UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Juliano Kasmirski Zatta

GATEWAY TSTP SEGURO USANDO INTEL SGX

Florianópolis

2019

Juliano Kasmirski Zatta

GATEWAY TSTP SEGURO USANDO INTEL SGX

Trabalho de Conclusão de Curso submetido ao Programa de Graduação em Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: Prof. Dr. Jean Everson Martina

Florianópolis

2019

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Zatta, Juliano Kasmirski
Gateway TSTP seguro usando Intel SGX / Juliano Kasmirski Zatta ; orientador, Jean Everson Martina, 2019.
102 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2019.

Inclui referências.

1. Ciências da Computação. 2. Gateway Seguro. 3. TSTP. 4. INTEL SGX. 5. Internet das Coisas. I. Martina, Jean Everson. II. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. III. Título.

Juliano Kasmirski Zatta

GATEWAY TSTP SEGURO USANDO INTEL SGX

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciências da Computação", e aprovado em sua forma final pelo Programa de Graduação em Ciências da Computação.

Florianópolis, 01 de novembro 2019.

Prof. José Francisco Danilo de Guadalupe Correa Fletes
Coordenador do Curso

Prof. Dr. Jean Everson Martina
Orientador

Banca Examinadora:

Prof. Dr. Antônio Augusto de Medeiros Fröhlich

Caciano dos Santos Machado

AGRADECIMENTOS

Agradeço à minha família e aos meus amigos, que sempre estiveram ao meu lado nesses longos anos da graduação. Dividindo bons momentos, incentivando que eu seguisse em frente, dando todo o suporte necessário e consolando nos momentos difíceis da vida.

Á minha mãe que me manteve durante a graduação e sempre se esforçou para que nunca faltasse nada e eu pudesse aprender o máximo possível. Ao meu irmão que deixou a esposa e filhos em casa algumas noites para conversar e me ajudar quando tive dúvidas a respeito da vida, quando precisei de ajuda por motivos quaisquer ou simplesmente para assistir uma partida de futebol e tomar uma cerveja.

Ao meu Pai (*in memoriam*), em especial, que não esteve presente durante toda a minha graduação, mas me deixou os ensinamentos de como ser um homem, justo, honrado, honesto e bem humorado em todos os momentos da vida.

Agradeço ao meus amigos Johnaltan, Gava, Tarciso e Leo, que sempre estiveram presentes durante a graduação. Compartilhando noites sem dormir fazendo trabalhos, lanches depois de horas de estudo, boas conversas na mesa do bar e momentos, que talvez sejam melhores não serem lembrados, das diversas festas que fomos.

Por fim agradeço ao Rafael Rodrigues e LabSmart onde fiz pesquisa durante dois anos da graduação. Ao LISHA onde tive o prazer de conhecer o Guto, suas ideias e sua cobrança, aproveitando para absorver o máximo de conhecimento sobre sistemas embarcados na minha curta passagem e ao Cancian pelas excelestes aulas, horas de conversa e auxílio antes e durante a graduação.

Ao LabSEC, o último laboratório que trabalhei, e aprendi muito sobre segurança da informação ao lado do Jean Martina e onde tive o prazer de fazer grandes amigos que levarei para a vida: Alek, Beavis, Perin e Pablo.

Um herói pode ser qualquer um, até mesmo um homem fazendo algo tão simples e reconfortante como colocar um casaco em torno dos ombros de um menino, para deixá-lo saber que o mundo não tinha terminado.

Batman

RESUMO

A presença cada vez maior de dispositivos em redes de sensores sem fio fez com que ampliasse a necessidade de protocolos seguros para a comunicação. Para melhorar esse cenário foi desenvolvido o Trustful Space-Time Protocol (TSTP), um protocolo leve e seguro para comunicação de dispositivos com baixa capacidade de processamento, este protocolo não se comunica com a internet e com isso surgiu a necessidade de desenvolver um nodo intermediário para enviar esses dados para um servidor externo através da internet. Este documento apresenta o desenvolvimento de um nodo executando em um ambiente seguro usando a tecnologia Intel Software Guard Extension, dentro de um enclave de memória decifrará as mensagens TSTP, cifrará usando TLS e enviará para um servidor remoto, também gerenciará as chaves criptográficas dos nodos sensores presentes na rede. Com isso qualquer computador, confiável ou não, poderá atuar como um gateway seguro para uma rede de sensores sem fio, sem comprometer confidencialidade e integridade dos dados gerados pelos sensores.

Palavras-chave: TSTP. SGX. IoT. Gateway Seguro.

ABSTRACT

The increasingly presence of devices in wireless sensor networks expanded the need of secure protocols to communicate those devices. To improve that scenario was developed the Trustful Space-Time Protocol (TSTP), a lightweight and secure protocol for communication of devices with low process capacity, this protocol does not communicate with the internet, so came the need of development a intermediate node to send the data generated from sensors to external server through internet. This text propose the development of a node executing inside safe environment using Intel Software Guard Extension. inside an enclave of memory it will decrypt the message from TSTP, encrypt using TLS and send it to a remote server, also it will manage all cryptographic keys from sensor nodes in the network. With this any computer, trustable or not, can actuate as a secure gateway to a wireless sensor network, without compromise the confidentiality or integrity of data generated by the sensors.
Keywords: TSTP. SGX. IoT. Secure Gateway.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

Tem-se criado diversas aplicações para Internet das Coisas (IdC), mas não foi dado muita ênfase à segurança da informação associada a estes dispositivos. Não são raros os casos de acessos indevidos a câmeras de segurança conectadas à Internet, também a fechaduras inteligentes e brinquedos, que são explorados e têm seus comportamentos alterados. Redes de sensores sem fio também estão suscetíveis a ataques tanto para espionar dados gerados quanto para alterá-los, violando a integridade, confidencialidade dos dados enviados aos servidores.

Diversos ataques a dispositivos desenvolvidos para aplicações específicas são desenvolvidos inclusive para transformar estes dispositivos em robôs para ataques em massa, onde se destaca o ataque de negação de serviço, que pode fazer um servidor ficar inacessível temporariamente devido à sobrecarga.

Visando cobrir uma lacuna em protocolos leves e seguros desenvolvidos para IdC e Redes de Sensores Sem Fio (RSSF) foi desenvolvido o *Trustful Space-Time Protocol* (TSTP) que segundo **??**) é um protocolo *cross-layer*, eficiente e com características confiáveis de tempo, localização e segurança e é compatível com dados no sistema internacional de unidades.

Os nodos presentes nesta RSSF enviam seus dados a um *gateway* e este, por sua vez, envia para um servidor, que processará e armazenará esses dados. Este *gateway*, atualmente é um dispositivo embarcado com baixa capacidade de processamento e pouca memória. Portanto, há dificuldades para processar as chaves criptográficas com a velocidade necessária e não há memória suficiente para guardar chaves trocadas com dezenas de nodos, que podem estar presentes em uma RSSF.

Para garantir propriedades básicas de segurança, mesmo em um ambiente não confiável, será usado uma tecnologia recente, desenvolvida pela Intel, o *Intel Software Guard Extension* (SGX), o qual será apresentado, em detalhes, neste documento.

1.1 OBJETIVOS GERAIS

Implementar um *gateway* intercomunicando o TSTP para a RSSF e HTTPS para se comunicar com o servidor na nuvem. O sistema garantirá os quatro atributos básicos da segurança da informação: integridade, confidencialidade, não-repúdio e autenticidade, na comunicação entre os nodos e a nuvem.

1.2 OBJETIVOS ESPECÍFICOS

1. Projetar e desenvolver um *gateway* funcional;

2. Fazer com que este *gateway* execute em um ambiente seguro;

3. Analisar possíveis vulnerabilidades

1.3 ESTRUTURA DO DOCUMENTO

No capítulo 2 são abordados os conceitos e ferramentas usados para o desenvolvimento deste trabalho. No capítulo 3 a proposta de como o *gateway* seria desenvolvido. No capítulo 4 é mostrado como foi feito o desenvolvimento do *gateway*, como foram utilizadas as ferramentas, análises feitas em cima do trabalho desenvolvido, pontos fortes do sistema e possíveis problemas de segurança. No capitulo 5 são conclusões do trabalho e sugestões de trabalhos para o futuro.

2 FUNDAMENTAÇÃO

2.1 REDES DE SENSORES SEM FIO

A miniaturização de dispositivos eletrônicos e uma redução considerável no consumo de energia, possibilitou o uso de sistemas inteligentes em quaisquer áreas. O baixo consumo de energia possibilita com que sensores sejam colocados nos mais diversos ambientes alimentados apenas por uma bateria durante meses ou anos, sendo esporádicas suas trocas. Outra possibilidade é alimentar esses sistemas com um conjunto composto por painéis solares e baterias aumentando a autonomia destes dispositivos para dezenas de anos.

Nos ambientes residenciais, comerciais e industriais se tem instalado cada vez mais sensores de todos os tipos: umidade, temperatura, pressão, luminosidade, consumo elétrico, dentre outros. Mas é necessário que esses dados sejam acessíveis para possibilitar simples observações e análises mais elaboradas em tempo real desses dados.

Esta RSSF pode ser feita usando diversas camadas físicas, dentre elas se destaca o uso do padrão IEEE 802.15.4, cujo caso de testes foi desenvolvido anteriormente.

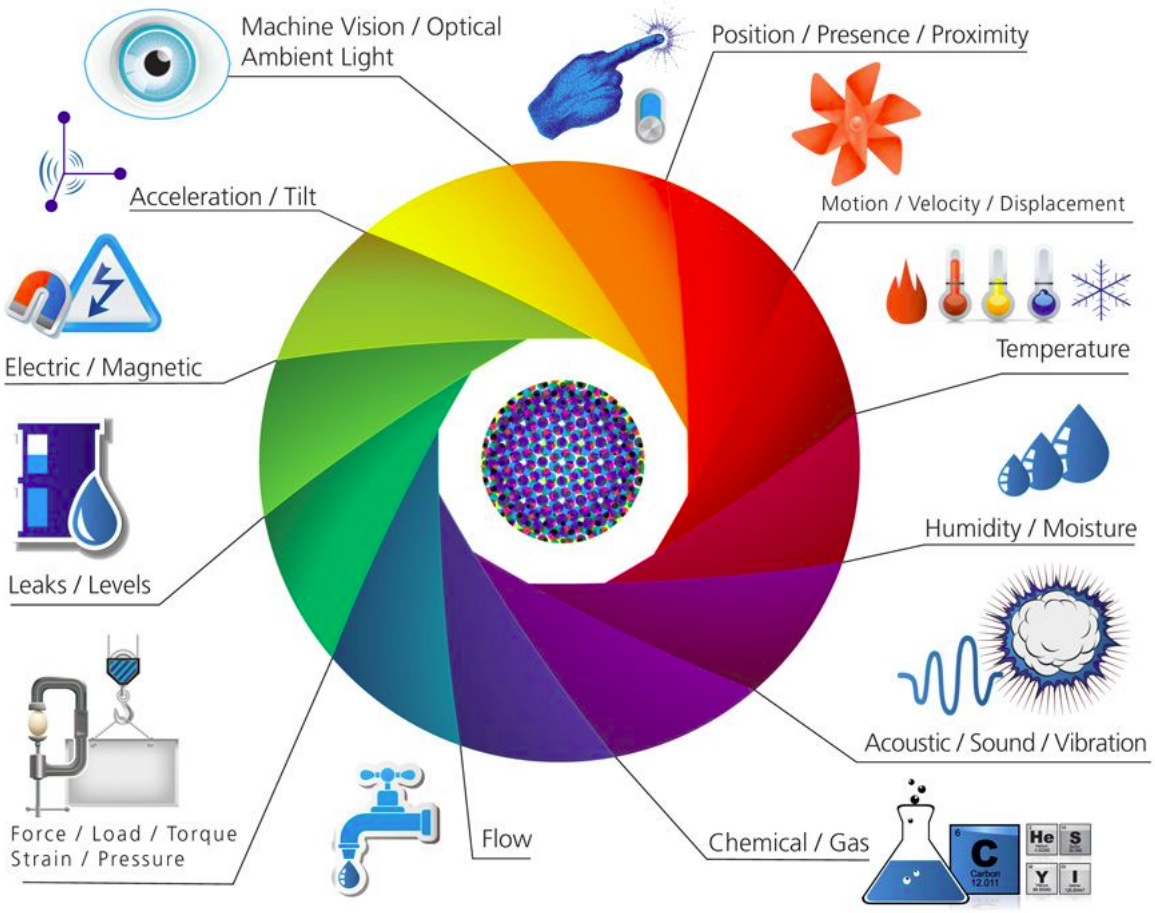
Considerando a rede mundial de computadores, a Internet, tem-se cunhado o termo Internet das Coisas (IdC), onde pequenos sistemas microcontrolados interagem com o ambiente ao seu redor, atuando e observando. Um conjunto destes sistemas pode desempenhar tarefas importantes como o controle dos semáforos de um cruzamento e controle de temperatura de uma casa, como exemplos. Para ser considerado como IdC estes dispositivos têm de se conectar à Internet. Diversos elementos medidos por sensores podem ser vistos na Figura 1.

Por serem dispositivos com baixa capacidade de processamento e de baixo consumo de energia são empregadas Redes de Sensores Sem Fio (RSSF) para enviar esses dados aos outros nodos da rede ou às centrais de processamento, com uma maior capacidade de processamento, onde esses dados serão tratados.

São desenvolvidos vários protocolos e padrões para RSSF. Normalmente, adotam-se protocolos leves para reduzir ainda mais o consumo de energia dos sensores e atuadores. Nesse aspecto se descarta o uso de TCP/IP e UDP/IP, usados na Internet, já que são protocolos mais onerosos e demandam uma maior capacidade de processamento e, consequentemente, maior consumo de energia. Um exemplo de protocolo para essas RSSF é o *Trustful Space-Time Protocol* (TSTP), que será explicado na seção seguinte.

Entre a internet e as RSSF é necessário a presença de um sistema intermediário que faça as traduções das mensagens entre o TSTP e a Internet, o *gateway*.

Figura 1 – Exemplo de tipos de sensores usados em redes IdC.



Fonte: (**??**).

2.2 TRUSTFUL SPACE-TIME PROTOCOL

É um protocolo que não separa suas tarefas em diferentes camadas, ou seja, um protocolo *cross-layer*. Este protocolo suporta inúmeros nodos, todos se intercomunicando e com *gateway* espalhados espacialmente para enviar esses dados para o servidor.

É baseado em localização e tempo, com essas informações faz-se o roteamento das mensagens dos sensores até o *gateway* e vice-versa. Outra característica importante presente neste protocolo é a segurança, garantindo a confidencialidade, autenticidade, integridade e não repúdio. O nodo sensor estabelece uma comunicação segura com o *gateway* e se autentica no mesmo.

Inicialmente cada sensor possui uma localização, relativa ao *gateway* e, ao ser ligado, sincroniza seu relógio com a RSSF, mantendo um tempo único para todos os nodos da rede, este tempo é ditado pelo *gateway*, que possui conexão com dispositivos externos à rede e se mantém sincronizado.

O roteamento é feito baseado na localização, usando um algoritmo guloso, onde

o nodo mais próximo do destino retransmite o pacote até chegar no nodo destino, um sensor ou o *gateway*. A distância entre o nodo e o destino é calculada traçando uma linha entre o nodo atual e destino. A mensagem é adicionada a uma fila de retransmissão, que e esta será feita apos um período de tempo proporcional à distancia do nodo para o destino, logo nodo mais próximo retransmite a mensagem primeiro e os demais, ouvindo a retransmissão, retiram o pacote da sua agenda de envio.

2.2.1 Segurança

O protocolo prevê mecanismos de segurança da informação, garantindo quatro propriedades básicas: confidencialidade, autenticidade e não-repúdio. Uma das características de segurança é o uso de uma base de tempo sincronizada com o *gateway* para que seja possível a comunicação, portanto a sincronia dos relógios deve ocorrer antes da inicialização da camada de segurança. O processo para estabelecer uma comunicação segura pode ser dividida em algumas etapas: inicialização, estabelecimento de chaves, autenticação do nodo e comunicação segura.

2.2.1.1 Inicialização

Esta etapa consiste com estabelecer, previamente, um id único para cada nodo sensor da rede e o id de todos os sensores de uma rede ser carregado de maneira segura no *gateway*. Normalmente se usa informações extraídas do dispositivo de hardware, como um *hash* do numero de série do chip, estabelecido pelo fabricante.

2.2.1.2 Estabelecimento de chaves

As chaves criptográficas são estabelecidas com o *gateway* usando método de Diffie-Hellman. Este método permite que sejam estabelecidas chaves simétricas conhecidas apenas pelo *gateway* e pelo sensor mesmo conversando por um canal público, estas chaves são chamadas de *Master Secret*, por **??**). Este método é suscetível ao ataque do homem do meio, mas que é mitigado no processo de autenticação do nodo sensor.

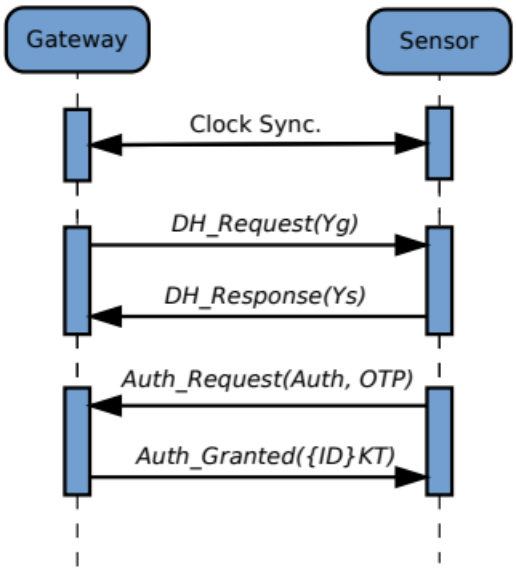
2.2.1.3 Autenticação do nodo

Após um nodo estabelecer uma *Master Secret* com o *gateway* é feita sua autenticação. No processo de autenticação é gerada uma requisição de autenticação, esta requisição contém um código de autenticação derivado do id único do sensor e uma *One-Time Password* (OTP). O *gateway* valida o código de autenticação do id único na sua base de dados

e verifica se o OTP enviado pelo sensor é válido. O OTP é um código de autenticação de mensagens derivado do *Master Secret*, um *timestamp* estabelecido com o *gateway* e o id único do sensor, usando o algoritmo poly-1305.

Após validado o *gateway* retorna uma mensagem de autenticidade concedida, com o id único do sensor cifrado usando uma chave derivada do *Master Secret*. Como apenas o *gateway* e o sensor conhecem o id único, o sensor pode confiar que está com uma comunicação segura estabelecida com o *gateway*. Uma visão geral dessa comunicação pode ser vista na Figura 2.

Figura 2 – Visão geral das mensagens trocadas para o estabelecimento de chaves.

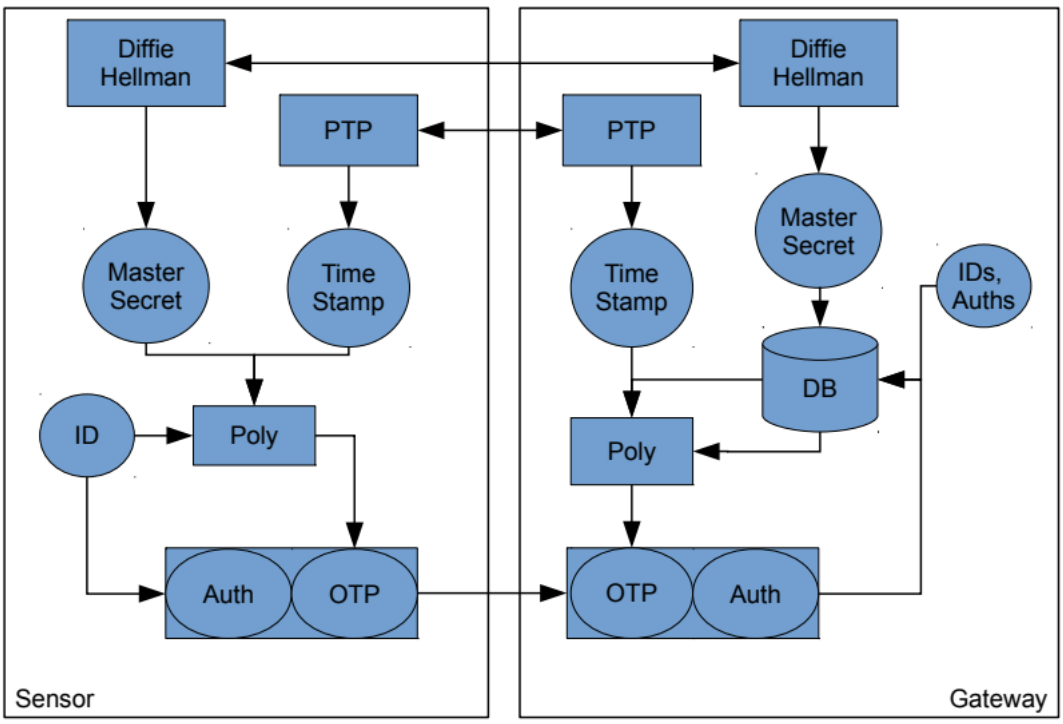


Fonte: (**??**).

2.2.1.4 Comunicação segura

Após a *Master Secret* ser estabelecida e autenticada, podem ser feitas comunicações seguras entre o sensor e *gateway*. Para isso as mensagens são cifradas usando um algoritmo de cifragem simétrica o *Advanced Encryption Standard* (AES) de 128 bits. A chave simétrica usada no AES é derivada: do *Master Secret*, do id e do *timestamp*. Todas as mensagens possuem um código de autenticação usando o poly-1305 para garantir a integridade da mensagem transmitida. A Figura 3 mostra uma visão geral da interação entre os blocos do protocolo.

Figura 3 – Visão geral dos blocos do protocolo.



Fonte: (**??**).

2.2.2 Smart Data

Uma característica que está fortemente associada ao TSTP é o *smart data*, um sistema de troca de mensagens onde um transdutor é, apenas, instanciado, seja local ou remoto. Para o remoto é passando uma região de interesse e o *smart data* localiza e solicita os dados para o nodo sensor que possui o transdutor local, caso exista. O sensor, ciente do interesse, responde com os valores solicitados. Funcionando com qualquer tipo de sensor ou nodo sensor que esteja conectado à RSSF, desde que esteja usando o TSTP.

2.3 INTEL SOFTWARE GUARD EXTENSION

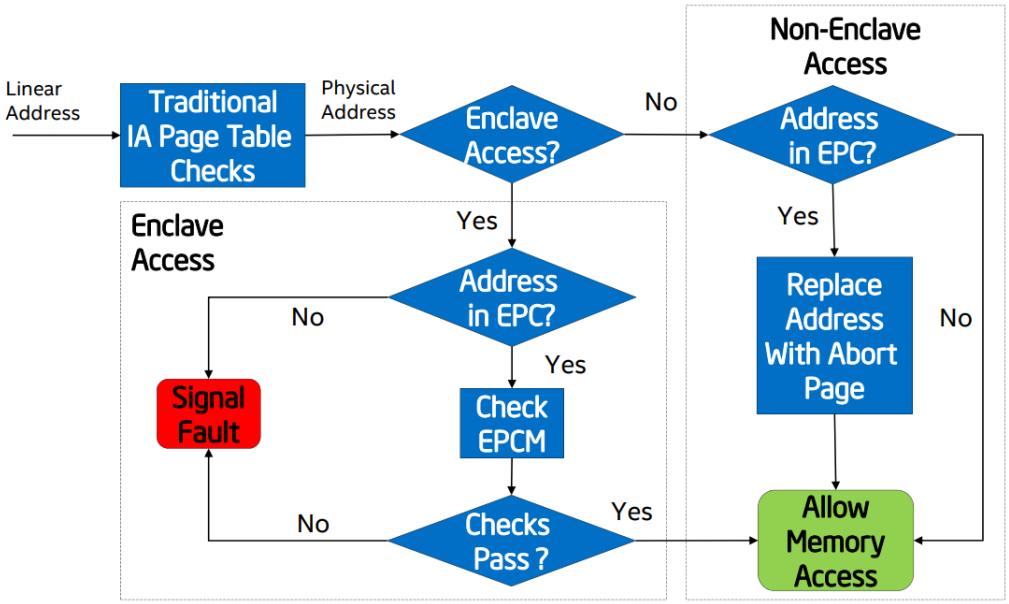
O *Intel Software Guard Extension* (SGX) é uma extensão do conjunto de instruções dos processadores da Intel e permite que uma aplicação seja executada em nível de usuário de maneira segura, mesmo em uma plataforma não confiável, como sistema operacional, máquina virtual ou BIOS infectados, um computador remoto não confiável e etc. Ele faz isso isolando o código e os dados do ambiente externo, como destacam **??**).

2.3.1 Funcionamento

Este conjunto de instruções provê ao desenvolvedor um enclave, uma região de memória que está contida na memória da plataforma em questão, mas inacessível por outros processos executando na mesma plataforma. Esse isolamento é feito com um hardware semelhante ao hardware que verifica se uma página pertence ao processo em execução e permite que essa página seja acessada, mas com algumas diferenças.

Funciona como mais uma camada de hardware verificando se o endereço físico requisitado pela CPU é um endereço contido na região de memória reservada para os enclaves e se a página solicitada pertence ao enclave que o alocou, caso a região solicitada não pertença ao enclave é gerado um sinal de falta e o processo pode ser abortado, porém se a região pertencer ao enclave é permitido o acesso àquela região da memória pela CPU. O fluxograma de verificação pode ser visto na Figura 4.

Figura 4 – Fluxograma de acesso aos dados de um enclave.



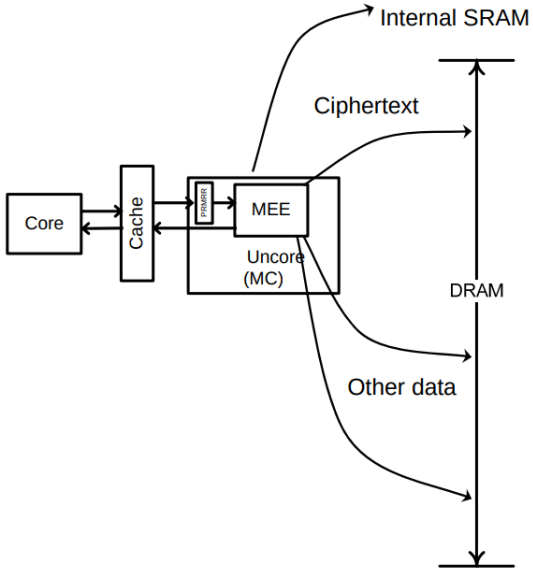
Fonte: (**??**).

Os dados ficam em texto plano dentro da pastilha de silício nas caches da CPU, sendo protegidos de acessos indevidos apenas pelo hardware que faz as verificações de acessos. Os dados mais usados são rapidamente acessados e não atrapalham o desempenho de um processo executando em um enclave, mas os deixa suscetíveis a ataques tipo *side-channel*.

Quando os dados não são usados recentemente eles podem ser retirados da cache da CPU, onde entra em atuação outra parte em hardware do SGX, o *Memory Encryption Engine*, que opera junto ao controlador de memória. Antes de enviar os dados à DRAM o *Memory Encryption Engine* cifra os dados usando AES-CTR e gera códigos de autenticação para cada página de memória, salvando em uma estrutura de dados dentro do

chip da CPU. As chaves criptográficas também ficam armazenadas na CPU. Para que os enclaves tenham disponíveis mais memória é feita uma hierarquização em forma de árvore desse processo fazendo com que apenas os dados do nodo raiz dessa cache fiquem na CPU, trazendo da DRAM os nodos intermediários quando necessário, sendo os nodos folha os dados usados pelos enclaves. Na figura 5, é visto que o *Memory Encryption Engine* fica entre a cache da CPU e a DRAM, cifrando os dados do enclave.

Figura 5 – Memory Encryption Engine.



Fonte: (**??**).

2.3.2 Modelo de programação

Para programar um enclave a Intel simplificou bastante o processo. O software é compilado como uma biblioteca dinamicamente ligada, sendo necessário apenas que o programador especifique as funções da fronteira do enclave sendo as chamadas gerenciadas pela por uma API disponibilizada. Existem dois tipos de chamadas *ecalls* e *ocalls*, que são chamadas de fora do enclave para dentro e vice-versa, respectivamente.

Recursos disponibilizados e gerenciados pelo sistema operacional não estão disponíveis dentro de um enclave, pois para isto seria necessário que o sistema tivesse acesso ao ambiente seguro. Por isso não há uma noção de horário dentro do enclave ou *timers* disponíveis, acesso a recursos de entrada e saída e nem chamadas de sistema. Para usar esses recursos o usuário deve definir e implementar chamadas de funções de código em área não segura, sendo responsável por características de segurança, como cifrar os dados antes de enviar para fora do enclave.

A biblioteca compilada é assinada usando uma chave RSA, disponibilizada pela Intel, e no momento da inicialização do enclave o hardware verifica sua autenticidade

para impedir que o código do enclave seja adulterado.

2.4 TLS

O Transport Layer Security (TLS) é um protocolo de comunicação seguro, ele define um conjunto de algoritmos criptográficos simétricos, assimétricos e autenticação de mensagens, define também mensagens de controle para negociação dos algoritmos e chaves criptográficas e mensagens para proteger o tráfego de dados entre os pares. O objetivo desse protocolo é fornecer um canal de comunicação seguro entre dois pares de comunicação, segundo **??**). Ele visa garantir integridade, confidencialidade e autenticidade da informação trafegada através dele. Funciona como uma camada intermediária entre a camada de aplicação e a camada de transporte no modelo da Internet. Sendo completamente transparente para a aplicação que esteja funcionando uma camada acima deste protocolo.

2.4.1 mbedTLS

O mbedTLS é uma biblioteca mantida pela ARM que implementa o protocolo TLS, como citado no site da biblioteca é uma biblioteca fácil de entender, usar, integrar e expandir. E por isso foi escolhida para o projeto. Por ela funcionar em sistemas embarcados, com ou sem sistema operacional torna-se ideal para integrar dentro do enclave. Para integração basta redefinir algumas funções, como a criação de um socket, que é feita usando funções de borda e solicitando para o sistema operacional não confiável. Outra necessidade é a implementação da geração de uma semente para os geradores de números aleatórios, funcionalidade que é disponibilizada dentro do enclave. Foi encontrada um repositório público já com as mudanças feitas, portanto não há necessidade de reimplementar, bastando reutilizar.

2.4.2 X.509

É um padrão que define o formato dos certificados de chaves públicas. Os certificados são construídos e assinados em um formato de árvore, em que o nodo raiz auto assina seus certificados. Os nodos um nível abaixo tem seus certificados assinados por um nodo imediatamente acima, sendo o nodo raiz o nível zero. Nodos nível 1 são assinados pelo nodo raiz, os nível dois são assinados por um nível 1 e assim sucessivamente.

Nos certificados são colocadas informações como versão do certificado, tipo de cifra usada, data de validade, número de série, informações de localização (país, estado, cidade e endereço), empresa para qual foi atribuída o certificado, um nome para o certificado, chave pública do sujeito e informações do emissor do certificado e sua assinatura. Na

Figura 6 podemos ver estas informações extraídas de um certificado enviado ao acessar o site principal da UFSC, nele podemos ver informações como nome, o emissor, datas de validade, chave pública, dentre outros.

Figura 6 – Certificado UFSC.

```
Version: 3 (0x2)
Serial Number:
    6b:89:ea:d2:8e:84:02:2d:7d:32:e4:3b
Signature Algorithm: sha256WithRSAEncryption
Issuer: C = BR, ST = Rio de Janeiro, L = Rio de Janeiro, OU = Gerencia de Servicos (GSer), O = Rede Nacional de Ensino e Pesquisa - RNP, CN = ICPEdu
Validity
    Not Before: Jun 11 13:31:05 2018 GMT
    Not After : Jun 11 13:31:05 2020 GMT
Subject: C = BR, ST = SC, L = Florianopolis, O = UNIVERSIDADE FEDERAL DE SANTA CATARINA, CN = *.ufsc.br
Subject Public Key Info:
    Public Key Algorithm: rsaEncryption
        RSA Public-Key: (2048 bit)
        Modulus:
            00:bf:66:6b:68:70:ec:a0:f7:79:09:8a:2d:c1:43:
            b8:8a:30:d3:28:47:8b:46:b6:64:ef:f8:1d:5d:ee:
            bb:be:9a:50:ff:bd:fa:b4:10:a2:b3:31:d5:ed:85:
            f1:2d:44:64:43:9d:e3:f1:70:70:16:0b:5b:9e:b3:
            a9:71:87:e9:37:9b:9f:be:8d:ab:fd:a5:0d:ce:88:
            b1:3d:42:18:3a:d5:a9:09:fe:4c:c0:0d:ff:2a:35:
            fa:5f:71:7e:5c:17:b5:28:94:07:eb:60:1f:cd:32:
            f8:ca:1a:d4:d4:cf:70:c0:e7:29:ce:8b:b4:8e:dd:
            0a:d1:ab:29:83:c9:79:4a:8b:e3:25:91:b8:af:27:
            cb:7a:c2:f1:2b:4b:f2:eb:28:cb:a1:2e:a8:98:46:
            b9:83:4c:9c:a7:c1:10:59:52:ee:af:a2:ef:9e:df:
            34:1e:d0:48:e8:6d:f1:9f:66:06:58:82:eb:10:fc:
            9b:d8:13:aa:c1:2c:31:bf:bf:13:9c:e3:92:55:e4:
            b3:d4:71:b1:b5:2c:8b:81:de:82:2c:57:ae:d6:a8:
            c0:d9:73:02:5d:0a:1a:17:39:63:d7:81:73:73:1f:
            1b:f6:de:41:8e:bd:56:2e:2e:81:0b:79:6c:07:6e:
            a7:b4:b3:59:ae:94:a9:64:b1:6c:6f:28:f0:bb:c0:
            a6:29
        Exponent: 65537 (0x10001)
```

Fonte: Próprio autor.

O Sujeito que quer se autenticar envia seu certificado e toda a cadeia de certificados que assinaram o seu até o nodo raiz. A entidade que quer verificar a autenticidade do certificado verifica se o certificado raiz está na lista de certificados confiáveis e confere as assinaturas de todos os certificados da cadeia, autenticando o nodo folha da árvore.

2.5 CASTALIA

Neste projeto foi usado o Castalia um framework para simulação de redes de sensores sem fio, já que o gateway opera se comunicando com uma RSSF. Este framework foi desenvolvido usando o OMNet++. O OMNet++ é uma biblioteca para simulação de eventos discretos, desenvolvida primeiramente para simulação de redes de computadores e é baseada em componentes, modular e expansível, como consta no site da biblioteca.

3 PROPOSTA

Para permitir que os dados gerados por nodos de uma rede de sensores sem fio cheguem aos servidores é necessária a aplicação de um *gateway* que faça a conversão dos protocolos entre a RSSF e a Internet. Este pode executar em um plataforma confiável, porém demanda mais recursos já que é necessário empregar um computador dedicado para esta aplicação.

Outra possibilidade é que o *gateway* execute em uma plataforma não confiável, mas dentro de um ambiente confiável. Para isso é proposto que este seja desenvolvido usando a extensão SGX, para garantir que os dados não percam sua confidencialidade, quando aplicável, e nem sua integridade.

O *gateway* contará com um dispositivo de hardware, que receberá o sinal da RSSF e enviará os dados para dentro do enclave, este os decifrará e os enviará ao servidor. O hardware usado pode ser o mesmo que o de um nodo sensor usado na rede, já que conta com a mesma tecnologia de rádio. Algumas tarefas da rede continuarão sob a responsabilidade do hardware que atuará junto ao SGX, são elas: geolocalização, o relógio sincronizado e o roteamento de mensagens. Estas informações trafegam em texto plano nas mensagens do protocolo e, portanto, não influenciam na segurança associada ao *gateway*. Mantendo o enclave com o mínimo de funções é um fator importante para diminuir a complexidade do programa e minimizar os pontos de ataque.

3.1 TROCA DE MENSAGENS

Mensagens destinadas ao *gateway* são enviadas, por um canal inseguro, para dentro do SGX, onde serão tratadas. Haverá, inicialmente, a troca de chaves e será gerada a *Master Secret* que ficará salva no enclave. A etapa seguinte, a autenticação, envolverá o servidor para onde o enclave enviará a solicitação de autenticação emitida pelo sensor e o servidor verificará na sua base de dados e responderá ao enclave que o nodo é válido na rede.

Estabelecida a conexão o servidor poderá manifestar interesse em algum dado que pode ou não estar presente naquela RSSF para o *gateway*, ele, que por sua vez, empacotará em um *smart data* e enviará esse interesse ao sensor que responderá com os dados lidos. Então esses dados serão tirados do *smart data* e enviados à ao servidor. O enclave e o servidor funcionarão no modelo cliente-servidor, onde o enclave atuará como cliente. Ele deverá estabelecer uma comunicação segura com o servidor usando HTTPS e enviar os dados e solicitações de autenticação de sensores através de uma interface disponibilizada pelo servidor.

4 DESENVOLVIMENTO

O *gateway* foi pensado e desenvolvido para operar dentro de um enclave de memória e evitar que dados sensíveis sejam expostos. Como característica do enclave não há possibilidade de acesso ao hardware, aos mediadores de hardware e nem a funcionalidades do sistema operacional em que o enclave esteja executando. Assim, os dados ficam protegidos mesmo que algum componente do computador esteja comprometido (hardware, sistema operacional, virtualizador e outros processos são alguns componentes).

Desde o princípio o *gateway* foi pensado para possuir apenas as funcionalidades estritamente necessárias ao seu funcionamento, diminuindo os pontos de possíveis ataques. Para isso, algumas partes do protocolo TSTP foram mantidas fora do enclave, essas partes navegam em texto plano e mantê-las fora não prejudica a segurança do *gateway*, pelo contrário, diminui a complexidade do enclave.

O projeto foi feito usando como base o mesmo dispositivo de hardware de um nodo sensor para fazer funcionar como uma ponte entre a camada física da RSSF para a o enclave. Como o TSTP já foi implementado para executar nesse dispositivo tarefas como geolocalização, temporização e roteamento já estão implementadas. Como essas tarefas estão bastante associadas à rede sem fio e não demandam grande quantidade de recursos, foram mantidas nesses dispositivos ao invés de serem levadas ao *gateway*. Apenas a informação sensível é repassada ao *gateway* para que este desempenhe sua tarefa.

Ele foi implementado para ser um processo em um sistema operacional Linux e se comunicar com o dispositivo de hardware através de uma interface serial disponibilizada pelo sistema e facilmente portável para outras interfaces. O processo do *gateway* envia os dados dessa comunicação o enclave e passa a execução para dentro do enclave. Devido ao enclave não possuir acesso a dispositivos de hardware do computador e nenhum controle sobre sua execução alguns recursos não estão disponíveis dentro do enclave, como é o caso dos relógios de tempo, a execução dentro do enclave não possui nenhuma base de tempo confiável, sendo necessário enviar essas informações para dentro dele.

Para execução dos testes o dispositivo de hardware foi simulado usando o OMNet++ com Castalia. Os dados a serem enviados ao enclave são enviados e recebidos usando uma interface serial simulada no Linux. Este simulador é desenvolvido para de RSSF, como é o caso do TSTP, o protocolo usado neste trabalho, e tem seu funcionamento igual ao de uma rede sem fio física, mas com parâmetros bem definidos e sem interferências de fatores externos.

4.1 COMUNICAÇÃO NA REDE SEM FIO

Na inicialização o dispositivo simulado define o tempo da rede, baseado em uma informação recebida do *gateway*. A partir desse momento ele mantém o controle da base de tempo da rede. Todos os nodos sensores que se conectarem à rede usarão essa base de tempo para aferir os tempos das medições, estabelecer a comunicação com o *gateway* e estabelecer as chaves criptográficas usadas na comunicação.

Os dados são recebidos pelo dispositivo através da rede sem fio simulada. Ele abre o pacote de dados e analisa as informações não cifradas e envia para o *gateway* através da interface serial, se este for o destinatário do pacote recebido. Se os dados não tiverem o *gateway* como destinatário o dispositivo se encarrega de descartar o pacote ou reenviar para a rede, funcionando apenas como um roteador.

4.1.1 Envio dos dados para o *gateway*

Por se tratar de um protocolo *cross-layer*, o pacote é movido entre suas camadas, mas a camada de segurança não é processada no dispositivo de hardware e sim enviada ao enclave. Para o envio dos dados para o enclave a classe que implementa a segurança foi substituída por um *proxy* que envia os dados para o enclave e recebe uma resposta.

A implementação dentro do enclave recebe os dados e retorna a resposta de status para o dispositivo de hardware dar continuidade aos tratamentos que não envolvem o dado, como destruir as estruturas alocadas na memoria e etc. De dentro do enclave também partem solicitações, de dados do dispositivo de hardware, como solicitação do horário da rede, geolocalização do *gateway* e envio de uma mensagem para a rede, além de chamadas para controle interno do TSTP no dispositivo, que serão apresentadas na seção seguinte.

4.2 ENCLAVE

O enclave recebe os dados do dispositivo de hardware e da sequência à camada seguinte do protocolo.

Na primeira conexão o enclave realiza uma requisição de Diffie-Hellman enviando a sua chave pública, o nodo sensor recebe a requisição e envia uma resposta com a sua chave pública para o enclave. Feito isso os dois estabeleceram o *Master Secret* e o enclave guarda este segredo em uma lista de nodos não autenticados.

O nodo sensor então faz uma solicitação de autenticação enviando o seu código de autenticação, que é um *hash* derivado do seu identificador único e a OTP, derivada do seu identificador, base de tempo da rede e do *Master Secret*. O enclave confere se o código de autenticação está na sua base de dados e associa a *Master Secret* ao respectivo

nodo sensor, esta associação é feita removendo o *Master Secret* da lista de nodos não autenticados e colocando em uma lista de nodos confiáveis, em seguida envia para o nodo sensor uma mensagem de autenticação deferida, que contém o identificador do sensor cifrado com a OTP.

Como apenas o *gateway* e o nodo sensor conhecem o identificador único do sensor, isso quer dizer que foi estabelecida uma conexão ponto-a-ponto entre o *gateway* e o sensor. Os identificadores dos nodos sensores são adicionados pelo dispositivo de hardware simulado, já que este conhece todos os nodos instanciados na simulação e não foi definida uma interface com o servidor para tráfego desses identificadores. É importante ressaltar que o identificadores não podem ficar fixos no código do enclave, pois este não é cifrado, é apenas assinado, e pode ser visualizado por qualquer pessoa ou computador que tenha acesso executável do enclave.

Após estabelecida a segurança na comunicação o *gateway*, demonstra interesse em um dado através da construção de um *smartdata* e envia para o nodo sensor. O interesse é feito dizendo o tipo de dado desejado (temperatura, por exemplo), a área em que o dado desejado pode ser amostrado e o tempo que essa informação é relevante, que pode ser um intervalo de tempo bem definido ou até o *gateway* ativamente enviar uma mensagem dizendo que não é mais necessário o envio desses dados. O interesse poderia partir do servidor, mas também não foi projetada nenhuma interface entre o servidor e o *gateway*, portanto os interesses ficam fixos no código do enclave.

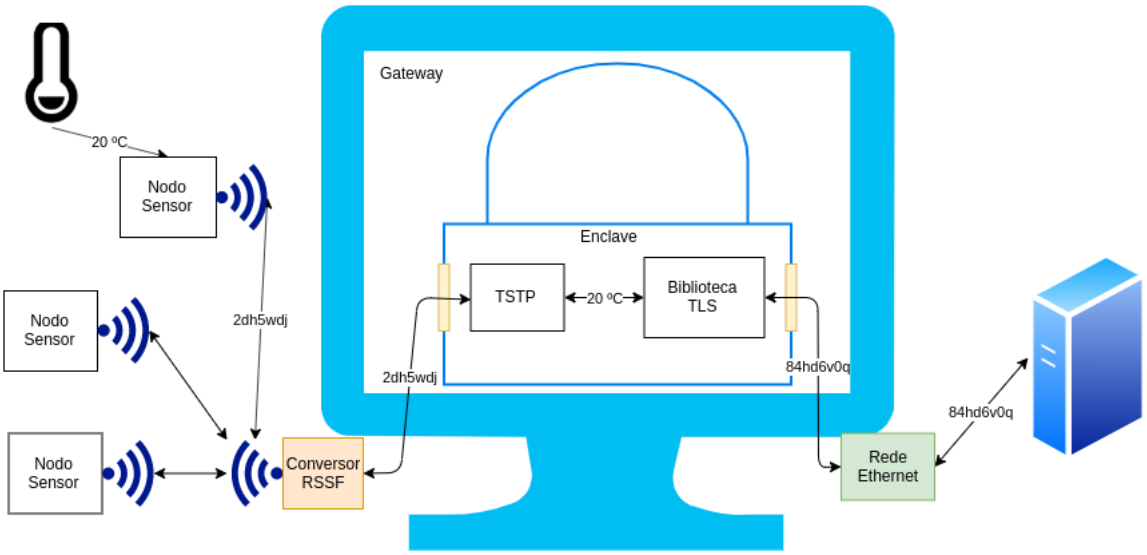
O nodo sensor que tenha o dado desejado, esteja dentro da área de interesse e esteja autenticado com o enclave responde com os dados solicitados cifrando e assinando esses dados. O enclave os recebe, decifra esses dados do TSTP, monta um pacote com a interface definida pelo servidor, estabelece uma conexão segura usando TLS e envia para ele. Esse fluxo de dados pode ser visto na Figura 7, usando uma amostra de temperatura como exemplo.

4.2.1 Funções da fronteira

O enclave deve ter bem definidas as funções da fronteira do enclave, pois essas são o ponto em que os dados entram e saem de um enclave. O principal ponto de ataque de um enclave são usando essas funções. Uma função mal definida pode permitir que todos os dados dentro de um enclave sejam vazados para fora, inutilizando todo o trabalho de construir em um ambiente seguro. As funções da fronteira funcionam como ponto de entrada para o enclave e são representadas pelas caixas amarelas na Figura 7.

Uma maneira de buscar diminuir esses pontos de ataque é deixando essa interface o mais simples e minimalista quanto possível. Para isso foram criadas apenas algumas chamadas bastante específicas. São duas funções de inicialização, para o TLS e o TSTP. Também há duas duas funções para executar o código, estas são chamadas por duas

Figura 7 – Fluxo de dados no gateway.



Fonte: Próprio autor.

threads separadas, uma faz a comunicação com o servidor e a outra faz a comunicação com o dispositivo de hardware. Estas funções são chamadas de fora do enclave e executadas dentro. Além dessas foram colocadas funções auxiliares que são chamadas de dentro do enclave, são as funções para inicialização, finalização, leitura e escrita em uma interface serial. E, na biblioteca TLS, funções para conectar, escutar, enviar e receber dados de um *socket*, dentre outras.

4.2.2 Troca de mensagens pela interface serial

Os dados no enclave são rebebidos através da chamada da função de leitura da interface serial e enviados pela função de escrita. Para facilitar a comunicação foi feito um *proxy* para a parte do protocolo que é executada no dispositivo de hardware, semelhante ao que foi feito na camada de segurança implementada no dispositivo.

Alguns tipos de mensagens foram definidos para realizar a comunicação entre o enclave e o dispositivo de hardware. O protocolo é simples, primeiro envia uma estrutura de tamanho fixo com o tipo de mensagem e o tamanho da mensagem, em seguida envia a mensagem. O enclave lê a estrutura e usa o tamanho enviado para ler a mensagem, o tipo de mensagem define qual será o tratador usado e a mensagem é repassada ao tratador. As mensagens usadas são descritas nas subseções seguintes.

4.2.2.1 *Call answer*

Este é um tipo usado para retornar valores, indicando que a mensagem é apenas uma resposta a uma chamada prévia. Esta é a única mensagem cujo conteúdo da mensagem não é bem definido, pois difere para cada tipo de chamada.

4.2.2.2 *Add peer*

Como o enclave não possui os identificadores, cada nodo instanciado no simulador tem seu identificador enviado ao enclave, que adiciona à sua base de dados para posterior autenticação dos nodos sensores. É uma particularidade da implementação junto ao simulador, a adição dos identificadores deverá ser feita usando uma interface com o servidor.

4.2.2.3 *Update*

A implementação do TSTP funciona usando o padrão 'observador' em que o observador se cadastra no observado e quando há alguma mudança o observado notifica o observador. O *proxy* de segurança no dispositivo de hardware é um observador e quando há uma mudança, como o recebimento de uma mensagem, ele é atualizado da mudança e envia essa mudança para dentro do enclave usando a mensagem do tipo *update*.

4.2.2.4 *Key Manager*

Esta mensagem faz com que seja executada a gerência das chaves criptográficas trocadas e as autenticações garantidas, que possuem um tempo de validade. Como o enclave não possui uma base de tempo, ele é alertado periodicamente pelo dispositivo que deve ser feita uma limpeza nas chaves que não estão sendo utilizadas ou fazer o acordo das novas chaves a serem utilizadas.

4.2.2.5 *Go*

Apenas uma mensagem que o dispositivo manda depois que suas estruturas internas foram inicializadas, permitindo que o enclave saiba quando o hardware está pronto para receber mensagens como o interesse em um *smartdata*.

4.2.2.6 *Now*

Esta mensagem é enviada do enclave para o dispositivo de hardware quando o código do enclave precisa do horário empregado na RSSF. O dispositivo responde com o horário.

4.2.2.7 *Here*

Semelhante a mensagem anterior, mas é enviada quando o enclave precisa saber a geolocalização do *gateway*.

4.2.2.8 *Alloc*

Na alocação de um *buffer* para ser enviado são feitas algumas inicializações, dependentes do meio que esta sendo transmitido o dado, então o enclave solicita para o dispositivo iniciar essas estruturas e coloca o dado cifrado antes de enviar para o nodo sensor.

4.2.2.9 *Send*

Esta, como o nome sugere, apenas envia o *buffer* para a RSSF.

4.2.2.10 *Marshall send*

Esta é semelhante à anterior, mas monta algumas partes relacionadas à localização, base de tempo e roteamento antes de enviar os dados.

4.2.3 Envio dos dados para um servidor

Quando o enclave é iniciado ele inicializa uma biblioteca TLS e abre uma conexão segura com o servidor, qualquer conexão com o ambiente externo ao enclave é mediada pelo sistema operacional, portanto o enclave solicita a abertura de uma conexão não segura com servidor e faz toda a parte de segurança dentro do enclave, restando ao sistema operacional apenas transmitir os dados cifrados.

Para garantir a segurança na comunicação com o servidor é usada a biblioteca mbedTLS. Ao abrir uma conexão com o servidor ela faz a negociação de chaves e verificação dos certificados. Para os certificados é usado o padrão de certificados de chaves públicas X.509. Todos os certificados auto-assinados confiáveis são colocados junto no executável da aplicação, esta faz o *pinning* e garante que há uma comunicação segura com o servidor,

independente dos certificados instalados na máquina que o enclave executa.

Depois de estabelecida a conexão, o enclave está pronto para enviar os dados para o servidor. Para enviar os dados foi desenvolvido um módulo observador que se acopla em um *smartdata*. A cada mensagem recebida de um nodo da RSSF, ele recebe uma notificação com a mensagem já decifrada, extrai os dados e codifica em um pacote JSON e envia para o servidor através de um canal seguro TLS. Na Figura 8 há exemplo de mensagens JSON recebidas pelo servidor.

Figura 8 – Exemplo de mensagens JSON.

Fonte: Próprio autor.

4.3 SEGURANÇA DO SISTEMA

O sistema tem um funcionamento bastante seguro, todos os dados sensíveis que entram ou saem do enclave são cifrados usando o TLS (lado servidor) ou TSTP (lado RSSF) e não podem ser retirados por meios convencionais. Dada a simplicidade das funções da fronteira do enclave torna mais fácil de ser feita uma verificação formal de que o que é executado no enclave não pode ser vazado para fora, embora esta não tenha sido feita por estar fora do escopo deste trabalho.

Para a conexão com o servidor ser realizada criou-se um certificado auto-assinado, que o *gateway* usa na sua lista de certificados confiáveis. Este certificado assinou um certificado que foi usado pelo servidor de testes. Podendo criar quantos certificados forem necessários para os servidores.

4.3.1 Problemas de segurança

Embora o sistema seja bastante seguro, há algumas limitações que podem ser críticas para o bom funcionamento do enclave e estas serão discutidas nas subseções seguintes.

4.3.1.1 Base de tempo não confiável

Como citado anteriormente o enclave não possui uma base de tempo confiável e este é um problema sério, pois o o protocolo TSTP necessita de uma base de tempo para a negociação de chaves criptográficas, gerenciamento do tempo de vida das chaves já negociadas e verificação da autenticidade das mensagens.

A base de tempo foi delegada ao dispositivo de hardware acoplado ao *gateway*, dessa forma um *gateway* mal intencionado consegue adulterar as bases de tempo que são enviadas do dispositivo de hardware para o enclave. Assim, o enclave pode não conseguir decifrar as mensagens recebidas dos nodos sensores, descartando os pacotes.

Este ataque impossibilita o funcionamento do *gateway*, mas não permite acesso aos dados cifrados pelo atacante. Detectando a inoperabilidade de um *gateway* o servidor pode alocar outro na RSSF para receber os dados solicitados. Também existe um ataque conhecido parecido na própria implementação do TSTP atacando a sincronia de tempo entre o gateway e o nodo sensor, como elucidado por **??**).

Uma maneira de contornar esse problema seria o desenvolvimento de um protocolo seguro para a comunicação entre o dispositivo de hardware e o enclave, que não fosse baseado em um *timestamp*. Fazendo a troca de mensagens como a base de tempo de forma segura entre o enclave e o dispositivo de hardware. Como o dispositivo não atuará com uma carga grande de processamento, pode-se usar o sistema com certificados X.509 para autenticar o dispositivo de hardware, colocando como nome do certificado um *hash* derivado do identificador único do dispositivo.

4.3.1.2 Ataques *side-channel*

Apesar de os dados de um enclave serem cifrados fora do chip do processador, estes ficam em texto plano na cache do processador e como diz **??**), o Intel SGX não oferece garantias contra ataques do tipo *side-channel*.

Ataques do tipo *side-channel* costumam ser bastante sofisticados, como analisar o consumo de energia do processador para obter algumas informações do que esta sendo executado, ataques de tempo como observar em quanto tempo é executado um procedimento e comparar com tempos conhecidos para obter informações.

Recentemente foi descoberto um novo ataque, o *Spectre* que usa uma falha nos processadores e consegue ler dados da cache de outros processos. Os dados de um enclave

ficam decifrados na cache portanto é possível extrair dados sensíveis de dentro de um enclave usando esse ataque. Como o SGX não protege de ataques *side-channel*, este problema está fora do escopo deste trabalho e não serão discutidas soluções.

5 CONSIDERAÇÕES FINAIS

Neste trabalho foi proposto e prototipado um *gateway* que intercomunicasse uma rede de sensores sem fio e um servidor remoto. Este *gateway* deveria operar de maneira segura mesmo em um ambiente não confiável. O protótipo foi feito usando a tecnologia Intel SGX.

5.1 RESULTADOS GERAIS

O *gateway* operou dentro de um enclave se comunicando com uma RSSF simulada usando o Castalia, realizou a conexão com um servidor usando uma autoridade certificadora criada pelo usuário e estabeleceu uma conexão segura com o servidor.

O *gateway* operou com o nodo sensor negociando chaves criptográficas com o nodo sensor, gerou interesses em dados de temperatura geolocalizados, enviou para o nodo sensor, o nodo sensor recebeu esses dados e respondeu com o valor da temperatura. O *gateway* recebeu esses dados decifrou, montou um pacote TLS e enviou para o servidor. Neste é possível ver os dados lidos do sensor.

Uma análise de possíveis vulnerabilidades do sistema foi feita, indicando possíveis soluções.

5.2 TRABALHOS FUTUROS

Este trabalho foi o desenvolvimento de uma aplicação, mas algumas partes foram levantadas e fugiram do escopo deste trabalho e ficam como sugestões de trabalhos futuros. É o caso de implementar o lado do servidor para manter a lista dos identificadores únicos dos nodos sensores e também implementar uma interface para que o *gateway* possa mandar o código de autenticação e receba o identificador único.

Também é necessário implementar uma interface no servidor para que possam colocar os interesses em dados, colocando localização tempo e tipo do dado (temperatura, pressão, umidade, etc). Além de uma interface para enviar para o *gateway* esse interesse. O *gateway* então está pronto para repassar esse interesse para o a RSSF, receber a resposta e enviar de volta ao servidor.

Detalhes de implementação, como fazer testes de performance capacidade de nodos suportados pelo *gateway* e, fazer um cadastro junto à Intel para permitir a autenticação do *gateway* no servidor usando o sistema de atestado remoto disponibilizado pela Intel.

REFERÊNCIAS

Andras. *OMNeT Discrete Event Simulator*. 2019c. <https://omnetpp.org/>. Acessado em 19 out. 2019.

ARM Limited. *SSL Library mbed TLS / PolarSSL*. 2019c. <https://tls.mbed.org/>. Acessado em 19 out. 2019.

COOPER, D. et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. maio 2008. 150 p. <https://tools.ietf.org/html/rfc5280>.

COSTAN, V.; DEVADAS, S. Intel sgx explained. *IACR Cryptology ePrint Archive*, p. 86, 2016. <http://css.csail.mit.edu/6.858/2017/readings/costan-sgx.pdf>.

Intel Corporation. *Intel® Software Guard Extensions*. jun. 2015. <https://software.intel.com/sites/default/files/332680-002.pdf>. Acessado em 19 de Out. de 2019.

POSTCAPES. *What Is The Internet of Things?* 2015. <https://www.postscapes.com/what-exactly-is-the-internet-of-things-infographic/>. Acessado em 19 de Out. de 2019.

RESCORLA, E. *The Transport Layer Security (TLS) Protocol Version 1.3*. ago. 2018. 160 p. <https://tools.ietf.org/html/rfc8446>.

RESNER, D. *Estabelecimento de Chaves e Comunicação Segura para a Internet das Coisas*. 67 p. — Federal University of Santa Catarina, Florianópolis, 2014. <http://www.lisha.ufsc.br/pub/Resner_BSC_2014.pdf>.

RESNER, D. *Performance Evaluation of the Trustful Space-Time Protocol*. 192 p. Dissertação (Mestrado) — Federal University of Santa Catarina, Florianópolis, 2018. <http://www.lisha.ufsc.br/pub/Resner_MSC_2018.pdf>.

RESNER, D.; FRöHLICH, A. A. Key Establishment and Trustful Communication for the Internet of Things. In: *4th International Conference on Sensor Networks (SENSORNETS 2015)*. Angers, France: [s.n.], 2015. p. 197–206. ISBN 978-989-758-086-4. <http://www.lisha.ufsc.br/pub/Resner_SENSORNETS_2015.pdf>.

APÊNDICE A – Artigo SBC

Gateway TSTP seguro usando Intel SGX

Juliano Kasmirski Zatta[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)

julianokzatta@gmail.com

Abstract. *The increasingly presence of devices in wireless sensor networks expanded the need of secure protocols to communicate those devices. To improve that scenario was developed the Trustful Space-Time Protocol (TSTP), a lightweight and secure protocol for communication of devices with low process capacity, this protocol does not communicate with the internet, so came the need of development a intermediate node to send the data generated from sensors to external server through internet. This text propose the development of a node executing inside safe environment using Intel Software Guard Extension. inside an enclave of memory it will decrypt the message from TSTP, encrypt using TLS and send it to a remote server, also it will manage all cryptographic keys from sensor nodes in the network. With this any computer, trustable or not, can actuate as a secure gateway to a wireless sensor network, without compromise the confidentiality or integrity of data generated by the sensors.*

Resumo. *A presença cada vez maior de dispositivos em redes de sensores sem fio fez com que ampliasse a necessidade de protocolos seguros para a comunicação. Para melhorar esse cenário foi desenvolvido o Trustful Space-Time Protocol (TSTP), um protocolo leve e seguro para comunicação de dispositivos com baixa capacidade de processamento, este protocolo não se comunica com a internet e com isso surgiu a necessidade de desenvolver um nodo intermediário para enviar esses dados para um servidor externo através da internet. Este documento apresenta o desenvolvimento de um nodo executando em um ambiente seguro usando a tecnologia Intel Software Guard Extension, dentro de um enclave de memória decifrará as mensagens TSTP, cifrará usando TLS e enviará para um servidor remoto, também gerenciará as chaves criptográficas dos nodos sensores presentes na rede. Com isso qualquer computador, confiável ou não, poderá atuar como um gateway seguro para uma rede de sensores sem fio, sem comprometer confidencialidade e integridade dos dados gerados pelos sensores.*

1. Introdução

Tem-se criado diversas aplicações para Internet das Coisas (IdC), mas não foi dado muita ênfase à segurança da informação associada a estes dispositivos. Não são raros os casos de acessos indevidos a câmeras de segurança conectadas à Internet, também a fechaduras inteligentes e brinquedos, que são explorados e têm seus comportamentos alterados. Redes de sensores sem fio também estão suscetíveis a ataques tanto para espionar dados gerados quanto para alterá-los, violando a integridade, confidencialidade dos dados enviados aos servidores.

Diversos ataques a dispositivos desenvolvidos para aplicações específicas são desenvolvidos inclusive para transformar estes dispositivos em robôs para ataques em massa, onde se destaca o ataque de negação de serviço, que pode fazer um servidor ficar inacessível temporariamente devido à sobrecarga.

Visando cobrir uma lacuna em protocolos leves e seguros desenvolvidos para IdC e Redes de Sensores Sem Fio (RSSF) foi desenvolvido o Trustful Space-Time Protocol (TSTP) que segundo [Resner 2018] é um protocolo cross-layer, eficiente e com características confiáveis de tempo, localização e segurança e é compatível com dados no sistema internacional de unidades.

Os nodos presentes nesta RSSF enviam seus dados a um gateway e este, por sua vez, envia para um servidor, que processará e armazenará esses dados. Este gateway, atualmente é um dispositivo embarcado com baixa capacidade de processamento e pouca memória. Portanto, há dificuldades para processar as chaves criptográficas com a velocidade necessária e não há memória suficiente para guardar chaves trocadas com dezenas de nodos, que podem estar presentes em uma RSSF.

Para garantir propriedades básicas de segurança, mesmo em um ambiente não confiável, será usado uma tecnologia desenvolvida pela Intel, o Intel Software Guard Extension (SGX).

2. Fundamentação

A miniaturização de dispositivos eletrônicos e uma redução considerável no consumo de energia, possibilitou o uso de sistemas inteligentes em quaisquer áreas. O baixo consumo de energia possibilita com que sensores sejam colocados nos mais diversos ambientes alimentados apenas por uma bateria durante meses ou anos, sendo esporádicas suas trocas. Outra possibilidade é alimentar esses sistemas com um conjunto composto por painéis solares e baterias aumentando a autonomia destes dispositivos para dezenas de anos.

Nos ambientes residenciais, comerciais e industriais se tem instalado cada vez mais sensores de todos os tipos: umidade, temperatura, pressão, luminosidade, consumo elétrico, dentre outros. Mas é necessário que esses dados sejam acessíveis para possibilitar simples observações e análises mais elaboradas em tempo real desses dados.

Considerando a rede mundial de computadores, a Internet, tem-se cunhado o termo Internet das Coisas (IdC), onde pequenos sistemas microcontrolados interagem com o ambiente ao seu redor, atuando e observando. Um conjunto destes sistemas pode desempenhar tarefas importantes como o controle dos semáforos de um cruzamento e controle de temperatura de uma casa, como exemplos. Para ser considerado como IdC estes dispositivos têm de se conectar à Internet.

Por serem dispositivos com baixa capacidade de processamento e de baixo consumo de energia são empregadas Redes de Sensores Sem Fio (RSSF) para enviar esses dados aos outros nodos da rede ou às centrais de processamento, com uma maior capacidade de processamento, onde esses dados serão tratados.

São desenvolvidos vários protocolos e padrões para RSSF. Normalmente, adotam-se protocolos leves para reduzir ainda mais o consumo de energia dos sensores e atuadores. Nesse aspecto se descarta o uso de TCP/IP e UDP/IP, usados na Internet, já que são protocolos mais onerosos e demandam uma maior capacidade de processamento

e, consequentemente, maior consumo de energia. Um exemplo de protocolo para essas RSSF é o TSTP.

Entre a internet e as RSSF é necessário a presença de um sistema intermediário que faça as traduções das mensagens entre o TSTP e a Internet, o gateway.

2.1. mbedTLS

O mbedTLS é uma biblioteca mantida pela ARM que implementa o protocolo TLS, como citado no site da biblioteca é uma biblioteca fácil de entender, usar, integrar e expandir. E por isso foi escolhida para o projeto. Por ela funcionar em sistemas embarcados, com ou sem sistema operacional torna-se ideal para integrar dentro do enclave. Para integração basta redefinir algumas funções, como a criação de um socket, que é feita usando funções de borda e solicitando para o sistema operacional não confiável. Outra necessidade é a implementação da geração de uma semente para os geradores de números aleatórios, funcionalidade que é disponibilizada dentro do enclave. Foi encontrada um repositório público já com as mudanças feitas, portanto não há necessidade de reimplementar, bastando reutilizar.

2.2. Intel Software Guard Extension

O SGX é uma extensão do conjunto de instruções dos processadores da Intel e permite que uma aplicação seja executada em nível de usuário de maneira segura, mesmo em uma plataforma não confiável, como sistema operacional, máquina virtual ou BIOS infectados, um computador remoto não confiável e etc. Ele faz isso isolando os dados do ambiente externo [Costan and Devadas 2016].

Este conjunto de instruções provê ao desenvolvedor um enclave, uma região de memória que está contida na memória da plataforma, mas inacessível por outros processos executando na mesma plataforma. Esse isolamento é feito com um hardware específico presente no chip da CPU.

3. O Protótipo

O gateway foi pensado e desenvolvido para operar dentro de um enclave de memória e evitar que dados sensíveis sejam expostos. Como característica do enclave não há possibilidade de acesso ao hardware, aos mediadores de hardware e nem a funcionalidades do sistema operacional em que o enclave esteja executando. Assim, os dados ficam protegidos mesmo que algum componente do computador esteja comprometido (hardware, sistema operacional, virtualizador e outros processos são alguns componentes).

Desde o princípio o gateway foi pensado para possuir apenas as funcionalidades estritamente necessárias ao seu funcionamento, diminuindo os pontos de possíveis ataques. Para isso, algumas partes do protocolo TSTP foram mantidas fora do enclave, essas partes navegam em texto plano e mantê-las fora não prejudica a segurança do gateway, pelo contrário, diminui a complexidade do enclave.

O projeto foi feito usando como base o mesmo dispositivo de hardware de um nodo sensor para fazer funcionar como uma ponte entre a camada física da RSSF para a o enclave. Como o TSTP já foi implementado para executar nesse dispositivo tarefas como geolocalização, temporização e roteamento já estão implementadas. Essas tarefas estão

bastante associadas à rede sem fio e não demandam grande quantidade de recursos, foram mantidas nesses dispositivos ao invés de serem levadas ao gateway. Apenas a informação sensível é repassada ao gateway para que este desempenhe sua tarefa.

Foi implementado para ser um processo em um sistema operacional Linux e se comunicar com o dispositivo de hardware através de uma interface serial disponibilizada pelo sistema e facilmente portável para outras interfaces. O processo do gateway envia os dados dessa comunicação o enclave e passa a execução para dentro do enclave. Devido ao enclave não possuir acesso a dispositivos de hardware do computador e nenhum controle sobre sua execução alguns recursos não estão disponíveis dentro do enclave, como é o caso dos relógios de tempo, a execução dentro do enclave não possui nenhuma base de tempo confiável, sendo necessário enviar essas informações para dentro dele.

Para execução dos testes o dispositivo de hardware foi simulado usando o OMNet++ com Castalia. Os dados a serem enviados ao enclave são enviados e recebidos usando uma interface serial simulada no Linux. Este simulador é desenvolvido para de RSSF, como é o caso do TSTP, o protocolo usado neste trabalho, e tem seu funcionamento igual ao de uma rede sem fio física, mas com parâmetros bem definidos e sem interferências de fatores externos.

3.1. Comunicação na rede sem fio

Na inicialização o dispositivo simulado define o tempo da rede, baseado em uma informação recebida do gateway. A partir desse momento ele mantém o controle da base de tempo da rede. Todos os nodos sensores que se conectarem à rede usarão essa base de tempo para aferir os tempos das medições, estabelecer a comunicação com o gateway e estabelecer as chaves criptográficas usadas na comunicação.

Os dados são recebidos pelo dispositivo através da rede sem fio simulada. Ele abre o pacote de dados e analisa as informações não cifradas e envia para o gateway através da interface serial, se este for o destinatário do pacote recebido. Se os dados não tiverem o gateway como destinatário o dispositivo se encarrega de descartar o pacote ou reenviar para a rede, funcionando apenas como um roteador.

3.1.1. Envio dos dados para o gateway

Por se tratar de um protocolo cross-layer, o pacote é movido entre suas camadas, mas a camada de segurança não é processada no dispositivo de hardware e sim enviada ao enclave. Para o envio dos dados para o enclave a classe que implementa a segurança foi substituída por um proxy que envia os dados para o enclave e recebe uma resposta.

A implementação dentro do enclave recebe os dados e retorna a resposta de status para o dispositivo de hardware dar continuidade aos tratamentos que não o envolvem, como destruir as estruturas alocadas na memoria e etc. De dentro do enclave também partem solicitações, de dados do dispositivo de hardware, como solicitação do horário da rede, geolocalização do gateway e envio de uma mensagem para a rede, além de chamadas para controle interno do TSTP.

3.2. Enclave

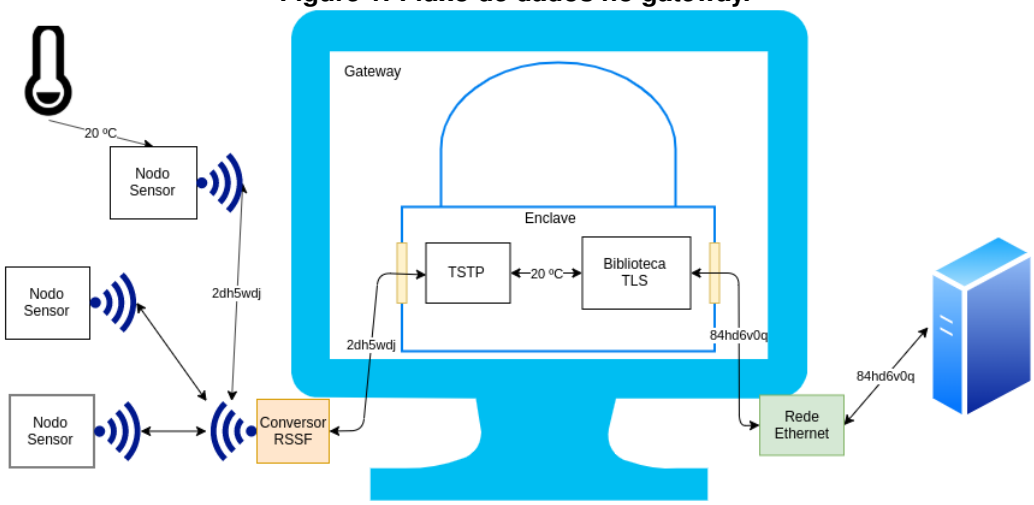
Na primeira conexão o enclave realiza a negociação de chaves Diffie-Hellman estabelecendo a Master Secret e o enclave guarda este segredo em uma lista de nodos não autenticados. Após isso é feita a autenticação do nodo sensor enviando o seu código de autenticação, que é um hash derivado do seu identificador único e a OTP, derivada do seu identificador, base de tempo da rede e do Master Secret. O enclave confere se o código de autenticação está na sua base de dados e associa a Master Secret ao respectivo nodo sensor e em seguida envia para o nodo sensor uma mensagem de autenticação deferida.

Como apenas o gateway e o nodo sensor conhecem o identificador único do sensor, isso quer dizer que foi estabelecida uma conexão ponto-a-ponto entre o gateway e o sensor. Os identificadores não podem ficar fixos no código do enclave, pois este não é cifrado, é apenas assinado, e pode ser visualizado por qualquer pessoa ou computador que tenha acesso executável do enclave.

Após estabelecida a segurança na comunicação, o gateway demonstra interesse em um dado dizendo o tipo de dado desejado (temperatura, por exemplo), a área em que o dado desejado pode ser amostrado e o tempo que essa informação é relevante.

O nodo sensor que tenha o dado desejado, esteja dentro da área de interesse e esteja autenticado com o enclave responde com os dados solicitados cifrando e assinando esses dados. O enclave os recebe, decifra esses dados do TSTP, monta um pacote com a interface definida pelo servidor, estabelece uma conexão segura usando TLS e envia para ele. Esse fluxo de dados pode ser visto na Figura 1, usando uma amostra de temperatura como exemplo.

Figure 1. Fluxo de dados no gateway.



Fonte: Próprio autor.

3.2.1. Envio dos dados para um servidor

Quando o enclave é iniciado ele inicializa uma biblioteca TLS e abre uma conexão segura com o servidor, qualquer conexão com o ambiente externo ao enclave é mediada

pelo sistema operacional, portanto o enclave solicita a abertura de uma conexão não segura com servidor e faz toda a parte de segurança dentro do enclave, restando ao sistema operacional apenas transmitir os dados cifrados.

Para garantir a segurança na comunicação com o servidor é usada a biblioteca mbedTLS. Ao abrir uma conexão com o servidor ela faz a negociação de chaves e verificação dos certificados. Para os certificados é usado o padrão de certificados de chaves públicas X.509. Todos os certificados auto-assinados confiáveis são colocados junto no executável da aplicação, esta faz o pinning e garante que há uma comunicação segura com o servidor, independente dos certificados instalados na máquina que o enclave executa.

Depois de estabelecida a conexão, o enclave está pronto para enviar os dados para o servidor. A cada mensagem recebida de um nodo da RSSF, ele decifra a mensagem, extrai os dados e codifica em um pacote JSON e envia para o servidor através de um canal seguro TLS.

4. Segurança do sistema

O sistema tem um funcionamento bastante seguro, todos os dados sensíveis que entram ou saem do enclave são cifrados usando o TLS (lado servidor) ou TSTP (lado RSSF) e não podem ser retirados por meios convencionais. Dada a simplicidade das funções da fronteira do enclave torna mais fácil de ser feita uma verificação formal de que o que é executado no enclave não pode ser vazado para fora, embora esta não tenha sido feita por estar fora do escopo deste trabalho.

Para a conexão com o servidor ser realizada criou-se um certificado auto-assinado, que o gateway usa na sua lista de certificados confiáveis. Este certificado assinou um certificado que foi usado pelo servidor de testes. Podendo criar quantos certificados forem necessários para os servidores.

Embora o sistema seja bastante seguro, há algumas limitações que podem ser críticas para o bom funcionamento do enclave e estas serão discutidas nas subseções seguintes.

4.1. Base de tempo não confiável

Como citado anteriormente o enclave não possui uma base de tempo confiável e este é um problema sério, pois o o protocolo TSTP necessita de uma base de tempo para a negociação de chaves criptográficas, gerenciamento do tempo de vida das chaves já negociadas e verificação da autenticidade das mensagens.

A base de tempo foi delegada ao dispositivo de hardware acoplado ao gateway, dessa forma um gateway mal intencionado consegue adulterar as bases de tempo que são enviadas do dispositivo de hardware para o enclave. Assim, o enclave pode não conseguir decifrar as mensagens recebidas dos nodos sensores, descartando os pacotes.

Este ataque impossibilita o funcionamento do gateway, mas não permite acesso aos dados cifrados pelo atacante. Detectando a inoperabilidade de um gateway o servidor pode alocar outro na RSSF para receber os dados solicitados. Também existe um ataque conhecido parecido na própria implementação do TSTP atacando a sincronia de tempo entre o gateway e o nodo sensor [Resner and Fröhlich 2015].

Uma maneira de contornar esse problema seria o desenvolvimento de um protocolo seguro para a comunicação entre o dispositivo de hardware e o enclave, que não fosse baseado em um timestamp. Fazendo a troca de mensagens como a base de tempo de forma segura entre o enclave e o dispositivo de hardware. Como o dispositivo não atuará com uma carga grande de processamento, pode-se usar o sistema com certificados X.509 para autenticar o dispositivo de hardware, colocando como nome do certificado um hash derivado do identificador único do dispositivo.

4.2. Ataques side-channel

Apesar de os dados de um enclave serem cifrados fora do chip do processador, estes ficam em texto plano na cache do processador e como diz [Costan and Devadas 2016], o Intel SGX não oferece garantias contra ataques do tipo side-channel.

Ataques do tipo side-channel costumam ser bastante sofisticados, como analisar o consumo de energia do processador para obter algumas informações do que esta sendo executado, ataques de tempo como observar em quanto tempo é executado um procedimento e comparar com tempos conhecidos para obter informações.

Recentemente foi descoberto um novo ataque, o Spectre que usa uma falha nos processadores e consegue ler dados da cache de outros processos. Os dados de um enclave ficam decifrados na cache portanto é possível extrair dados sensíveis de dentro de um enclave usando esse ataque. Como o SGX não protege de ataques side-channel, este problema está fora do escopo deste trabalho e não serão discutidas soluções.

5. Trabalhos Futuros

Este trabalho foi o desenvolvimento de uma aplicação, mas algumas partes foram levantadas e fugiram do escopo deste trabalho e ficam como sugestões de trabalhos futuros. É o caso de implementar o lado do servidor para manter a lista dos identificadores únicos dos nodos sensores e também implementar uma interface para que o gateway possa mandar o código de autenticação e receba o identificador único.

Também é necessário implementar uma interface no servidor para que possam colocar os interesses em dados, colocando localização tempo e tipo do dado (temperatura, pressão, umidade, etc). Além de uma interface para enviar para o gateway esse interesse. O gateway então está pronto para repassar esse interesse para o a RSSF, receber a resposta e enviar de volta ao servidor.

References

ARM Limited (2019c). Ssl library mbed tls / polarssl.

Costan, V. and Devadas, S. (2016). Intel sgx explained. *IACR Cryptology ePrint Archive*, page 86.

Resner, D. (2018). Performance Evaluation of the Trustful Space-Time Protocol. Master's thesis, Federal University of Santa Catarina, Florianópolis.

Resner, D. and Fröhlich, A. A. (2015). Key Establishment and Trustful Communication for the Internet of Things. In *4th International Conference on Sensor Networks (SENSORNETS 2015)*, pages 197–206, Angers, France.

APÊNDICE B – Codigo Fonte

Este anexo contém o código-fonte desenvolvido para este trabalho. A implementação foi feita no contexto da simulação desenvolvida para a dissertação de mestrado **??**) e do projeto **??**), usando as linguagens C++ e EDL.

Os códigos-fonte que foram aproveitados de trabalhos anteriores e não sofreram modificações não serão apresentados neste anexo. O arquivo "ca_bundle.h"será em partes em partes suprimido por apresentar apenas uma lista de certificados confiáveis usados na implementação. Demais arquivos poderão ter trechos de código usados em testes removidos por não afetarem o funcionamento do *gateway*, tornando mais legível o código presente neste documento.

Listing B.1 – App/s_client.cpp

```
1 #include <iostream>
2 #include <stdio.h>
3 #include <sgx_urts.h>
4 #include <sgx_error.h>
5
6 #include "Enclave_u.h"
7 #include "Utils.h"
8 #include "Serial.h"
9
10 using namespace std;
11
12 sgx_enclave_id_t eid = 0;
13
14 int main() {
15     int ret;
16     int i;
17     sgx_status_t err;
18     char c[100];
19     ret = initialize_enclave(&eid);
20     if (ret != 0) {
21         cerr << "failed to initialize the enclave" << endl;
22         exit(-1);
23     }
24     printf("Enclave %lu created\n", eid);
25     err = initializeTSTP(eid, &ret);
26     err = runTSTP(eid);
27     while (1)
28         ;
29     exit:
30     sgx_destroy_enclave(eid);
31     cout << "Info: all enclave closed successfully." << endl;
32     return 0;
33 }
```

Listing B.2 – App/Serial.h

```
1
2 #pragma once
3
4 #include <string.h>
5 #include <stdlib.h>
6 #include <stdio.h>
```

```
 7 #include <unistd.h>
 8 #include <fcntl.h>
 9 #include <termios.h>
10 #include <errno.h>
11 #include <pthread.h>
12
13 class Serial {
14 private:
15   const char *device;
16   struct termios old_config;
17   struct termios config;
18   speed_t old_ispeed;
19   speed_t old_ospeed;
20   speed_t ispeed;
21   speed_t ospeed;
22   int tty_fd;
23   int tty_errno;
24 public:
25   Serial(const char *_device);
26   ~Serial(void);
27   ssize_t read(char *buff, size_t length);
28   ssize_t write(const char *buff, size_t length);
29   int getErrno();
30 };
```

Listing B.3 – App/Serial.cpp

```
1
2 #include "Serial.h"
3 #include "Enclave_u.h"
4
5 #include <unistd.h>
6
7 Serial *dev;
8 extern sgx_enclave_id_t eid;
9 void *runThreadEnclave(void *arg);
10 pthread_t thread;
11
12 int Serial_init(const char *device) {
13     int  iret1;
14     dev = new Serial(device);
15     /* Create independent threads each of which will execute function */
16     iret1 = pthread_create(&thread, NULL, runThreadEnclave, NULL);
17     if(iret1) {
18         fprintf(stderr,"Error - pthread_create() return code: %d\n",
    iret1);
19     }
20     return 0;
21 }
22
23 void Serial_deinit() {
24     pthread_join(thread, NULL);
25     delete dev;
26 }
27
28 int Serial_read(void *data, unsigned int size) {
29     return dev->read((char *)data, size);
30 }
31
```

```
32 int Serial_write(const void *data, unsigned int size) {
33     return dev->write((const char *)data, size);
34 }
35
36 void *runThreadEnclave(void *arg) {
37     runThread(eid);
38     return NULL;
39 }
40
41 Serial::Serial(const char *_device) {
42   device = _device;
43   while ((tty_fd = open(device, O_RDWR/* | O_NONBLOCK*/)) < 0)
44       ;
45   tcgetattr(tty_fd,&old_config);
46   memset(&config,0,sizeof(struct termios));
47   config.c_iflag=0;
48   config.c_oflag=0;
49   config.c_cflag=CS8|CREAD|CLOCAL;  // 8n1, see termios.h for more
       information
50   config.c_lflag=0;
51   config.c_cc[VMIN]=1;
52   config.c_cc[VTIME]=5;
53   cfsetospeed(&config,B115200);             // 115200 baud
54   cfsetispeed(&config,B115200);             // 115200 baud
55   tcsetattr(tty_fd,TCSANOW,&config);
56 }
57
58 Serial::~Serial(void) {
59     tcsetattr(tty_fd,TCSANOW,&old_config);
60     close(tty_fd);
61 }
62
63 ssize_t Serial::read(char *buff, size_t length) {
64   tty_errno = 0;
65   ssize_t ret = ::read(tty_fd, buff, length);
66   if (ret < 0) {
67     tty_errno = errno;
68   }
69   return ret;
70 }
71
72 ssize_t Serial::write(const char *buff, size_t length) {
73   tty_errno = 0;
74   ssize_t ret = ::write(tty_fd, buff, length);
75   if (ret < 0) {
76     tty_errno = errno;
77   }
78   return ret;
79 }
80
81 int Serial::getErrno() {
82   return tty_errno;
83 }
84
85 void tsleep(unsigned int size) {
86     usleep(size);
87 }
```

Listing B.4 – Enclave/Enclave.edl

```
enclave {
    from "mbedtls_sgx.edl" import *;
    from "sgx_tstdc.edl" import *;
    from "TSTP/Serial.edl" import *;
    include "../common/ssl_context.h"

    trusted {
        /* define ECALLs here. */
        public int initializeTSTP(void);
        public void runTSTP(void);
    };

    untrusted {
        /* define OCALLs here. */
    };
};
```

Listing B.5 – Enclave/ecalls.cpp

```

#include "Enclave_t.h"
#include "Log.h"

#include <string.h>

#include "SSLConnection.h"

#include <enclave_init.h>

SSLConnection *conn;

int initializeTSTP(void) {
    printf_sgx("\nINIT: \n");
    SSLConnection *conn = new SSLConnection("127.0.0.1", "8443");
    _SYS::init(conn);
    return 0;
}

void runTSTP(void) {
    printf_sgx("\nRUN: \n");
    _SYS::run();
}
```

Listing B.6 – Enclave/ca_bundle.h

```
// THIS FILE IS GENERATED BY pem2def.py. DONT EDIT.

#ifndef TRUSTED_CERTS_H
#define TRUSTED_CERTS_H

#define ca_bundle \
"LISHA TEST Root\r\n"\
"============\r\n"\
"-----BEGIN CERTIFICATE-----\r\n"\
( .. )
"-----END CERTIFICATE-----\r\n"\
"\r\n\r\n"\
```

```
13
14
15 #endif
```

Listing B.7 – Enclave/SSLConnection.h

```
1
2 #pragma once
3
4 #include "SSLX509.h"
5 #include "mbedtls/ssl.h"
6
7 #include "mbedtls/net_v.h"
8 #include "mbedtls/net_f.h"
9 #include "mbedtls/entropy.h"
10 #include "mbedtls/ctr_drbg.h"
11 #include "mbedtls/certs.h"
12 #include "mbedtls/x509.h"
13 #include "mbedtls/error.h"
14 #include "mbedtls/debug.h"
15
16 #include "Enclave_t.h"
17 #include "Log.h"
18 #include "pprint.h"
19
20 #define SERVER    127.0.0.1
21 #define PORT      4433
22
23 class SSLConnection {
24 private:
25     const char *server;
26     const char *port;
27
28     mbedtls_net_context server_fd;
29     mbedtls_entropy_context entropy;
30     mbedtls_ctr_drbg_context ctr_drbg;
31     mbedtls_ssl_context ssl;
32     mbedtls_ssl_config conf;
33
34     SSLX509 *x509;
35
36     bool connected;
37
38     // Marked as unused
39     mbedtls_ssl_session saved_session;
40     unsigned char psk[MBEDTLS_PSK_MAX_LEN];
41     size_t psk_len;
42
43     void resetSession();
44
45 public:
46     SSLConnection(const char* server, const char *port);
47     ~SSLConnection();
48
49     bool connect();
50     bool setup();
51     bool handshake();
52     bool verifyCertificates();
53
```

```
54      int write(const void *p_data, size_t p_len);
55      int read(void *p_data, size_t p_len);
56
57      bool reconnect();
58      bool close();
59
60      void saveSession(void);
61
62  };
```

Listing B.8 – Enclave/SSLConnection.cpp

```
1
2   #include "SSLConnection.h"
3   #include "glue.h"
4
5   #include <string.h>
6
7   #define mbedtls_printf printf_sgx
8   #ifdef DEBUG
9   #define THRESHOLD_DEBUG 0
10  static int my_verify( void *data, mbedtls_x509_crt *crt, int depth,
        uint32_t *flags )
11  {
12      char buf[1024];
13      ((void) data);
14      mbedtls_printf( "\nVerify requested for (Depth %d):\n", depth );
15      mbedtls_x509_crt_info( buf, sizeof( buf ) - 1, "", crt );
16      mbedtls_printf( "%s", buf );
17      if ( ( *flags ) == 0 )
18          mbedtls_printf( "  This certificate has no flags\n" );
19      else
20      {
21          mbedtls_x509_crt_verify_info( buf, sizeof( buf ), "  ! ", *flags
        );
22          mbedtls_printf( "%s\n", buf );
23      }
24      return( 0 );
25  }
26  static void my_debug( void *ctx, int level,
27                        const char *file, int line,
28                        const char *str )
29  {
30      const char *p, *basename;
31      (void)(ctx);
32      /* Extract basename from file */
33      for ( p = basename = file; *p != '\0'; p++ )
34          if ( *p == '/' || *p == '\\' )
35              basename = p + 1;
36          mbedtls_printf("%s:%04d: |%d| %s", basename, line, level, str );
37  }
38  #else
39  #define THRESHOLD_DEBUG 0
40  #endif
41
42  SSLConnection::SSLConnection(const char* _server, const char *_port) {
43      int ret;
44      server = _server;
45      port = _port;
```

```
46      psk_len = 0;
47      connected = false;
48      // Load certificates
49
50      mbedtls_net_init(&server_fd);
51      mbedtls_ssl_init(&ssl);
52      mbedtls_ssl_config_init(&conf);
53      memset(&saved_session, 0, sizeof( mbedtls_ssl_session ));
54      mbedtls_ctr_drbg_init(&ctr_drbg);
55
56      // DEBUG LEVEL TLS
57      mbedtls_debug_set_threshold( THRESHOLD_DEBUG );
58      // DEBUG LEVEL TLS
59      /*
60       * 0. Initialize the RNG and the session data
61       */
62      LL_LOG("Seeding the random number generator..." );
63      mbedtls_entropy_init(&entropy);
64      ret = mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &
        entropy,NULL,0);
65      if (ret != 0) {
66          LL_CRITICAL(" mbedtls_ctr_drbg_seed returned -%#x", -ret);
67          // ???
68      }
69      /*
70       * 1. Load the trusted CA
71       */
72      x509 = SSLX509::getX509();
73  }
74
75  SSLConnection::~SSLConnection() {
76      /*
77       * Cleanup and exit
78       */
79      mbedtls_net_free(&server_fd);
80      mbedtls_ssl_session_free(&saved_session);
81      mbedtls_ssl_free(&ssl);
82      mbedtls_ssl_config_free(&conf);
83      mbedtls_ctr_drbg_free(&ctr_drbg);
84      mbedtls_entropy_free(&entropy);
85      SSLX509::destroy();
86  }
87
88  bool SSLConnection::connect(void) {
89      int ret;
90      if (connected == true) {
91          return true;
92      }
93      /*
94       * 2. Start the connection
95       */
96
97      LL_LOG("connecting to %s:%s:%s...", "TCP", server, port);
98      ret = mbedtls_net_connect(&server_fd, server, port,
        MBEDTLS_NET_PROTO_TCP);
99      if (ret != 0) {
100         LL_CRITICAL(" mbedtls_net_connect returned -%#x", -ret);
101         return false;
```

```
102     }
103
104     ret = mbedtls_net_set_block(&server_fd);
105     if (ret != 0 ) {
106         LL_CRITICAL(" net_set_(non)block() returned -%#x", -ret);
107         return false;
108     }
109
110     if (!setup()) {
111         return false;
112     }
113     if (!handshake()) {
114         return false;
115     }
116     if (!verifyCertificates()) {
117         return false;
118     }
119     connected = true;
120     return true;
121 }
122
123 bool SSLConnection::setup(void) {
124     int ret;
125     /*
126      * 3. Setup stuff
127      */
128     LL_LOG("Setting up the SSL/TLS structure...");
129
130     ret = mbedtls_ssl_config_defaults(&conf, MBEDTLS_SSL_IS_CLIENT,
131                                       MBEDTLS_SSL_TRANSPORT_STREAM,
    MBEDTLS_SSL_PRESET_DEFAULT);
132     if (ret != 0) {
133         LL_CRITICAL("mbedtls_ssl_config_defaults returned -%#x", -ret);
134         return false;
135     }
136
137     ret  = mbedtls_ssl_conf_max_frag_len(&conf,
    MBEDTLS_SSL_MAX_FRAG_LEN_NONE);
138     if (ret != 0) {
139         mbedtls_printf("  mbedtls_ssl_conf_max_frag_len returned %d\n\n"
    , ret);
140         return false;
141     }
142
143     mbedtls_ssl_conf_rng(&conf, mbedtls_ctr_drbg_random, &ctr_drbg);
144
145     mbedtls_ssl_conf_read_timeout(&conf, 0);
146     mbedtls_ssl_conf_session_tickets(&conf,
    MBEDTLS_SSL_SESSION_TICKETS_ENABLED);
147     mbedtls_ssl_conf_renegotiation(&conf,
    MBEDTLS_SSL_RENEGOTIATION_DISABLED);
148
149     // Implicit:
150     // mbedtls_ssl_conf_ca_chain(conf, &certsChain, &revokedChain);
151     // mbedtls_ssl_conf_own_cert(conf, &clientCerts, &pkey);
152     ret = x509->ssl_conf(&conf);
153     if (ret != 0) {
154         mbedtls_printf("  mbedtls_ssl_conf_own_cert returned %d\n\n",
```

```
    ret);
155         return false;
156     }
157 
158     const char *psk_identity = "Client_identity";
159     ret = mbedtls_ssl_conf_psk(&conf, psk, psk_len,
160                                (const unsigned char *) psk_identity,
    strlen(psk_identity));
161     if (ret != 0) {
162         mbedtls_printf("  mbedtls_ssl_conf_psk returned %d\n\n", ret);
163         return false;
164     }
165 
166     ret = mbedtls_ssl_setup(&ssl, &conf);
167     if (ret != 0) {
168         LL_CRITICAL("mbedtls_ssl_setup returned -%#x", -ret);
169         return false;
170     }
171 
172     ret = mbedtls_ssl_set_hostname(&ssl, server);
173     if (ret != 0) {
174         LL_CRITICAL("mbedtls_ssl_set_hostname returned %d\n\n", ret);
175         return false;
176     }
177 
178     mbedtls_ssl_set_bio(&ssl, &server_fd, mbedtls_net_send,
    mbedtls_net_recv,
179                         mbedtls_net_recv_timeout);
180     return true;
181 }
182 
183 bool SSLConnection::handshake() {
184     int ret;
185     /*
186      * 4. Handshake
187      */
188     LL_LOG( "Performing the SSL/TLS handshake" );
189     ret = mbedtls_ssl_handshake(&ssl);
190     while (ret != 0) {
191         if (ret != MBEDTLS_ERR_SSL_WANT_READ && ret !=
    MBEDTLS_ERR_SSL_WANT_WRITE) {
192             LL_CRITICAL( "mbedtls_ssl_handshake returned -%#x", -ret );
193             if ( ret == MBEDTLS_ERR_X509_CERT_VERIFY_FAILED )
194                 LL_CRITICAL(
195                     "Unable to verify the server's certificate. "
196                     "Either it is invalid,"
197                     "or you didn't set ca_file or ca_path "
198                     "to an appropriate value."
199                     "Alternatively, you may want to use "
200                     "auth_mode=optional for testing purposes." );
201                 return false;
202         }
203         ret = mbedtls_ssl_handshake(&ssl);
204     }
205     LL_LOG( "Hand shake succeeds: [%s, %s]",
206             mbedtls_ssl_get_version( &ssl ), mbedtls_ssl_get_ciphersuite
    (&ssl));
207 
```

```
208     ret = mbedtls_ssl_get_record_expansion(&ssl);
209     if (ret >= 0)
210         LL_DEBUG("Record expansion is [%d]", ret);
211     else
212         LL_DEBUG("Record expansion is [unknown (compression)]");
213 
214     LL_LOG("  . Saving session for reuse..." );
215 
216     ret = mbedtls_ssl_get_session(&ssl, &saved_session);
217     if (ret != 0) {
218         LL_CRITICAL("mbedtls_ssl_get_session returned -%#x", -ret );
219         return false;
220     }
221 
222     LL_LOG("Maximum fragment length is [%u]",
223            (unsigned int) mbedtls_ssl_get_max_frag_len(&ssl));
224     return true;
225 }
226 
227 bool SSLConnection::verifyCertificates() {
228     int flags;
229     /*
230      * 5. Verify the server certificate
231      */
232     LL_LOG( "Verifying peer X.509 certificate..." );
233     flags = mbedtls_ssl_get_verify_result(&ssl);
234     if (flags != 0) {
235         char vrfy_buf[512];
236         mbedtls_printf(" failed\n");
237         mbedtls_x509_crt_verify_info(vrfy_buf, sizeof( vrfy_buf ), "  !
    ", flags);
238         mbedtls_printf("%s\n", vrfy_buf);
239     }
240     else
241         LL_LOG("X.509 Verifies");
242     return true;
243 }
244 
245 int SSLConnection::write(const void *pv_data, size_t p_len) {
246     int ret, written, frags;
247     const uint8_t *p_data = (const uint8_t *)pv_data;
248     if (!connected) {
249         return -1;
250     }
251     /*
252      * 6. Write to the server
253      */
254     for (written = 0, frags = 0; written < p_len; written += ret, frags
    ++) {
255         do {
256             ret = mbedtls_ssl_write(&ssl, p_data + written, p_len -
    written);
257             if (ret < 0) {
258                 if (ret != MBEDTLS_ERR_SSL_WANT_READ &&
259                     ret != MBEDTLS_ERR_SSL_WANT_WRITE) {
260                     mbedtls_printf("  mbedtls_ssl_write returned -%#x\n"
    , -ret);
261                 connected = false;
```

```
262                 return ret;
263                     }
264             }
265         } while (ret <= 0);
266     }
267    return written;
268 }
269
270 int SSLConnection::read(void *pv_data, size_t p_len) {
271     int ret, readed;
272     uint8_t *p_data = (uint8_t *)pv_data;
273     if (!connected) {
274         return -1;
275     }
276     /*
277      * 7. Read the HTTP response
278      */
279     readed = 0;
280     memset(p_data, 0, p_len);
281     ret = mbedtls_ssl_read(&ssl, p_data, p_len);
282     while (ret == MBEDTLS_ERR_SSL_WANT_READ || ret ==
    MBEDTLS_ERR_SSL_WANT_WRITE){
283         ret = mbedtls_ssl_read(&ssl, p_data, p_len);
284     }
285     if (ret <= 0) {
286         switch(ret) {
287             case MBEDTLS_ERR_SSL_PEER_CLOSE_NOTIFY:
288                 mbedtls_printf(" connection was closed gracefully\n" );
289                 break;
290             case 0:
291             case MBEDTLS_ERR_NET_CONN_RESET:
292                 mbedtls_printf(" connection was reset by peer\n" );
293                 ret = -1;
294                 break;
295             default:
296                 mbedtls_printf(" mbedtls_ssl_read returned -0x%x\n", -
    ret );
297                 break;
298         }
299         connected = false;
300         return ret;
301     }
302     readed = ret;
303     return readed;
304 }
305
306 bool SSLConnection::reconnect() {
307     int ret;
308     /*
309      * 9. Reconnect?
310      */
311     if (connected == true) {
312         return true;
313     }
314     mbedtls_net_free( &server_fd );
315     mbedtls_printf( "  . Reconnecting with saved session..." );
316
317     ret = mbedtls_ssl_session_reset(&ssl);
```

```
318     if (ret != 0) {
319         mbedtls_printf( "  mbedtls_ssl_session_reset returned -%#x", -
    ret );
320         return false;
321     }
322 
323     ret = mbedtls_ssl_set_session(&ssl, &saved_session);
324     if (ret != 0) {
325         mbedtls_printf( "  mbedtls_ssl_conf_session returned %d\n\n",
    ret );
326         return false;
327     }
328 
329     ret = mbedtls_net_connect(&server_fd, server, port,
    MBEDTLS_NET_PROTO_TCP);
330     if (ret != 0) {
331         mbedtls_printf( "  mbedtls_net_connect returned -%#x", -ret );
332         return false;
333     }
334 
335     ret = mbedtls_net_set_block( &server_fd );
336     if (ret != 0) {
337         mbedtls_printf( "  net_set_(non)block() returned -%#x",
338                         -ret );
339         return false;
340     }
341 
342     ret = mbedtls_ssl_handshake(&ssl);
343     while (ret != 0) {
344         if (ret != MBEDTLS_ERR_SSL_WANT_READ && ret !=
    MBEDTLS_ERR_SSL_WANT_WRITE) {
345             mbedtls_printf( "  mbedtls_ssl_handshake returned -%#x", -
    ret );
346             return false;
347         }
348         ret = mbedtls_ssl_handshake(&ssl);
349     }
350 
351     connected = true;
352     return true;
353 }
354 
355 bool SSLConnection::close() {
356     int ret;
357     if (connected == false) {
358         return true;
359     }
360     connected = false;
361     /*
362      * 8. Done, cleanly close the connection
363      */
364     do {
365         ret = mbedtls_ssl_close_notify(&ssl);
366     } while (ret == MBEDTLS_ERR_SSL_WANT_WRITE);
367     LL_LOG( "closed %s:%s", server, port);
368     return true;
369 }
```

Listing B.9 – Enclave/SSLX509.h

```
1
2 #pragma once
3
4 #include "ca_bundle.h"
5 #include "mbedtls/ssl.h"
6 #include "mbedtls/x509.h"
7
8 // This lib is not thread safe
9
10 class SSLX509 {
11 private:
12     static SSLX509 *singleton;
13     static int activeUsages;
14
15     mbedtls_x509_crt caCerts;
16     mbedtls_x509_crt clientCerts;
17     mbedtls_x509_crl revokedChain;
18     mbedtls_pk_context pkey;
19
20     void init(void);
21     void deinit(void);
22
23     SSLX509();
24     ~SSLX509();
25
26 public:
27     static SSLX509 *getX509(void);
28     static void destroy(void);
29     void ssl_conf_ca_chain(mbedtls_ssl_config *conf);
30     int ssl_conf_own_cert(mbedtls_ssl_config *conf);
31     int ssl_conf(mbedtls_ssl_config *conf);
32 };
```

Listing B.10 – Enclave/SSLX509.cpp

```
1
2 #include "SSLX509.h"
3
4 #include "ca_bundle.h"
5 #include "Log.h"
6
7 #include <stdlib.h>
8 #include <new>
9
10
11 SSLX509 *SSLX509::singleton = NULL;
12 int SSLX509::activeUsages = 0;
13
14 SSLX509::SSLX509(void) {
15     init();
16     activeUsages = 0;
17 }
18
19 SSLX509::~SSLX509(void) {
20     deinit();
21 }
22
```

```
23 SSLX509 *SSLX509::getX509(void) {
24     if (singleton == NULL) {
25         singleton = new SSLX509();
26     }
27     singleton->activeUsages++;
28     return singleton;
29 }
30
31 void SSLX509::destroy(void) {
32     activeUsages--;
33     if (activeUsages > 0) {
34         return;
35     }
36     delete singleton;
37     singleton = NULL;
38 }
39
40 void SSLX509::init(void) {
41     int ret;
42
43     mbedtls_x509_crt_init(&caCerts);
44     mbedtls_x509_crt_init(&clientCerts);
45     mbedtls_x509_crl_init(&revokedChain);
46     mbedtls_pk_init(&pkey);
47     LL_LOG( "Loading the CA root certificate");
48     ret = mbedtls_x509_crt_parse(&caCerts,
49                                  (const unsigned char *) ca_bundle,
50                                  sizeof(ca_bundle));
51     if (ret != 0) {
52         LL_CRITICAL("  mbedtls_x509_crt_parse returned -%#x", -ret);
53         return;
54     }
55 }
56
57 void SSLX509::deinit(void) {
58     mbedtls_x509_crt_free(&caCerts);
59     mbedtls_x509_crt_free(&clientCerts);
60     mbedtls_pk_free(&pkey);
61     mbedtls_x509_crl_free(&revokedChain);
62 }
63
64 void SSLX509::ssl_conf_ca_chain(mbedtls_ssl_config *conf) {
65     mbedtls_ssl_conf_ca_chain(conf, &caCerts, &revokedChain);
66 }
67
68
69 int SSLX509::ssl_conf_own_cert(mbedtls_ssl_config *conf) {
70     mbedtls_ssl_conf_own_cert(conf, &clientCerts, &pkey);
71 }
72
73
74 int SSLX509::ssl_conf(mbedtls_ssl_config *conf) {
75     ssl_conf_ca_chain(conf);
76     return ssl_conf_own_cert(conf);
77 }
```

Listing B.11 – Enclave/TSTP/enclave_init.h

```
1 #ifndef __enclave_init_h
```

```
#define __enclave_init_h

#include <epos_common.h>

__BEGIN_SYS

void init(SSLConnection *);
void run(void);

__END_SYS

#endif
```

Listing B.12 – Enclave/TSTP/enclave_init.cpp

```
#include <tstp_stub.h>
#include <transducer.h>
#include "SSLConnection.h"
#include <stdio.h>

__BEGIN_SYS

SSLConnection *conn;

template<typename T>
class Printer: public Smart_Data_Common::Observer
{
public:
    Printer(T * t) : _data(t) {
        db<Printer>(TRC) << "Printer::Printer()" << endl;
        _data->attach(this);
        typename T::DB_Series s = t->db_series();
        char buff[512];
        snprintf(buff, sizeof(buff),
            "\"series\":{"
            "\"version\":\"%i\","
            "\"unit\":\"%lu\","
            "\"x\":\"%li\","
            "\"y\":\"%li\","
            "\"z\":\"%li\","
            "\"r\":\"%lu\","
            "\"t0\":\"%llu\","
            "\"t1\":\"%llu\""
            "\"dev\":\"%lu\""
            "},"
            "\"credentials\":{"
            "\"domain\":\"tutorial\","
            "\"username\":\"tutorial\","
            "\"password\":\"tuto2018\""
            "}"
            "}",
            s.version, s.unit, s.x, s.y, s.z, s.r, s.t0, s.t1, s.dev
        );
        conn->write(buff, strlen(buff));
    }
    ~Printer() { _data->detach(this); }

    void update(Smart_Data_Common::Observed * obs) {
        char buff[512];
```

```
45          db<Temperature>(TRC) << "Printer::update(obs=" << obs << ")" <<
    endl;
46          printf_sgx("---------Readed := %lf\n", ((T *)obs)->db_record().
    value);
47          typename T::DB_Record r = ((T *)obs)->db_record();
48          snprintf(buff, sizeof(buff),
49              "{\"smartdata\":"
50              "{"
51              "\"version\":\"%i\","
52              "\"unit\":\"%lu\","
53              "\"value\":\"%lf\","
54              "\"error\":\"%i\","
55              "\"confidence\":\"%i\","
56              "\"x\":\"%li\","
57              "\"y\":\"%li\","
58              "\"z\":\"%li\","
59              "\"t\":\"%llu\","
60              "\"dev\":\"%lu\""
61              "},"
62              "\"credentials\":"
63              "{"
64              "\"domain\":\"tutorial\","
65              "\"username\":\"tutorial\","
66              "\"password\":\"tuto2018\""
67              "}"
68              "}",
69              r.version, r.unit, r.value, r.error, r.confidence, r.x, r.y,
     r.z, r.t, r.dev
70          );
71          conn->write(buff, strlen(buff));
72      }
73
74 private:
75      T * _data;
76 };
77
78 TSTP *tstp;
79
80 int go;
81
82
83 void init(SSLConnection *c) {
84   conn = c;
85   tstp = new TSTP_Stub();
86   conn->connect();
87 }
88
89 #include "Enclave_t.h"
90
91 void run(void) {
92      typedef TSTP::Coordinates Coordinates;
93      typedef TSTP::Region Region;
94      printf_sgx("--------------------\n");
95      while (go <= 0) {
96          tsleep(10);
97      }
98
99      // Interest center points
```

```
100     Coordinates center_dummy0(1937,1153,0);
101     // Regions of interest
102     Region region_dummy0(center_dummy0, 0, 0, -1);
103 //
104     Temperature temp(region_dummy0, 2000000, 1000000);
105     printf_sgx("--------------------\n");
106     Printer<Temperature> t(&temp);
107 
108     while (true)
109         ;
110 }
111 
112 __END_SYS
```

Listing B.13 – Enclave/TSTP/Serial.edl

```
1 enclave {
2     trusted {
3         /* define ECALLs here. */
4         public void runThread(void);
5     };
6 
7     untrusted {
8         /* define OCALLs here. */
9         int Serial_init([in, string] const char *device);
10        void Serial_deinit(void);
11        int Serial_read([out, count=size, size=1] void *data, unsigned
      int size);
12        int Serial_write([in, count=size, size=1] const void *data,
      unsigned int size);
13        void tsleep(unsigned int size);
14     };
15 };
```

Listing B.14 – Enclave/TSTP/serial_trusted.h

```
1 
2 #ifndef __serial_trusted_h
3 #define __serial_trusted_h
4 
5 #include <unistd.h>
6 #include <EPOS/epos_common.h>
7 
8 __BEGIN_SYS
9 class Serial_Trusted;
10 __END_SYS
11 
12 #include <ieee802_15_4.h>
13 #include <tstp_common.h>
14 
15 __BEGIN_SYS
16 
17 class Runnable {
18 public:
19     virtual void *run(void *) = 0;
20 };
21 
22 class Serial_Trusted
23 {
```

```
24 private:
25     Serial_Trusted(const char *_device, Runnable *runnable, void *arg);
26     ~Serial_Trusted(void);
27 public:
28
29     static Serial_Trusted *getInstance(Runnable *runnable, void *arg);
30
31     ssize_t read(void *buff, size_t length);
32     ssize_t write(const void *buff, size_t length);
33
34 private:
35     friend void ::runThread(void);
36     static Serial_Trusted *_ser;
37     Runnable *_runnable;
38     void *_arg;
39 };
40
41 __END_SYS
42
43 #endif
```

Listing B.15 – Enclave/TSTP/serial_trusted.cpp

```
1
2 #include <serial_trusted.h>
3 #include <Enclave_t.h>
4
5 __BEGIN_SYS
6
7 Serial_Trusted *Serial_Trusted::_ser;
8
9
10 Serial_Trusted::Serial_Trusted(const char *_device, Runnable *runnable,
     void *arg) {
11     db<Serial_Trusted>(TRC) << "Serial_Trusted(s=" << _device << ")" <<
     endl;
12     sgx_status_t stat;
13     int ret;
14     stat = Serial_init(&ret, _device);
15     _runnable = runnable;
16     _arg = arg;
17 }
18
19 Serial_Trusted::~Serial_Trusted() {
20     Serial_deinit();
21 }
22
23 ssize_t Serial_Trusted::read(void *buff, size_t length) {
24     int readed;
25     sgx_status_t stat;
26     stat = Serial_read(&readed, buff, length);
27     return readed;
28 }
29
30 ssize_t Serial_Trusted::write(const void *buff, size_t length) {
31     int writed;
32     sgx_status_t stat;
33     stat = Serial_write(&writed, buff, length);
34     return writed;
```

```
35 }
36
37 Serial_Trusted *Serial_Trusted::getInstance(Runnable *runnable, void *
       arg) {
38     if (_ser == NULL) {
39         _ser = new Serial_Trusted("/tmp/serialSGX", runnable, arg);
40     }
41     return _ser;
42 }
43
44 __END_SYS
45
46 void runThread(void) {
47     _SYS::Serial_Trusted::_ser->_runnable->run(_SYS::Serial_Trusted::
       _ser->_arg);
48     return;
49 }
```

Listing B.16 – Enclave/TSTP/sgx_com_data_structures.h

```
1  // Included inside TSTP_Stub
2
3  enum CallTypes {
4      CALL_ANSWER,
5      ADD_PEER,
6      ALLOC,
7      SEND,
8      UPDATE,
9      NOW,
10     HERE,
11     MARSHAL_SEND,
12     KEY_MANAGER,
13     GO,
14     LAST, // to set sizeof es \/
15 } __attribute__((packed));
16
17 struct CallControl {
18     CallTypes type;
19     uint16_t dataSize;
20 } __attribute__ ((packed));
21
22 struct CallAddPeer {
23     Node_ID id;
24     Region valid_region;
25 } __attribute__ ((packed));
26
27 struct ENUM_STR {
28     enum CallTypes type;
29     const char *str;
30 };
31
32 const struct ENUM_STR es[LAST] = {
33     {CALL_ANSWER,   "\"CALL_ANSWER\"" },
34     {ADD_PEER,      "\"ADD_PEER\""    },
35     {ALLOC,         "\"ALLOC\""       },
36     {SEND,          "\"SEND\""        },
37     {UPDATE,        "\"UPDATE\""      },
38     {NOW,           "\"NOW\""         },
39     {HERE,          "\"HERE\""        },
```

```
40      {MARSHAL_SEND,   "\"MARSHAL_SEND\""},
41      {KEY_MANAGER,    "\"KEY_MANAGER\"" },
42      {GO,             "\"GO\""          }
43  };
44
45  friend OStream & operator<<(OStream & db, const CallControl cc) {
46      return db << "(tp=" << instance()->es[cc.type].str << ",sz=" << cc.
        dataSize << ")";
47  }
```

Listing B.17 – Enclave/TSTP/smart_data.h

```
1   // EPOS Smart Data Declarations
2
3   #ifndef __smart_data_h
4   #define __smart_data_h
5
6   #include <epos_common.h>
7   #include <list.h>
8   #include <observer.h>
9   #include <tstp.h>
10
11  __BEGIN_SYS
12
13  class Smart_Data_Common: public _UTIL::Observed
14  {
15  public:
16      typedef _UTIL::Observed Observed;
17      typedef _UTIL::Observer Observer;
18
19      struct DB_Series {
20          unsigned char version;
21          unsigned long unit;
22          long x;
23          long y;
24          long z;
25          unsigned long r;
26          unsigned long long t0;
27          unsigned long long t1;
28          unsigned long dev;
29          friend OStream & operator<<(OStream & os, const DB_Series & d) {
30              os << "{ve=" << d.version << ",u=" << d.unit << ",dst=(" <<
        d.x << "," << d.y << "," << d.z << ")+" << d.r << ",t=[" << d.t0 << "
        ," << d.t1 << "]}";
31              return os;
32          }
33      }__attribute__((packed));
34
35      struct SI_Record {
36          unsigned char version;
37          unsigned long unit;
38          double value;
39          unsigned char error;
40          unsigned char confidence;
41          long x;
42          long y;
43          long z;
44          unsigned long long t;
45          unsigned long dev;
```

```
46          friend OStream & operator<<(OStream & os, const SI_Record & d) {
47              os << "{ve=" << d.version << ",u=" << d.unit << ",va=" << d.
    value << ",e=" << d.error << ",src=(" << d.x << "," << d.y << "," <<
    d.z << "),t=" << d.t << ",d=" << d.dev << "}";
48              return os;
49          }
50      }__attribute__((packed));
51
52      template<unsigned int S>
53      struct Digital_Record {
54          unsigned char version;
55          unsigned long unit;
56          unsigned char value[S];
57          unsigned char error;
58          unsigned char confidence;
59          long x;
60          long y;
61          long z;
62          unsigned long long t;
63          unsigned long dev;
64          friend OStream & operator<<(OStream & os, const Digital_Record<S
    > & d) {
65              os << "{ve=" << d.version << ",u=" << d.unit << ",va=[
    digital]" << ",e=" << d.error << ",src=(" << d.x << "," << d.y << ","
     << d.z << "),t=" << d.t << ",d=" << d.dev << "}";
66              return os;
67          }
68      }__attribute__((packed));
69 };
70
71 template <typename T>
72 struct Smart_Data_Type_Wrapper
73 {
74     typedef T Type;
75 };
76
77 // Smart Data encapsulates Transducers (i.e. sensors and actuators),
    local or remote, and bridges them with TSTP
78 // Transducers must be Observed objects, must implement either sense()
    or actuate(), and must define UNIT, NUM, and ERROR.
79 template<typename Transducer>
80 class Smart_Data: public Smart_Data_Common, private TSTP::Observer/*,
    private Transducer::Observer*/
81 {
82      friend class Smart_Data_Type_Wrapper<Transducer>::Type; // friend S
    is OK in C++11, but this GCC does not implement it yet. Remove after
    GCC upgrade.
83
84 private:
85      typedef TSTP::Buffer Buffer;
86      typedef typename TSTP::Interested Interested;
87
88 public:
89      static const unsigned int UNIT = Transducer::UNIT;
90      static const unsigned int NUM = Transducer::NUM;
91      static const unsigned int ERROR = Transducer::ERROR;
92      typedef typename TSTP::Unit::Get<NUM>::Type Value;
93      typedef typename IF<(Transducer::UNIT >> 31) == 1, SI_Record,
```

```
    Digital_Record<Transducer::UNIT & 0xFFFF>>::Result DB_Record;
94
95      enum {
96          STATIC_VERSION = (1 << 4) + (1 << 0),
97          MOBILE_VERSION = (1 << 4) + (2 << 0),
98      };
99
100     enum Mode {
101         PRIVATE    = (0),
102         ADVERTISED = (1 << 0),
103         COMMANDED  = (1 << 1) | ADVERTISED,
104         CUMULATIVE = (1 << 2),
105         DISPLAYED  = (1 << 3),
106         PREDICTIVE = (1 << 4)
107     };
108
109     static const unsigned int REMOTE = -1;
110
111     typedef uint64_t Microsecond;
112
113     typedef TSTP::Unit Unit;
114     typedef TSTP::Error Error;
115     typedef TSTP::Coordinates Coordinates;
116     typedef TSTP::Region Region;
117     typedef TSTP::Time Time;
118     typedef TSTP::Time_Offset Time_Offset;
119
120 public:
121     // Remote, event-driven (period = 0) or time-triggered data source
122     Smart_Data(const Region & region, const Microsecond & expiry, const
    Microsecond & period = 0, const Mode & mode = PRIVATE)
123     : _unit(UNIT), _value(0), _error(ERROR), _coordinates(0), _time(0),
    _expiry(expiry), _device(REMOTE), _mode(static_cast<Mode>(mode & (~
    COMMANDED))) {
124         db<Smart_Data>(TRC) << "Smart_Data::Smart_Data(Region=" <<
    region << ",expiry=" << expiry << ",period=" << period << ")" << endl
    ;
125         _interested = new Interested(this, region, UNIT, TSTP::SINGLE,
    0, expiry, period);
126         TSTP::_instance->attach(this, _interested);
127     }
128
129     ~Smart_Data() {
130         db<Smart_Data>(TRC) << "Smart_Data::~Smart_Data(" << ")" << endl
    ;
131         TSTP::_instance->detach(this, _interested);
132         delete _interested;
133     }
134
135     DB_Record db_record() {
136         Value v = this->operator Value();
137         Time t = time();
138         TSTP::Global_Coordinates c = location();
139
140         DB_Record ret;
141         ret.version = STATIC_VERSION;
142         ret.unit = _unit;
143         ret.x = c.x;
```

```
144         ret.y = c.y;
145         ret.z = c.z;
146         ret.t = t;
147         ret.confidence = 0;
148         ret.dev = _device;
149 
150         if(EQUAL<DB_Record, SI_Record>::Result) {
151             SI_Record * si_record = reinterpret_cast<SI_Record*>(&ret);
152             si_record->error = error();
153             si_record->value = v;
154         } else {
155             assert(sizeof(Value) == Transducer::UNIT & 0xFFFF);
156             Digital_Record<Transducer::UNIT & 0xFFFF> * digital_record =
     reinterpret_cast<Digital_Record<Transducer::UNIT & 0xFFFF>*>(&ret);
157             digital_record->error = 0;
158             memcpy(&digital_record->value, &v, sizeof(Value));
159         }
160         return ret;
161     }
162 
163     DB_Series db_series(){
164         DB_Series ret;
165 
166         ret.version = STATIC_VERSION;
167         ret.unit = _unit;
168 
169         if(_interested) {
170             TSTP::Global_Coordinates c = TSTP::_instance->absolute(
    _interested->region().center);
171             ret.x = c.x;
172             ret.y = c.y;
173             ret.z = c.z;
174             ret.r = _interested->region().radius;
175             ret.t0 = TSTP::_instance->absolute(_interested->region().t0)
    ;
176             ret.t1 = TSTP::_instance->absolute(_interested->region().t1)
    ;
177         } else {
178             TSTP::Global_Coordinates c = location();
179             ret.x = c.x;
180             ret.y = c.y;
181             ret.z = c.z;
182             ret.r = 0;
183             ret.t0 = 0;
184             ret.t1 = -1;
185         }
186         ret.dev = _device;
187 
188         return ret;
189     }
190 
191     operator Value() {
192         if(expired()) {
193             // Other data sources must have called update() timely
194             db<Smart_Data>(WRN) << "Smart_Data::get(this=" << this << ",
    exp=" <<_time +  _expiry << ",val=" << _value << ") => expired!" <<
    endl;
195         }
```

```
196         Value ret = _value;
197         if(((_mode & CUMULATIVE) == CUMULATIVE))
198             _value = 0;
199         return ret;
200     }
201 
202     Smart_Data & operator=(const Value & v) {
203         if(_device != REMOTE)
204             Transducer::actuate(_device, this, v);
205         if(_interested)
206             _interested->command(v);
207 
208         return *this;
209     }
210 
211     bool expired() const {
212         return TSTP::_instance->now() > (_time + _expiry);
213     }
214 
215     TSTP::Global_Coordinates location() const {
216         return TSTP::_instance->absolute(_coordinates);
217     }
218     const Time time() const {
219         return TSTP::_instance->absolute(_time);
220     }
221     const Error & error() const {
222         return _error;
223     }
224     const Unit & unit() const {
225         return _unit;
226     }
227 
228     friend Debug & operator<<(Debug & db, const Smart_Data & d) {
229         db << "{";
230         if(d._device != REMOTE) {
231             switch(d._mode) {
232             case PRIVATE:    db << "PRI."; break;
233             case ADVERTISED: db << "ADV."; break;
234             case COMMANDED:  db << "CMD."; break;
235             }
236             db << "[" << d._device << "]:";
237         }
238         if(d._interested) db << "In" << ((d._interested->period()) ? "TT
    " : "ED");
239         db << ":u=" << d._unit << ",v=" << d._value << ",e=" << int(d.
    _error) << ",c=" << d._coordinates << ",t=" << d._time << ",x=" << d.
    _expiry << "}";
240         return db;
241     }
242 
243 private:
244     void update(TSTP::Observed * obs, int64_t subject, TSTP::Buffer *
    buffer) {
245         TSTP::Packet * packet = buffer->frame()->data<TSTP::Packet>();
246         db<Smart_Data>(TRC) << "Smart_Data::update(obs=" << obs << ",
    cond=" << reinterpret_cast<void *>(subject) << ",data=" << packet <<
    ")" << endl;
247         switch(packet->type()) {
```

```
248         case TSTP::RESPONSE: {
249             TSTP::Response * response = reinterpret_cast<TSTP::Response
    *>(packet);
250             db<Smart_Data>(INF) << "Smart_Data:update[R]:msg=" <<
    response << " => " << *response << endl;
251             if(response->time() > _time) {
252                 if ((_mode & CUMULATIVE) == CUMULATIVE)
253                     _value += response->value<Value>();
254                 else
255                     _value = response->value<Value>();
256                 _error = response->error();
257                 _coordinates = response->origin();
258                 _time = response->time();
259                 db<Smart_Data>(INF) << "Smart_Data:update[R]:this=" <<
    this << " => " << *this << endl;
260                 notify();
261             }
262         }
263         default:
264             break;
265         }
266     }
267
268 private:
269     Unit _unit;
270     Value _value;
271     Error _error;
272     Coordinates _coordinates;
273     TSTP::Time _time;
274     TSTP::Time _expiry;
275
276     unsigned int _device;
277     Mode _mode;
278     Interested *_interested;
279 };
280
281 // Smart Data Transform and Aggregation functions
282
283 class No_Transform
284 {
285 public:
286     No_Transform() {}
287
288     template<typename T, typename U>
289     void apply(T * result, U * source) {
290         typename U::Value v = *source;
291         *result = v;
292     }
293 };
294
295 template<typename T>
296 class Percent_Transform
297 {
298 public:
299     Percent_Transform(typename T::Value min, typename T::Value max) :
    _min(min), _step((max - min) / 100) {}
300
301     template<typename U>
```

```
302     void apply(U * result, T * source) {
303         typename T::Value v = *source;
304         if(v < _min)
305             *result = 0;
306         else {
307             v = (v - _min) / _step;
308             if(v > 100)
309                 v = 100;
310 
311             *result = v;
312         }
313     }
314 
315 private:
316     typename T::Value _min;
317     typename T::Value _step;
318 };
319 
320 template<typename T>
321 class Inverse_Percent_Transform: private Percent_Transform<T>
322 {
323 public:
324     Inverse_Percent_Transform(typename T::Value min, typename T::Value
    max) : Percent_Transform<T>(min, max) {}
325 
326     template<typename U>
327     void apply(U * result, T * source) {
328         typename U::Value r;
329         Percent_Transform<T>::apply(&r, source);
330         *result = 100 - r;
331     }
332 };
333 
334 class Sum_Transform
335 {
336 public:
337     Sum_Transform() {}
338 
339     template<typename T, typename ...U>
340     void apply(T * result, U * ... sources) {
341         *result = sum<T::Value, U...>((*sources)...);
342     }
343 
344 private:
345     template<typename T, typename U, typename ...V>
346     T sum(const U & s0, const V & ... s) { return s0 + sum<T, V...>(s
    ...); }
347 
348     template<typename T, typename U>
349     T sum(const U & s) { return s; }
350 };
351 
352 class Average_Transform: private Sum_Transform
353 {
354 public:
355     Average_Transform() {}
356 
357     template<typename T, typename ...U>
```

```
358     void apply(T * result, U * ... sources) {
359         typename T::Value r;
360         Sum_Transform::apply(&r, sources...);
361         *result = r / sizeof...(U);
362     }
363 };
364
365 // Smart Data Actuator
366
367 template<typename Destination, typename Transform, typename ...Sources>
368 class Actuator: public Smart_Data_Common::Observer
369 {
370 public:
371     Actuator(Destination * d, Transform * a, Sources * ... s)
372         : _destination(d), _transform(a), _sources{s...}
373     {
374         attach(s...);
375     }
376     ~Actuator() {
377         unsigned int index = 0;
378         detach((reinterpret_cast<Sources*>(_sources[index++]))...);
379     }
380
381     void update(Smart_Data_Common::Observed * obs) {
382         unsigned int index = 0;
383         _transform->apply(_destination, (reinterpret_cast<Sources*>(
    _sources[index++]))...);
384     }
385
386 private:
387     template<typename T, typename ...U>
388     void attach(T * t, U * ... u) { t->attach(this); attach(u...); }
389     template<typename T>
390     void attach(T * t) { t->attach(this); }
391
392     template<typename T, typename ...U>
393     void detach(T * t, U * ... u) { t->detach(this); detach(u...); }
394     template<typename T>
395     void detach(T * t) { t->detach(this); }
396
397 private:
398     Destination * _destination;
399     Transform * _transform;
400     void * _sources[sizeof...(Sources)];
401 };
402
403 __END_SYS
404
405 #endif
```

Listing B.18 – Enclave/TSTP/tstp.h

```
1 #ifndef TSTP_H_
2 #define TSTP_H_
3
4 #include <ostream>
5 #include <math.h>
6 #include <stlport/algorithm>
7
```

```
 8 #include <tstp_common.h>
 9 #include <hash.h>
10 #include <array.h>
11 #include <list.h>
12 #include <diffie_hellman.h>
13 #include <poly1305.h>
14
15 __BEGIN_SYS
16
17 typedef NIC_Common::NIC_Base<IEEE802_15_4, true> NIC;
18
19 class TSTP: public TSTP_Common, private IEEE802_15_4::Observer
20 {
21     template<typename> friend class Smart_Data;
22     friend class Epoch;
23     friend class Security;
24     friend class TSTP_Stub;
25
26 public:
27     typedef NIC::Buffer Buffer;
28
29     // Packet
30     template<Scale S>
31     class _Packet: public Header
32     {
33     private:
34         typedef unsigned char Data[MTU];
35
36     public:
37         _Packet() {}
38
39         Header * header() { return this; }
40
41         template<typename T>
42         T * data() { return reinterpret_cast<T *>(&_data); }
43
44         friend OStream & operator<<(OStream & db, const _Packet & p) {
45             switch(reinterpret_cast<const Header &>(p).type()) {
46                 case INTEREST:
47                     db << reinterpret_cast<const Interest &>(p);
48                     break;
49                 case RESPONSE:
50                     db << reinterpret_cast<const Response &>(p);
51                     break;
52                 case COMMAND:
53                     db << reinterpret_cast<const Command &>(p);
54                     break;
55                 case CONTROL: {
56                     switch(reinterpret_cast<const Control &>(p).subtype
    ()) {
57                         case DH_RESPONSE:
58                             db << reinterpret_cast<const DH_Response &>(
    p);
59                             break;
60                         case AUTH_REQUEST:
61                             db << reinterpret_cast<const Auth_Request
    &>(p);
62                             break;
```

```
63                          case DH_REQUEST:
64                              db << reinterpret_cast<const DH_Request &>(p
    );
65                              break;
66                          case AUTH_GRANTED:
67                              db << reinterpret_cast<const Auth_Granted
    &>(p);
68                              break;
69                          case REPORT:
70                              db << reinterpret_cast<const Report &>(p);
71                              break;
72                          case KEEP_ALIVE:
73                              db << reinterpret_cast<const Keep_Alive &>(p
    );
74                              break;
75                          case EPOCH:
76                              db << reinterpret_cast<const Epoch &>(p);
77                              break;
78                          default:
79                              break;
80                      }
81                  }
82                  default:
83                      break;
84              }
85              return db;
86          }
87      private:
88          Data _data;
89      } __attribute__((packed));
90      typedef _Packet<SCALE> Packet;
91
92      // TSTP observer/d conditioned to a message's address (ID)
93      typedef _UTIL::Data_Observer<Buffer, int64_t> Observer;
94      typedef _UTIL::Data_Observed<Buffer, int64_t> Observed;
95
96      // Hash to store TSTP Observers by type
97      class Interested;
98      typedef _UTIL::Hash<Interested, 10, Unit> Interests;
99      class Responsive;
100     typedef _UTIL::Hash<Responsive, 10, Unit> Responsives;
101
102     // Interest/Response Modes
103     enum Mode {
104         // Response
105         SINGLE = 0, // Only one response is desired for each interest
    job (desired, but multiple responses are still possible)
106         ALL    = 1, // All possible responses (e.g. from different
    sensors) are desired
107         // Interest
108         DELETE = 2  // Revoke an interest
109     };
110
111 #include "tstp_types.h"
112 // TSTP Security
113 #include "tstp_security.h"
114
115 public:
```

```
116     TSTP();
117     ~TSTP();
118
119     // Local network Space-Time
120     virtual Coordinates here() = 0;
121     virtual Time now() = 0;
122
123     static Coordinates sink(unsigned int unit = 0) {
124         return _instance->sink();
125     }
126
127     virtual Coordinates sinkCoord(unsigned int unit) = 0;
128
129     // Global Space-Time
130     Global_Coordinates absolute(const Coordinates & coordinates) {
    return _global_coordinates + coordinates; }
131     Time absolute(const Time & t) {
132         if((t == static_cast<Time>(-1)) || (t == 0))
133            return t;
134         return _epoch + t;
135     }
136
137     void attach(Observer * obs, void * subject) { _observed.attach(obs,
    int64_t(subject)); }
138     void detach(Observer * obs, void * subject) { _observed.detach(obs,
    int64_t(subject)); }
139     bool notify(void * subject, Buffer * buf);
140
141 protected:
142     void bootstrap();
143
144 private:
145     inline int min(int a, int b) { return a>b ? b : a; }
146     Region destination(Buffer * buf) {
147         switch(buf->frame()->data<Frame>()->type()) {
148             case INTEREST:
149                 return buf->frame()->data<Interest>()->region();
150             case RESPONSE:
151                 return Region(sink(), 0, buf->frame()->data<Response>()
    ->time(), buf->frame()->data<Response>()->expiry());
152             case COMMAND:
153                 return buf->frame()->data<Command>()->region();
154             case CONTROL:
155                 switch(buf->frame()->data<Control>()->subtype()) {
156                     default:
157                     case DH_RESPONSE:
158                     case AUTH_REQUEST: {
159                         Time origin = buf->frame()->data<Header>()->time
    ();
160                         Time deadline = origin + min(static_cast<
    unsigned long long>(_security->KEY_MANAGER_PERIOD), _security->
    KEY_EXPIRY) / 2;
161                         return Region(sink(), 0, origin, deadline);
162                     }
163                     case DH_REQUEST: {
164                         Time origin = buf->frame()->data<Header>()->time
    ();
165                         Time deadline = origin + min(static_cast<
```

```
    unsigned long long>(_security->KEY_MANAGER_PERIOD), _security->
    KEY_EXPIRY) / 2;
166                         return Region(buf->frame()->data<DH_Request>()->
    destination().center, buf->frame()->data<DH_Request>()->destination()
    .radius, origin, deadline);
167                     }
168                     case AUTH_GRANTED: {
169                         Time origin = buf->frame()->data<Header>()->time
    ();
170                         Time deadline = origin + min(static_cast<
    unsigned long long>(_security->KEY_MANAGER_PERIOD), _security->
    KEY_EXPIRY) / 2;
171                         return Region(buf->frame()->data<Auth_Granted>()
    ->destination().center, buf->frame()->data<Auth_Granted>()->
    destination().radius, origin, deadline);
172                     }
173                     case REPORT: {
174                         return Region(sink(), 0, buf->frame()->data<
    Report>()->time(), -1);
175                     }
176                     case KEEP_ALIVE: {
177                         while(true) {
178                             Coordinates fake(here().x + (Random::random
    () % (RADIO_RANGE / 3)), here().y + (Random::random() % (RADIO_RANGE
    / 3)), (here().z + Random::random() % (RADIO_RANGE / 3)));
179                             if(fake != here())
180                                 return Region(fake, 0, 0, -1); // Should
     never be destined_to_me
181                         }
182                     }
183                     case EPOCH: {
184                         return buf->frame()->data<Epoch>()->destination
    ();
185                     }
186                 }
187             default:
188                 trace() << "TSTP::destination(): ERROR: unrecognized
    frame type " << buf->frame()->data<Frame>()->type() << endl;
189                 return Region(TSTP::here(), 0, TSTP::now() - 2, TSTP::
    now() - 1);
190         }
191     }
192
193     virtual void marshal(Buffer * buf) {
194     }
195
196     void update(NIC::Observed * obs, NIC::Protocol prot, NIC::Buffer *
    buf);
197
198     typedef _UTIL::Vector<Coordinates, MAX_SINKS> Sinks;
199
200     virtual Buffer * alloc(unsigned int size) = 0;
201     virtual int send(NIC::Buffer * buf) = 0;
202     virtual int marshal_send(Buffer *buf) = 0;
203
204 public:
205     OStream & trace() {
206         return db<TSTP>(TRC);
```

```
207     }
208
209 private:
210     NIC * _nic;
211     Interests _interested;
212     Responsives _responsives;
213     Observed _observed; // Channel protocols are singletons
214     Time _epoch;
215     Global_Coordinates _global_coordinates;
216
217     unsigned int _unit;
218     static TSTP *_instance;
219
220     friend void run(void);
221     Security * _security;
222
223     Time _expiry;
224
225     unsigned int _allocated_buffers;
226     static Frame_ID id;
227     bool _encrypt;
228 };
229
230 __END_SYS
231
232 #endif
```

Listing B.19 – Enclave/TSTP/tstp.cpp

```
 1
 2 #include <tstp.h>
 3
 4 #include <cstring>
 5 #include <numeric>
 6
 7 __USING_SYS
 8
 9 unsigned int TSTP_Common::RADIO_RANGE;
10 bool TSTP_Common::drop_expired;
11 TSTP::Frame_ID TSTP::id;
12 TSTP * TSTP::_instance;
13
14 TSTP::TSTP() : _nic(0), _security(0), _allocated_buffers(0)
15 {
16     trace() << "TSTP::TSTP()" << endl;
17     _instance = this;
18     _security = new Security_SGX(this);
19 }
20
21 TSTP::~TSTP()
22 {
23     trace() << "TSTP::~TSTP()" << endl;
24     if(_security)
25         delete _security;
26 }
27 void TSTP::bootstrap()
28 {
29     trace() << "TSTP::bootstrap()" << endl;
30 }
```

```
31
32 bool TSTP::notify(void * subject, Buffer * buf)
33 {
34     static unsigned int seq_num = 0;
35     trace() << "TSTP::notify(s=" << subject << ",b=" << buf << ")" <<
    endl;
36     if((subject == 0) || (!_observed.notify(int64_t(subject), buf))) {
37         trace() << "Invalid Subject";
38     }
39
40     return true;
41 }
42
43 void TSTP::update(NIC::Observed * obs, NIC::Protocol prot, Buffer * buf)
44 {
45     trace() << "TSTP::update(obs=" << obs << ",buf=" << buf << ")" <<
    endl;
46
47     if(buf->is_microframe || !buf->destined_to_me || !buf->trusted)
48         return;
49
50     Packet * packet = buf->frame()->data<Packet>();
51
52     if(packet->time() > now())
53         return;
54
55     switch(packet->type()) {
56     case INTEREST: {
57         Interest * interest = reinterpret_cast<Interest *>(packet);
58         trace() << "TSTP::update:interest=" << interest << " => " << *
    interest << endl;
59         // Check for local capability to respond and notify interested
    observers
60         Responsives::List * list = _responsives[interest->unit()]; //
    TODO: What if sensor can answer multiple formats (e.g. int and float)
61         if(list)
62             for(Responsives::Element * el = list->head(); el; el = el->
    next()) {
63                 Responsive * responsive = el->object();
64                 if((now() < interest->region().t1) && interest->region()
    .contains(responsive->origin(), interest->region().t1))
65                     notify(responsive, buf);
66             }
67     } break;
68     case RESPONSE: {
69         Response * response = reinterpret_cast<Response *>(packet);
70         trace() << "TSTP::update:response=" << response << " => " << *
    response;
71         trace() << "response->time() = " << response->time() << endl;
72         trace() << "now() = " << now() << endl;
73         if(response->time() < now()) {
74             // Check region inclusion and notify interested observers
75             Interests::List * list = _interested[response->unit()];
76             if(list && !list->empty()) {
77                 for(Interests::Element * el = list->head(); el; el = el
    ->next()) {
78                     Interested * interested = el->object();
79                     if(interested->region().contains(response->origin(),
```

```
    response->time()))
80                      notify(interested, buf);
81                  }
82              } else
83                  notify(0, buf);
84          }
85      } break;
86      case COMMAND: {
87          Command * command = reinterpret_cast<Command *>(packet);
88          trace() << "TSTP::update:command=" << command << " => " << *
    command << endl;
89          // Check for local capability to respond and notify interested
    observers
90          Responsives::List * list = _responsives[command->unit()]; //
    TODO: What if sensor can answer multiple formats (e.g. int and float)
91          if(list)
92              for(Responsives::Element * el = list->head(); el; el = el->
    next()) {
93                  Responsive * responsive = el->object();
94                  if(command->region().contains(responsive->origin(), now
    ()))
95                      notify(responsive, buf);
96              }
97      } break;
98      case CONTROL: {
99          switch(buf->frame()->data<Control>()->subtype()) {
100             case DH_REQUEST:
101                 trace() << "TSTP::update: DH_Request: " << *buf->frame()
    ->data<DH_Request>() << endl;
102                 break;
103             case DH_RESPONSE:
104                 trace() << "TSTP::update: DH_Response: " << *buf->frame
    ()->data<DH_Response>();
105                 break;
106             case AUTH_REQUEST:
107                 trace() << "TSTP::update: Auth_Request: " << *buf->frame
    ()->data<Auth_Request>() << endl;
108                 break;
109             case AUTH_GRANTED:
110                 trace() << "TSTP::update: Auth_Granted: " << *buf->frame
    ()->data<Auth_Granted>() << endl;
111                 break;
112             case REPORT: {
113                 trace() << "TSTP::update: Report: " << *buf->frame()->
    data<Report>() << endl;
114                 Report * report = reinterpret_cast<Report *>(packet);
115                 if(report->time() < now()) {
116                     // Check region inclusion and advertise interested
    observers
117                     Interests::List * list = _interested[report->unit()
    ];
118                     if(list)
119                         for(Interests::Element * el = list->head(); el;
    el = el->next()) {
120                             Interested * interested = el->object();
121                             if(interested->region().contains(report->
    origin(), report->time()))
122                                 interested->advertise();
```

```
123                         }
124                     if(report->epoch_request() && (here() == sink())) {
125                         trace() << "TSTP::update: responding to Epoch
    request";
126                         Buffer * buf = alloc(sizeof(Epoch));
127                         marshal_send(buf);
128                     }
129                 }
130             } break;
131             case KEEP_ALIVE:
132                 trace() << "TSTP::update: Keep_Alive: " << *buf->frame()
    ->data<Keep_Alive>();
133                 break;
134             case EPOCH: {
135                 trace() << "TSTP::update: Epoch: " << *buf->frame()->
    data<Epoch>() << endl;
136                 Epoch * epoch = reinterpret_cast<Epoch *>(packet);
137                 if(here() != sink()) {
138                     _global_coordinates = epoch->coordinates();
139                     _epoch = epoch->epoch();
140                     trace() << "TSTP::update: Epoch: adjusted epoch
    Space-Time to: " << _global_coordinates << ", " << _epoch << endl;
141                 }
142             } break;
143             default:
144                 trace() << "TSTP::update: Unrecognized Control subtype:
    " << buf->frame()->data<Control>()->subtype() << endl;
145                 break;
146         }
147     } break;
148     default:
149         trace() << "TSTP::update: Unrecognized packet type: " << packet
    ->type() << endl;
150         break;
151     }
152 }
```

Listing B.20 – Enclave/TSTP/tstp_security_sgx.h

```
1
2
3 class Security_SGX: public Security
4 {
5 public:
6     typedef Diffie_Hellman::Shared_Key Master_Secret;
7     friend void run(void);
8     friend class TSTP_Stub;
9
10 private:
11     class Peer;
12     typedef _UTIL::Simple_List<Peer> Peers;
13     class Peer
14     {
15     public:
16         Peer(const Node_ID & id, const Region & v, TSTP * tstp)
17             : _id(id), _valid(v), _el(this), _auth_time(0), _tstp(tstp)
    {
18             ((Security_SGX *)_tstp->_security)->_cipher.encrypt(_id, _id
    , _auth);
```

```
19          }
20
21          void valid(const Region & r) { _valid = r; }
22          const Region & valid() const { return _valid; }
23
24          bool valid_deploy(const Coordinates & where, const Time & when)
    {
25              return _valid.contains(where, when);
26          }
27
28          bool valid_request(const Node_Auth & auth, const Coordinates &
    where, const Time & when) {
29              return !memcmp(auth, _auth, sizeof(Node_Auth)) && _valid.
    contains(where, when);
30          }
31
32          const Time & authentication_time() { return _auth_time; }
33
34          Peers::Element * link() { return &_el; }
35
36          const Master_Secret & master_secret() const { return
    _master_secret; }
37          void master_secret(const Master_Secret & ms) {
38              _master_secret = ms;
39              _auth_time = _tstp->now();
40          }
41
42          const Node_Auth & auth() const { return _auth; }
43          const Node_ID & id() const { return _id; }
44
45          friend OStream & operator<<(OStream & db, const Peer & p) {
46              db << "{id=" << p._id << ",au=" << p._auth << ",v=" << p.
    _valid << ",ms=" << p._master_secret << ",el=" << &p._el << "}";
47              return db;
48          }
49
50      private:
51          Node_ID _id;
52          Node_Auth _auth;
53          Region _valid;
54          Master_Secret _master_secret;
55          Peers::Element _el;
56          Time _auth_time;
57          TSTP * _tstp;
58      };
59
60      struct Pending_Key;
61      typedef _UTIL::Simple_List<Pending_Key> Pending_Keys;
62      class Pending_Key
63      {
64      public:
65          Pending_Key(const Public_Key & pk, TSTP * tstp) : _tstp(tstp),
    _master_secret_calculated(false), _creation(_tstp->now()),
    _public_key(pk), _el(this) {}
66
67          bool expired() { return _tstp->now() - _creation > _tstp->
    _security->KEY_EXPIRY; }
68
```

```
69          const Master_Secret & master_secret() {
70              if(_master_secret_calculated)
71                  return _master_secret;
72              _master_secret = ((Security_SGX *)_tstp->_security)->_dh.
    shared_key(_public_key);
73              _master_secret_calculated = true;
74              _tstp->trace() << "TSTP::Security_SGX::Pending_Key: Master
    Secret set: " << _master_secret;
75              return _master_secret;
76          }
77
78          Pending_Keys::Element * link() { return &_el; };
79
80          friend OStream & operator<<(OStream & db, const Pending_Key & p)
     {
81              db << "{msc=" << p._master_secret_calculated << ",c=" << p.
    _creation << ",pk=" << p._public_key << ",ms=" << p._master_secret <<
     ",el=" << &p._el << "}";
82              return db;
83          }
84
85      private:
86          TSTP * _tstp;
87          bool _master_secret_calculated;
88          Time _creation;
89          Public_Key _public_key;
90          Master_Secret _master_secret;
91          Pending_Keys::Element _el;
92      };
93
94  public:
95      Security_SGX(TSTP * tstp);
96
97      virtual ~Security_SGX();
98
99      void add_peer(const unsigned char * peer_id, unsigned int id_len,
    const Region & valid_region);
100
101     void bootstrap();
102     bool bootstrapped() { return _bootstrapped; }
103
104     bool synchronized() { return true; }
105
106     void marshal(Buffer * buf);
107
108     void update(NIC::Observed * obs, NIC::Protocol prot, NIC::Buffer *
    buf);
109
110 private:
111     bool unpack(const Peer * peer, unsigned char * msg, const unsigned
    char * mac, Time reception_time);
112
113     void encrypt(const void * msg, const Peer * peer, unsigned char *
    out);
114
115     OTP otp(const Master_Secret & master_secret, const Node_ID & id);
116
117     bool verify_auth_request(const Master_Secret & master_secret, const
```

```
    Node_ID & id, const OTP & otp);
118
119     int key_manager();
120
121 private:
122     Cipher _cipher;
123     Node_ID _id;
124     Node_Auth _auth;
125     Diffie_Hellman _dh;
126     Pending_Keys _pending_keys;
127     Peers _pending_peers;
128     Peers _trusted_peers;
129     volatile bool _peers_lock;
130     bool _key_manager;
131     unsigned int _dh_requests_open;
132     TSTP * _tstp;
133     bool _bootstrapped;
134 };
```

Listing B.21 – Enclave/TSTP/tstp_security_sgx.cpp

```
 1
 2 #include "tstp.h"
 3
 4 __BEGIN_SYS
 5
 6 TSTP::Security_SGX::Security_SGX(TSTP * tstp) : _key_manager(false),
     _dh_requests_open(0), _tstp(tstp), _bootstrapped(false) {
 7     _tstp->trace() << "TSTP::Security_SGX() => id=" << _tstp->_unit <<
     endl;
 8     Bignum<sizeof(Node_ID)> id(_tstp->_unit);
 9     new (&_id) Node_ID(&id, sizeof(id));
10     _cipher.encrypt(_id, _id, _auth);
11     bootstrap();
12 }
13
14 void TSTP::Security_SGX::bootstrap()
15 {
16     _tstp->trace() << "TSTP::Security_SGX::bootstrap()" << endl;
17
18     KEY_EXPIRY = 1l*60*60*1000000;
19     DHREQUEST_AUTH_DELAY = 1000000;
20     KEY_MANAGER_PERIOD = 10*1000000;
21     POLY_TIME_WINDOW = 60*1000000;
22     _bootstrapped = true;
23
24     if(bootstrapped())
25         _tstp->bootstrap();
26 }
27
28 // TSTP::Security_SGX
29 // Class attributes
30
31 // Methods
32 void TSTP::Security_SGX::update(NIC::Observed * obs, NIC::Protocol prot,
     Buffer * buf)
33 {
34     bool was_bootstrapped = bootstrapped();
35
```

```
36      _tstp->trace() << "TSTP::Security_SGX::update(obs=" << obs << ",buf=
    " << buf << ")" << endl;
37
38      if(!buf->is_microframe && buf->destined_to_me) {
39
40          switch(buf->frame()->data<Header>()->type()) {
41
42              case CONTROL: {
43                  _tstp->trace() << "TSTP::Security_SGX::update(): Control
     message received" << endl;
44                  switch(buf->frame()->data<Control>()->subtype()) {
45                      case REPORT: {
46                          _tstp->trace() << "TSTP::Security_SGX::update():
     Report" << endl;
47                          buf->trusted = true;
48                      } break;
49
50                      case DH_REQUEST: {
51                      } break;
52
53                      case DH_RESPONSE: {
54                          if(_dh_requests_open) {
55                              DH_Response * dh_resp = buf->frame()->data<
    DH_Response>();
56                              _tstp->trace() << "TSTP::Security_SGX::
    update(): DH_Response message received: " << *dh_resp << endl;
57
58                              bool valid_peer = false;
59                              for(Peers::Element * el = _pending_peers.
    head(); el; el = el->next())
60                                  if(el->object()->valid_deploy(dh_resp->
    origin(), _tstp->now())) {
61                                      valid_peer = true;
62                                      _tstp->trace() << "Valid peer found:
     " << *el->object() << endl;
63                                      break;
64                                  }
65
66                              if(valid_peer) {
67                                  _dh_requests_open--;
68                                  Pending_Key * pk = new Pending_Key(buf->
    frame()->data<DH_Response>()->key(), _tstp);
69                                  _pending_keys.insert(pk->link());
70                                  _tstp->trace() << "TSTP::Security_SGX::
    update(): Inserting new Pending Key: " << *pk << endl;
71                              }
72                          }
73                      } break;
74
75                      case AUTH_REQUEST: {
76
77                          Auth_Request * auth_req = buf->frame()->data<
    Auth_Request>();
78                          _tstp->trace() << "TSTP::Security_SGX::update():
     Auth_Request message received: " << *auth_req << endl;
79
80                          Peer * auth_peer = 0;
81                          for(Peers::Element * el = _pending_peers.head();
```

```
    el; el = el->next()) {
82                              Peer * peer = el->object();
83
84                              if(peer->valid_request(auth_req->auth(),
    auth_req->origin(), _tstp->now())) {
85                                  _tstp->trace() << "valid_request " <<
    endl;
86                                  for(Pending_Keys::Element * pk_el =
    _pending_keys.head(); pk_el; pk_el = pk_el->next()) {
87                                      Pending_Key * pk = pk_el->object();
88                                      if(verify_auth_request(pk->
    master_secret(), peer->id(), auth_req->otp())) {
89                                          peer->master_secret(pk->
    master_secret());
90                                          _pending_peers.remove(el);
91                                          _trusted_peers.insert(el);
92                                          auth_peer = peer;
93
94                                          _pending_keys.remove(pk_el);
95                                          delete pk_el->object();
96
97                                          break;
98                                      }
99                                  }
100                                 if(auth_peer)
101                                     break;
102                             }
103                         }
104
105                         if(auth_peer) {
106                             Node_Auth encrypted_auth;
107                             encrypt(auth_peer->auth(), auth_peer,
    encrypted_auth);
108
109                             Buffer * resp = _tstp->alloc(sizeof(
    Auth_Granted));
110                             new (resp->frame()) Auth_Granted(Region::
    Space(auth_peer->valid().center, auth_peer->valid().radius),
    encrypted_auth);
111                             _tstp->trace() << "TSTP::Security_SGX:
    Sending Auth_Granted message " << resp->frame()->data<Auth_Granted>()
     << endl;
112                             _tstp->marshal_send(resp);
113                         } else
114                             _tstp->trace() << "TSTP::Security_SGX::
    update(): No peer found" << endl;
115                 } break;
116
117                 case AUTH_GRANTED: {
118                 } break;
119
120                 default: break;
121             }
122         } break;
123         case RESPONSE: {
124             _tstp->trace() << "TSTP::Security_SGX::update():
    Response message received";
125             _tstp->trace() << " from " << buf->frame()->data<Header
```

```
    >()->origin() << endl;
126                 Response * resp = buf->frame()->data<Response>();
127                 Time reception_time = _tstp->now();
128                 for(Peers::Element * el = _trusted_peers.head(); el; el
    = el->next()) {
129                     if(el->object()->valid_deploy(buf->frame()->data<
    Header>()->origin(), _tstp->now())) {
130                         _tstp->trace() << "Found valid peer" << endl;
131                         unsigned char * data = resp->data<unsigned char
    >();
132                         if(unpack(el->object(), data, &data[sizeof(
    Master_Secret)], reception_time)) {
133                             buf->trusted = true;
134                             _tstp->trace() << "Unpack done!" << endl;
135                             break;
136                         } else {
137                             _tstp->trace() << "Unpack failed" << endl;
138                         }
139                     }
140                 }
141             } break;
142             case INTEREST: {
143                 buf->trusted = true;
144             } break;
145             default: break;
146         }
147     } else
148         buf->trusted = true;
149
150     if((!was_bootstrapped) && bootstrapped())
151         _tstp->bootstrap();
152 }
153
154 void TSTP::Security_SGX::marshal(Buffer * buf)
155 {
156     _tstp->trace() << "TSTP::Security_SGX::marshal(buf=" << buf << ")"
    << endl;
157     buf->trusted = false;
158 }
159
160 TSTP::Security_SGX::~Security_SGX()
161 {
162     _tstp->trace() << "TSTP::~Security_SGX()" << endl;
163     while(auto el = _trusted_peers.remove_head())
164         delete el->object();
165     while(auto el = _pending_peers.remove_head())
166         delete el->object();
167     while(auto el = _pending_keys.remove_head())
168         delete el->object();
169
170 }
171
172 void TSTP::Security_SGX::add_peer(const unsigned char * peer_id,
    unsigned int id_len, const Region & valid_region) {
173     Node_ID id(peer_id, id_len);
174     Peer * peer = new Peer(id, valid_region, _tstp);
175     _pending_peers.insert(peer->link());
176     if(!_key_manager) {
```

```
177         _key_manager = true;
178     }
179 }
180
181 bool TSTP::Security_SGX::unpack(const Peer * peer, unsigned char * msg,
    const unsigned char * mac, Time reception_time) {
182     unsigned char original_msg[sizeof(Master_Secret)];
183     memcpy(original_msg, msg, sizeof(Master_Secret));
184
185     const unsigned char * ms = reinterpret_cast<const unsigned char *>(&
    peer->master_secret());
186     const unsigned char * id = reinterpret_cast<const unsigned char *>(&
    peer->id());
187
188     // mi = ms ^ _id
189     static const unsigned int MI_SIZE = sizeof(Node_ID) > sizeof(
    Master_Secret) ? sizeof(Node_ID) : sizeof(Master_Secret);
190     unsigned char mi[MI_SIZE];
191     unsigned int i;
192     for(i = 0; (i < sizeof(Node_ID)) && (i < sizeof(Master_Secret)); i
    ++)
193         mi[i] = id[i] ^ ms[i];
194     for(; i < sizeof(Node_ID); i++)
195         mi[i] = id[i];
196     for(; i < sizeof(Master_Secret); i++)
197         mi[i] = ms[i];
198
199     reception_time /= POLY_TIME_WINDOW;
200     _tstp->trace() << "t = " << reception_time << endl;
201
202     OTP key;
203     unsigned char nonce[16];
204     Poly1305 poly(id, ms);
205
206     memset(nonce, 0, 16);
207     memcpy(nonce, &reception_time, sizeof(Time) < 16u ? sizeof(Time) :
    16u);
208     poly.stamp(key, nonce, mi, MI_SIZE);
209     if(_tstp->_encrypt)
210         _cipher.decrypt(original_msg, key, msg);
211     if(poly.verify(mac, nonce, msg, sizeof(Master_Secret)))
212         return true;
213
214     reception_time--;
215     memset(nonce, 0, 16);
216     memcpy(nonce, &reception_time, sizeof(Time) < 16u ? sizeof(Time) :
    16u);
217     poly.stamp(key, nonce, mi, MI_SIZE);
218     if(_tstp->_encrypt)
219         _cipher.decrypt(original_msg, key, msg);
220     if(poly.verify(mac, nonce, msg, sizeof(Master_Secret)))
221         return true;
222
223     reception_time += 2;
224     memset(nonce, 0, 16);
225     memcpy(nonce, &reception_time, sizeof(Time) < 16u ? sizeof(Time) :
    16u);
226     poly.stamp(key, nonce, mi, MI_SIZE);
```

```
227     if(_tstp->_encrypt)
228         _cipher.decrypt(original_msg, key, msg);
229     if(poly.verify(mac, nonce, msg, sizeof(Master_Secret)))
230         return true;
231 
232     memcpy(msg, original_msg, sizeof(Master_Secret));
233     return false;
234 }
235 
236 void TSTP::Security_SGX::encrypt(const void * msg, const Peer * peer,
    unsigned char * out) {
237     OTP key = otp(peer->master_secret(), peer->id());
238     _cipher.encrypt((const unsigned char *)msg, key, out);
239 }
240 
241 TSTP::OTP TSTP::Security_SGX::otp(const Master_Secret & master_secret,
    const Node_ID & id) {
242     const unsigned char * ms = reinterpret_cast<const unsigned char *>(&
    master_secret);
243 
244     // mi = ms ^ _id
245     static const unsigned int MI_SIZE = sizeof(Node_ID) > sizeof(
    Master_Secret) ? sizeof(Node_ID) : sizeof(Master_Secret);
246     unsigned char mi[MI_SIZE];
247     unsigned int i;
248     for(i = 0; (i < sizeof(Node_ID)) && (i < sizeof(Master_Secret)); i
    ++)
249         mi[i] = id[i] ^ ms[i];
250     for(; i < sizeof(Node_ID); i++)
251         mi[i] = id[i];
252     for(; i < sizeof(Master_Secret); i++)
253         mi[i] = ms[i];
254 
255     Time t = _tstp->now() / POLY_TIME_WINDOW;
256 
257     unsigned char nonce[16];
258     memset(nonce, 0, 16);
259     memcpy(nonce, &t, sizeof(Time) < 16u ? sizeof(Time) : 16u);
260 
261     OTP out;
262     Poly1305(id, ms).stamp(out, nonce, mi, MI_SIZE);
263     return out;
264 }
265 
266 bool TSTP::Security_SGX::verify_auth_request(const Master_Secret &
    master_secret, const Node_ID & id, const OTP & otp) {
267     const unsigned char * ms = reinterpret_cast<const unsigned char *>(&
    master_secret);
268 
269     // mi = ms ^ _id
270     static const unsigned int MI_SIZE = sizeof(Node_ID) > sizeof(
    Master_Secret) ? sizeof(Node_ID) : sizeof(Master_Secret);
271     unsigned char mi[MI_SIZE];
272     unsigned int i;
273     for(i = 0; (i < sizeof(Node_ID)) && (i < sizeof(Master_Secret)); i
    ++)
274         mi[i] = id[i] ^ ms[i];
275     for(; i < sizeof(Node_ID); i++)
```

```
276         mi[i] = id[i];
277     for(; i < sizeof(Master_Secret); i++)
278         mi[i] = ms[i];
279 
280     unsigned char nonce[16];
281     Time t = _tstp->now() / POLY_TIME_WINDOW;
282 
283     Poly1305 poly(id, ms);
284 
285     memset(nonce, 0, 16);
286     memcpy(nonce, &t, sizeof(Time) < 16u ? sizeof(Time) : 16u);
287     if(poly.verify(otp, nonce, mi, MI_SIZE))
288         return true;
289 
290     t--;
291     memset(nonce, 0, 16);
292     memcpy(nonce, &t, sizeof(Time) < 16u ? sizeof(Time) : 16u);
293     if(poly.verify(otp, nonce, mi, MI_SIZE))
294         return true;
295 
296     t += 2;
297     memset(nonce, 0, 16);
298     memcpy(nonce, &t, sizeof(Time) < 16u ? sizeof(Time) : 16u);
299     if(poly.verify(otp, nonce, mi, MI_SIZE))
300         return true;
301 
302     return false;
303 }
304 
305 int TSTP::Security_SGX::key_manager() {
306 
307     Peers::Element * last_dh_request = 0;
308 
309     _tstp->trace() << "TSTP::Security_SGX::key_manager()" << endl;
310 
311     // Cleanup expired pending keys
312     Pending_Keys::Element * next_key;
313     for(Pending_Keys::Element * el = _pending_keys.head(); el; el =
    next_key) {
314         next_key = el->next();
315         Pending_Key * p = el->object();
316         if(p->expired()) {
317             _pending_keys.remove(el);
318             delete p;
319             _tstp->trace() << "TSTP::Security_SGX::key_manager():
    removed pending key" << endl;
320         }
321     }
322 
323     // Cleanup expired peers
324     Peers::Element * next;
325     for(Peers::Element * el = _trusted_peers.head(); el; el = next) {
326         next = el->next();
327         Peer * p = el->object();
328         if(!p->valid_deploy(p->valid().center, _tstp->now())) {
329             _trusted_peers.remove(el);
330             delete p;
331             _tstp->trace() << "TSTP::Security_SGX::key_manager():
```

```
    permanently removed trusted peer";
332         }
333     }
334     for(Peers::Element * el = _pending_peers.head(); el; el = next) {
335         next = el->next();
336         Peer * p = el->object();
337         if(!p->valid_deploy(p->valid().center, _tstp->now())) {
338             _pending_peers.remove(el);
339             delete p;
340             _tstp->trace() << "TSTP::Security_SGX::key_manager():
    permanently removed pending peer";
341         }
342     }
343
344     // Cleanup expired established keys
345     for(Peers::Element * el = _trusted_peers.head(); el; el = next) {
346         next = el->next();
347         Peer * p = el->object();
348         if(_tstp->now() - p->authentication_time() > KEY_EXPIRY) {
349             _trusted_peers.remove(el);
350             _pending_peers.insert(el);
351             _tstp->trace() << "TSTP::Security_SGX::key_manager():
    trusted peer's key expired";
352         }
353     }
354
355     // Send DH Request to at most one peer
356     Peers::Element * el;
357     if(last_dh_request && last_dh_request->next())
358         el = last_dh_request->next();
359     else
360         el = _pending_peers.head();
361
362     last_dh_request = 0;
363
364     for(; el; el = el->next()) {
365         Peer * p = el->object();
366         if(p->valid_deploy(p->valid().center, _tstp->now())) {
367             last_dh_request = el;
368             Buffer * buf = _tstp->alloc(sizeof(DH_Request));
369             new (buf->frame()->data<DH_Request>()) DH_Request(Region::
    Space(p->valid().center, p->valid().radius), _dh.public_key());
370             _dh_requests_open++;
371             _tstp->marshal_send(buf);
372             _tstp->trace() << "TSTP::Security_SGX::key_manager(): Sent
    DH_Request: "  << *buf->frame()->data<DH_Request>() << endl;
373             break;
374         }
375     }
376
377     return 0;
378 }
379
380 __END_SYS
```

Listing B.22 – Enclave/TSTP/tstp_stub.h

```
1 #ifndef TSTP_STUB_H_
2 #define TSTP_STUB_H_
```

```
 3
 4 #include <tstp.h>
 5 #include <hash.h>
 6 #include <array.h>
 7 #include <list.h>
 8 #include <diffie_hellman.h>
 9 #include <poly1305.h>
10 #include <nic.h>
11
12 #include "serial_trusted.h"
13
14
15 #include <sgx_thread.h>
16
17 __BEGIN_SYS
18
19 class TSTP_Stub: public TSTP, public Runnable
20 {
21 #include "sgx_com_data_structures.h"
22
23 #include "tstp_security_sgx.h"
24
25 public:
26     TSTP_Stub(void) {
27         db<TSTP_Stub>(TRC) << "TSTP_Stub::TSTP_Stub => " << this << endl
    ;
28         _encrypt = true;
29         packetManagerThread = 0;
30         getAnswerThread= 0;
31         _answer = NULL;
32         _dev = Serial_Trusted::getInstance(this, this);
33         sgx_thread_mutex_init(&mutexAnswer, NULL);
34         sgx_thread_mutex_init(&mutexWrite, NULL);
35         sgx_thread_cond_init(&condAnswer, NULL);
36         sgx_thread_mutex_lock(&mutexAnswer);
37     }
38
39     void write(enum CallTypes type, void *buf, uint16_t length) {
40         CallControl cc;
41         sgx_thread_mutex_lock(&mutexWrite);
42         cc.type = type;
43         cc.dataSize = length;
44         db<TSTP_Stub>(TRC) << "TSTP_Stub::write(cc=" << cc << ")" <<
    endl;
45         _dev->write(&cc, sizeof(CallControl));
46         _dev->write(buf, cc.dataSize);
47         freeWrite();
48     }
49
50     void freeWrite(void) {
51         sgx_thread_mutex_unlock(&mutexWrite);
52     }
53
54     void *getAnswer();
55     void setAnswer(void *ans);
56
57     int send(NIC::Buffer * buf);
58     Buffer * alloc(unsigned int size);
```

```
59      // Local network Space-Time
60      Coordinates here();
61      Time now();
62      Coordinates sinkCoord(unsigned int unit);
63      void marshal(Buffer * buf);
64      int marshal_send(Buffer *buf);
65
66      virtual void *run(void *arg);
67      void packetManager(void);
68
69      static TSTP_Stub *instance() {
70          return (TSTP_Stub *)_instance;
71      }
72
73  private:
74      sgx_thread_mutex_t mutexAnswer;
75      sgx_thread_cond_t condAnswer;
76      sgx_thread_mutex_t mutexWrite;
77      sgx_thread_t packetManagerThread;
78      sgx_thread_t getAnswerThread;
79      void *_answer;
80      static Serial_Trusted * _dev;
81  };
82
83  __END_SYS
84
85  #endif
```

Listing B.23 – Enclave/TSTP/tstp_stub.cpp

```
1
2   #include <tstp_stub.h>
3
4   __BEGIN_SYS
5
6   Serial_Trusted *TSTP_Stub::_dev;
7
8   void *TSTP_Stub::run(void *arg) {
9       packetManagerThread = sgx_thread_self();
10      while (true) {
11          packetManager();
12      }
13      return NULL;
14  }
15
16  void *TSTP_Stub::getAnswer() {
17      void *ans;
18      db<TSTP_Stub>(TRC) << "TSTP_Stub::getAnswer()";
19      getAnswerThread = sgx_thread_self();
20      if (getAnswerThread == packetManagerThread) {
21          _answer = NULL;
22          db<TSTP_Stub>(TRC) << " => packet=" << _answer << endl;
23          while (_answer == NULL) {
24              packetManager();
25          }
26      } else {
27          db<TSTP_Stub>(TRC) << " => other" << endl;
28          sgx_thread_cond_wait(&condAnswer, &mutexAnswer);
29      }
```

```
30      ans = _answer;
31      _answer = NULL;
32      return ans;
33  }
34
35  void TSTP_Stub::setAnswer(void *ans) {
36      db<TSTP_Stub>(TRC) << "TSTP_Stub::setAnswer()" << endl;
37      if (_answer) {
38          free(_answer);
39      }
40      _answer = ans;
41      if (sgx_thread_self() != getAnswerThread) {
42          sgx_thread_cond_signal(&condAnswer);
43      }
44  }
45
46  void TSTP_Stub::packetManager(void) {
47      CallControl cc;
48      void *data;
49
50      _dev->read(&cc, sizeof(CallControl));
51      db<TSTP_Stub>(TRC) << "TSTP_Stub::packetManager(cc=" << cc << ")" <<
        endl;
52      if (cc.dataSize > 0) {
53          data = malloc(cc.dataSize);
54          _dev->read(data, cc.dataSize);
55      } else {
56          data = NULL;
57      }
58      switch (cc.type) {
59          case CALL_ANSWER: {
60              setAnswer(data);
61          } break;
62          case ADD_PEER: {
63              struct CallAddPeer *peer = (struct CallAddPeer *)data;
64              db<TSTP_Stub>(TRC) << "TSTP_Stub::add_peer(id=" << peer->id
      << ",r=" << peer->valid_region << ")" << endl;
65              _security->add_peer(peer->id, sizeof(Node_ID), peer->
      valid_region);
66              ::free(data);
67          } break;
68          case UPDATE: {
69              uint8_t trust;
70              db<TSTP_Stub>(TRC) << "TSTP_Stub::update(" << ")" << endl;
71              Buffer *buf = (Buffer *)data;
72              _security->update(NULL, 0, buf);
73              TSTP::update(NULL, 0, buf);
74              if (buf->trusted) {
75                  trust = 1;
76              } else {
77                  trust = 0;
78              }
79              write(CALL_ANSWER, &trust, sizeof(uint8_t));
80              free(data);
81          } break;
82          case KEY_MANAGER: {
83              int32_t ret = 0;
84              ret = ((TSTP::Security_SGX *)_security)->key_manager();
```

```
85              write(CALL_ANSWER, &ret, sizeof(int32_t));
86          } break;
87          case GO: {
88              extern int go;
89              db<TSTP_Stub>(TRC) << "TSTP_Stub::go(" << ")" << endl;
90              go++;
91          } break;
92          default: {
93              ::free(data);
94          } break;
95      }
96 }
97
98 int TSTP_Stub::send(NIC::Buffer * buf) {
99      db<TSTP_Stub>(TRC) << "TSTP_Stub::send(bf" << buf << ")" << endl;
100     int32_t *ptr, ret;
101     write(SEND, buf, sizeof(Buffer));
102     delete buf;
103     ptr= (int32_t *)getAnswer();
104     ret = *ptr;
105     free(ptr);
106     return ret;
107 }
108
109 TSTP_Stub::Buffer *TSTP_Stub::alloc(unsigned int size) {
110     db<TSTP_Stub>(TRC) << "TSTP_Stub::alloc(sz=" << size << ")" << endl;
111     _allocated_buffers++;
112     Buffer *buf;
113     uint16_t sz = size;
114     write(ALLOC, &sz, sizeof(uint16_t));
115     buf = (Buffer *)getAnswer();
116     return buf;
117 }
118
119 TSTP_Stub::Time TSTP_Stub::now() {
120     db<TSTP_Stub>(TRC) << "TSTP_Stub::now()" << endl;
121     write(NOW, NULL, 0);
122     Time *ptr, t;
123     ptr = (Time *)getAnswer();
124     db<TSTP_Stub>(TRC) << "TSTP_Stub::now(ptr="<< ptr <<")" << endl;
125     t = *ptr;
126     db<TSTP_Stub>(TRC) << "TSTP_Stub::now(t="<< (uint64_t)t <<")" <<
    endl;
127     free(ptr);
128     return t;
129 }
130
131 // Local network Space-Time
132 TSTP_Stub::Coordinates TSTP_Stub::here() {
133     db<TSTP_Stub>(TRC) << "TSTP_Stub::here()" << endl;
134     write(HERE, NULL, 0);
135     Coordinates *ptr, c;
136     ptr = (Coordinates *)getAnswer();
137     c = *ptr;
138     db<TSTP_Stub>(TRC) << "TSTP_Stub::here(c="<< c <<")" << endl;
139     free(ptr);
140     return c;
141 }
```

```
142
143 TSTP_Stub::Coordinates TSTP_Stub::sinkCoord(unsigned int unit = 0) {
144     return 0;
145 }
146
147 void TSTP_Stub::marshal(Buffer *buf) {
148 }
149
150 int TSTP_Stub::marshal_send(Buffer *buf) {
151     db<TSTP_Stub>(TRC) << "TSTP_Stub::marshal_send()" << endl;
152     int32_t *ptr, ret;
153     write(MARSHAL_SEND, buf, sizeof(Buffer));
154     delete buf;
155     ptr= (int32_t *)getAnswer();
156     ret = *ptr;
157     free(ptr);
158     return ret;
159 }
160
161 __END_SYS
```

Listing B.24 – Enclave/TSTP/tstp_types.h

```
1
2 // Interest Message
3 class Interest: public Header
4 {
5 public:
6     Interest(const Region & region, const Unit & unit, const Mode & mode
      , const Error & precision, const Microsecond & expiry, const
      Microsecond & period = 0)
7     : Header(INTEREST, 0, 0, TSTP::_instance->now(), TSTP::_instance->
      here(), TSTP::_instance->here()), _region(region), _unit(unit), _mode
      (mode), _precision(0), _expiry(expiry), _period(period) {}
8
9     const Unit & unit() const { return _unit; }
10    const Region & region() const { return _region; }
11    Microsecond period() const { return _period; }
12    Time_Offset expiry() const { return _expiry; }
13    Mode mode() const { return static_cast<Mode>(_mode); }
14    Error precision() const { return static_cast<Error>(_precision); }
15
16    bool time_triggered() { return _period; }
17    bool event_driven() { return !time_triggered(); }
18
19    friend OStream & operator<<(OStream & db, const Interest & m) {
20        db << reinterpret_cast<const Header &>(m) << ",u=" << m._unit <<
      ",m=" << ((m._mode == ALL) ? 'A' : 'S') << ",e=" << int(m._precision
      ) << ",x=" << m._expiry << ",re=" << m._region << ",p=" << m._period;
21        return db;
22    }
23
24 protected:
25    Region _region;
26    Unit _unit;
27    unsigned char _mode : 2;
28    unsigned char _precision : 6;
29    Time_Offset _expiry;
30    Microsecond _period; // TODO: should be Time_Offset
```

```
31      CRC _crc;
32  } __attribute__((packed));
33
34  // Response (Data) Message
35  class Response: public Header
36  {
37      friend class TSTP;
38  public:
39      typedef unsigned char Data[MTU - sizeof(Unit) - sizeof(Error) -
        sizeof(Time_Offset) - sizeof(Message_Auth) - sizeof(CRC)];
40
41  public:
42      Response(TSTP * tstp, const Unit & unit, const Error & error = 0,
        const Time & expiry = 0)
43      : Header(RESPONSE, 0, 0, TSTP::_instance->now(), TSTP::_instance->
        here(), TSTP::_instance->here()), _unit(unit), _error(error), _expiry
        (expiry) {}
44
45      const Unit & unit() const { return _unit; }
46      Error error() const { return _error; }
47      Time expiry() const { return _time + _expiry; }
48      void expiry(const Time & e) { _expiry = e - _time; }
49
50      template<typename T>
51      void value(const T & v) { *reinterpret_cast<Value<Unit::GET<T>::NUM>
        *>(&_data) = v; }
52
53      template<typename T>
54      T value() { return *reinterpret_cast<Value<Unit::GET<T>::NUM> *>(&
        _data); }
55
56      template<typename T>
57      T * data() { return reinterpret_cast<T *>(&_data); }
58
59      friend OStream & operator<<(OStream & db, const Response & m) {
60          db << reinterpret_cast<const Header &>(m) << ",u=" << m._unit <<
         ",e=" << int(m._error) << ",x=" << m._expiry << ",d=" << OStream::
        hex << *const_cast<Response &>(m).data<unsigned>() << OStream::dec;
61          return db;
62      }
63
64  protected:
65      Unit _unit;
66      Error _error;
67      Time_Offset _expiry;
68      Data _data;
69      Message_Auth _auth; // Data size is variable, so _auth is not
        directly accessible
70      CRC _crc;
71  } __attribute__((packed));
72
73  // Command Message
74  class Command: public Header
75  {
76  private:
77      typedef unsigned char Data[MTU - sizeof(Region) - sizeof(Unit) -
        sizeof(CRC)];
78
```

```
79 public:
80     Command(const Unit & unit, const Region & region)
81     : Header(COMMAND, 0, 0, TSTP::_instance->now(), TSTP::_instance->
       here(), TSTP::_instance->here()), _region(region), _unit(unit) {}
82
83     const Region & region() const { return _region; }
84     const Unit & unit() const { return _unit; }
85
86     template<typename T>
87     T * command() { return reinterpret_cast<T *>(&_data); }
88
89     template<typename T>
90     T * data() { return reinterpret_cast<T *>(&_data); }
91
92     friend OStream & operator<<(OStream & db, const Command & m) {
93         db << reinterpret_cast<const Header &>(m) << ",u=" << m._unit <<
       ",reg=" << m._region;
94         return db;
95     }
96
97 protected:
98     Region _region;
99     Unit _unit;
100    Data _data;
101    CRC _crc;
102 } __attribute__((packed));
103
104 // Diffie-Hellman Request Security Bootstrap Control Message
105 class DH_Request: public Control
106 {
107 public:
108    DH_Request(const Region::Space & dst, const Public_Key & k)
109    : Control(DH_REQUEST, 0, 0, TSTP::_instance->now(), TSTP::_instance
       ->here(), TSTP::_instance->here()), _destination(dst), _public_key(k)
       { }
110
111    const Region::Space & destination() { return _destination; }
112    void destination(const Region::Space & d) { _destination = d; }
113
114    Public_Key key() { return _public_key; }
115    void key(const Public_Key & k) { _public_key = k; }
116
117    friend OStream & operator<<(OStream & db, const DH_Request & m) {
118        db << reinterpret_cast<const Control &>(m) << ",d=" << m.
       _destination << ",k=" << m._public_key;
119        return db;
120    }
121
122 private:
123    Region::Space _destination;
124    Public_Key _public_key;
125    CRC _crc;
126 };
127
128 // Diffie-Hellman Response Security Bootstrap Control Message
129 class DH_Response: public Control
130 {
131 public:
```

```
132     DH_Response(const Public_Key & k)
133     : Control(DH_RESPONSE, 0, 0, TSTP::_instance->now(), TSTP::_instance
        ->here(), TSTP::_instance->here()), _public_key(k) { }
134
135     Public_Key key() { return _public_key; }
136     void key(const Public_Key & k) { _public_key = k; }
137
138     friend OStream & operator<<(OStream & db, const DH_Response & m) {
139         db << reinterpret_cast<const Control &>(m) << ",k=" << m.
        _public_key;
140         return db;
141     }
142
143 private:
144     Public_Key _public_key;
145     CRC _crc;
146 };
147
148 // Authentication Request Security Bootstrap Control Message
149 class Auth_Request: public Control
150 {
151 public:
152     Auth_Request(const Node_Auth & a, const OTP & o)
153     : Control(AUTH_REQUEST, 0, 0, TSTP::_instance->now(), TSTP::
        _instance->here(), TSTP::_instance->here()), _auth(a), _otp(o) { }
154
155     const Node_Auth & auth() const { return _auth; }
156     void auth(const Node_Auth & a) { _auth = a; }
157
158     const OTP & otp() const { return _otp; }
159     void otp(const OTP & o) { _otp = o; }
160
161     friend OStream & operator<<(OStream & db, const Auth_Request & m) {
162         db << reinterpret_cast<const Control &>(m) << ",a=" << m._auth
        << ",o=" << m._otp;
163         return db;
164     }
165
166 private:
167     Node_Auth _auth;
168     OTP _otp;
169     CRC _crc;
170 };
171
172 // Authentication Granted Security Bootstrap Control Message
173 class Auth_Granted: public Control
174 {
175 public:
176     Auth_Granted(const Region::Space & dst, const Node_Auth & a)
177     : Control(AUTH_GRANTED, 0, 0, TSTP::_instance->now(), TSTP::
        _instance->here(), TSTP::_instance->here()), _destination(dst), _auth
        (a) { }
178
179     const Region::Space & destination() { return _destination; }
180     void destination(const Coordinates  & d) { _destination = d; }
181
182     const Node_Auth & auth() const { return _auth; }
183     void auth(const Node_Auth & a) { _auth = a; }
```

```
184
185     friend OStream & operator<<(OStream & db, const Auth_Granted & m) {
186         db << reinterpret_cast<const Control &>(m) << ",d=" << m.
    _destination << ",a=" << m._auth;
187         return db;
188     }
189
190 private:
191     Region::Space _destination;
192     Node_Auth _auth;
193     CRC _crc;
194 };
195
196 // Report Control Message
197 class Report: public Control
198 {
199 public:
200     Report(const Unit & unit, const Error & error = 0, bool
    epoch_request = false)
201     : Control(REPORT, 0, 0, TSTP::_instance->now(), TSTP::_instance->
    here(), TSTP::_instance->here()), _unit(unit), _error(error),
    _epoch_request(epoch_request) { }
202
203     const Unit & unit() const { return _unit; }
204     Error error() const { return _error; }
205     bool epoch_request() const { return _epoch_request; }
206
207     friend OStream & operator<<(OStream & db, const Report & r) {
208         db << reinterpret_cast<const Control &>(r) << ",u=" << r._unit
    << ",e=" << r._error;
209         return db;
210     }
211
212 private:
213     Unit _unit;
214     Error _error;
215     bool _epoch_request;
216     CRC _crc;
217 } __attribute__((packed));
218
219 // Epoch Control Message
220 class Epoch: public Control
221 {
222 public:
223     Epoch(const Region & dst, const Time & ep = TSTP::_instance->_epoch,
     const Global_Coordinates & coordinates = TSTP::_instance->
    _global_coordinates)
224     : Control(EPOCH, 0, 0, TSTP::_instance->now(), TSTP::_instance->here
    (), TSTP::_instance->here()), _destination(dst), _epoch(ep),
    _coordinates(coordinates) { }
225
226     const Region & destination() const { return _destination; }
227     const Time epoch() const { return _epoch; }
228     const Global_Coordinates & coordinates() const { return _coordinates
    ; }
229
230     friend OStream & operator<<(OStream & db, const Epoch & e) {
231         db << reinterpret_cast<const Control &>(e) << ",e=" << e._epoch
```

```
    << ",c=" << e._coordinates;
232         return db;
233     }
234
235 private:
236     Region _destination;
237     Time _epoch;
238     Global_Coordinates _coordinates;
239     CRC _crc;
240 } __attribute__((packed));
241
242 // TSTP Smart Data bindings
243 // Interested (binder between Interest messages and Smart Data)
244 class Interested: public Interest
245 {
246 public:
247     template<typename T>
248     Interested(T * data, const Region & region, const Unit & unit, const
        Mode & mode, const Precision & precision, const Microsecond & expiry
    , const Microsecond & period = 0)
249     : Interest(region, unit, mode, precision, expiry, period), _link(
    this, T::UNIT) {
250         db<TSTP>(TRC) << "TSTP::Interested(d=" << data << ",r=" <<
    region << ",p=" << period << ") => " << reinterpret_cast<const
    Interest &>(*this) << endl;
251         TSTP::_instance->_interested.insert(&_link);
252         advertise();
253     }
254     ~Interested() {
255         db<TSTP>(TRC) << "TSTP::~Interested(this=" << this << ")" <<
    endl;
256         TSTP::_instance->_interested.remove(&_link);
257         revoke();
258     }
259
260     void advertise() { send(); }
261     void revoke() { _mode = DELETE; send(); }
262
263     template<typename Value>
264     void command(const Value & v) {
265         db<TSTP>(TRC) << "TSTP::Interested::command(v=" << v << ")" <<
    endl;
266         Buffer * buf = TSTP::_instance->alloc(sizeof(Command));
267         Command * command = new (buf->frame()->data<Command>()) Command(
    unit(), region());
268         memcpy(command->command<Value>(), &v, sizeof(Value));
269         db<TSTP>(INF) << "TSTP::Interested::command:command=" << command
     << " => " << (*command) << endl;
270         TSTP::_instance->marshal_send(buf);
271     }
272
273 private:
274     void send() {
275         db<TSTP>(TRC) << "TSTP::Interested::send() => " <<
    reinterpret_cast<const Interest &>(*this) << endl;
276         Buffer * buf = TSTP::_instance->alloc(sizeof(Interest));
277         memcpy(buf->frame()->data<Interest>(), this, sizeof(Interest));
278         TSTP::_instance->marshal_send(buf);
```

```
279     }
280
281 private:
282     Interests::Element _link;
283 };
284
285 // Responsive (binder between Smart Data (Sensors) and Response messages)
286 class Responsive: public Response
287 {
288 public:
289     template<typename T>
290     Responsive(T * data, const Unit & unit, const Error & error, const Time & expiry, bool epoch_request = false)
291     : Response(unit, error, expiry), _size(sizeof(Response) - sizeof(Response::Data) + sizeof(typename T::Value)), _t0(0), _t1(0), _link(this, T::UNIT) {
292         db<TSTP>(TRC) << "TSTP::Responsive(d=" << data << ",s=" << _size << ") => " << this << endl;
293         db<TSTP>(INF) << "TSTP::Responsive() => " << reinterpret_cast<const Response &>(*this) << endl;
294         _responsives.insert(&_link);
295         advertise(epoch_request);
296     }
297     ~Responsive() {
298         _tstp->trace() << "TSTP::~Responsive(this=" << this << ")";
299         _tstp->_responsives.remove(&_link);
300     }
301
302     using Header::time;
303     using Header::origin;
304
305     void t0(const Time & t) { _t0 = t; }
306     void t1(const Time & t) { _t1 = t; }
307     Time t0() const { return _t0; }
308     Time t1() const { return _t1; }
309
310     void respond(const Time & expiry) { send(expiry); }
311     void advertise(bool epoch_request = false) {
312         _tstp->trace() << "TSTP::Responsive::advertise()";
313         Buffer * buf = _tstp->alloc(sizeof(Report));
314         new (buf->frame()->data<Report>()) Report(unit(), error(), epoch_request);
315         _tstp->marshal_send(buf);
316     }
317
318 private:
319     void send(const Time & expiry) {
320         if((_time >= _t0) && (_time <= _t1)) {
321             assert(expiry > _tstp->now());
322             _tstp->trace() << "TSTP::Responsive::send(x=" << expiry << ")";
323             Buffer * buf = _tstp->alloc(_size);
324             Response * response = buf->frame()->data<Response>();
325             memcpy(response, this, _size);
326             response->expiry(expiry);
327             _tstp->trace() << "TSTP::Responsive::send:response=" << response << " => " << (*response);
```

```
328             _tstp->marshal_send(buf);
329         }
330     }
331
332 private:
333     TSTP * _tstp;
334     unsigned int _size;
335     Time _t0;
336     Time _t1;
337     Responsives::Element _link;
338 };
```

Listing B.25 – Enclave/TSTP/EPOS/libc_unimplemented.h

```
1  #ifndef LIBC_UNIMPLEMENTED_H
2  #define LIBC_UNIMPLEMENTED_H
3
4  #include <endian.h>
5  #include <stdint.h>
6
7  extern "C"
8  {
9
10 uint32_t htole32(uint32_t v);
11 uint16_t htole16(uint16_t v);
12 uint32_t le32toh(uint32_t v);
13 uint16_t le16toh(uint16_t v);
14 uint32_t htobe32(uint32_t v);
15 uint16_t htobe16(uint16_t v);
16 uint32_t be32toh(uint32_t v);
17 uint16_t be16toh(uint16_t v);
18
19 }
20
21 #endif
```

Listing B.26 – Enclave/TSTP/EPOS/libc_unimplemented.cpp

```
1
2
3  #include <libc_unimplemented.h>
4  #include <stdlib.h>
5
6  static uint32_t byteswap16(uint32_t v) {
7      return __builtin_bswap16(v);
8  }
9
10 static uint32_t byteswap32(uint32_t v) {
11     return __builtin_bswap32(v);
12 }
13
14 uint32_t htole32(uint32_t v) {
15 #if BYTE_ORDER == LITTLE_ENDIAN
16     return v;
17 #elif BYTE_ORDER == BIG_ENDIAN
18     return byteswap32(v);
19 #endif
20 }
21
```

```
22 uint16_t htole16(uint16_t v) {
23 #if BYTE_ORDER == LITTLE_ENDIAN
24     return v;
25 #elif BYTE_ORDER == BIG_ENDIAN
26     return byteswap16(v);
27 #endif
28 }
29
30 uint32_t le32toh(uint32_t v) {
31 #if BYTE_ORDER == LITTLE_ENDIAN
32     return v;
33 #elif BYTE_ORDER == BIG_ENDIAN
34     return byteswap32(v);
35 #endif
36 }
37
38 uint16_t le16toh(uint16_t v) {
39 #if BYTE_ORDER == LITTLE_ENDIAN
40     return v;
41 #elif BYTE_ORDER == BIG_ENDIAN
42     return byteswap16(v);
43 #endif
44 }
45
46 uint32_t htobe32(uint32_t v) {
47 #if BYTE_ORDER == LITTLE_ENDIAN
48     return byteswap32(v);
49 #elif BYTE_ORDER == BIG_ENDIAN
50     return v;
51 #endif
52 }
53
54 uint16_t htobe16(uint16_t v) {
55 #if BYTE_ORDER == LITTLE_ENDIAN
56     return byteswap16(v);
57 #elif BYTE_ORDER == BIG_ENDIAN
58     return v;
59 #endif
60 }
61
62 uint32_t be32toh(uint32_t v) {
63 #if BYTE_ORDER == LITTLE_ENDIAN
64     return byteswap32(v);
65 #elif BYTE_ORDER == BIG_ENDIAN
66     return v;
67 #endif
68 }
69
70 uint16_t be16toh(uint16_t v) {
71 #if BYTE_ORDER == LITTLE_ENDIAN
72     return byteswap16(v);
73 #elif BYTE_ORDER == BIG_ENDIAN
74     return v;
75 #endif
76 }
```

Listing B.27 – Enclave/TSTP/EPOS/transducer.h

```
1 // EPOS ARM Cortex Smart Transducer Declarations
```

```
2
3 #ifndef __cortex_transducer_h
4 #define __cortex_transducer_h
5
6 #include <smart_data.h>
7
8 __BEGIN_SYS
9
10 enum CUSTOM_UNITS
11 {
12     UNIT_RFID = TSTP::Unit::DIGITAL | 1,
13     UNIT_SWITCH = TSTP::Unit::DIGITAL | 2,
14     UNIT_TEST = TSTP::Unit::DIGITAL | 3,
15 };
16
17 class Temperature_Sensor
18 {
19 public:
20     static const unsigned int UNIT = TSTP::Unit::Temperature;
21     static const unsigned int NUM = TSTP::Unit::I32;
22     static const int ERROR = 0; // Unknown
23
24     static const bool INTERRUPT = false;
25     static const bool POLLING = true;
26
27     typedef _UTIL::Observer Observer;
28     typedef _UTIL::Observed Observed;
29     typedef Smart_Data<Temperature_Sensor>::Value Value;
30
31     static const unsigned int SIZE = 100;
32
33 public:
34     Temperature_Sensor() { }
35
36     static void sense(unsigned int dev, Smart_Data<Temperature_Sensor> *
      data) {
37         if(!_serie) bootstrap();
38         if(_index < SIZE)
39             data->_value = _serie[_index++];
40         else
41             data->_value = 0;
42     }
43
44     static void actuate(unsigned int dev, Smart_Data<Temperature_Sensor>
      * data, const Smart_Data<Temperature_Sensor>::Value & command) {}
45
46     static void attach(Observer * obs) { _observed.attach(obs); }
47     static void detach(Observer * obs) { _observed.detach(obs); }
48
49     static void bootstrap() {
50         _serie = new Value[SIZE];
51         for(unsigned int i = 0 ; i < SIZE; i++){
52             for(unsigned int j = 0; i < SIZE && j < SIZE/4; i++, j++)
53                 _serie[i] = i*2+00;
54             for(unsigned int j = 0; i < SIZE && j < SIZE/4; i++, j++)
55                 _serie[i] = i*3+00;
56             for(unsigned int j = 0; i < SIZE && j < SIZE/4; i++, j++)
57                 _serie[i] = i*4+00;
```

```
58              for(unsigned int j = 0; i < SIZE && j < SIZE/4; i++, j++)
59                  _serie[i] = i*10+00;
60          }
61          _index = 0;
62      }
63
64  private:
65      static bool notify() { return _observed.notify(); }
66
67      static void init();
68
69  private:
70      static Observed _observed;
71      static Value * _serie;
72      static unsigned int _index;
73  };
74
75  typedef Smart_Data<Temperature_Sensor> Temperature;
76
77  __END_SYS
78  #endif
```

Listing B.28 – Enclave/TSTP/EPOS/transducer.h

```
1  // EPOS ARM Cortex Smart Transducer Declarations
2
3  #ifndef __cortex_transducer_h
4  #define __cortex_transducer_h
5
6  #include <smart_data.h>
7
8  __BEGIN_SYS
9
10 enum CUSTOM_UNITS
11 {
12     UNIT_RFID = TSTP::Unit::DIGITAL | 1,
13     UNIT_SWITCH = TSTP::Unit::DIGITAL | 2,
14     UNIT_TEST = TSTP::Unit::DIGITAL | 3,
15 };
16
17 class Temperature_Sensor
18 {
19 public:
20     static const unsigned int UNIT = TSTP::Unit::Temperature;
21     static const unsigned int NUM = TSTP::Unit::I32;
22     static const int ERROR = 0; // Unknown
23
24     static const bool INTERRUPT = false;
25     static const bool POLLING = true;
26
27     typedef _UTIL::Observer Observer;
28     typedef _UTIL::Observed Observed;
29     typedef Smart_Data<Temperature_Sensor>::Value Value;
30
31     static const unsigned int SIZE = 100;
32
33 public:
34     Temperature_Sensor() { }
35
```

```
36      static void sense(unsigned int dev, Smart_Data<Temperature_Sensor> *
        data) {
37          if(!_serie) bootstrap();
38          if(_index < SIZE)
39              data->_value = _serie[_index++];
40          else
41              data->_value = 0;
42      }
43
44      static void actuate(unsigned int dev, Smart_Data<Temperature_Sensor>
        * data, const Smart_Data<Temperature_Sensor>::Value & command) {}
45
46      static void attach(Observer * obs) { _observed.attach(obs); }
47      static void detach(Observer * obs) { _observed.detach(obs); }
48
49      static void bootstrap() {
50          _serie = new Value[SIZE];
51          for(unsigned int i = 0 ; i < SIZE; i++){
52              for(unsigned int j = 0; i < SIZE && j < SIZE/4; i++, j++)
53                  _serie[i] = i*2+00;
54              for(unsigned int j = 0; i < SIZE && j < SIZE/4; i++, j++)
55                  _serie[i] = i*3+00;
56              for(unsigned int j = 0; i < SIZE && j < SIZE/4; i++, j++)
57                  _serie[i] = i*4+00;
58              for(unsigned int j = 0; i < SIZE && j < SIZE/4; i++, j++)
59                  _serie[i] = i*10+00;
60          }
61          _index = 0;
62      }
63
64  private:
65      static bool notify() { return _observed.notify(); }
66
67      static void init();
68
69  private:
70      static Observed _observed;
71      static Value * _serie;
72      static unsigned int _index;
73  };
74
75  typedef Smart_Data<Temperature_Sensor> Temperature;
76
77  __END_SYS
78  #endif
```