涉密论文 □　公开论文 □

浙江大学

本 科 生 毕 业 论 文（设计）

题目　基于 SGX 的 API 密钥安全保护技术研究

姓名与学号　刘丁豪 3150104928

指导教师　陈建海

年级与专业　大四 计算机科学与技术

所在学院　计算机科学与技术

提交日期　2019.5.27

浙江大学本科生毕业论文（设计）承诺书

1. 本人郑重地承诺所呈交的毕业论文（设计），是在指导教师的指导下严格按照学校和学院有关规定完成的。

2. 本人在毕业论文（设计）中除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 浙江大学 或其他教育机构的学位或证书而使用过的材料。

3. 与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

4. 本人承诺在毕业论文（设计）工作过程中没有伪造数据等行为。

5. 若在本毕业论文(设计)中有侵犯任何方面知识产权的行为，由本人承担相应的法律责任。

6. 本人完全了解 浙江大学 有权保留并向有关部门或机构送交本论文（设计）的复印件和磁盘，允许本论文（设计）被查阅和借阅。本人授权 浙江大学 可以将本论文（设计）的全部或部分内容编入有关数据库进行检索和传播，可以采用影印、缩印或扫描等复制手段保存、汇编本论文（设计）。

作者签名：　　导师签名：

签字日期：　　年　　月　　日　　签字日期：　　年　　月　　日

致　谢

感谢纪守领老师对研究进度、实验设计与论文写作上的长期指导；感谢梁斯庄老师在论文写作与修改上的大力指导，特别是在英文论文上倾注的心血；感谢陈建海老师在实验资源准备与论文写作、图表绘制等多方面的指导与支持；感谢何钦铭老师在研究方向与技术上提供的指导；感谢浙江大学智能计算与系统实验室在科研环境和研究经费上支持；感谢父母与爷爷奶奶在本人学习及生活上提供的精神及经济支持与帮助；感谢张凯帆在本人大学期间给予的学习与生活上的支持；感谢华为公司提供的企业实习机会，令本人收获颇丰；感谢檀景辉老师在本人实习期间给予的技术指导与帮助；感谢何见听、王备、卢令令、张淼、翁海琴、侯文龙、胡思昊、周骏丰等实验室师兄师姐在本人科研过程中给予的大力帮助；感谢游瑞杰老师在本人大学期间给予的生活与学校事务上的帮助。

摘　要

随着云计算技术的成熟，当今应用程序可以广泛使用各种各样的云服务。通常应用程序通过应用程序编程接口（API）来访问云服务提供商（CSP）提供的云服务。出于安全性考虑，用户程序的 API 请求需要通过某种方式进行身份认证，如基于 API 密钥的签名。API 密钥便成为解锁云上用户资源的关键。然而，目前并没有一种完备的 API 密钥保护方案，特别是对那些需要频繁访问云服务的自动化程序。因 API 密钥泄露导致的安全问题层出不穷。

为了加强客户程序中 API 密钥的安全性，在本文中我们提出了 API Key Guard（AKGuard）系统来保护应用程序 API 密钥。AKGuard 应用因特尔 SGX 技术，将 API 密钥的导入、存储和使用过程纳入 SGX 安全区（enclave）的保护。其作为第三方库来管理应用程序的 API 密钥，并为 API 请求计算签名，在确保整个 API 密钥使用安全的同时可被轻松应用到多种应用程序中，无须开发者大幅调整原有程序。为了验证系统实用性，我们用 AKGuard 分别为主流云厂商（谷歌云，微软 Azure，亚马逊 AWS，百度云，阿里云，腾讯云）的服务请求进行认证，共计测试了 721800 个云 API 请求。测试结果表明 AKGuard 的密钥存储量充足，对应用程序的 API 请求吞吐量影响也在合理范围内（一般小于 10%）。

关键词：可信执行环境；SGX；API 安全；密钥保护

Abstract

The mature of cloud computing brings various cloud services for applications. These cloud services are accessed by applications via Application Programming Interfaces (API), which are provided by the Cloud Service Providers (CSP). With a concern of security, API requests from client applications are usually required to be authenticated by some methods, e.g., API keys. API keys then unlock the treasure on the cloud. However, there is no satisfactory solution yet for client applications to protect their API keys, especially for automated applications that frequently access cloud services.

To enhance the security of API keys in client applications, in this paper, we present API Key Guard (AKGuard), a system to safeguarding API keys. AKGuard serves as a third-party service to manage API keys of applications and compute signatures for API requests. We describe the design principles and architecture of AKGuard and implemented it using Intel Software Guard Extensions (SGX). In order to validate the practicality of our system, we applied AKGuard to authenticate API requests on well-known CSPs including Google Cloud, Microsoft Azure, Amazon AWS, Baidu Cloud, Alibaba Cloud and Tencent Cloud. We evaluated AKGuard against 721,800 practical cloud API requests in the real world. The evaluation results showed that the key storage capacity of AKGuard is sufficient and the performance loss of request throughput is under control (usually within 10%).

Key words: TEE、SGX、API Security、Key Protection

目　录

第一部分　毕业论文

第一部分

毕业论文

1 绪论

1.1 API密钥保护问题背景与研究意义

随着云服务等第三方网络服务的兴起，借助第三方平台服务进行信息存储、管理、计算等的成本不断降低，因而涌现出越来越多的云服务提供商（如谷歌云[1]、亚马逊AWS[2]、微软Azure[3]、阿里云[4]、百度云[5]、腾讯云[6]）跨服务应用程序，包括平台的后端程序、网络游戏、量化交易程序、移动端APP等。种类多样的云服务为开发者的数据管理工作提供了极大的灵活性。为了促进云服务的使用，云服务提供商（CSP）通常会为开发者提供应用程序编程接口（API），进而令开发者的应用程序可以直接访问相应的云服务。实际上，除了云服务提供商的云服务，还有大量网络服务以web API的形式提供给开发者。如数字货币交易平台Bitfinex[7]、Bitstamp[8]和Binance[9]等，均为其用户提供了API，用于构建自动交易机器人（如Gekko[10]、Blackbird[11]）并进行程序化交易。这些web API扩展了应用程序的能力，并已经成为诸多商业系统的核心。

第三方API的广泛使用也让API安全保护的地位愈发重要。在大多数情况下，API请求均需要在发送前经过某种方式的认证。目前已有的身份认证方法有OAuth认证[12], [13]、令牌认证等等。在本文中，我们主要考虑基于API密钥的身份认证方法，一种因其高易用性和可靠性而被广泛应用于云服务场景中的身份认证方法。部分云服务接口出于安全性考虑，甚至只允许使用API密钥进行访问，如Google Cloud Natural Language API[14]。API密钥一般为对称密钥，由用户在服务提供方的平台（如阿里云）上进行申请，并为密钥授予特定的权限（如云MySQL数据库服务只读权限）。平台方存有API密钥的备份以进行验证工作。用户通过API密钥计算请求签名字段供服务端校验。在以API密钥为基础的身份认证机制中，密钥信息本身不会以明文形式传递，即使信息传输信道被监听也不会泄露隐私数据。几乎所有开放的网络服务API均支持通过API密钥进行身份认证保护。

API密钥为用户程序与服务平台直接建立了信任桥梁，但一旦API密钥遗失，对服务平台以及用户均可能造成重大损失。对用户而言，API密钥代表用户自己的身份，而API密钥本身不会识别其调用者。API密钥一旦遗失，攻击者便

立即拥有了API密钥对应的权限，进而利用用户的API密钥篡改或窃取用户的云端数据;对于按照API使用次数付费的第三方服务,还可能造成额外的服务费用。对于平台而言，攻击者可利用窃取的API密钥恶意访问服务接口，增加服务器负担。尽管如此，仍有大量开发者为了方便使用而将API密钥硬编码在程序中[15]，或将API密钥以明文存储在文件或数据库中[16]，存在极大安全隐患。在2018年3月的币安数字货币交易所的安全事件中，攻击者疑似利用窃取的大量API密钥进行程序化恶意交易以操纵币价，获取巨额利益，并给全球数字货币市场带来了极其恶劣的影响[17]。令人震惊的是，如出一辙的手法仅仅在4个月后就再次得手，并造成了约4500万美元的损失[18]。

保护API密钥是应用程序安全调用第三方服务接口的基础，如果API密钥保护不当，整个基于API密钥的身份认证机制便形同虚设。本文将Intel SGX[19]技术引入API密钥身份验证过程，借助可信执行环境保障API密钥的使用、传输和存储安全。同时，本文的系统可以应用于多种实际的云服务场景和厂商，并最大限度便捷开发者在生产过程中使用该技术。

1.2 研究现状及本文贡献

在本节，我们简要分析API密钥在工业界和学术界的保护策略，以及本文贡献。API密钥保护策略又大致可以分类为工业界最佳实践、基于软件的保护方法和基于硬件的保护方法。

1.2.1 工业界最佳实践

鉴于API密钥保护的重要意义，许多云服务提供商均给出了各自的API密钥使用指导来帮助开发者管理自己的API密钥。通过调研主流云平台的开发文档，可将指导策略分为如下三点：

（1） 设置API密钥权限限制。此策略用于限制API密钥可以干什么。几乎所有的云服务提供商（如谷歌云[1]、微软Azure[3]、腾讯云[6]）均建议开发者削弱API密钥的权限，如使用只有部分权限（如只读）的API密钥而非根密钥、指定API密钥生效的云服务（如对象存储）并使用不同的API密钥访问不同的云服务。

（2） 设置API密钥上下文限制。此策略用于限制谁可以使用API密钥。一般的限制条件包括IP地址和HTTP refer。我们也可以将API密钥的使用限制在指定的Android应用（添加包名称和SHA-1签名证书的指纹）或IOS应用（添加IOS bundle identifier）[20]。此外还建议开发者为不同的应用程序申请不同的API密钥。

（3） 遵循安全的密钥管理策略。此策略规范了开发者如何使用API密钥。一方面，开发者应当在代码开源之前进行详尽的代码审核并确保源码中没有嵌入API密钥。另一方面，开发者不应该将API密钥存储在应用程序文件树之中的文件里。受到推荐的做法是将API密钥隐藏在环境变量或非工程目录下的文件中。开发者还应该删除不需要的API密钥并定期重新生成。

这些最佳实践可以提高API密钥保护的安全性，但无法为开发者提供完备的保障。设置API密钥的种种限制条件仅仅可以减少API密钥丢失的损失。即便API密钥被隐藏在环境变量中，密钥本身仍然以明文存储。攻击者完全有机会窃取这些密钥并非法使用。

1.2.2 基于软件的保护方法

不同于可以只存储信息摘要的密码信息，API密钥内容需要作为签名算法的输入信息，因而应用程序必须使用密钥明文信息。一些云服务提供商建议在实际使用过程中对API密钥进行混淆[20]。然而，具体的混淆过程并没有明确给出。目前有一些专业的代码混淆工具（如DexGuard[21]、ProGuard[22]、JavaScript Obfuscator[23]），但这些工具通常是为特定的应用程序（如Android应用、JavaScript应用）设计的，无法提供普适性的服务。此外，攻击者仍有可能将混淆后的信息解码或逆向工程来获取机密信息[24], [25]。

目前将信息加密仍然是最有效的信息保护手段之一。对于加密策略，我们可以使用一些加密与密钥管理服务（KMS）工具来保护API密钥。Openstack Barbican[26]是一套用于进行安全存储和机密信息管理的REST API工具。包括密码、加密密钥、X.509证书等在内的机密信息均可纳入到保护范围。但是基于软件的加密策略容易受到具有系统权限的攻击者的攻击。除此之外，开发者要学习掌握一个KMS系统的使用方法，并将其加入到原有的应用程序中也是一个不

小的负担。值得注意的是，即使进行加密，使用API密钥时程序也必须对密钥信息进行解密，攻击者可通过对程序内存进行监视来窃取密钥信息。

另一个常用策略是把API密钥集中到一个相对安全的代理服务器上，随后把所有的应用程序请求均路由到此服务器上（如Approov[27]）。这种策略将机密信息从应用程序端彻底移除，因而攻击者不可能通过逆向等手段从应用程序中窃取API密钥。在这种情况下，API密钥的保护责任从应用程序端转移到了代理服务器端。代理服务器则同样需要借助其他技术或方法（如加密、可信硬件）来确保自己的安全性。此外，代理服务器与应用程序直接也通过API进行交互，这些API本身也会成为此策略的重要弱点。

1.2.3 基于硬件的保护方法

硬件安全模块（HSM）有能力为API密钥提供高安全性的管理服务，可以消除上述提到的密钥存储和使用过程中存在的安全隐患。但是硬件安全模块价格昂贵，其对于独立开发者和小型开发团队可能并不适合。

HongQian Karen Lu 等人首次提出了基于安全元件（SE）的密钥保护与使用机制[28]，可以对 API 密钥的使用流进行完整的保护。这种方法同样可以解决API密钥存储和使用过程中的安全隐患，但使用安全元件需要用户认证，这将这套方案的使用场景限制在了需要人参与交互的场景。因而此方案对于需要频繁访问云服务的应用程序及其开发者可能并不友好。

Fortanix 自保护密钥管理服务（SDKMS）声称是世界上首个基于Intel SGX的云安全解决方案[29], [30]。SDKMS可以安全地生成、存储和使用加密密钥和证书，以及包括API密钥在内的其他机密信息。不幸的是，我们仍然可以从其官方代码中发现从环境变量中获取的API密钥，这些密钥被用于访问其自身的SDK[31]。为了保护已有的机密信息而引入新的机密信息是不明智的。除此之外，根据其官方文档，仍不清楚SDKMS是否支持SGX远程认证。

Intel构建了一套基于SGX的Openstack Barbican插件，名为BarbiE[32]。BarbiE提供了与安全硬件相媲美的高安全性，同时也具有与基于软件的方法相近的低成本。BarbiE还支持多重认证以及在多方场景中的多用户密钥分发机制。然而，对于那些未曾使用Barbican进行密钥管理的应用程序，开发者可能需要花大力气来学习使用方法与重构程序。

1.2.4 本文贡献

总体而言，本文贡献大致可总结为如下四点：

（1） 我们提出了API Key Guard （AKGuard），一种为应用程序提供安全API密钥管理的第三方服务。AKGuard运用SGX硬件保护技术保障API密钥在存储和签名环节的安全性。开发者无需重写或大幅重构其原有程序即可方便地将AKGuard引入程序中并保护API密钥。

（2） 我们实现了AKGuard的一个系统原型，该原型系统可以方便地扩展，并且可以在保障安全的前提下为多种应用程序（服务器后端、量化交易程序等）提供服务。我们计划在必要的代码检查和程序改良后将其开源。

（3） 我们定义了一套安全的数据流，并分析了针对AKGuard的潜在攻击风险与应对策略。我们的系统可以抵御不可信操作系统（OS）以及不安全网络环境带来的风险。

（4） 我们通过为来自六个主流云服务提供商（谷歌云、亚马逊AWS、微软Azure、百度云、阿里云、腾讯云）的六种不同云服务提供API密钥保护与签名来测试AKGuard的使用性能，同时也显示出AKGuard有能力支持多种云服务提供商的云服务。

1.3 本文结构

本文的第一章简述了课题的研究背景、研究意义以及研究现状；第二章简述了API密钥身份认证技术和Intel SGX技术的基础理论与方法；第三章阐述了三项我们进行的SGX数据安全保护研究，并总结了SGX的适用场景；第四章描述了我们设计的API密钥保护系统AKGuard的系统模型、安全假设和预期目标；第五章详述了AKGuard的系统设计；第六章给出了相关的实验设置与结果；第七章简述了我们未来对AKGuard系统的改进计划；第八章对本文进行了总结。

2 API 密钥认证与 Intel SGX 技术综述

2.1 基于API密钥的身份认证技术

2.1.1 基于API密钥的身份认证原理

API密钥本质上是随机字符串，字符串长度在不同的云服务提供商中又有所不同，一般为32-80字节。API密钥通常由两段构成：一段为Access Key，一段为Secret Key，如下表所示（密钥申请自亚马逊AWS[2]）：

表 2.1 API 密钥示例

Access Key	AKIAVNKMVVAAPXFY5I4B
Secret Key	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

Access Key 的地位相当于用户账号，Secret Key 则大致相当于账号的密码。一般情况下，云服务提供商会自行保存一份用户 API 密钥的副本。在认证过程中，应用程序使用 Secret Key 来生成一个合法的消息签名，并将其添加到相应的 API 请求中。最常用的签名生成算法是哈希消息认证码（HMAC）[33]。Access Key 本身则直接添加到请求字段中。当云服务提供商的认证服务器接收到用户的 API 请求后，认证服务器首先会根据 Access Key 找到用户 Secret Key 的副本。之后，认证服务器用相同的签名生成算法重新计算签名，并与用户提供的签名进行比较。如果签名一致则视为合法请求并进行相应的请求处理，否则视为非法请求返回签名无效信息。图 2.1 显示了百度云 API 请求的流程[34]：

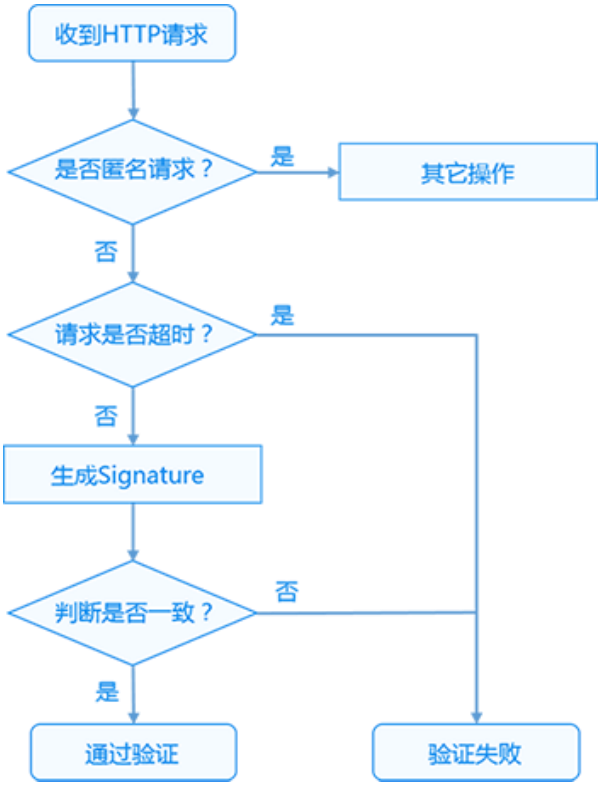


图2.1 百度云API请求认证

在如上认证流程中Secret Key必须被全存储与使用，绝对不能泄露到公开环境中。一般而言，云服务提供商可通过API网关或硬件安全模块等技术确保服务端机密信息的安全。如果认证过程使用的是非对称加密算法生成的签名，那么服务端甚至不需要存储任何机密信息。因此，API密钥的安全性主要取决于应用程序开发者自身的能力。

2.1.2 哈希消息认证码

哈希消息认证码（HMAC）[33]是一类特殊的消息认证码（MAC）。在生成消息认证码的过程中，需要使用一个密码学哈希函数（如SHA256，MD5），并以一段消息和一个密钥值作为输入信息。而与此对应的消息认证码算法就称之为HMAC-X，X代表所使用的密码学哈希函数（如HMAC-SHA256，HMAC-MD5）。虽然消息认证码不是标准的签名，但其凭借着高易用性和不错的安全性而广泛应用于客户端身份认证过程。

为了抵御重放攻击，在计算哈希消息认证码的过程中通常需要引入噪音信息来确保每次计算结果的唯一性。目前业界有两种常用的防御手段：OATH 质疑-应答算法（OCRA）[35]和基于时间的一次性密码认证算法（TOTP）[36]。

在OATH质疑-应答算法中，客户端首先向认证服务器发送API请求，之后认证服务器并不会直接通过，而是返回一条未认证消息，并附带一个随机的nonce字符串。客户端收到nonce字符串后将其添加到消息认证码计算的输入信息中，并生成新的签名信息。由于每次nonce字符串仅仅使用一次，所以签名本身可以保障唯一性。

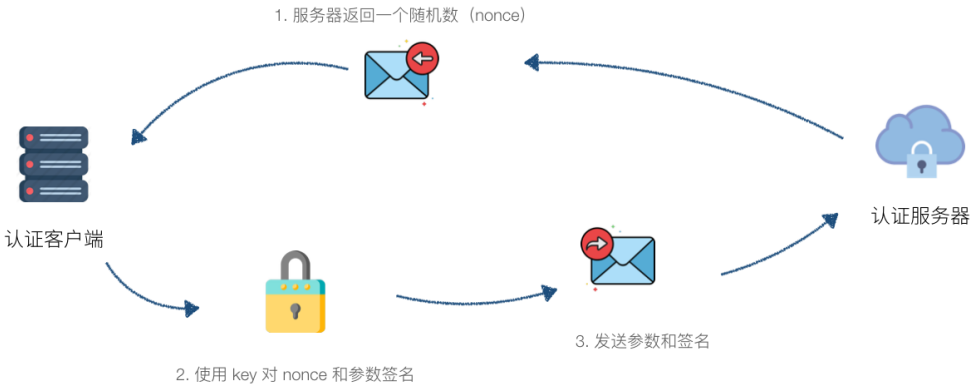


图2.2 OATH质疑-应答算法

OATH质疑-应答算法需要一次额外的信息交互。为了消除获取nonce值的额外请求，基于时间的一次性密码认证算法将时间戳信息而非nonce字符串作为签名信息的额外输入。通过此种方式生成的签名有一个有效时间窗口（如1分钟）。在这种情况下，客户程序和认证服务器需要提前进行时间同步。目前，我们的系统只支持基于时间的一次性密码认证算法。

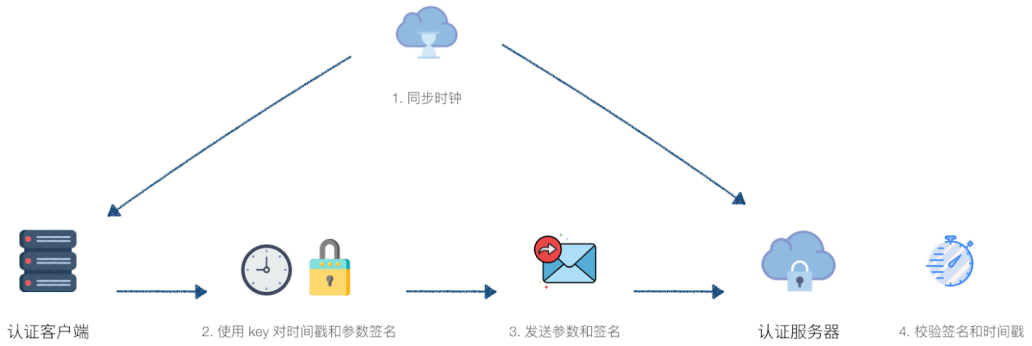


图2.3 基于时间的一次性密码认证算法

2.2 Intel SGX技术

2.2.1 SGX基础概念综述

Intel 软件保护扩展（SGX）[19]是一套为Intel体系处理器而增加的指令与保护机制，其允许应用程序为代码和数据创建可信的加密内存区域，称为安全区（enclave）。Intel SGX可被抽象为一个具有较小攻击面的可信执行环境。安全区拥有基于硬件支持的机密性和完整性保护，可防止来自系统层面的嗅探和干扰。安全区内的数据只可被安全区内的代码访问。即使面对特权软件（操作系统、虚拟机监视器等），安全区仍可以保护内部信息不被泄露或篡改。安全区可以被编译为动态库并被加载到多种编程语言的应用程序中。SGX还通过系统安全区（AE）为开发者提供了可信单调计数器和时间戳。

图2.1展示了Intel SGX的保护流程，并可大致分为6个步骤。在编程过程中，应用程序被开发者人工分为不可信部分（Untrusted Part of App）与可信部分（Trusted Part of App）两个部分。在图示步骤1中，应用程序被编译为可信与不可信两部分，并开始执行。在步骤2中，应用程序创建并初始化SGX安全区，安全区内存受到硬件保护。步骤3中，应用程序在不可信部分调用安全区内的可信函数，执行过程转到应用程序的可信部分进行。步骤4中，SGX安全区内的可信函数可以获取安全区内的所有机密数据并加以处理，这些数据的访问请求如果来自安全区外的函数，则将被直接拒绝。步骤5中安全区内的可信函数完成相应功能并向应用程序的不可信部分返回处理结果。即使可信函数返回，安全区内

的非返回机密数据仍然只保留在安全区内。步骤6中应用程序在非可信部分继续完成其他处理流程。

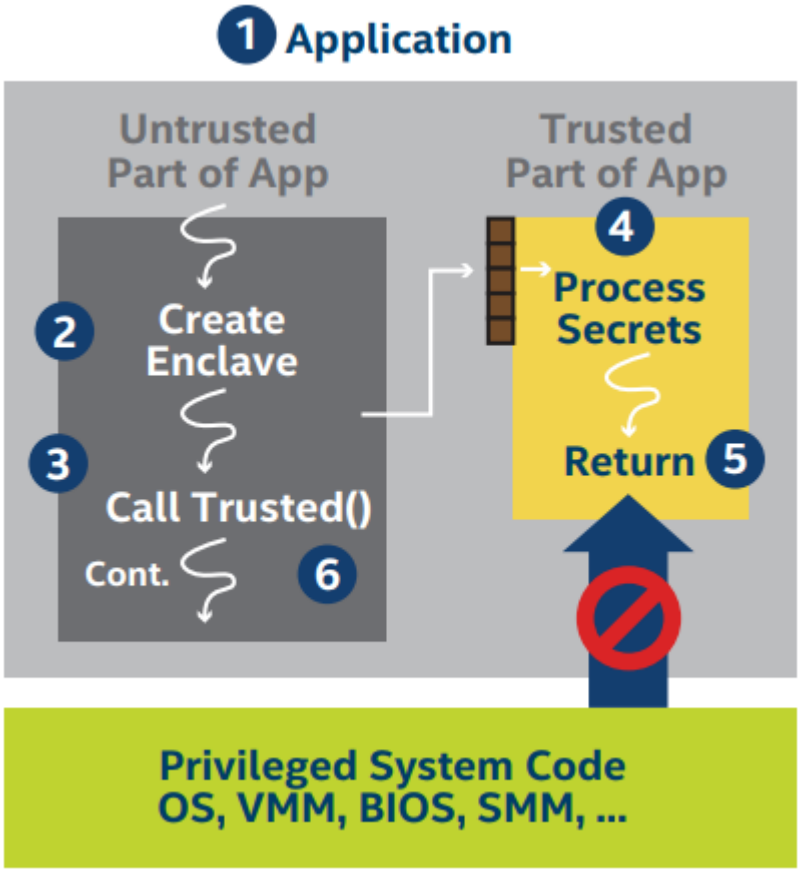


图2.1 SGX功能执行流程

Intel SGX技术的可信内存空间大小有硬性限制，一般为64MB或128MB。因而，活跃安全区的数量也需要加以限制。一般建议应用程序开辟的安全区数量为5到20个。

2.2.2 SGX 远程认证

虽然借助 Intel SGX 硬件保护技术，可以为现有的应用程序提供高安全性的保护，但事情并没有万事大吉。一个最常见而难以解决的问题是：应用程序安全区如何获取机密信息？有时可以要求用户输入机密信息，但提供可信的输入输出设备并非易事，输入数据极有可能被泄露给恶意软件。一部分关键数据甚至可能不是用户生成的，如由权威机构颁发的数字证书（包含用户私钥）。即使机密信息源与 SGX 服务器之间建立了加密通信信道，我们也无法保证 SGX 服

务器未被入侵。高权限攻击者可通过控制服务器网络层截取信息。

为了应对此类难题，Intel SGX 推出了远程认证（Remote Attestation）机制[37]。远程认证是一项 SGX 的高级特性，它通过一套定制的 Sigma 协议实现客户端和服务端的 Diffie-Hellman 密钥交换（DHKE），完成客户端到 SGX 安全区间的无缝安全对接。SGX 远程认证允许一个远程系统（非 SGX 程序）验证安全区内的程序，并为二者建立安全的通信信道。SGX 远程认证许安全区证明自身可信，并为我们提供了一个向安全区安全提供机密信息的渠道。

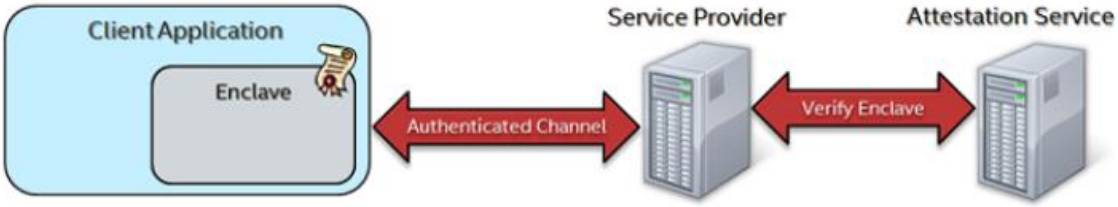


图2.2 SGX远程认证概览

如图 2.2 所示，SGX 远程认证主要包括三个组件：客户端程序、服务提供者、远程认证服务器。客户端程序为开发者编写的使用 SGX 保护的应用程序，其允许环境需要支持 SGX；服务提供者代表提供机密信息的用户服务器，其运行环境不需要 SGX 支持；远程认证服务器为 Intel 提供的辅助远程认证流的服务器。三者之间的数据交互流如图 2.3 所示。其中，ISV Application 表示客户端程序（以下简称为客户端），其中使用的 SGX 功能部分表示为 Intel SGX Runtime；ISV Remote Attestation Server 为用于提供机密数据的服务提供者（以下简称为服务端）；Intel Attestation Server 为 Intel 的远程认证服务器。

远程认证共包括 5 个交互数据流（msg0 – msg4）。

远程认证的第一步为客户端请求服务端提供机密信息（Request secret）。请求可通过访问服务端的特定 API 完成。随后服务端响应该请求并发出认证 Challenge。为了响应此 Challenge，客户端构造 msg0，构造流程为：（1）客户端不可信部分执行 ECALL 进入安全区；（2）在安全区内调用 sgx_ra_init()，将结果和 DHKE 上下文参数返回不可信部分；（3）调用 sgx_get_extended_epid_group_id()。其中，sgx_ra_init()将服务端的公钥（预先硬编码到安全区）作为参数，并在远程认证过程中向 DHKE 返回不透明上下文。

如果安全区使用了系统安全区（AE）功能（可信时间、单调计数器），其也需要包含在认证序列中（alt 框内部分）。完成 sgx_ra_init()后，客户端调用 sgx_get_extended_epid_group_id()检索 Intel Enhanced Privacy ID（Intel EPID）的扩展组 ID（GID）。Intel EPID 是一个用于身份验证的匿名签名方案，详细信息参考文献。GID 作为 msg0 的信息发送至服务端（GID 为非 0 值视为认证失败）。

远程认证的第 2 部发送信息 msg1 也为客户端到服务端信息，可以选择与 msg0 一同发送或先后发送。客户端调用 sgx_ra_get_msg1()构建包含面向 DHKE 客户端公钥的 msg1 信息（当 SGX 安全区开发者导入 sgx_tkey_exchange.edl 时 SGX SDK 自动生成该函数）。

接收到客户端的 msg1 信息后，服务端检查其中相关的值并生成自己的 DHKE 参数，之后向 Intel 认证服务器发送查询请求，以检查客户端发送的 EPID GID 签名撤销列表（SigRL）。在处理 msg1 与构造 msg2 时，具体流程为：（1）使用 P-256 曲线生成随机 EC 密钥，作为 Gb 参数；（2）从 msg1 中获得 Ga 参数，并从 Ga 和 Gb 中导出密钥导出密钥（KDK）；（3）在字节序列“0x01 || SMK || 0x00 || 0x80 || 0x00”上执行 AES-128 CMAC 算法从 KDK 中导出 SMK；（4）确定向客户端响应的请求引用类型；（5）设置 KDF_ID（通常为 0x1）；（6）使用服务端的 EC 私钥计算 ECDSA 签名；（7）计算 AES-128 CMAC，其中以步骤（3）中导出的 SMK 作为加密密钥；（8）查询 Intel 认证服务器，获取 SigRL。

客户端接收到 msg2 后，调用 sgx_ra_proc)msg2()生成 msg3，该函数执行如下工作：（1）验证服务端的签名；（2）检查 SigRL；（3）返回 msg3，其中包含用于验证安全区的引用。

服务端接收到 msg3 后，服务端需要（1）验证 msg3 中的 Ga 参数是否匹配 msg1 中的 Ga 参数；（2）验证 $CMAC_{SMK}(M)$；（3）验证客户端提供的认证凭证；（4）验证通过后提取安全区的认证状态与 PSE，检查安全区标识、版本和 ID，并决定是否信任安全区和 PSE；（5）生成 msg4 并发送至客户端。msg4 的格式由服务端指定，其至少需要包含（1）安全区是否可信；（2）PSE 是否可信。其可选择性包含（1）PIB；（2）向安全区提供的机密信息，内容应先用从 KDK 导出的密钥进行加密；（3）安全区重新认证时间；（4）当前时间；（5）客户端

可以信任的其他服务器的公钥。

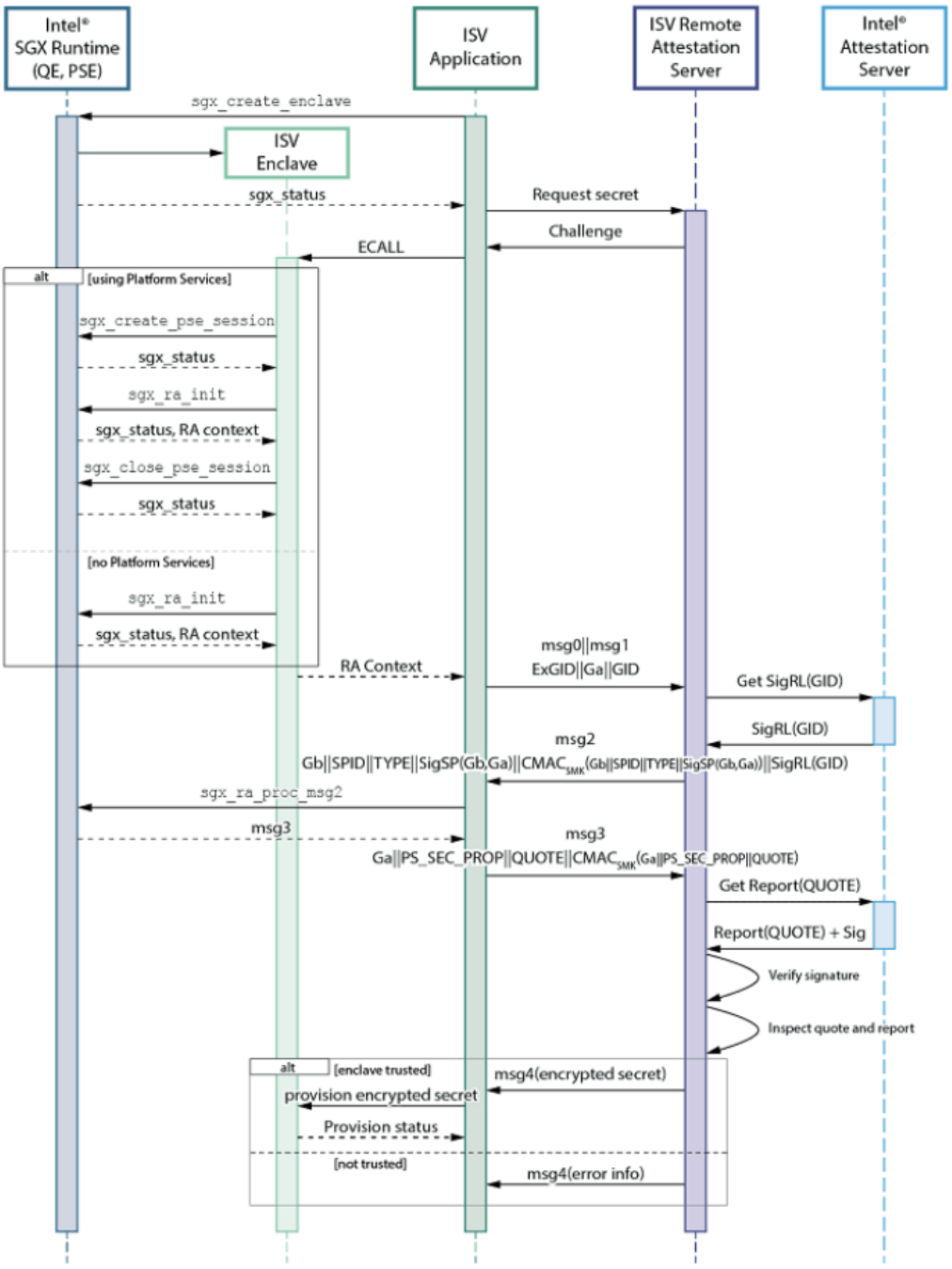


图2.3 Intel SGX远程认证流

2.2.3 SGX SSL

Intel SGX 本身不支持将额外的库直接加入到 SGX 安全区程序中。Intel SSL[38]密码库是Intel提供的用于为SGX安全区提供密码学支持的密码库。Intel

SSL 基于 OpenSSL 开源工程，但为了能使其运行于 SGX 安全区，Intel 对其进行了专门的修改重构。机密信息（如 API 密钥）的处理便不再需要离开安全区进行，如图 2.4 所示。

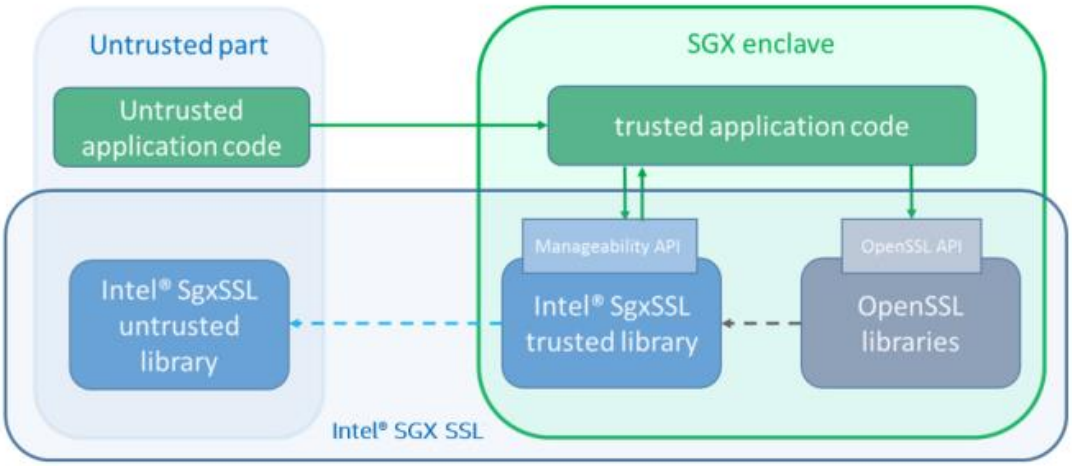


图2.4 Intel SGX SSL架构图

在我们的系统中，只使用了可信的 SGX SSL 库，这意味着所有机密信息相关的操作函数均会在不离开安全区的情况下返回。

3 基于 SGX 的数据安全保护研究

在本章，我们将介绍我们已经完成的作为 API 密钥保护系统 AKGuard 基础的 SGX 数据安全保护研究工作。具体包括：（1）基于 SGX 的工控白名单管理系统研究（2）基于 SGX 的远端存储服务方法研究；（3）基于 SGX 的区块链交易平台密钥保护研究。通过这三项研究工作，我们熟悉了 SGX 的保护特性与编程方法，并逐渐摸索出了 SGX 适合的应用场景，为 API 密钥保护系统 AKGuard 的设计与实现打下了基础。

3.1 基于SGX的工控白名单管理系统研究

3.1.1 研究背景

白名单技术广泛用于工业控制系统的安全保护中。相比黑名单，白名单的管

理维护更加方便，只需指定可信的设备与网络地址等信息，就可以基本保障工控系统交互环境的安全。目前白名单技术在工业防火墙及其他防护软件中的应用取得了不错的成效。

尽管白名单机制在工控安全保护中成效显著，但目前缺乏相关机制保障白名单内容的不可篡改性。对于高权限攻击者，存在更改白名单以令恶意软件绕过系统检测的可能。一旦恶意软件通过篡改白名单加入到可信软件列表，其便可长期驻足于工业系统中，红不堪设想。

3.1.2 研究内容

本项研究提出了一种基于 SGX 的工控白名单管理系统，借助 SGX 硬件的安全保护功能为工控白名单提供不可篡改的性质。系统架构如图 3.1 所示：

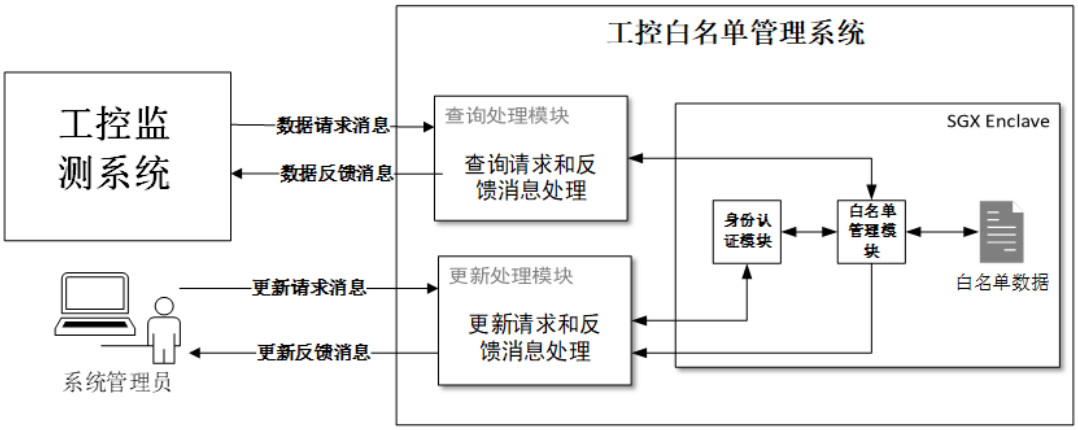


图3.1 基于SGX的远端存储服务系统框架

在本项研究的设计中，白名单管理系统自成体系，并为其他需要白名单支持的工控监测系统提供服务（一般白名单由工控监测系统自行构建并维护）。系统主要包括4个模块：查询处理模块、更新处理模块、身份认证模块和白名单管理模块。系统的运行流程如下：

（1） 工控白名单管理系统在没有白名单数据的情况下初始化，创建SGX安全区，并初始化安全区内的身份认证模块和白名单管理模块。

（2） 系统完成初始化后，系统管理员向白名单管理系统发送数据更新请求，将预先准备的白名单数据更新至SGX安全区内。一并发送的还有自己的身份凭证信息（如密码）。更新处理模块在安全区外接收更新消息并发送至安全区内的身份认证模块，根据身份凭证信息对管

理员进行身份检查。检查通过后通过安全区内的白名单管理模块更新白名单数据，并发送结果反馈信息。

（3） 完成白名单信息导入后，白名单管理系统开始向其他工控提供服务，接收来自工控监测系统的白名单查询请求，并通过查询处理模块和白名单管理模块执行查询消息检索。同时白名单管理系统接收来自系统管理员的实时白名单数据更新。

相关研究工作已申请国家发明专利（《基于SGX软件防护扩展指令的工控白名单管理系统及方法》，专利号201910400840X）并受理。

3.1.3 研究内容评价

该研究作为 SGX 安全保护应用的一个尝试，解决了工控系统白名单保护的一个子问题（完整性保护）。但应用该系统需要配备支持 SGX 功能的 Intel CPU，目前在工控系统中并不流行。而且对于需要更高安全性的重要工业系统，信任 Intel 的假设也并不合理。总体而言，本项研究尚不成熟，其应用前景有待进一步讨论。

3.2 基于SGX的远端存储服务研究

3.2.1 研究背景

远端存储是当前进行数据管理的一种流行手段。借助第三方云平台的远端存储服务提供商，我们可以进行更有效率与更加经济的数据管理。然而，与之对应的是日益猖獗的服务器攻击行为。存有重要数据的远端存储服务器一般不在本地管理，网络环境等因素更不可控。一旦遭受攻击，便会给企业带来不可估量的灾难，特别是数据价值导向的公司。安全保护远端存储服务器具有重要意义。

3.2.2 研究内容

本项研究提出了一种基于 SGX 的远端存储服务系统，借助 SGX 完成远端数据的加密与密钥管理，保护关键数据。系统架构如图 3.2 所示：

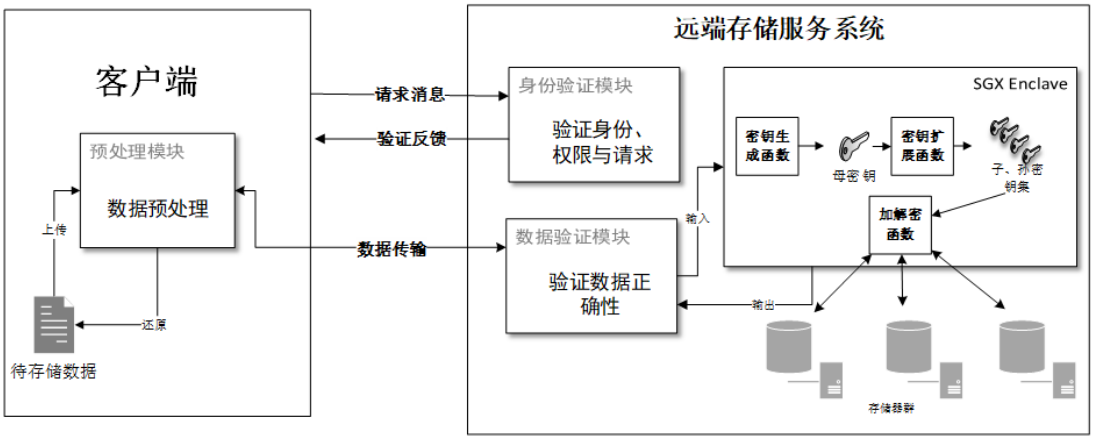


图3.2 基于SGX的远端存储服务系统框架

在本项研究的设计中，远端存储服务系统运行于远端存储服务器，与用户一侧的客户端通信并提供数据存储服务。系统主要包括 4 个模块：数据预处理模块、身份验证模块、数据验证模块和 SGX 加密模块（SGX Enclave 内）。系统的运行流程如下：

（1） 远端存储服务系统启动，创建 SGX 安全区，初始化安全区内部的密钥生成函数、密钥扩展函数与加解密函数。

（2） 系统初始化完成后，安全区内通过密钥生成函数生成用于母密钥。之后母密钥经过密钥扩展函数生成若干子密钥。子密钥用于加密远端存储服务系统上的关键数据。

（3） 客户端为了访问远端存储服务系统，首先与身份验证模块进行交互，通过用户凭证（如密码）完成身份验证。

（4） 客户端身份验证通过后，客户端构建相应的数据存储或查询请求，由数据预处理模块对数据格式进行预处理。之后经由安全传输信道发送至远端存储服务系统的数据验证模块。

（5） 数据验证模块验证数据完整性，之后远端存储服务系统根据用户身份在安全区内调出该用户的数据加解密密钥。对于数据存储请求，由加解密函数通过加解密密钥对数据执行加密，并存储与本地系统；对于数据查询请求，远端存储服务系统在本地执行检索，向加解密函数返回加密数据，之后加解密函数利用用户加解密密钥解密数据并返回数据验证模块。

（6） 数据验证模块检查解密数据，确认无误后将其传给客户端，客户端的预处理模块还原数据并反馈给用户。

相关研究工作已申请国家发明专利(《一种基于 SGX 的远端存储服务方法及系统》，专利号 CN201810470662.3，公开号 CN108768978A）并已公开。

3.2.3 研究内容评价

该项研究为远端存储服务的数据安全性存储提供了一个基于硬件的解决方案，将数据加解密过程移植到 SGX 安全区内，相关的密钥也只在安全区内存储与使用，可保证存储数据的安全。但不足之处在于，身份验证模块和数据验证模块不受硬件保护，易成为系统的攻击点。伪装用户身份的攻击者有可能发动绕过 SGX 硬件的攻击。SGX 安全区无法识别调用者身份，如何解决该特性带来的问题将成为 SGX 应用场景的重要限制因素。

3.3 基于SGX的区块链密钥保护研究

3.3.1 研究背景

区块链是新兴数字货币（比特币等）的底层分布式技术存储，实现了一种去中心化的货币发行与交易机制。区块链以现代密码学技术为基础，用户的数字资产由用户的非对称私钥保护，该私钥拥有区块链用户在链上的几乎全部权限（如执行转账)。一旦用户私钥丢失，意味着用户的全部数字资产随时面临失窃的威胁。

目前数字货币交易的主要途径是数字货币交易平台。为了方便执行交易，数字货币交易平台一般会保留用户的区块链私钥以构建合法交易。一旦数字货币交易平台遭受恶意攻击并泄露用户私钥，交易平台所有用户的数字资产均会受到威胁。相比单个用户，攻击数字货币交易平台更加有利可图，已有诸多大型区块链交易平台因黑客攻击而一蹶不振（Mt.Gox[39]、Bithumb[40]等)。

3.3.2 研究内容

本项研究提出了一种基于 SGX 的区块链交易平台——用户框架，将区块链交易平台的交易构造及其与平台用户的交互以 SGX 硬件保护。系统架构如图 3.3 所示：

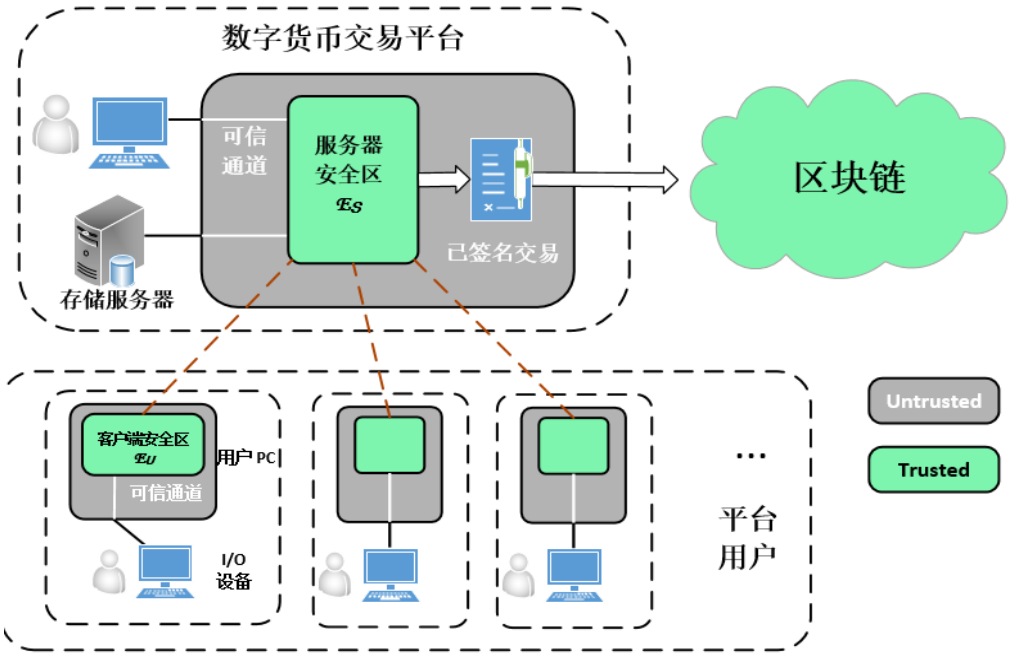


图 3.3 基于 SGX 的数字货币交易平台系统框架

整个系统主要由三部分构成：数字货币交易平台、交易平台用户和区块链链上系统。其中的数字货币交易平台和交易平台用户端（用户 PC）均需要配备有 SGX 功能的 CPU。统的运行流程如下：

（1） 数字货币交易平台及用户端初始化系统并创建 SGX 安全区。其中对于数字货币交易平台，创建 SGX 安全区后还需要通过一个存储服务器将部分用户的区块链私钥导入安全区。

（2） 在用户端，用户提供 SGXIO 将自己在平台上准备进行的区块链交易信息输入用户端 SGX 安全区。

（3） 用户端安全区根据用户输入信息构建区块链交易（交易对象、金额、时间等），并与安全区内的用户身份信息一般构造发往交易平台的平台信息。

（4） 数字货币交易平台的安全区与用户端安全区通过 SGX 远程认证建立通信信道，用户端将构造好的平台信息通过该信道发送至平台端安全区。

（5） 平台端在安全区内通过接收到的平台信息检查用户身份。检查通过后在安全区内构造区块链交易并签名。如果安全区内检索不到对应

用户的区块链密钥，则从安全区向存储服务器发起查询，并更新安全区内存储的密钥序列。

（6） 数字货币交易平台将完成签名的交易传出安全区并发送至区块链网络。交易的共识由区块链网络部分进行。自此交易完成。

相关研究工作已申请国家发明专利(《一种基于 SGX 的区块链用户密钥保护方法和装置》，专利号 CN201710989478.5）并已授权。

3.3.3 研究内容评价

该项研究中设计的系统可以在数字货币交易平台环境以及用户端操作环境均不安全的情况下正常运作，有较高的安全性。但与之对应的代价是要求所有参与方均需配备 SGX 功能的 CPU，此假设目前而言相当牵强。除此之外，每次用户交易均需进行一次 SGX 远程认证，对于高频交易者而言代价过大。应用此系统无论对交易平台还是用户均不够方便。当前的交易平台主要通过用户在平台上申请 API 密钥的方式作为身份凭证，身份验证便捷而且也具有一定的安全性保障。本系统的一个弱点在于：对用户端而言，为了保护区块链密钥，引入了新的验证口令（用于在 SGXIO 中输入，并在用户端安全区内验证)。

3.4 SGX应用场景总结

经过如上研究，我们总结出适合采用 SGX 保护的应用场景如下：

（1） 自动化应用程序。自动化应用程序本身的特性可最大限度降低人工交互。一方面，在此类应用场景中，通过人工授权（密码、密钥等）的方式管理信息受到限制，比较适合通过硬件方式实施保护。另一方面，对于非自动化应用程序，多数情况下通过人工授权的方式保护数据就已经足够（如以某口令作为机密信息的加密密钥执行加解密)，SGX 的应用优势并不明显。

（2） 强机密信息保护需求。SGX 提供对安全区数据的机密性保护，防止外界嗅探，适合为具有强机密信息保护需求的应用程序提供服务，保护敏感信息。对于涉及机密信息的操作（如签名)，使用 SGX 提供保护也相当合适。

（3） 程序运行环境不可控。对于多数情况下完全运行在用户端的程序，

其安全性本身就相对较高。因此，SGX 更适合那些需要长期运行在某服务器或托管在云上的应用程序。其运行环境不完全由用户控制，因而适合通过 SGX 进行保护。

根据如上特点，经过长期探索，我们找到了适用于 SGX 的理想应用场景：API 密钥保护。经过相关文献调研，发现 SGX 应用于 API 密钥保护的研究尚未成熟，因而我们将此领域选为最终的研究目标。

4 SGX API 密钥保护问题陈述

在本章，我们首先正式描述 AKGuard 系统及其他相关的系统参与者。之后我们将给出具体的 AKGuard 安全模型和设计目标。

4.1 系统模型

我们的系统包含三个部分：应用程序开发者，应用程序服务器和云服务提供商，如图 4.1 所示：

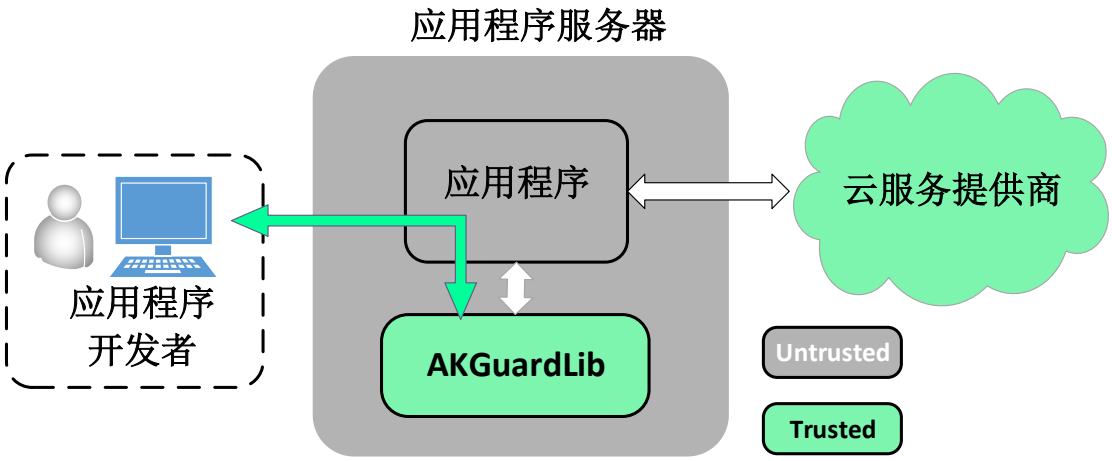


图 4.1 AKGuard 系统框架

应用程序开发者：应用程序开发者拥有从云服务提供商申请获得的 API 密钥，这些 API 密钥用于其程序进行 API 请求身份认证过程。应用程序开发者需要在应用程序开始生成 API 请求前将 API 密钥提供给 AKGuardLib，AKGuard 系统的核心部分，用于执行 API 密钥的存储和签名。我们将在后续章节详细介绍应用程序开发者提供 API 密钥的过程。

应用程序服务器： 应用程序服务器是支持开发者的应用程序运行的服务器，AKGuardLib 也同样运行在此服务器上。在这里，AKGuardLib 作为第三方动态库为应用程序服务器上的应用程序提供服务。该服务器需要配备支持 Intel SGX 功能的 CPU。此外，如果满足此条件，该服务器也可以是一台云服务器。我们在 SGX 安全区中实现了 AKGuardLib。我们将在后续章节详细介绍 AKGuardLib 的内部构造。

云服务提供商：云服务提供商通过 web API 向应用程序开发者提供云服务。云服务提供商在这里也可以是通过 web API 提供网络服务的网站或其他应用程序。除此之外，云服务提供商需要支持基于 API 密钥的身份认证机制。

4.2 安全模型

在本节，我们简要概述 AKguard 系统的安全模型以及相关的系统假设：

（1） 云服务提供商安全性。我们的系统暂不考虑云服务提供商的安全性。我们假设云服务提供商一侧是安全的。这对于使用云 API 的开发者而言是最基础的安全前提。云服务提供商有能力保障云端必要认证数据的安全性，这意味着其不会泄露自己用户的 API 密钥。合法的 API 请求一旦发送到云服务提供商一侧，我们就认为请求被正常处理。非法的 API 请求（如没有正确签名字段的 API 请求）则会被服务端直接拒绝。

（2） 安全区安全性。我们假设 Intel SGX 是安全的并且不会被攻击者以任何方式攻破。安全区内的数据和代码对其他程序而言是不可访问的。本文暂不考虑针对 SGX 安全区的旁信道攻击[41], [42], [43], [44], [45]。在本文中，我们假设受到 SGX 保护的可信函数总是能正常执行并返回预期结果。

（3） 客户端安全性。尽管生成 API 请求的应用程序可能运行在一个不可信的操作系统上，我们仍需要假设应用程序开发者有一个可信的客户端设备（如开发者的个人计算机）并完全掌控在自己手中，作为向 AKGuardLib 提供机密信息的信息源。这个客户端设备可能并不适合运行开发者的应用程序（如需要长时间运行的后端程序，或需

要占用庞大资源的程序)。然而，这个客户端设备适合用来从云服务提供商处获取 API 密钥并提供给 AKGuardLib。

（4） 系统环境。为了运行我们的系统，支持 SGX 的 CPU 是必要条件。应用程序服务器的操作系统以及网络环境可以是不可信的（如被攻击者控制)。然而，本文不考虑应用程序服务器的物理性硬件损坏(如暴力破坏）和意外事故（如火灾)。

4.3 设计目标

在本节我们给出 AKGuard 的如下预期特性：

（1） 抗攻击性。本系统中的所有数据流和机密信息相关操作均保障安全。即使是具有系统权限的攻击者（如恶意操作系统）和不可信网络环境也不会导致 API 密钥的泄露或恶意篡改。我们的系统不会引入新的待保护机密信息。此外，攻击者无法利用我们的系统来认证自己生成的恶意 API 请求。

（2） 高扩展性：我们的系统不是为某特定使用场景或程序设计的，而是为普适的基于API密钥的身份认证场景设计的。由于大多数基于API密钥的身份认证过程比较相似，我们有能力为尽可能多的需要访问云服务的应用程序提供服务。此外，我们系统的存储能力不会成为实际使用过程中的限制。

（3） 开发者友好。应用程序开发者只需对已有程序进行小幅修改即可使用我们的程序来为 API 密钥提供保护，无需将应用程序以某种特定编程语言重写或大幅重构，系统学习成本几乎为零。我们的系统也不依赖任何已有的密钥管理系统。

（4） 高性能。我们系统的性能损耗是可接受的（低于网络 IO 时间的 10%)。即使是在并发条件下，应用程序的最大 API 请求吞吐量同样不会受到明显影响（性能损失小于 10%)。

5 AKGuard 系统设计

在本章，我们首先给出 AKGuardLib 的系统设计；第二，我们聚焦于系统架构中各个组件之间的数据流；第三，我们将分析针对 AKGuard 系统的潜在攻击手法与应对策略；最后，我们给出使用我们系统的具体指导方法。

5.1 AKGuardLib系统设计

我们将 AKGuardLib 的核心模块实现在了 Intel SGX 安全区内，包括远程认证模块、可信时间模块和签名模块。AKGuardLib 的系统设计图如图 5.1 所示：

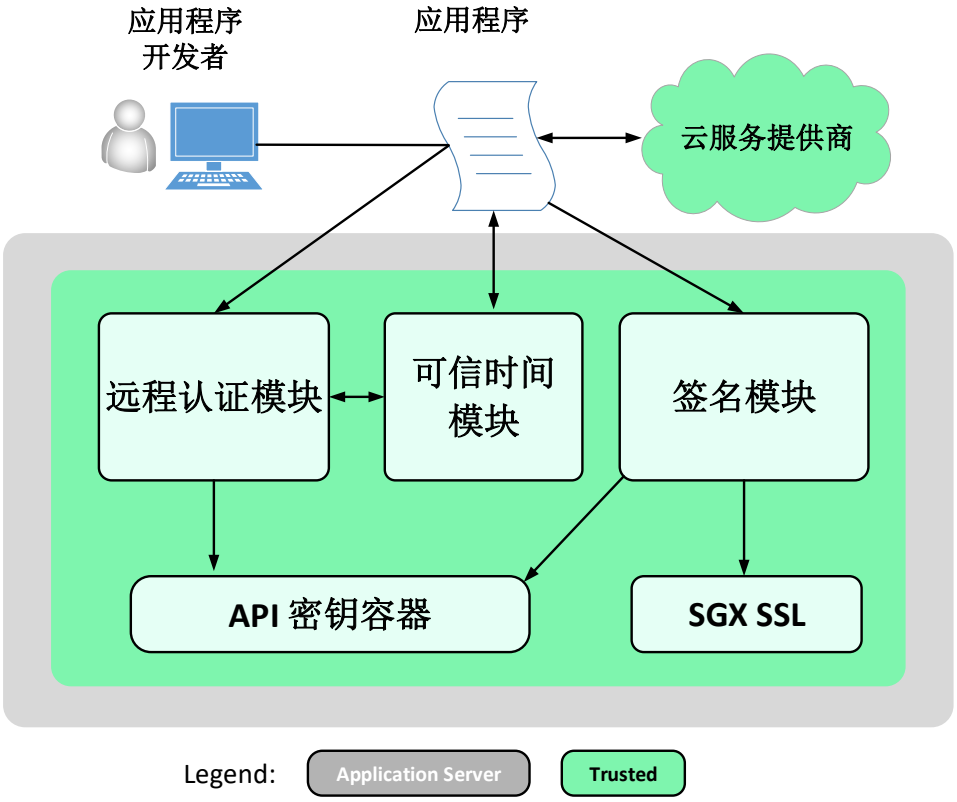


图 5.1 AKGuardLib 系统设计

5.1.1 可信时间模块

在大多数情况下，时间信息（如时间戳）都是构建 API 请求认证所必须的信息之一。通常，时间戳也是计算签名信息的必要信息（如 TOTP）。通过这种策略获取的签名可以抵御重放攻击。可信时间模块为 SGX 安全区之外的应用程序提供了精确到秒级的可信时间戳。此精确度对绝大多数应用程序均已足够。

因而，应用程序不必从应用程序服务器的操作系统来获取时间，规避了系统时间被攻击者恶意篡改的风险。

可信时间模块使用系统安全区（AE）服务来获取可信时间。系统安全区是SGX 框架的一部分，在获取可信时间戳的过程中同样不需要执行系统调用。然而，在实际测试过程中，我们发现：通过这种方式获取的时间戳和真实的系统时间并不一致（在我们的 SGX 设备上获取的可信时间比真实系统时间慢 12685700 秒，即使机器睡眠或重启，此时间差仍保持不变）。因此，我们需要在该模块返回可信时间前进行一些调整工作（在我们的 SGX 设备上指在返回前加上 12685700 秒）。由于此信息非机密信息， 我们可以在 AKGuardLib 的源代码中直接进行调整，或者在应用程序开发者向 AKGuardLib 提供 API 密钥的同时顺带提供时间差值。

4.1.2 签名模块

签名模块负责为应用程序生成的 API 请求计算合法的签名字段。其中所需的 API 密钥获取自 SGX 安全区内的 API Key Container 而非从其他系统位置以明文方式获取。我们在该模块中实现了三个签名算法以适应主流云服务提供商的身份认证策略。

最常见的方法是直接通过 API 密钥和一段预先构造字符串计算哈希消息认证码作为签名，如图 5.2 所示：

Algorithm 1 HAMC based signature-a **(inside an enclave)**

Input: accesskey, type, str, encode_method
Output: signature

1: **function** GETSIG($accesskey, type, str, encode\_method$)
2: $secretkey \leftarrow$ GETSECRETKEY($accesskey$)
3: $hmac \leftarrow$ HMAC($type, secretkey, str$)
4: $signature \leftarrow$ ENCODE($encode\_method, hmac$)
5: **return** $\langle signature \rangle$
6: **end function**

图5.2 签名算法1

该算法的输入参数 accesskey 是 API 密钥的第一个元素，代表了用户身份，且不含有任何需要保护的机密信息，可以由应用程序直接提供。输入参数 type 代表使用的哈希消息认证码生成算法所使用的哈希算法（如 SHA256）。输入参数 str 是一段不含机密信息的待签名字段，该字段的内容生成方式由各云服务提供商指定，并由客户应用程序自行生成字段内容。输入参数 encode_method 代表签名的编码格式（如 base64 编码、十六进制编码）。在该算法中，我们首先根据 accesskey 的值从 SGX 安全区内的 API Key Container 中获取相应的 secretkey（第 2 行）。之后签名模块根据哈希消息认证码的哈希算法类型、secretkey 和待签名字段的值计算哈希消息认证码（第 3 行）。最后，我们将哈希消息认证码编码为云服务提供商指定的格式并输出，作为最终的签名（第 4-5 行）。此签名方案可应用于阿里云、腾讯云和微软 Azure 等。

为了进一步提高安全性，一些云服务提供商（如亚马逊 AWS、百度云）通过一个临时生成的签名密钥，而非一成不变的 secretkey 来计算哈希消息认证码作为签名。我们通过如下算法处理此种签名计算过程：

Algorithm 2 HAMC based signature-b **(inside an enclave)**

Input: accesskey, sign_key_info, type, str, encode_method
Output: signature

```
1:  function GetSigKey(accesskey, sign_key_info, type)
2:      secretkey ← GetSecretKey(accesskey)
3:      for sign_key_info_str in sign_key_info do
4:          if hmac is None then
5:              Let key ← secretkey
6:          else
7:              Let key ← hmac
8:          end if
9:          hmac ← HMAC(type, key, sign_key_info_str)
10:     end for
11:     signingkey ← hmac
12:     return ⟨signingkey⟩
13: end function
14:
15: function GetSig(signingkey, type, str, encode_method)
16:     hmac ← HMAC(type, signingkey, str)
17:     signature ← Encode(encode_method, hmac)
18:     return ⟨signature⟩
19: end function
```

图5.3 签名算法2

在此算法中，相比签名算法 1，我们添加了一个新的输入参数：sign_key_info。此参数包含用于计算签名密钥所必须的字符串（组），且不含任何机密信息。在百度云中，sign_key_info 仅包含一段字符串：bce-auth-v1/{accesskey}/{timestamp}/{expirationPeriodInSeconds}；而在亚马逊 AWS 中则包含 4 段字符串：dataStamp，regionName，serviceName，“aws4 request”。前三者在计算时替换为云服务对应的日期、地区和服务名称。在签名算法 2 的函数 GetSigKey 中，我们首先以和签名算法 1 相同的方式获取 secretkey 值（第 2 行）。接下来，我们需要通过循环计算来获取签名密钥 signingkey（第 3-10 行）。图 5.4 显示了一个 GetSigKey 函数计算示例，其中的 sign_key_info 包含了 4 个字符串。sign_key_info 中的字符串用于计算哈希消息认证码，并作为待签名字段。在第一次计算中，哈希消息认证码算法的密钥信息输入为 secretkey，此后每次计算得到的哈希消息认证码又作为下一步计算的密钥输入信息，与循环神经网络的计算过程类似。循环次数与 sign_key_info 包含的字符串数目相同。最后得到的哈希消息认证码作为签名密钥被输出（第 11-12 行）。签名算法 2 剩余的输入参数和计算过程与签名算法 1 一致（第 15-19 行）。

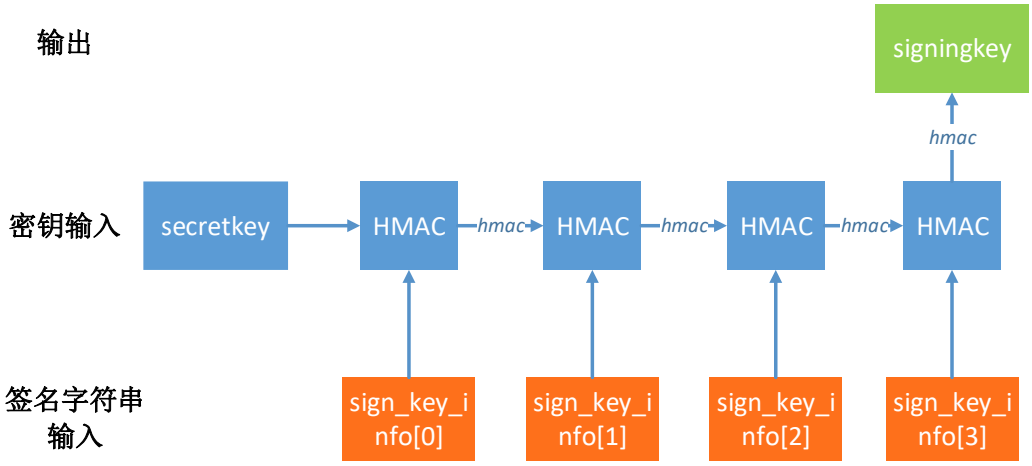


图 5.4 GetSigKey 函数计算示例

除此之外，少数云服务提供商支持在客户身份认证过程中使用标准的 RSA 签名算法（如谷歌云）。在这种情况下，云服务提供商在签名检查过程中便无需使用用户的机密信息（用户公钥对签名检查而言已经足够）。因此，我们认为此种签名方法更加安全可靠，虽然其性能有所下降。该签名算法如图 5.5 所示：

Algorithm 3 RSA based signature **(inside an enclave)**

Input: keyID, type, str, encode_method
Output: signature

1: **function** GETSIG($keyID, type, str, encode\_method$)
2: $privatekey \leftarrow$ GETPRIVATEKEY($keyID$)
3: $sig \leftarrow$ RSASIGN($type, privatekey, str$)
4: $signature \leftarrow$ ENCODE($encode\_method, sig$)
5: **return** $\langle signature \rangle$
6: **end function**

图5.5 签名算法3

首先我们通过 KeyID 参数在安全区内获取 RSA 签名私钥（第 2 行），之后我们根据 RSA 签名私钥、摘要类型（如 SHA256）和待签名字符串计算 RSA 签名（第 3 行）。最后，我们将签名编码为目标格式并输出（第 4 行）。

签名模块中使用的密码学函数由 SGX SSL 支持。

5.1.3 远程认证模块

远程认证模块为应用程序提供 SGX 远程认证接口，进而可使 AKGuardLib 可以证明：

（1） AKGuardLib 自身的身份。

（2） AKGuardLib 未被篡改。

（3） AKGuardLib 运行在一个支持 Intel SGX 的服务器上

之后，应用程序开发者可以通过一条可信信道向 AKGuardLib 提供 API 密钥。应用程序便可利用 AKGuardLib 安全生成签名信息。应用程序开发者和远程认证模块的数据流将在下一节进行分析。

5.2 数据流分析

在本节，我们将分析 AKGuard 中的数据流，以及为何这些数据流是安全的。图 4.6 显示了 AKGuard 中的数据流分布。这些数据流可以被分为三部分：应用程序开发者和 AKGuardLib 之间的数据流（①，②），应用程序和 AKGuardLib 之间的数据流（③，④），以及应用程序和云服务提供商之间的数据流（⑤，⑥）。

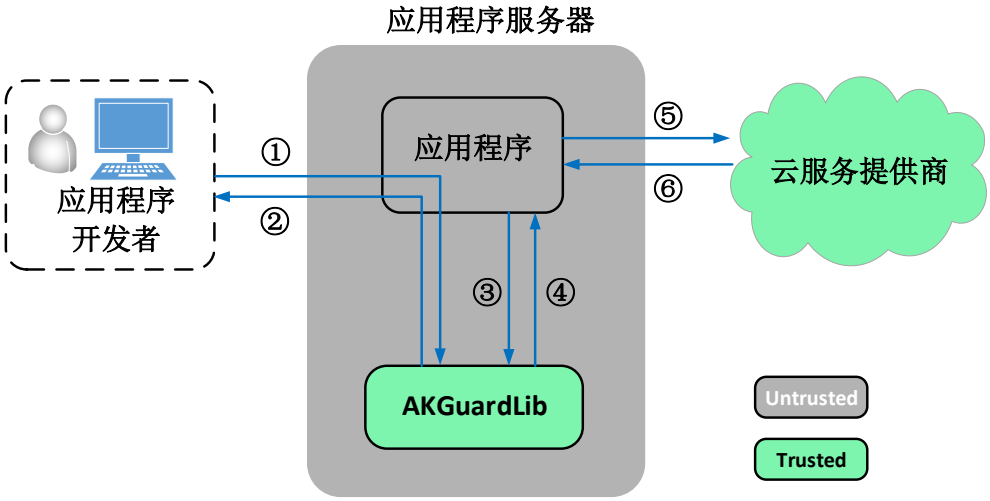


图5.6 AKGuard数据流图

应用程序开发者和 AKGuardLib 之间的数据流通过 SGX 远程认证实施。由于 AKGuardLib 以库的形式为应用程序提供服务，此阶段的数据流需要流经应用程序一端。在这里，信息 ① 和信息 ② 代表用于在应用程序开发者和 AKGuardLib 之间实施 Diffie-Hellman 密钥交换协议而经过调整的 Sigma 协议。通过该协议获取的共享密钥用于加密二者之间的通信信道，进而 API 密钥可以以密文形式安全地传输给 AKGuardLib。同时，AKGuardLib 可以利用共享密钥解密密文获取 API 密钥。可选地，信息 ① 可以包含额外的时间信息，用于调整可信时间模块生成的可信时间戳。由于时间信息并非需要保护的机密信息，开发者同样可以在 AKGuardLib 的源代码中进行调整。

在应用程序和 AKGuardLib 之间的数据流中，信息 ③ 代表应用程序发送给 AKGuardLib 的签名请求信，以及附带的必要计算参数。应用程序根据实际使用的云服务调用 5.1.2 节所述的签名算法函数。信息 ④ 代表 AKGuardLib 计算返回的签名信息，用于应用程序构建完整的 API 请求。值得注意的是，信息 ③ 和信息 ④ 均不含任何机密信息。信息 ④ 中含有的签名信息只可以用于认证应用程序构建的特定 API 请求。机密的 API 密钥信息在 AKGuardLib 内存储与使用。因此，应用程序和 AKGuardLib 之间的数据流是安全的，虽然这部分数据流有可能泄露给隐藏在应用程序服务器中的攻击者。

信息 ⑤ 表示由应用程序构造的拥有合法签名信息的 API 请求。签名信息可以被添加到请求的 header 字段、body 字段或目标 URL 中，具体的添加位置

取决于应用程序自身。正如信息 ④ 一样，信息 ⑤ 也同样没有泄露任何机密信息。信息 ⑥ 表示云服务提供商的返回信息。我们建议应用程序不要发送有敏感信息返回值的 API 请求。幸运的是，有大量符合此条件的 API 请求（如云端数据更新、数据计算以及对非敏感信息的查询）。

5.3 潜在攻击场景分析

尽管基于上一节的分析，我们系统中的数据流可以保障安全性，但仍不排除有关于 AKGuard 在其他攻击场景下是否安全的质疑。在本节，我们分析两类针对我们系统的可能攻击场景，并进一步展示 AKGuard 在此类场景下的可靠性。

5.3.1 应用程序伪装

作为第三方服务，AKGuard 可为应用程序服务器上的多种应用程序提供服务。由此引出如下问题：攻击者如果将自己的恶意程序伪装成合法应用程序来访问 AKGuard，并要求其为自己构建的恶意 API 请求签名，那么 AKGuard 是否能够识别？由于动态库在不同进程之间不共享数据，攻击者通过直接访问签名服务仅能得到一个空的 AKGuardLib，没有任何 API 密钥存储其中。不过攻击者仍有机会在开发者向 AKGuardLib 提供 API 密钥之前将开发者自己的应用程序替换为自己的恶意程序。对于此种类型的攻击，应用程序开发者只需在提供 API 密钥之前检查目标应用程序的摘要（如 SHA256）即可令攻击失效。因此，通过应用程序伪装，攻击者无法发动有效的攻击。

5.3.2 签名重放

在我们的安全假设中，AKGuardLib 提供的签名信息被认为是透明的。攻击者可以获取其他程序请求的签名信息，但无法用此签名令自己构造的恶意请求有效。但在某些场景中，一个签名可以在自己过期之前通过一条 URL 允许任何人访问应用程序开发者的云端数据。通常情况下，出于安全目的，签名的有效时间非常短暂（如数秒之内）。然而，拥有系统权限的攻击者可以通过向应用程序提供篡改过的系统时间来令这种保护机制失效。为了防止此类攻击，应用程序可以从 AKGuardLib 中获取可信时间戳，并用此时间戳设置签名的过期时间。

5.4 使用指导

为了利用 SGX 远程认证机制，应用程序开发者需要预先获取签名证书并在 Intel 开发者服务中进行注册。开发者的公钥应当被硬编码到 AKGuardLib 的安全区中。与安全区签名机制相结合，就可以保障 AKGuard 只与指定的远程客户端（应用程序开发者）进行交互。

应用程序也可以在一开始的某个安全的时间从系统环境（环境变量、文件等）中读入 API 密钥并将其传入 AKGuardLib。之后，应用程序删除程序内存中的 API 密钥和存储在系统环境中的 API 密钥即可。虽然此种方法可以免去 SGX 远程认证过程，但要找到一个“安全的时间”并不容易。因此，我们不建议通过这种方式向 AKGuardLib 提供机密信息。考虑到机密信息提供过程并非一个十分频繁的过程（每次应用程序开发者的 API 密钥更新时），这值得一个更加安全的实践过程。

为了达到更高等级的安全性，开发者可以将自己的应用程序重写成完全的 SGX 应用程序，并将签名之外的其他机密信息相关函数也一并移到 SGX 安全区内。Lind 等人提出了一个名为 Glamdring[46]的工具来帮助开发者自动划分 C 语言应用程序的可信部分与非可信部分，减轻开发者的负担。这部分内容超出了本文的讨论范围。

使用 AKGuard 的过程可以被总结如下：

（1） 启动应用程序服务器上的目标应用程序。

（2） 在应用程序内初始化 AKGuardLib 安全区。

（3） 应用程序开发者检查目标程序数字摘要。

（4） 应用程序开发者通过远程认证提供 API 密钥。

（5） 目标应用程序构建 API 请求并向 AKGuardLib 发送签名请求。

（6） 目标应用程序通过带有签名的 API 请求访问云服务。

6 实验与分析

6.1 实验设置

我们设计实现了一套 AKGuard 的系统原型，并运行在一台装有 Ubuntu 16.04（64 bit）操作系统的服务器上。服务器配有 Intel(R) Xeon(R) E3-1240 v6 CPU (3.70 GHz)，并有 16GB 内存。用于开发原型系统的 SGX SDK 版本为 1.9。SGX SSL 库基于 openssl-1.1.0e 构建。所有的实验均在这一台服务器上进行。

我们使用 Python 2.7.12 编写了 6 个应用程序，通过 AKGuardLib 访问 6 家主流云服务提供商的云端 API。云服务提供商的详细信息和服务类型如表 6.1 所示：

表 6.1 云服务提供商基本信息

云服务提供商	云服务类型	API 请求类型	签名算法
谷歌云	Cloud Storage	Download Objects	RSA Signature (Algorithm 3)
微软 Azure	Blob Storage	Get Blobs	HMAC-SHA256 (Algorithm 1)
亚马逊 AWS	DynamoDB	Query Tables	HMAC-SHA256 (Algorithm 2)
百度云	Baidu Object Storage (BOS)	GetObject	HMAC-SHA256 (Algorithm 2)
阿里云	Object Storage Service (OSS)	GetObject	HMAC-SHA1 (Algorithm 1)
腾讯云	Tencent MySQL DB	Describe Tables	HMAC-SHA1 (Algorithm 1)

6.2 存储能力测试

我们测试了 AKGuardLib 的存储能力。在实施过程中，我们使用 C++ STL map 实现密钥存储功能。就密钥格式，亚马逊 AWS、百度云、阿里云和腾讯云提供定长字符串作为 API 密钥的 Secret Key（长度分别为 40 字节、32 字节、30

字节和 32 字节）。在我们实施实验时，这些是最常见的 API 密钥类型。Azure 提供的是 base64 编码后的字符串作为 API 密钥（长度为 88 字节）。其解码后的 API 密钥不可显示为人类可读的格式。谷歌云提供的是 2048 位长度的标准 RSA 密钥，用于在客户端身份认证过程中生成标准的 RSA 签名。我们以字符串形式直接存储密钥，其占用大小为 1704 字节。

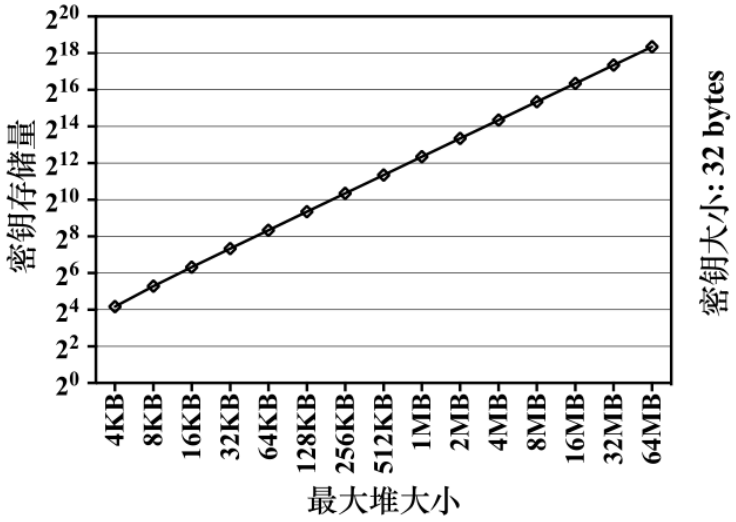


图6.1 密钥存储能力 vs 安全区堆大小

我们首先测试了 AKGuardLib 的密钥存储能力与 AKGuardLib 安全区最大堆大小之间的关系，如图 6.1 所示。测试使用了长度为 32 字节的密钥。由图可知，AKGuardLib 的密钥存储能力与其安全区堆大小基本呈线性关系，随着安全区堆大小的增加，密钥存储能力也基本线性增长。虽然此时 AKGuardLib 安全区的空间利用率并不高（存储 32 位密钥空间利用率约为 16%），AKGuardLib 仍可存储最多达到超过 300000 个 API 密钥（64MB 堆大小情况下，Intel SGX 对安全区大小有一个硬性限制，一般为 64MB 或 128MB[47]）。

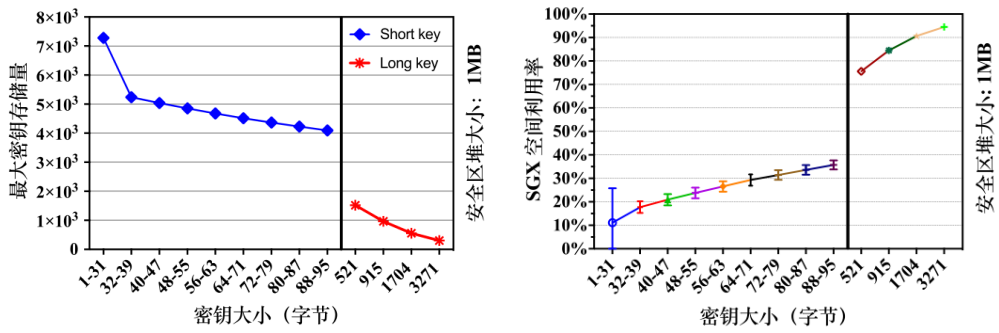


图6.2 密钥存储能力 vs 密钥大小（左）

SGX安全区空间利用率 vs 密钥大小（右）

针对其他长度密钥的测试结果如图 6.2 所示。其中，左图显示了 AKGuardLib 密钥存储能力与密钥大小之间的关系，其中最大堆大小设置为 1MB。实验结果符合预期，即 AKGuardLib 密钥存储数量随着密钥长度的增加而下降。然而，右图则显示出，AKGuardLib 的安全区空间利用率虽密钥长度的增加而上升。究其原因，密钥以红黑树节点的形式存储，在存储小密钥的过程中还产生了大量树结构，占据了存储空间，降低了空间利用率。图 6.2 的最后三组密钥长度分别对应标准 512 位、1024 位、2048 位和 4096 位 RSA 密钥以字符串存储的实际长度。应用开发者可以根据图 6.1 和图 6.2 的测试结果估计自己应用程序所需的 AKGuardLib 安全区堆大小并合理分配，以节约宝贵的安全区堆空间。

6.3 基础能力测试

针对 6.1 节提到的 6 个应用程序，我们分别运行了 300 次，并记录了相关的统计数据，包括：平均运行时间、运行阶段时间占比、最长运行时间、最短运行时间、以及标准差，如表 6.2 所示。所有时间数据单位均为毫秒。获取可信时间（Get Trusted Time）和签名（Sign）操作均在 AKGuardLib 的安全区内进行，其余操作在应用程序端进行。

表 5.2 AKGuard 基础能力测试

Time (ms) / Stages	Google Cloud					Amazon AWS					Microsoft Azure				
	mean	%	t_{max}	t_{min}	σ_t	**mean**	%	t_{max}	t_{min}	σ_t	**mean**	%	t_{max}	t_{min}	σ_t
Get Trusted Time	19.55	7.06	24.66	6.94	1.40	20.07	9.27	21.10	18.98	0.37	20.00	9.27	21.09	16.22	0.52
Sign	4.168	1.51	5.604	1.224	0.972	0.142	0.07	0.218	0.126	0.01	0.129	0.06	0.203	0.054	0.02
Web IO	253.00	91.36	1251.66	197.09	67.28	195.44	90.27	249.15	156.10	17.03	194.92	90.32	425.49	164.02	17.67
Other	0.189	0.07	0.289	0.020	0.04	0.860	0.39	1.129	0.137	0.09	0.758	0.35	0.934	0.067	0.12
Total	277	100	1276	210	67	216	100	270	177	17	216	100	446	186	18

Time (ms) / Stages	Baidu Cloud					Alibaba Cloud					Tencent Cloud				
	mean	%	t_{max}	t_{min}	σ_t	**mean**	%	t_{max}	t_{min}	σ_t	**mean**	%	t_{max}	t_{min}	σ_t
Get Trusted Time	20.05	5.02	21.02	19.23	0.35	17.30	8.34	29.58	15.79	1.54	20.02	4.57	27.38	16.25	0.77
Sign	0.133	0.03	0.198	0.128	0.01	0.06	0.03	0.165	0.042	0.02	0.111	0.03	0.182	0.043	0.01
Web IO	377.98	94.74	2394.19	297.58	169.43	190.19	91.62	1788.43	113.64	159.74	416.81	95.23	3445.53	352.47	202.14
Other	0.802	0.21	0.981	0.079	0.10	0.029	0.01	0.102	0.018	0.02	0.737	0.17	0.874	0.055	0.11
Total	399	100	2415	319	169	208	100	1806	130	160	438	100	3467	373	202

对于主流云服务提供商的支持显示出了 AKGuard 的应用普适性。由表 6.2 可知，应用程序的运行时间在 208 毫秒到 438 毫秒之间。除此之外，对于全部

测试应用程序，网络延时（Web IO）均是执行 API 请求的主要时间消耗。只有不到 10%的响应时间花费在 AKGuardLib 执行的操作上。图 6.3 对此六个应用程序的时间消耗构成进行了横向对比，进一步印证了我们的结论。

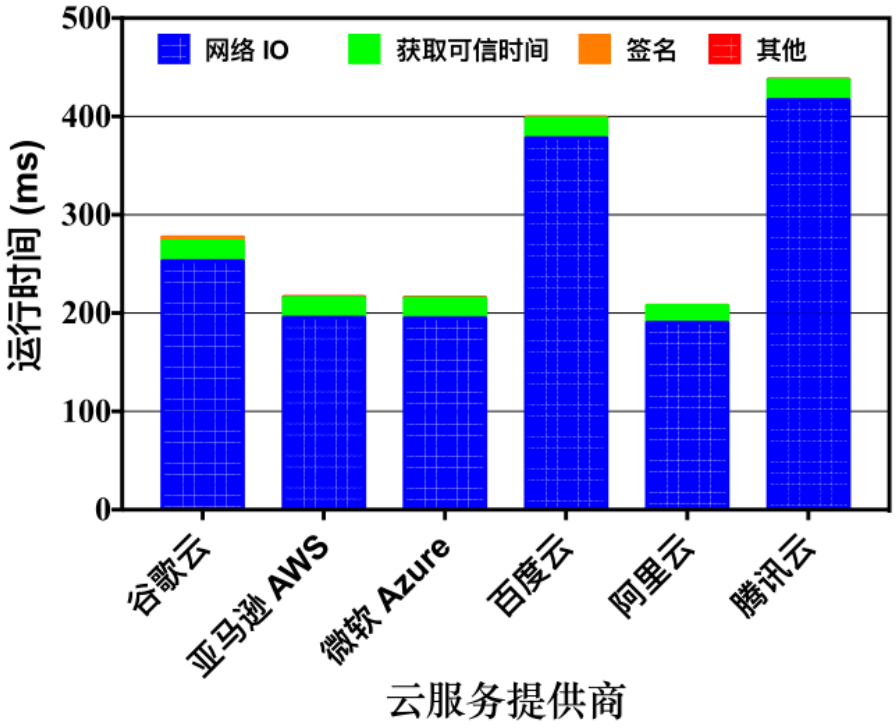
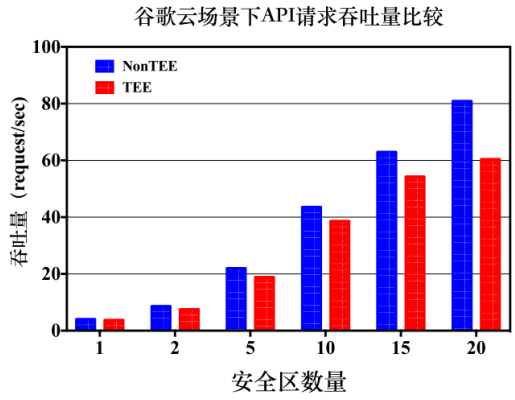


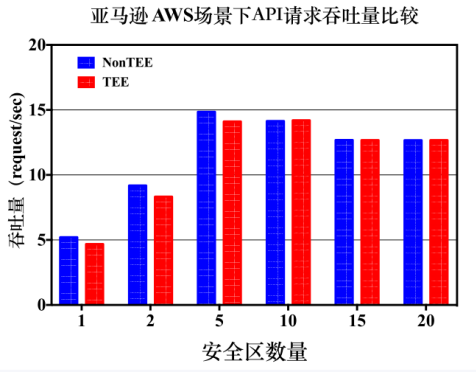
图 6.3 API 请求时间消耗构成对比

6.4 并发能力测试

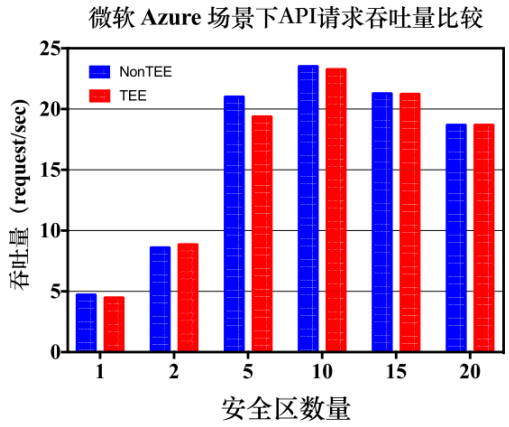
实验的最后部分是 AKGuard 在并发访问下的性能表现测试。对于网络应用程序，并发是重要的性能提升手段。我们在单机上分别测试了 AKGuard 在 1 到 20 进程并发访问下的性能，每个进程对应一个独立的 SGX 安全区。测试结果如图 6.4 所示。对于每个应用程序，我们分别在其使用 AKGuard 和不使用 AKGuard 的情况下进行了 10 轮实验；每轮实验，我们令应用程序分别以 1、2、5、10、15、20 的并发度执行 API 请求，每个并发度下并行发起共计 1000 个 API 请求。图中的红色柱状图代表使用 AKGuard 进行 API 密钥管理的实验吞吐量结果，蓝色柱状图代表不使用 AKGuard 的实验吞吐量结果（直接使用程序代码中硬编码的 API 密钥）。



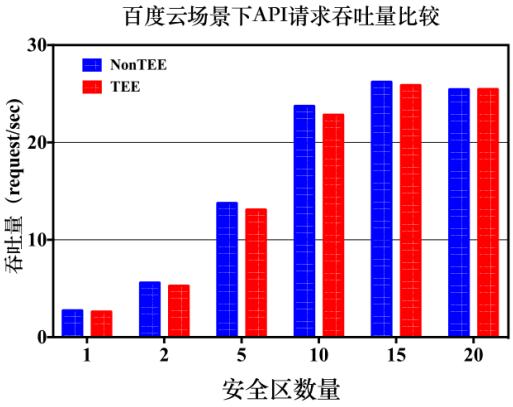
（a）谷歌云



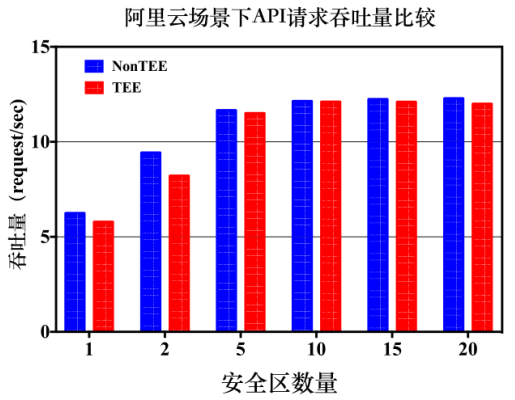
（b）亚马逊 AWS



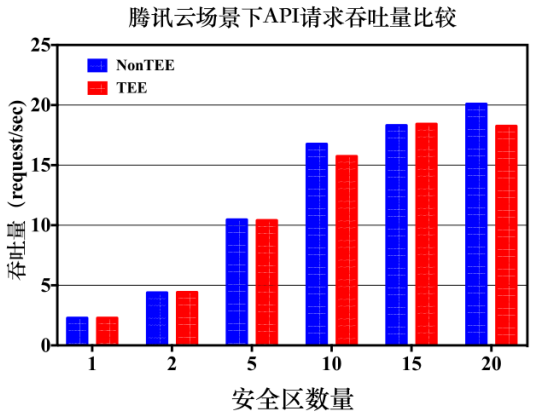
（c）微软 Azure



（d）百度云



（e）阿里云



（f）腾讯云

图 6.4 单 SGX 设备并发请求吞吐量

单 SGX 设备每秒可以处理 12 到 60 云 API 请求，具体的吞吐量取决于应用程序所访问的云服务以及对应的云服务提供商。实验结果表明，AKGuard 在并发访问的情景下不会明显影响应用程序的 API 访问吞吐量。对于访问谷歌 Cloud

Storage 的 API，性能损失最大，约为 25%。其余云服务的最大 API 请求访问吞吐量几乎和不使用 AKGuardLib 的情况相当（性能损失在 1%到 9%之间）。

6.5 程序不足

由于 AKGuard 的运行需要有支持 Intel SGX 功能的 CPU，我们无法为移动应用程序提供 API 密钥保护服务。幸运的是，仍然有相当规模的应用程序可以使用我们的系统（如 web 服务器后端程序、量化交易程序等）。其中很多应用程序都需要频繁而安全地访问 web API。一些已有的工作研究了移动设备上的 web API 安全解决方案[48], [49], [50]，此领域的研究不在本文的讨论范围之内。

7 未来工作展望

我们计划在 AKGuard 的第一版原型系统的基础上再进行一些改进工作。未来的改进计划如下：

7.1 系统恢复

在我们之前的分析中，我们假设 AKGuardLib 所在的操作环境是稳定的（不考虑意外事故的发生），尽管操作系统可以是不可信的。在现实世界中，这个假设可能有些牵强。例如，断电之类的突发事故可以轻易将 AKGuardLib 的安全区销毁，令其中的所有数据丢失。尽管这并不会造成关键数据的泄露，但重新初始化 AKGuardLib 的安全区并导入密钥会带来不必要的麻烦。为了应对此类事故，我们计划引入 SGX Sealing 机制来设计 AKGuardLib 的故障恢复系统。

7.2 支持Windows操作系统

在我们的实验中，AKGuard 仅仅可以为 Linux 操作系统上的应用程序提供服务。Intel 已经公开了用于 Windows 10 操作系统的 SGX 开发工具，并且已经集成到了 Visual Studio 2015 及其更高版本。考虑到 Windows 系统的流行性，我们计划构建一个可支持 Windows 10 应用程序的 AKGuard 系统。之后我们将在

Windows 10 上进行相关的实验并分析其性能。目前苹果设备还不支持在其 BIOS 中开启 SGX 支持（尽管其 CPU 是支持 SGX 功能的），我们计划在其支持 SGX 功能后设计用于 Apple Mac 操作系统的 AKGuard。

7.3 通过移动设备提供API密钥

如前文所述，AKGuard 使用 SGX 远程认证技术来实现机密信息导入。在我们的实现中，我们使用一个 Linux 应用程序来代表应用程序开发者向 AKGuardLib 提供 API 密钥。SGX SDK[51]提供了整个认证流的封装，包括所需的密码学支持。然而，目前仅有对于 Windows 操作系统和 Linux 操作系统的 SGX SDK。在很多情况下，使用移动设备完成机密信息导入更加灵活方便。但这就意味着， 我们需要在不借助 SGX SDK 的情况下设计实现整个远程认证流。除此之外，这个用于提供 API 密钥的移动应用程序也需要根据开发者移动设备的操作系统（如 Android，IOS）分别实现。

7.4 更高的安全性和执行性能

在我们目前的 AKGuard 实现中，我们还没有采用任何高级工具或技术来提高程序的安全性或执行性能。目前有很多致力于提高 SGX 应用程序安全性和效率的研究工作，供我们借鉴。就提高安全性而言，Seo 等人公开了 SGX-Shield[52]来为 SGX 应用程序提供地址空间布局随机化（ASLR）的特性。Chen 等人设计了一个基于 LLVM 的工具，名为 HyperRace[53]，它可以根除由超线程带来的针对 SGX 应用程序的旁信道攻击。Fulkerson 等人提出的 ZeroTrace[54]将最新的 Oblivious RAM（ORAM）技术与 Intel SGX 技术结合，将二者的取长补短，安全性进一步提升。就提高 SGX 程序性能而言，Weisse 等人进行了一项 SGX 性能综合定量评价研究，并设计了 SGX 接口框架 HotCalls[55]，其可以在很大程度上减小应用程序延迟。Meysam 等人提出了 VAULT[56]来减少 SGX 应用程序中的换页负载。我们相信这些技术对于我们进一步优化 AKGuard 大有裨益。

8 总结

我们设计了一个第三方服务系统 AKGuard 来帮助应用程序管理 API 密钥。该系统克服了我们先前 SGX 数据保护研究工作中的不足，较好地发挥了 SGX 的技术优势。所有机密数据相关的操作和信息均纳入 Intel SGX 硬件的保护。AKGuard 可以应用于多种应用程序,无需开发者重写或大幅修改原有应用程序。AKGuardLib 库可以作为 C 语言动态库应用于 Linux 操作系统上的应用程序。除此之外，AKGuard 可以适配多种不同云服务提供商的不同客户端身份认证方法。安全性分析和实验结果也显示了我们系统的实用性。我们相信 AKGuard 为解决 API 密钥管理问题提供了一个行之有效的解决方案。

参考文献

[1] 谷歌云. https://cloud.google.com. 最近访问于 2019 年 5 月.

[2] 亚马逊网络服务（AWS）. https://aws.amazon.com. 最近访问于 2019 年 5 月.

[3] 微软 Azure. https://azure.microsoft.com. 最近访问于 2019 年 5 月.

[4] 阿里云. https://www.alibabacloud.com. 最近访问于 2019 年 5 月.

[5] 百度智能云. https://cloud.baidu.com. 最近访问于 2019 年 5 月.

[6] 腾讯云. https://intl.cloud.tencent.com. 最近访问于 2019 年 5 月.

[7] “Bitfinex - bitcoin, litecoin and ethereum exchange and margin trading platform.” https://www.bitfinex.com. 最近访问于 2019 年 5 月.

[8] “Bitstamp - buy and sell bitcoin.” https://www.bitstamp.net. 最近访问于 2019 年 5 月.

[9] “Binance - exchange the world.” https://www.binance.com. 最近访问于 2019 年 5 月.

[10] “Gekko - open source bitcoin trading bot platform.” https://gekko.wizb.it 最近访问于 2019 年 5 月.

[11] “Blackbird bitcoin arbitrage.” https://github.com/butor/blackbird. 最近访问于 2019 年 5 月.

[12] Hammer-Lahav E. The oauth 1.0 protocol[R]. 2010.

[13] Hardt D. The OAuth 2.0 authorization framework[R]. 2012.

[14] Google Cloud Natural Language API Documents. https://cloud.google.com/natural-language/docs/. 2019

[15] “Thousands of apps found with hardcoded api creds.” https://www.itnews.com.au/news/thousands-of-apps-found-with-hardcoded-api-creds-447877. 最近访问于 2019 年 5 月.

[16] “An issue of django-rest-framework: Api keys stored in clear text.” https://github.com/encode/django-rest-framework/issues/4227. 最近访问于 2019 年 5 月.

[17] “Binance cryptocurrency exchange hacked from trading bot via api keys.” https://bitcoinexchangeguide.com/binance-cryptocurrency-exchange-hacked-from-trading-bot-via-api-keys. 最近访问于 2019 年 5 月.

[18] “A thorough investigation of the binance hack.” https://www.hodlbot.io/blog/a-thorough-investigation-of-the-binance-hack. 最近访问于 2019 年 5 月.

[19] F. Mckeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, “Innovative instructions and software model for isolated execution,” in Proceedings of International Workshop on Hardware and Architectural Support for Security and Privacy (HASP), 2013, pp. 1–1

[20] “Google map api key best practices,” https://developers.google.com/maps/api-key-best-practices. 最近访问于 2019 年 5 月.

[21] “Dexguard,” https://www.guardsquare.com/en/products/dexguard. 最近访问于 2019 年 5 月.

[22] “Proguard,” https://www.guardsquare.com/en/products/proguard. 最近访问于 2019 年 5 月.

[23] “Javascript-obfuscator,” https://github.com/javascript-obfuscator/javascript-obfuscator. 最近访问于 2019 年 5 月.

[24] “Reverse engineering obfuscated assemblies.” https://resources.infosecinstitute.com/reverse-engineering-obfuscated-assemblies. 最近访

问于 2019 年 5 月.
[25] “Reverse engineering a javascript obfuscated dropper.” https://resources.infosecinstitute.com/reverse-engineering-javascript-obfuscated-dropper. 最近访问于 2019 年 5 月.
[26] “Openstack barbican.” https://wiki.openstack.org/wiki/Barbican. 最近访问于 2019 年 5 月.
[27] “Approov.” https://www.approov.io. 最近访问于 2019 年 5 月.
[28] H. K. Lu, “Keeping your api keys in a safe,” in Proceedings of 2014 IEEE 7th International Conference on Cloud Computing. IEEE, 2014, pp. 962–965.
[29] “Fortanix self-defending key management service.” https://www.fortanix.com/products/sdkms. 最近访问于 2019 年 5 月.
[30] V. Phegade, J. Schrater, A. Kumar, and A. Kashyap, “Self-defending key management service with intel software guard extensions.”
[31] “Python sdk sample code of fortanix.” https://d2bzqwib4mjc49.cloudfront.net/sdkms-sdk-sample.py. 最近访问于 2019 年 5 月.
[32] S. Chakrabarti, B. Baker, and M. Vij, “Intel sgx enabled key manager service with openstack barbican,” arXiv preprint arXiv:1712.07694, 2017.
[33] Krawczyk H, Bellare M, Canetti R. HMAC: Keyed-hashing for message authentication[R]. 1997.
[34] 百度智能云鉴权认证机制. https://cloud.baidu.com/doc/Reference/AuthenticationMechanism.html#3.20API.E8.AE.A4.E8.AF.81.E4.BC.98.E5.8A.BF. 最近访问于 2019 年 5 月.
[35] D. M’Raihi, J. Rydell, S. Bajaj, S. Machani, and D. Naccache, “Ocra: Oath challenge-response algorithm,” Tech. Rep., 2011.
[36] D. M’Raihi, S. Machani, M. Pei, and J. Rydell, “Totp: Time-based one-time password algorithm,” Tech. Rep., 2011.
[37] Intel SGX Remote Attestation. https://software.intel.com/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example. 最近访问于 2019 年 5 月.
[38] “Intel software guard extensions ssl.” https://github.com/intel/intel-sgx-ssl. 最近访问于 2019 年 5 月.
[39] “The history of the Mt Gox Hack: Bitcoin’s biggest heist.” https://blockonomi.com/mt-gox-hack/. 最近访问于 2019 年 5 月.
[40] “Crypto Exchange Bithumb Hacked for $13 Million in Suspected Insider Job.” https://www.coindesk.com/crypto-exchange-bithumb-hacked-for-13-million-in-suspected-insider-job. 最近访问于 2019 年 5 月.
[41] Y. Xu, W. Cui, and M. Peinado, “Controlled-channel attacks: Deterministic side channels for untrusted operating systems,” in Proceedings of 2015 IEEE Symposium on Security and Privacy (SP 2015), 2015, pp.640–656.
[42] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, “Cache attacks on intel sgx,” in Proceedings of the 10th European Workshop on Systems Security (EuroSec). ACM, 2017, p. 2.
[43] V. Costan and S. Devadas, “Intel sgx explained.” IACR CryptologyePrint Archive, vol. 2016, no. 086, pp. 1–118, 2016.
[44] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, “Inferring fine-grained

control flow inside {SGX} enclaves with branch shadowing," in Proceedings of the 26th {USENIX} Security Symposium ({USENIX} Security 17), 2017, pp. 557–574.

[45] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in Proceedings of the 27th {USENIX} Security Symposium ({USENIX} Security 18), 2018, pp. 991–1008.

[46] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza et al., "Glamdring: Automatic application partitioning for intel {SGX}," in Proceedings of 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17), 2017, pp. 285–298.

[47] Intel Corporation, "Intel sgx sdk developer reference linux 1.9 open source." p. 4, 2017.

[48] A. Mendoza and G. Gu, "Mobile application web api reconnaissance: Web-to-mobile inconsistencies & vulnerabilities," in Proceedings of 2018 IEEE Symposium on Security and Privacy (SP 2018). IEEE, 2018, pp. 756–769.

[49] X. Zhang, Y.-a. Tan, Y. Xue, Q. Zhang, Y. Li, C. Zhang, and J. Zheng, "Cryptographic key protection against frost for mobile devices," Cluster Computing, vol. 20, no. 3, pp. 2393–2402, 2017.

[50] L. Wu, J. Wang, K.-K. R. Choo, and D. He, "Secure key agreement and key protection for mobile device user authentication," IEEE Transactions on Information Forensics and Security, vol. 14, no. 2, pp. 319–330, 2019.

[51] Intel. Corporation, "Intel software guard extensions sdk." https://software.intel.com/en-us/sgx-sdk. 最近访问于 2019 年 5 月.

[52] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs." in Proceedings of 2017 Network and Distributed System Security Symposium (NDSS 2017), 2017.

[53] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, "Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races," in Proceedings of 2018 IEEE Symposium on Security and Privacy (SP 2018). IEEE, 2018, pp. 178–194.

[54] D. R. Fulkerson et al., "Zero-one matrices with zero trace." Pacific Journal of Mathematics, vol. 10, no. 3, pp. 831–836, 1960.

[55] O. Weisse, V. Bertacco, and T. Austin, "Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves," in Proceedings of ACM SIGARCH Computer Architecture News, vol. 45, no. 2. ACM, 2017, pp. 81–93.

[56] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in sgx with efficient integrity verification structures," in Proceedings of ACM SIGPLAN Notices, vol. 53, no. 2. ACM, 2018, pp. 665–678.

作者简历

姓名：刘丁豪　性别：男　民族：汉　出生年月：1997-03-05　籍贯：山西省临汾市

2012.09-2015.06　临汾市第一中学校

2015.09-2019.06　浙江大学攻读学士学位

获奖情况：浙江大学学业一等奖学金，美国数学建模竞赛二等奖

参加项目：基于区块链的个性化教育服务交易与监管技术研究

本科生毕业论文（设计）任务书

一、题目：基于 SGX 的 API 密钥安全保护技术研究

二、指导教师对毕业论文（设计）的进度安排及任务要求：

1. 至少阅读 20 篇以上与“API 密钥保护”、“SGX”相关的文献；

2. 论文应合理概括研究问题的背景与意义；

3. 对论文的个人工作应进行详细叙述，工作量符合要求；

4. 论文的实验设计应思路清晰，准备翔实；

5. 列出相应的未来工作，并进行合理总结；

6. 论文字数达到毕业设计要求，并符合学院论文查重标准。

起讫日期 **2018** 年 **9** 月 **20** 日 至 **2019** 年 **5** 月 **27** 日

指导教师（签名）_____职称_____

三、系或研究所审核意见:

负责人（签名）_____

年　　月　　日

毕 业 论 文（设计） 考 核

一、指导教师对毕业论文（设计）的评语：

指导教师(签名）

年 月 日

二、答辩小组对毕业论文（设计）的答辩评语及总评成绩：

成绩比例	文献综述占（10%）	开题报告占（15%）	外文翻译占（5%）	毕业论文（设计）质量及答辩占（70%）	总评成绩
分值					

答辩小组负责人（签名）

年 月 日