对学习Intel SGX的进展汇报

> 我是在寒假前（2021年1月底）开始在李佳轩学长的指导下学习Trusted Execution Environment (TEE) 和 Intel Software Guard Extension (Intel SGX)。开始学习后，李佳轩学长与我每周日会进行一次线上或者线下会议。会议内容主要是我汇报与总结一周的学习进展，学长为我解答途中遇到的困难，以及布置下周的任务。我目前（2021年3月中）已经初步学习并实现了Intel SGX 的基本应用功能，即将开始学习相关的理论知识。因此，我想向王琦老师您汇报一下我在这一学习阶段中具体的学习情况。

学习总结

这一部分是对我每周的学习进展进行一个总结。

> 寒假在家期间，由于远程会议十分不方便，我前三周都写了一个简短的周报，用以更清晰的汇报进展。
>
> 第一周到第三周的周报详见附录一 周报汇总.pdf。
>
> SGX应用实现代码详见附录二至五
>
> - HelloEnclave
> - Local Attestation
> - Remote Attestation (server)
> - Remote Attestation (client)

首次会议

学长向我介绍了TEE与SGX的基础知识与工作流程，并规划了第一阶段要完成的几项任务：

1. 安装并配置Intel SGX的运行环境
2. 创建一个Enclave
3. 通过Ecall和Ocall实现可信部分与不可信部分的相互调用
4. 实现同一CPU上两个不同Enclave的Local Attestation
5. 通过Intel Attestation Service (IAS) 实现不同CPU上两个Enclave的Remote Attestation
6. 通过Sealing实现Enclave在硬盘上的保存

第一周（2021/2/7）

这周主要是完成了Intel SGX编译运行环境的初步搭建，具体进行了：

1. 安装与配置Ubuntu 20.04 LTS
2. 安装Intel SGX SDK
3. 尝试安装SGX驱动（第二周才发现并没有安装成功）
4. 安装Intel SGX PSW
5. 编译运行样例代码

第二周（2021/2/14）

这周发现在重新编译样例代码失败后，发现驱动并没有正确安装。即使Intel 官方提供的工具显示我的CPU支持SGX服务，并且已经Enable了，驱动安装程序也显示成功安装，但仍检测不到sgx 的驱动。在搜寻解决方案无果后，初步怀疑是电脑自带的BIOS没有支持SGX服务。不过我仍可以使用模拟模式来编译运行代码。模拟模式和硬件模式的区别只在于能否真正使用SGX指令集，二者的代码部分是相同的，所以使用模拟模式并不影响接下来的学习。

在确定了上述问题之后，我开始学习创建一个Enclave，并实现Ecall与Ocall。

> Intel SGX应用通常是用c/c++来实现的。简单来说，一个SGX应用通常被分为 App 和 Enclave 两个部分。App部分是不可信的，与普通c/c++程序没有区别；而Enclave是可信的，在运行时由CPU来保证运行过程对于操作系统是不可知的。这两个部分的切换是通过调用Ecall与Ocall函数来实现的。App调用Ecall来进入Enclave，而Ocall的作用相反。

我实现了一个HelloEnclave的SGX应用，相关代码在HelloEnclave。应用会经历以下过程：

1. 进入App部分
2. 在App内调用SGX库函数来创建与初始化指定的Enclave
3. 调用Ecall进入Enclave
4. 在Enclave内生成“Hello World”信息，并调用Ocall传回App
5. App将信息打印出来

第三周（2021/2/21）

这周要实现的内容所涉及的文件与函数更多，向上周那样使用vim来进行代码阅读与编写的效率很低，所以我选择了Clion IDE来辅助。由于之前没有使用Clion编译运行小型项目的经验，并且此前从未接触过Makefile，所以在实现程序前花了近两天时间来配置Clion IDE并学习Makefile的使用。

这周学习了两个Enclave之间的Local Attestation

> Local Attestation是在本地Enclave之间的验证过程，用于确认两个Enclave是否由同一个CPU所创建。两个Enclave通过Diffie-Hellman Key Exchange算法交换密钥，同时交换用以进行Local Attestation的report。

我实现了一个LocalAttestation的SGX应用，相关代码在Local Attestation 。应用会经历以下过程：

1. 进入App部分
2. 在App内创建两个Enclave，称作e1和e2
3. 调用e1的Ecall来发起验证请求
4. e1和e2轮流发送msg0(request), msg1, msg2, msg3。其中包含了双方用于Local Attestation的report
5. 成功建立session
6. 相互交换信息
7. 关闭session

第四周（2021/2/28）

这周由于返校，前前后后占用了许多时间，所以除了了解Remote Attestation的流程外，并没有实质性的进展。

第五周（2021/3/7）

这周学习了Remote Attestation。

> Remote Attestation与Local Attestation相似。在Local Attestation中， 本地CPU充当着担保验证结果的角色。但在Remote Attestation中，为保证可信度，由Intel的IAS来担任这一角色。Intel拥有所有生产的CPU的相关信息，所以可以通过中心化的IAS来为Remote Attestation提供验证服务。
>
> 事实上，Remote Attestation要验证的只是被验证方的Enclave是否为所在CPU所创建的。验证方可以是Enclave，也可以是普通的应用程序。在建立session的过程中，被验证方会通过Enclave向CPU请求一份report，发往验证方。在收到report后，验证方通过Intel提供的EPID将report发给IAS进行验证。

这周通过阅读Intel SGX官方文档与样例代码，已经了解了Remote Attestation的工作流程。实现这一过程原本是需要实现Server和Client两个应用程序，但样例代码是通过将Server与Client合并为一个程序，并使用函数调用的方式模拟远程网络通信的。我此前并没有接触过c++如何进行网络通信，也不清楚样例代码的Server与Client在哪些地方进行了复用，所以这周并没有完成代码的实现。

在学习其它Remote Attestation实现代码的过程中，我从网站sgx101.com上了解到Google Protocol Buffer比直接使用socket更加安全与高效，虽然网站上的Remote Attestation样例代码并没有成功运行，但我也从中了解到了Server与Client的大致结构。

由于使用IAS验证需要事先在Intel注册这项服务，所以这周也完成了Intel的账号注册与IAS服务申请，并取得了EPID等验证材料。

第六周（2021/3/14）

这周成功实现了Remote Attestation。

我最终选择了使用最基础的socket来进行网络通信。在学习了如何使用socket，并了解了每次Server与Client通信时所发送的信息格式后，开始编写相关代码。最终成功的使Server与Client完成了Remote Attestation。

相关代码在Remote Attestation (server)与Remote Attestation (client)，应用会经历以下过程：

1. 启动Server，等待Client的连接
2. 启动Client，Client与Server建立连接
3. Client向Server发送请求
4. Server同意，并请求Client发送report
5. Client App进入Enclave，取得report与EPID并发送给Server
6. Server将report与EPID发送给IAS进行验证，并取得验证结果

阶段学习小结

经过了一个多月的学习，我对于SGX的功能与工作流程有了整体上的认识，也能够实现其基础功能了。除此之外，我也熟悉了Linux的使用，学会如何阅读c/c++项目代码，学习了如何用c++实现socket通信。

下一阶段计划

接下来是对于理论知识的学习阶段。需要学习Intel SGX在实现Attestation、Sealing等功能时所涉及的理论知识。之后具体学习计划会在这周的会议上讨论。

由于还有两门课在推免方案中，所以本学期我会更倾向于优先花更多的时间在专业课程上；下半年课程压力稍微轻松些，我会更专注于在这条路上的学习。

按照李佳轩学长的计划，他会指导我在今年至少跟着写出一篇文章，完成一个专利。我也会朝着这个方向继续努力，争取在今年取得更多的成果。

附录一：周报汇总

第一周周报

> 2021/2/7

本周已完成任务

- 安装sgx运行环境
- 运行样例代码（模拟&硬件模式）
- 学习sgx101的课程
- 学习vim

将来的任务

- 创建enclave
- ecall&ocall
- local&remote
- load&read

第二周周报

> 2021/2/14

本周已完成任务

- 创建enclave
- 通过调用ecall 和 ocall来打印“hello world” (模拟模式)

这周我发现上周的硬件模式并没有跑通，只是因为我在编译硬件模式前没有执行[make clean] 命令，导致程序一直运行在模拟模式下。

排查之后发现是sgx驱动没装好，但在多次尝试之后，即使驱动程序显示成功安装，系统中也依然找不到相应的硬件，仍处于未安装驱动的状态。

所以本周任务是在模拟模式下完成的

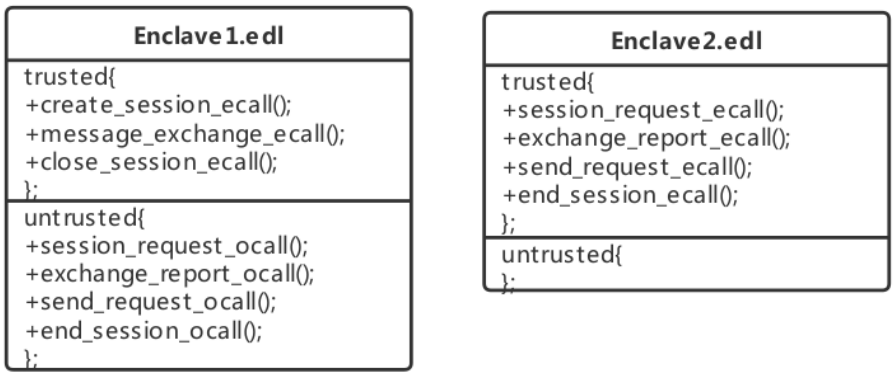
下周计划任务

- 实现local&remote attestation
- 实现sealing

第三周周报

> 2021/2/21

本周已完成任务

- Local attestation
  - 

- message exchange

Local Attestation总结

process

1. create Encalve1 and Enclave2
2. create session (Diffie-Hellman Key Exchange)
    1. Enclave1 send a session request to Enclave2 (Session Request)
        1. sgx_dh_init_session();
        2. session_request_ocall();
    2. Enclave2 process request and send msg1 to Enclave1
        1. sgx_dh_init_session();
        2. sgx_dh_responder_gen_msg1();
        3. return msg3;
    3. Enclave1 process msg1 and send msg2 to Enclave2 (Exchange Report)
        1. sgx_dh_initiator_proc_msg1();
        2. exchange_report_ocall();
    4. Enclave2 process msg2 and send msg3 to Enclave2
        1. sgx_dh_responder_gen_msg2();
        2. return msg3;
    5. Enclave1 process msg3 and the session created
        1. sgx_dh_initiator_proc_msg3();
3. message exchange
    1. Enclave1 send message exchange request to Enclave2
        1. send_request_ecall();
        2. sgx_rijndael128GCM_decrypt();
    2. Enclave2 send response to Enclave1
        1. generate_response_ecall();
        2. sgx_rijndael128GCM_decrypt();
4. close session
    1. Enclave1 send close session request
        1. end_session_ocall();

2. Enclave2 close session and return
    1. end_session_ecall();
3. Enclave1 close session
    1. close_session_ecall();
5. destroy Enclave1 and Enclave2

下周计划任务

- 申请EPID?
- 实现remote attestation
- 实现enclave在硬盘上的load和read

附录二：HelloEnclave 代码实现

App

App.h

```
#ifndef _APP_H_
#define _APP_H_

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

#include "sgx_error.h"       /* sgx_status_t */
#include "sgx_eid.h"     /* sgx_enclave_id_t */

#ifndef TRUE
# define TRUE 1
#endif

#ifndef FALSE
# define FALSE 0
#endif

# define TOKEN_FILENAME   "enclave.token"
# define ENCLAVE_FILENAME "enclave.signed.so"

extern sgx_enclave_id_t global_eid;    /* global enclave id */

#if defined(__cplusplus)
extern "C" {
#endif

#if defined(__cplusplus)
}
#endif

#endif /* !_APP_H_ */
```

App.cpp

```
#include <stdio.h>
#include <string.h>

# include <unistd.h>
# include <pwd.h>
```

```
# define MAX_PATH FILENAME_MAX

#include "sgx_urts.h"
#include "App.h"
#include "Enclave_u.h"

/* Global EID shared by multiple threads */
sgx_enclave_id_t global_eid = 0;

typedef struct _sgx_errlist_t {
    sgx_status_t err;
    const char *msg;
    const char *sug; /* Suggestion */
} sgx_errlist_t;

/* Error code returned by sgx_create_enclave */
static sgx_errlist_t sgx_errlist[] = {
        {
                SGX_ERROR_UNEXPECTED,
                "Unexpected error occurred.",
                NULL
        },
        {
                SGX_ERROR_INVALID_PARAMETER,
                "Invalid parameter.",
                NULL
        },
        {
                SGX_ERROR_OUT_OF_MEMORY,
                "Out of memory.",
                NULL
        },
        {
                SGX_ERROR_ENCLAVE_LOST,
                "Power transition occurred.",
                "Please refer to the sample \"PowerTransition\" for details."
        },
        {
                SGX_ERROR_INVALID_ENCLAVE,
                "Invalid enclave image.",
                NULL
        },
        {
                SGX_ERROR_INVALID_ENCLAVE_ID,
                "Invalid enclave identification.",
                NULL
        },
        {
                SGX_ERROR_INVALID_SIGNATURE,
                "Invalid enclave signature.",
                NULL
        },
        {
                SGX_ERROR_OUT_OF_EPC,
                "Out of EPC memory.",
                NULL
        },
        {
```

```
                SGX_ERROR_NO_DEVICE,
                "Invalid SGX device.",
                "Please make sure SGX module is enabled in the BIOS, and install
SGX driver afterwards."
        },
        {
                SGX_ERROR_MEMORY_MAP_CONFLICT,
                "Memory map conflicted.",
                NULL
        },
        {
                SGX_ERROR_INVALID_METADATA,
                "Invalid enclave metadata.",
                NULL
        },
        {
                SGX_ERROR_DEVICE_BUSY,
                "SGX device was busy.",
                NULL
        },
        {
                SGX_ERROR_INVALID_VERSION,
                "Enclave version was invalid.",
                NULL
        },
        {
                SGX_ERROR_INVALID_ATTRIBUTE,
                "Enclave was not authorized.",
                NULL
        },
        {
                SGX_ERROR_ENCLAVE_FILE_ACCESS,
                "Can't open enclave file.",
                NULL
        },
};

/* Check error conditions for loading enclave */
void print_error_message(sgx_status_t ret) {
    size_t idx = 0;
    size_t ttl = sizeof sgx_errlist / sizeof sgx_errlist[0];

    for (idx = 0; idx < ttl; idx++) {
        if (ret == sgx_errlist[idx].err) {
            if (NULL != sgx_errlist[idx].sug)
                printf("Info: %s\n", sgx_errlist[idx].sug);
            printf("Error: %s\n", sgx_errlist[idx].msg);
            break;
        }
    }

    if (idx == ttl)
        printf("Error code is 0x%X. Please refer to the \"Intel SGX SDK Developer
Reference\" for more details.\n",
               ret);
}

/* Initialize the enclave:
```

```
 *   Step 1: try to retrieve the launch token saved by last transaction
 *   Step 2: call sgx_create_enclave to initialize an enclave instance
 *   Step 3: save the launch token if it is updated
 */
int initialize_enclave(void) {
    char token_path[MAX_PATH] = {'\0'};
    sgx_launch_token_t token = {0};
    sgx_status_t ret = SGX_ERROR_UNEXPECTED;
    int updated = 0;

    /* Step 1: try to retrieve the launch token saved by last transaction
     *         if there is no token, then create a new one.
     */
    /* try to get the token saved in $HOME */
    const char *home_dir = getpwuid(getuid())->pw_dir;

    if (home_dir != NULL &&
        (strlen(home_dir) + strlen("/") + sizeof(TOKEN_FILENAME) + 1) <=
MAX_PATH) {
        /* compose the token path */
        strncpy(token_path, home_dir, strlen(home_dir));
        strncat(token_path, "/", strlen("/"));
        strncat(token_path, TOKEN_FILENAME, sizeof(TOKEN_FILENAME) + 1);
    } else {
        /* if token path is too long or $HOME is NULL */
        strncpy(token_path, TOKEN_FILENAME, sizeof(TOKEN_FILENAME));
    }

    FILE *fp = fopen(token_path, "rb");
    if (fp == NULL && (fp = fopen(token_path, "wb")) == NULL) {
        printf("Warning: Failed to create/open the launch token file \"%s\".\n",
token_path);
    }

    if (fp != NULL) {
        /* read the token from saved file */
        size_t read_num = fread(token, 1, sizeof(sgx_launch_token_t), fp);
        if (read_num != 0 && read_num != sizeof(sgx_launch_token_t)) {
            /* if token is invalid, clear the buffer */
            memset(&token, 0x0, sizeof(sgx_launch_token_t));
            printf("Warning: Invalid launch token read from \"%s\".\n",
token_path);
        }
    }
    /* Step 2: call sgx_create_enclave to initialize an enclave instance */
    /* Debug Support: set 2nd parameter to 1 */
    ret = sgx_create_enclave(ENCLAVE_FILENAME, SGX_DEBUG_FLAG, &token, &updated,
&global_eid, NULL);
    if (ret != SGX_SUCCESS) {
        print_error_message(ret);
        if (fp != NULL) fclose(fp);
        return -1;
    }

    /* Step 3: save the launch token if it is updated */
    if (updated == FALSE || fp == NULL) {
        /* if the token is not updated, or file handler is invalid, do not
perform saving */
```

```
        if (fp != NULL) fclose(fp);
        return 0;
    }

    /* reopen the file with write capablity */
    fp = freopen(token_path, "wb", fp);
    if (fp == NULL) return 0;
    size_t write_num = fwrite(token, 1, sizeof(sgx_launch_token_t), fp);
    if (write_num != sizeof(sgx_launch_token_t))
        printf("Warning: Failed to save launch token to \"%s\".\n", token_path);
    fclose(fp);
    return 0;
}

/* OCall functions */
void ocall_print_string(const char *str) {
    /* Proxy/Bridge will check the length and null-terminate
     * the input string to prevent buffer overflow.
     */
    printf("%s", str);
}


/* Application entry */
int SGX_CDECL main(int argc, char *argv[]) {
    (void) (argc);
    (void) (argv);


    /* Initialize the enclave */
    if (initialize_enclave() < 0) {
        printf("Enter a character before exit ...\n");
        getchar();
        return -1;
    }

    printf_helloworld(global_eid);

    /* Destroy the enclave */
    sgx_destroy_enclave(global_eid);

    return 0;
}
```

Enclave

Enclave.edl

```
enclave {

    /* Import ECALL/OCALL from sub-directory EDLs.
```

```
     *  [from]: specifies the location of EDL file.
     *  [import]: specifies the functions to import,
     *  [*]: implies to import all functions.
     */

    trusted {
        public void printf_helloworld();
    };

    /*
     * ocall_print_string - invokes OCALL to display string buffer inside the
enclave.
     *  [in]: copy the string buffer to App outside.
     *  [string]: specifies 'str' is a NULL terminated buffer.
     */
    untrusted {
        void ocall_print_string([in, string] const char *str);
    };

};
```

Enclave.h

```
#ifndef _ENCLAVE_H_
#define _ENCLAVE_H_

#include <stdlib.h>
#include <assert.h>

#if defined(__cplusplus)
extern "C" {
#endif

void printf(const char *fmt, ...);
void printf_helloworld();

#if defined(__cplusplus)
}
#endif

#endif /* !_ENCLAVE_H_ */
```

Enclave.cpp

```
#include <stdarg.h>
#include <stdio.h>      /* vsnprintf */

#include "Enclave.h"
#include "Enclave_t.h"  /* print_string */

/*
```

```
 * printf:
 *   Invokes OCALL to display the enclave buffer to the terminal.
 */
void printf(const char *fmt, ...)
{
    char buf[BUFSIZ] = {'\0'};
    va_list ap;
    va_start(ap, fmt);
    vsnprintf(buf, BUFSIZ, fmt, ap);
    va_end(ap);
    ocall_print_string(buf);
}

void printf_helloworld()
{
    printf("Hello World\n");
}
```

附录三：Local Attestation 代码实现

> 由于文件数量较多，忽略各头文件

App

App.cpp

```
#include <stdio.h>
#include <map>
#include <assert.h>

#include "sgx_eid.h"
#include "sgx_urts.h"

#include "EnclaveInitiator_u.h"
#include "EnclaveResponder_u.h"
#include "sgx_utils.h"

#define ENCLAVE_INITIATOR_NAME "libenclave_initiator.signed.so"
#define ENCLAVE_RESPONDER_NAME "libenclave_responder.signed.so"

sgx_enclave_id_t initiator_enclave_id = 0, responder_enclave_id = 0;

int main(int argc, char* argv[])
{
    int update = 0;
    uint32_t ret_status;
    sgx_status_t status;
    sgx_launch_token_t token = {0};

    (void)argc;
    (void)argv;


    // load initiator and responder enclaves
    if (SGX_SUCCESS != sgx_create_enclave(ENCLAVE_INITIATOR_NAME, SGX_DEBUG_FLAG,
&token, &update, &initiator_enclave_id, NULL)
        || SGX_SUCCESS != sgx_create_enclave(ENCLAVE_RESPONDER_NAME,
SGX_DEBUG_FLAG, &token, &update, &responder_enclave_id, NULL)) {
        printf("failed to load enclave...\n");
        goto destroy_enclave;
    }
    printf("succeed to load enclaves...\n");



    // create ECDH session using initiator enclave, it would create session with
responder enclave
    status = create_session_ecall(initiator_enclave_id, &ret_status);
    if (status != SGX_SUCCESS || ret_status != 0) {
        printf("failed to establish secure channel: ECALL return 0x%x, error code
is 0x%x.\n", status, ret_status);
        goto destroy_enclave;
```

```
    }
    printf("succeed to establish secure channel.\n");

    // test message exchanging between initiator enclave and responder enclave
    status = message_exchange_ecall(initiator_enclave_id, &ret_status);
    if (status != SGX_SUCCESS || ret_status != 0) {
        printf("test_message_exchange Ecall failed: ECALL return 0x%x, error code
is 0x%x.\n", status, ret_status);
        goto destroy_enclave;
    }
    printf("Succeed to exchange secure message...\n");

    // close ECDH session
    status = close_session_ecall(initiator_enclave_id, &ret_status);
    if (status != SGX_SUCCESS || ret_status != 0) {
        printf("test_close_session Ecall failed: ECALL return 0x%x, error code is
0x%x.\n", status, ret_status);
        goto destroy_enclave;
    }
    printf("Succeed to close Session...\n");

    destroy_enclave:
    sgx_destroy_enclave(initiator_enclave_id);
    sgx_destroy_enclave(responder_enclave_id);

    printf("Destroyed enclaves\n");

    return 0;
}
```

UntrustedEnclaveMessageExchange.cpp

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include "sgx_eid.h"
#include "error_codes.h"
#include "datatypes.h"
#include "sgx_urts.h"
#include "UntrustedEnclaveMessageExchange.h"
#include "sgx_dh.h"

#include "fifo_def.h"
#include "EnclaveResponder_u.h"

extern sgx_enclave_id_t responder_enclave_id;
extern "C"
uint32_t session_request_ocall(sgx_dh_msg1_t *dh_msg1, uint32_t *session_id) {
    uint32_t retcode;
    session_request_ecall(responder_enclave_id, &retcode, dh_msg1, session_id);
    return retcode == SGX_SUCCESS ? SGX_SUCCESS : INVALID_SESSION;
}
```

```
uint32_t exchange_report_ocall(sgx_dh_msg2_t *dh_msg2, sgx_dh_msg3_t *dh_msg3,
uint32_t session_id) {
    uint32_t retcode;
    exchange_report_ecall(responder_enclave_id, &retcode, dh_msg2, dh_msg3,
session_id);
    return retcode == SGX_SUCCESS ? SGX_SUCCESS : INVALID_SESSION;
}

uint32_t send_request_ocall(uint32_t session_id, secure_message_t *req_message,
size_t req_message_size,
                            size_t max_payload_size, secure_message_t
*resp_message, size_t resp_message_size) {
    uint32_t retcode;
    generate_response_ecall(responder_enclave_id, &retcode, req_message,
req_message_size, max_payload_size, resp_message,
                        resp_message_size, session_id);
    return retcode == SGX_SUCCESS ? SGX_SUCCESS : INVALID_SESSION;
}


uint32_t end_session_ocall(uint32_t session_id) {
    uint32_t retcode;
    end_session_ecall(responder_enclave_id, &retcode, session_id);
    return retcode == SGX_SUCCESS ? SGX_SUCCESS : INVALID_SESSION;
}
```

EnclaveInitiator (e1)

EnclaveInitiator.edl

```
enclave {
    include "sgx_eid.h"
    include "datatypes.h"
    include "dh_session_protocol.h"

    trusted{
            public uint32_t create_session_ecall();
            public uint32_t message_exchange_ecall();
            public uint32_t close_session_ecall();
    };

    untrusted{
            uint32_t session_request_ocall([out] sgx_dh_msg1_t *dh_msg1,[out]
uint32_t *session_id);
            uint32_t exchange_report_ocall([in] sgx_dh_msg2_t *dh_msg2, [out]
sgx_dh_msg3_t *dh_msg3, uint32_t session_id);
            uint32_t send_request_ocall(uint32_t session_id, [in, size =
req_message_size] secure_message_t* req_message, size_t req_message_size, size_t
max_payload_size, [out, size=resp_message_size] secure_message_t* resp_message,
size_t resp_message_size);
            uint32_t end_session_ocall(uint32_t session_id);
    };
```

```
};
```

EnclaveInitiator.cpp

```
// Enclave1: Defines the exported functions for the .so application
#include "sgx_eid.h"
#include "EnclaveInitiator_t.h"
#include "EnclaveMessageExchange.h"
#include "error_codes.h"
#include "Utility_E1.h"
#include "sgx_dh.h"
#include "sgx_utils.h"
#include <map>

#define UNUSED(val) (void)(val)

#define RESPONDER_PRODID 1

std::map<sgx_enclave_id_t, dh_session_t>g_src_session_info_map;

dh_session_t g_session;

// This is hardcoded responder enclave's MRSIGNER for demonstration purpose. The
content aligns to responder enclave's signing key
sgx_measurement_t g_responder_mrsigner = {
    {
        0x83, 0xd7, 0x19, 0xe7, 0x7d, 0xea, 0xca, 0x14, 0x70, 0xf6, 0xba, 0xf6,
0x2a, 0x4d, 0x77, 0x43,
        0x03, 0xc8, 0x99, 0xdb, 0x69, 0x02, 0x0f, 0x9c, 0x70, 0xee, 0x1d, 0xfc,
0x08, 0xc7, 0xce, 0x9e
    }
};

/* Function Description:
 *   This is ECALL routine to create ECDH session.
 *   When it succeeds to create ECDH session, the session context is saved in
g_session.
 * */
extern "C" uint32_t create_session_ecall()
{
        return create_session(&g_session);
}

/* Function Description:
 *   This is ECALL routine to transfer message with ECDH peer
 * */
uint32_t message_exchange_ecall()
{
    uint32_t ke_status = SUCCESS;
    uint32_t target_fn_id, msg_type;
    char* marshalled_inp_buff;
    size_t marshalled_inp_buff_len;
    char* out_buff;
    size_t out_buff_len;
    size_t max_out_buff_size;
    char* secret_response;
```

```
    uint32_t secret_data;

    target_fn_id = 0;
    msg_type = MESSAGE_EXCHANGE;
    max_out_buff_size = 50; // it's assumed the maximum payload size in response
message is 50 bytes, it's for demonstration purpose
    secret_data = 0x12345678; //Secret Data here is shown only for purpose of
demonstration.

    //Marshals the secret data into a buffer
    ke_status = marshal_message_exchange_request(target_fn_id, msg_type,
secret_data, &marshalled_inp_buff, &marshalled_inp_buff_len);
    if(ke_status != SUCCESS)
    {
        return ke_status;
    }

    //Core Reference Code function
    ke_status = send_request_receive_response(&g_session, marshalled_inp_buff,
                                                marshalled_inp_buff_len,
max_out_buff_size, &out_buff, &out_buff_len);
    if(ke_status != SUCCESS)
    {
        SAFE_FREE(marshalled_inp_buff);
        SAFE_FREE(out_buff);
        return ke_status;
    }

    //Un-marshal the secret response data
    ke_status = umarshal_message_exchange_response(out_buff, &secret_response);
    if(ke_status != SUCCESS)
    {
        SAFE_FREE(marshalled_inp_buff);
        SAFE_FREE(out_buff);
        return ke_status;
    }

    SAFE_FREE(marshalled_inp_buff);
    SAFE_FREE(out_buff);
    SAFE_FREE(secret_response);
    return SUCCESS;
}


/* Function Description:
 *   This is ECALL interface to close secure session*/
uint32_t close_session_ecall()
{
    uint32_t ke_status = SUCCESS;

    ke_status = close_session(&g_session);

    //Erase the session context
    memset(&g_session, 0, sizeof(dh_session_t));
    return ke_status;
}

/* Function Description:
```

```
 *   This is to verify peer enclave's identity.
 * For demonstration purpose, we verify below points:
 *   1. peer enclave's MRSIGNER is as expected
 *   2. peer enclave's PROD_ID is as expected
 *   3. peer enclave's attribute is reasonable: it's INITIALIZED'ed enclave; in
non-debug build configuration, the enclave isn't loaded with enclave debug mode.
 **/
extern "C" uint32_t verify_peer_enclave_trust(sgx_dh_session_enclave_identity_t*
peer_enclave_identity)
{
    if (!peer_enclave_identity)
        return INVALID_PARAMETER_ERROR;

    // check peer enclave's MRSIGNER
    if (memcmp((uint8_t *)&peer_enclave_identity->mr_signer,
(uint8_t*)&g_responder_mrsigner, sizeof(sgx_measurement_t)))
        return ENCLAVE_TRUST_ERROR;

    // check peer enclave's product ID and enclave attribute (should be
INITIALIZED'ed)
    if (peer_enclave_identity->isv_prod_id != RESPONDER_PRODID || !
(peer_enclave_identity->attributes.flags & SGX_FLAGS_INITTED))
        return ENCLAVE_TRUST_ERROR;

    // check the enclave isn't loaded in enclave debug mode, except that the
project is built for debug purpose
#if defined(NDEBUG)
    if (peer_enclave_identity->attributes.flags & SGX_FLAGS_DEBUG)
        return ENCLAVE_TRUST_ERROR;
#endif

    return SUCCESS;
}

/* Function Description: Operates on the input secret and generate the output
secret
 * */
uint32_t get_message_exchange_response(uint32_t inp_secret_data)
{
    uint32_t secret_response;

    //User should use more complex encryption method to protect their secret,
below is just a simple example
    secret_response = inp_secret_data & 0x11111111;

    return secret_response;

}

//Generates the response from the request message
/* Function Description:
 *   process request message and generate response
 * Parameter Description:
 *   [input] decrypted_data: this is pointer to decrypted message
 *   [output] resp_buffer: this is pointer to response message, the buffer is
allocated inside this function
 *   [output] resp_length: this points to response length
 * */
```

```
extern "C" uint32_t message_exchange_response_generator(char* decrypted_data,
                                                        char** resp_buffer,
                                                        size_t* resp_length)
{
    ms_in_msg_exchange_t *ms;
    uint32_t inp_secret_data;
    uint32_t out_secret_data;
    if(!decrypted_data || !resp_length)
    {
        return INVALID_PARAMETER_ERROR;
    }
    ms = (ms_in_msg_exchange_t *)decrypted_data;

    if(umarshal_message_exchange_request(&inp_secret_data,ms) != SUCCESS)
        return ATTESTATION_ERROR;

    out_secret_data = get_message_exchange_response(inp_secret_data);

    if(marshal_message_exchange_response(resp_buffer, resp_length,
out_secret_data) != SUCCESS)
        return MALLOC_ERROR;

    return SUCCESS;
}
```

EnclaveMessageExchange.cpp

```
#include "sgx_trts.h"
#include "sgx_utils.h"
#include "EnclaveMessageExchange.h"
#include "sgx_eid.h"
#include "error_codes.h"
#include "sgx_ecp_types.h"
#include "sgx_thread.h"
#include <map>
#include "dh_session_protocol.h"
#include "sgx_dh.h"
#include "sgx_tcrypto.h"
#include "../EnclaveInitiator/EnclaveInitiator_t.h"
//#include "LocalAttestationCode_t.h"

#ifdef __cplusplus
extern "C" {
#endif

uint32_t message_exchange_response_generator(char *decrypted_data, char
**resp_buffer, size_t *resp_length);
uint32_t verify_peer_enclave_trust(sgx_dh_session_enclave_identity_t
*peer_enclave_identity);

#ifdef __cplusplus
}
#endif

#define MAX_SESSION_COUNT  16
```

```
//number of open sessions
uint32_t g_session_count = 0;

uint32_t generate_session_id(uint32_t *session_id);

uint32_t end_session(sgx_enclave_id_t src_enclave_id);

//Array of open session ids
session_id_tracker_t *g_session_id_tracker[MAX_SESSION_COUNT];

//Map between the source enclave id and the session information associated with
that particular session
std::map<sgx_enclave_id_t, dh_session_t> g_dest_session_info_map;

//Create a session with the destination enclave
uint32_t create_session(dh_session_t *session_info) {
    sgx_dh_msg1_t dh_msg1;            //Diffie-Hellman Message 1
    sgx_key_128bit_t dh_aek;        // Session Key
    sgx_dh_msg2_t dh_msg2;            //Diffie-Hellman Message 2
    sgx_dh_msg3_t dh_msg3;            //Diffie-Hellman Message 3
    uint32_t session_id;
    uint32_t retstatus;
    sgx_status_t status = SGX_SUCCESS;
    sgx_dh_session_t sgx_dh_session;
    sgx_dh_session_enclave_identity_t responder_identity;

    if (!session_info) {
        return INVALID_PARAMETER_ERROR;
    }

    memset(&dh_aek, 0, sizeof(sgx_key_128bit_t));
    memset(&dh_msg1, 0, sizeof(sgx_dh_msg1_t));
    memset(&dh_msg2, 0, sizeof(sgx_dh_msg2_t));
    memset(&dh_msg3, 0, sizeof(sgx_dh_msg3_t));
    memset(session_info, 0, sizeof(dh_session_t));

    //Intialize the session as a session initiator
    status = sgx_dh_init_session(SGX_DH_SESSION_INITIATOR, &sgx_dh_session);
    if (SGX_SUCCESS != status) {
        return status;
    }

    //Ocall to request for a session with the destination enclave and obtain
session id and Message 1 if successful
    status = session_request_ocall(&retstatus, &dh_msg1, &session_id);
    if (status == SGX_SUCCESS) {
        if ((uint32_t) retstatus != SUCCESS)
            return ((uint32_t) retstatus);
    } else {
        return ATTESTATION_SE_ERROR;
    }
    //Process the message 1 obtained from desination enclave and generate message
2
    status = sgx_dh_initiator_proc_msg1(&dh_msg1, &dh_msg2, &sgx_dh_session);
    if (SGX_SUCCESS != status) {
        return status;
    }
```

```
    //Send Message 2 to Destination Enclave and get Message 3 in return
    status = exchange_report_ocall(&retstatus, &dh_msg2, &dh_msg3, session_id);
    if (status == SGX_SUCCESS) {
        if ((uint32_t) retstatus != SUCCESS)
            return ((uint32_t) retstatus);
    } else {
        return ATTESTATION_SE_ERROR;
    }

    //Process Message 3 obtained from the destination enclave
    status = sgx_dh_initiator_proc_msg3(&dh_msg3, &sgx_dh_session, &dh_aek,
&responder_identity);
    if (SGX_SUCCESS != status) {
        return status;
    }

    // Verify the identity of the destination enclave
    if (verify_peer_enclave_trust(&responder_identity) != SUCCESS) {
        return INVALID_SESSION;
    }

    memcpy(session_info->active.AEK, &dh_aek, sizeof(sgx_key_128bit_t));
    session_info->session_id = session_id;
    session_info->active.counter = 0;
    session_info->status = ACTIVE;
    memset(&dh_aek, 0, sizeof(sgx_key_128bit_t));
    return status;
}

//Request for the response size, send the request message to the destination
enclave and receive the response message back
uint32_t send_request_receive_response(dh_session_t *session_info,
                                       char *inp_buff,
                                       size_t inp_buff_len,
                                       size_t max_out_buff_size,
                                       char **out_buff,
                                       size_t *out_buff_len) {
    const uint8_t *plaintext;
    uint32_t plaintext_length;
    sgx_status_t status;
    uint32_t retstatus;
    secure_message_t *req_message;
    secure_message_t *resp_message;
    uint8_t *decrypted_data;
    uint32_t decrypted_data_length;
    uint32_t plain_text_offset;
    uint8_t l_tag[TAG_SIZE];
    size_t max_resp_message_length;
    plaintext = (const uint8_t *) (" ");
    plaintext_length = 0;

    if (!session_info || !inp_buff) {
        return INVALID_PARAMETER_ERROR;
    }
    //Check if the nonce for the session has not exceeded 2^32-2 if so end
session and start a new session
    if (session_info->active.counter == ((uint32_t) -2)) {
        close_session(session_info);
```

```
        create_session(session_info);
    }

    //Allocate memory for the AES-GCM request message
    req_message = (secure_message_t *) malloc(sizeof(secure_message_t) +
inp_buff_len);
    if (!req_message)
        return MALLOC_ERROR;
    memset(req_message, 0, sizeof(secure_message_t) + inp_buff_len);

    const uint32_t data2encrypt_length = (uint32_t) inp_buff_len;

    //Set the payload size to data to encrypt length
    req_message->message_aes_gcm_data.payload_size = data2encrypt_length;

    //Use the session nonce as the payload IV
    memcpy(req_message->message_aes_gcm_data.reserved, &session_info-
>active.counter,
           sizeof(session_info->active.counter));

    //Set the session ID of the message to the current session id
    req_message->session_id = session_info->session_id;

    //Prepare the request message with the encrypted payload
    status = sgx_rijndael128GCM_encrypt(&session_info->active.AEK, (uint8_t *)
inp_buff, data2encrypt_length,
                                        reinterpret_cast<uint8_t *>(&
(req_message->message_aes_gcm_data.payload)),
                                        reinterpret_cast<uint8_t *>(&
(req_message->message_aes_gcm_data.reserved)),
                                        sizeof(req_message-
>message_aes_gcm_data.reserved), plaintext, plaintext_length,
                                        &(req_message-
>message_aes_gcm_data.payload_tag));

    if (SGX_SUCCESS != status) {
        SAFE_FREE(req_message);
        return status;
    }

    //Allocate memory for the response payload to be copied
    *out_buff = (char *) malloc(max_out_buff_size);
    if (!*out_buff) {
        SAFE_FREE(req_message);
        return MALLOC_ERROR;
    }
    memset(*out_buff, 0, max_out_buff_size);

    //Allocate memory for the response message
    resp_message = (secure_message_t *) malloc(sizeof(secure_message_t) +
max_out_buff_size);
    if (!resp_message) {
        SAFE_FREE(req_message);
        return MALLOC_ERROR;
    }

    memset(resp_message, 0, sizeof(secure_message_t) + max_out_buff_size);
```

```
    //Ocall to send the request to the Destination Enclave and get the response
message back
    status = send_request_ocall(&retstatus, session_info->session_id,
req_message,
                                (sizeof(secure_message_t) + inp_buff_len),
max_out_buff_size,
                                resp_message, (sizeof(secure_message_t) +
max_out_buff_size));
    if (status == SGX_SUCCESS) {
        if ((uint32_t) retstatus != SUCCESS) {
            SAFE_FREE(req_message);
            SAFE_FREE(resp_message);
            return ((uint32_t) retstatus);
        }
    } else {
        SAFE_FREE(req_message);
        SAFE_FREE(resp_message);
        return ATTESTATION_SE_ERROR;
    }

    max_resp_message_length = sizeof(secure_message_t) + max_out_buff_size;

    if (sizeof(resp_message) > max_resp_message_length) {
        SAFE_FREE(req_message);
        SAFE_FREE(resp_message);
        return INVALID_PARAMETER_ERROR;
    }

    //Code to process the response message from the Destination Enclave

    decrypted_data_length = resp_message->message_aes_gcm_data.payload_size;
    plain_text_offset = decrypted_data_length;
    decrypted_data = (uint8_t *) malloc(decrypted_data_length);
    if (!decrypted_data) {
        SAFE_FREE(req_message);
        SAFE_FREE(resp_message);
        return MALLOC_ERROR;
    }
    memset(&l_tag, 0, 16);

    memset(decrypted_data, 0, decrypted_data_length);

    //Decrypt the response message payload
    status = sgx_rijndael128GCM_decrypt(&session_info->active.AEK, resp_message-
>message_aes_gcm_data.payload,
                                        decrypted_data_length, decrypted_data,
                                        reinterpret_cast<uint8_t *>(&
(resp_message->message_aes_gcm_data.reserved)),
                                        sizeof(resp_message-
>message_aes_gcm_data.reserved),
                                        &(resp_message-
>message_aes_gcm_data.payload[plain_text_offset]),
                                        plaintext_length,
                                        &resp_message-
>message_aes_gcm_data.payload_tag);

    if (SGX_SUCCESS != status) {
        SAFE_FREE(req_message);
```

```
        SAFE_FREE(decrypted_data);
        SAFE_FREE(resp_message);
        return status;
    }

    // Verify if the nonce obtained in the response is equal to the session nonce
+ 1 (Prevents replay attacks)
    if (*((uint32_t *) resp_message->message_aes_gcm_data.reserved) !=
(session_info->active.counter + 1)) {
        SAFE_FREE(req_message);
        SAFE_FREE(resp_message);
        SAFE_FREE(decrypted_data);
        return INVALID_PARAMETER_ERROR;
    }

    //Update the value of the session nonce in the source enclave
    session_info->active.counter = session_info->active.counter + 1;

    memcpy(out_buff_len, &decrypted_data_length, sizeof(decrypted_data_length));
    memcpy(*out_buff, decrypted_data, decrypted_data_length);

    SAFE_FREE(decrypted_data);
    SAFE_FREE(req_message);
    SAFE_FREE(resp_message);
    return SUCCESS;
}

//Close a current session
uint32_t close_session(dh_session_t *session_info) {
    sgx_status_t status;
    uint32_t retstatus;

    if (!session_info) {
        return INVALID_PARAMETER_ERROR;
    }

    //Ocall to ask the destination enclave to end the session
    status = end_session_ocall(&retstatus, session_info->session_id);
    if (status == SGX_SUCCESS) {
        if ((uint32_t) retstatus != SUCCESS)
            return ((uint32_t) retstatus);
    } else {
        return ATTESTATION_SE_ERROR;
    }
    return SUCCESS;
}

//Returns a new sessionID for the source destination session
uint32_t generate_session_id(uint32_t *session_id) {
    uint32_t status = SUCCESS;

    if (!session_id) {
        return INVALID_PARAMETER_ERROR;
    }
    //if the session structure is uninitialized, set that as the next session ID
    for (int i = 0; i < MAX_SESSION_COUNT; i++) {
        if (g_session_id_tracker[i] == NULL) {
            *session_id = i;
```

```
            return status;
        }
    }

    status = NO_AVAILABLE_SESSION_ERROR;

    return status;

}
```

EnclaveResponder (e2)

EnclaveResponder.edl

```
enclave {
    include "sgx_eid.h"
    include "datatypes.h"
    include "../Include/dh_session_protocol.h"
    trusted{
            public uint32_t session_request_ecall([out] sgx_dh_msg1_t *dh_msg1,
[out] uint32_t *session_id);
            public uint32_t exchange_report_ecall([in] sgx_dh_msg2_t *dh_msg2,
[out] sgx_dh_msg3_t *dh_msg3, uint32_t session_id);
            public uint32_t generate_response_ecall([in, size = req_message_size]
secure_message_t* req_message, size_t req_message_size, size_t max_payload_size,
[out, size=resp_message_size] secure_message_t* resp_message, size_t
resp_message_size, uint32_t session_id);
            public uint32_t end_session_ecall(uint32_t session_id);
    };
};
```

EnclaveResponder.cpp

```
// Enclave2 : Defines the exported functions for the DLL application
#include "sgx_eid.h"
#include "EnclaveResponder_t.h"
#include "EnclaveMessageExchange.h"
#include "error_codes.h"
#include "Utility_E2.h"
#include "sgx_dh.h"
#include "sgx_utils.h"
#include <map>

#define UNUSED(val) (void)(val)

std::map<sgx_enclave_id_t, dh_session_t>g_src_session_info_map;

// this is expected initiator's MRSIGNER for demonstration purpose
sgx_measurement_t g_initiator_mrsigner = {
        {
```

```
                0xc3, 0x04, 0x46, 0xb4, 0xbe, 0x9b, 0xaf, 0x0f, 0x69, 0x72, 0x84,
0x23, 0xea, 0x61, 0x3e, 0xf8,
                0x1a, 0x63, 0xe7, 0x2a, 0xcf, 0x74, 0x39, 0xfa, 0x05, 0x49, 0x00,
0x1f, 0xd5, 0x48, 0x28, 0x35
        }
};

/* Function Description:
 *   this is to verify peer enclave's identity
 * For demonstration purpose, we verify below points:
 *   1. peer enclave's MRSIGNER is as expected
 *   2. peer enclave's PROD_ID is as expected
 *   3. peer enclave's attribute is reasonable that it should be INITIALIZED and
without DEBUG attribute (except the project is built with DEBUG option)
 * */
extern "C" uint32_t verify_peer_enclave_trust(sgx_dh_session_enclave_identity_t*
peer_enclave_identity)
{
    if(!peer_enclave_identity)
        return INVALID_PARAMETER_ERROR;

    // check peer enclave's MRSIGNER
    if (memcmp((uint8_t *)&peer_enclave_identity->mr_signer,
(uint8_t*)&g_initiator_mrsigner, sizeof(sgx_measurement_t)))
        return ENCLAVE_TRUST_ERROR;

    if(peer_enclave_identity->isv_prod_id != 0 || !(peer_enclave_identity-
>attributes.flags & SGX_FLAGS_INITTED))
        return ENCLAVE_TRUST_ERROR;

    // check the enclave isn't loaded in enclave debug mode, except that the
project is built for debug purpose
#if defined(NDEBUG)
    if (peer_enclave_identity->attributes.flags & SGX_FLAGS_DEBUG)
        return ENCLAVE_TRUST_ERROR;
#endif

    return SUCCESS;
}

/* Function Description: Operates on the input secret and generates the output
secret */
uint32_t get_message_exchange_response(uint32_t inp_secret_data)
{
    uint32_t secret_response;

    //User should use more complex encryption method to protect their secret,
below is just a simple example
    secret_response = inp_secret_data & 0x11111111;

    return secret_response;

}

/* Function Description: Generates the response from the request message
 * Parameter Description:
 * [input] decrtyped_data: pointer to decrypted data
```

```
 * [output] resp_buffer: pointer to response message, which is allocated in this
function
 * [output] resp_length: this is response length */
extern "C" uint32_t message_exchange_response_generator(char* decrypted_data,
                                                        char** resp_buffer,
                                                         size_t* resp_length)
{
    ms_in_msg_exchange_t *ms;
    uint32_t inp_secret_data;
    uint32_t out_secret_data;

    if(!decrypted_data || !resp_length)
        return INVALID_PARAMETER_ERROR;

    ms = (ms_in_msg_exchange_t *)decrypted_data;

    if(umarshal_message_exchange_request(&inp_secret_data,ms) != SUCCESS)
        return ATTESTATION_ERROR;

    out_secret_data = get_message_exchange_response(inp_secret_data);

    if(marshal_message_exchange_response(resp_buffer, resp_length,
out_secret_data) != SUCCESS)
        return MALLOC_ERROR;

    return SUCCESS;
}
```

EnclaveMessageExchange.cpp

```
#include "sgx_trts.h"
#include "sgx_utils.h"
#include "EnclaveMessageExchange.h"
#include "sgx_eid.h"
#include "error_codes.h"
#include "sgx_ecp_types.h"
#include "sgx_thread.h"
#include <map>
#include "dh_session_protocol.h"
#include "sgx_dh.h"
#include "sgx_tcrypto.h"

#ifdef __cplusplus
extern "C" {
#endif

uint32_t enclave_to_enclave_call_dispatcher(char *decrypted_data, size_t
decrypted_data_length, char **resp_buffer,
                                            size_t *resp_length);
uint32_t message_exchange_response_generator(char *decrypted_data, char
**resp_buffer, size_t *resp_length);
uint32_t verify_peer_enclave_trust(sgx_dh_session_enclave_identity_t
*peer_enclave_identity);
```

```
#ifdef __cplusplus
}
#endif

#define MAX_SESSION_COUNT  16

//number of open sessions
uint32_t g_session_count = 0;

uint32_t generate_session_id(uint32_t *session_id);

extern "C" uint32_t end_session_ecall(uint32_t session_id);

//Array of open session ids
session_id_tracker_t *g_session_id_tracker[MAX_SESSION_COUNT];

//Map between the session id and the session information associated with that
particular session
std::map<uint32_t, dh_session_t> g_dest_session_info_map;

//Create a session with the destination enclave

//Handle the request from Source Enclave for a session
extern "C" uint32_t session_request_ecall(sgx_dh_msg1_t *dh_msg1,
                                          uint32_t *session_id) {
    dh_session_t session_info;
    sgx_dh_session_t sgx_dh_session;
    sgx_status_t status = SGX_SUCCESS;

    if (!session_id || !dh_msg1) {
        return INVALID_PARAMETER_ERROR;
    }
    //Intialize the session as a session responder
    status = sgx_dh_init_session(SGX_DH_SESSION_RESPONDER, &sgx_dh_session);
    if (SGX_SUCCESS != status) {
        return status;
    }

    //get a new SessionID
    if ((status = (sgx_status_t) generate_session_id(session_id)) != SUCCESS)
        return status; //no more sessions available

    //Allocate memory for the session id tracker
    g_session_id_tracker[*session_id] = (session_id_tracker_t *)
malloc(sizeof(session_id_tracker_t));
    if (!g_session_id_tracker[*session_id]) {
        return MALLOC_ERROR;
    }

    memset(g_session_id_tracker[*session_id], 0, sizeof(session_id_tracker_t));
    g_session_id_tracker[*session_id]->session_id = *session_id;
    session_info.status = IN_PROGRESS;

    //Generate Message1 that will be returned to Source Enclave
    status = sgx_dh_responder_gen_msg1((sgx_dh_msg1_t *) dh_msg1,
&sgx_dh_session);
    if (SGX_SUCCESS != status) {
        SAFE_FREE(g_session_id_tracker[*session_id]);
```

```
        return status;
    }
    memcpy(&session_info.in_progress.dh_session, &sgx_dh_session,
sizeof(sgx_dh_session_t));
    //Store the session information under the corresponding source enclave id key
    g_dest_session_info_map.insert(std::pair<uint32_t, dh_session_t>(*session_id,
session_info));

    return status;
}

//Verify Message 2, generate Message3 and exchange Message 3 with Source Enclave
extern "C" uint32_t exchange_report_ecall(sgx_dh_msg2_t *dh_msg2,
                                          sgx_dh_msg3_t *dh_msg3,
                                          uint32_t session_id) {

    sgx_key_128bit_t dh_aek;   // Session key
    dh_session_t *session_info;
    uint32_t status = SUCCESS;
    sgx_dh_session_t sgx_dh_session;
    sgx_dh_session_enclave_identity_t initiator_identity;

    if (!dh_msg2 || !dh_msg3) {
        return INVALID_PARAMETER_ERROR;
    }

    memset(&dh_aek, 0, sizeof(sgx_key_128bit_t));
    do {
        //Retrieve the session information for the corresponding source enclave
id
        std::map<uint32_t, dh_session_t>::iterator it =
g_dest_session_info_map.find(session_id);
        if (it != g_dest_session_info_map.end()) {
            session_info = &it->second;
        } else {
            status = INVALID_SESSION;
            break;
        }

        if (session_info->status != IN_PROGRESS) {
            status = INVALID_SESSION;
            break;
        }

        memcpy(&sgx_dh_session, &session_info->in_progress.dh_session,
sizeof(sgx_dh_session_t));

        dh_msg3->msg3_body.additional_prop_length = 0;
        //Process message 2 from source enclave and obtain message 3
        sgx_status_t se_ret = sgx_dh_responder_proc_msg2(dh_msg2,
                                                         dh_msg3,
                                                         &sgx_dh_session,
                                                         &dh_aek,
                                                         &initiator_identity);
        if (SGX_SUCCESS != se_ret) {
            status = se_ret;
            break;
        }
```

```
        //Verify source enclave's trust
        if (verify_peer_enclave_trust(&initiator_identity) != SUCCESS) {
            return INVALID_SESSION;
        }

        //save the session ID, status and initialize the session nonce
        session_info->session_id = session_id;
        session_info->status = ACTIVE;
        session_info->active.counter = 0;
        memcpy(session_info->active.AEK, &dh_aek, sizeof(sgx_key_128bit_t));
        memset(&dh_aek, 0, sizeof(sgx_key_128bit_t));
        g_session_count++;
    } while (0);

    if (status != SUCCESS) {
        end_session_ecall(session_id);
    }

    return status;
}

//Process the request from the Source enclave and send the response message back
to the Source enclave
extern "C" uint32_t generate_response_ecall(secure_message_t *req_message,
                                            size_t req_message_size,
                                            size_t max_payload_size,
                                            secure_message_t *resp_message,
                                            size_t resp_message_size,
                                            uint32_t session_id) {
    const uint8_t *plaintext;
    uint32_t plaintext_length;
    uint8_t *decrypted_data;
    uint32_t decrypted_data_length;
    uint32_t plain_text_offset;
    ms_in_msg_exchange_t *ms;
    size_t resp_data_length;
    size_t resp_message_calc_size;
    char *resp_data;
    uint8_t l_tag[TAG_SIZE];
    size_t header_size, expected_payload_size;
    dh_session_t *session_info;
    secure_message_t *temp_resp_message;
    uint32_t ret;
    sgx_status_t status;

    plaintext = (const uint8_t *) (" ");
    plaintext_length = 0;

    if (!req_message || !resp_message) {
        return INVALID_PARAMETER_ERROR;
    }

    //Get the session information from the map corresponding to the source
enclave id
    std::map<uint32_t, dh_session_t>::iterator it =
g_dest_session_info_map.find(session_id);
    if (it != g_dest_session_info_map.end()) {
```

```
        session_info = &it->second;
    } else {
        return INVALID_SESSION;
    }

    if (session_info->status != ACTIVE) {
        return INVALID_SESSION;
    }

    //Set the decrypted data length to the payload size obtained from the message
    decrypted_data_length = req_message->message_aes_gcm_data.payload_size;

    header_size = sizeof(secure_message_t);
    expected_payload_size = req_message_size - header_size;

    //Verify the size of the payload
    if (expected_payload_size != decrypted_data_length)
        return INVALID_PARAMETER_ERROR;

    memset(&l_tag, 0, 16);
    plain_text_offset = decrypted_data_length;
    decrypted_data = (uint8_t *) malloc(decrypted_data_length);
    if (!decrypted_data) {
        return MALLOC_ERROR;
    }

    memset(decrypted_data, 0, decrypted_data_length);

    //Decrypt the request message payload from source enclave
    status = sgx_rijndael128GCM_decrypt(&session_info->active.AEK, req_message-
>message_aes_gcm_data.payload,
                                        decrypted_data_length, decrypted_data,
                                        reinterpret_cast<uint8_t *>(&
(req_message->message_aes_gcm_data.reserved)),
                                        sizeof(req_message-
>message_aes_gcm_data.reserved),
                                        &(req_message-
>message_aes_gcm_data.payload[plain_text_offset]),
                                        plaintext_length,
                                        &req_message-
>message_aes_gcm_data.payload_tag);

    if (SGX_SUCCESS != status) {
        SAFE_FREE(decrypted_data);
        return status;
    }

    //Casting the decrypted data to the marshaling structure type to obtain type
of request (generic message exchange/enclave to enclave call)
    ms = (ms_in_msg_exchange_t *) decrypted_data;

    // Verify if the nonce obtained in the request is equal to the session nonce
    if (*((uint32_t *) req_message->message_aes_gcm_data.reserved) !=
session_info->active.counter ||
        *((uint32_t *) req_message->message_aes_gcm_data.reserved) > ((uint32_t)
-2)) {
        SAFE_FREE(decrypted_data);
        return INVALID_PARAMETER_ERROR;
```

```
    }

    if (ms->msg_type == MESSAGE_EXCHANGE) {
        //Call the generic secret response generator for message exchange
        ret = message_exchange_response_generator((char *) decrypted_data,
&resp_data, &resp_data_length);
        if (ret != 0) {
            SAFE_FREE(decrypted_data);
            SAFE_FREE(resp_data);
            return INVALID_SESSION;
        }
    } else {
        SAFE_FREE(decrypted_data);
        return INVALID_REQUEST_TYPE_ERROR;
    }


    if (resp_data_length > max_payload_size) {
        SAFE_FREE(resp_data);
        SAFE_FREE(decrypted_data);
        return OUT_BUFFER_LENGTH_ERROR;
    }

    resp_message_calc_size = sizeof(secure_message_t) + resp_data_length;

    if (resp_message_calc_size > resp_message_size) {
        SAFE_FREE(resp_data);
        SAFE_FREE(decrypted_data);
        return OUT_BUFFER_LENGTH_ERROR;
    }

    //Code to build the response back to the Source Enclave
    temp_resp_message = (secure_message_t *) malloc(resp_message_calc_size);
    if (!temp_resp_message) {
        SAFE_FREE(resp_data);
        SAFE_FREE(decrypted_data);
        return MALLOC_ERROR;
    }

    memset(temp_resp_message, 0, sizeof(secure_message_t) + resp_data_length);
    const uint32_t data2encrypt_length = (uint32_t) resp_data_length;
    temp_resp_message->session_id = session_info->session_id;
    temp_resp_message->message_aes_gcm_data.payload_size = data2encrypt_length;

    //Increment the Session Nonce (Replay Protection)
    session_info->active.counter = session_info->active.counter + 1;

    //Set the response nonce as the session nonce
    memcpy(&temp_resp_message->message_aes_gcm_data.reserved, &session_info-
>active.counter,
           sizeof(session_info->active.counter));

    //Prepare the response message with the encrypted payload
    status = sgx_rijndael128GCM_encrypt(&session_info->active.AEK, (uint8_t *)
resp_data, data2encrypt_length,
                                        reinterpret_cast<uint8_t *>(&
(temp_resp_message->message_aes_gcm_data.payload)),
```

```
                                        reinterpret_cast<uint8_t *>(&
(temp_resp_message->message_aes_gcm_data.reserved)),
                                        sizeof(temp_resp_message-
>message_aes_gcm_data.reserved), plaintext,
                                        plaintext_length,
                                        &(temp_resp_message-
>message_aes_gcm_data.payload_tag));

    if (SGX_SUCCESS != status) {
        SAFE_FREE(resp_data);
        SAFE_FREE(decrypted_data);
        SAFE_FREE(temp_resp_message);
        return status;
    }

    memset(resp_message, 0, sizeof(secure_message_t) + resp_data_length);
    memcpy(resp_message, temp_resp_message, sizeof(secure_message_t) +
resp_data_length);

    SAFE_FREE(decrypted_data);
    SAFE_FREE(resp_data);
    SAFE_FREE(temp_resp_message);

    return SUCCESS;
}


//Respond to the request from the Source Enclave to close the session
extern "C" uint32_t end_session_ecall(uint32_t session_id) {
    uint32_t status = SUCCESS;
    int i;
    dh_session_t session_info;
    //uint32_t session_id;

    //Get the session information from the map corresponding to the source
enclave id
    std::map<uint32_t, dh_session_t>::iterator it =
g_dest_session_info_map.find(session_id);
    if (it != g_dest_session_info_map.end()) {
        session_info = it->second;
    } else {
        return INVALID_SESSION;
    }

    //session_id = session_info.session_id;
    //Erase the session information for the current session
    g_dest_session_info_map.erase(session_id);

    //Update the session id tracker
    if (g_session_count > 0) {
        //check if session exists
        for (i = 1; i <= MAX_SESSION_COUNT; i++) {
            if (g_session_id_tracker[i - 1] != NULL && g_session_id_tracker[i -
1]->session_id == session_id) {
                memset(g_session_id_tracker[i - 1], 0,
sizeof(session_id_tracker_t));
                SAFE_FREE(g_session_id_tracker[i - 1]);
                g_session_count--;
```

```
                break;
            }
        }
    }

    return status;

}


//Returns a new sessionID for the source destination session
uint32_t generate_session_id(uint32_t *session_id) {
    uint32_t status = SUCCESS;

    if (!session_id) {
        return INVALID_PARAMETER_ERROR;
    }
    //if the session structure is uninitialized, set that as the next session ID
    for (int i = 0; i < MAX_SESSION_COUNT; i++) {
        if (g_session_id_tracker[i] == NULL) {
            *session_id = i;
            return status;
        }
    }

    status = NO_AVAILABLE_SESSION_ERROR;

    return status;

}
```

附录四：Remote Attestation (server) 代码实现

App

app.cpp

```
#include <stdio.h>
#include <limits.h>
#include <unistd.h>
// Needed for definition of remote attestation messages.
#include "remote_attestation_result.h"

#include "isv_enclave_u.h"

// Needed to call untrusted key exchange library APIs, i.e. sgx_ra_proc_msg2.
#include "sgx_ukey_exchange.h"

// Needed to get service provider's information, in your real project, you will
// need to talk to real server.
#include "network_ra.h"

// Needed to create enclave and do ecall.
#include "sgx_urts.h"

// Needed to query extended epid group id.
#include "sgx_uae_service.h"

#include "service_provider.h"

#ifndef SAFE_FREE
#define SAFE_FREE(ptr)     \
    {                      \
        if (NULL != (ptr)) \
        {                  \
            free(ptr);     \
            (ptr) = NULL;  \
        }                  \
    }
#endif

// In addition to generating and sending messages, this application
// can use pre-generated messages to verify the generation of
// messages and the information flow.
#include "sample_messages.h"

#define ENCLAVE_PATH "isv_enclave.signed.so"

#define LENOFMSE 16

uint8_t *msg1_samples[] = {msg1_sample1, msg1_sample2};
uint8_t *msg2_samples[] = {msg2_sample1, msg2_sample2};
```

```
uint8_t *msg3_samples[] = {msg3_sample1, msg3_sample2};
uint8_t *attestation_msg_samples[] =
    {attestation_msg_sample1, attestation_msg_sample2};

// Some utility functions to output some of the data structures passed between
// the ISV app and the remote attestation service provider.
void PRINT_BYTE_ARRAY(
    FILE *file, void *mem, uint32_t len)
{
    if (!mem || !len)
    {
        fprintf(file, "\n( null )\n");
        return;
    }
    uint8_t *array = (uint8_t *)mem;
    fprintf(file, "%u bytes:\n{\n", len);
    uint32_t i = 0;
    for (i = 0; i < len - 1; i++)
    {
        fprintf(file, "0x%x, ", array[i]);
        if (i % 8 == 7)
            fprintf(file, "\n");
    }
    fprintf(file, "0x%x ", array[i]);
    fprintf(file, "\n}\n");
}

void PRINT_ATTESTATION_SERVICE_RESPONSE(
    FILE *file,
    ra_samp_response_header_t *response)
{
    if (!response)
    {
        fprintf(file, "\t\n( null )\n");
        return;
    }

    fprintf(file, "RESPONSE TYPE:   0x%x\n", response->type);
    fprintf(file, "RESPONSE STATUS: 0x%x 0x%x\n", response->status[0],
            response->status[1]);
    fprintf(file, "RESPONSE BODY SIZE: %u\n", response->size);

    if (response->type == TYPE_RA_MSG2)
    {
        sgx_ra_msg2_t *p_msg2_body = (sgx_ra_msg2_t *)(response->body);

        fprintf(file, "MSG2 gb - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->g_b), sizeof(p_msg2_body->g_b));

        fprintf(file, "MSG2 spid - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->spid), sizeof(p_msg2_body->spid));

        fprintf(file, "MSG2 quote_type : %hx\n", p_msg2_body->quote_type);

        fprintf(file, "MSG2 kdf_id : %hx\n", p_msg2_body->kdf_id);

        fprintf(file, "MSG2 sign_gb_ga - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->sign_gb_ga),
```

```
                        sizeof(p_msg2_body->sign_gb_ga));

        fprintf(file, "MSG2 mac - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->mac), sizeof(p_msg2_body->mac));

        fprintf(file, "MSG2 sig_rl - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->sig_rl),
                        p_msg2_body->sig_rl_size);
    }
    else if (response->type == TYPE_RA_ATT_RESULT)
    {
        sample_ra_att_result_msg_t *p_att_result =
            (sample_ra_att_result_msg_t *)(response->body);
        fprintf(file, "ATTESTATION RESULT MSG platform_info_blob - ");
        PRINT_BYTE_ARRAY(file, &(p_att_result->platform_info_blob),
                        sizeof(p_att_result->platform_info_blob));

        fprintf(file, "ATTESTATION RESULT MSG mac - ");
        PRINT_BYTE_ARRAY(file, &(p_att_result->mac), sizeof(p_att_result->mac));

        fprintf(file, "ATTESTATION RESULT MSG secret.payload_tag - %u bytes\n",
                p_att_result->secret.payload_size);

        fprintf(file, "ATTESTATION RESULT MSG secret.payload - ");
        PRINT_BYTE_ARRAY(file, p_att_result->secret.payload,
                        p_att_result->secret.payload_size);
    }
    else
    {
        fprintf(file, "\nERROR in printing out the response. "
                       "Response of type not supported %d\n",
                response->type);
    }
}

extern char sendbuf[BUFSIZ]; //数据传送的缓冲区
extern char recvbuf[BUFSIZ]; //数据接受的缓冲区

int myaesencrypt(const ra_samp_request_header_t *p_msgenc,
                 uint32_t msg_size,
                 sgx_enclave_id_t id,
                 sgx_status_t *status,
                 sgx_ra_context_t context)
{
    if (!p_msgenc||
        (msg_size != LENOFMSE))
    {
        return -1;
    }
    int ret = 0;

    int busy_retry_time = 4;
    uint8_t p_data[LENOFMSE] = {0};
    uint8_t out_data[LENOFMSE] = {0};
    uint8_t testdata[LENOFMSE] = {0};
    ra_samp_response_header_t *p_msg2_full = NULL;
    uint8_t msg2_size = 16; //只处理16字节的数据
```

```
    memcpy_s(p_data, LENOFMSE, p_msgenc, msg_size);
    do
    {
        ret = enclave_encrypt(
            id,
            status,
            p_data,
            LENOFMSE,
            out_data);
        fprintf(stdout, "\nD %d %d",id, *status);
        ret = enclave_encrypt(
            id,
            status,
            out_data,
            LENOFMSE,
            testdata);
        fprintf(stdout, "\nD %d %d",id, *status);

    } while (SGX_ERROR_BUSY == ret && busy_retry_time--);
    fprintf(stdout, "\nData of Encrypt is\n");
    PRINT_BYTE_ARRAY(stdout, p_data, 16);
    fprintf(stdout, "\nData of Encrypted is\n");
    PRINT_BYTE_ARRAY(stdout, out_data, 16);
    PRINT_BYTE_ARRAY(stdout, testdata, 16);
    p_msg2_full = (ra_samp_response_header_t *)malloc(msg2_size +
sizeof(ra_samp_response_header_t));
    if (!p_msg2_full)
    {
        fprintf(stderr, "\nError, out of memory in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    memset(p_msg2_full, 0, msg2_size + sizeof(ra_samp_response_header_t));
    p_msg2_full->type = TYPE_RA_MSGENC;
    p_msg2_full->size = msg2_size;
    p_msg2_full->status[0] = 0;
    p_msg2_full->status[1] = 0;

    if (memcpy_s(&p_msg2_full->body[0], msg2_size, &out_data[0], msg2_size))
    {
        fprintf(stderr, "\nError, memcpy failed in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    memset(sendbuf, 0, BUFSIZ);
    if (memcpy_s(sendbuf,
                 msg2_size + sizeof(ra_samp_response_header_t),
                 p_msg2_full,
                 msg2_size + sizeof(ra_samp_response_header_t)))
    {
        fprintf(stderr, "\nError, memcpy failed in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }

    if (SendToClient(msg2_size + sizeof(ra_samp_response_header_t)) < 0)
    {
```

```
        fprintf(stderr, "\nError, send encrypted data failed in [%s].",
__FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    SAFE_FREE(p_msg2_full);

    return ret;
}

//原本设计为32字节的消息长度，前16个字节是token，但是由于时间关系直接解密了
int myaesdecrypt(const ra_samp_request_header_t *p_msgenc,
                 uint32_t msg_size,
                 sgx_enclave_id_t id,
                 sgx_status_t *status,
                 sgx_ra_context_t context)
{
    if (!p_msgenc ||
        (msg_size != LENOFMSE))
    {
        return -1;
    }
    int ret = 0;
    fprintf(stdout, "\nD %d %d",id, *status);
    int busy_retry_time = 4;
    uint8_t p_data[LENOFMSE] = {0};
    uint8_t out_data[LENOFMSE] = {0};
    ra_samp_response_header_t *p_msg2_full = NULL;
    uint8_t msg2_size = 16; //只处理16字节的数据
    memcpy_s(p_data, LENOFMSE, p_msgenc, msg_size);
    do
    {
        ret = enclave_decrypt(
            id,
            status,
            p_data,
            LENOFMSE,
            out_data);
    } while (SGX_ERROR_BUSY == ret && busy_retry_time--);
    if(ret != SGX_SUCCESS)
        return ret;
    fprintf(stdout, "\nData of Decrypt is\n");
    PRINT_BYTE_ARRAY(stdout, p_data, 16);
    fprintf(stdout, "\nData of Decrypted is\n");
    PRINT_BYTE_ARRAY(stdout, out_data, 16);
    p_msg2_full = (ra_samp_response_header_t *)malloc(msg2_size +
sizeof(ra_samp_response_header_t));
    if (!p_msg2_full)
    {
        fprintf(stderr, "\nError, out of memory in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    memset(p_msg2_full, 0, msg2_size + sizeof(ra_samp_response_header_t));
    p_msg2_full->type = TYPE_RA_MSGDEC;
    p_msg2_full->size = msg2_size;
    // The simulated message2 always passes.  This would need to be set
    // accordingly in a real service provider implementation.
```

```
    p_msg2_full->status[0] = 0;
    p_msg2_full->status[1] = 0;

    if (memcpy_s(&p_msg2_full->body[0], msg2_size, &out_data[0], msg2_size))
    {
        fprintf(stderr, "\nError, memcpy failed in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    memset(sendbuf, 0, BUFSIZ);
    if (memcpy_s(sendbuf,
                 msg2_size + sizeof(ra_samp_response_header_t),
                 p_msg2_full,
                 msg2_size + sizeof(ra_samp_response_header_t)))
    {
        fprintf(stderr, "\nError, memcpy failed in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }

    if (SendToClient(msg2_size + sizeof(ra_samp_response_header_t)) < 0)
    {
        fprintf(stderr, "\nError, send encrypted data failed in [%s].",
__FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    SAFE_FREE(p_msg2_full);
    fprintf(stdout, "\nSend Decrypt Data Done.");
    return ret;
}

int myaessetkey(const ra_samp_request_header_t *p_msgdec,
                uint32_t msg_size,
                sgx_enclave_id_t id,
                sgx_status_t *status,
                sgx_ra_context_t context)
{
    if (!p_msgdec  ||
        (msg_size != LENOFMSE * 2))
    {
        return -1;
    }
    int ret = 0;
    int busy_retry_time = 4;
    uint8_t p_data[LENOFMSE * 2] = {0};
    uint8_t out_data[LENOFMSE] =
{'K','E','Y','S','E','T','S','U','C','C','E','S'};
    ra_samp_response_header_t *p_msg2_full = NULL;
    uint8_t msg2_size = 16; //只处理16字节的数据
    memcpy_s(p_data, msg_size, p_msgdec, msg_size);
    //应该调用isv_enclave_u.h中生成的函数
    do
    {
        ret = enclave_generate_key(
            id,
            status,
            p_data,
```

```
            msg_size);
    } while (SGX_ERROR_BUSY == ret && busy_retry_time--);
    if(ret != SGX_SUCCESS)
        return ret;
    p_msg2_full = (ra_samp_response_header_t *)malloc(msg2_size +
sizeof(ra_samp_response_header_t));
    if (!p_msg2_full)
    {
        fprintf(stderr, "\nError, out of memory in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    memset(p_msg2_full, 0, msg2_size + sizeof(ra_samp_response_header_t));
    p_msg2_full->type = TYPE_RA_MSGSETKEY;
    p_msg2_full->size = msg2_size;
    // The simulated message2 always passes.  This would need to be set
    // accordingly in a real service provider implementation.
    p_msg2_full->status[0] = 0;
    p_msg2_full->status[1] = 0;

    if (memcpy_s(&p_msg2_full->body[0], msg2_size, &out_data[0], msg2_size))
    {
        fprintf(stderr, "\nError, memcpy failed in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    memset(sendbuf, 0, BUFSIZ);
    if (memcpy_s(sendbuf,
                 msg2_size + sizeof(ra_samp_response_header_t),
                 p_msg2_full,
                 msg2_size + sizeof(ra_samp_response_header_t)))
    {
        fprintf(stderr, "\nError, memcpy failed in [%s].", __FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }

    if (SendToClient(msg2_size + sizeof(ra_samp_response_header_t)) < 0)
    {
        fprintf(stderr, "\nError, send encrypted data failed in [%s].",
__FUNCTION__);
        ret = SP_INTERNAL_ERROR;
        return ret;
    }
    SAFE_FREE(p_msg2_full);
    return ret;
}
// This sample code doesn't have any recovery/retry mechanisms for the remote
// attestation. Since the enclave can be lost due S3 transitions, apps
// susceptible to S3 transitions should have logic to restart attestation in
// these scenarios.
#define _T(x) x
int main(int argc, char *argv[])
{
    int ret = 0;
    ra_samp_request_header_t *p_msg0_full = NULL;
    ra_samp_response_header_t *p_msg0_resp_full = NULL;
    ra_samp_request_header_t *p_msg1_full = NULL;
```

```
    ra_samp_response_header_t *p_msg2_full = NULL;
    sgx_ra_msg3_t *p_msg3 = NULL;
    ra_samp_response_header_t *p_att_result_msg_full = NULL;
    sgx_enclave_id_t enclave_id = 0;
    int enclave_lost_retry_time = 1;
    int busy_retry_time = 4;
    sgx_ra_context_t context = INT_MAX;
    sgx_status_t status = SGX_SUCCESS;
    ra_samp_request_header_t *p_msg3_full = NULL;
    ra_samp_request_header_t *p_msgaes_full = NULL;

    int32_t verify_index = -1;
    int32_t verification_samples = sizeof(msg1_samples) /
sizeof(msg1_samples[0]);

    FILE *OUTPUT = stdout;
    ra_samp_request_header_t *p_req;
    ra_samp_response_header_t **p_resp;
    ra_samp_response_header_t *p_resp_msg;
    int server_port = 12333;
    int buflen = 0;
    uint32_t extended_epid_group_id = 0;
    { // creates the cryptserver enclave.

        ret = sgx_get_extended_epid_group_id(&extended_epid_group_id);
        if (SGX_SUCCESS != ret)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, call sgx_get_extended_epid_group_id fail
[%s].",
                    __FUNCTION__);
            return ret;
        }
        fprintf(OUTPUT, "\nCall sgx_get_extended_epid_group_id success.");

        int launch_token_update = 0;
        sgx_launch_token_t launch_token = {0};
        memset(&launch_token, 0, sizeof(sgx_launch_token_t));
        do
        {
            ret = sgx_create_enclave(_T(ENCLAVE_PATH),
                                     SGX_DEBUG_FLAG,
                                     &launch_token,
                                     &launch_token_update,
                                     &enclave_id, NULL);
            if (SGX_SUCCESS != ret)
            {
                ret = -1;
                fprintf(OUTPUT, "\nError, call sgx_create_enclave fail [%s].",
                        __FUNCTION__);
                goto CLEANUP;
            }
            fprintf(OUTPUT, "\nCall sgx_create_enclave success.");

            ret = enclave_init_ra(enclave_id,
                                  &status,
                                  false,
                                  &context);
```

```
            //Ideally, this check would be around the full attestation flow.
        } while (SGX_ERROR_ENCLAVE_LOST == ret && enclave_lost_retry_time--);

        if (SGX_SUCCESS != ret || status)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, call enclave_init_ra fail [%s].",
                    __FUNCTION__);
            goto CLEANUP;
        }
        fprintf(OUTPUT, "\nCall enclave_init_ra success.");
    }

    //服务进程，对接受的数据进行响应
    fprintf(OUTPUT, "\nstart socket....\n");
    server(server_port);

    //如果接受的信息类型为服务类型，就解析
    do
    {
        //阻塞调用socket
        buflen = RecvfromCient();
        if (buflen > 0 && buflen < BUFSIZ)
        {
            p_req = (ra_samp_request_header_t *)malloc(buflen+2);

            fprintf(OUTPUT, "\nPrepare receive struct");
            if (NULL == p_req)
            {
                ret = -1;
                goto CLEANUP;
            }
            if (memcpy_s(p_req, buflen+ 2, recvbuf, buflen))
            {
                fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in
[%s].",
                        __FUNCTION__);
                ret = -1;
                goto CLEANUP;
            }
            //todo: 添加一个检查p_req的函数，由于时间紧张，就先放一放
            fprintf(OUTPUT, "\nrequest type is %d",p_req->type);
            switch (p_req->type)
            {
            //收取msg0，进行验证
            case TYPE_RA_MSG0:
                fprintf(OUTPUT, "\nProcess Message 0");
                ret = sp_ra_proc_msg0_req((const sample_ra_msg0_t *)((uint8_t
*)p_req + sizeof(ra_samp_request_header_t)),
                                          p_req->size);
                fprintf(OUTPUT, "\nProcess Message 0 Done");
                if (0 != ret)
                {
                    fprintf(stderr, "\nError, call sp_ra_proc_msg1_req fail
[%s].",
                            __FUNCTION__);
                }
                SAFE_FREE(p_req);
```

```
                break;
            //收取msg1，进行验证并返回msg2
            case TYPE_RA_MSG1:
                fprintf(OUTPUT, "\nBuffer length is %d\n", buflen);
                p_resp_msg = (ra_samp_response_header_t
*)malloc(sizeof(ra_samp_response_header_t)+170);//简化处理
                memset(p_resp_msg, 0, sizeof(ra_samp_response_header_t)+170);
                fprintf(OUTPUT, "\nProcess Message 1\n");
                ret = sp_ra_proc_msg1_req((const sample_ra_msg1_t *)((uint8_t
*)p_req + sizeof(ra_samp_request_header_t)),
                                          p_req->size,
                                          &p_resp_msg);
                fprintf(OUTPUT, "\nProcess Message 1 Done");
                if (0 != ret)
                {
                    fprintf(stderr, "\nError, call sp_ra_proc_msg1_req fail
[%s].",
                            __FUNCTION__);
                }
                else
                {
                    memset(sendbuf, 0, BUFSIZ);
                    if (memcpy_s(sendbuf, BUFSIZ, p_resp_msg,
sizeof(ra_samp_response_header_t) + p_resp_msg->size))
                    {
                        fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed
in [%s].",
                                __FUNCTION__);
                        ret = -1;
                        goto CLEANUP;
                    }
                    fprintf(OUTPUT, "\nSend Message 2\n");
                    PRINT_BYTE_ARRAY(OUTPUT, p_resp_msg, 176);
                    int buflen = SendToClient(sizeof(ra_samp_response_header_t) +
p_resp_msg->size);
                    fprintf(OUTPUT, "\nSend Message 2 Done,send length = %d",
buflen);
                }
                SAFE_FREE(p_req);
                SAFE_FREE(p_resp_msg);
                break;
            //收取msg3，返回attestation result
            case TYPE_RA_MSG3:
                fprintf(OUTPUT, "\nProcess Message 3");
                p_resp_msg = (ra_samp_response_header_t
*)malloc(sizeof(ra_samp_response_header_t)+200);//简化处理
                memset(p_resp_msg, 0, sizeof(ra_samp_response_header_t)+200);
                ret = sp_ra_proc_msg3_req((const sample_ra_msg3_t *)((uint8_t
*)p_req +

sizeof(ra_samp_request_header_t)),
                                          p_req->size,
                                          &p_resp_msg);
                if (0 != ret)
                {
                    fprintf(stderr, "\nError, call sp_ra_proc_msg3_req fail
[%s].",
                            __FUNCTION__);
```

```
                }
                else
                {
                    memset(sendbuf, 0, BUFSIZ);
                    if (memcpy_s(sendbuf, BUFSIZ, p_resp_msg,
sizeof(ra_samp_response_header_t) + p_resp_msg->size))
                    {
                        fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed
in [%s].",
                                __FUNCTION__);
                        ret = -1;
                        goto CLEANUP;
                    }
                    fprintf(OUTPUT, "\nSend attestation data\n");
                    PRINT_BYTE_ARRAY(OUTPUT, p_resp_msg,
sizeof(ra_samp_response_header_t) + p_resp_msg->size);
                    int buflen = SendToClient(sizeof(ra_samp_response_header_t) +
p_resp_msg->size);
                    fprintf(OUTPUT, "\nSend attestation data Done,send length =
%d", buflen);
                }
                SAFE_FREE(p_req);
                SAFE_FREE(p_resp_msg);
                break;

            //进行解密
            case TYPE_RA_MSGDEC:
                fprintf(OUTPUT, "\nProcess Decrypt");
                fprintf(OUTPUT, "\nDecrypt 1 %d %x",enclave_id, status);
                /*SGX_ERROR_MAC_MISMATCH 0x3001 Indicates verification error for
reports, sealed datas, etc */
                ret = myaesdecrypt((const ra_samp_request_header_t *)((uint8_t
*)p_req +

 sizeof(ra_samp_request_header_t)),
                                   p_req->size,
                                   enclave_id,
                                   &status,
                                   context);
                fprintf(OUTPUT, "\nDecrypt Done %d %d",enclave_id, status);
                if (0 != ret)
                {
                    fprintf(stderr, "\nError, call decrypt fail [%s].",
                            __FUNCTION__);
                }
                SAFE_FREE(p_req);
                goto CLEANUP;
            //进行加密
            case TYPE_RA_MSGENC:
                fprintf(OUTPUT, "\nProcess Encrypt");
                ret = myaesencrypt((const ra_samp_request_header_t *)((uint8_t
*)p_req +

 sizeof(ra_samp_request_header_t)),
                                   p_req->size,
                                   enclave_id,
                                   &status,
                                   context);
```

```
                fprintf(OUTPUT, "\nEncrypt Done %d %d",enclave_id, status);
                if (0 != ret)
                {
                    fprintf(stderr, "\nError, call encrypt fail [%s].",
                            __FUNCTION__);
                }
                SAFE_FREE(p_req);
                break;

            case TYPE_RA_MSGSETKEY:
                //本来的逻辑是验证数据是不是enclave传过来的, token
                fprintf(OUTPUT, "\nSet Key");
                ret = myaessetkey((const ra_samp_request_header_t *)((uint8_t
*)p_req +

sizeof(ra_samp_request_header_t)),
                                  p_req->size,
                                  enclave_id,
                                  &status,
                                  context);
                if (0 != ret)
                {
                    fprintf(stderr, "\nError, call encrypt fail [%s].",
                            __FUNCTION__);
                }
                SAFE_FREE(p_req);
                break;
            default:
                ret = -1;
                fprintf(stderr, "\nError, unknown ra message type. Type = %d
[%s].",
                        p_req->type, __FUNCTION__);
                break;
            }
        }
    } while (true);

CLEANUP:
    // Clean-up
    // Need to close the RA key state.
    if (INT_MAX != context)
    {
        int ret_save = ret;
        ret = enclave_ra_close(enclave_id, &status, context);
        if (SGX_SUCCESS != ret || status)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, call enclave_ra_close fail [%s].",
                    __FUNCTION__);
        }
        else
        {
            // enclave_ra_close was successful, let's restore the value that
            // led us to this point in the code.
            ret = ret_save;
        }
        fprintf(OUTPUT, "\nCall enclave_ra_close success.");
    }
```

```
    sgx_destroy_enclave(enclave_id);

    ra_free_network_response_buffer(p_msg0_resp_full);
    ra_free_network_response_buffer(p_msg2_full);
    ra_free_network_response_buffer(p_att_result_msg_full);

    // p_msg3 is malloc'd by the untrusted KE library. App needs to free.
    SAFE_FREE(p_msg3);
    SAFE_FREE(p_msg3_full);
    SAFE_FREE(p_msg1_full);
    printf("\nExit ...\n");
    return ret;
}
```

Enclave

enclave.edl

```
enclave {
    from "sgx_tkey_exchange.edl" import *;

    include "sgx_key_exchange.h"
    include "sgx_trts.h"

    trusted {
        public sgx_status_t enclave_init_ra(int b_pse,
                                            [out] sgx_ra_context_t *p_context);
        public sgx_status_t enclave_ra_close(sgx_ra_context_t context);
        public sgx_status_t verify_att_result_mac(sgx_ra_context_t context,
                                                  [in,size=message_size] uint8_t*
message,
                                                  size_t message_size,
                                                  [in,size=mac_size] uint8_t*
mac,
                                                  size_t mac_size);
        public sgx_status_t put_secret_data(sgx_ra_context_t context,
                                            [in,size=secret_size] uint8_t*
p_secret,
                                            uint32_t secret_size,
                                            [in,count=16] uint8_t* gcm_mac);
        public sgx_status_t enclave_generate_key([in,size=secret_size]uint8_t
*p_data,
                                            uint32_t secret_size);
        public sgx_status_t enclave_encrypt([in,size=secret_size] uint8_t*
p_data,
                                            uint32_t secret_size,
                                            [out,size=secret_size] uint8_t*
out_data);
        public sgx_status_t enclave_decrypt([in,size=secret_size] uint8_t*
p_data,
                                            uint32_t secret_size,
```

```
                                                [out,size=secret_size] uint8_t*
out_data);
    };

};
```

enclave.cpp

```
#include <assert.h>
#include "isv_enclave_t.h"
#include "sgx_tkey_exchange.h"
#include "sgx_tcrypto.h"
#include "string.h"
// This is the public EC key of the SP. The corresponding private EC key is
// used by the SP to sign data used in the remote attestation SIGMA protocol
// to sign channel binding data in MSG2. A successful verification of the
// signature confirms the identity of the SP to the ISV app in remote
// attestation secure channel binding. The public EC key should be hardcoded in
// the enclave or delivered in a trustworthy manner. The use of a spoofed public
// EC key in the remote attestation with secure channel binding session may lead
// to a security compromise. Every different SP the enlcave communicates to
// must have a unique SP public key. Delivery of the SP public key is
// determined by the ISV. The TKE SIGMA protocl expects an Elliptical Curve key
// based on NIST P-256
static const sgx_ec256_public_t g_sp_pub_key = {
    {
        0x72, 0x12, 0x8a, 0x7a, 0x17, 0x52, 0x6e, 0xbf,
        0x85, 0xd0, 0x3a, 0x62, 0x37, 0x30, 0xae, 0xad,
        0x3e, 0x3d, 0xaa, 0xee, 0x9c, 0x60, 0x73, 0x1d,
        0xb0, 0x5b, 0xe8, 0x62, 0x1c, 0x4b, 0xeb, 0x38
    },
    {
        0xd4, 0x81, 0x40, 0xd9, 0x50, 0xe2, 0x57, 0x7b,
        0x26, 0xee, 0xb7, 0x41, 0xe7, 0xc6, 0x14, 0xe2,
        0x24, 0xb7, 0xbd, 0xc9, 0x03, 0xf2, 0x9a, 0x28,
        0xa8, 0x3c, 0xc8, 0x10, 0x11, 0x14, 0x5e, 0x06
    }

};

// Used to store the secret passed by the SP in the sample code. The
// size is forced to be 8 bytes. Expected value is
// 0x01,0x02,0x03,0x04,0x0x5,0x0x6,0x0x7
uint8_t g_secret[8] = {0};
sgx_ec_key_128bit_t sk_key;
//lhadd
sgx_ec_key_128bit_t aes_key;
sgx_ec_key_128bit_t aes2_key;

#ifdef SUPPLIED_KEY_DERIVATION

#pragma message ("Supplied key derivation function is used.")

typedef struct _hash_buffer_t
```

```
{
    uint8_t counter[4];
    sgx_ec256_dh_shared_t shared_secret;
    uint8_t algorithm_id[4];
} hash_buffer_t;

const char ID_U[] = "SGXRAENCLAVE";
const char ID_V[] = "SGXRASERVER";

// Derive two keys from shared key and key id.
bool derive_key(
    const sgx_ec256_dh_shared_t *p_shared_key,
    uint8_t key_id,
    sgx_ec_key_128bit_t *first_derived_key,
    sgx_ec_key_128bit_t *second_derived_key)
{
    sgx_status_t sgx_ret = SGX_SUCCESS;
    hash_buffer_t hash_buffer;
    sgx_sha_state_handle_t sha_context;
    sgx_sha256_hash_t key_material;

    memset(&hash_buffer, 0, sizeof(hash_buffer_t));
    /* counter in big endian  */
    hash_buffer.counter[3] = key_id;

    /*convert from little endian to big endian */
    for (size_t i = 0; i < sizeof(sgx_ec256_dh_shared_t); i++)
    {
        hash_buffer.shared_secret.s[i] = p_shared_key->s[sizeof(p_shared_key-
>s)-1 - i];
    }

    sgx_ret = sgx_sha256_init(&sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
        return false;
    }
    sgx_ret = sgx_sha256_update((uint8_t*)&hash_buffer, sizeof(hash_buffer_t),
sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_update((uint8_t*)&ID_U, sizeof(ID_U), sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_update((uint8_t*)&ID_V, sizeof(ID_V), sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_get_hash(sha_context, &key_material);
    if (sgx_ret != SGX_SUCCESS)
```

```
    {
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_close(sha_context);

    assert(sizeof(sgx_ec_key_128bit_t)* 2 == sizeof(sgx_sha256_hash_t));
    memcpy(first_derived_key, &key_material, sizeof(sgx_ec_key_128bit_t));
    memcpy(second_derived_key, (uint8_t*)&key_material +
sizeof(sgx_ec_key_128bit_t), sizeof(sgx_ec_key_128bit_t));

    // memset here can be optimized away by compiler, so please use memset_s on
    // windows for production code and similar functions on other OSes.
    memset(&key_material, 0, sizeof(sgx_sha256_hash_t));

    return true;
}

//isv defined key derivation function id
#define ISV_KDF_ID 2

typedef enum _derive_key_type_t
{
    DERIVE_KEY_SMK_SK = 0,
    DERIVE_KEY_MK_VK,
} derive_key_type_t;

sgx_status_t key_derivation(const sgx_ec256_dh_shared_t* shared_key,
    uint16_t kdf_id,
    sgx_ec_key_128bit_t* smk_key,
    sgx_ec_key_128bit_t* sk_key,
    sgx_ec_key_128bit_t* mk_key,
    sgx_ec_key_128bit_t* vk_key)
{
    bool derive_ret = false;

    if (NULL == shared_key)
    {
        return SGX_ERROR_INVALID_PARAMETER;
    }

    if (ISV_KDF_ID != kdf_id)
    {
        //fprintf(stderr, "\nError, key derivation id mismatch in [%s].",
__FUNCTION__);
        return SGX_ERROR_KDF_MISMATCH;
    }

    derive_ret = derive_key(shared_key, DERIVE_KEY_SMK_SK,
        smk_key, sk_key);
    if (derive_ret != true)
    {
        //fprintf(stderr, "\nError, derive key fail in [%s].", __FUNCTION__);
        return SGX_ERROR_UNEXPECTED;
    }

    derive_ret = derive_key(shared_key, DERIVE_KEY_MK_VK,
        mk_key, vk_key);
```

```
    if (derive_ret != true)
    {
        //fprintf(stderr, "\nError, derive key fail in [%s].", __FUNCTION__);
        return SGX_ERROR_UNEXPECTED;
    }
    return SGX_SUCCESS;
}
#else
#pragma message ("Default key derivation function is used.")
#endif

// This ecall is a wrapper of sgx_ra_init to create the trusted
// KE exchange key context needed for the remote attestation
// SIGMA API's. Input pointers aren't checked since the trusted stubs
// copy them into EPC memory.
//
// @param b_pse Indicates whether the ISV app is using the
//              platform services.
// @param p_context Pointer to the location where the returned
//                  key context is to be copied.
//
// @return Any error return from the create PSE session if b_pse
//         is true.
// @return Any error returned from the trusted key exchange API
//         for creating a key context.

sgx_status_t enclave_init_ra(
        int b_pse,
        sgx_ra_context_t *p_context) {
    // isv enclave call to trusted key exchange library.
    sgx_status_t ret;
#ifdef SUPPLIED_KEY_DERIVATION
    ret = sgx_ra_init_ex(&g_sp_pub_key, b_pse, key_derivation, p_context);
#else
    ret = sgx_ra_init(&g_sp_pub_key, b_pse, p_context);
#endif
    return ret;
}


// Closes the tKE key context used during the SIGMA key
// exchange.
//
// @param context The trusted KE library key context.
//
// @return Return value from the key context close API

sgx_status_t SGXAPI enclave_ra_close(
        sgx_ra_context_t context) {
sgx_status_t ret;
ret = sgx_ra_close(context);
return ret;
}


// Verify the mac sent in att_result_msg from the SP using the
// MK key. Input pointers aren't checked since the trusted stubs
// copy them into EPC memory.
```

```
//
//
// @param context The trusted KE library key context.
// @param p_message Pointer to the message used to produce MAC
// @param message_size Size in bytes of the message.
// @param p_mac Pointer to the MAC to compare to.
// @param mac_size Size in bytes of the MAC
//
// @return SGX_ERROR_INVALID_PARAMETER - MAC size is incorrect.
// @return Any error produced by tKE  API to get SK key.
// @return Any error produced by the AESCMAC function.
// @return SGX_ERROR_MAC_MISMATCH - MAC compare fails.

sgx_status_t verify_att_result_mac(sgx_ra_context_t context,
                                   uint8_t* p_message,
                                   size_t message_size,
                                   uint8_t* p_mac,
                                   size_t mac_size)
{
    sgx_status_t ret;
    sgx_ec_key_128bit_t mk_key;

    if(mac_size != sizeof(sgx_mac_t))
    {
        ret = SGX_ERROR_INVALID_PARAMETER;
        return ret;
    }
    if(message_size > UINT32_MAX)
    {
        ret = SGX_ERROR_INVALID_PARAMETER;
        return ret;
    }

    do {
        uint8_t mac[SGX_CMAC_MAC_SIZE] = {0};

        ret = sgx_ra_get_keys(context, SGX_RA_KEY_MK, &mk_key);
        if(SGX_SUCCESS != ret)
        {
            break;
        }
        ret = sgx_rijndael128_cmac_msg(&mk_key,
                                       p_message,
                                       (uint32_t)message_size,
                                       &mac);
        if(SGX_SUCCESS != ret)
        {
            break;
        }
        if(0 == consttime_memequal(p_mac, mac, sizeof(mac)))
        {
            ret = SGX_ERROR_MAC_MISMATCH;
            break;
        }

    }
    while(0);
```

```
    return ret;
}


// Generate a secret information for the SP encrypted with SK.
// Input pointers aren't checked since the trusted stubs copy
// them into EPC memory.
//
// @param context The trusted KE library key context.
// @param p_secret Message containing the secret.
// @param secret_size Size in bytes of the secret message.
// @param p_gcm_mac The pointer the the AESGCM MAC for the
//                 message.
//
// @return SGX_ERROR_INVALID_PARAMETER - secret size if
//         incorrect.
// @return Any error produced by tKE  API to get SK key.
// @return Any error produced by the AESGCM function.
// @return SGX_ERROR_UNEXPECTED - the secret doesn't match the
//         expected value.

sgx_status_t put_secret_data(
    sgx_ra_context_t context,
    uint8_t *p_secret,
    uint32_t secret_size,
    uint8_t *p_gcm_mac)
{
    sgx_status_t ret = SGX_SUCCESS;

    do {
        if(secret_size != 8)
        {
            ret = SGX_ERROR_INVALID_PARAMETER;
            break;
        }

        ret = sgx_ra_get_keys(context, SGX_RA_KEY_SK, &sk_key);
        if(SGX_SUCCESS != ret)
        {
            break;
        }

        uint8_t aes_gcm_iv[12] = {0};
        ret = sgx_rijndael128GCM_decrypt(&sk_key,
                                         p_secret,
                                         secret_size,
                                         &g_secret[0],
                                         &aes_gcm_iv[0],
                                         12,
                                         NULL,
                                         0,
                                         (const sgx_aes_gcm_128bit_tag_t *)
                                            (p_gcm_mac));

        uint32_t i;
        bool secret_match = true;
        for(i=0;i<secret_size;i++)
        {
```

```
            if(g_secret[i] != i)
            {
                secret_match = false;
            }
        }

        if(!secret_match)
        {
            ret = SGX_ERROR_UNEXPECTED;
        }

        // Once the server has the shared secret, it should be sealed to
        // persistent storage for future use. This will prevents having to
        // perform remote attestation until the secret goes stale. Once the
        // enclave is created again, the secret can be unsealed.
    } while(0);
    return ret;
}

// Generate a secret information for the SP encrypted with SK.
// Input pointers aren't checked since the trusted stubs copy
// them into EPC memory.
//
// @param context The trusted KE library key context.
// @param p_secret Message containing the secret.
// @param secret_size Size in bytes of the secret message.
// @param p_gcm_mac The pointer the the AESGCM MAC for the
//                 message.
//
// @return SGX_ERROR_INVALID_PARAMETER - secret size if
//         incorrect.
// @return Any error produced by tKE  API to get SK key.
// @return Any error produced by the AESGCM function.
// @return SGX_ERROR_UNEXPECTED - the secret doesn't match the
//         expected value.

sgx_status_t enclave_generate_key(
    uint8_t *p_data,
    uint32_t secret_size)
{
    sgx_status_t ret = SGX_SUCCESS;

    if(secret_size != 32)
    return SGX_ERROR_INVALID_METADATA;

    uint8_t aes_gcm_iv[12] = {0};
    uint8_t out_data[32] = {0};
    int i = 0;
    sgx_aes_gcm_128bit_tag_t c_gcm_mac;
    do {
        //首先验证16个字节是不是token
        ret = sgx_rijndael128GCM_decrypt(&sk_key,
                                         p_data,
                                         32,
                                         &out_data[0],
                                         &aes_gcm_iv[0],
                                         12,
                                         NULL,
```

```
                                        0,
                                        &c_gcm_mac);
        if(SGX_SUCCESS != ret)
        {
            break;
        }
    } while(0);
    //token这个硬编码成5到20
    bool secret_match = true;
    for(i=0;i<16;i++)
    {
        if(out_data[i] != i+5)
        {
            secret_match = false;
        }
    }
    if(secret_match == true)
    {//设定后16字节为key
        memcpy((void* )&aes_key, &out_data[16], 16);
        memcpy((void* )&aes2_key, &out_data[16], 16);
    }

    return ret;
}

sgx_status_t enclave_encrypt(
    uint8_t *p_data,
    uint32_t secret_size,
    uint8_t *out_data)
{
    sgx_status_t ret = SGX_SUCCESS;
    sgx_aes_gcm_128bit_tag_t c_gcm_mac;
    do {
        uint8_t aes_gcm_iv[12] = {0};
        ret = sgx_rijndael128GCM_encrypt(&aes_key,
                                        p_data,
                                        secret_size,
                                        out_data,
                                        &aes_gcm_iv[0],
                                        12,
                                        NULL,
                                        0,
                                        &c_gcm_mac);
        if(SGX_SUCCESS != ret)
        {
            break;
        }
    } while(0);
    return ret;
}

sgx_status_t enclave_decrypt(
    uint8_t *p_data,
    uint32_t secret_size,
    uint8_t *out_data)
{
    sgx_status_t ret = SGX_SUCCESS;
    sgx_aes_gcm_128bit_tag_t c_gcm_mac;
```

```
    memcpy(out_data, &aes2_key,16);
    do {
        uint8_t aes_gcm_iv[12] = {0};
        ret = sgx_rijndael128GCM_decrypt(&aes2_key,
                                         p_data,
                                         secret_size,
                                         out_data,
                                         &aes_gcm_iv[0],
                                         12,
                                         NULL,
                                         0,
                                         (const sgx_aes_gcm_128bit_tag_t
*)&c_gcm_mac);
        if(SGX_SUCCESS != ret)
        {

            break;
        }
    } while(0);
    return ret;
}
```

Service Provider

network_ra_server.cpp

提供socket通信服务

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
//add
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

#include "network_ra.h"
#include "service_provider.h"


// Used to send requests to the service provider sample.  It
// simulates network communication between the ISV app and the
// ISV service provider.  This would be modified in a real
// product to use the proper IP communication.
//
// @param server_url String name of the server URL
// @param p_req Pointer to the message to be sent.
// @param p_resp Pointer to a pointer of the response message.

// @return int
char sendbuf[BUFSIZ];  //数据传送的缓冲区
char recvbuf[BUFSIZ];  //数据接受的缓冲区
```

```
int server_sockfd;//服务器端套接字
int client_sockfd;//客户端套接字

int ra_network_send_receive(const char *server_url,
    const ra_samp_request_header_t *p_req,
    ra_samp_response_header_t **p_resp)
{
    int ret = 0;
    ra_samp_response_header_t* p_resp_msg;

    if((NULL == server_url) ||
        (NULL == p_req) ||
        (NULL == p_resp))
    {
        return -1;
    }

    switch(p_req->type)
    {

    case TYPE_RA_MSG0:
        ret = sp_ra_proc_msg0_req((const sample_ra_msg0_t*)((uint8_t*)p_req
            + sizeof(ra_samp_request_header_t)),
            p_req->size);
        if (0 != ret)
        {
            fprintf(stderr, "\nError, call sp_ra_proc_msg1_req fail [%s].",
                __FUNCTION__);
        }
        break;

    case TYPE_RA_MSG1:
        ret = sp_ra_proc_msg1_req((const sample_ra_msg1_t*)((uint8_t*)p_req
            + sizeof(ra_samp_request_header_t)),
            p_req->size,
            &p_resp_msg);
        if(0 != ret)
        {
            fprintf(stderr, "\nError, call sp_ra_proc_msg1_req fail [%s].",
                __FUNCTION__);
        }
        else
        {
            *p_resp = p_resp_msg;
        }
        break;

    case TYPE_RA_MSG3:
        ret =sp_ra_proc_msg3_req((const sample_ra_msg3_t*)((uint8_t*)p_req +
            sizeof(ra_samp_request_header_t)),
            p_req->size,
            &p_resp_msg);
        if(0 != ret)
        {
            fprintf(stderr, "\nError, call sp_ra_proc_msg3_req fail [%s].",
                __FUNCTION__);
        }
        else
```

```
        {
            *p_resp = p_resp_msg;
        }
        break;

    default:
        ret = -1;
        fprintf(stderr, "\nError, unknown ra message type. Type = %d [%s].",
            p_req->type, __FUNCTION__);
        break;
    }

    return ret;
}

int server(int port)
{
    FILE *OUTPUT = stdout;
    int len;
    struct sockaddr_in my_addr;   //服务器网络地址结构体
    struct sockaddr_in remote_addr; //客户端网络地址结构体
    socklen_t sin_size;

    memset(&my_addr,0,sizeof(my_addr)); //数据初始化--清零
    my_addr.sin_family=AF_INET; //设置为IP通信
    my_addr.sin_addr.s_addr=INADDR_ANY;//服务器IP地址--允许连接到所有本地地址上
    my_addr.sin_port=htons(port); //服务器端口号

    /*创建服务器端套接字--IPv4协议，面向连接通信，TCP协议*/
    if((server_sockfd=socket(PF_INET,SOCK_STREAM,0))<0)
    {
        perror("socket");
        return 1;
    }

        /*将套接字绑定到服务器的网络地址上*/
    if (bind(server_sockfd,(struct sockaddr *)&my_addr,sizeof(struct sockaddr))
<0)
    {
        perror("bind");
        return 1;
    }

    /*监听连接请求--监听队列长度为5*/
    listen(server_sockfd,5);

    sin_size=sizeof(struct sockaddr_in);

    /*等待客户端连接请求到达*/
    if((client_sockfd=accept(server_sockfd,(struct sockaddr
*)&remote_addr,&sin_size))<0)
    {
        perror("accept");
        return 1;
    }
    fprintf(OUTPUT, "\naccepted\n");
    return 0;
}
```

```c
int SendToClient(int len)
{
    len=send(client_sockfd,sendbuf,len,0);//发送欢迎信息
}

int RecvfromCient()
{
    /*接收客户端的数据*/
    int len = 0;
    memset(recvbuf, 0, BUFSIZ);
    len=recv(client_sockfd,recvbuf,BUFSIZ,0);
    if (len > 0 || len < BUFSIZ)
        recvbuf[len] = 0;
    return len;
}

int Cleanupsocket()
{
    close(client_sockfd);
    close(server_sockfd);
    return 0;
}
// Used to free the response messages.  In the sample code, the
// response messages are allocated by the SP code.
//
//
// @param resp Pointer to the response buffer to be freed.

void ra_free_network_response_buffer(ra_samp_response_header_t *resp)
{
    if(resp!=NULL)
    {
        free(resp);
    }
}
```

附录五：Remote Attestation (client) 代码实现

App

app.cpp

```
#include <stdio.h>
#include <limits.h>
#include <unistd.h>
// Needed for definition of remote attestation messages.
#include "remote_attestation_result.h"

#include "isv_enclave_u.h"

// Needed to call untrusted key exchange library APIs, i.e. sgx_ra_proc_msg2.
#include "sgx_ukey_exchange.h"

// Needed to get service provider's information, in your real project, you will
// need to talk to real server.
#include "network_ra.h"

// Needed to create enclave and do ecall.
#include "sgx_urts.h"

// Needed to query extended epid group id.
#include "sgx_uae_service.h"

#include "service_provider.h"

#ifndef SAFE_FREE
#define SAFE_FREE(ptr)     \
    {                      \
        if (NULL != (ptr)) \
        {                  \
            free(ptr);     \
            (ptr) = NULL;  \
        }                  \
    }
#endif

// In addition to generating and sending messages, this application
// can use pre-generated messages to verify the generation of
// messages and the information flow.
#include "sample_messages.h"

#define ENCLAVE_PATH "isv_enclave.signed.so"

uint8_t *msg1_samples[] = {msg1_sample1, msg1_sample2};
uint8_t *msg2_samples[] = {msg2_sample1, msg2_sample2};
uint8_t *msg3_samples[] = {msg3_sample1, msg3_sample2};
uint8_t *attestation_msg_samples[] =
```

```
    {attestation_msg_sample1, attestation_msg_sample2};

extern char sendbuf[BUFSIZ]; //数据传送的缓冲区
extern char recvbuf[BUFSIZ];

// Some utility functions to output some of the data structures passed between
// the ISV app and the remote attestation service provider.
void PRINT_BYTE_ARRAY(
    FILE *file, void *mem, uint32_t len)
{
    if (!mem || !len)
    {
        fprintf(file, "\n( null )\n");
        return;
    }
    uint8_t *array = (uint8_t *)mem;
    fprintf(file, "%u bytes:\n{\n", len);
    uint32_t i = 0;
    for (i = 0; i < len - 1; i++)
    {
        fprintf(file, "0x%x, ", array[i]);
        if (i % 8 == 7)
            fprintf(file, "\n");
    }
    fprintf(file, "0x%x ", array[i]);
    fprintf(file, "\n}\n");
}

void PRINT_ATTESTATION_SERVICE_RESPONSE(
    FILE *file,
    ra_samp_response_header_t *response)
{
    if (!response)
    {
        fprintf(file, "\t\n( null )\n");
        return;
    }

    fprintf(file, "RESPONSE TYPE:   0x%x\n", response->type);
    fprintf(file, "RESPONSE STATUS: 0x%x 0x%x\n", response->status[0],
            response->status[1]);
    fprintf(file, "RESPONSE BODY SIZE: %u\n", response->size);

    if (response->type == TYPE_RA_MSG2)
    {
        sgx_ra_msg2_t *p_msg2_body = (sgx_ra_msg2_t *)(response->body);

        fprintf(file, "MSG2 gb - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->g_b), sizeof(p_msg2_body->g_b));

        fprintf(file, "MSG2 spid - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->spid), sizeof(p_msg2_body->spid));

        fprintf(file, "MSG2 quote_type : %hx\n", p_msg2_body->quote_type);

        fprintf(file, "MSG2 kdf_id : %hx\n", p_msg2_body->kdf_id);

        fprintf(file, "MSG2 sign_gb_ga - ");
```

```
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->sign_gb_ga),
                         sizeof(p_msg2_body->sign_gb_ga));

        fprintf(file, "MSG2 mac - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->mac), sizeof(p_msg2_body->mac));

        fprintf(file, "MSG2 sig_rl - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->sig_rl),
                         p_msg2_body->sig_rl_size);
    }
    else if (response->type == TYPE_RA_ATT_RESULT)
    {
        sample_ra_att_result_msg_t *p_att_result =
            (sample_ra_att_result_msg_t *)(response->body);
        fprintf(file, "ATTESTATION RESULT MSG platform_info_blob - ");
        PRINT_BYTE_ARRAY(file, &(p_att_result->platform_info_blob),
                         sizeof(p_att_result->platform_info_blob));

        fprintf(file, "ATTESTATION RESULT MSG mac - ");
        PRINT_BYTE_ARRAY(file, &(p_att_result->mac), sizeof(p_att_result->mac));

        fprintf(file, "ATTESTATION RESULT MSG secret.payload_tag - %u bytes\n",
                p_att_result->secret.payload_size);

        fprintf(file, "ATTESTATION RESULT MSG secret.payload - ");
        PRINT_BYTE_ARRAY(file, p_att_result->secret.payload,
                         p_att_result->secret.payload_size);
    }
    else
    {
        fprintf(file, "\nERROR in printing out the response. "
                      "Response of type not supported %d\n",
                response->type);
    }
}

// This sample code doesn't have any recovery/retry mechanisms for the remote
// attestation. Since the enclave can be lost due S3 transitions, apps
// susceptible to S3 transitions should have logic to restart attestation in
// these scenarios.
#define _T(x) x
int main(int argc, char *argv[])
{
    int ret = 0;
    ra_samp_request_header_t *p_msg0_full = NULL;
    ra_samp_response_header_t *p_msg0_resp_full = NULL;
    ra_samp_request_header_t *p_msg1_full = NULL;
    ra_samp_response_header_t *p_msg2_full = NULL;
    sgx_ra_msg3_t *p_msg3 = NULL;
    ra_samp_response_header_t *p_att_result_msg_full = NULL;
    sgx_enclave_id_t enclave_id = 0;
    int enclave_lost_retry_time = 1;
    int busy_retry_time = 4;
    sgx_ra_context_t context = INT_MAX;
    sgx_status_t status = SGX_SUCCESS;
    ra_samp_request_header_t *p_msg3_full = NULL;

    int32_t verify_index = -1;
```

```
    int32_t verification_samples = sizeof(msg1_samples) /
sizeof(msg1_samples[0]);

    FILE *OUTPUT = stdout;

#define VERIFICATION_INDEX_IS_VALID() (verify_index > 0 && \
                                       verify_index <= verification_samples)
#define GET_VERIFICATION_ARRAY_INDEX() (verify_index - 1)

    if (argc > 1)
    {

        verify_index = atoi(argv[1]);

        if (VERIFICATION_INDEX_IS_VALID())
        {
            fprintf(OUTPUT, "\nVerifying precomputed attestation messages "
                            "using precomputed values# %d\n",
                    verify_index);
        }
        else
        {
            fprintf(OUTPUT, "\nValid invocations are:\n");
            fprintf(OUTPUT, "\n\tisv_app\n");
            fprintf(OUTPUT, "\n\tisv_app <verification index>\n");
            fprintf(OUTPUT, "\nValid indices are [1 - %d]\n",
                    verification_samples);
            fprintf(OUTPUT, "\nUsing a verification index uses precomputed "
                            "messages to assist debugging the remote attestation
"
                            "service provider.\n");
            return -1;
        }
    }

    // SOCKET: connect to server
    if (client("127.0.0.1", 12333) != 0)
    {
        fprintf(OUTPUT, "Connect Server Error, Exit!\n");
        return 0;
    }
    // Preparation for remote attestation by configuring extended epid group id.
    {
        uint32_t extended_epid_group_id = 0;
        ret = sgx_get_extended_epid_group_id(&extended_epid_group_id);
        if (SGX_SUCCESS != ret)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, call sgx_get_extended_epid_group_id fail
[%s].",
                    __FUNCTION__);
            return ret;
        }
        fprintf(OUTPUT, "\nCall sgx_get_extended_epid_group_id success.");

        p_msg0_full = (ra_samp_request_header_t *)
            malloc(sizeof(ra_samp_request_header_t) + sizeof(uint32_t));
        if (NULL == p_msg0_full)
```

```
        {
            ret = -1;
            goto CLEANUP;
        }
        p_msg0_full->type = TYPE_RA_MSG0;
        p_msg0_full->size = sizeof(uint32_t);

        *(uint32_t *)((uint8_t *)p_msg0_full + sizeof(ra_samp_request_header_t))
= extended_epid_group_id;
        {

            fprintf(OUTPUT, "\nMSG0 body generated -\n");

            PRINT_BYTE_ARRAY(OUTPUT, p_msg0_full->body, p_msg0_full->size);
        }
        // The ISV application sends msg0 to the SP.
        // The ISV decides whether to support this extended epid group id.
        fprintf(OUTPUT, "\nSending msg0 to remote attestation service
provider.\n");

        // SOCKET: send & recv
        ret = ra_network_send_receive("http://SampleServiceProvider.intel.com/",
                                      p_msg0_full,
                                      &p_msg0_resp_full);
        if (ret != 0)
        {
            fprintf(OUTPUT, "\nError, ra_network_send_receive for msg0 failed "
                            "[%s].",
                    __FUNCTION__);
            goto CLEANUP;
        }
        fprintf(OUTPUT, "\nSent MSG0 to remote attestation service.\n");
    }
    // Remote attestation will be initiated the ISV server challenges the ISV
    // app or if the ISV app detects it doesn't have the credentials
    // (shared secret) from a previous attestation required for secure
    // communication with the server.
    {
        // ISV application creates the ISV enclave.
        int launch_token_update = 0;
        sgx_launch_token_t launch_token = {0};
        memset(&launch_token, 0, sizeof(sgx_launch_token_t));
        do
        {
            ret = sgx_create_enclave(_T(ENCLAVE_PATH),
                                     SGX_DEBUG_FLAG,
                                     &launch_token,
                                     &launch_token_update,
                                     &enclave_id, NULL);
            if (SGX_SUCCESS != ret)
            {
                ret = -1;
                fprintf(OUTPUT, "\nError, call sgx_create_enclave fail [%s].",
                        __FUNCTION__);
                goto CLEANUP;
            }
            fprintf(OUTPUT, "\nCall sgx_create_enclave success.");
```

```
            ret = enclave_init_ra(enclave_id,
                                  &status,
                                  false,
                                  &context);
            //Ideally, this check would be around the full attestation flow.
        } while (SGX_ERROR_ENCLAVE_LOST == ret && enclave_lost_retry_time--);

        if (SGX_SUCCESS != ret || status)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, call enclave_init_ra fail [%s].",
                    __FUNCTION__);
            goto CLEANUP;
        }
        fprintf(OUTPUT, "\nCall enclave_init_ra success.");

        // isv application call uke sgx_ra_get_msg1
        p_msg1_full = (ra_samp_request_header_t *)
            malloc(sizeof(ra_samp_request_header_t) + sizeof(sgx_ra_msg1_t));
        if (NULL == p_msg1_full)
        {
            ret = -1;
            goto CLEANUP;
        }
        p_msg1_full->type = TYPE_RA_MSG1;
        p_msg1_full->size = sizeof(sgx_ra_msg1_t);
        do
        {
            ret = sgx_ra_get_msg1(context, enclave_id, sgx_ra_get_ga,
                                  (sgx_ra_msg1_t *)((uint8_t *)p_msg1_full +
sizeof(ra_samp_request_header_t)));
            sleep(3); // Wait 3s between retries
        } while (SGX_ERROR_BUSY == ret && busy_retry_time--);
        if (SGX_SUCCESS != ret)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, call sgx_ra_get_msg1 fail [%s].",
                    __FUNCTION__);
            goto CLEANUP;
        }
        else
        {
            fprintf(OUTPUT, "\nCall sgx_ra_get_msg1 success.\n");

            fprintf(OUTPUT, "\nMSG1 body generated -\n");

            PRINT_BYTE_ARRAY(OUTPUT, p_msg1_full->body, p_msg1_full->size);
        }

        if (VERIFICATION_INDEX_IS_VALID())
        {

            memcpy_s(p_msg1_full->body, p_msg1_full->size,
                     msg1_samples[GET_VERIFICATION_ARRAY_INDEX()],
                     p_msg1_full->size);

            fprintf(OUTPUT, "\nInstead of using the recently generated MSG1, "
                            "we will use the following precomputed MSG1 -\n");
```

```
            PRINT_BYTE_ARRAY(OUTPUT, p_msg1_full->body, p_msg1_full->size);
        }

        // The ISV application sends msg1 to the SP to get msg2,
        // msg2 needs to be freed when no longer needed.
        // The ISV decides whether to use linkable or unlinkable signatures.
        p_msg2_full = (ra_samp_response_header_t *)malloc(180);
        memset(p_msg2_full, 0, 180);
        if (NULL == p_msg2_full)
        {
            ret = -1;
            goto CLEANUP;
        }
        ret = ra_network_send_receive("http://SampleServiceProvider.intel.com/",
                                      p_msg1_full,
                                      &p_msg2_full);

        if ((ret == 0) || (p_msg2_full == NULL))
        {
            fprintf(OUTPUT, "\nError, ra_network_send_receive for msg1 failed "
                            "[%s].",
                    __FUNCTION__);
            if (VERIFICATION_INDEX_IS_VALID())
            {
                fprintf(OUTPUT, "\nBecause we are in verification mode we will "
                                "ignore this error.\n");
                fprintf(OUTPUT, "\nInstead, we will pretend we received the "
                                "following MSG2 - \n");

                SAFE_FREE(p_msg2_full);
                ra_samp_response_header_t *precomputed_msg2 =
                    (ra_samp_response_header_t
*)msg2_samples[GET_VERIFICATION_ARRAY_INDEX()];
                const size_t msg2_full_size = sizeof(ra_samp_response_header_t) +
precomputed_msg2->size;
                p_msg2_full =
                    (ra_samp_response_header_t *)malloc(msg2_full_size);
                if (NULL == p_msg2_full)
                {
                    ret = -1;
                    goto CLEANUP;
                }
                memcpy_s(p_msg2_full, msg2_full_size, precomputed_msg2,
                         msg2_full_size);

                PRINT_BYTE_ARRAY(OUTPUT, p_msg2_full,
                                 sizeof(ra_samp_response_header_t) + p_msg2_full-
>size);
            }
            else
            {
                goto CLEANUP;
            }
        }
        else
        {
            // Successfully sent msg1 and received a msg2 back.
```

```
            // Time now to check msg2.
            if (TYPE_RA_MSG2 != p_msg2_full->type)
            {

                fprintf(OUTPUT, "\nError, didn't get MSG2 in response to MSG1. "
                                "[%s]. receive type is %d\n",
                        __FUNCTION__, p_msg2_full->type);

                PRINT_BYTE_ARRAY(OUTPUT, p_msg2_full,
                                 176);
                if (VERIFICATION_INDEX_IS_VALID())
                {
                    fprintf(OUTPUT, "\nBecause we are in verification mode we "
                                    "will ignore this error.");
                }
                else
                {
                    goto CLEANUP;
                }
            }

            fprintf(OUTPUT, "\nSent MSG1 to remote attestation service "
                            "provider. Received the following MSG2:\n");
            PRINT_BYTE_ARRAY(OUTPUT, p_msg2_full,
                             sizeof(ra_samp_response_header_t) + p_msg2_full-
>size);

            fprintf(OUTPUT, "\nA more descriptive representation of MSG2:\n");
            PRINT_ATTESTATION_SERVICE_RESPONSE(OUTPUT, p_msg2_full);

            if (VERIFICATION_INDEX_IS_VALID())
            {
                // The response should match the precomputed MSG2:
                ra_samp_response_header_t *precomputed_msg2 =
                    (ra_samp_response_header_t *)
                        msg2_samples[GET_VERIFICATION_ARRAY_INDEX()];
                if (MSG2_BODY_SIZE !=
                        sizeof(ra_samp_response_header_t) + p_msg2_full->size ||
                    memcmp(precomputed_msg2, p_msg2_full,
                           sizeof(ra_samp_response_header_t) + p_msg2_full-
>size))
                {
                    fprintf(OUTPUT, "\nVerification ERROR. Our precomputed "
                                    "value for MSG2 does NOT match.\n");
                    fprintf(OUTPUT, "\nPrecomputed value for MSG2:\n");
                    PRINT_BYTE_ARRAY(OUTPUT, precomputed_msg2,
                                     sizeof(ra_samp_response_header_t) +
precomputed_msg2->size);
                    fprintf(OUTPUT, "\nA more descriptive representation "
                                    "of precomputed value for MSG2:\n");
                    PRINT_ATTESTATION_SERVICE_RESPONSE(OUTPUT,
                                                       precomputed_msg2);
                }
                else
                {
                    fprintf(OUTPUT, "\nVerification COMPLETE. Remote "
                                    "attestation service provider generated a "
                                    "matching MSG2.\n");
```

```
                }
            }
        }

        sgx_ra_msg2_t *p_msg2_body = (sgx_ra_msg2_t *)((uint8_t *)p_msg2_full +
sizeof(ra_samp_response_header_t));

        uint32_t msg3_size = 0;
        if (VERIFICATION_INDEX_IS_VALID())
        {
            // We cannot generate a valid MSG3 using the precomputed messages
            // we have been using. We will use the precomputed msg3 instead.
            msg3_size = MSG3_BODY_SIZE;
            p_msg3 = (sgx_ra_msg3_t *)malloc(msg3_size);
            if (NULL == p_msg3)
            {
                ret = -1;
                goto CLEANUP;
            }
            memcpy_s(p_msg3, msg3_size,
                     msg3_samples[GET_VERIFICATION_ARRAY_INDEX()], msg3_size);
            fprintf(OUTPUT, "\nBecause MSG1 was a precomputed value, the MSG3 "
                            "we use will also be. PRECOMPUTED MSG3 - \n");
        }
        else
        {
            busy_retry_time = 2;
            // The ISV app now calls uKE sgx_ra_proc_msg2,
            // The ISV app is responsible for freeing the returned p_msg3!!
            do
            {
                ret = sgx_ra_proc_msg2(context,
                                       enclave_id,
                                       sgx_ra_proc_msg2_trusted,
                                       sgx_ra_get_msg3_trusted,
                                       p_msg2_body,
                                       p_msg2_full->size,
                                       &p_msg3,
                                       &msg3_size);
            } while (SGX_ERROR_BUSY == ret && busy_retry_time--);
            if (!p_msg3)
            {
                fprintf(OUTPUT, "\nError, call sgx_ra_proc_msg2 fail. "
                                "p_msg3 = 0x%p [%s].",
                        p_msg3, __FUNCTION__);
                ret = -1;
                goto CLEANUP;
            }
            if (SGX_SUCCESS != (sgx_status_t)ret)
            {
                fprintf(OUTPUT, "\nError, call sgx_ra_proc_msg2 fail. "
                                "ret = 0x%08x [%s].",
                        ret, __FUNCTION__);
                ret = -1;
                goto CLEANUP;
            }
            else
            {
```

```
                fprintf(OUTPUT, "\nCall sgx_ra_proc_msg2 success.\n");
                fprintf(OUTPUT, "\nMSG3 - \n");
            }
        }

        PRINT_BYTE_ARRAY(OUTPUT, p_msg3, msg3_size);

        p_msg3_full = (ra_samp_request_header_t *)malloc(
            sizeof(ra_samp_request_header_t) + msg3_size);
        if (NULL == p_msg3_full)
        {
            ret = -1;
            goto CLEANUP;
        }
        p_msg3_full->type = TYPE_RA_MSG3;
        p_msg3_full->size = msg3_size;
        if (memcpy_s(p_msg3_full->body, msg3_size, p_msg3, msg3_size))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }

        // The ISV application sends msg3 to the SP to get the attestation
        // result message, attestation result message needs to be freed when
        // no longer needed. The ISV service provider decides whether to use
        // linkable or unlinkable signatures. The format of the attestation
        // result is up to the service provider. This format is used for
        // demonstration.  Note that the attestation result message makes use
        // of both the MK for the MAC and the SK for the secret. These keys are
        // established from the SIGMA secure channel binding.
        p_att_result_msg_full = (ra_samp_response_header_t *)malloc(180);
        memset(p_msg2_full, 0, 180);
        ret = ra_network_send_receive("http://SampleServiceProvider.intel.com/",
                                      p_msg3_full,
                                      &p_att_result_msg_full);
        if (ret == 0 || p_att_result_msg_full == NULL)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, sending msg3 failed [%s].", __FUNCTION__);
            goto CLEANUP;
        }
        fprintf(OUTPUT, "\nReceive attestation data is\n");
        PRINT_BYTE_ARRAY(OUTPUT, p_att_result_msg_full, 180);
        sample_ra_att_result_msg_t *p_att_result_msg_body =
            (sample_ra_att_result_msg_t *)((uint8_t *)p_att_result_msg_full +
sizeof(ra_samp_response_header_t));
        if (TYPE_RA_ATT_RESULT != p_att_result_msg_full->type)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError. Sent MSG3 successfully, but the message "
                            "received was NOT of type att_msg_result. Type = "
                            "%d. [%s].",
                    p_att_result_msg_full->type,
                    __FUNCTION__);
            goto CLEANUP;
        }
```

```
        else
        {
            fprintf(OUTPUT, "\nSent MSG3 successfully. Received an attestation "
                            "result message back\n.");
            if (VERIFICATION_INDEX_IS_VALID())
            {
                if (ATTESTATION_MSG_BODY_SIZE != p_att_result_msg_full->size ||
                    memcmp(p_att_result_msg_full->body,

attestation_msg_samples[GET_VERIFICATION_ARRAY_INDEX()],
                           p_att_result_msg_full->size))
                {
                    fprintf(OUTPUT, "\nSent MSG3 successfully. Received an "
                                    "attestation result message back that did "
                                    "NOT match the expected value.\n");
                    fprintf(OUTPUT, "\nEXPECTED ATTESTATION RESULT -");
                    PRINT_BYTE_ARRAY(OUTPUT,

attestation_msg_samples[GET_VERIFICATION_ARRAY_INDEX()],
                                     ATTESTATION_MSG_BODY_SIZE);
                }
            }
        }

        fprintf(OUTPUT, "\nATTESTATION RESULT RECEIVED - ");
        PRINT_BYTE_ARRAY(OUTPUT, p_att_result_msg_full->body,
                         p_att_result_msg_full->size);
        fprintf(OUTPUT, "\natt data Body - ");
        PRINT_BYTE_ARRAY(OUTPUT, p_att_result_msg_body,
                         p_att_result_msg_full->size);

        if (VERIFICATION_INDEX_IS_VALID())
        {
            fprintf(OUTPUT, "\nBecause we used precomputed values for the "
                            "messages, the attestation result message will "
                            "not pass further verification tests, so we will "
                            "skip them.\n");
            goto CLEANUP;
        }

        // Check the MAC using MK on the attestation result message.
        // The format of the attestation result message is ISV specific.
        // This is a simple form for demonstration. In a real product,
        // the ISV may want to communicate more information.
        ret = verify_att_result_mac(enclave_id,
                                    &status,
                                    context,
                                    (uint8_t *)&p_att_result_msg_body-
>platform_info_blob,
                                    sizeof(ias_platform_info_blob_t),
                                    (uint8_t *)&p_att_result_msg_body->mac,
                                    sizeof(sgx_mac_t));
        if ((SGX_SUCCESS != ret) ||
            (SGX_SUCCESS != status))
        {
            ret = -1;
            fprintf(OUTPUT, "\nError: INTEGRITY FAILED - attestation result "
                            "message MK based cmac failed in [%s].",
```

```
                    __FUNCTION__);
            goto CLEANUP;
        }

        bool attestation_passed = true;
        // Check the attestation result for pass or fail.
        // Whether attestation passes or fails is a decision made by the ISV
Server.
        // When the ISV server decides to trust the enclave, then it will return
success.
        // When the ISV server decided to not trust the enclave, then it will
return failure.
        if (0 != p_att_result_msg_full->status[0] || 0 != p_att_result_msg_full-
>status[1])
        {
            fprintf(OUTPUT, "\nError, attestation result message MK based cmac "
                            "failed in [%s]. %d %d ",
                    __FUNCTION__,
                    p_att_result_msg_full->status[0],
                    p_att_result_msg_full->status[1]);
            attestation_passed = false;
            goto CLEANUP;
        }
        // The attestation result message should contain a field for the Platform
        // Info Blob (PIB).  The PIB is returned by attestation server in the
attestation report.
        // It is not returned in all cases, but when it is, the ISV app
        // should pass it to the blob analysis API called
sgx_report_attestation_status()
        // along with the trust decision from the ISV server.
        // The ISV application will take action based on the update_info.
        // returned in update_info by the API.
        // This call is stubbed out for the sample.
        //
        // sgx_update_info_bit_t update_info;
        // ret = sgx_report_attestation_status(
        //     &p_att_result_msg_body->platform_info_blob,
        //     attestation_passed ? 0 : 1, &update_info);

        // Get the shared secret sent by the server using SK (if attestation
        // passed)
        if (attestation_passed)
        {
            fprintf(OUTPUT,
                    "\nthe size of secret is %d, the secret is\n",
                    p_att_result_msg_body->secret.payload_size);
            PRINT_BYTE_ARRAY(OUTPUT, &p_att_result_msg_body->secret, 40);
            fprintf(OUTPUT, "\nthe context is:\n");
            PRINT_BYTE_ARRAY(OUTPUT, &context, sizeof(context));
            ret = put_secret_data(enclave_id,
                                  &status,
                                  context,
                                  p_att_result_msg_body->secret.payload,
                                  p_att_result_msg_body->secret.payload_size,
                                  p_att_result_msg_body->secret.payload_tag);
            if ((SGX_SUCCESS != ret) || (SGX_SUCCESS != status))
            {
                fprintf(OUTPUT, "\nError, attestation result message secret "
```

```
                                "using SK based AESGCM failed in [%s]. ret = "
                                "0x%0x. status = 0x%0x",
                        __FUNCTION__, ret,
                        status);
                goto CLEANUP;
            }
        }
        else
        {
            fprintf(OUTPUT, "\nRemote attestation fail in [%s]", __FUNCTION__);
            goto CLEANUP;
        }

        //fprintf(OUTPUT, "\nSecret successfully received from server.");
        fprintf(OUTPUT, "\nRemote attestation success!");

        //三个过程共用，每次用完都释放
        ra_samp_request_header_t *p_setkeyreq = NULL;
        ra_samp_response_header_t *p_response = NULL;
        int recvlen = 0;
        //这时候开始与远程enclave建立加密信道

        //set key:g_sp_db就是协商出来的加密秘钥
        uint8_t token_with_key[32] = {0};
        fprintf(OUTPUT, "\nStart generate server key!");
        ret = generate_server_key(enclave_id, &status, token_with_key, 32);
        if ((SGX_SUCCESS != ret) || (SGX_SUCCESS != status))
        {
            fprintf(OUTPUT, "\nError ");
            goto CLEANUP;
        }
        //传key到server enclave中

        uint8_t out_data[16] = {'K', 'E', 'Y', 'S', 'E', 'T', 'S', 'U', 'C', 'C',
'E', 'S'};
        p_setkeyreq = (ra_samp_request_header_t
*)malloc(sizeof(ra_samp_request_header_t) + 32);
        memset(p_setkeyreq, 0 ,sizeof(ra_samp_request_header_t) + 32);
        p_setkeyreq->type = TYPE_RA_MSGSETKEY;
        p_setkeyreq->size = 32;
        if (memcpy_s(p_setkeyreq->body, 32, token_with_key, 32))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }
        fprintf(OUTPUT, "\nGenerate server aes key Success.");
        memset(sendbuf, 0, BUFSIZ);
        memcpy_s(sendbuf, BUFSIZ, p_setkeyreq, sizeof(ra_samp_request_header_t) +
32);
        SendToServer(sizeof(ra_samp_request_header_t) + 32);
        SAFE_FREE(p_setkeyreq);
        recvlen = RecvfromServer();
        //TODO:检查返回信息
        p_response = (ra_samp_response_header_t
*)malloc(sizeof(ra_samp_response_header_t) + 32);
```

```
        if (memcpy_s(p_response, recvlen, recvbuf, recvlen))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }

        if ((p_response->type != TYPE_RA_MSGSETKEY) ||
            (memcmp(p_response->body, out_data, 16) != 0))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }
        else
        {
            fprintf(OUTPUT, "\nSuccess Set Server KEY ");
        }
        SAFE_FREE(p_response);

        //test remote aes service 加密
        uint8_t test_data[16] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15};
        uint8_t encryptdata[16] = {0};
        uint8_t decryptdata[16] = {0};

        p_setkeyreq = (ra_samp_request_header_t
*)malloc(sizeof(ra_samp_request_header_t) + 16);
        p_setkeyreq->type = TYPE_RA_MSGENC;
        p_setkeyreq->size = 16;

        if (memcpy_s(p_setkeyreq->body, 16, test_data, 16))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }
        memset(sendbuf, 0, BUFSIZ);
        memcpy_s(sendbuf, BUFSIZ, p_setkeyreq, sizeof(ra_samp_request_header_t) +
16);
        SendToServer(sizeof(ra_samp_request_header_t) + 16);
        recvlen = RecvfromServer();
        //TODO:检查返回信息
        p_response = (ra_samp_response_header_t
*)malloc(sizeof(ra_samp_response_header_t) + 32);

        if (memcpy_s(p_response, recvlen, recvbuf, recvlen))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }

        if ((p_response->type != TYPE_RA_MSGENC))
```

```
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }
        else
        {
            memcpy_s(encryptdata, 16, p_response->body, 16);
            fprintf(OUTPUT, "\nSuccess Encrypt");
            PRINT_BYTE_ARRAY(OUTPUT, test_data, 16);
            PRINT_BYTE_ARRAY(OUTPUT, encryptdata, 16);
        }
        SAFE_FREE(p_response);

        //解密

        p_setkeyreq = (ra_samp_request_header_t
*)malloc(sizeof(ra_samp_request_header_t) + 16);
        p_setkeyreq->type = TYPE_RA_MSGDEC;
        p_setkeyreq->size = 16;

        if (memcpy_s(p_setkeyreq->body, 16, encryptdata, 16))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }
        memset(sendbuf, 0, BUFSIZ);
        memcpy_s(sendbuf, BUFSIZ, p_setkeyreq, sizeof(ra_samp_request_header_t) +
16);
        SendToServer(sizeof(ra_samp_request_header_t) + 16);
        recvlen = RecvfromServer();
        //检查返回信息
        p_response = (ra_samp_response_header_t
*)malloc(sizeof(ra_samp_response_header_t) + 32);

        if (memcpy_s(p_response, recvlen, recvbuf, recvlen))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }

        if ((p_response->type != TYPE_RA_MSGDEC))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }
        else
        {
            memcpy_s(decryptdata, 16, p_response->body, 16);
            fprintf(OUTPUT, "\nSuccess Decrypt ");
            PRINT_BYTE_ARRAY(OUTPUT, encryptdata, 16);
```

```
            PRINT_BYTE_ARRAY(OUTPUT, decryptdata, 16);
        }
        SAFE_FREE(p_response);
    }

CLEANUP:
    // Clean-up
    // Need to close the RA key state.
    if (INT_MAX != context)
    {
        int ret_save = ret;
        ret = enclave_ra_close(enclave_id, &status, context);
        if (SGX_SUCCESS != ret || status)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, call enclave_ra_close fail [%s].",
                    __FUNCTION__);
        }
        else
        {
            // enclave_ra_close was successful, let's restore the value that
            // led us to this point in the code.
            ret = ret_save;
        }
        fprintf(OUTPUT, "\nCall enclave_ra_close success.");
    }

    sgx_destroy_enclave(enclave_id);

    ra_free_network_response_buffer(p_msg0_resp_full);
    ra_free_network_response_buffer(p_msg2_full);
    ra_free_network_response_buffer(p_att_result_msg_full);

    // p_msg3 is malloc'd by the untrusted KE library. App needs to free.
    SAFE_FREE(p_msg3);
    SAFE_FREE(p_msg3_full);
    SAFE_FREE(p_msg1_full);
    SAFE_FREE(p_msg0_full);
    printf("\nExit ...\n");
    return ret;
}
```

Enclave

enclave.edl

```
enclave {
    from "sgx_tkey_exchange.edl" import *;

    include "sgx_key_exchange.h"
    include "sgx_trts.h"

    trusted {
```

```
        public sgx_status_t enclave_init_ra(int b_pse,
                                            [out] sgx_ra_context_t *p_context);
        public sgx_status_t enclave_ra_close(sgx_ra_context_t context);
        public sgx_status_t verify_att_result_mac(sgx_ra_context_t context,
                                                  [in,size=message_size] uint8_t*
message,
                                                  size_t message_size,
                                                  [in,size=mac_size] uint8_t*
mac,
                                                  size_t mac_size);
        public sgx_status_t put_secret_data(sgx_ra_context_t context,
                                            [in,size=secret_size] uint8_t*
p_secret,
                                            uint32_t secret_size,
                                            [in,count=16] uint8_t* gcm_mac);
        public sgx_status_t generate_server_key([in,size=32] uint8_t* out_data,
                                            uint32_t secret_size);
    };
};
```

enclave.cpp

```
#include <assert.h>
#include "isv_enclave_t.h"
#include "sgx_tkey_exchange.h"
#include "sgx_tcrypto.h"
#include "string.h"

// This is the public EC key of the SP. The corresponding private EC key is
// used by the SP to sign data used in the remote attestation SIGMA protocol
// to sign channel binding data in MSG2. A successful verification of the
// signature confirms the identity of the SP to the ISV app in remote
// attestation secure channel binding. The public EC key should be hardcoded in
// the enclave or delivered in a trustworthy manner. The use of a spoofed public
// EC key in the remote attestation with secure channel binding session may lead
// to a security compromise. Every different SP the enlcave communicates to
// must have a unique SP public key. Delivery of the SP public key is
// determined by the ISV. The TKE SIGMA protocl expects an Elliptical Curve key
// based on NIST P-256
static const sgx_ec256_public_t g_sp_pub_key = {
    {
        0x72, 0x12, 0x8a, 0x7a, 0x17, 0x52, 0x6e, 0xbf,
        0x85, 0xd0, 0x3a, 0x62, 0x37, 0x30, 0xae, 0xad,
        0x3e, 0x3d, 0xaa, 0xee, 0x9c, 0x60, 0x73, 0x1d,
        0xb0, 0x5b, 0xe8, 0x62, 0x1c, 0x4b, 0xeb, 0x38
    },
    {
        0xd4, 0x81, 0x40, 0xd9, 0x50, 0xe2, 0x57, 0x7b,
        0x26, 0xee, 0xb7, 0x41, 0xe7, 0xc6, 0x14, 0xe2,
        0x24, 0xb7, 0xbd, 0xc9, 0x03, 0xf2, 0x9a, 0x28,
        0xa8, 0x3c, 0xc8, 0x10, 0x11, 0x14, 0x5e, 0x06
    }

};
```

```
// Used to store the secret passed by the SP in the sample code. The
// size is forced to be 8 bytes. Expected value is
// 0x01,0x02,0x03,0x04,0x0x5,0x0x6,0x0x7
uint8_t g_secret[8] = {0};
sgx_ec_key_128bit_t sk_key;
#ifdef SUPPLIED_KEY_DERIVATION

#pragma message ("Supplied key derivation function is used.")

typedef struct _hash_buffer_t
{
    uint8_t counter[4];
    sgx_ec256_dh_shared_t shared_secret;
    uint8_t algorithm_id[4];
} hash_buffer_t;

const char ID_U[] = "SGXRAENCLAVE";
const char ID_V[] = "SGXRASERVER";

// Derive two keys from shared key and key id.
bool derive_key(
    const sgx_ec256_dh_shared_t *p_shared_key,
    uint8_t key_id,
    sgx_ec_key_128bit_t *first_derived_key,
    sgx_ec_key_128bit_t *second_derived_key)
{
    sgx_status_t sgx_ret = SGX_SUCCESS;
    hash_buffer_t hash_buffer;
    sgx_sha_state_handle_t sha_context;
    sgx_sha256_hash_t key_material;

    memset(&hash_buffer, 0, sizeof(hash_buffer_t));
    /* counter in big endian  */
    hash_buffer.counter[3] = key_id;

    /*convert from little endian to big endian */
    for (size_t i = 0; i < sizeof(sgx_ec256_dh_shared_t); i++)
    {
        hash_buffer.shared_secret.s[i] = p_shared_key->s[sizeof(p_shared_key-
>s)-1 - i];
    }

    sgx_ret = sgx_sha256_init(&sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
        return false;
    }
    sgx_ret = sgx_sha256_update((uint8_t*)&hash_buffer, sizeof(hash_buffer_t),
sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_update((uint8_t*)&ID_U, sizeof(ID_U), sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
```

```
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_update((uint8_t*)&ID_V, sizeof(ID_V), sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_get_hash(sha_context, &key_material);
    if (sgx_ret != SGX_SUCCESS)
    {
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_close(sha_context);

    assert(sizeof(sgx_ec_key_128bit_t)* 2 == sizeof(sgx_sha256_hash_t));
    memcpy(first_derived_key, &key_material, sizeof(sgx_ec_key_128bit_t));
    memcpy(second_derived_key, (uint8_t*)&key_material +
sizeof(sgx_ec_key_128bit_t), sizeof(sgx_ec_key_128bit_t));

    // memset here can be optimized away by compiler, so please use memset_s on
    // windows for production code and similar functions on other OSes.
    memset(&key_material, 0, sizeof(sgx_sha256_hash_t));

    return true;
}

//isv defined key derivation function id
#define ISV_KDF_ID 2

typedef enum _derive_key_type_t
{
    DERIVE_KEY_SMK_SK = 0,
    DERIVE_KEY_MK_VK,
} derive_key_type_t;

sgx_status_t key_derivation(const sgx_ec256_dh_shared_t* shared_key,
    uint16_t kdf_id,
    sgx_ec_key_128bit_t* smk_key,
    sgx_ec_key_128bit_t* sk_key,
    sgx_ec_key_128bit_t* mk_key,
    sgx_ec_key_128bit_t* vk_key)
{
    bool derive_ret = false;

    if (NULL == shared_key)
    {
        return SGX_ERROR_INVALID_PARAMETER;
    }

    if (ISV_KDF_ID != kdf_id)
    {
        //fprintf(stderr, "\nError, key derivation id mismatch in [%s].",
__FUNCTION__);
        return SGX_ERROR_KDF_MISMATCH;
    }
```

```
    derive_ret = derive_key(shared_key, DERIVE_KEY_SMK_SK,
        smk_key, sk_key);
    if (derive_ret != true)
    {
        //fprintf(stderr, "\nError, derive key fail in [%s].", __FUNCTION__);
        return SGX_ERROR_UNEXPECTED;
    }

    derive_ret = derive_key(shared_key, DERIVE_KEY_MK_VK,
        mk_key, vk_key);
    if (derive_ret != true)
    {
        //fprintf(stderr, "\nError, derive key fail in [%s].", __FUNCTION__);
        return SGX_ERROR_UNEXPECTED;
    }
    return SGX_SUCCESS;
}
#else
#pragma message ("Default key derivation function is used.")
#endif

// This ecall is a wrapper of sgx_ra_init to create the trusted
// KE exchange key context needed for the remote attestation
// SIGMA API's. Input pointers aren't checked since the trusted stubs
// copy them into EPC memory.
//
// @param b_pse Indicates whether the ISV app is using the
//              platform services.
// @param p_context Pointer to the location where the returned
//                  key context is to be copied.
//
// @return Any error return from the create PSE session if b_pse
//         is true.
// @return Any error returned from the trusted key exchange API
//         for creating a key context.

sgx_status_t enclave_init_ra(
        int b_pse,
        sgx_ra_context_t *p_context) {
    // isv enclave call to trusted key exchange library.
    sgx_status_t ret;
#ifdef SUPPLIED_KEY_DERIVATION
    ret = sgx_ra_init_ex(&g_sp_pub_key, b_pse, key_derivation, p_context);
#else
    ret = sgx_ra_init(&g_sp_pub_key, b_pse, p_context);
#endif
    return ret;
}


// Closes the tKE key context used during the SIGMA key
// exchange.
//
// @param context The trusted KE library key context.
//
// @return Return value from the key context close API
```

```
sgx_status_t SGXAPI enclave_ra_close(
        sgx_ra_context_t context) {
sgx_status_t ret;
ret = sgx_ra_close(context);
return ret;
}


// Verify the mac sent in att_result_msg from the SP using the
// MK key. Input pointers aren't checked since the trusted stubs
// copy them into EPC memory.
//
//
// @param context The trusted KE library key context.
// @param p_message Pointer to the message used to produce MAC
// @param message_size Size in bytes of the message.
// @param p_mac Pointer to the MAC to compare to.
// @param mac_size Size in bytes of the MAC
//
// @return SGX_ERROR_INVALID_PARAMETER - MAC size is incorrect.
// @return Any error produced by tKE  API to get SK key.
// @return Any error produced by the AESCMAC function.
// @return SGX_ERROR_MAC_MISMATCH - MAC compare fails.

sgx_status_t verify_att_result_mac(sgx_ra_context_t context,
                                   uint8_t* p_message,
                                   size_t message_size,
                                   uint8_t* p_mac,
                                   size_t mac_size)
{
    sgx_status_t ret;
    sgx_ec_key_128bit_t mk_key;

    if(mac_size != sizeof(sgx_mac_t))
    {
        ret = SGX_ERROR_INVALID_PARAMETER;
        return ret;
    }
    if(message_size > UINT32_MAX)
    {
        ret = SGX_ERROR_INVALID_PARAMETER;
        return ret;
    }

    do {
        uint8_t mac[SGX_CMAC_MAC_SIZE] = {0};

        ret = sgx_ra_get_keys(context, SGX_RA_KEY_MK, &mk_key);
        if(SGX_SUCCESS != ret)
        {
            break;
        }
        ret = sgx_rijndael128_cmac_msg(&mk_key,
                                       p_message,
                                       (uint32_t)message_size,
                                       &mac);
        if(SGX_SUCCESS != ret)
        {
```

```
            break;
        }
        if(0 == consttime_memequal(p_mac, mac, sizeof(mac)))
        {
            ret = SGX_ERROR_MAC_MISMATCH;
            break;
        }

    }
    while(0);

    return ret;
}


// Generate a secret information for the SP encrypted with SK.
// Input pointers aren't checked since the trusted stubs copy
// them into EPC memory.
//
// @param context The trusted KE library key context.
// @param p_secret Message containing the secret.
// @param secret_size Size in bytes of the secret message.
// @param p_gcm_mac The pointer the the AESGCM MAC for the
//                 message.
//
// @return SGX_ERROR_INVALID_PARAMETER - secret size if
//         incorrect.
// @return Any error produced by tKE  API to get SK key.
// @return Any error produced by the AESGCM function.
// @return SGX_ERROR_UNEXPECTED - the secret doesn't match the
//         expected value.

sgx_status_t put_secret_data(
    sgx_ra_context_t context,
    uint8_t *p_secret,
    uint32_t secret_size,
    uint8_t *p_gcm_mac)
{
    sgx_status_t ret = SGX_SUCCESS;
    sgx_ec_key_128bit_t sk_key;

    do {
        if(secret_size != 8)
        {
            ret = SGX_ERROR_INVALID_PARAMETER;
            break;
        }

        ret = sgx_ra_get_keys(context, SGX_RA_KEY_SK, &sk_key);
        if(SGX_SUCCESS != ret)
        {
            break;
        }

        uint8_t aes_gcm_iv[12] = {0};
        ret = sgx_rijndael128GCM_decrypt(&sk_key,
                                         p_secret,
                                         secret_size,
```

```
                                        &g_secret[0],
                                        &aes_gcm_iv[0],
                                        12,
                                        NULL,
                                        0,
                                        (const sgx_aes_gcm_128bit_tag_t *)
                                           (p_gcm_mac));

        uint32_t i;
        bool secret_match = true;
        for(i=0;i<secret_size;i++)
        {
            if(g_secret[i] != i)
            {
                secret_match = false;
            }
        }

        if(!secret_match)
        {
            ret = SGX_ERROR_UNEXPECTED;
        }

        // Once the server has the shared secret, it should be sealed to
        // persistent storage for future use. This will prevents having to
        // perform remote attestation until the secret goes stale. Once the
        // enclave is created again, the secret can be unsealed.
    } while(0);
    return ret;
}

sgx_status_t generate_server_key(
    uint8_t *token_with_key,
    uint32_t secret_size)
{
    sgx_status_t ret = SGX_SUCCESS;
    if(secret_size != 32)//加密数据
    {
        ret = SGX_ERROR_UNEXPECTED;
        return ret;
    }
    uint8_t token[32] = {0};
    ret = sgx_read_rand(token, 32);//随机生成秘钥
    uint8_t aes_gcm_iv[12] = {0};
    sgx_aes_gcm_128bit_tag_t c_gcm_mac;
    //token用于server识别这个enclave已经有了sk_key,原理就是用client key加密以后server解密还是5到12

    do {

        ret = sgx_rijndael128GCM_encrypt(&sk_key,
                                        &token[0],
                                        secret_size,
                                        token_with_key,
                                        &aes_gcm_iv[0],
                                        12,
                                        NULL,
                                        0,
```

```
                                        &c_gcm_mac);
    } while(0);

    for(int i=0;i<16;i++)
    {
        token[i] = i+5;
    }
    return ret;
}
```

Service Provider

network_ra_client.cpp

提供socket通信服务

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include "network_ra.h"
#include "service_provider.h"
//add
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

char sendbuf[BUFSIZ];  //数据传送的缓冲区
char recvbuf[BUFSIZ];
int  client_sockfd;//客户端套接字



void PRINT_BYTE_ARRAY(
    FILE *file, void *mem, uint32_t len)
{
    if(!mem || !len)
    {
        fprintf(file, "\n( null )\n");
        return;
    }
    uint8_t *array = (uint8_t *)mem;
    fprintf(file, "%u bytes:\n{\n", len);
    uint32_t i = 0;
    for(i = 0; i < len - 1; i++)
    {
        fprintf(file, "0x%x, ", array[i]);
        if(i % 8 == 7) fprintf(file, "\n");
    }
    fprintf(file, "0x%x ", array[i]);
    fprintf(file, "\n}\n");
}
```

```
// Used to send requests to the service provider sample.  It
// simulates network communication between the ISV app and the
// ISV service provider.  This would be modified in a real
// product to use the proper IP communication.
//
// @param server_url String name of the server URL
// @param p_req Pointer to the message to be sent.
// @param p_resp Pointer to a pointer of the response message.

// @return int
// 修改成真正的网络通讯
int ra_network_send_receive(const char *server_url,
    const ra_samp_request_header_t *p_req,
    ra_samp_response_header_t **p_resp)
{
    FILE* OUTPUT = stdout;
    int ret = 0;
    int len = 0;
    int msg2len = 0;

    if((NULL == server_url) ||
        (NULL == p_req) ||
        (NULL == p_resp))
    {
        return -1;
    }
    switch(p_req->type)
    {

    case TYPE_RA_MSG0:
        memset(sendbuf, 0, BUFSIZ);
        memcpy_s(sendbuf, BUFSIZ, p_req, sizeof(ra_samp_request_header_t)+p_req-
>size);
        len = SendToServer(sizeof(ra_samp_request_header_t)+p_req->size);
        sleep(1);//等待起作用
        if (0 == len)
        {
            fprintf(stderr, "\nError,Send MSG0 fail [%s].",
                __FUNCTION__);
        }
        break;

    case TYPE_RA_MSG1:
        memset(sendbuf, 0, BUFSIZ);
        memcpy_s(sendbuf, BUFSIZ, p_req, sizeof(ra_samp_request_header_t)+p_req-
>size);
        ret = SendToServer(sizeof(ra_samp_request_header_t)+p_req->size);
        fprintf(stdout, "\nSend MSG1 To Server [%s].",__FUNCTION__);
        ret = RecvfromServer();
        msg2len = sizeof(ra_samp_response_header_t)+sizeof(sample_ra_msg2_t);

        memcpy_s(*p_resp, msg2len, recvbuf, ret);
        break;

    case TYPE_RA_MSG3:
        memset(sendbuf, 0, BUFSIZ);
        memcpy_s(sendbuf, BUFSIZ, p_req, sizeof(ra_samp_request_header_t)+p_req-
>size);
```

```
        ret = SendToServer(sizeof(ra_samp_request_header_t)+p_req->size);
        ret = RecvfromServer();
        memcpy_s(*p_resp, ret, recvbuf, ret);
        fprintf(stderr, "\nMsg3 ret = %d [%s].", ret);
        PRINT_BYTE_ARRAY(OUTPUT, *p_resp, ret);
        break;

    default:
        ret = -1;
        fprintf(stderr, "\nError, unknown ra message type. Type = %d [%s].",
            p_req->type, __FUNCTION__);
        break;
    }

    return ret;
}
int client(const char ip[16],int port)
{
    int len;
    struct sockaddr_in remote_addr; //服务器端网络地址结构体
    memset(&remote_addr,0,sizeof(remote_addr)); //数据初始化--清零
    remote_addr.sin_family=AF_INET; //设置为IP通信
    remote_addr.sin_addr.s_addr=inet_addr(ip);//服务器IP地址
    remote_addr.sin_port=htons(port); //服务器端口号

    /*创建客户端套接字--IPv4协议，面向连接通信，TCP协议*/
    if((client_sockfd=socket(AF_INET,SOCK_STREAM,0))<0)
    {
        perror("socket");
        return 1;
    }

    /*将套接字绑定到服务器的网络地址上*/
    if(connect(client_sockfd,(struct sockaddr *)&remote_addr,sizeof(struct
sockaddr))<0)
    {
        perror("connect");
        return 1;
    }
    printf("connected to server\n");
    return 0;
}

int SendToServer(int len)
{
    len=send(client_sockfd, sendbuf, len, 0);//发送
}

int RecvfromServer()
{
    /*接收服务端的数据*/
    int len = 0;
    len=recv(client_sockfd,recvbuf,BUFSIZ,0);
    if (len > 0 )
        recvbuf[len] = 0;
    return len;
}
```

```
int Cleanupsocket()
{
    close(client_sockfd);
    return 0;
}

// Used to free the response messages.  In the sample code, the
// response messages are allocated by the SP code.
//
//
// @param resp Pointer to the response buffer to be freed.

void ra_free_network_response_buffer(ra_samp_response_header_t *resp)
{
    if(resp!=NULL)
    {
        free(resp);
    }
}
```