EnclaveZMQ: 基于 SGX 实现 ZMQ 的安全通信

张磊 zhanglei486@126.com

摘要

本文提出 EnclaveZMQ，基于 SGX 实现 ZMQ 的安全通信协议 Curve ZMQ。通过 SGX 保护 CurveZMQ 中的长期私钥和协议过程中的各种临时密钥，在 Enclave 安全区实现 CurveZMQ 的握手协议和通信消息保护。SGX 的内存隔离可以应对白盒环境下对密钥和协议的各种威胁，SGX 还可以用于构建 ZMQ 的 TCB。实验显示基于 SGX 实现 ZMQ 的安全通信，对性能影响很小，可以满足 ZMQ 作为一种高性能安全消息中间件，应用于金融服务、游戏开发、嵌入式系统、科学研究，以及航天系统中。

一、介绍

本文提出 EnclaveZMQ，基于 SGX 实现 ZMQ 的安全通信协议 Curve ZMQ。通过 SGX 保护 CurveZMQ 中的长期私钥和协议过程中的各种临时密钥，在 Enclave 安全区实现 CurveZMQ 的握手协议和通信消息保护。SGX 的内存隔离可以应对白盒环境下对密钥和协议的各种威胁，SGX 还可以用于构建 ZMQ 的 TCB。实验显示基于 SGX 实现 ZMQ 的安全通信，对性能影响很小，可以满足 ZMQ 作为一种高性能安全消息中间件，应用于金融服务、游戏开发、嵌入式系统、科学研究，以及航天系统中。

第二部分，给出 ZMQ、CurveZMQ、SGX 的相关背景知识。

第三部分，对 ZMQ 面临的风险进行分析

第四部分，从系统架构、密钥管理、协议实现及状态管理、算法实现、应用程序与 Enclave 接口、系统流程几个方面对 Enclave 的设计思想进行详细介绍。

第五部分，从性能角度分析 EnclaveZMQ 同样满足 ZMQ 使用于高性能消息通信的设计思想。

第六部分，给出总结性结论和改进建议。

二、背景知识

2.1. ZeroMQ

ZeroMQ(以下 ZeroMQ 简称 ZMQ)是由 i Matix 公司开发的一款开源的消息中间件。最初的设计目标是在股票交易系统中实现极快的数据交换，所以性能是设计的首要考虑因素。ZeroMQ 目前已经在很广泛的范围内获得使用，包括:金融服务、游戏开发、嵌入式系统、科学研究，以及航天系统中。

ZMQ 是一个简单好用的传输层，像框架一样的一个 socket 库，它使得 Socket 编程更加简单、简洁和性能更高。ZMQ 是一个消息处理队列库，可在多个线程、内核和主机盒之间弹性伸缩。

ZMQ 可以通过发布-订阅、任务分发、和请求-回复等模式来建立 N-N 的 Socket 连接。 ZeroMQ 的异步 I / O 模型为我们提供可扩展的基于异步消息处理任务的多核应用程序。 它有一系列语言 API（几乎囊括所有编程语言），并能够在大多数操作系统上运行。

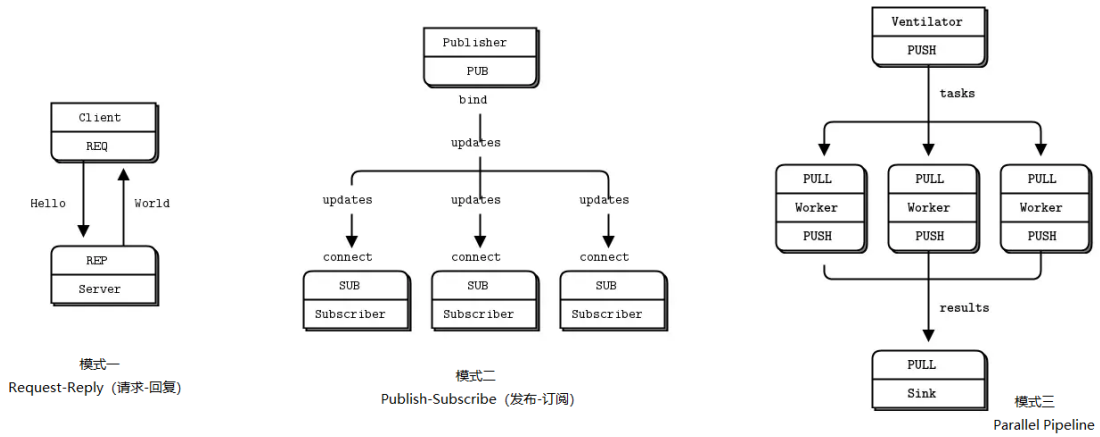

图 1 ZMQ 三种基本的通信模型

■ Request-Reply（请求-回复）

以 “Hello World” 为例。客户端发起请求，并等待服务端回应请求。客户端发送一个简单的 “Hello”，服务端则回应一个 “World”。可以有 N 个客户端，一个服务端，因此是 1-N 连接。

■ Publish-Subscribe（发布-订阅）

以一个天气预报的例子来介绍该模式，服务端不断地更新各个城市的天气，客户端可以订阅自己感兴趣（通过一个过滤器）的城市的天气信息。

■ Parallel Pipeline

Parallel Pipeline 处理模式如下：

- ventilator 分发任务到各个 worker；
- 每个 worker 执行分配到的任务；
- 最后由 sink 收集从 worker 发来的结果。

2.2. CurveZMQ

CurveZMQ 是一个基于 CurveCP 的协议，为 ZMQ 提供安全消息通信，以适应更多的应用场景。CurveZQM 可以工作应用层（在 libzmq 的 API 之上），也可以在 libzmq 内部调用，对应用透明。

CurveZMQ 使用 Curve25519 椭圆曲线提供非对称运算，该曲线是由丹尼尔·J·伯恩斯（英语 ： Daniel J. Bernstein）Daniel J. Bernstein 于 2006 年设计的，

可以在 256bits 密钥的情形下，有更高的性能，与目前广泛使用的 NIST 系列标准椭圆曲线相比 Curve25519 椭圆曲线设计更加开放安全。在运行过程中会生成 Curve25519 椭圆曲线的长期公私密钥对和临时密钥对，基于 Diffie-Hellman 运算得到握手阶段的临时共享密钥和消息保护阶段的共享会话密钥。

CurveZMQ 使用流密码 XSalsa20 的加密认证模式(AEAD)提供握手阶段和消息通信阶段的机密性和完整性保护。Salsa20 使用 64 比特 nonce，XSalsa20 使用 192 比特 nonce。Salsa20 是一种流加密算法，由丹尼尔·J·伯恩斯坦提交到 eSTREAM。2004 年 ECRYPT 启动了 eSTREAM 流密码计划的研究项目 Salsa20 是最终胜出的 7 个算法之一。Salsa20 是基于 hash 函数设计的流密码算法，其核心部分是一个基于 32 比特加、比特异或以及旋转操作的 512 比特输入 512 比特输出的 hash 函数。

Libzmq 使用的密码函数库是 TweetNaCl。TweetNaCl 是一个在 100 条推特内 (14000 字) 内实现的加密库，TweetNaCl 仅包含一对 .h/.c 文件，但是实现了 NaCl 的 25 个主要方法，提供了以下功能：不对称加密 (x25519-xsalsa20-poly1305)，对称加密 (xsalsa20-poly1305)，签名 (ed25519)，哈希 (sha512)。

CurveZMQ分为握手阶段和消息保护阶段。握手阶段由HELLO、WELCOME、INITIATE、READY 四个交互过程完成，握手完成后，双方共享数据保护密钥，使用数据保护密钥对 ZMQ 各种模式的通信消息进行保护。

Client to Server	**Server to Client**
HELLO Packet	
	WELCOME Packet
INITIATE Packet	
	READY Packet
Message packet packet: (C',Box[...](C'->S')) where ... is a message	
	Message packet: (Box[...](S'->C')) where ... is a message

图 2 CurveZMQ 握手协议及通信消息流程

CurveZMQ 协议可以抵抗中间人攻击、重放攻击、在线监听、数据修改等常见的攻击，具有良好的安全特性。

CurveZMQ 的应用场景：

- 保护客户端和服务端直接通信，即中间不需要通过别的阶段转发。这种情形下，CurveZMQ 对应用层透明，直接在 libzmq 库中执行，可用于 ZMQ 的各种通信模式（publish-subscribe, pipeline, 等）。
- 保护客户端和服务端跨越多个不可信节点时的通信，这时需要在应用层协议使用 CurveZMQ 进行异步请求应答(request-reply)通信模式。

下面使用 ABNF 语法描述 Curve ZMQ 协议过程。

```
curvezmq = C:hello ( S:welcome | S:error )
           C:initiate ( S:ready | S:error )
           *message

;   HELLO command, 200 octets
hello = %d5 "HELLO" hello-version hello-padding hello-client hello-nonce
hello-box
hello-version = %x1 %x0     ; CurveZMQ major-minor version
hello-padding = 72%x00      ; Anti-amplification padding
hello-client = 32OCTET      ; Client public transient key C'
hello-nonce = 8OCTET        ; Short nonce, prefixed by "CurveZMQHELLO---"
hello-box = 80OCTET         ; Signature, Box [64 * %x0](C'->S)

;   WELCOME command, 168 octets
welcome = %d7 "WELCOME" welcome-nonce welcome-box
welcome-nonce = 16OCTET     ; Long nonce, prefixed by "WELCOME-"
welcome-box = 144OCTET      ; Box [S' + cookie](S->C')
;   This is the text sent encrypted in the box
cookie = cookie-nonce cookie-box
cookie-nonce = 16OCTET      ; Long nonce, prefixed by "COOKIE--"
cookie-box = 80OCTET        ; Box [C' + s'](K)

;   INITIATE command, 257+ octets
initiate = %d8 "INITIATE" initiate-cookie initiate-nonce initiate-box
initiate-cookie = cookie    ; Server-provided cookie
initiate-nonce = 8OCTET     ; Short nonce, prefixed by "CurveZMQINITIATE"
initiate-box = 144*OCTET    ; Box [C + vouch + metadata](C'->S')
;   This is the text sent encrypted in the box
vouch = vouch-nonce vouch-box
vouch-nonce = 16OCTET       ; Long nonce, prefixed by "VOUCH---"
vouch-box = 80OCTET         ; Box [C',S](C->S')
metadata = *property
property = name value
name = OCTET *name-char
name-char = ALPHA | DIGIT | "-" | "_" | "." | "+"
value = value-size value-data
value-size = 4OCTET         ; Size in network order
value-data = *OCTET         ; 0 or more octets

;   READY command, 30+ octets
ready = %d5 "READY" ready-nonce ready-box
ready-nonce = 8OCTET        ; Short nonce, prefixed by "CurveZMQREADY---"
ready-box = 16*OCTET        ; Box [metadata](S'->C')
```

```
;   ERROR command, 7+ octets
error = %d5 "ERROR" error-reason
error-reason = OCTET 0*255VCHAR

;   MESSAGE command, 33+ octets
message = %d7 "MESSAGE" message_nonce message-box
message-nonce = 8OCTET      ; Short nonce, prefixed by
"CurveZMQMESSAGE-"
message-box = 17*OCTET      ; Box [payload](S'->C') or (C'->S')
;   This is the text sent encrypted in the box
payload = payload-flags payload-data
payload-flags = OCTET       ; Explained below
payload-data = *octet       ; 0 or more octets
```

CurveZMQ 协议过程中使用的密钥层次列表：

	客户端(Client)	**服务端(Server)**
长期密钥 (非对称密钥对)	client-privkey, client-pubk server-pubk（预置服务器公钥）	server-privkey, server-pubk
协商过程 临时非对称密钥对	client-transient-privkey, client-transient-pubk	server-transient-privkey, server-transient-pubk
协商过程 临时对称密钥	ecdh.agree(client-transient-privkey, server-pubk)	ecdh.agree(server-privkey, client-transient-pubk)
	ecdh.agree(client-privkey, server-transient-pubk)	ecdh.agree(server-transient-privkey, client-pubk)
		K: 32 字节随机产生密钥
消息保护密钥	ecdh.agree(client-transient-privkey, server-transient-pubk)	ecdh.agree(server-transient-privkey, client- transient-pubk)

2.3. Intel SGX

Intel SGX 是 Intel Software Guard Extension 的缩写。SGX 是 Intel 指令集架构（ISA）的扩展，主要提供了一些指令用于创建一个可信执行环境（TEE）Enclave。用户态应用程序可以在 Enclave 中安全执行，而不被恶意的 OS、hypervisor(VMM) 所攻击。

SGX 的保护是针对应用程序的地址空间的。SGX 利用处理器提供的指令，

在内存中划分处一部分区域（叫做 EPC），并将应用程序地址空间中的 Enclave 映射到这部分内存区域。这部分内存区域是加密的，通过 CPU 中的内存控制单元进行加密和地址转化。

SGX 内存保护原理：当处理器访问 Enclave 中数据时，CPU 自动切换到一个新的 CPU 模式，叫做 enclave 模式。enclave 模式会强制对每一个内存访问进行额外的硬件检查。由于数据是放在 EPC 中，为了防止已知的内存攻击（如，内存嗅探），EPC 中的内存内容会被内存加密引擎（MEE）加密的。EPC 中的内存内容只有当进入 CPU package 时，才会解密；返回 EPC 内存中会被加密。

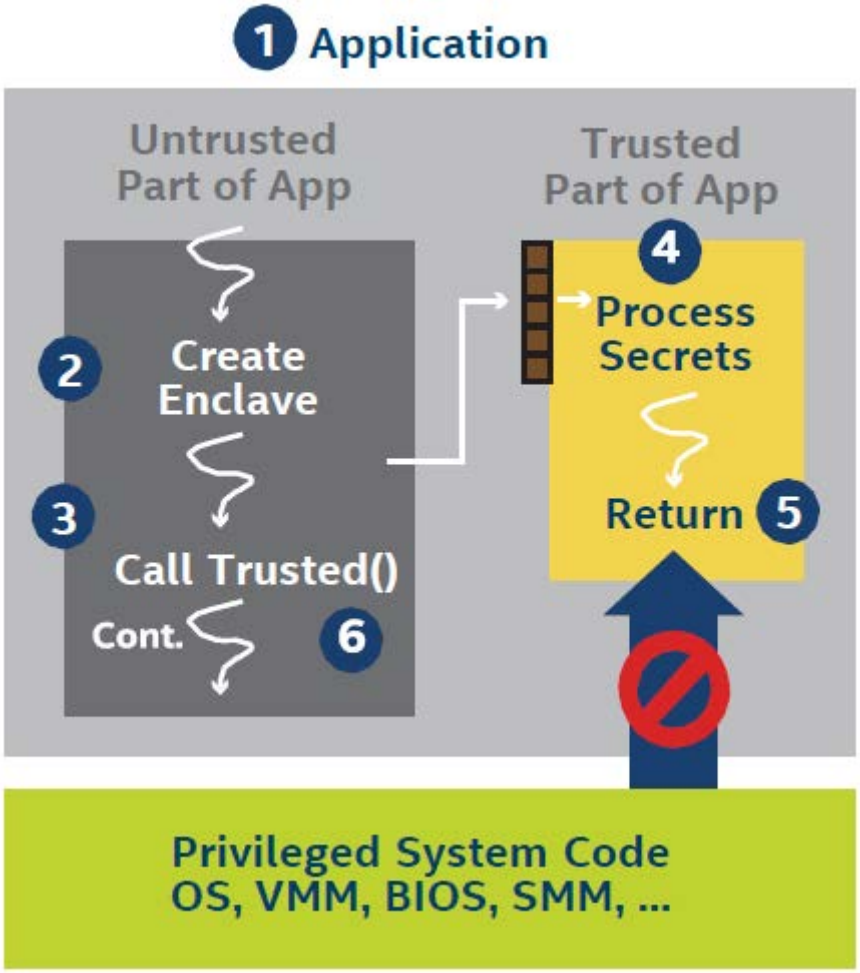

图 3 基于 SGX 的应用程序保护流程

如图 3 所示，SGX 将合法软件的安全操作封装在一个 enclave 中，保护其不受恶意软件的攻击，特权或者非特权的软件都无法访问 enclave，也就是说，一旦软件和数据位于 enclave 中，即便操作系统或者和 VMM（Hypervisor）也无法影响 enclave 里面的代码和数据。Enclave 的安全边界只包含 CPU 和它自身。SGX 创建的 enclave 也可以理解为一个可信执行环境 TEE（Trusted Execution Environment）。不过其与 ARM TrustZone（TZ）还是有一点小区别的，TZ 中通过 CPU 划分为两个隔离环境（安全世界和正常世界），两者之间通过 SMC 指令通信；而 SGX 中一个 CPU 可以运行多个安全 enclaves，并发执行亦可。当然，在 TZ 的安全世界内部实现多个相互隔离的安全服务亦可达到同样的效果。

SGX 的 8 个设计目标：

（1）允许应用开发者保护敏感信息不被运行在更高特权等级下的欺诈软件

非法访问和修改。

（2）能够使应用可以保护敏感代码和数据的机密性和完整性并不会被正常的系统软件对平台资源进行管理和控制的功能所扰乱。

（3）使消费者的计算设备保持对其平台的控制并自由选择下载或不下载他们选择的应用程序和服务。

（4）使平台能够验证一个应用程序的可信代码并且提供一个源自处理器内的包含此验证方式和其他证明代码已经正确的在可信环境下得到初始化的凭证的符号化凭证。

（5）能够使用成熟的工具和处理器开发可信的应用软件.

（6）可信应用程序的性能可以根据对应应用程序底层的处理器进行调整。

（7）使软件开发商通过他们选择的分销渠道可以自行决定可信软件的发布和更新的频率。

（8）能够使应用程序定义代码和数据安全区即使在攻击者已经获得平台的实际控制并直接攻击内存的境况下也能保证安全和隐秘。

三、威胁分析

消息队列已经逐渐成为企业 IT 系统、云基础设施通信的核心手段。它具有低耦合、可靠投递、广播、流量控制、最终一致性等一系列功能，成为异步 RPC 的主要手段之一。ZMQ 作为一种高性能安全消息中间件，在很多领域有着广泛的应用。下面从通信协议、通信主机、软件来源三个方面面临的威胁进行分析。

3.1. 通信协议面临的威胁

在网络上的通信面临以下的四种威胁：

(1) 截获——从网络上窃听他人的通信内容。

(2) 中断——有意中断他人在网络上的通信。

(3) 篡改——故意篡改网络上传送的报文。

(4) 伪造——伪造信息在网络上传送。

截获信息的攻击称为被动攻击，而更改信息和拒绝用户使用资源的攻击称为主动攻击。

3.2. 通信主机面临的威胁

通信主机上面临：病毒、蠕虫、木马、逻辑炸弹等恶意软件的威胁，这些恶意软件可以利用硬件和软件漏洞获取特权执行权限，进而访问其它程序内存中的敏感信息，如密钥等。

3.3. 软件来源威胁

ZMQ 作为一个开源的消息中间件，用户在使用时，需要验证中间件的软件

来源可靠性，通过 SGX 可以构建软件的可信计算基，确认运行正确的消息中间件，避免来自错误或恶意的软件来源。

四、设计思想

CurveZMQ 协议可以应对通信协议面临的各种威胁，但协议中的长期和临时密钥运行的通信主机面临各种恶意程序、硬件软件漏铜的威胁。EncalveZMQ 方案基于 Intel SGX 在非可信主机上实现安全区(Enclave)，实现对 CurveZMQ 协议的密钥管理、协议实现。

4.1. 系统架构

EnclaveZMQ 分为应用程序和 Enclave 程序两部分，两部分之间通过设计的接口进行通信。

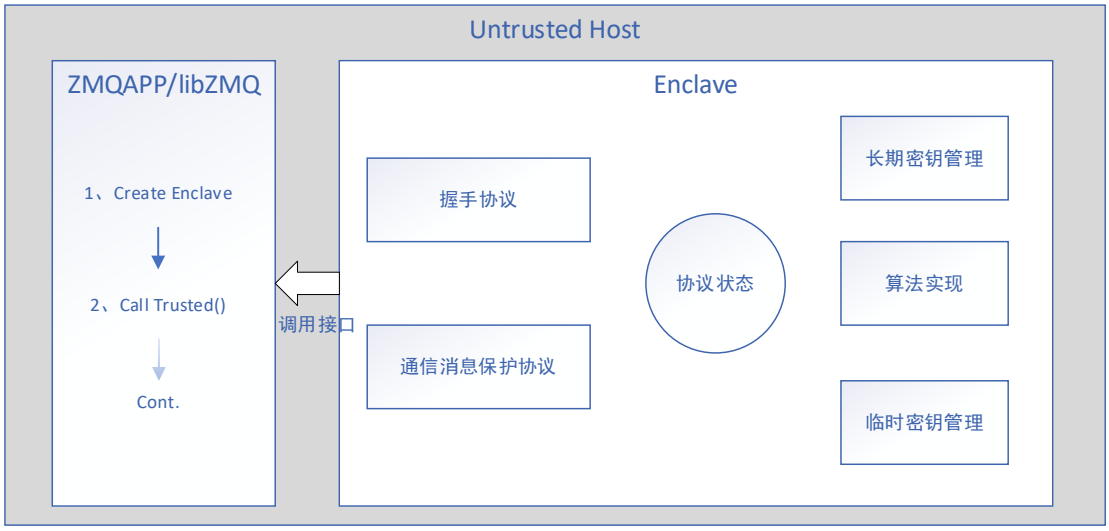

图 4 EnclaveZMQ 系统架构

Enclave 区的功能包括：握手协议模块、通信消息保护协议模块、协议状态管理模块、长期密钥管理模块、算法实现模块、临时密钥管理模块。从 4.2 开始会各模块进行详细介绍。

4.2. 密钥管理

2.2 节中给出了 CurveZMQ 协议使用的各种密钥，可分为两类：长期密钥和临时密钥。下面对这两类密钥的管理进行详细介绍。

4.2.1 长期密钥管理

长期密钥，即客户端和服务端使用的公私密钥。该密钥分别代表客户端和服务端的身份，可用于协议执行过程中对客户端和服务端进行身份认证，防止中

间人攻击等。该密钥在系统部署时进行生成，需长期保存。

Intel SGX SDK 提供了 sgx_seal_data 和 sgx_unseal_data 两个函数用于对 Enclave 中的敏感数据进行封装和解封装，该过程使用固定的算法 AES-GCM 对敏感数据进行保护。Intel SGX 支持两种策略的封装：

- 按 Enclave 身份封装
- 按 Sealing 身份封装

其中按 Enclave 身份封装，只有对应的 Encalve 才能解封，按 Sealing 身份封装时，使用同一签名私钥签的 Encalve 都可以解封装。这里使用 Encalve 身份策略进行长期私钥的封装导出。

被封装后的长期私钥可以保存在非可信区域的非易失存储区，使用时导入到 Encalve 的内存区域使用。

4.2.2 临时密钥管理

Intel SGX 针对 enclave 的保护机制主要包括两个部分：一是 enclave 内存访问语义的变化，二是应用程序地址映射关系的保护，这两项功能共同完成对 Enclave 的机密性和完整性的保护。SGX 在系统内分配一块被保护的物理内存区域 EPC，用来存放 Enclave 和 SGX 数据结构，保证内存保护机制在物理上锁住 EPC 内存区域，将外部的访问请求视为引用了不存在的内存，使得外部的实体(直接存储器访问、图像引擎等)无法访问。

将 CurveZMQ 协议过程中生成的临时非对称公私密钥对，基于长期密钥对和临时密钥对使用 ECDH 生成的共享临时密钥，保存在 Encalve 的内存区域，可以保障密钥的安全性。

4.3. 协议实现及状态管理

CurveZMQ 协议包括握手协议和通信数据保护协议，协议的实现和中间状态管理，在 Encalve 区实现。

Enclave 外部的应用程序不能访问 Enclave 内存; Enclave 内部的代码在 EPC 范围内只能访问属于自己的内存区域，不能访问别的 Enclave 内存；对于 PRM 以外的内存，则按照系统中其他的保护机制进行访问.这样的内存保护机制，防止了 Enclave 内部运行的程序被其他恶意软件盗取隐私信息和篡改。

CurveZMQ 协议的实现和状态管理的 Enclave 实现有效保证了协议的正确运行。

4.4. 算法实现

Libzmq 使用的密码函数库是 TweetNaCl，仅包含一对 .h/.c 文件，将其放在 Enclave 的代码中编译，运行在 Encalve 区域，有效保障了算法的安全可靠执行，以及密钥和算法之间的安全交互。

在第五章会对 Enclave 区运行的算法性能进行评估，比较在非可信区域和 Encalve 区域之间的性能差距，以便于评估 EnclaveZMQ 的实现能够满足 ZMQ 适

用于高性能消息通信的设计目标。

4.5. 应用程序与 Encalve 间接口

应用程序与 Encalve 之间通过自定义接口完成交互，实现相关的密钥管理、协议数据生成与处理等操作。

图 5 应用程序与 Encalve 之间接口

应用程序与 Encalve 之间接口分为两类，密钥管理接口和协议处理接口。

密钥管理接口包括：CreateKeyPair、ExportSealedPrivateKey 和 ImportSealedPrivateKey。

协议处理接口：Produce/ProcessHello、Produce/ProcessWelcome、Produce/ProcessInitiate、Produce/ProcessReady、Produce/ProcessMessage。

4.6. 系统流程

EnclaveZMQ 的系统流程分为三个阶段，如图 6 所示：

a) 密钥初始化阶段

流程中 a.1-a.2 为客户端和服务端产生长期公私密钥对，并进行封装导出。

流程中 a.3 需要通过离线或其它可信的通道，在客户端预置服务端公钥。

b) 握手阶段

b.1-b.4 为 CurveZMQ 的握手协议交互流程，协议执行过程中遇到错误时，会返回 Error 消息。

c) 消息通信阶段

握手阶段成功后，得到消息保护的会话密钥，使用该密钥对消息进行机密性和完整性保护。

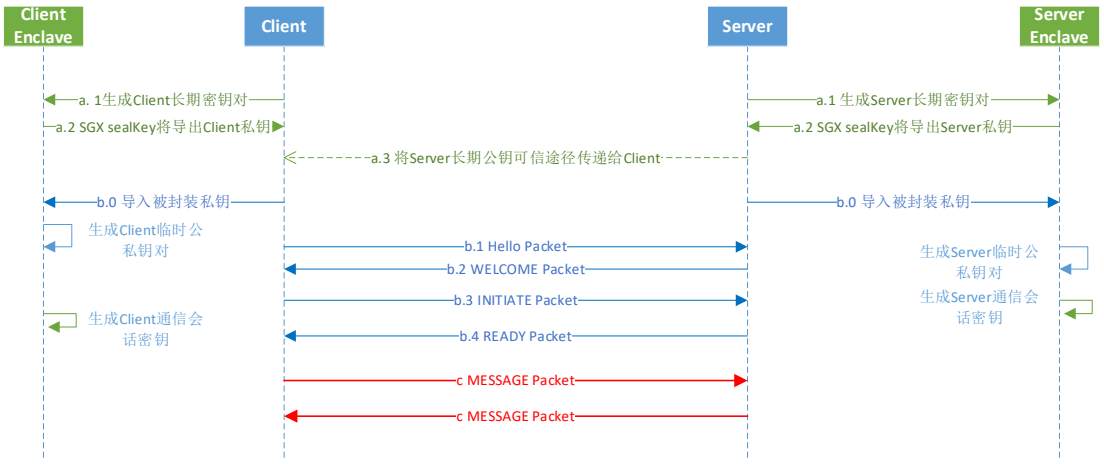

图 6 EnclaveZMQ 系统流程

五、性能分析

根据第四章的设计，使用 OpenEncalve 框架实现了 CurveZMQ 协议，并对协议中使用的非对称算法（Curve25519 椭圆曲线）和对称算法（XSalsa20 的加密认证模式），进行了性能测试。

非对称算法（Curve25519 椭圆曲线）测试了密钥生成每秒钟的生成次数，对称算法（XSalsa20 的加密认证模式）测试了加密 64 字节数据的情形下，每秒钟的加密次数。

本次性能测试主机的配置为：CPU 是 Intel® Cor™ i5-9600KF @ 3.70GHZx6，内存 16G。单线程情形下测试得到两个指标分别如图 7 和图 8 所示。

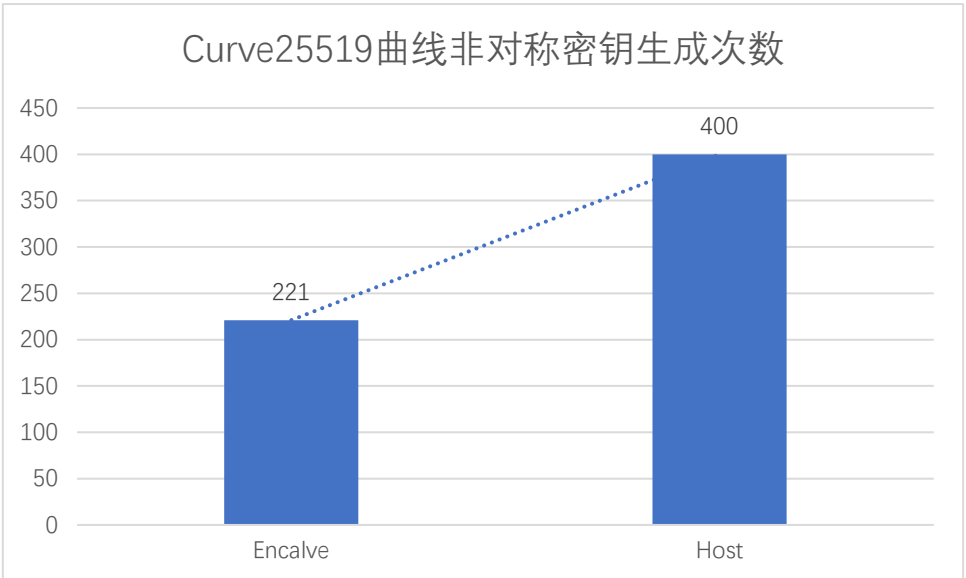

图 7 非对称密钥生成次数对比

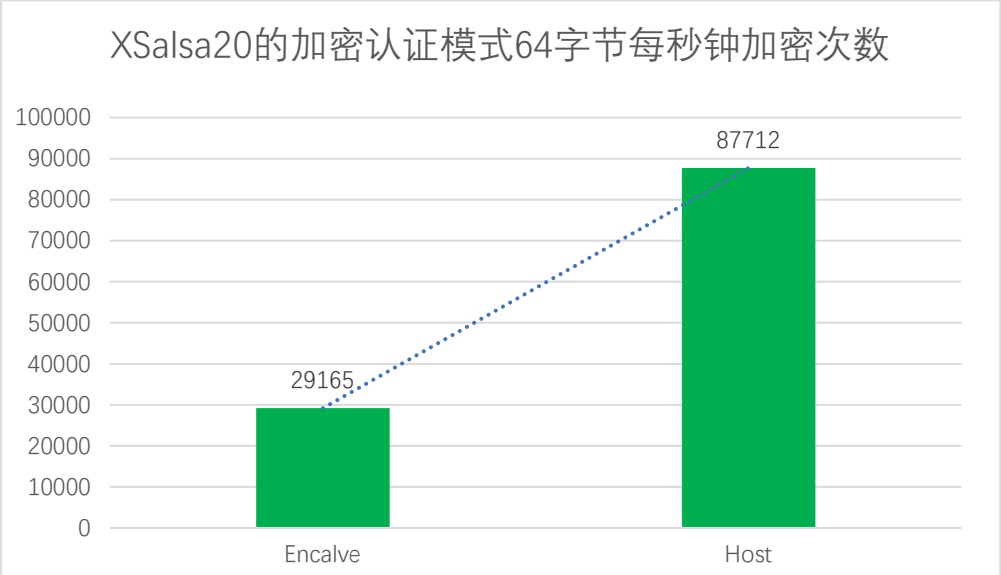

图 8 XSalsa20 64 字节加密次数对比

图 7 和图 8 中的测试数据看到，非对称算法（Curve25519 椭圆曲线）密钥生成每秒钟的生成次数，Encalve 环境大概是 Host 环境的 1/2；对称算法（XSalsa20 的加密认证模式）加密 64 字节数据的情形下，每秒钟的加密次数，Encalve 环境大概是 Host 环境的 1/3。

六、结论

本文讨论了在 Intel® Encalve 可执行环境中实现 ZMQ 的安全协议 CurveZMQ 的方案。本方案中，将 CurveZMQ 的随机数产生、密钥生成与管理、协议实现、状态管理在 Encalve 可执行环境中实现，能有效应对各类安全风险，消除对应的安全威胁，可以有效保障 ZMQ 在高安全要求环境下的安全有效运行。

本方案中，还给出了 Enclave 和 Host 环境下的通信函数接口，及相应的数据类型，可以很方便的适配到 libzmq 的代码中。

通过第五章的性能测试，给出 Enclave 环境和 Host 环境下，两个算法密码算法的性能对比，Encalve 环境中的密码算法性能可以应对实际应用的需求。

综上所述，Intel® Encalve 可执行环境和 ZMQ 协议结合为高安全要求环境提供了一种有效的解决方案。