium ({USENIX} Security 18). 2018: 991 – 1008.
  - [18] Chen G, Chen S, Xiao Y, et al. SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution[C]//2019 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, 2019: 142-157.
  - [19] Lee S, Shih M W, Gera P, et al. Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing[C]//26th {USENIX} Security Symposium ({USENIX} Security 17). 2017: 557-574.
  - [20] Xu Y, Cui W, Peinado M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems[C]//2015 IEEE Symposium on Security and Privacy. IEEE, 2015: 640-656.
  - [21] Van Bulck J, Weichbrodt N, Kapitza R, et al. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution[C]//26th {USENIX} Security Symposium ({USENIX} Security 17). 2017: 1041-1056.
  - [22] Weichbrodt N, Kurmus A, Pietzuch P, et al. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves[C]//European Symposium on Research in Computer Security. Springer, Cham, 2016: 440-457.
  - [23] Sanchez Vicarte J R, Schreiber B, Paccagnella R, et al. Game of threads: Enabling

- asynchronous poisoning attacks[C]//Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 2020: 35-52.
- [24] Van Bulck J, Piessens F, Strackx R. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018: 178-195.
- [25] Van Schaik S, Milburn A, Österlund S, et al. RIDL: Rogue in-flight data load[C]//2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019: 88-105.
- [26] Baumann A, Peinado M, Hunt G. Shielding applications from an untrusted cloud with Haven[C]//Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation. 2014: 267-283.
- [27] Tsai C C, Porter D E, Vij M. Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}[C]//2017 {USENIX} Annual Technical Conference ({USENIX} {ATC} 17). 2017: 645-658.
- [28] Arnaudov S, Trach B, Gregor F, et al. {SCONE}: Secure linux containers with intel {SGX}[C]//12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). 2016: 689-703.
- [29] Hunt T, Zhu Z, Xu Y, et al. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data[C]//12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). 2016: 533-549
- [30] Lind J, Priebe C, Muthukumaran D, et al. Glamdring: Automatic application partitioning for intel {SGX}[C]//2017 {USENIX} Annual Technical Conference ({USENIX} {ATC} 17). 2017: 285-298.
- [31] Shih M W, Lee S, Kim T, et al. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs[C]//NDSS. 2017.
- [32] Chen G, Wang W, Chen T, et al. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races[C]//2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018: 178-194.
- [33] Ahmad A, Joe B, Xiao Y, et al. OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX[C]//NDSS. 2019.

- [34] Orenbach M, Lifshits P, Minkin M, et al. Eleos: ExitLess OS services for SGX enclaves[C]//Proceedings of the Twelfth European Conference on Computer Systems. 2017: 238-253.
- [35] Orenbach M, Michalevsky Y, Fetzer C, et al. CoSMIX: a compiler-based system for secure memory instrumentation and execution in enclaves[C]//2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19). 2019: 555-570.
- [36] Orenbach M, Baumann A, Silberstein M. Autarky: closing controlled channels with self-paging enclaves[C]//Proceedings of the Fifteenth European Conference on Computer Systems. 2020: 1-16.
- [37] Gruss D, Lettner J, Schuster F, et al. Strong and efficient cache side-channel protection using hardware transactional memory[C]//26th {USENIX} Security Symposium ({USENIX} Security 17). 2017: 217-233.
- [38] Matetic S, Ahmed M, Kostiainen K, et al. {ROTE}: Rollback protection for trusted execution[C]//26th {USENIX} Security Symposium ({USENIX} Security 17). 2017: 1289-1306.
- [39] Morris J, Smalley S, Kroah-Hartman G. Linux security modules: General security support for the linux kernel[C]//USENIX Security Symposium. ACM Berkeley, CA, 2002: 17-31.
- [40] Smalley S, Vance C, Salamon W. Implementing SELinux as a Linux security module[J]. NAI Labs Report, 2001, 1(43): 139.
- [41] Tian H, Zhang Y, Xing C, et al. Sgxkernel: A library operating system optimized for intel SGX[C]//Proceedings of the Computing Frontiers Conference. 2017: 35-44.
- [42] Tian H, Zhang Q, Yan S, et al. Switchless calls made practical in intel SGX[C]//Proceedings of the 3rd Workshop on System Software for Trusted Execution. 2018: 22-27.
- [43] Van Bulck J, Piessens F, Strackx R. SGX-Step: A practical attack framework for precise enclave execution control[C]//Proceedings of the 2nd Workshop on System Software for Trusted Execution. 2017: 1-6.
- [44] Oleksenko O, Trach B, Krahn R, et al. Varys: Protecting {SGX} enclaves from practical side-channel attacks[C]//2018 {Usenix} Annual Technical Conference ({USENIX} {ATC} 18). 2018: 227-240.



## 致 谢

在中科院软件所基础软件实验室的三年硕士研究生生涯即将结束。大量调研经历使我对研究工作有了更深入的理解，研究过程中与他人的沟通合作锻炼了我的表达能力和合作能力，端正了我的科研态度。在此由衷感谢各位老师和同学的帮助。

首先非常感谢我的导师马恒太副研究员。马老师在我的科研和生活中提供了许多帮助。他丰富的阅历及严谨的治学风格对我具有很大的启发作用，指引我在科研路上脚踏实地地不断探索前行。对我的毕业设计提出了许多宝贵意见，帮助我改进毕业设计。

非常感谢周启明老师的悉心指导。周老师也极大地助力我的科研和生活。在我的科研方向和科研历程上提供了全面帮助。在毕业设计期间帮助我研究分析课题的框架和细节，在工程实践上提供积极的指导，不断帮助我完善论文。

感谢研究员贺也平老师。他教会我如何研究一门学问，如何在科研道路上发现问题并解决问题。他还经常指出我所陷入的思维困境，帮助我脱离困境继续研究。

感谢实验室的其他老师。他们对我的毕业设计提出了许多建设性的意见，帮助我更好地完成毕业设计。

感谢吴晓慧学姐、林少峰学长和霍天霖学长。他们帮助我不断推敲研究课题，让研究课题更具意义。与我不断讨论实现细节，确定下一步实施计划。

感谢丁羽。他对 Intel SGX 及其它安全相关领域都有长足的研究，并且不厌其烦地教会我许多关于 SGX 的细节知识，帮助我确认工程实践中的许多实现细节。

感谢家人和朋友的支持。他们在我硕士期间遇到各类学术及生活问题时悉心地照顾我，帮助我缓解心理压力并摆脱困境，替我解决了许多烦恼。

最后感谢参与论文评审答辩工作的各位老师，您辛苦了！



## 作者简历及攻读学位期间发表的学术论文与研究成果

### 作者简历

2014 年 9 月——2018 年 7 月，在北京交通大学计算机与信息技术学院获得学士学位。

2018 年 9 月——2021 年 7 月，在中国科学院大学软件研究所攻读硕士学位。

### 已发表的学术论文

1. Xiaohui W, Yeping H, Qiming Z, Hengtai M, Liang H, Wenhao W, Liheng C. Partial-SMT: Core-scheduling Protection Against SMT Contention-based Attacks[C]//2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). IEEE, 2020: 378-385.

### 参加的研究项目

1. 核高基课题：“开源操作系统内核分析和安全评估”（项目编号：2012ZX01039-004）
2. 中科院先导 A 课题：“关键软硬件系统安全性分析评估核心技术研发及验证”（项目编号：XDA-Y01-01）

