UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Davi Resner

ESTABELECIMENTO DE CHAVES E COMUNICAÇÃO SEGURA PARA A INTERNET DAS COISAS

Florianópolis

2014

Davi Resner

ESTABELECIMENTO DE CHAVES E COMUNICAÇÃO SEGURA PARA A INTERNET DAS COISAS

Trabalho de Conclusão de Curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação.

Florianópolis

2014

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

Resner, Davi
Estabelecimento de chaves e comunicação segura para a Internet das Coisas / Davi Resner ; orientador, Antônio Augusto Fröhlich ; coorientador, Jean Martina. - Florianópolis, SC, 2014.
67 p.

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

Inclui referências

1. Ciências da Computação. 2. Internet das Coisas. 3. Redes de Sensores Sem Fio. 4. Segurança. 5. Sistemas Embarcados. I. Fröhlich, Antônio Augusto. II. Martina, Jean. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

Davi Resner

ESTABELECIMENTO DE CHAVES E COMUNICAÇÃO SEGURA PARA A INTERNET DAS COISAS

Trabalho de Conclusão de Curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação

Florianópolis, 24 de julho 2014.

Prof. Dr. Antônio Augusto Fröhlich
Orientador

Banca Examinadora:

Prof. Dr. Jean Martina
Coorientador

Bsc. Peterson Clayton de Oliveira

Msc. Rodrigo Vieira Steiner

A todos que atuaram em minha formação.
Familiares,
Amigos e amigas,
Professores e professoras,
Escritores e escritoras,
Músicos e musicistas.

"But look at it this way. What really is the point of trying to teach anything to anybody? (...) What I mean is that if you really want to understand something, the best way is to try and explain it to someone else. That forces you to sort it out in your own mind. And the more slow and dim-witted your pupil, the more you have to break things down into more and more simple ideas. And that's really the essence of programming. By the time you've sorted out a complicated idea into little steps that even a stupid machine can deal with, you've certainly learned something about it yourself. The teacher usually learns more than the pupil. Isn't that true?"

Douglas Adams

(Dirk Gently's Holistic Detective Agency, 1987)

RESUMO

Este trabalho descreve um protocolo que provê uma solução prática para o problema de estabelecimento de chaves criptográficas e comunicação segura no contexto de Redes de Sensores Sem Fio, no qual eficiência computacional é um requisito imprescindível. Tal protocolo foi implementado no sistema operacional EPOS na forma de uma camada de rede que provê os princípios de integridade, autenticidade e confidencialidade da informação. Os testes foram realizados na plataforma EPOSMoteII e a análise dos resultados indica que a implementação é adequada para o cenário de sistemas embarcados de baixo poder de processamento.

Palavras-chave: Internet das Coisas, Redes de Sensores sem Fio, Segurança, Sistemas Embarcados

ABSTRACT

This work describes a protocol that provides a practical solution for the problem of cryptographic key establishment and secure communication in the context of Wireless Sensor Networks, in which computational efficiency is a fundamental requirement. Such protocol was implemented in the EPOS operating system and takes the form of a network layer that provides the principles of data integrity, authenticity and confidentiality. Tests were executed on the EPOSMoteII platform and the analysis of the results shows that the implementation is adequate to be used in the scenario of embedded systems with low proccessing power.

Keywords: Internet of Things, Wireless Sensor Networks, Security, Embedded Systems

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

A ideia de Internet das Coisas (IdC) consiste em um interfaceamento de dispositivos da vida cotidiana com a internet. Significa geladeiras, televisões, ventiladores, lâmpadas e carros recebendo e enviando mensagens para o mundo. Pela grande quantidade com a qual serão empregados, os dispositivos utilizados para possibilitar tal comportamento devem ser de baixo custo e energeticamente eficientes; consequentemente, não terão o maior poder de processamento disponível no mercado. Uma rede de sensores sem fio (RSSF) é bastante similar a certos cenários de IdC: é uma rede composta por vários dispositivos capazes de coletar informações do ambiente e de comunicar-se sem o uso de fios.

Muitos protocolos comumente utilizados na internet tradicional (e.g. TCP/IP) funcionam bem para cenários em que a necessidade de eficiência de processamento e energia não seja tão crítica, mas no mundo embarcado da IdC e das RSSF muito pode ser feito para melhorar o desempenho (ELKHODR; SHAHRESTANI; CHEUNG, 2013).

Um requisito fundamental para um protocolo de comunicação na IdC é segurança. Em muitos cenários é importante que mensagens possam ser transmitidas de forma íntegra, confidencial e autêntica (SUO et al., 2012). Ninguém gostaria, por exemplo, de ter sua casa controlada por estranhos sem permissão – falha de autenticidade –, ou que uma usina nuclear disparasse seus alarmes de catástrofe porque algumas mensagens com as informações reais reportadas pelos sensores foram manipuladas – falha de integridade. Há ainda aspectos de confidencialidade: mensagens privadas destinadas a um dispositivo devem apenas poder ser entendidas por aquele dispositivo.

É importante reconhecer que as soluções da internet convencional para tais problemas de segurança muitas vezes não são adequadas na Internet das Coisas, tornando assim esse um tema aberto de pesquisa (ELKHODR; SHAHRESTANI; CHEUNG, 2013). Além da constante restrição em termos de desempenho, podemos imaginar cenários em que algumas *coisas*, por exemplo, enviarão mensagens muito raramente e então se tornarão incomunicáveis por um longo período de tempo. É preciso saber o quanto se pode confiar nessa mensagem sem uma subsequente "conversa" entre os dispositivos (FROHLICH et al., 2013).

Com estes problemas em mente, um novo protocolo para estabelecimento de chaves criptográficas é proposto, especialmente criado para Redes de Sensores Sem Fio.

O restante deste texto está organizado da seguinte forma: a próxima seção apresenta os objetivos que este trabalho espera alcançar. O capítulo 2 apresenta todos os conceitos e algoritmos que foram utilizados para a elaboração deste trabalho, sendo que alguns conceitos mais matemáticos encontram-se no Apêndice A.1. O capítulo 3 apresenta alguns trabalhos com uma proposta similar ao deste, ressaltando como este difere daqueles. O protocolo de estabelecimento de chaves e comunicação segura é apresentado no capítulo 4 e analisado no 5. O capítulo 6 traz as considerações finais e apresenta alguns tópicos sugeridos como trabalhos futuros.

1.1 OBJETIVOS

Implementação de um protocolo que permita segurança da comunicação em uma rede de sensores sem fio de modo eficiente. A implementação inicialmente se limitará a um cenário no qual sensores se comunicam apenas com *gateways*.

1.1.1 Objetivo Geral

Implementação de um protocolo que permita que sensores em uma rede de sensores sem fio se comuniquem com um *gateway* com os princípios de integridade, autenticidade e confidencialidade

disponíveis. O protocolo deve ser computacionalmente viável em tal contexto, o que significa que deve apresentar baixo consumo de processamento, memória e energia.

1.1.2 Objetivos Específicos

- Descrever e implementar o protocolo de estabelecimento de chaves criptográficas;
- Argumentar através de análise qualitativa de todo o módulo de segurança que o protocolo é suficientemente seguro;
- Mostrar através de análise quantitativa do tempo de processamento, consumo energético e de memória que o protocolo é computacionalmente viável para o contexto.

2 FUNDAMENTAÇÃO

Neste capítulo são apresentados os principais conceitos e algoritmos utilizados durante a elaboração deste trabalho.

2.1 INTERNET DAS COISAS

O termo Internet das Coisas (IdC) apareceu por volta de 1999 e foi originalmente uma visão na qual todos os objetos físicos estariam marcados e unicamente identificados por meio de transponders ou leitores RFID (ELKHODR; SHAHRESTANI; CHEUNG, 2013).

A definição atual de IdC é nebulosa. Neste trabalho, os termos Internet das Coisas e Rede de Sensores Sem Fio serão usados de forma intercambiável e denotarão um cenário que consiste de sensores e atuadores equipados de microcontroladores capazes de se comunicar com um *gateway*, o qual se encarrega de prover uma interface entre esses nodos e uma rede externa maior (e.g. a internet), permitindo assim o acesso a tais nodos por meio de qualquer dispositivo com uma conexão a esta rede externa.

Uma vasta gama de possíveis aplicações deste conceito é apresentada em (ATZORI; IERA; MORABITO, 2010), desde auxílio em logística até táxis robóticos. Um exemplo bastante ilustrativo é o de *smart home*, ou casa inteligente. Já é possível acoplar microcontroladores capazes de se comunicar por rádio a vários dispositivos domésticos comuns, como lâmpadas, condicionadores de ar e tomadas. Outros sensores (e.g. de luminosidade, movimento e CO2) podem ser implementados no ambiente e toda informação relevante captada por eles enviada para um servidor (ou *gateway*), que pode ser um computador comum dentro da própria residência rodando *software* específico. O *gateway* então, com base nas informações dos sensores e preferências do usuário, envia comandos para os atuadores, apagando automaticamente as lâmpadas quando ninguém for detectado numa sala, por exemplo. O *gateway* também pode prover uma interface com a internet, permitindo, por exemplo, que o usuário ligue o ar-condicionado de seu quarto 10 minutos antes de chegar em casa através de um *smartphone*, encontrando assim uma temperatura agradável logo que chegar.

2.2 SENSORES

A implementação deste trabalho foi feita no módulo para redes de sensores sem fio chamado EPOSMoteII (LISHA, 2014), baseado na Platform-in-Package MC13224V (FREESCALE, 2012) da Freescale. O EPOSMote faz o papel de microcontrolador capaz de comunicação, citado anteriormente, que pode ser acoplado a sensores e atuadores diversos para controlá-los, ou ainda atuar como um nodo *stand-alone* da rede. Suas configurações ilustram bem o tipo de limitação que se encontra em RSSF: processador ARM7 de 24Mhz, 128Kbytes de memória flash, 80Kbytes de memória ROM e 96Kbytes de memória RAM. Este módulo também conta com um rádio embutido conformante com a norma IEEE 802.15.4. O sistema operacional utilizado é o EPOS (LISHA, 2014), e a linguagem, C++.

2.3 SEGURANÇA DE INFORMAÇÃO

Aplicações como *smart home* já são tecnicamente possíveis, mas sua ubiquidade ainda depende da solução de uma série de desafios técnicos e sociais. Segurança de comunicação é evidentemente necessária nesse contexto e é reconhecido como um assunto de pesquisa em aberto (ATZORI; IERA;

MORABITO, 2010), assim como o tema mais específico de estabelecimento e gerenciamento de chaves criptográficas (RIAHI et al., 2013). As soluções de segurança empregadas na internet convencional muitas vezes não servem para RSSF, pois sensores apresentam poder de processamento muito inferior a qualquer computador *desktop* comum, bem como uma grande necessidade de eficiência energética (um sensor pode ficar meses dispondo apenas de uma pequena bateria 9V comum). Novas soluções devem ser encontradas com estas limitações em mente, e esta é a proposta deste trabalho.

Os princípios de segurança da informação que o protocolo proposto visa garantir a nodos de uma RSSF são:

- **Autenticidade:** A garantia de que a mensagem é genuína e foi de fato enviada por quem diz ser seu remetente.
- **Confidencialidade:** A garantia de que apenas nodos autorizados tem acesso ao conteúdo de uma mensagem.
- **Integridade:** A garantia de que uma mensagem não foi modificada de forma não autorizada e não detectável.

Várias técnicas e algoritmos criptográficos foram postos para trabalhar em conjunto para atingir-se estas garantias quando necessário de maneira eficiente. O restante deste capítulo é dedicado à exposição dos principais conceitos e algoritmos envolvidos.

2.4 NOTAÇÃO

No decorrer deste trabalho, a notação chamada por vários autores de notação padrão para protocolos de segurança será usada (similar à usada em (STEINER; NEUMAN; SCHILLER, 1988) e (CARLSEN, 1994)). Em nossos cenários, Arthur (A) e Beeblebrox (B) serão dois participantes que seguem os protocolos honestamente. Zaphod (Z) aparecerá como a personagem tradicionalmente conhecida como *adversário*. Ela tenta sabotar o protocolo e a segurança da comunicação entre Arthur e Beeblebrox.

As seguintes convenções serão usadas:

K_A - Uma chave criptográfica privada, conhecida por A.

P_A - Chave criptográfica pública de A.

K_{AB} - Uma chave criptográfica compartilhada por A e B.

T_i - A $timestamp$ da rede no momento i.

N - Um *nonce*.

$\{M\}_{K_A}$ - Mensagem M cifrada com a chave K_A.

$A \rightarrow B : M$ - A envia a mensagem M para B.

No final de algumas seções aparecerá uma notação mais gráfica e intuitiva, conforme a Figura 1, que denota o seguinte cenário: Arthur calcula $a = 1 + 1$ e envia o resultado para Beeblebrox, que então calcula $b = a + 2$ e envia o resultado para Arthur.

2.5 CIFRADORES

Algoritmos criptográficos podem ser divididos em dois grandes tipos: simétricos e assimétricos. Cifradores simétricos utilizam-se de uma única chave, conhecida por ambos remetente e destinatário da mensagem, para cifrar e decifrar uma mensagem. Já no caso dos algoritmos assimétricos, cada

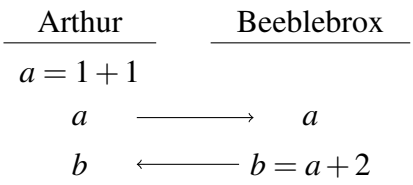

Figura 1 – Exemplo de notação de troca de mensagens

nodo tem um par de chaves, sendo uma pública e uma privada. Mensagens cifradas com a chave pública podem ser decifradas apenas com a privada correspondente e vice-versa. Ilustrar-se-á o funcionamento de cada tipo com uma explicação de dois algoritmos característicos: o Cifrador de César para criptografia simétrica e RSA para assimétrica.

2.5.1 Cifrador de César

Este foi o primeiro cifrador de substituições utilizado de que se tem notícia, e também o mais simples (STALLINGS, 2011), creditado a Júlio César. Consiste basicamente em, para cada letra da mensagem a ser enviada, trocá-la pela letra que fica três lugares adiante no alfabeto. Exemplificando com nosso alfabeto atual (e ignorando a apóstrofe):

```
Mensagem:  DON'T PANIC
Cifrada:   GRQ'W SDQLF
```

Ao receber a mensagem cifrada, o destinatário ciente do cifrador utilizado simplesmente troca cada letra pela letra três lugares atrás no alfabeto, recuperando a mensagem original.

Agora, imagine que as letras podem ser deslocadas não necessariamente 3 posições, mas qualquer número k de posições. Neste caso, não é suficiente que o destinatário saiba qual o processo de cifragem utilizado: para recuperar a mensagem original, deve-se conhecer também o valor de k.

Se atribuirmos um valor numérico a cada letra do alfabeto da seguinte forma: $A = 0, B = 1, \ldots, Z = 25$, com k sendo a chave, l_i sendo a i-ésima letra da mensagem orignal e c_i a i-ésima letra da mensagem cifrada, o processo de cifragem pode ser implementado segundo a equação 2.1

$$c_i = (l_i + k) \bmod 26 \tag{2.1}$$

e, de maneira análoga, o processo de decifragem pela equação 2.2

$$l_i = (c_i - k) \bmod 26 \tag{2.2}$$

Este valor k é a *chave criptográfica*. Note que o mesmo valor é utilizado tanto no processo de cifragem quanto no de decifragem. Assim sendo, k é classificada como *chave simétrica*, e o Cifrador de César é um *cifrador simétrico*. Este cifrador é extremamente frágil, porém por sua simplicidade é muito eficiente em termos de processamento. Essa eficiência é em geral característica de cifradores simétricos e existem cifradores cuja análise, intuição e o tempo de uso indicam um alto nível de segurança, como o AES.

Apesar de sua eficiência ser um grande atrativo para RSSF, um dos maiores desafios do uso de cifradores simétricos é a questão de como fazer cada parte conhecer o valor da chave e ao mesmo tempo mantê-la segura de usuários não autorizados, ou em outras palavras, como realizar o estabelecimento e gerenciamento de chaves. Outros algoritmos, expostos adiante, são utilizados neste trabalho para solucionar tal problema.

2.5.2 RSA

RSA é um cifrador que trata a entrada como um conjunto de blocos. Escolhe-se um valor n e um valor i tal que $2^i < n \leq 2^{i+1}$. Então divide-se a mensagem em blocos de tamanho i e o bloco é interpretado como um número binário M. Arthur possui um par de chaves (K_A, P_A), escolhidas de modo que, para cada bloco x possível, $x^{K_A P_A} \bmod n = x$ (como encontrar tais valores não vem ao caso, mas não é um processo complicado). Os valores n, i e P_A são públicos, e apenas Arthur conhece o valor de K_A.

Para enviar uma mensagem que apenas Arthur consiga decifrar, Beeblebrox busca sua chave P_A e calcula o valor cifrado $\{M\}_{P_A}$ segundo a equação 2.3

$$\{M\}_{P_A} = M^{P_A} \bmod n \tag{2.3}$$

Ao receber o bloco $\{M\}_{P_A}$, Arthur o decifra usando sua chave secreta K_A:

$$M = \{M\}_{P_A}^{K_A} \bmod n = (M^{P_A})^{K_A} \bmod n = M^{P_A K_A} \bmod n \tag{2.4}$$

Note que a chave utilizada para a cifragem é diferente da chave utilizada na decifragem, ou seja, RSA é um cifrador *assimétrico*.

O principal ponto a se notar é que as operações de exponenciação modular utilizadas são ordens de grandeza mais complexas computacionalmente do que as substituições do exemplo anterior, principalmente para valores práticos (grandes) de chaves. Cifradores assimétricos em geral apresentam esta desvantagem de eficiência se comparados a cifradores simétricos; o custo para cifragem assimétrica de cada mensagem numa RSSF seria proibitivo num caso geral. Porém, esquemas como o RSA são mais versáteis que, por exemplo, o AES (simétrico), além de que o problema de distribuição de chaves é muito mais simples, já que algumas chaves são públicas e outras são conhecidas apenas por seu único usuário, não havendo necessidade de compartilhamento de chaves secretas. RSA é versátil por poder ser usado também para autenticação de mensagens, bastando utilizar a chave secreta para cifragem e a pública correspondente para decifragem. Porém, existem técnicas de autenticação que utilizam-se de cifradores simétricos, sendo assim mais atrativas para este contexto; logo, criptografia assimétrica será evitada ao máximo, aparecendo apenas na fase inicial de comunicação.

2.5.3 AES

Advanced Encryption Standard (NIST, 2001) é uma especificação para cifragem de dados eletrônicos amplamente utilizada, baseada no cifrador simétrico Rijndael (DAEMEN; RIJMEN, 2003).

Por ser um algoritmo computacionalmente eficiente e requerer pouca memória, AES tem se mostrado propício para utilização no cenário de IdC. Além disso o EPOSMoteII provê uma implementação em *hardware* do algoritmo (FREESCALE, 2012), tornando-o extremamente eficiente. (FROHLICH et al., 2013) apresenta resultados de testes mostrando altíssimo desempenho do AES no EPOSMoteII.

Por esses motivos o AES foi escolhido como cifrador de mensagens e procurou-se utilizá-lo o máximo possível, não só no processo de cifragem, mas também como peça importante no protocolo de estabelecimento de chaves.

Seu funcionamento interno não é importante para os fins deste trabalho; ele será visto como uma caixa preta. Tudo que precisa-se saber é que ele é um cifrador simétrico eficiente e confiável (i.e. é muito difícil um atacante recuperar a chave ou mensagem original conhecendo apenas a mensagem cifrada) que toma como entrada um bloco de mensagem com 16 bytes de tamanho, uma chave de 16 bytes e retorna a mensagem cifrada, de tamanho 16 bytes. A especificação do algoritmo permite tamanhos diferentes de 16 bytes, mas a implementação em *hardware* disponível trabalha apenas com

este tamanho.

2.6 DIFFIE-HELLMAN

O consagrado algoritmo Diffie-Hellman (DIFFIE; HELLMAN, 1976) permite que dois participantes em um canal inseguro utilizem criptografia assimétrica para gerar uma chave criptográfica simétrica conhecida apenas por eles. Este algoritmo é utilizado como peça fundamental no estabelecimento de chaves, pois permite que nodos sem nenhum conhecimento prévio um do outro gerem chaves simétricas, eliminando assim a necessidade de carregar previamente chaves compartilhadas. Como é comum para os cifradores assimétricos, este é um algoritmo custoso, mas só é utilizado uma vez, no início da comunicação, permitindo que criptografia simétrica seja usada para as demais mensagens, portanto seu custo é não-proibitivo e justificado pelo benefício proporcionado. Efetivamente ganha-se a facilidade da distribuição de chaves assimétricas (ao invés de simétricas) pelo custo de uma operação assimétrica por chave gerada.

Na versão tradicional do algoritmo, dois parâmetros globais devem ser definidos na rede: um número primo p que define o corpo finito F_p (Apêndice A.1) em que as operações serão feitas e um gerador de F_p, $\alpha < p$ (um valor comum é $\alpha = 2$). Por definir a ordem do corpo, p define o tamanho das chaves geradas.

Imagine que Arthur e Beeblebrox conhecem os parâmetros globais e o algoritmo e querem estabelecer uma chave simétrica conhecida apenas por eles. Arthur escolhe um número aleatório $K_A < p$ que será sua chave privada e calcula sua chave pública $P_A = \alpha^{K_A} \bmod p$. Similarmente, Beeblebrox calcula K_B e P_B. Ambos enviam seus valores P para o outro e então Arthur pode calcular a chave

$$K_{AB} = (P_B)^{K_A} \bmod p = (\alpha^{K_B})^{K_A} \bmod p \tag{2.5}$$

E Beeblebrox

$$K_{AB} = (P_A)^{K_B} \bmod p = (\alpha^{K_A})^{K_B} \bmod p \tag{2.6}$$

Chegando assim ambos a um mesmo valor K_{AB}, revelando muito pouca informação sobre sua chave secreta. O processo é ilustrado na Figura 2.

A única informação que um observador externo tem sobre a chave K_A é o valor $P_A = \alpha^{K_A} \bmod p$. A segurança do Diffie-Hellman se baseia no problema do *logaritmo discreto*, abordado na seção 2.6.2.

Arthur		Beeblebrox
Escolhe $K_A < p$		Escolhe $K_B < p$
$P_A = \alpha^{K_A} \bmod p$	$\longrightarrow$	P_A
P_B	$\longleftarrow$	$P_B = \alpha^{K_B} \bmod p$
$K_{AB} = (P_B)^{K_A} \bmod p$		$K_{AB} = (P_A)^{K_B} \bmod p$

Figura 2 – Exemplo de transação Diffie-Hellman.

2.6.1 Man-in-the-middle

Um grande problema do Diffie-Hellman é sua vulnerabilidade a um ataque conhecido como *Man-in-the-middle* ou *Masquerade*. Retomemos nosso exemplo anterior, mas com um novo personagem, Zaphod. Zaphod é um agente governamental encarregado de espionar comunicações alheias, e ele sabe que nossos dois herois estão realizando o algoritmo Diffie-Hellman. Assumimos que Zaphod pode interceptar mensagens, ou seja, "retirar" mensagens quaisquer da rede antes que seu destinatário a receba. Zaphod pode espionar a comunicação seguindo os seguintes passos (STALLINGS, 2011), também ilustrados na Figura 3:

1. Zaphod calcula os valores K_{Z1}, K_{Z2} e os correspondentes P_{Z1} e P_{Z2}, como explicado anteriormente,
2. Quando Arthur enviar P_A para Beeblebrox, Zaphod intercepta a mensagem e no lugar dela transmite P_{Z1} para Beeblebrox,
3. Zaphod calcula $K_{AZ} = (P_A)^{K_{Z2}} \bmod p$,
4. Beeblebrox recebe P_{Z1}, calcula $K_{BZ} = (P_{Z1})^{K_B} \bmod p$ e envia P_B para Arthur,
5. Zaphod intercepta a mensagem e no lugar dela transmite P_{Z2} para Arthur,
6. Zaphod calcula $K_{BZ} = (P_B)^{K_{Z1}} \bmod p$,
7. Arthur recebe P_{Z2} e calcula $K_{AZ} = (P_{Z2})^{K_A} \bmod p$.

Agora, Arthur e Beeblebrox acreditam que compartilham uma chave privada secreta, mas na verdade Arthur e Zaphod compartilham K_{AZ} e Zaphod e Beeblebrox compartilham K_{BZ}. As próximas mensagens podem ser comprometidas da seguinte forma:

1. Arthur envia $\{M\}_{K_{AZ}}$ para Beeblebrox,
2. Zaphod intercepta a mensagem e a decifra,
3. Zaphod envia a Beeblebrox a mensagem $\{M\}_{K_{BZ}}$ ou $\{M'\}_{K_{BZ}}$. No primeiro caso Zaphod apenas espiona a comunicação tida por secreta; no segundo, ele modifica as mensagens como quiser.

Arthur e Beeblebrox conversam normalmente, sem fazer ideia de que sua comunicação está sendo monitorada ou até modificada. Este tipo de ataque é possível porque o Diffie-Hellman não provê os princípios de autenticação e integridade. Para eliminar a vulnerabilidade a este tipo de ataque, o protocolo deste trabalho introduz estes princípios utilizando outros algoritmos em conjunto com o Diffie-Hellman.

2.6.2 O problema do logaritmo discreto

Logaritmo discreto é a operação inversa da exponenciação modular mostrada em 2.6. Com o conhecimento de F_p, p, α, P_A definidos anteriormente, o logaritmo discreto de P_A é definido como:

$$x = \log_\alpha P_A \bmod p = \log_\alpha \alpha^{K_A} \bmod p = K_A \tag{2.7}$$

Uma operação de exponenciação modular pode ser efetuada com $O(\log K_A)$ operações sobre F_p. Logaritmos discretos, por sua vez, comprovadamente tem complexidade temporal $O(\sqrt{p})$ para o caso geral, no qual nenhum tipo de estrutura interna possa ser encontrada no corpo finito (STUDHOLME,

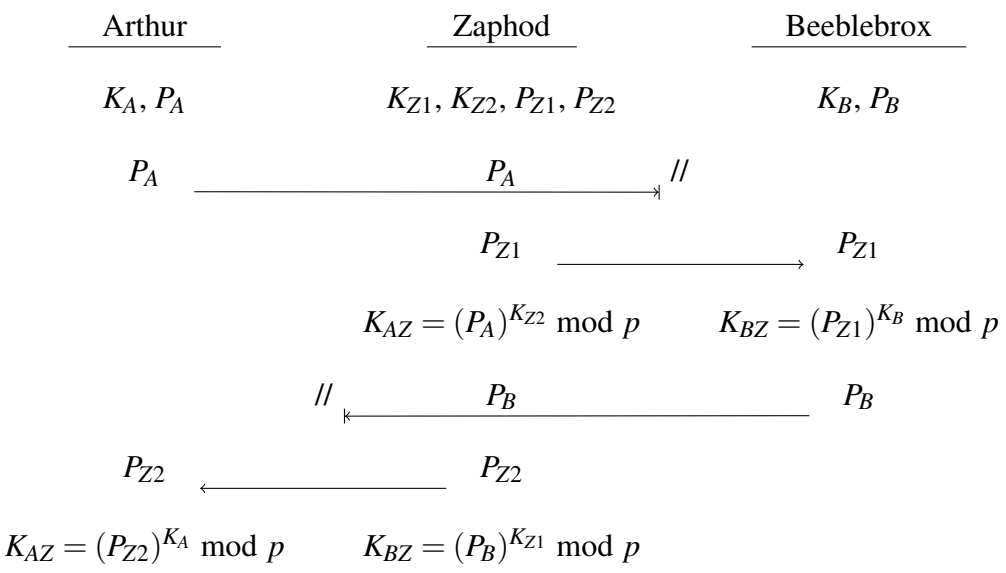

Figura 3 – Exemplo de ataque *man-in-the-middle*.

2013), logo, são muito mais difíceis de serem calculados. Operações cuja inversa é muito mais custosa que a própria são interessantes para criptografia, pois significam que é relativamente fácil usar uma chave para produzir uma mensagem cifrada, mas é muito difícil recuperar a chave a partir da mensagem cifrada.

A segurança do algoritmo de Diffie-Hellman baseia-se na dificuldade do problema do logaritmo discreto através da *Decision Diffie-Hellman Assumption* (DDH), uma conjectura que diz que as triplas $(\alpha^a, \alpha^b, \alpha^{ab})$ e $(\alpha^a, \alpha^b, \alpha^c)$, sendo a, b, c elementos aleatórios do F_p em questão, são computacionalmente indistinguíveis. Caso seja eficiente calcular o logaritmo discreto, DDH é trivialmente falsa, pois basta calcular $a = \log_\alpha \alpha^a \bmod p$ e checar se $(\alpha^b)^a = x$, sendo x o terceiro elemento da tripla. Caso positivo, $x = \alpha^{ab}$ e caso negativo $x \neq \alpha^{ab}$. Surpreendentemente, não foi encontrado um corpo para o qual se conheça um método mais eficiente do que este para o problema da DDH (BONEH, 1998), então acredita-se que o algoritmo de Diffie-Hellman é tão seguro quanto o problema do logaritmo discreto é computacionalmente difícil.

Não obstante, nem todos os corpos finitos atendem à DDH. Existem métodos capazes de solucionar o problema do logaritmo discreto de maneira rápida caso o corpo envolvido atenda a algumas suposições. (PADMAVATHY; BHAGVATI, 2009) apresenta um método capaz de encontrar logaritmos discretos em um corpo finito primo de ordem $O(10^{28})$ em poucos milisegundos, porém supõe que os fatores de $p-1$ sejam pequenos e conhecidos previamente. Esta suposição é bastante restritiva, pois é fácil gerar primos p' tais que $p'-1$ possua fatores garantidamente grandes, e fatoração de grandes inteiros é em si um problema computacionalmente difícil.

Métodos sub-exponenciais existem para o caso mais genérico de corpos finitos que apresentem uma propriedade chamada *smoothness* (STUDHOLME, 2013), que é o caso de corpos finitos sobre números primos. O *Index Calculus Method* (ICM) é um exemplo notável, que, em linhas gerais, usa essa propriedade para produzir uma pequena base de fatores sobre o corpo, e então produz várias relações lineares entre seus logaritmos randomicamente, até que se chegue em um sistema de equações lineares solucionável. Um logaritmo discreto específico pode então ser encontrado com a ajuda deste sistema. Intuitivamente, o que se faz é procurar aleatoriamente alguma "estrutura" interna do corpo e se basear nela para os cálculos. A complexidade deste método é, assumindo uma seleção ótima da base de fatores:

$$O(e^{(\sqrt{2}+o(1))*\sqrt{\ln p}*\sqrt{\ln \ln p}}) \tag{2.8}$$

Por exemplo, para o número primo de 128bits 204957496037838376002989446472925645899, utilizando duas casas decimais e ignorando-se o fator O(1), a complexidade seria

$$O(e^{(\sqrt{2}*9.39*2.11)}) = O(1.47 * 10^{12}) \tag{2.9}$$

Interpretando este valor como o número de operações necessárias, e imaginando que cada operação tome apenas 1 ciclo de *clock* para ser executada, um *core* do computador no qual este trabalho está sendo escrito (com frequência nominal 2.44GHz) demoraria 10 minutos para achar um logaritmo discreto deste corpo. Entretanto, estes valores são altamente especulativos e otimistas. O gráfico da Figura 4 foi retirado de (STUDHOLME, 2013) e apresenta o tempo de processamento de uma implementação real do ICM rodando em um computador Athlon XP 1700+, de 1.4GHz, para variáveis ordens de corpos finitos. Baseado nestes resultados, o autor faz previsões de tempo para solução de um logaritmo discreto com esta implementação (Tabela 1).

Se a informação tiver de ser mantida confidencial por mais tempo, duas soluções que evitam

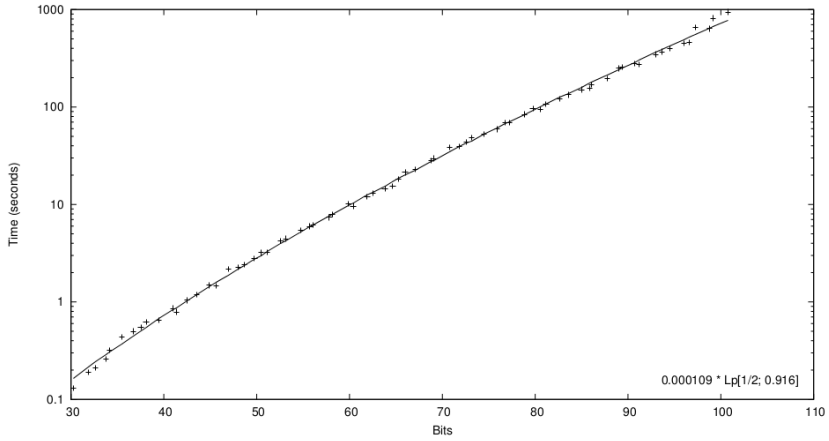

Figura 4 – Tempo de processamento de uma implementação do ICM em função do tamanho do primo (STUDHOLME, 2013)

que este tipo de ataque não seja um problema são: utilizar chaves de tamanho suficiente; utilizar algoritmos imunes a este método. A primeira solução é problemática pois, à medida que as chaves aumentam, também aumenta-se o *overhead* de processamento dos algoritmos criptográficos. Também, à medida que o poder computacional do atacante aumenta, aumenta-se o tamanho de chave passível de ser quebrada em tempo razoável. A segunda solução é explorada na seção seguinte.

Tabela 1 – Previsão de tempo necessário para encontrar logaritmos discretos em corpos de diferentes ordens com o método ICM (STUDHOLME, 2013).

Tamanho do primo	**Tempo**
117 bits	1 hora
154 bits	1 dia
179 bits	1 semana
198 bits	1 mês
234 bits	1 ano

2.6.3 Curvas Elípticas

Não é comprovado que logaritmo discreto (e DDH) seja um problema intratável em algum corpo finito, bem como nunca foi encontrado um corpo finito que comprovadamente não tenha nenhuma es-

trutura interna explorável por métodos como o ICM. Todavia, existe um tipo de grupo no qual ainda não foi encontrado tal estrutura, tornando-o, até onde se sabe, imune a estes ataques (STUDHOLME, 2013). Este grupo é definido pelos pontos de uma curva elíptica prima, mais operação de adição e multiplicação (Apêndice A.2).

Dado um tamanho de chave, o algoritmo de Diffie-Hellman baseado em curvas elípticas (do inglês, ECDH) oferece um maior nível de segurança do que a versão original. Apesar de esta ser mais eficiente se usado um mesmo tamanho de chave, a segurança extra faz com que o ECDH seja mais eficiente se levado em conta o nível de segurança que se deseja alcançar. Por exemplo, para se estabelecer uma chave simétrica de 128 bits, NIST recomenda (NSA, 2009) que valores de 3072 bits sejam utilizados no Diffie-Hellman tradicional contra 256 bits no ECDH. Estima-se que o ECDH seja 10 vezes mais eficiente para este tamanho de chave (NSA, 2009), sendo que esta diferença aumenta ainda mais conforme a chave aumenta.

O ECDH – sumarizado na Figura 5 – funciona de maneira análoga ao algoritmo de Diffe-Hellman explicado anteriormente, porém baseando-se em aritmética de curvas elípticas. Os parâmetros públicos da rede são (STALLINGS, 2011):

G - Um ponto da curva elíptica, chamado de ponto base

p - Um primo que define F_p

n - A ordem do grupo utilizado, proporcional a p e tão grande quanto as chaves a serem geradas.

A geração do par de chaves de cada usuário se dá da seguinte forma: a chave privada é um inteiro aleatório $K_A < n$, e a chave pública é calculada como $P_A = K_A * G$. Após a troca de parâmetros públicos ($A \rightarrow B : P_A$ e $B \rightarrow A : P_B$), a chave final é calculada como:

$$K_{AB} = K_A * P_B = K_B * P_A \tag{2.10}$$

Arthur		Beeblebrox
Escolhe $K_A < n$		Escolhe $K_B < n$
$P_A = K_A * G$	$\longrightarrow$	P_A
P_B	$\longleftarrow$	$P_B = K_B * G$
$K_{AB} = P_B * K_A$		$K_{AB} = P_A * K_B$

Figura 5 – Exemplo de transação ECDH.

2.7 AUTENTICAÇÃO

Um Message Authentication Code (MAC) é uma função, muitas vezes não inversível, que toma como entrada uma mensagem e uma chave criptográfica e computa um pedaço de informação de tamanho fixo. Durante este trabalho, o termo MAC será usado como denotando ora a função, ora o resultado da função. Este código serve para autenticar a mensagem em questão e pode garantir a detecção e consequentemente proteção contra as seguintes ameaças (STALLINGS, 2011):

- ***Masquerade:*** Um nodo X maliciosamente se passar por um nodo Y.

- **Modificação de conteúdo:** A modificação sem autorização do conteúdo de uma mensagem durante o trajeto entre o remetente e o destinatário.
- **Modificação de sequência:** Qualquer modificação não autorizada em uma sequência de mensagens, incluindo reordenamento.
- **Modificação de tempo:** Qualquer atraso inserido na entrega de mensagens; especialmente o *replay* de mensagens antigas.

Para entender como um MAC pode proteger contra tais ameaças, analisemos o seguinte cenário, denotado na Figura 6:

Arthur quer agora enviar a mensagem M para Beeblebrox e ambos compartilham de uma chave simétrica K_{AB}, secreta para o resto do mundo. Imagine também que Arthur não se importa que um terceiro leia a mensagem durante o caminho, mas ele quer ter suficiente certeza de que, se Beeblebrox receber a mensagem, ele terá condições de checar se ela foi modificada por um terceiro durante o trajeto, bem como se ela foi de fato escrita por Arthur. Para isso, ambos podem combinar que ao enviar a mensagem, Arthur vai computar uma segunda mensagem $S_a = MAC(M, K_{AB})$, sendo que o tamanho de S_a é fixo e muitas vezes menor que o de M. Arthur envia então não só a mensagem M, mas a mensagem MS_a (a concatenação das duas mensagens).

Ao receber MS_a, Beeblebrox (que conhece a chave K_{AB}) calcula $S_b = MAC(M, K_{AB})$ e checa se $S_b = S_a$. Caso positivo, Beeblebrox pode estar seguro de que S_a foi calculado por alguém que conhece a chave K_{AB}. Como apenas Arthur e Beeblebrox a conhecem, Beeblebrox sabe que Arthur de fato escreveu aquela mensagem.

Beeblebrox também pode ficar seguro de que a mensagem não foi modificada por alguém

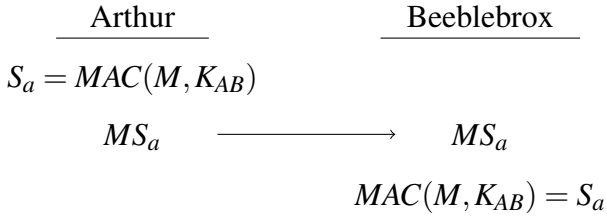

Figura 6 – Exemplo de uso de MAC para autenticação.

que não conheça K_{AB}. Para ilustrar o motivo, na Figura 7 Zaphod intercepta a mensagem MS_a e a substitui pela mensagem $M_z S_a$. Assim, Beeblebrox calculará $S'_b = MAC(M_z, K_{AB})$ e verá que $S'_b \neq S_a$, pois $M_z \neq M$. O mesmo vale caso Zaphod modifique a parte S_a da mensagem, que não pode (ou seja, é extremamente improvável) assumir um novo valor válido sem que ele conheça K_{AB}.

Para proteger contra os dois últimos ataques listados, em geral outras informações devem ser embutidas na mensagem ou no código MAC, como números de sequência e *timestamps*.

2.7.1 One-Time Passwords

Na internet convencional, *One-Time Passwords* (OTP) são senhas descartáveis utilizadas como um segundo fator de autenticação, juntamente com o *login* e senha do usuário. Estas senhas são geradas independentemente pelos dois lados participantes da autenticação a partir de uma mesma semente, ou *Master Secret*, estabelecida previamente (IDALINO, 2012). A propriedade interessante dos OTPs é que a cada novo cálculo o resultado é diferente, por meio do uso de contadores ou *timestamps*.

Por serem sempre diferentes, OTPs podem ser usados para proteger o processo de autenticação contra ataques de *replay*. O protocolo deste trabalho lança mão de OTPs utilizando como *Master*

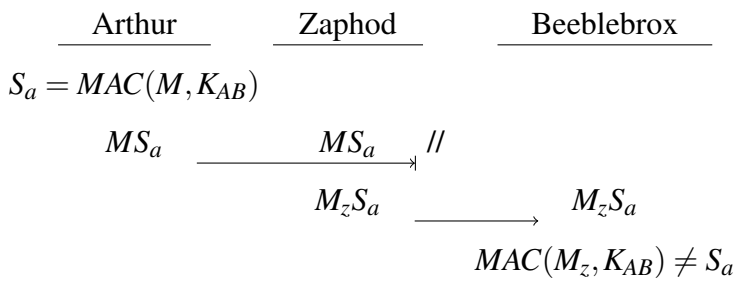

Figura 7 – Exemplo de tentativa de ataque contra mensagem autenticada.

Secret a chave estabelecida pelo Diffie-Hellman mais um identificador único do sensor. O tempo da rede (ou *timestamp*) é utilizado para garantir que cada cálculo resulte em um OTP diferente. O mecanismo utilizado para combinar toda essa informação e gerar um OTP é o algoritmo Poly1305-AES (seção 2.7.3).

Para que o cálculo em cada lado resulte em um mesmo OTP, o algoritmo gerador deve ser alimentado com as mesmas informações, incluindo, neste caso, a *timestamp*. Isso torna necessário que os relógios dos participantes estejam sincronizados, e alcançar o grau desejado de sincronia é a função do PTP, explicado a seguir.

2.7.2 Precision Time Protocol

O protocolo de sincronização temporal implementado no EPOS (OLIVEIRA; OKAZAKI; FRÖHLICH, 2012), conformante com a norma IEEE 1588, permite que sensores sincronizem seus relógios com o de um nodo denominado *master*. O *master* troca mensagens específicas contendo o tempo local com nodos *slave*, que então as comparam com *seu* tempo local e, levando em conta também o tempo entre mensagens, conseguem calcular com grande precisão o *offset* entre os dois relógios, ajustando o seu de acordo. A Figura 8, retirada de (FROHLICH et al., 2013), ilustra as mensagens trocadas pelo protocolo.

Já que essa troca de mensagens costuma ser feita através de *multicast*, a implementação do PTP do EPOS define nodos passivos denominados *listeners*, que ouvem as mensagens trocadas entre mestres e escravos e calculam seus próprios *offsets* com base nelas, sem inserir mais mensagens na rede.

2.7.3 Poly1305-AES

Poly1305-AES (que daqui em diante será chamado de Poly) é o MAC escolhido para o protocolo deste trabalho. Este algoritmo é altamente atrativo primariamente por utilizar-se do AES para alguns dos cálculos e pela disponibilidade de uma prova de que ele garante um nível de segurança muito próximo ao do próprio AES, sendo também comparavelmente eficiente (BERNSTEIN, 2005).

Todos os valores a seguir são interpretados no formato *16-byte unsigned little endian*. Os seguintes parâmetros são tomados como entrada:

- m: Uma mensagem de tamanho arbitrário l,
- n: Um *nonce* (valor que deve ser utilizado apenas uma vez),
- k, r: Duas chaves criptográficas secretas, de 16 bytes cada.

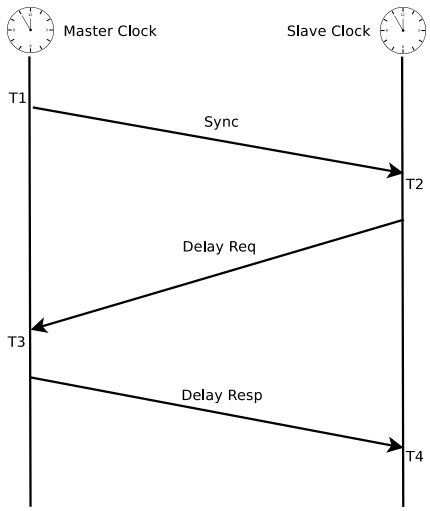

Figura 8 – Mensagens trocadas pelo PTP (FROHLICH et al., 2013)

O primeiro passo é, para cada pedaço m_i de 16 bytes da mensagem, calcular c_i como sendo m_i concatenado com um byte de valor 1. Caso exista um último pedaço de tamanho menor que 16, concatena-se 1 e preenche-se os bytes faltantes com 0. No final, c terá tamanho $q = \lceil l/16 \rceil$.

O autenticador *MAC* gerado é definido como

$$MAC = (((c_1 \cdot r^q + c_2 \cdot r^{q-1} + \cdots + c_q \cdot r^1) \bmod 2^{130} - 5) + AES_k(n)) \bmod 2^{128} \tag{2.11}$$

Algumas responsabilidades do usuário são definidas por (BERNSTEIN, 2005) para que o nível de segurança do Poly seja mantido. O par de chaves k, r deve ser secreto e imprevisível. No protocolo deste trabalho, k é o identificador secreto único de cada sensor e r é a chave secreta gerada pelo algoritmo Diffie-Hellman, então esta responsabilidade é cumprida. Também requer-se que n seja de fato utilizado apenas uma vez. Como o protocolo utiliza a *timestamp* atual como n, esta também é cumprida. Mais adiante será explicado com mais detalhes o modo como Poly é utilizado no protocolo.

Note que o número $2^{130} - 5$ (por exemplo) na equação 2.11 é um valor muito grande, maior do que o maior inteiro representável em 32 bits, tamanho suportado nativamente pelo EPOSMote. A necessidade de lidar com números dessa magnitude (ou muito maiores) em diferentes partes do protocolo exigiu a implementação de uma biblioteca para tratá-los, descrita a seguir.

2.8 BIGNUM

É muito comum em criptografia a necessidade de lidar com aritmética envolvendo números muito grandes, que extrapolam os valores máximos permitidos pelos tipos primitivos da linguagem C++ em processadores comuns. Em computação, esse tipo de número e aritmética é conhecido como *Bignum*. Uma biblioteca Bignum foi implementada no EPOS e suas principais características são descritas a seguir. A implementação foi altamente inspirada em (MENEZES; OORSCHOT; VANSTONE, 1996) e (BROWN et al., 2001).

2.8.1 Representação

A classe Bignum representa na verdade um corpo finito F_p, pois a aritmética de curvas elípticas trabalha dentro de tal corpo. As operações são definidas através dos métodos, os parâmetros (e.g. p) são atributos da classe e o elemento que cada objeto denota é representado por um *array* de *dígitos* no formato *little-endian* (ou seja, o dígito menos significativo ocupa a posição 0 do *array*, seguido pelo segundo menos significativo e assim por diante). Um dígito é um tipo de dado definido pelo usuário

em tempo de compilação, porém recomenda-se que para melhor desempenho utilize-se um tipo que corresponda ao tamanho de uma *word* nativa do processador. A representação dos dígitos pode ser *little-* ou *big-endian*. Neste trabalho assumiremos que um dígito é definido como um *unsigned int*, de 32 bits.

Inicialmente optou-se por incluir como atributo de um objeto Bignum o tamanho em dígitos do seu número atualmente representado. Utilizou-se técnicas de programação como *Thin Template* para possibilitar ao usuário definir o tamanho máximo em tempo de compilação, de modo a permitir que apenas o necessário fosse alocado a cada Bignum sem com isso acarretar em *overhead* de código gerado. Porém, observou-se que o código de muitas operações se torna mais simples (resultando então em menos código gerado) se assumir-se que todos os *arrays* tem o mesmo tamanho, como recomendado em (MENEZES; OORSCHOT; VANSTONE, 1996) e (BROWN et al., 2001). Por tratar-se de um corpo finito módulo p, a escolha evidente foi então fazer com que todos os Bignums contivessem um *array* tão grande quanto o necessário para representar p. O tamanho em dígitos deste *array* é chamado *word*. Os operadores C++ das operações aritméticas mais comuns foram então sobrescritos para prover ao usuário uma interface natural.

2.8.2 Aritmética

A implementação do Diffie-Hellman sobre curvas elípticas exige suporte às quatro operações básicas sobre F_p: adição, subtração, multiplicação e divisão. Esta seção apresenta como cada uma foi implementada. Nos pseudocódigos a seguir, a e b são *arrays* de dígitos representando um inteiro, como explicado anteriormente. Alguns atributos de Bignum mencionados na subseção anterior também são utilizados (e.g. *word*, p).

2.8.2.1 Adição

A adição de elementos a e b do corpo finito formado pelos inteiros módulo p é definida como $(a+b) \bmod p$. Para a adição modular, utiliza-se como base uma operação de adição simples, a qual chamaremos de *simple_add* (algoritmo 1). Esta operação soma dois *arrays* dígito por dígito, levando em conta *carries*, retornando um *carry* de 1 se houver *overflow*. Vale apontar que as operações deste algoritmo são implementáveis de forma muito eficiente utilizando-se *shifts* e tipos de tamanho duplo (e.g. *unsigned long long*), ou até mesmo em *assembly* com operações *add with carry*, suportada por muitos processadores. O algoritmo 2 é adaptado do algoritmo 1 de (BROWN et al., 2001).

Algoritmo 1 – Adição simples (simple_add)

Entrada: $a, b \in [0, p-1]$; $carry \in \{0,1\}$.
Saída: $carry \in \{0,1\}$; $c = (a+b) \bmod 2^{32*word}$

1: **Para cada** $i \in [0, word]$:
2: $\quad c_i = (a_i + b_i + carry) \bmod 2^{32}$
3: $\quad carry = (a_i + b_i + carry)/2^{32}$
4: **Retorne** *carry* **e** c

Algoritmo 2 – Adição modular

Entrada: $a, b \in [0, p-1]$.
Saída: $c = (a+b) \bmod p$

1: $c \leftarrow simple\_add(a, b)$
2: **Se** houver *carry*, **então** $c \leftarrow simple\_sub(c, p)$
3: **Se** $c > p$, **então** $c \leftarrow simple\_sub(c, p)$
4: **Retorne** c

2.8.2.2 Subtração

Subtração funciona de modo similar à adição. O algoritmo 3 corresponde ao 1, porém o *carry* é interpretado como um *borrow*. O algoritmo 4 foi adaptado do algoritmo 2 de (BROWN et al., 2001).

Algoritmo 3 – Subtração simples (simple_sub)

Entrada: $a, b \in [0, p-1]$; $borrow \in \{0, 1\}$.
Saída: $borrow \in \{0, 1\}$; $c = a - b$

1: **Para cada** $i \in [0, word]$:
2: **Se** $a_i < b_i + borrow$:
3: $c_i = 2^{32} - (b_i + borrow) + a_i$
4: $borrow \leftarrow 1$
5: **Senão**:
6: $c_i = a_i - (b_i + borrow)$
7: $borrow \leftarrow 0$
8: **Retorne** *borrow* **e** c

Algoritmo 4 – Subtração modular

Entrada: $a, b \in [0, p-1]$.
Saída: $c = (a-b) \bmod p$

1: $c \leftarrow simple\_sub(a, b)$
2: **Se** houver *carry*, **então** $c \leftarrow simple\_sub(c, p)$
3: **Retorne** c

2.8.2.3 Multiplicação

O algoritmo 5 descreve uma rotina de multiplicação básica e é baseado no algoritmo 3 de (BROWN et al., 2001). (uv) significa o resultado de 64 bits da operação de multiplicação de dois números de 32 bits, sendo u os 32 bits mais significativos e v os restantes. Nas linhas 6 e 7, *carry* se refere ao *overflow* da última soma de dois números de 32 bits. O resultado c da rotina terá tamanho $2 * word$ e deve-se, após a execução, aplicar o módulo p a ele.

Para aplicação do módulo neste caso, utiliza-se o eficiente algoritmo de redução de Barrett (BARRETT, 1987), algoritmo 6 (algoritmo 6 de (BROWN et al., 2001)). Este algoritmo requer o

Algoritmo 5 – Multiplicação

Entrada: $a, b \in [0, p-1]$.
Saída: $c = a * b$

1: $r0 \leftarrow r1 \leftarrow r2 \leftarrow 0$
2: **Para cada** $k \in [0, word * 2 - 2]$:
3: **Para cada** $\{(i, j) \mid i + j = k, 0 \leq i, j < word\}$:
4: $(uv) = a_i * b_j$
5: $r0 \leftarrow r0 + v$
6: $r1 \leftarrow r1 + u + carry$
7: $r2 \leftarrow r2 + carry$
8: $c_k \leftarrow r0$, $r0 \leftarrow r1$, $r1 \leftarrow r2$, $r2 \leftarrow 0$
9: $c_{2*word-1} \leftarrow r0$
10: **Retorne** c

cálculo da constante μ, porém como esta constante depende apenas de parâmetros estáticos da rede, ela pode ser calculada apenas uma vez, até mesmo antes da compilação. A *base* mencionada é a base com a qual os dígitos estão representados, por exemplo, 2^{32} para o caso assumido. Os módulos e divisões pela base podem ser implementados de forma muito eficiente através da manipulação dos índices do *array*, simulando *shifts*.

Algoritmo 6 – Redução modular de Barrett

Entrada: base $b > 3$, $k = word$, $a \in \left[0, b^{2k} - 1\right]$, $\mu = \left\lfloor b^{2k}/p \right\rfloor$.
Saída: $a \bmod p$

1: $q \leftarrow \left\lfloor \left\lfloor (a/b^{k-1} \right\rfloor * \mu)/b^{k+1} \right\rfloor$
2: $r \leftarrow (q * p) \bmod b^{k+1}$
3: $r \leftarrow ((a \bmod b^{k+1}) - r) \bmod b^{k+1}$
4: **Enquanto** $r \geq p$: $r = r - p$
5: **Retorne** r

2.8.2.4 Divisão

Divisão sobre F_p é definida através da inversa multiplicativa modular, que por sua vez é definida como:

$$\forall a \in [0, p-1] : a * a^{-1} \equiv 1 (\bmod p) \tag{2.12}$$

E por ser a inversa multiplicativa apresenta a seguinte propriedade:

$$\forall a, b \in [0, p-1] : a/b (\bmod p) \equiv a * b^{-1} (\bmod p) \tag{2.13}$$

Logo, o custo de uma divisão é igual ao custo de uma inversão mais uma multiplicação. O algoritmo 7 implementa a inversão multiplicativa modular, e é o algoritmo 12 de (BROWN et al., 2001), que por sua vez é uma eficiente variante para corpos finitos módulo p do Algoritmo Estendido de Euclides.

2.8.2.5 Aritmética de Curvas Elípticas

As curvas elípticas sobre F_p implementadas precisam definir operações de soma e multiplicação (Apêndice A.2), e para isso utilizam-se das operações detalhadas nesta seção até aqui. Um ponto $P = (x_P, y_P)$ da curva é representado internamente através de *coordenadas Jacobianas*, como mostrado mais eficiente em (BROWN et al., 2001). Essa representação utiliza 3 coordenadas por ponto, e o ponto jacobiano (x, y, z) corresponde ao ponto $(x/z^2, y/z^3)$. Para calcular um ponto jacobiano $(x_R, y_R, z_R) = 2(x, y, z)$, utiliza-se a seguinte fórmula (todas as operações são em F_p):

$$A = 4x \cdot y^2, \quad B = 8 \cdot y^4, \quad C = 3(x - z^2) \cdot (x + z^2)$$

$$x_R = -2A + C^2, \quad y_R = C \cdot (A - x_R) - B, \quad z_R = 2y \cdot z \tag{2.14}$$

Esta representação permite definir eficientes operações de soma entre pontos convencionais e jacobianos, de modo que os pontos possam ser guardados e enviados em coordenadas convencionais, e coordenadas jacobianas são usadas apenas para cálculos internos. A fórmula a seguir calcula $(x_R, y_R, z_R) = (x_C, y_C, 1) + (x_J, y_J, z_J)$:

$$A = x_C \cdot z_J^2, \quad B = y_C \cdot z_J^3, \quad C = A - x_J, \quad D = B - y_J$$

$$x_R = D^2 - (C^3 + 2x_J \cdot C), \quad y_R = D \cdot (x_J \cdot C^2 - x_R) - y_J \cdot C^3, \quad z_R = z_J \cdot C \tag{2.15}$$

Multiplicação é implementada pelo algoritmo 8, correspondente ao algoritmo 13 de (BROWN et al., 2001). Na linha 3, a operação 2.14 é utilizada, e na linha 4, 2.15. Depois da multiplicação, realiza-se uma inversão e quatro multiplicações em F_p para converter o ponto de volta para coordenadas convencionais.

Algoritmo 7 – Inversão multiplicativa modular

Entrada: $a \in [1, p-1]$.
Saída: $C = a^{-1} \bmod p$

$u \leftarrow a, v \leftarrow p, A \leftarrow 1, C \leftarrow 0$
Enquanto $u \neq 0$:
    Enquanto u for par:
        $u \leftarrow u/2$
        Se A for par, $A \leftarrow A/2$, **senão** $A \leftarrow (A+p)/2$
    Enquanto v for par:
        $v \leftarrow v/2$
        Se C for par, $C \leftarrow C/2$, **senão** $C \leftarrow (C+p)/2$
    Se $u \geq v$:
        $u \leftarrow (u-v) \bmod p$
        $A \leftarrow (A-C) \bmod p$
    Senão:
        $v \leftarrow (v-u) \bmod p$
        $C \leftarrow (C-A) \bmod p$
Retorne C

Algoritmo 8 – Multiplicação de pontos

Entrada: $k = (k_{m-1}, \cdots, k_1, k_0)_2$, $P \in E_p(a,b)$.
Saída: $Q = kP$

$Q \leftarrow O$
Para cada i **de** $m-1$ **até** 0:
    $Q \leftarrow 2Q$
    Se $k_i = 1$ **então** $Q \leftarrow Q + P$
Retorne Q

3 TRABALHOS RELACIONADOS

TinySec (KARLOF; SASTRY; WAGNER, 2004), segundo (FRÖHLICH; STEINER; RUFINO, 2011), implementa uma arquitetura para comunicação segura em RSSF, a qual provê cifragem e autenticação de mensagens. A implementação é escrita em nesC (aproximadamente 3000 linhas de código) e é destinada às plataformas Mica, Mica2 e Mica2Dot, requerendo 7146 bytes de código e 728 bytes de memória RAM. TinySec pode ser utilizado de dois modos: *authenticated encryption* (cifragem mais autenticação) e *authentication only* (apenas autenticação). No primeiro modo, o *payload* é cifrado com o cifrador Skipjack (NSA, 1998) e provê autenticação por meio de um MAC anexado à mensagem. No segundo modo, o MAC é anexado de modo similar, porém a mensagem não é cifrada. Apesar de possibilitar esta flexibilidade de modo de uso, a inclusão de um MAC na mensagem acarreta em um custo a mais de uso do rádio, e consequentemente de consumo energético. Ademais, o nível de segurança provido pelo MAC é proporcional ao seu tamanho (SUN et al., 2010), bem como o custo mencionado. A solução do TinySec para reduzir o consumo energético é reduzir também o nível de segurança, por meio de uma redução do tamanho do MAC. TinySec não apresenta uma solução contra ataques de *replay*.

MiniSec (LUK et al., 2007), segundo (FRÖHLICH; STEINER; RUFINO, 2011), é uma estrutura similar ao TinySec que propõe solver os problemas deste. Isso é alcançado principalmente pelo uso de um modo de cifragem que provê confidencialidade e autenticação na mesma iteração. Este modo de cifragem requer que um Vetor de Inicialização (IV) seja compartilhado entre os participantes da comunicação, e MiniSec reduz o tráfego de rede com uma técnica que permite transmitir apenas uma parte de tal Vetor sem prejudicar o nível de segurança. Ataques de *replay* são prevenidos por meio do uso de contadores sincronizados, mas também são mandados apenas alguns bits do contador por pacote, para reduzir o tráfego na rede. Porém, (JINWALA et al., 2009) aponta que uma rotina de ressincronização bastante custosa deve ser executada caso os contadores percam a sincronia (por entrega de pacotes fora de ordem, por exemplo).

Além destas estruturas, existem outras descrições de protocolos completos, incluindo uma eficiente proposta (HUANG et al., 2003) baseada em Curvas Elípticas que utiliza a ideia de combinar operações de criptografia simétrica e assimétrica. Os autores propõem um esquema em que os servidores executem operações mais complexas para que os sensores sejam poupados algum trabalho. Os autores chamam esta estratégia de um esquema *híbrido*. (PAN; WANG; MA, 2011) apresenta algumas falhas deste protocolo, mostrando que algumas das suas alegações quanto à segurança provida não são válidas. Tais falhas são corrigidas por uma proposta de alteração no protocolo.

Um protocolo baseado em IDs únicos dos sensores é apresentado em (LI-PING; YI, 2009) e uma análise superficial de custo de processamento e banda de rede indica que esta abordagem pode ser eficiente. Em todos esses trabalhos, entretanto, ou o sensor deve ter informação sensível e específica pré-carregada, ou assume-se que um terceiro agente distribuidor de certificados participe provendo chaves públicas certificadas através de um canal seguro alternativo. O protocolo proposto neste texto contrasta por focar na redução deste esforço pré-implementação da rede e por propor uma solução prática para estabelecimento de chaves.

4 O PROTOCOLO

Neste capítulo propõe-se um protocolo para estabelecimento de chaves criptográficas e comunicação segura para Redes de Sensores sem Fio e Internet das Coisas baseado nos conceitos apresentados anteriormente. O protocolo é dividido em 4 fases: Inicialização, Estabelecimento de Chaves, Autenticação e Comunicação Segura, que devem acontecer nesta ordem. Cada uma dessas é apresentada nas seções após a seção 4.1, que apresenta as condições do ambiente que o protocolo assume. Uma análise mais detalhada acerca da eficiência e segurança provida, bem como das vantagens e desvantagens de alguns aspectos do protocolo é objeto do capítulo 5.

Para melhor visualização da descrição apresentada ao longo deste capítulo, a Figura 9 apresenta uma visão geral da interação entre os diversos algoritmos envolvidos no protocolo. A Figura 10 apresenta um exemplo completo de iteração do protocolo para uma rede composta apenas por um sensor e o *gateway*.

4.1 PREMISSAS

As seguintes condições são assumidas acerca do ambiente para que o protocolo possa funcionar corretamente. Algumas dessas premissas são discutidas no capítulo 5.

1. Cada sensor tem um identificador único na rede, chamado ID. Um ID não precisa ter um formato específico, pode ser, por exemplo, o número de série do microcontrolador.

2. Assume-se uma topologia de estrela da rede. Cada sensor consegue alcançar e ser alcançado por um nodo central, denominado *gateway*.

3. O *gateway* assume o papel de *master* no PTP, um sensor é designado *slave* e os demais operam em modo *listener*.

4. Comunicação segura é necessária apenas na comunicação entre sensor e *gateway*.

5. Um sensor autenticado não sairá da rede e voltará mais tarde para tentar se autenticar novamente.

6. O *gateway* é uma máquina local, segura e controlável, preferencialmente com maior poder de processamento que os sensores.

4.2 INICIALIZAÇÃO

Esta seção identifica os passos que devem ser tomados por sensores e *gateway* antes de efetivamente iniciar-se as trocas de mensagens do protocolo.

4.2.1 Inicialização do *Gateway*

O *gateway* deve definir os parâmetros globais do Diffie-Hellman sobre Curvas Elípticas: p, n, G, bem como a constante μ (seções 2.6.3 e 2.8.2.3). Estes parâmetros devem ser tornados públicos no contexto da rede de algum modo, por exemplo por meio de *broadcasting* ou, como feito nesta implementação, carregando-os na memória dos sensores em tempo de compilação. Nesta etapa, o *gateway* deve também gerar seu par de chaves ECDH (P_g, K_g).

Para que um sensor possa se autenticar, o ID dele tem que ser cadastrado no *gateway* antecipadamente. Delega-se ao usuário a responsabilidade de fazê-lo de um modo seguro, já que é assumido que o *gateway* é uma máquina local, confiável e controlável. Assim que um ID – digamos, ID_i – é cadastrado, o *gateway* calcula e guarda associado àquele ID um código *Auth*:

$$Auth_i = AES(ID_i, ID_i) = \{ID_i\}_{ID_i} \tag{4.1}$$

Daqui em diante, a notação de cifragem $\{M\}_k$ passa a significar o resultado da cifragem da mensagem M sobre a chave k *com o cifrador AES*. O objetivo do código *Auth*, mais adiante no protocolo, será mostrar uma evidência de conhecimento do ID correspondente sem revelá-lo. O processo de cadastro – bem como de remoção – de IDs pode acontecer a qualquer momento, permitindo ao usuário um controle "manual" de quais sensores podem se autenticar em dado momento.

O *gateway* também assume o papel de *master* no PTP, e deve nesta fase inicializar o protocolo e começar a enviar mensagens de sincronização periodicamente, permitindo assim que sensores que entrem na rede sincronizem seus relógios. Por ser executado antes de qualquer autenticação, as mensagens do PTP não são seguradas por este protocolo.

Assim que estes passos foram tomados, o *gateway* está pronto para enviar mensagens *DH_Request* e assim entrar na segunda fase do protocolo, apresentada na seção 4.3.

4.2.2 Inicialização de Sensores

A cada sensor é designado um ID único na rede, e a estratégia para garantir isso é responsabilidade do usuário. Aqui, será assumido que o ID encontra-se previamente carregado no sensor em memória não-volátil. Ao ser inicializado, o sensor deve calcular e guardar seu código $Auth = \{ID\}_{ID}$.

Ao ingressar numa rede, é responsabilidade do sensor configurar seus parâmetros ECDH de acordo com os parâmetros globais publicados pelo *gateway*. Caso esses parâmetros sejam conhecidos com devida antecedência pode-se alternativamente carregá-los em tempo de compilação, como foi feito nesta implementação. Então, o par de chaves (P_s, K_s) deve ser calculado.

Independentemente da inicialização ECDH, o sensor deve sincronizar seu relógio com o do *gateway* seguindo o protocolo PTP, respondendo mensagens de sincronização caso seja um *slave* ou apenas ouvindo caso seja um *listener*.

Uma vez que os parâmetros ECDH estejam inicializados e o relógio sincronizado, o sensor permanece inativo esperando uma mensagem *DH_Request*. Assim que esta chegar, passa-se para a fase 2 do protocolo.

4.3 ESTABELECIMENTO DE CHAVES

Esta fase consiste em uma transação ECDH tradicional. O objetivo é estabelecer uma chave simétrica conhecida apenas pelos dois participantes, a qual só será utilizada se validada posteriormente (seção 4.4).

A primeira mensagem é enviada pelo *gateway*, e é denominada *DH_Request(P_g)*. A parte entre parênteses denota um valor que é enviado juntamente com a mensagem – neste caso P_g, a chave pública do *gateway*. Esta mensagem pode ser enviada por *multicast* ou para um nodo em específico.

Assim que uma mensagem *DH_Request(P_g)* chega ao sensor de destino – por exemplo, o sensor s – este primeiramente responde com *DH_Response(P_s)* e então calcula a chave simétrica $K_{sg} = K_s * P_g$, a qual será denominada *Master Secret*. Similarmente, o *gateway* recebe *DH_response(P_s)* e calcula $K_{sg} = K_g * P_s$.

Agora, sensor e *gateway* compartilham de uma chave, porém como o processo ainda não in-

cluiu nenhum aspecto de autenticação, nenhuma das partes tem nenhuma garantia de que compartilha uma chave com um nodo confiável e não-malicioso da rede. É este problema que a próxima fase do protocolo visa solucionar.

4.4 AUTENTICAÇÃO

O *Master Secret* estabelecido na fase anterior deve ser validado antes de poder ser usado. Caso um sensor envie mensagens cifradas com uma K_{sg} não autenticada, o *gateway* não deve aceitá-la (o mesmo vale para mensagens do *gateway* para o sensor).

Para autenticar uma chave, o sensor envia para o *gateway* a mensagem *Auth_Request(*$Auth_s, OTP_s$*)*, sendo:

$$OTP_s = Poly(MI_s, K_{sg}, ID_s, T_i) \tag{4.2}$$

$$MI_s = K_{sg} \oplus ID_s \tag{4.3}$$

Ao receber uma mensagem *Auth_Request(*$Auth_s, OTP_s$*)* esperada, o *gateway* deve verificar se existe uma chave esperando autenticação que esteja implícita em OTP_s. O processo (apresentado em forma de pseudocódigo no algoritmo 9) começa com uma busca em seu banco de dados, através de $Auth_s$, pelo ID_s correspondente. Então, para cada chave K_{ag} pendendo autenticação,é calculado $MI_a = K_{ag} \oplus ID_s$, $OTP_a = Poly(MI_a, K_{ag}, ID_s, T_i)$ e checado se $OTP_a = OTP_s$. Caso nenhum OTP satisfaça a igualdade, o *gateway* não responde. Caso OTP_s seja encontrado, o *gateway* tem evidência de que a chave K_{ag} é na verdade K_{sg} e foi de fato compartilhada com o sensor autêntico identificado por ID_s, e associa estes dois valores para comunicação futura. O *gateway* então envia a última mensagem: *Auth_Granted(*$\{ID_s\}_{KT_{sg}}$*)* para prover evidência ao sensor de que este nodo é de fato um *gateway* autêntico, que conhece ID_s. A chave KT_{sg} é uma chave derivada de K_{sg} com o mecanismo explicado na seção a seguir.

Ao receber *Auth_Granted(*$\{ID_s\}_{KT_{sg}}$*)*, o sensor tem condições de derivar KT_{sg} para decifrar a mensagem e checar se ID_s está correto. Caso positivo, o sensor passa a confiar no *Master Secret* K_{sg}. Caso contrário, o sensor não deve se considerar autenticado e deve começar novamente o protocolo, esperando por uma nova mensagem *DH_Request*.

Algoritmo 9 – Associação de *MasterSecret* a ID

Entrada: $Auth_s, OTP_s$
Saída: *Verdadeiro* se deve-se confiar em *s*, *Falso* caso contrário

1: Busque ID_s no banco de dados, através de $Auth_s$
2: **Se** ID_s não foi encontrado:
3: **Retorne** *Falso*
4: **Para cada** K_{ag} pendendo autenticação:
5: $MI_a \leftarrow K_{ag} \oplus ID_s$
6: $OTP_a \leftarrow Poly(MI_a, K_{ag}, ID_s, T_i)$
7: **Se** $OTP_a = OTP_s$:
8: Valide K_{ag}, associando-a a ID_s
9: **Retorne** *Verdadeiro*
10: **Retorne** *Falso*

4.5 COMUNICAÇÃO SEGURA

Uma vez que o *Master Secret* K_{sg} foi autenticado, ele está pronto para ser utilizado como chave para cifragem AES, porém não diretamente. Quando um sensor precisa enviar uma mensagem M segura para o *gateway* (e vice-versa), ele deve derivar a chave KT_{sg} segundo a equação 4.4, adicionar à mensagem um *checksum* C e então enviar a mensagem $Secure\_Message(\{CM\}_{KT_{sg}})$.

$$KT_{sg} = Poly(MI_s, K_{sg}, ID_s, T_i) \tag{4.4}$$

Se o destinatário é um sensor, este deriva KT_{sg}, decifra a mensagem para obter CM e checa se o *checksum* está correto (se não estiver, ignora-se a mensagem). Caso o destinatário seja o *gateway*, ele deve derivar uma KT_{ag} para todos os sensores a autenticados, e então tentar decifrar a mensagem com cada chave até que uma delas gere o *checksum* corretamente (se nenhuma gerar, a mensagem é descartada). Caso a implementação permita acesso a um endereço de uma camada inferior da rede, pode-se usá-lo como heurística para acelerar essa busca por K_{sg}.

Caso C seja gerado corretamente, o destinatário tem evidência para acreditar que a mensagem M foi enviada pelo único outro nodo da rede que conhece o *Master Secret* já validado, e então confia que a mensagem é autêntica, confidencial, íntegra e foi enviada dentro do *timestamp* atual.

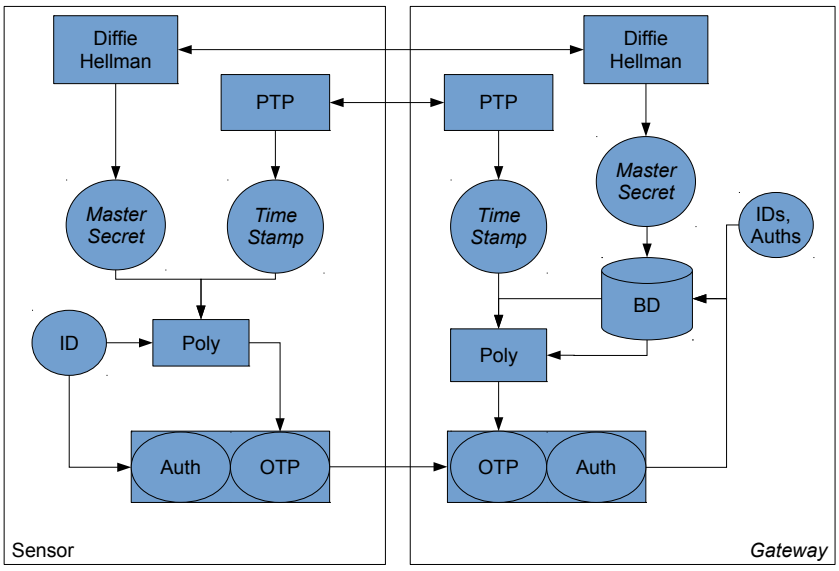

Figura 9 – Visão geral da interação entre vários algoritmos no protocolo

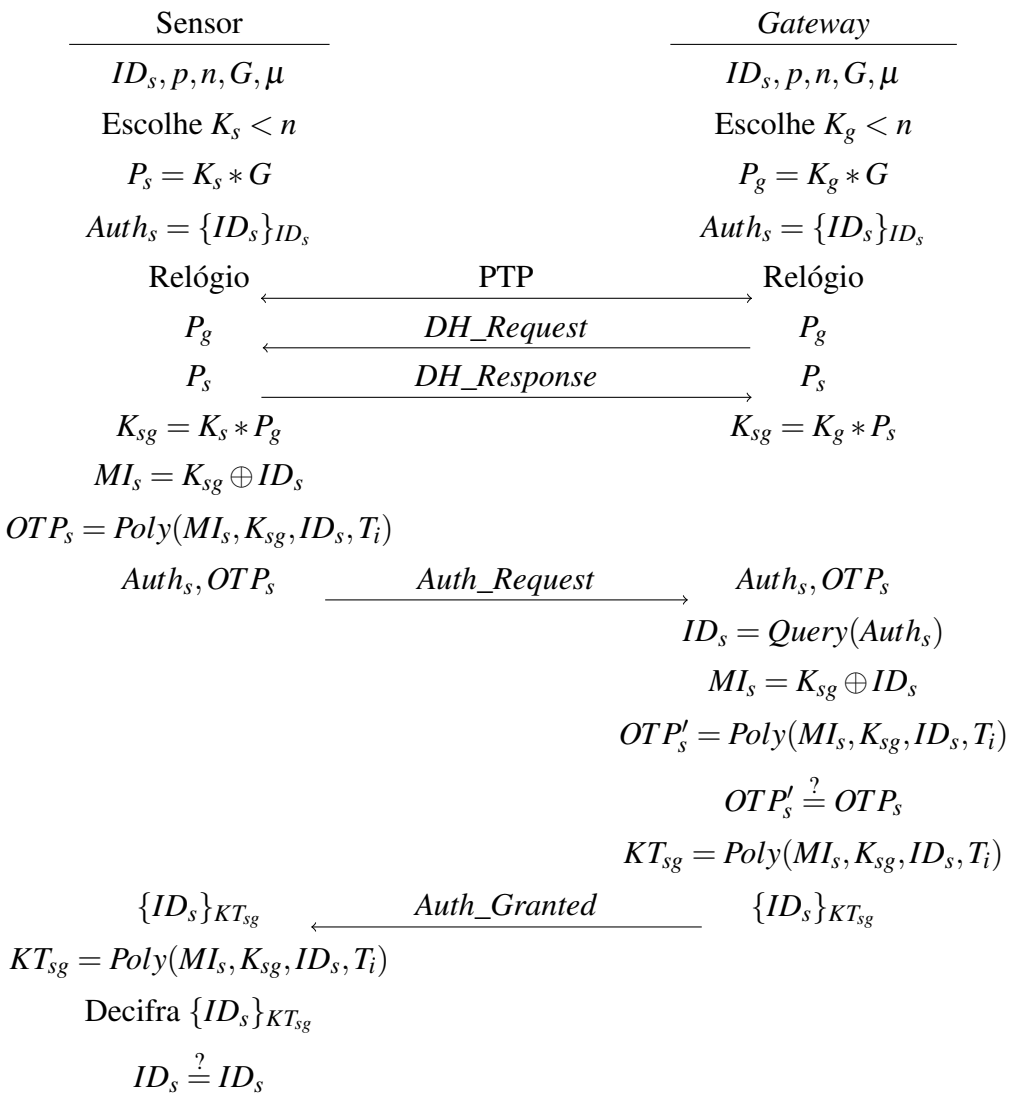

Figura 10 – Exemplo de iteração do Protocolo de Estabelecimento de Chaves.

5 AVALIAÇÃO

Este capítulo tem o objetivo de analisar a viabilidade e eficiência do protocolo proposto. A análise é dividida em duas partes: a seção 5.1 traz uma discussão de caráter qualitativo acerca da segurança do protocolo, suas principais características positivas e limitações. A seção seguinte, 5.2, apresenta uma descrição de testes rodados sobre uma implementação do protocolo, bem como os dados levantados e análise dos mesmos.

5.1 ANÁLISE DA PROPOSTA

5.1.1 Princípios de segurança providos

Se uma mensagem segura $\{CM\}_{KT_{sg}}$ consegue ser decifrada pelo destinatário de modo a gerar C corretamente, significa que:

- M é **confidencial**, pois apenas s e g conhecem K_{sg}, e consequentemente KT_{sg}.
- M é **autêntica**, pois só s e g conhecem ambos K_{sg} e ID_s, utilizados para gerar KT_{sg}.
- M é **íntegra**, pois é extremamente improvável que $\{CM\}_{KT_{sg}}$ seja alterada cegamente de forma que o *checksum* resultante continue válido.
- $\{CM\}_{KT_{sg}}$ foi gerada na *timestamp* atual da rede.

5.1.2 *Checksum*

O protocolo em si é agnóstico da forma como o *checksum* é implementado. Técnicas para reduzir o *overhead* de comunicação acarretado pela inclusão dele são citadas como trabalho futuro. Em especial, aponta-se para uma integração com o *checksum* normalmente presente em camadas inferiores da rede, de acordo com a filosofia de mistura entre diferentes camadas, ou *cross-layer*.

5.1.3 Derivação de chaves

A ordem dos parâmetros na derivação das chaves KT e de OTPs foi pensada de forma a concordar com as responsabilidades do usuário definidas por (BERNSTEIN, 2005). Por conveniência, repete-se aqui a equação (2.11) de descrição do Poly1305-AES e as equações de derivação de chave (4.2 e 4.3):

$$Poly(c,r,k,n) = (((c_1 \cdot r^q + c_2 \cdot r^{q-1} + \cdots + c_q \cdot r^1) \bmod 2^{130} - 5) + AES_k(n)) \bmod 2^{128}$$

$$MI_s = K_{sg} \oplus ID_s$$

$$OTP_s = Poly(MI_s, K_{sg}, ID_s, T_i)$$

A *timestamp* T_i, sendo i o momento atual, é usada como parâmetro n pois (BERNSTEIN, 2005) afirma que este parâmetro deve ser um *nonce*. Os parâmetros r,k são respectivamente ocupados por K_{sg}, ID_s pois segundo (BERNSTEIN, 2005) este par deve ser secreto. O primeiro parâmetro MI_s é usado com o intuito de aproveitar o tamanho total da chave e do ID caso eles excedam 16 bytes, pois Poly usa

apenas este tamanho de cada um dos outros parâmetros.

5.1.4 IDs

O identificador único de cada sensor não precisa ter uma forma específica, porém algumas recomendações impactam o nível de segurança provido. Um atacante pode se passar por um sensor autêntico caso descubra um ID válido e o use antes do sensor autêntico correspondente àquele ID (ver seção 5.1.8), então o mais desejável é que seja muito difícil um atacante descobrir um ID válido. Para isso, 3 fatores influentes são o sigilo com o qual os IDs são guardados, o tamanho e a previsibilidade deles.

Sigilo é uma questão que fica a cargo do usuário, pois muito depende do contexto específico no qual a rede está inserida, incluindo fatores humanos e logísticos como a terceirização da equipe que gera e instala os IDs. Idealmente, os IDs devem ser mantidos em sigilo e revelados apenas a quem precisa sabê-los.

O tamanho do ID está diretamente relacionado ao nível de segurança. Quanto maior o ID, mais difícil será para um atacante adivinhá-lo cegamente. Os códigos *Auth* e OTP se tornam mais seguros pelo aumento do tamanho do texto plano se utilizado um modo de cifragem do AES que aproveite todo esse tamanho (e.g. modo CBC). Um ID menor do que 128 bits é altamente desencorajado, pois afetaria os tamanhos de valores utilizados como chave em *Auth* e OTP, o que diminuiria a segurança do AES na derivação deles.

Previsibilidade é uma questão importante pois um atacante que tenha uma ideia do formato dos IDs pode muito mais facilmente encontrá-los através de tentativa e erro. Assim, aconselha-se que os IDs sejam, na medida do possível, aleatórios.

Em suma, de modo ideal recomenda-se que os IDs sejam números aleatórios de pelo menos 128 bits, os quais são revelados apenas para quem é incumbido de carregá-los no *gateway*. Caso este cenário não seja viável, deve-se pensar nesses três fatores na medida do possível. Se deseja-se usar o número de série não-secreto de 10 bits de cada sensor como ID, por exemplo, pode-se ao invés disso utilizá-los como semente de uma função secreta geradora de números pseudo-aleatórios de maior magnitude e usar o resultado como ID.

Com essas recomendações em mente, pode-se pensar que seria muito mais fácil e eficiente simplesmente carregar chaves simétricas pré-estabelecidas, ou utilizar o próprio ID diretamente como chave. Nessa estratégia, porém, a descoberta de um ID imediatamente permite a um atacante decifrar *todas* as mensagens – passadas e futuras – que o sensor correspondente trocar, enquanto no protocolo proposto o atacante deve, além de descobrir o ID, solucionar o problema do logaritmo discreto para curvas elípticas, o que por si só é um problema nada trivial. Por não derivar chaves exclusivamente a partir do ID, o protocolo proposto provê segurança mesmo que eles sejam empregados descuidadosamente (ver seção 5.1.8), permitindo assim um certo relaxamento quanto aos requerimentos acerca do ID, enquanto usá-los diretamente como chave requereria um cuidado tão grande quanto a segurança desejada.

5.1.5 Tamanho dos parâmetros ECDH

O tamanho dos parâmetros ECDH a serem utilizados implica num aumento do tempo que um atacante demoraria para solucionar o problema do logaritmo discreto para curvas elípticas e descobrir uma chave K_{sg} baseado apenas nas mensagens abertas trocadas durante o algoritmo Diffie-Hellman. NIST apresenta (Tabela 2) uma estimativa de equivalência de nível de segurança entre o algoritmo AES com diferentes tamanhos de chave e chaves usadas em criptografia com Curvas Elípticas. Por

exemplo, estima-se que o mesmo esforço seja necessário para quebrar uma mensagem cifrada com AES 128 bits do que com Curvas Elípticas 256 bits, então se ECDH for utilizado para gerar chaves que serão usadas por um AES128, recomenda-se que use-se parâmetros de 256 bits no ECDH.

O protocolo proposto deriva chaves de 128 bits para uso com AES, então um aumento dos parâmetros ECDH além de 128 bits não teria nenhuma influência sobre um ataque exclusivamente contra o AES. A segurança do processo de derivação de chaves cresce proporcionalmente ao tamanho do *Master Secret* ECDH por consequência do parâmetro *MI*, que utiliza todo o tamanho do *Master Secret*.

Tabela 2 – Equivalência de nível de segurança para diferentes tamanhos (em bits) de chaves AES e ECC (NSA, 2009).

Chave AES	**Chave ECC**
80	160
112	224
128	256
192	384
256	521

5.1.6 *Timestamps*

As *timestamps* utilizadas no protocolo não são o valor do relógio diretamente. Ele deve ser arredondado para um múltiplo de um período de tempo pré-definido para a rede, que definirá a *granularidade* da *timestamp*. Deve-se aqui haver um ponto de contato com as camadas inferiores da pilha de comunicação, pois a *timestamp* utilizada para os cálculos deve refletir o momento em que a mensagem é de fato enviada. Em redes que implementam *duty cycling*, por exemplo, um remetente deve enviar mensagens de preâmbulo por um tempo X antes de efetivamente enviar a mensagem desejada, então a *timestamp* utilizada para cifragem de uma mensagem a ser enviada neste cenário deve ser T_{i+X}.

Deve-se assegurar que dentro de uma *timestamp* uma mensagem consiga viajar do remetente até o destinatário. Não se deve utilizar *timestamps* que mudem com periodicidade muito baixa, pois ataques de *replay* são possíveis dentro de uma mesma *timestamp*.

5.1.7 PTP inseguro

Já que o protocolo requer relógios sincronizados como pré-condição e o PTP é utilizado para garantir essa sincronia, as trocas de mensagens PTP devem acontecer de modo não-seguro, ao menos enquanto houver sensores não-autenticados. Isso permite que atacantes se passem por *masters* ou *slaves* e impeçam que sensores autênticos sincronizem seus relógios corretamente. Entretanto, as *timestamps* geradas para uso no protocolo são usadas apenas como mecanismo de proteção a *replays*, então um ataque ao PTP resultaria apenas em negação de serviço, sem qualquer informação sigilosa comprometida ou autenticação manipulada. Vulnerabilidade a negação de serviço é um problema (no caso geral) intrínseco a redes *wireless*, pois um atacante pode simplesmente injetar muitas mensagens na mesma faixa de frequência utilizada, impedindo assim *qualquer* comunicação. Este ataque é conhecido como *jamming* de canal.

5.1.8 Possíveis ataques

Nesta seção são considerados alguns outros possíveis ataques ao protocolo, seus efeitos e medidas preventivas.

1. *Adivinhar um ID*
   Ataque: O atacante introduz na rede um dispositivo que tenta se autenticar utilizando uma série de diferentes IDs escolhidos aleatoriamente ou com alguma heurística, até que encontre um valor válido.
   Se funcionar: Um nodo não-autorizado é implementado e tratado como confiável. Se o legítimo dono do ID tentar autenticar-se mais tarde, será tratado como um atacante e bloqueado (ver próximo ataque).
   Prevenção: Como é o *gateway* quem dispara a primeira mensagem do protocolo, ele tem controle sobre quantas tentativas de autenticação podem ser feitas em determinado tempo, então um atacante não pode tentar um número arbitrário de IDs. Além disso, qualquer nodo que seja detectado tentando autenticar-se mais do que poucas vezes pode ser marcado e ignorado. Mecanismos para este tipo de detecção podem envolver localização geográfica, e são apontados como trabalho futuro. Se a primeira tentativa for correta, este ataque é equivalente ao próximo.

2. *Conhecer um ID*
   Ataque: O atacante conhece um ID válido e insere na rede um dispositivo malicioso utilizando ele.
   Se funcionar: Um nodo não-autorizado é implementado e tratado como confiável. Se o legítimo dono do ID tentar autenticar-se mais tarde, será tratado como um atacante e bloqueado.
   Prevenção: O *gateway* tem condições de determinar se já existe um nodo autenticado por dado ID. Se um segundo nodo tentar usar o mesmo ID de outro nodo autenticado, o *gateway* bloqueia o que veio depois. Este ataque somente pode ter sucesso caso o atacante descubra o ID válido, insira e autentique o nodo antes que o nodo legítimo se autentique; então, o atacante deve não só encontrar o ID de algum modo, mas usá-lo numa pequena e específica janela de tempo (já que é o *gateway* quem inicia o processo de autenticação).
   Esta prevenção implica que um nodo sem memória persistente que seja reiniciado por algum motivo sem avisar o *gateway* não vai conseguir se autenticar novamente. Se isto for um problema, uma política de renovação de chaves pode ser implementada, que periodicamente invalida chaves e roda o protocolo de estabelecimento para gerar novas chaves. Deste modo um sensor reiniciado vai ficar inativo apenas até o próximo período de renovação de chaves, mas em contrapartida cada período destes abre uma nova oportunidade para ataques de conhecimento de ID.

3. *Capturar um nodo*
   Ataque: O atacante fisicamente obtém um nodo legítimo da rede.
   Se funcionar: Assume-se que o atacante tenha controle completo sobre o nodo e sua memória. O atacante poderia obter toda a informação criptográfica já validada e utilizá-la para efetivamente inserir um nodo malicioso na rede.
   Prevenção: Este ataque é, no caso geral, uma ameaça intrínseca a RSSF. Todavia, já que nenhum nodo possui conhecimento – explícito ou implícito – acerca de chaves de outros nodos, é garantido que a captura de um nodo não revela dados criptográficos privativos sobre o *gateway* (além da chave compartilhada com o sensor capturado) ou sobre qualquer outro nodo da rede.

5.1.9 Comunicação segura sensor-sensor

Em muitos cenários de RSSF a comunicação acontece exclusivamente entre sensores e *gateway*, e uma descrição exata de mecanismo para sensores se comunicarem entre si com segurança não é contemplada neste trabalho. Porém, pode-se imaginar um esquema em que o *gateway* funcione como gerador de certificados e dois sensores que desejam se comunicar peçam a ele uma chave de grupo temporária. Pode-se também pensar em uma hierarquização da topologia da rede, com alguns sensores atuando como "mini-*gateways*" para um grupo de sensores.

5.2 ANÁLISE DA IMPLEMENTAÇÃO

Esta seção analisa o desempenho do código escrito para implementar o protocolo no EPOS, permitindo a uma aplicação comunicar-se seguramente com certa transparência. No cenário de testes, um EPOSMoteII foi utilizado como *gateway* e outros dois como sensores, sendo um deles *slave* e outro *listener* no PTP. A camada inferior de rede implementava *duty cycling*, de modo que cada nodo da rede mantinha seu rádio desligado e o ligava apenas a cada 1 segundo para checar se havia alguma mensagem. Para enviar uma mensagem, o remetente enviava preâmbulos por 1.5s. Então, pelo grande *overhead* causado pela rede não foi medido o tempo total de uma rodada do protocolo. Porém, ele pode ser facilmente estimado para uma rede sem *overhead* somando os valores apresentados nesta seção.

Nos gráficos apresentados, cada barra representa a média aritmética de algumas (entre 2 e 13) iterações dos algoritmos e parâmetros correspondentes com diferentes valores e em diferentes EPOS-Motes. As curvas elípticas utilizadas são as recomendadas por (CERTICOM RESEARCH, 2000), e os números nos nomes (e.g. secp128r1) indicam os tamanhos em bits dos parâmetros.

5.2.1 Tamanho do código

A Tabela 3 mostra o tamanho das principais seções dos códigos gerados, incluindo as bibliotecas implementadas, a aplicação e todo o sistema operacional. Ao alterar os tamanhos de chave e ID o tamanho do código também altera-se ligeiramente, então os valores mostrados são médias dos tamanhos para os parâmetros variados. Para fins comparativos, a linha "PTP" representa um código de teste do PTP, sem o módulo de segurança.

Tabela 3 – Tamanho médio em bytes de diferentes seções dos códigos gerados.

Código	.text	.data	.bss	Comparação
PTP	47032	204	6140	100.00%
Sensores	65124	220	6156	133.96%
Gateway	68660	220	6156	138.67%

5.2.2 AES

A implementação em *hardware* utilizada do algoritmo AES se mostrou extremamente rápida. O processo de 1000 cifragens utilizando mensagens e chaves aleatórias de 16 bytes cada levou um total de $16288\mu s$, sendo o tempo mínimo para uma cifragem $16\mu s$, máximo $43\mu s$ e desvio padrão

amostral de apenas $2.20\mu s$.

5.2.3 Poly1305-AES

A Figura 11 apresenta o tempo tomado pela execução da implementação do algoritmo Poly1305-AES (que utiliza a biblioteca Bignum implementada neste trabalho) para derivações de chaves KT ou OTPs segundo as equações 4.2 e 4.3. O tempo total de checagem de OTP e preparação da mensagem de confirmação por parte do *gateway* é mostrado no gráfico da Figura 12. Este processo envolve duas derivações de OTP, uma comparação de dois OTPs, busca pelos dados do sensor e uma cifragem AES.

Em ambos os gráficos nota-se uma grande diferença no caso em que o ID e os parâmetros

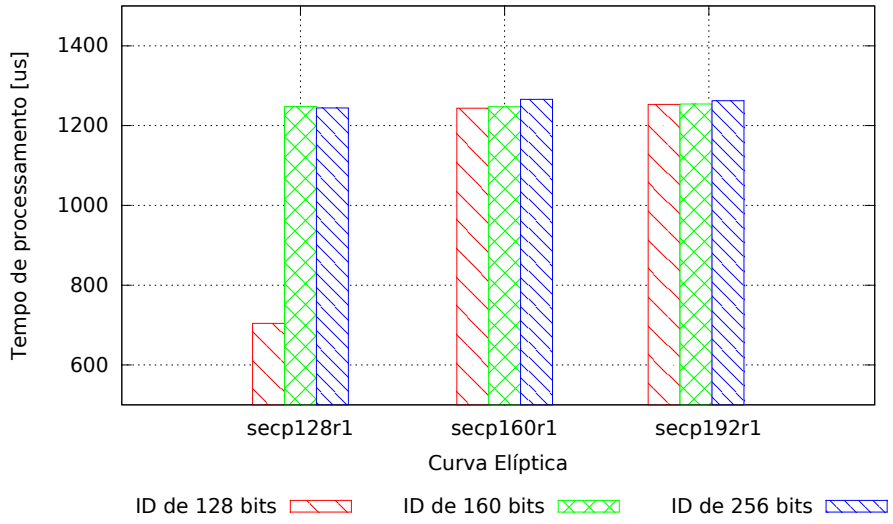

Figura 11 – Tempos médios (μs) de derivação de OTPs

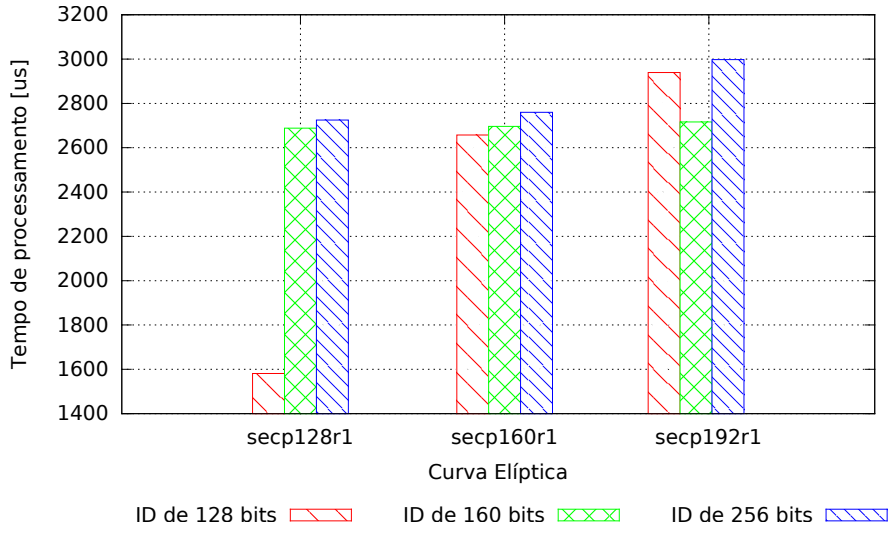

Figura 12 – Tempos médios (μs) de checagem e confirmação de autenticação

ECDH tem 128 bits. Isso acontece pois o Poly opera sobre blocos deste tamanho, e em todos os outros casos pelo menos um parâmetro é maior do que 128 bits. Como pode-se também notar nos gráficos, parâmetros com tamanhos dentro do mesmo múltiplo de 128 apresentam relativamente pouca diferença em termos de tempo de processamento.

5.2.4 ECDH

Os tempos de processamento mostrados até aqui são insignificantes perto do tempo tomado pela execução da implementação do ECDH. A Figura 13 mostra os tempos médios levados para realizar *uma* das duas multiplicações de pontos ECC do protocolo. Vale lembrar, porém, que enquanto Poly é utilizado a cada mensagem enviada e recebida, este par de multiplicações gera uma chave com validade muito longa (horas, dias, meses, dependendo da aplicação), portanto acontecem muito esporadicamente, ou até apenas uma vez durante todo o tempo de vida de um nodo na rede.

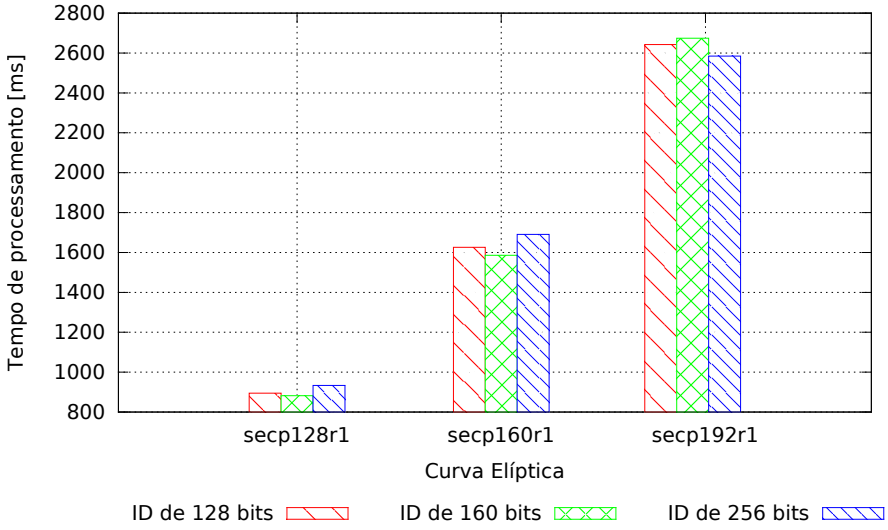

Figura 13 – Tempos médios (ms) de multiplicação de pontos ECDH

6 CONCLUSÃO

Este trabalho descreveu, implementou e analisou um protocolo para estabelecimento de chaves criptográficas e comunicação segura no contexto de Redes de Sensores Sem Fio e Internet das Coisas. A proposta foi implementada para o sistema operacional EPOS e plataforma EPOSMoteII na forma de uma camada de rede que permite à aplicação comunicar-se com os princípios de integridade, autenticidade e confidencialidade garantidos com certa transparência.

O objetivo de descrever e implementar o protocolo foi cumprido no capítulo 4, juntamente com o código desenvolvido. O segundo objetivo, de análise da segurança, foi cumprido com a primeira parte da discussão do capítulo 5. O último objetivo (análise de performance) foi abordado na segunda parte do capítulo 5. Embora consumo de energia não tenha sido analisado, pode-se argumentar que o tempo de processamento viável e o baixo *overhead* de comunicação exigido implica num impacto reduzido no consumo energético.

6.1 TRABALHOS FUTUROS

Alguns pontos foram identificados porém não abordados por este trabalho, e o estudo deles poderia se mostrar muito favorável ao protocolo aqui apresentado:

- Integração deste protocolo com o PTP e até outros protocolos, culminando numa pilha *cross-layer*.
- Integração de mecanismos de localização geográfica para detectar nodos maliciosos.
- Estudo de técnicas para otimizar a garantia de integridade de forma a reduzir o custo de comunicação extra gerado.
- Especificação da comunicação segura sensor-sensor.
- Consideração de questões de privacidade.
- Otimização das operações aritméticas utilizando algoritmos mais eficientes ou *hardware*.
- Estudo de viabilidade para topologias *multi-hop*, considerando ataques ao protocolo de roteamento.

REFERÊNCIAS

ATZORI, L.; IERA, A.; MORABITO, G. The internet of things: A survey. v. 54, p. 2787–2805, 2010.

BARRETT, P. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In: ODLYZKO, A. (Ed.). **Advances in Cryptology - CRYPTO' 86**. Springer Berlin Heidelberg, 1987, (Lecture Notes in Computer Science, v. 263). p. 311–323. ISBN 978-3-540-18047-0. Disponível em: <http://dx.doi.org/10.1007/3-540-47721-7_24>.

BERNSTEIN, D. J. The poly1305-aes message-authentication code. In: **Proceedings of Fast Software Encryption**. Paris, França: [s.n.], 2005. p. 32–49.

BONEH, D. The decision diffie-hellman problem. In: BUHLER, J. (Ed.). **Algorithmic Number Theory**. Springer Berlin Heidelberg, 1998, (Lecture Notes in Computer Science, v. 1423). p. 48–63. ISBN 978-3-540-64657-0. Disponível em: <http://dx.doi.org/10.1007/BFb0054851>.

BROWN, M. et al. Software implementation of the nist elliptic curves over prime fields. In: NACCACHE, D. (Ed.). **Topics in Cryptology - CT-RSA 2001**. [S.l.]: Springer Berlin Heidelberg, 2001, (Lecture Notes in Computer Science, v. 2020). p. 250–265. ISBN 978-3-540-41898-6.

CARLSEN, U. Generating formal cryptographic protocol specifications. In: **Research in Security and Privacy, 1994. Proceedings., 1994 IEEE Computer Society Symposium on**. [S.l.: s.n.], 1994. p. 137–146.

CERTICOM RESEARCH. **Standards for Efficient Cryptography (SEC) SEC 2: Recommended Elliptic Curve Domain Parameters**. [S.l.], September 2000.

DAEMEN, J.; RIJMEN, V. Aes proposal: Rijndael. p. 1, Set. 2003.

DIFFIE, W.; HELLMAN, M. New directions in cryptography. **IEEE Transactions on Information Theory**, v. 22, n. 6, p. 644–654, 1976.

ELKHODR, M.; SHAHRESTANI, S.; CHEUNG, H. The internet of things: Visions & challenges. In: **TENCON Spring Conference**. [S.l.: s.n.], 2013. p. 218 – 222.

FREESCALE. **MC1322x Advanced ZigBee^{TM}-Compliant SoC Platform for the 2.4 GHz IEEE®802.15.4 Standard Reference Manual**. [S.l.], Jan. 2012.

FRÖHLICH, A. A.; STEINER, R.; RUFINO, L. M. A trustful infrastructure for the internet of things based on eposmote. In: **9th IEEE International Conference on Dependable, Autonomic and Secure Computing**. Sydney, Australia: [s.n.], 2011. p. 63–68. ISBN 978-1-4673-0006-3.

FROHLICH, A. A. et al. A cross-layer approach to trustfulness in the internet of things. In: **9th Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS)**. Paderborn, Alemanha: [s.n.], 2013.

HUANG, Q. et al. Fast authenticated key establishment protocols for self-organizing sensor networks. In: **Proceedings of the 2Nd ACM International Conference on Wireless Sensor Networks and Applications**. New York, NY, USA: ACM, 2003. (WSNA '03), p. 141–150. ISBN 1-58113-764-8. Disponível em: <http://doi.acm.org/10.1145/941350.941371>.

IDALINO, T. B. **Utilizando dispositivos móveis na geração de senhas descartáveis**. 2012.

JINWALA, D. et al. Replay protection at the link layer security in wireless sensor networks. In: **Computer Science and Information Engineering, 2009 WRI World Congress on**. [S.l.: s.n.], 2009. v. 1, p. 160 –165.

KARLOF, C.; SASTRY, N.; WAGNER, D. Tinysec: a link layer security architecture for wireless sensor networks. In: **Proceedings of the 2nd international conference on Embedded networked sensor systems**. New York, NY, USA: ACM, 2004. (SenSys '04), p. 162–175. ISBN 1-58113-879-2. Disponível em: <http://doi.acm.org/10.1145/1031495.1031515>.

LI-PING, Z.; YI, W. An id-based key agreement protocol for wireless sensor networks. In: **1st International Conference on Information Science and Engineering (ICISE)**. [S.l.: s.n.], 2009. p. 2542 – 2545.

LISHA. **EPOS Project**. jun. 2014. Disponível em: <http://epos.lisha.ufsc.br>.

LUK, M. et al. Minisec: A secure sensor network communication architecture. In: **Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on**. [S.l.: s.n.], 2007. p. 479 –488.

MENEZES, A.; OORSCHOT, P. van; VANSTONE, S. **Handbook of Applied Cryptography**. [S.l.]: CRC Press, 1996. 599-606, 613-627 p.

NIST. Advanced encryption standard (aes). **Federal Information Processing Standards**, Publication 197, Nov. 2001.

NSA. **National Security Agency, Skipjack and KEA algorithm specifications**. May 1998. Disponível em: <http://cryptome.org/jya/skipjack-spec.htm>.

NSA. **The Case for Elliptic Curve Cryptography**. jan. 2009. Disponível em: <http://www.nsa.gov/business/programs/elliptic_curve.shtml>.

OLIVEIRA, P.; OKAZAKI, A. M.; FRÖHLICH, A. A. Sincronização de tempo a nível de so utilizando o protocolo ieee1588. In: **Simpósio Brasileiro de Engenharia de Sistemas Computacionais**. Natal, Brasil: [s.n.], 2012.

PADMAVATHY, R.; BHAGVATI, C. Methods to solve discrete logarithm problem for ephemeral keys. In: **Advances in Recent Technologies in Communication and Computing, 2009. ARTCom '09. International Conference on**. [S.l.: s.n.], 2009. p. 704–708.

PAN, J.; WANG, L.; MA, C. Analysis and improvement of an authenticated key exchange protocol. In: BAO, F.; WENG, J. (Ed.). **Information Security Practice and Experience**. Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 6672). p. 417–431. ISBN 978-3-642-21030-3. Disponível em: <http://dx.doi.org/10.1007/978-3-642-21031-0_31>.

RIAHI, A. et al. A systemic approach for iot security. In: **Distributed Computing in Sensor Systems (DCOSS), 2013 IEEE International Conference on**. Cambridge, MA: [s.n.], 2013. p. 351–355.

STALLINGS, W. **Cryptography And Network Security**. 5. ed. [S.l.]: Pearson, 2011.

STEINER, J. G.; NEUMAN, C.; SCHILLER, J. I. Kerberos: An authentication service for open network systems. In: **IN USENIX CONFERENCE PROCEEDINGS**. [S.l.: s.n.], 1988. p. 191–202.

STUDHOLME, C. **The Discrete Log Problem**. jun. 2013. Disponível em: <http://www.cs.toronto.edu/ cvs/dlog/>.

SUN, H.-M. et al. An authentication scheme balancing authenticity and transmission for wireless sensor networks. In: **Computer Symposium (ICS), 2010 International**. [S.l.: s.n.], 2010. p. 222 –227.

SUO, H. et al. Security in the internet of things: A review. In: **Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on**. [S.l.: s.n.], 2012. v. 3, p. 648–651.

APÊNDICE A – Conceitos Matemáticos

A.1 CONCEITOS DE ÁLGEBRA ABSTRATA

Corpos finitos tem um papel crucial em muitos algoritmos criptográficos – inclusive na definição da aritmética de Curvas Elípticas – e são elementos do ramo da matemática chamado álgebra abstrata, que se preocupa com conjuntos cujos elementos são passíveis de sofrer operações algébricas. Corpos finitos são definidos em termos de grupos, aneis e corpos. As definições deste apêndice são em sua maioria retiradas de (STALLINGS, 2011).

A.1.1 Grupos

Um **grupo** $\{C, \circ\}$ é definido por um conjunto de elementos C e uma operação $\circ$ que associa a cada par $(a,b) \in C$ um elemento $(a \circ b) \in C$; pode ser **finito** se contiver um número finito de elementos e **infinito** caso contrário. A **ordem** de um grupo finito é definida pelo número de elementos no grupo. Grupos são definidos também pelas seguintes propriedades:

(A1) Fechamento: $(a,b) \in C \rightarrow (a \circ b) \in C$.

(A2) Associatividade: $\forall a,b,c \in C : a \circ (b \circ c) = (a \circ b) \circ c$.

(A3) Identidade: $\forall a \in C, \exists e \in C : a \circ e = e \circ a = a$.

(A4) Inversa: $\forall a \in C, \exists a' \in C : a \circ a' = a' \circ a = e$.

Um **grupo abeliano** é um grupo que satisfaz a seguinte propriedade adicional:

(A5) Comutatividade: $\forall a,b \in C : a \circ b = b \circ a$.

No contexto de grupos, **exponenciação** é definida como a aplicação repetida do operador (e.g. $a^3 = a \circ a \circ a$), com as definições $a^0 = e$ e $a^{-b} = (a')^b$. Um grupo é dito **cíclico** se todo elemento de C for uma potência α^k, para um $\alpha \in C$ fixo, que é denominado **gerador** do grupo. Um grupo cíclico é sempre abeliano e pode ser finito ou infinito.

A.1.2 Aneis

Um **anel** $\{R, +, \times\}$ é um conjunto de elementos R e duas operações $+$ e $\times$, geralmente chamadas adição e multiplicação[1], respectivamente. As seguintes propriedades valem para aneis:

(A1-A5): $\{R, +\}$ apresenta todas as propriedades de um grupo.

(M1) Fechamento multiplicativo: $(a,b) \in R \rightarrow (ab) \in R$.

(M2) Associatividade multiplicativa: $\forall a,b,c \in R : a(bc) = (ab)c$.

(M3) Distributividade: $\forall a,b,c \in R : a(b+c) = ab + ac$,
$\forall a,b,c \in R : (a+b)c = ac + bc$.

Um **anel comutativo** apresenta a seguinte propriedade adicional:

(M4) Comutatividade multiplicativa: $\forall a,b \in R : ab = ba$.

E um **anel de integridade** apresenta ainda mais duas propriedades:

[1] O operador de multiplicação é comumente omitido, de modo que $ab = a \times b$.

(M5) Identidade multiplicativa: $\forall a \in R, \exists I \in C : aI = Ia = a$.

(M6) Ausência de divisores nulos: $a,b \in R \wedge ab = 0 \rightarrow a = 0 \vee b = 0$.

Um exemplo de anel de integridade é o conjunto de todos os inteiros com as operações de adição e multiplicação.

A.1.3 Corpos e Corpos Finitos Primos

Um **corpo** é um anel de integridade com a seguinte propriedade extra:

(M7) Inversa multiplicativa: $\forall a \in R \backslash \{0\}, \exists a^{-1} \in R : aa^{-1} = a^{-1}a = I$.

Ou seja, um corpo é um conjunto em que se pode fazer operações de adição, subtração, multiplicação e divisão sem sair do conjunto. Para um n inteiro, $\{\mathbb{Z}_n, +_n, \times_n\}$ é um anel comutativo[2], porém só admitirá inversa multiplicativa para elementos que forem primos relativos a n. Isso implica que este conjunto só é um corpo (finito) se n for um número primo. Este tipo de corpo será usado pela aritmética de curvas elípticas, e recebe o nome de corpo finito de ordem prima, ou simplesmente **corpo finito primo**, cuja ordem é n. Corpos finitos primos são denotados na literatura por F_p ou $GF(p)$ (de *Galois Field*, homenagem a Évariste Galois, tido como o primeiro matemático a estudar corpos finitos (STALLINGS, 2011)).

A.2 CURVAS ELÍPTICAS

Criptografia sobre Curvas Elípticas (ECC) utiliza-se de um tipo especial de curvas elípticas. Uma **curva elíptica** genérica é definida por uma equação específica de duas variáveis e coeficientes, mais uma operação de adição e multiplicação. Este trabalho utiliza-se de ECC sobre curvas elípticas primas, que operam sobre o conjunto $\mathbb{Z}_p$. **Curvas elípticas primas** são definidas pela equação:

$$y^2 \bmod p = (x^3 + ax + b) \bmod p \tag{A.1}$$

Definimos o conjunto $E_p(a,b)$ como todos os pares de inteiros $(x,y) \mid x,y \in \mathbb{Z}_p$ que satisfazem a equação A.1 para $a,b \in \mathbb{Z}_p$, p dados, mais um ponto O chamado **ponto no infinito**. É demonstrável (STALLINGS, 2011) que um grupo finito abeliano pode ser definido baseado no conjunto $E_p(a,b)$, dada a seguinte condição:

$$(4a^3 + 27b^2) \bmod p \neq 0 \bmod p \tag{A.2}$$

As regras que definem adição para tal grupo são as seguintes, para todos os pontos $P = (x_P, y_P), Q = (x_Q, y_Q) \in E_p(a,b)$:

1. $P + O = P$.

2. $P + (x_P, -y_P) = O$. Ou seja, $-P = (x_P, -y_P)$.

3. $P \neq -Q \rightarrow P + Q = (x_R, y_R)$:

$$x_R = (\lambda^2 - x_P - x_Q) \bmod p$$
$$y_R = (\lambda(x_P - x_R) - y_P) \bmod p$$

[2] $\mathbb{Z}_n$ é o conjunto dos inteiros pertencentes ao intervalo $[0, n-1]$, $+_n$ é a operação de adição módulo n e $\times_n$ a de multiplicação módulo n.

$$\lambda = \begin{cases} \frac{y_Q - y_P}{x_Q - x_P} \bmod p : P \neq Q \\ \\ \frac{3x_P^2 + a}{2y_P} \bmod p : P = Q \end{cases}$$

4. $\forall n \in \mathbb{Z}_p : nP = \underbrace{P + P + \ldots + P}_{n\text{-vezes}}$.

A **ordem** N do grupo $E_p(a,b)$ é proporcional a p, e é limitada por:

$$p + 1 - 2\sqrt{p} \leq N \leq p + 1 + 2\sqrt{p} \tag{A.3}$$

ANEXO A – Artigo

Estabelecimento de Chaves e Comunicação Segura para a Internet das Coisas

Davi Resner[1]

[1] Universidade Federal de Santa Catarina (UFSC)
Departamento de Informática e Estatística (INE)
Laboratório de Integração de Software e Hardware (LISHA)
Florianópolis, SC, Brasil

davir@lisha.ufsc.br

Abstract. *This work describes a protocol that provides a practical solution for the problem of cryptographic key establishment and secure communication in the context of Wireless Sensor Networks, in which computational efficiency is a fundamental requirement. Such protocol was implemented in the EPOS operating system and takes the form of a network layer that provides the principles of data integrity, authenticity and confidentiality. Tests were executed on the EPOSMoteII platform and the analysis of the results shows that the implementation is adequate to be used in the scenario of embedded systems with low proccessing power.*

Resumo. *Este trabalho descreve um protocolo que provê uma solução prática para o problema de estabelecimento de chaves criptográficas e comunicação segura no contexto de Redes de Sensores Sem Fio, no qual eficiência computacional é um requisito imprescindível. Tal protocolo foi implementado no sistema operacional EPOS na forma de uma camada de rede que provê os princípios de integridade, autenticidade e confidencialidade da informação. Os testes foram realizados na plataforma EPOSMoteII e a análise dos resultados indica que a implementação é adequada para o cenário de sistemas embarcados de baixo poder de processamento.*

1. Introdução

A ideia de Internet das Coisas (IdC, ou do inglês IoT) consiste em um interfaceamento de dispositivos da vida cotidiana com a internet. Significa geladeiras, televisões, ventiladores, lâmpadas e carros recebendo e enviando mensagens para o mundo. Pela grande quantidade com a qual serão empregados, os dispositivos utilizados para possibilitar tal comportamento devem ser de baixo custo e energeticamente eficientes; consequentemente, não terão o maior poder de processamento disponível no mercado. Uma rede de sensores sem fio (RSSF) é bastante similar a certos cenários de IdC: é uma rede composta por vários dispositivos capazes de coletar informações do ambiente e de comunicar-se sem o uso de fios.

Muitos protocolos comumente utilizados na internet tradicional (e.g. TCP/IP) funcionam bem para cenários em que a necessidade de eficiência de processamento e energia não seja tão crítica, mas no mundo embarcado da IdC e das RSSF muito pode ser feito

para melhorar o desempenho [Elkhodr et al. 2013].

Um requisito fundamental para um protocolo de comunicação na IdC é segurança. Em muitos cenários é importante que mensagens possam ser transmitidas de forma íntegra, confidencial e autêntica [Suo et al. 2012]. Ninguém gostaria, por exemplo, de ter sua casa controlada por estranhos sem permissão – falha de autenticidade –, ou que uma usina nuclear disparasse seus alarmes de catástrofe porque algumas mensagens com as informações reais reportadas pelos sensores foram manipuladas – falha de integridade. Há ainda aspectos de confidencialidade: mensagens privadas destinadas a um dispositivo devem apenas poder ser entendidas por aquele dispositivo.

É importante reconhecer que as soluções da internet convencional para tais problemas de segurança muitas vezes não são adequadas na Internet das Coisas, tornando assim esse um tema aberto de pesquisa [Elkhodr et al. 2013]. Além da constante restrição em termos de desempenho, podemos imaginar cenários em que algumas *coisas*, por exemplo, enviarão mensagens muito raramente e então se tornarão incomunicáveis por um longo período de tempo. É preciso saber o quanto se pode confiar nessa mensagem sem uma subsequente "conversa" entre os dispositivos [Frohlich et al. 2013].

Com estes problemas em mente, um novo protocolo para estabelecimento de chaves criptográficas e comunicação segura é proposto, especialmente criado para Redes de Sensores Sem Fio.

O restante deste texto está organizado da seguinte forma: a seção 2 apresenta trabalhos relacionados, ressaltando como este difere daqueles. A seção 3 apresenta alguns dos principais blocos utilizados como base para este protocolo, que é apresentado na seção 4. A seção seguinte discute alguns pontos importantes do protocolo, bem como mostra resultados de medições de tempo de processamento. Finalmente, a seção 7 conclui o texto e apresenta alguns possíveis trabalhos futuros.

2. Trabalhos Relacionados

TinySec [Karlof et al. 2004], segundo [Fröhlich et al. 2011], implementa uma arquitetura para comunicação segura em RSSF, a qual provê cifragem e autenticação de mensagens. A implementação é escrita em nesC (aproximadamente 3000 linhas de código) e é destinada às plataformas Mica, Mica2 e Mica2Dot, requerendo 7146 bytes de código e 728 bytes de memória RAM. TinySec pode ser utilizado de dois modos: *authenticated encryption* (cifragem mais autenticação) e *authentication only* (apenas autenticação). No primeiro modo, o *payload* é cifrado com o cifrador Skipjack [NSA 1998] e provê autenticação por meio de um MAC anexado à mensagem. No segundo modo, o MAC é anexado de modo similar, porém a mensagem não é cifrada. Apesar de possibilitar esta flexibilidade de modo de uso, a inclusão de um MAC na mensagem acarreta em um custo a mais de uso do rádio, e consequentemente de consumo energético. Ademais, o nível de segurança provido pelo MAC é proporcional ao seu tamanho [Sun et al. 2010], bem como o custo mencionado. A solução do TinySec para reduzir o consumo energético é reduzir também o nível de segurança, por meio de uma redução do tamanho do MAC. TinySec não apresenta uma solução contra ataques de *replay*.

MiniSec [Luk et al. 2007], segundo [Fröhlich et al. 2011], é uma estrutura similar ao TinySec que propõe solver os problemas deste. Isso é alcançado principalmente pelo uso de um modo de cifragem que provê confidencialidade e autenticação na mesma iteração. Este modo de cifragem requer que um Vetor de Inicialização (IV) seja compartilhado entre os participantes da comunicação, e MiniSec reduz o tráfego de rede com uma

técnica que permite transmitir apenas uma parte de tal Vetor sem prejudicar o nível de segurança. Ataques de *replay* são prevenidos por meio do uso de contadores sincronizados, mas também são mandados apenas alguns bits do contador por pacote, para reduzir o tráfego na rede. Porém, [Jinwala et al. 2009] aponta que uma rotina de ressincronização bastante custosa deve ser executada caso os contadores percam a sincronia (por entrega de pacotes fora de ordem, por exemplo).

Além destas estruturas, existem outras descrições de protocolos completos, incluindo uma eficiente proposta [Huang et al. 2003] baseada em Curvas Elípticas que utiliza a ideia de combinar operações de criptografia simétrica e assimétrica. Os autores propõem um esquema em que os servidores executem operações mais complexas para que os sensores sejam poupados algum trabalho. Os autores chamam esta estratégia de um esquema *híbrido*. [Pan et al. 2011] apresenta algumas falhas deste protocolo, mostrando que algumas das suas alegações quanto à segurança provida não são válidas. Tais falhas são corrigidas por uma proposta de alteração no protocolo.

Um protocolo baseado em IDs únicos dos sensores é apresentado em [Li-ping and Yi 2009] e uma análise superficial de custo de processamento e banda de rede indica que esta abordagem pode ser eficiente. Em todos esses trabalhos, entretanto, ou o sensor deve ter informação sensível e específica pré-carregada, ou assume-se que um terceiro agente distribuidor de certificados participe provendo chaves públicas certificadas através de um canal seguro alternativo. O protocolo proposto neste texto contrasta por focar na redução deste esforço pré-implementação da rede e por propor uma solução prática para estabelecimento de chaves.

3. Elementos Fundamentais

Esta seção apresenta os principais "blocos" em que este trabalho se baseou.

3.1. EPOS e EPOSMote

A implementação deste trabalho foi feita para o módulo para redes de sensores sem fio chamado EPOSMoteII [LISHA 2014] (figura 1), que é baseado na Platform-in-Package MC13224V [MCR 2012] da Freescale. O EPOSMote faz o papel de microcontrolador capaz de comunicação, que pode ser acoplado a sensores e atuadores diversos para controlá-los, ou ainda atuar como um nodo *stand-alone* da rede. Conta com um processador ARM7 de 24Mhz, 128Kbytes de memória flash, 80Kbytes de memória ROM e 96Kbytes de memória RAM. Este módulo também possui um rádio embutido conformante com a norma IEEE 802.15.4. O sistema operacional utilizado é o EPOS [LISHA 2014], e a linguagem, C++.

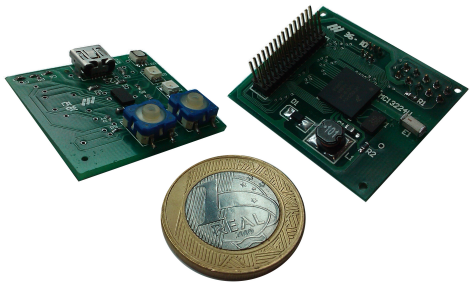

Figura 1. EPOSMoteII ao lado de uma moeda de R$1.

3.2. PTP

O protocolo de sincronização temporal implementado no EPOS [Oliveira et al. 2012], conformante com a norma IEEE 1588, permite que sensores sincronizem seus relógios com o de um nodo denominado *master*. O *master* troca mensagens específicas contendo o tempo local com nodos *slave*, que então as comparam com *seu* tempo local e, levando em conta também o tempo entre mensagens, conseguem calcular com grande precisão o *offset* entre os dois relógios, ajustando o seu de acordo. A Figura 2, retirada de [Frohlich et al. 2013], ilustra as mensagens trocadas pelo protocolo.

Já que essa troca de mensagens costuma ser feita através de *multicast*, a implementação do PTP do EPOS define nodos passivos denominados *listeners*, que ouvem as mensagens trocadas entre mestres e escravos e calculam seus próprios *offsets* com base nelas, sem inserir mais mensagens na rede.

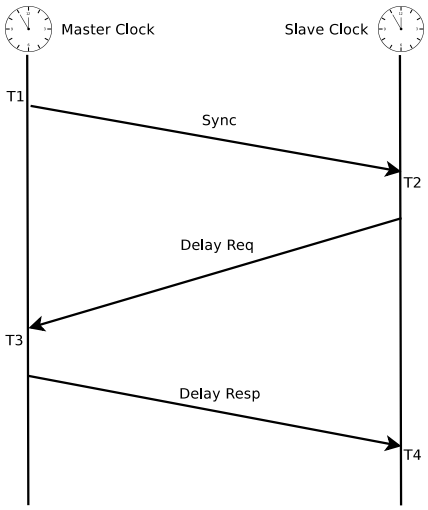

Figura 2. Mensagens trocadas pelo PTP [Frohlich et al. 2013]

3.3. AES em *hardware*

Advanced Encryption Standard é um cifrador simétrico considerado resistente contra ataques matemáticos. Consiste de um cifrador de bloco cujos blocos tem 128 bits, e as chaves podem assumir tamanhos de 128, 192 ou 256 bits.

O uso de aceleração em *hardware* não apenas melhora o desempenho de sistemas para segurança, como também permite que recursos permaneçam disponíveis para trabalho útil [Chang et al. 2010]. Assim, a implementação em *hardware* do AES disponível no EPOSMoteII é utilizada para cifrar e decifrar todos os dados necessários. A implementação utiliza um tamanho de chave fixo de 128 bits e é extremamente eficiente [Fröhlich et al. 2011].

3.4. Poly1305-AES

Poly1305-AES [Bernstein 2005] é um algoritmo gerador de MAC (*Message Authentication Code*) que utiliza-se do AES (aproveitando assim a aceleração de *hardware*) para calcular um código de 16 bytes de acordo com a equação 1, em que:

- c é de tamanho q, e é derivado a partir da mensagem de entrada,
- n é um *nonce*,
- k, r são duas chaves secretas, de 16 bytes cada,
- $AES_k(m)$ é o resultado produzido pela execução do AES utilizando a mensagem m e chave k como entrada.

$$mac = (((c_1 \cdot r^q + \cdots + c_q \cdot r^1) \bmod 2^{130} - 5) + AES_k(n)) \bmod 2^{128} \quad (1)$$

É provado que Poly1305-AES é tão seguro quanto o próprio AES [Bernstein 2005], e pode ser implementado com eficiência comparável.

3.5. Diffie-Hellman

O clássico algoritmo de Diffie-Hellman é utilizado neste protocolo. Mais especificamente, utiliza-se Diffie-Hellman sobre Curvas Elípticas Primas (F_p), considerado mais robusto e eficiente que a versão tradicional [NSA 2009]. A implementação foi altamente baseada nos trabalhos de [Menezes et al. 1996] e [Brown et al. 2001].

4. Protocolo Proposto

O protocolo é baseado no algoritmo Diffie-Hellman (mais especificamente, Diffie-Hellman sobre Curvas Elípticas), com alguns passos extras para garantir autenticação baseada em identificadores únicos conhecidos previamente. A seguinte notação é utilizada:

- ID_a: Identificador único do sensor a.
- (Y_a, P_a): Chaves Diffie-Hellman pública e privada, respectivamente, do sensor a.
- (Y_g, P_g): Chaves Diffie-Hellman do *gateway*.
- $DH(P_a, Y_b)$: Um algoritmo Diffie-Hellman, que calcula o *Master Secret* K_{ab}.
- T: A *timestamp* atual da rede, truncada para uma janela definível.
- $Auth_s$: Um *hash* sobre ID_s, por exemplo $AES(ID_s, ID_s)$.
- $\{M\}_k$: Uma mensagem M cifrada utilizando a chave k, e.g.: $AES(M, k)$.

4.1. Premissas

As seguintes condições são assumidas acerca do ambiente para que o protocolo possa funcionar corretamente. Algumas dessas premissas são discutidas na seção 5.

1. Cada sensor tem um identificador único na rede, chamado ID. Um ID não precisa ter um formato específico, pode ser, por exemplo, o número de série do microcontrolador.
2. Assume-se uma topologia de estrela da rede. Cada sensor consegue alcançar e ser alcançado por um nodo central, denominado *gateway*.
3. O *gateway* assume o papel de *master* no PTP, um sensor é designado *slave* e os demais operam em modo *listener*.
4. Comunicação segura é necessária apenas na comunicação entre sensor e *gateway*.
5. Um sensor autenticado não sairá da rede e voltará mais tarde para tentar se autenticar novamente.
6. O *gateway* é uma máquina local, segura e controlável, preferencialmente com maior poder de processamento que os sensores.

4.2. Inicialização

O ID de cada sensor confiável deve ser carregado manualmente no *gateway* antes do início da comunicação. O usuário é encarregado de fazer isto de maneira segura.

Quando um sensor é implementado, ele deve ouvir os parâmetros Diffie-Hellman públicos da rede (que podem alternativamente ser carregados em tempo de compilação, já que tais parâmetros são usualmente conhecidos então), gerar seu par de chaves Diffie-Hellman, sincronizar seu relógio através do PTP e então permanecer apenas ouvindo a rede até que o *gateway* envie uma mensagem $DH\_Request(Y_g)$, adentrando assim na próxima fase do protocolo.

4.3. Estabelecimento de Chaves

Esta fase consiste em uma transação Diffie-Hellman tradicional. O objetivo é estabelecer uma chave simétrica conhecida apenas pelos dois participantes, a qual só será utilizada se validada posteriormente (seção 4.4).

A primeira mensagem é enviada pelo *gateway*, e é denominada ***DH_Request(Y_g)***. A parte entre parênteses denota um valor que é enviado juntamente com a mensagem – neste caso Y_g. Esta mensagem pode ser enviada por *multicast* ou para um nodo em específico.

Assim que uma mensagem ***DH_Request(Y_g)*** chega ao sensor de destino s, este primeiramente responde com ***DH_Response(Y_s)*** e então calcula a chave simétrica $K_{sg} = P_s * Y_g$, a qual será denominada *Master Secret*. Similarmente, o *gateway* recebe ***DH_response(Y_s)*** e calcula $K_{sg} = P_g * Y_s$.

Agora, sensor e *gateway* compartilham de uma chave, porém como o processo ainda não incluiu nenhum aspecto de autenticação, nenhuma das partes tem nenhuma garantia de que compartilha uma chave com um nodo confiável e não-malicioso da rede. É este problema que a próxima fase do protocolo visa solucionar.

4.4. Autenticação

O *Master Secret* estabelecido na fase anterior deve ser validado antes de poder ser usado. Para autenticar uma chave, o sensor envia para o *gateway* a mensagem ***Auth_Request($Auth_s, OTP_s$)***, sendo:

$$OTP_s = Poly(K_{sg} \oplus ID_s, K_{sg}, ID_s, T) \tag{2}$$

Ao receber uma mensagem ***Auth_Request($Auth_s, OTP_s$)*** esperada, o *gateway* deve verificar se existe uma chave esperando autenticação que esteja implícita em OTP_s. O processo começa com uma busca em seu banco de dados, através de $Auth_s$, pelo ID_s correspondente. Então, para cada chave K_{ag} pendendo autenticação, é calculado $OTP_a = Poly(K_{ag} \oplus ID_s, K_{ag}, ID_s, T)$ e checado se $OTP_a = OTP_s$. Caso nenhum OTP satisfaça a igualdade, o *gateway* não responde. Caso OTP_s seja encontrado, o *gateway* tem evidência de que a chave K_{ag} é na verdade K_{sg} e foi de fato compartilhada com o sensor autêntico identificado por ID_s, e associa estes dois valores para comunicação futura. O *gateway* então envia a última mensagem: ***Auth_Granted($\{ID_s\}_{KT_{sg}}$)*** para prover evidência ao sensor de que este nodo é de fato um *gateway* autêntico, que conhece ID_s. A chave KT_{sg} é uma chave derivada de K_{sg} com o mecanismo explicado na seção a seguir.

Ao receber *Auth_Granted*($\{ID_s\}_{KT_{sg}}$), o sensor tem condições de derivar KT_{sg} para decifrar a mensagem e checar se ID_s está correto. Caso positivo, o sensor passa a confiar no *Master Secret* K_{sg}. Caso contrário, o sensor não deve se considerar autenticado e deve começar novamente o protocolo, esperando por uma nova mensagem *DH_Request*.

4.5. Comunicação Segura

Uma vez que o *Master Secret* K_{sg} foi autenticado, ele está pronto para ser utilizado como chave para cifragem AES, porém não diretamente. Quando um sensor precisa enviar uma mensagem M segura para o *gateway* (e vice-versa), ele deve derivar a chave KT_{sg} segundo a equação 3, adicionar à mensagem um *checksum* C e então enviar a mensagem *Secure_Message*($\{CM\}_{KT_{sg}}$).

$$KT_{sg} = Poly(K_{sg} \oplus ID_s, K_{sg}, ID_s, T) \tag{3}$$

Se o destinatário é um sensor, este deriva KT_{sg}, decifra a mensagem para obter CM e checa se o *checksum* está correto (se não estiver, ignora-se a mensagem). Caso o destinatário seja o *gateway*, ele deve derivar uma KT_{ag} para todos os sensores a autenticados, e então tentar decifrar a mensagem com cada chave até que uma delas gere o *checksum* corretamente (se nenhuma gerar, a mensagem é descartada). Caso a implementação permita acesso a um endereço de uma camada inferior da rede, pode-se usá-lo como heurística para acelerar essa busca por K_{sg}.

Caso C seja gerado corretamente, o destinatário tem evidência para acreditar que a mensagem M foi enviada pelo único outro nodo da rede que conhece o *Master Secret* já validado, e então confia que a mensagem é autêntica, confidencial, íntegra e foi enviada dentro da janela de tempo atual.

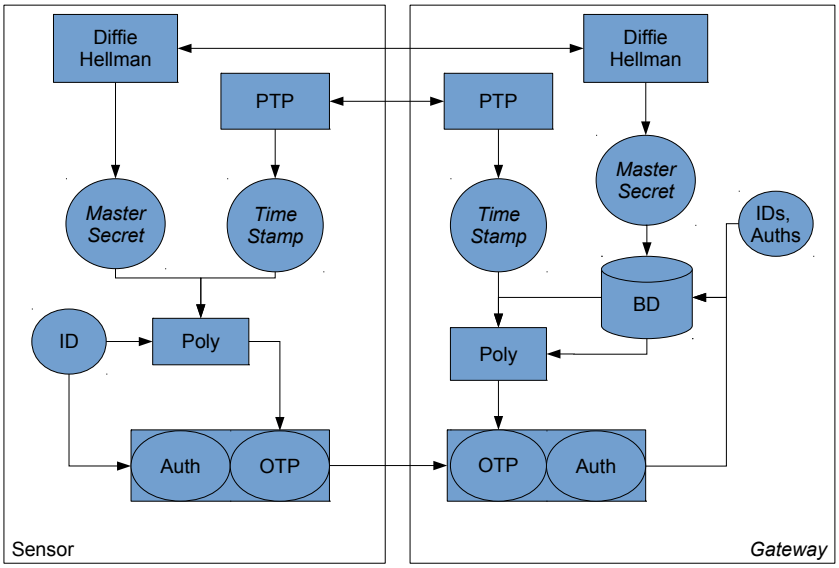

Figura 3. Visão geral da interação entre vários algoritmos no protocolo

5. Avaliação

Esta seção tem o objetivo de analisar a viabilidade e eficiência do protocolo proposto. A análise é dividida em duas partes: a seção 5.1 traz uma discussão de caráter qualitativo acerca da segurança do protocolo, suas principais características positivas e limitações. A seção seguinte, 6, apresenta uma descrição de testes executados sobre uma implementação do protocolo, bem como os dados levantados e análise dos mesmos.

5.1. Avaliação da Proposta

5.2. Princípios de segurança providos

Se uma mensagem segura $\{CM\}_{KT_{sg}}$ consegue ser decifrada pelo destinatário de modo a gerar C corretamente, significa que:

- M é **confidencial**, pois apenas s e g conhecem K_{sg}, e consequentemente KT_{sg}.
- M é **autêntica**, pois só s e g conhecem ambos K_{sg} e ID_s, utilizados para gerar KT_{sg}.
- M é **íntegra**, pois é extremamente improvável que $\{CM\}_{KT_{sg}}$ seja alterada cegamente de forma que o *checksum* resultante continue válido.
- $\{CM\}_{KT_{sg}}$ foi gerada na *timestamp* atual da rede.

5.3. IDs

O identificador único de cada sensor não precisa ter uma forma específica, porém algumas recomendações impactam o nível de segurança provido. Um atacante pode se passar por um sensor autêntico caso descubra um ID válido e o use antes do sensor autêntico correspondente àquele ID (ver seção 5.6), então o mais desejável é que seja muito difícil um atacante descobrir um ID válido. Para isso, 3 fatores influentes são o sigilo com o qual os IDs são guardados, o tamanho e a previsibilidade deles.

Sigilo é uma questão que fica a cargo do usuário, pois muito depende do contexto específico no qual a rede está inserida, incluindo fatores humanos e logísticos como a terceirização da equipe que gera e instala os IDs. Idealmente, os IDs devem ser mantidos em sigilo e revelados apenas a quem precisa sabê-los.

O tamanho do ID está diretamente relacionado ao nível de segurança. Quanto maior o ID, mais difícil será para um atacante adivinhá-lo cegamente. Os códigos *Auth* e OTP se tornam mais seguros pelo aumento do tamanho do texto plano se utilizado um modo de cifragem do AES que aproveite todo esse tamanho (e.g. modo CBC). Um ID menor do que 128 bits é altamente desencorajado, pois afetaria os tamanhos de valores utilizados como chave em *Auth* e OTP, o que diminuiria a segurança do AES na derivação deles.

Previsibilidade é uma questão importante pois um atacante que tenha uma ideia do formato dos IDs pode muito mais facilmente encontrá-los através de tentativa e erro. Assim, aconselha-se que os IDs sejam, na medida do possível, aleatórios.

Em suma, de modo ideal recomenda-se que os IDs sejam números aleatórios de pelo menos 128 bits, os quais são revelados apenas para quem é incumbido de carregá-los no *gateway*. Caso este cenário não seja viável, deve-se pensar nesses três fatores na medida do possível. Se deseja-se usar o número de série não-secreto de 10 bits de cada sensor como ID, por exemplo, pode-se ao invés disso utilizá-los como semente de uma função secreta geradora de números pseudo-aleatórios de maior magnitude e usar o

resultado como ID.

Com essas recomendações em mente, pode-se pensar que seria muito mais fácil e eficiente simplesmente carregar chaves simétricas pré-estabelecidas, ou utilizar o próprio ID diretamente como chave. Nessa estratégia, porém, a descoberta de um ID imediatamente permite a um atacante decifrar *todas* as mensagens – passadas e futuras – que o sensor correspondente trocar, enquanto no protocolo proposto o atacante deve, além de descobrir o ID, solucionar o problema do logaritmo discreto para curvas elípticas, o que por si só é um problema nada trivial. Por não derivar chaves exclusivamente a partir do ID, o protocolo proposto provê segurança mesmo que eles sejam empregados descuidadosamente (ver seção 5.6), permitindo assim um certo relaxamento quanto aos requerimentos acerca do ID, enquanto usá-los diretamente como chave requereria um cuidado tão grande quanto a segurança desejada.

5.4. *Timestamps*

As *timestamps* utilizadas no protocolo não são o valor do relógio diretamente. Ele deve ser arredondado para um múltiplo de um período de tempo pré-definido para a rede, que definirá a *granularidade* da *timestamp*. Deve-se aqui haver um ponto de contato com as camadas inferiores da pilha de comunicação, pois a *timestamp* utilizada para os cálculos deve refletir o momento em que a mensagem é de fato enviada. Em redes que implementam *duty cycling*, por exemplo, um remetente deve enviar mensagens de preâmbulo por um tempo X antes de efetivamente enviar a mensagem desejada, então a *timestamp* utilizada para cifragem de uma mensagem a ser enviada neste cenário deve ser T_{i+X}.

Deve-se assegurar que dentro de uma *timestamp* uma mensagem consiga viajar do remetente até o destinatário. Não se deve utilizar *timestamps* que mudem com periodicidade muito baixa, pois ataques de *replay* são possíveis dentro de uma mesma *timestamp*.

5.5. PTP inseguro

Já que o protocolo requer relógios sincronizados como pré-condição e o PTP é utilizado para garantir essa sincronia, as trocas de mensagens PTP devem acontecer de modo não-seguro, ao menos enquanto houver sensores não-autenticados. Isso permite que atacantes se passem por *masters* ou *slaves* e impeçam que sensores autênticos sincronizem seus relógios corretamente. Entretanto, as *timestamps* geradas para uso no protocolo são usadas apenas como mecanismo de proteção contra *replays*, então um ataque ao PTP resultaria apenas em negação de serviço, sem qualquer informação sigilosa comprometida ou autenticação manipulada. Vulnerabilidade a negação de serviço é um problema (no caso geral) intrínseco a redes *wireless*, pois um atacante pode simplesmente injetar muitas mensagens na mesma faixa de frequência utilizada, impedindo assim *qualquer* comunicação. Este ataque é conhecido como *jamming* de canal.

5.6. Possíveis ataques

Nesta seção são considerados alguns outros possíveis ataques ao protocolo, seus efeitos e medidas preventivas.

1. *Adivinhar um ID*
   Ataque: O atacante introduz na rede um dispositivo que tenta se autenticar utilizando uma série de diferentes IDs escolhidos aleatoriamente ou com alguma

heurística, até que encontre um valor válido.
Se funcionar: Um nodo não-autorizado é implementado e tratado como confiável. Se o legítimo dono do ID tentar autenticar-se mais tarde, será tratado como um atacante e bloqueado (ver próximo ataque).
Prevenção: Como é o *gateway* quem dispara a primeira mensagem do protocolo, ele tem controle sobre quantas tentativas de autenticação podem ser feitas em determinado tempo, então um atacante não pode tentar um número arbitrário de IDs. Além disso, qualquer nodo que seja detectado tentando autenticar-se mais do que poucas vezes pode ser marcado e ignorado. Mecanismos para este tipo de detecção podem envolver localização geográfica, e são apontados como trabalho futuro. Se a primeira tentativa for correta, este ataque é equivalente ao próximo.

2. *Conhecer um ID*
   Ataque: O atacante conhece um ID válido e insere na rede um dispositivo malicioso utilizando ele.
   Se funcionar: Um nodo não-autorizado é implementado e tratado como confiável. Se o legítimo dono do ID tentar autenticar-se mais tarde, será tratado como um atacante e bloqueado.
   Prevenção: O *gateway* tem condições de determinar se já existe um nodo autenticado por dado ID. Se um segundo nodo tentar usar o mesmo ID de outro nodo autenticado, o *gateway* bloqueia o que veio depois. Este ataque somente pode ter sucesso caso o atacante descubra o ID válido, insira e autentique o nodo antes que o nodo legítimo se autentique; então, o atacante deve não só encontrar o ID de algum modo, mas usá-lo numa pequena e específica janela de tempo (já que é o *gateway* quem inicia o processo de autenticação).
   Esta prevenção implica que um nodo sem memória persistente que seja reiniciado por algum motivo sem avisar o *gateway* não vai conseguir se autenticar novamente. Se isto for um problema, uma política de renovação de chaves pode ser implementada, que periodicamente invalida chaves e roda o protocolo de estabelecimento para gerar novas chaves. Deste modo um sensor reiniciado vai ficar inativo apenas até o próximo período de renovação de chaves, mas em contrapartida cada período destes abre uma nova oportunidade para ataques de conhecimento de ID.

3. *Capturar um nodo*
   Ataque: O atacante fisicamente obtém um nodo legítimo da rede.
   Se funcionar: Assume-se que o atacante tenha controle completo sobre o nodo e sua memória. O atacante poderia obter toda a informação criptográfica já validada e utilizá-la para efetivamente inserir um nodo malicioso na rede.
   Prevenção: Este ataque é, no caso geral, uma ameaça intrínseca a RSSF. Todavia, já que nenhum nodo possui conhecimento – explícito ou implícito – acerca de chaves de outros nodos, é garantido que a captura de um nodo não revela dados criptográficos privativos sobre o *gateway* (além da chave compartilhada com o sensor capturado) ou sobre qualquer outro nodo da rede.

5.7. Comunicação segura sensor-sensor

Em muitos cenários de RSSF a comunicação acontece exclusivamente entre sensores e *gateway*, e uma descrição exata de mecanismo para sensores se comunicarem entre si com segurança não é contemplada neste trabalho. Porém, pode-se imaginar um esquema em que o *gateway* funcione como gerador de certificados e dois sensores que desejam se comunicar peçam a ele uma chave de grupo temporária. Pode-se também pensar em uma hierarquização da topologia da rede, com alguns sensores atuando como "mini-*gateways*" para um grupo de sensores.

6. Análise da Implementação

Esta seção analisa o desempenho do código escrito para implementar o protocolo no EPOS, permitindo a uma aplicação comunicar-se seguramente com certa transparência. No cenário de testes, um EPOSMoteII foi utilizado como *gateway* e outros dois como sensores, sendo um deles *slave* e outro *listener* no PTP. A camada inferior de rede implementava *duty cycling*, de modo que cada nodo da rede mantinha seu rádio desligado e o ligava apenas a cada 1 segundo para checar se havia alguma mensagem. Para enviar uma mensagem, o remetente enviava preâmbulos por 1.5s. Então, pelo grande *overhead* causado pela rede não foi medido o tempo total de uma rodada do protocolo. Porém, ele pode ser facilmente estimado para uma rede sem *overhead* somando os valores apresentados nesta seção.

Nos gráficos apresentados, cada barra representa a média aritmética de algumas (entre 2 e 13) iterações dos algoritmos e parâmetros correspondentes com diferentes valores e em diferentes EPOSMotes. As curvas elípticas utilizadas são as recomendadas por [SEC 2000], e os números nos nomes (e.g. secp128r1) indicam os tamanhos em bits dos parâmetros.

6.1. Tamanho do código

A Tabela 1 mostra o tamanho das principais seções dos códigos gerados, incluindo as bibliotecas implementadas, a aplicação e todo o sistema operacional. Ao alterar os tamanhos de chave e ID o tamanho do código também altera-se ligeiramente, então os valores mostrados são médias dos tamanhos para os parâmetros variados. Para fins comparativos, a linha "PTP" representa um código de teste do PTP, sem o módulo de segurança.

Código	.text	.data	.bss	Comparação
PTP	47032	204	6140	100.00%
Sensores	65124	220	6156	133.96%
Gateway	68660	220	6156	138.67%

Tabela 1. Tamanho médio em bytes de diferentes seções dos códigos gerados.

6.2. Poly1305-AES

A Figura 4 apresenta o tempo tomado pela execução da implementação do algoritmo Poly1305-AES para derivações de chaves KT ou OTPs segundo a equação 2. O tempo

total de checagem de OTP e preparação da mensagem de confirmação por parte do *gateway* é mostrado no gráfico da Figura 5. Este processo envolve duas derivações de OTP, uma comparação de dois OTPs, busca pelos dados do sensor e uma cifragem AES.

Em ambos os gráficos nota-se uma grande diferença no caso em que o ID e os

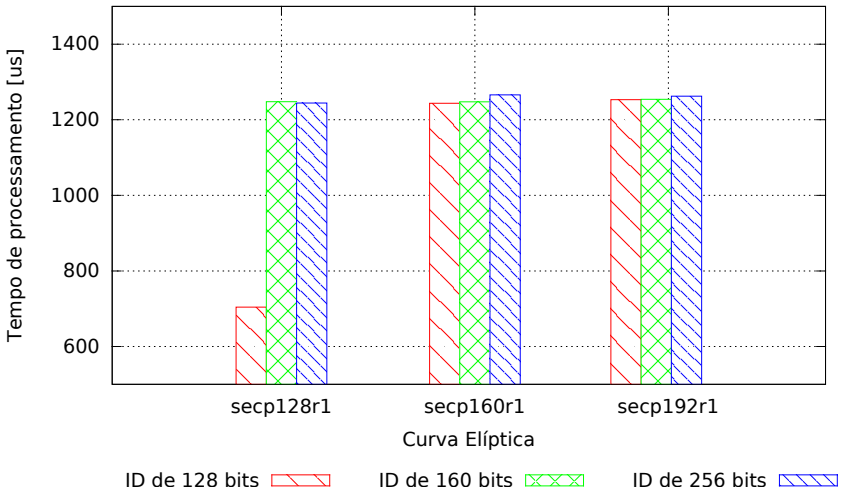

Figura 4. Tempos médios (μs) de derivação de OTPs

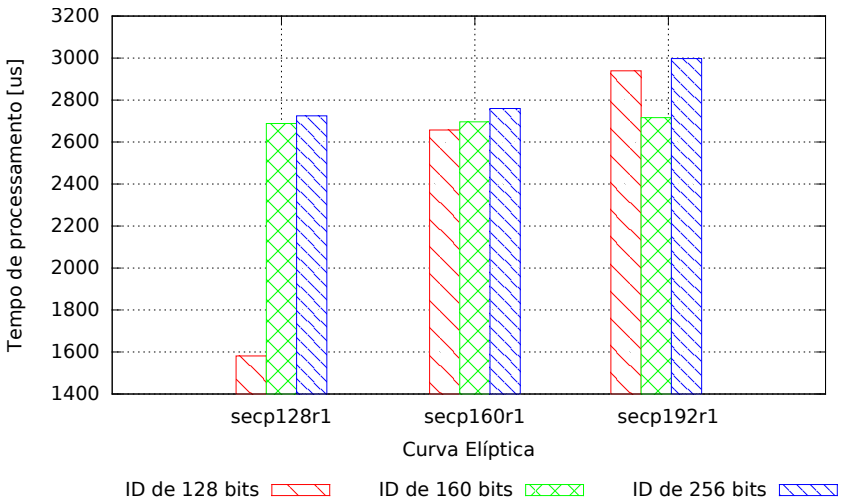

Figura 5. Tempos médios (μs) de checagem e confirmação de autenticação

parâmetros ECDH tem 128 bits. Isso acontece pois o Poly opera sobre blocos deste tamanho, e em todos os outros casos pelo menos um parâmetro é maior do que 128 bits. Como pode-se também notar nos gráficos, parâmetros com tamanhos dentro do mesmo múltiplo de 128 apresentam relativamente pouca diferença em termos de tempo de processamento.

6.3. Diffie-Hellman sobre Curvas Elípticas (ECDH)

Os tempos de processamento mostrados até aqui são insignificantes perto do tempo tomado pela execução da implementação do ECDH. A Figura 6 mostra os tempos médios

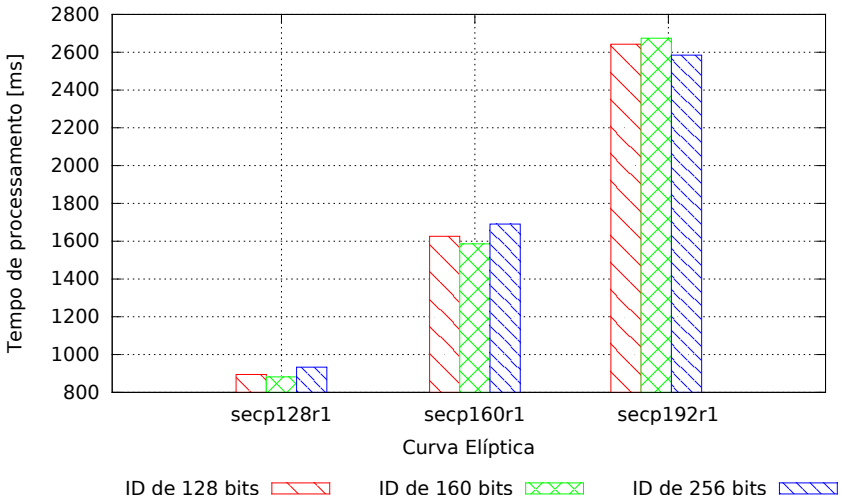

Figura 6. Tempos médios (ms) de multiplicação de pontos ECDH

levados para realizar *uma* das duas multiplicações de pontos de curva elíptica do protocolo. Vale lembrar, porém, que enquanto Poly é utilizado a cada mensagem enviada e recebida, este par de multiplicações gera uma chave com validade muito longa (horas, dias, meses, dependendo da aplicação), portanto acontecem muito esporadicamente, ou até apenas uma vez durante todo o tempo de vida de um nodo na rede.

7. Conclusão

Este trabalho descreveu, implementou e analisou um protocolo para estabelecimento de chaves criptográficas e comunicação segura no contexto de Redes de Sensores Sem Fio e Internet das Coisas. A proposta foi implementada para o sistema operacional EPOS e plataforma EPOSMoteII na forma de uma camada de rede que permite à aplicação comunicar-se com os princípios de integridade, autenticidade e confidencialidade garantidos com certa transparência.

7.1. Trabalhos Futuros

Alguns pontos foram identificados porém não abordados por este trabalho, e o estudo deles poderia se mostrar muito favorável ao protocolo aqui apresentado:

- Integração deste protocolo com o PTP e até outros protocolos, culminando numa pilha *cross-layer*.
- Integração de mecanismos de localização geográfica para detectar nodos maliciosos.
- Estudo de técnicas para otimizar a garantia de integridade de forma a reduzir o custo de comunicação extra gerado.
- Especificação da comunicação segura sensor-sensor.
- Consideração de questões de privacidade.
- Otimização das operações aritméticas utilizando algoritmos mais eficientes ou *hardware*.
- Estudo de viabilidade para topologias *multi-hop*, considerando ataques ao protocolo de roteamento.

Referências

(2000). *Standards for Efficient Cryptography (SEC) SEC 2: Recommended Elliptic Curve Domain Parameters*. Certicom Research.

(2012). *MC1322x Advanced ZigBee™-Compliant SoC Platform for the 2.4 GHz IEEE®802.15.4 Standard Reference Manual*. Freescale.

Bernstein, D. J. (2005). The poly1305-aes message-authentication code. In *Proceedings of Fast Software Encryption*, pages 32–49, Paris, França.

Brown, M., Hankerson, D., López, J., and Menezes, A. (2001). Software implementation of the nist elliptic curves over prime fields. In Naccache, D., editor, *Topics in Cryptology - CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 250–265. Springer Berlin Heidelberg.

Chang, J.-T., Liu, S., Gaudiot, J., and Liu, C. (2010). Hardware-assisted security mechanism: The acceleration of cryptographic operations with low hardware cost. In *Performance Computing and Communications Conference (IPCCC), 2010 IEEE 29th International*, pages 327 –328.

Elkhodr, M., Shahrestani, S., and Cheung, H. (2013). The internet of things: Visions & challenges. In *TENCON Spring Conference*, pages 218 – 222.

Fröhlich, A. A., Steiner, R., and Rufino, L. M. (2011). A trustful infrastructure for the internet of things based on eposmote. In *9th IEEE International Conference on Dependable, Autonomic and Secure Computing*, pages 63–68, Sydney, Australia.

Frohlich, A. A., Okazaki, A. M., Steiner, R. V., Oliveira, P., and Martina, J. E. (2013). A cross-layer approach to trustfulness in the internet of things. In *9th Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, Paderborn, Alemanha.

Huang, Q., Cukier, J., Kobayashi, H., Liu, B., and Zhang, J. (2003). Fast authenticated key establishment protocols for self-organizing sensor networks. In *Proceedings of the 2Nd ACM International Conference on Wireless Sensor Networks and Applications*, WSNA '03, pages 141–150, New York, NY, USA. ACM.

Jinwala, D., Patel, D., Patel, S., and Dasgupta, K. (2009). Replay protection at the link layer security in wireless sensor networks. In *Computer Science and Information Engineering, 2009 WRI World Congress on*, volume 1, pages 160 –165.

Karlof, C., Sastry, N., and Wagner, D. (2004). Tinysec: a link layer security architecture for wireless sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 162–175, New York, NY, USA. ACM.

Li-ping, Z. and Yi, W. (2009). An id-based key agreement protocol for wireless sensor networks. In *1st International Conference on Information Science and Engineering (ICISE)*, pages 2542 – 2545.

LISHA (2014). Epos project. `http://epos.lisha.ufsc.br`.

Luk, M., Mezzour, G., Perrig, A., and Gligor, V. (2007). Minisec: A secure sensor network communication architecture. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 479 –488.

Menezes, A., van Oorschot, P., and Vanstone, S. (1996). *Handbook of Applied Cryptography*. CRC Press.

NSA (1998). National security agency, skipjack and kea algorithm specifications. `http://cryptome.org/jya/skipjack-spec.htm`.

NSA (2009). The case for elliptic curve cryptography. `http://www.nsa.gov/business/programs/elliptic\_curve.shtml`.

Oliveira, P., Okazaki, A. M., and Fröhlich, A. A. (2012). Sincronização de tempo a nível de so utilizando o protocolo ieee1588. In *Simpósio Brasileiro de Engenharia de Sistemas Computacionais*, Natal, Brasil.

Pan, J., Wang, L., and Ma, C. (2011). Analysis and improvement of an authenticated key exchange protocol. In Bao, F. and Weng, J., editors, *Information Security Practice and Experience*, volume 6672 of *Lecture Notes in Computer Science*, pages 417–431. Springer Berlin Heidelberg.

Sun, H.-M., Chang, S.-Y., Tello, A., and Chen, Y.-H. (2010). An authentication scheme balancing authenticity and transmission for wireless sensor networks. In *Computer Symposium (ICS), 2010 International*, pages 222 –227.

Suo, H., Wan, J., Zou, C., and Liu, J. (2012). Security in the internet of things: A review. In *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*, volume 3, pages 648–651.

ANEXO B – Documentação (em inglês) da implementação

EPOS' Secure Communication Support

Davi Resner
Software/Hardware Integration Lab
Federal University of Santa Catarina
davir@lisha.ufsc.br

May 30, 2014

Abstract

Communication security is an important concern in Wireless Sensor Networks. In many scenarios, there is the need to make sure that exchanged messages have the properties of authentication, confidentiality and integrity. This report shows how to use EPOS' secure communication classes to enable sensors to establish a symmetric key with a central server and then use this key to communicate with the aforementioned principles granted transparently. Each parameter comes followed by the filename and line number in which it is set. This document is meant to go along with the secure_nic_master.cc and secure_nic_slave.cc applications.

Keywords: Internet of Things; Wireless Sensor Networks; Trustfulness; Security; Key Establishment; Confidentiality; Authentication; Integrity

1 Overview

EPOS' Secure Key Bootstrapping and Authentication protocol is based on the Elliptic Curve version of Diffie-Hellman. It enables sensors to establish a symmetric key with a trusted server and used it to encrypt messages and communicate securely. The protocol relies on unique sensor IDs known beforehand only by the server and the corresponding sensor.

AES is used as an encryption engine; PTP [1] is used to synchronize clocks and thus be able to use timestamps to prevent replay attacks; Poly1305-AES [2] is the One-Time-Password generation engine.

To use the protocol, a number of parameters need to be set the same on each node, which are explained throughout this document.

For a good yet not dense explanation of the Diffie-Hellman protocol and Elliptic Curves, refer to [3]. The implementation of the required arithmetic was greatly based on [4] [5]. EPOS' PTP implementation is explained in [1]. A little information about the protocol explained here can be found at [6].

2 Machine-specific parameters

You can tweak these two parameters to optimize the implementation to a particular machine. If your machine has a 32-bit architecture, you shouldn't need

to change these.

Bignum base (or digit size): include/traits.h:230
The base in which the bignums are represented. For best performance, this should be set to the native processor's word size (usually unsigned int). The digit size is defined as the size of the type of the base (e.g. sizeof unsigned int). Always use an unsigned type.

Double-digit: include/traits.h:231
A type of size two times larger than the one defined as digit. Should also be unsigned.

3 Network Parameters

These parameters are public and define the elliptic curve to be used by the Diffie-Hellman algorithm for this specific network. These values should be set equally in every node to enter the network. You may use the values provided as comments in the model applications without a problem.

Modulo (or Prime):
app/secure_nic_master.cc:30
app/secure_nic_slave.cc:31
This should be a big prime number. It is written in little-endian, digit-by-digit format. In general, the bigger this number is, the more secure the network becomes, but the complexity of the Bignum operations grow accordingly. You can use SECV [7] recommendations of elliptic curve parameters.

Bignum size: include/traits.h:232
The size in digits of each bignum. This should be the same size as the defined modulo. The bigger this value is, the more complex each bignum operation gets, but the network security increases accordingly. NIST suggests [8] that, to provide the same security as a 128-bit symmetric key, a 256-bit prime should be used. You can adjust this value to your processing constraints.

Barrett μ:
app/secure_nic_master.cc:31
app/secure_nic_slave.cc:32
The modular reduction after multiplication is implemented with the Barrett Reduction [4] algorithm. This is a constant used by the method to accelerate reduction, and is defined as:

$$\mu = \lfloor b^{2*k}/p \rfloor \tag{1}$$

Where b is the base (usually 2^{32}), k is the size in digits of the prime, and p is the prime (defined in **Modulo**). μ will always be one digit larger than the prime. It is written in little-endian, digit-by-digit format.

Base point:
app/secure_nic_master.cc:28,29
app/secure_nic_slave.cc:29,30

A little-endian, byte-by-byte representation of the elliptic curve point used as Base Point by Diffie-Hellman. You can use SECV [7] recommendations of elliptic curve parameters.

Time window: include/traits.h:256
Every time a time stamp is used by the security module, it is rounded up to a multiple of this value. This is effectively the time window in which any given secure message will be considered valid. Replay attacks are possible within the same time window, so you should not set this value too high. A value too low might make it impossible to send valid messages due to duty cycling in the network, so you should take that into account.

4 User's Guide

This guide is divided in two: one for the server and one for the sensor nodes. Each of these is subdivided in three phases, that must be followed in order: 1) Initialization; 2) Key Establishment and Authentication; 3) Secure Communication.

4.1 Deploying a Server

4.1.1 Initialization

These values characterize this particular node and prepare for execution. For the server, the following parameters need to be set:

NIC Address: app/secure_nic_master.cc:158
The MAC address of the server. This value should be known by every node.

PTP role: app/secure_nic_master.cc:117-119
Tells the PTP object that this is a master node, which will be used as a reference clock by the other nodes, and will issue synchronization messages.

Random seed: app/secure_nic_master.cc:123
It is important to make sure that each node has its own seed, because Diffie-Hellman private parameters are chosen at random, and two nodes ideally shouldn't have the same private keys.

PTP: app/secure_nic_master.cc:120,145
Line 120 tells the PTP object to initialize its internal parameters. Line 145 sets up a periodic thread which will issue a PTP synchronization message every configured time period.

Secure_NIC: app/secure_nic_master.cc:125
The Secure_NIC constructor will calculate this node's Diffie-Hellman keypair. This is a costly process and might take a few seconds, depending on the prime size in use. It takes as parameters a boolean which is true *iff* this is a server, the cipher to be used, the Poly1305 implementation and the NIC used

as the lower layer.

Trusted IDs: app/secure_nic_master.cc:128-136
The ID of every node to be trusted by this server should be informed via the insert_trusted_id method. Any node with an ID not registered will *not* be able to authenticate.

4.1.2 Key Establishment and Authentication

The server just needs to start accepting authentication requests. The authentication process is then carried out automatically when requests arrive.

Start accepting requests: app/secure_nic_master.cc:139
The server will *not* try to authenticate any node unless this value is set. You can write to this variable anytime to start/stop accepting connections. It starts as false by default.

4.1.3 Secure Communication

Nodes that are authenticated with the server can send and receive secure messages to/from it. If a destination node is *not* authenticated with this server, calling the send method will result in no message being sent.

Receiving secure messages: app/secure_nic_master.cc:69,74,76,142
You can receive secure messages by registering an observer to the secure nic. Every time a secure message arrives, the update method will be called. Decryption is handled transparently, and the message is delivered already decrypted.

Sending secure messages: app/secure_nic_master.cc:81
You can send secure messages by just calling the send method. If the destination node is authenticated, the message will be encrypted and sent.

4.2 Deploying a Sensor

4.2.1 Initialization

These values characterize this particular node and prepare for execution. For the sensors, the following parameters need to be set:

Server's NIC Address: app/secure_nic_slave.cc:19
The MAC address of the server. This value should be known by every node.

NIC Address: app/secure_nic_slave.cc:20,161
The MAC address of this sensor. Every node in the network should have a unique MAC address. If this is not the case, the authentication process may go wrong.

PTP role: app/secure_nic_slave.cc:97,98
Tells the PTP object that this is a slave node, which will respond to synchronization messages and adjust its clock according to the server's.

Random seed: app/secure_nic_slave.cc:106
It is important to make sure that each node has its own seed, because Diffie-Hellman private parameters are chosen at random, and two nodes ideally shouldn't have the same private keys.

PTP: app/secure_nic_slave.cc:100
Tell the PTP object to initialize its internal parameters.

Secure_NIC: app/secure_nic_slave.cc:111
The Secure_NIC constructor will calculate this node's Diffie-Hellman keypair. This is a costly process and might take a few seconds, depending on the prime size in use. It takes as parameters a boolean which is true *iff* this is a server, the cipher to be used, the Poly1305 implementation and the NIC used as the lower layer.

IDs app/secure_nic_slave.cc:112
Every trusted node must have an unique ID, known to the gateway. Use the set_id method to set the ID of this node in the Secure_NIC object.

4.2.2 Key Establishment and Authentication

The sensor nodes need to ask the server for authentication.

Request authentication from server: app/secure_nic_slave.cc:121
Use this method to send the first message in the authentication protocol. The rest of the messages will be handled automatically.

Poll for authentication: app/secure_nic_slave.cc:124
The authenticated() method will return true only when this node finishes the protocol correctly, thus being authenticated to the server and sharing a symmetric key with it.

4.2.3 Secure Communication

Nodes that are authenticated with the server can send and receive secure messages to/from it. If the server is *not* authenticated with this node, calling the send method will result in no message being sent.

Receiving secure messages: app/secure_nic_slave.cc:75-80,134
You can receive secure messages by registering an observer to the secure nic. Every time a secure message arrives, the update method will be called. Decryption is handled transparently, and the message is delivered already decrypted.

Sending secure messages: app/secure_nic_slave.cc:141

You can send secure messages by just calling the send method. If the destination node is authenticated, the message will be encrypted and sent.

5 Using Bignums with different moduli

If you need to operate on several finite fields in the same application, this is possible. If the fields use moduli of the same size, you can just use the set_mod method in Bignum to set a different modulo per-object. Keep in mind that *you* must handle allocation of the modulo data (the data is not copied).

If you need to use moduli of different sizes, you can create a class that extends Bignum and overwrite the word and sz_word parameters, as well as the modulo and Barrett μ. This is exactly what is done by Poly1305-AES, so check out the files include/poly1305.h and src/abstraction/poly1305.cc.

6 Questions?

Feel free to contact me via email: davir@lisha.ufsc.br.

References

[1] P. Oliveira, A. M. Okazaki, and A. A. Fröhlich, "Sincronização de tempo a nível de so utilizando o protocolo ieee1588," in *Simpósio Brasileiro de Engenharia de Sistemas Computacionais*, Natal, Brazil, November 2012.

[2] D. J. Bernstein, "The poly1305-aes message-authentication code," in *Proceedings of Fast Software Encryption*, Paris, France, February 2005, pp. 32–49.

[3] W. Stallings, *Cryptography And Network Security*, 5th ed. Pearson, 2011.

[4] M. Brown, D. Hankerson, J. López, and A. Menezes, "Software implementation of the nist elliptic curves over prime fields," in *Topics in Cryptology - CT-RSA 2001*, ser. Lecture Notes in Computer Science, D. Naccache, Ed. Springer Berlin Heidelberg, 2001, vol. 2020, pp. 250–265.

[5] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.

[6] A. A. Fröhlich, A. M. Okazaki, R. V. Steiner, P. Oliveira, and J. E. Martina, "A cross-layer approach to trustfulness in the internet of things," in *9th Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, Paderborn, Germany, June 2013.

[7] *Standards for Efficient Cryptography (SEC) SEC 2: Recommended Elliptic Curve Domain Parameters*, Certicom Research, September 2000.

[8] N. S. Agency. (2014, May) The case for elliptic curve cryptography. [Online]. Available: http://www.nsa.gov/business/programs/elliptic_curve.shtml

ANEXO C – Código Fonte

LISTA DE CÓDIGOS-FONTE

Este anexo contém o código-fonte desenvolvido durante este trabalho. A implementação foi feita no contexto do sistema operacional EPOS, plataforma EPOSMoteII e linguagem C++. Os códigos-fonte já presentes no EPOS que foram aproveitados, porém não desenvolvidos no contexto deste trabalho, não se encontram neste anexo.

O arquivo "traits.h" é parte do EPOS e contém configurações para vários elementos do sistema. Aqui, reproduz-se apenas o que foi adicionado a este arquivo por este trabalho. Os demais arquivos são apresentados integralmente, exceto pela remoção de pequenas porções de código comentado (em sua maioria usados apenas durante fase de depuração) e mudanças de quebras de linha para adequação ao formato do presente documento.

Código C.1 – traits.h

```
1  /* (...) */
2
3  template <> struct Traits<Bignum> : public Traits<void>
4  {
5      // You can edit these values
6      typedef unsigned int digit;
7      typedef unsigned long long double_digit;
8      static const unsigned int word = 4;
9
10     // You shouldn't edit these
11     static const unsigned int sz_digit = sizeof(digit);
12     static const unsigned int sz_word = sz_digit * word;
13     static const unsigned int double_word = 2 * word;
14     static const unsigned int bits_in_digit = sz_digit * 8;
15 };
16
17 template <> struct Traits<Diffie_Hellman> : public Traits<void>
18 {
19 //     static const bool debugged = true;
20
21     // Don't edit these, unless you really know what you're doing
22     static const unsigned int SECRET_SIZE = Traits<Bignum>::sz_word;
23     static const unsigned int PUBLIC_KEY_SIZE = Traits<Bignum>::sz_word * 2;
24 };
25
26 template <> struct Traits<Secure_NIC> : public Traits<void>
27 {
28 //     static const bool debugged = true;
29
30     static const int PROTOCOL_ID = 42;
31     static const unsigned int ID_SIZE = 20;
32     static const unsigned long long TIME_WINDOW = 3000000U; // In Microseconds
33 };
34
35 /* (...) */
```

Código C.2 – bignum.h

```
1  #ifndef __bignum_h
2  #define __bignum_h
3
4  #include < utility / random.h>
5  #include <cpu.h>
6
7  //#define DEBUG_BIGNUM
8
9  __BEGIN_SYS;
10
```

```
11  class Bignum
12  {
13      public:
14      typedef Traits<Bignum>::digit digit ;
15      typedef Traits<Bignum>::double_digit double_digit;
16      static const unsigned int sz_digit = Traits<Bignum>::sz_digit;
17      static const unsigned int word = Traits<Bignum>::word;
18      static const unsigned int sz_word = Traits<Bignum>::sz_word;
19      static const unsigned int double_word = Traits<Bignum>::double_word;
20      static const unsigned int bits_in_digit = Traits<Bignum>::bits_in_digit;
21
22      static const digit default_mod[word];
23      static const digit default_barrett_u[word+1];
24
25      union
26      {
27          digit data[word];
28          unsigned char byte_data[sz_word];
29      };
30
31      protected:
32
33      // All operations will be done modulo _mod
34      const digit * _mod;
35      // Auxiliary constant to help multiplication
36      // _barrett_u = TODO
37      const digit * _barrett_u;
38
39      public:
40
41      // -----------------------------------------------------
42      // Sets data to a random number, smaller than _mod
43      // -----------------------------------------------------
44      void random() __attribute__((noinline))
45      {
46          int i;
47          for(i=word-1; i>=0 && (_mod[i] == 0); i--)
48              data[i]=0;
49          data[i] = Pseudo_Random::random() % _mod[i];
50          for(--i;i>=0;i--)
51              data[i] = Pseudo_Random::random();
52      }
53
54      // -----------------------------------------------------------
55      // Sets data using a little -endian byte-by-byte representation
56      // -----------------------------------------------------------
57      inline int set_bytes(const unsigned char * values, unsigned int len)
58      {
59          unsigned int i;
60          for(i=0; i<len && i<sz_word; i++)
61              byte_data[i] = values[i];
62          for(; i<sz_word;i++)
63              byte_data[i] = 0;
64          return i;
65      }
66
67      // -----------------------------------------------------------
68      // -Sets the module to be applied in every operation on this object
69      // -The user must handle allocation
70      // -To be used only when a module different than the default is needed
```

```
71      // ------------------------------------------------------------------
72      inline void set_mod(digit * mod, digit * barrett_u)
73      {
74          _mod = mod;
75          _barrett_u = barrett_u;
76      }
77
78      friend OStream &operator<< (OStream &out, const Bignum &b)
79      {
80          unsigned int i;
81          out << '[';
82          for(i=0;i<word;i++)
83          {
84              out << (unsigned int)b.data[i];
85              if (i < word-1)
86                  out << ", ";
87          }
88          out << "]";
89          return out;
90      }
91
92      public:
93      // ------------------------------------------------------------------
94      // Sets data using a little-endian byte-by-byte representation
95      // ------------------------------------------------------------------
96      Bignum(unsigned char * bytes, unsigned int len)
97      {
98          set_bytes(bytes, len);
99          _mod = default_mod;
100         _barrett_u = default_barrett_u;
101     }
102
103     // ------------------------
104     // Convert from unsigned int
105     // ------------------------
106     Bignum(unsigned int n = 0) __attribute__((noinline))
107     {
108         *this = n;
109         _mod = default_mod;
110         _barrett_u = default_barrett_u;
111     }
112
113     //------------------------------------
114     // Returns true if this number is even,
115     // false otherwise
116     //------------------------------------
117     inline bool is_even(){ return !(data[0] % 2); }
118
119     // ------------
120     // XOR operator
121     // ------------
122     inline void operator^=(const Bignum &b) //__attribute__( ( noinline ) )
123     {
124         for(unsigned int i=0; i<word; i++)
125             data[i] ^= b.data[i];
126     }
127
128     // ------------------------
129     // Convert from unsigned int
130     // ------------------------
```

```
131     inline void operator=(unsigned int n) //__attribute__( (  noinline  ) )
132     {
133         if (sizeof(n) <= sz_digit)
134         {
135             data[0] = n;
136             for(unsigned int i=1;i<word;i++)
137                 data[i] = 0;
138         }
139         else
140         {
141             unsigned int i;
142             for(i=0; (n != 0) && (i<word); i++)
143             {
144                 data[i] = n;
145                 n >>= bits_in_digit ;  //Compiler warning can be ignored because of the 'if'
146             }
147             for(; i<word;i++)
148                 data[i] = 0;
149         }
150     }
151
152     // ----------------
153     // Copies data only
154     // ----------------
155     inline void operator=(const Bignum &b) //__attribute__( ( noinline ) )
156     {
157         for(unsigned int i=0;i<word;i++)
158             data[i] = b.data[i];
159     }
160
161     // --------------------
162     // Comparison operators
163     // --------------------
164     inline bool operator==(const Bignum &b) const //__attribute__( ( noinline ) )
165     { return (cmp(data, b.data, word) == 0); }
166     inline bool operator!=(const Bignum &b) const //__attribute__( ( noinline ) )
167     { return (cmp(data, b.data, word) != 0); }
168     inline bool operator>=(const Bignum &b) const //__attribute__( ( noinline ) )
169     { return (cmp(data, b.data, word) >= 0); }
170     inline bool operator<=(const Bignum &b) const //__attribute__( ( noinline ) )
171     { return (cmp(data, b.data, word) <= 0); }
172     inline bool operator>(const Bignum &b) const //__attribute__( ( noinline ) )
173     { return (cmp(data, b.data, word) > 0); }
174     inline bool operator<(const Bignum &b) const //__attribute__( ( noinline ) )
175     { return (cmp(data, b.data, word) < 0); }
176
177     // ------------------------------------
178     // data = (data * b.data) % _mod
179     // ------------------------------------
180     void operator*=(const Bignum &b)__attribute__((noinline))
181     {
182         if (b == 1) return;
183 #ifdef DEBUG_BIGNUM
184         kout << "HERE" << endl;
185         kout << *this << endl;
186         kout << "*" << endl;
187         kout << b << endl;
188         kout << "%" << endl;
189         kout << '[' ;
190         for(int i=0;i<word-1;i++)
```

```
191                 kout << _mod[i] << ",  ";
192             kout << _mod[word-1] << ']' << endl;
193             kout << "==" << endl;
194 #endif
195             digit  _mult_result[double_word];
196 
197             // -------------------------------
198             // _mult_result = data * b.data
199             // -------------------------------
200             simple_mult(_mult_result, data, b.data, word);
201 
202             // -------------------------
203             // Barrett modular reduction
204             // -------------------------
205             barrett_reduction(data, _mult_result, word);
206 #ifdef DEBUG_BIGNUM
207             kout << *this << endl;
208 #endif
209         }
210 
211         // -----------------------------------
212         // data = (data + b.data) % _mod
213         // -----------------------------------
214         void operator+=(const Bignum &b)__attribute__((noinline))
215         {
216 #ifdef DEBUG_BIGNUM
217             kout << "HERE" << endl;
218             kout << *this << endl;
219             kout << "+" << endl;
220             kout << b << endl;
221             kout << "%" << endl;
222             kout << '[';
223             for(int i=0;i<word-1;i++)
224                 kout << _mod[i] << ",  ";
225             kout << _mod[word-1] << ']' << endl;
226             kout << "==" << endl;
227 #endif
228             if(simple_add(data, data, b.data, word))
229                 simple_sub(data, data, _mod, word);
230             if(cmp(data, _mod, word) >= 0)
231                 simple_sub(data, data, _mod, word);
232 #ifdef DEBUG_BIGNUM
233             kout << *this << endl;
234 #endif
235         }
236 
237         // -----------------------------------
238         // data = (data - b.data) % _mod
239         // -----------------------------------
240         void operator-=(const Bignum &b)__attribute__((noinline))
241         {
242 #ifdef DEBUG_BIGNUM
243             kout << "HERE" << endl;
244             kout << *this << endl;
245             kout << "-" << endl;
246             kout << b << endl;
247             kout << "%" << endl;
248             kout << '[';
249             for(int i=0;i<word-1;i++)
250                 kout << _mod[i] << ",  ";
```

```
251             kout << _mod[word-1] << ']' << endl;
252             kout << "==" << endl;
253     #endif
254             if (simple_sub(data, data, b.data, word))
255                 simple_add(data, data, _mod, word);
256     #ifdef DEBUG_BIGNUM
257             kout << *this << endl;
258     #endif
259         }
260
261         //-------------------------------------------------------------------
262         // Calculates the modular multiplicative inverse of this number
263         //-------------------------------------------------------------------
264         void inverse() __attribute__((noinline))
265         {
266             Bignum A(1), u, v, zero(0);
267             for(unsigned int i=0;i<word;i++)
268             {
269                 u.data[i] = data[i];
270                 v.data[i] = _mod[i];
271             }
272             *this = 0;
273             while(u != zero)
274             {
275                 while(u.is_even())
276                 {
277                     u.divide_by_two();
278                     if (A.is_even())
279                         A.divide_by_two();
280                     else
281                     {
282                         bool carry = simple_add(A.data, A.data, _mod, word);
283                         A.divide_by_two(carry);
284                     }
285                 }
286                 while(v.is_even())
287                 {
288                     v.divide_by_two();
289                     if (is_even())
290                         divide_by_two();
291                     else
292                     {
293                         bool carry = simple_add(data, data, _mod, word);
294                         divide_by_two(carry);
295                     }
296                 }
297                 if (u >= v)
298                 {
299                     u -= v;
300                     A -= *this;
301                 }
302                 else
303                 {
304                     v-=u;
305                     *this -= A;
306                 }
307             }
308         }
309
310         //-------------------------------------------------------------------
```

```
311     // Shift left (actually shift right, because of little endianness)
312     // - Does not apply modulo
313     // - Returns carry bit
314     //-----------------------------------------------------------------------
315     bool multiply_by_two(bool carry = 0)__attribute__((noinline))
316     {
317 #ifdef DEBUG_BIGNUM
318         bool debug = !carry;
319         if (debug)
320         {
321             kout << "HERE" << endl;
322             kout << *this << endl;
323             kout << "<<" << endl;
324             kout << "[1]" << endl;
325             kout << "%" << endl;
326             kout << '[';
327             for(int i=0;i<word-1;i++)
328                 kout << _mod[i] << ", ";
329             kout << _mod[word-1] << ']' << endl;
330             kout << "==" << endl;
331         }
332 #endif
333         bool next_carry;
334         for(unsigned int i=0;i<word;i++)
335         {
336             next_carry = data[i] >> (bits_in_digit -1);
337             data[i] <<= 1;
338             data[i] += (digit)carry;
339             carry = next_carry;
340         }
341 #ifdef DEBUG_BIGNUM
342         if (debug)
343             kout << *this << endl;
344 #endif
345         return carry;
346     }
347
348     //-----------------------------------------------------------------------
349     // Shift right (actually shift left, because of little endianness)
350     // - Does not apply modulo
351     // - Returns carry bit
352     //-----------------------------------------------------------------------
353     bool divide_by_two(bool carry = 0)__attribute__((noinline))
354     {
355 #ifdef DEBUG_BIGNUM
356         bool debug = !carry;
357         if (debug)
358         {
359             kout << "HERE" << endl;
360             kout << *this << endl;
361             kout << ">>" << endl;
362             kout << "[1]" << endl;
363             kout << "%" << endl;
364             kout << '[';
365             for(int i=0;i<word-1;i++)
366                 kout << _mod[i] << ", ";
367             kout << _mod[word-1] << ']' << endl;
368             kout << "==" << endl;
369         }
370 #endif
```

```
371         bool next_carry;
372         for(int i=word-1;i>=0;i--)
373         {
374             next_carry = data[i] % 2;
375             data[i] >>= 1;
376             data[i] += (digit)carry << (bits_in_digit -1);
377             carry = next_carry;
378         }
379 #ifdef DEBUG_BIGNUM
380         if (debug)
381             kout << *this << endl;
382 #endif
383         return carry;
384     }
385
386     protected:
387     //--------------------
388     // C-like comparison
389     // Returns:
390     // 1  if a > b
391     // -1 if a < b
392     // 0  if a == b
393     //--------------------
394     static inline int cmp(const digit * a, const digit *b, int size)
395     {
396         for(int i=size-1; i>=0; i--)
397         {
398             if (a[i] > b[i]) return 1;
399             else if (a[i] < b[i]) return -1;
400         }
401         return 0;
402     }
403
404     // ------------------------------------------------
405     // res = a - b
406     // returns: borrow bit
407     // -No modulo applied
408     // -a, b and res are assumed to have size 'size'
409     // -a, b, res are allowed to point to the same place
410     // ------------------------------------------------
411     static inline bool simple_sub(digit * res, const digit * a, const digit * b, unsigned int size)
412         __attribute__((noinline))
413     {
414         double_digit borrow = 0;
415         double_digit aux = ((double_digit)1) << bits_in_digit ;
416         for(unsigned int i=0; i<size; i++)
417         {
418             double_digit anow = a[i];
419             double_digit bnow = ((double_digit)b[i]) + borrow;
420             borrow = (anow < bnow);
421             if (borrow)
422                 res[i] = (aux - bnow) + anow;
423             else
424                 res[i] = anow - bnow;
425         }
426         return borrow;
427     }
428
429     // ------------------------------------------------
430     // res = a + b
```

```
431     // returns: carry bit
432     // -No modulo applied
433     // -a, b and res are assumed to have size 'size'
434     // -a, b, res are allowed to point to the same place
435     // ------------------------------------------------------------
436     static inline bool simple_add(digit * res, const digit * a, const digit * b, unsigned int size)
437         __attribute__((noinline))
438     {
439         bool carry = 0;
440         for(unsigned int i=0; i<size; i++)
441         {
442             double_digit tmp = (double_digit)carry + (double_digit)a[i] + (double_digit)b[i];
443             res[i] = tmp;
444             carry = tmp >> bits_in_digit;
445         }
446         return carry;
447     }
448
449     // ------------------------------------------------------------
450     // res = (a * b)
451     // - Does not apply module
452     // - a and b are assumed to be of size 'size'
453     // - res is assumed to be of size '2*size'
454     // ------------------------------------------------------------
455     static inline void simple_mult(digit * res, const digit * a, const digit * b, unsigned int size)
456     {
457         unsigned int i;
458         double_digit r0=0, r1=0, r2=0;
459         for(i=0;i<size*2-1;i++)
460         {
461             for(unsigned int j=0;(j<size) && (j<=i); j++)
462             {
463                 unsigned int k = i - j;
464                 if (k<size)
465                 {
466                     double_digit aj = a[j];
467                     double_digit bk = b[k];
468                     double_digit prod = aj * bk;
469                     r0 += (digit)prod;
470                     r1 += (prod >> bits_in_digit) + (r0 >> bits_in_digit);
471                     r0 = (digit)r0;
472                     r2 += (r1 >> bits_in_digit);
473                     r1 = (digit)r1;
474                 }
475             }
476             res[i] = r0;
477             r0 = r1;
478             r1 = r2;
479             r2 = 0;
480         }
481         res[i] = r0;
482     }
483
484     // ------------------------------------------------------------
485     // res = a % _mod
486     // - Intended to be used after a multiplication
487     // - res is assumed to be of size 'size'
488     // - a is assumed to be of size '2*size'
489     // ------------------------------------------------------------
490     inline void barrett_reduction(digit * res, const digit * a, unsigned int size)
```

```
491     {
492         digit  q[size+1];
493
494         // ------------------------------------------------------------
495         // q = floor( ( floor( a/base^(size-1) ) * barrett_u ) / base^(size+1))
496         // ------------------------------------------------------------
497         double_digit r0=0, r1=0, r2=0;
498         unsigned int i;
499         for(i=0; i<(2*(size+1))-1 ;i++)
500         {
501             for(unsigned int j=0;(j<size+1) && (j<=i); j++)
502             {
503                 unsigned int k = i - j;
504                 if (k<size+1)
505                 {
506                     // shifting left ( little endian) size-1 places
507                     // a is assumed to have size size*2
508                     double_digit aj = a[j+(size-1)];
509                     double_digit bk = _barrett_u[k];
510                     double_digit prod = aj * bk;
511                     r0 += (digit)prod;
512                     r1 += (prod >> bits_in_digit) + (r0 >> bits_in_digit);
513                     r0 = (digit)r0;
514                     r2 += r1 >> bits_in_digit;
515                     r1 = (digit)r1;
516                 }
517             }
518             // shifting left ( little endian) size+1 places
519             if (i>=size+1)
520                 q[i-(size+1)] = r0;
521             r0 = r1;
522             r1 = r2;
523             r2 = 0;
524         }
525         q[i-(size+1)] = r0;
526
527         digit  r[size+1];
528         // ------------------------------
529         // r = (q * _mod) % base^(size+1)
530         // ------------------------------
531         r0=0, r1=0, r2=0;
532         for(i=0;i<size+1;i++)
533         {
534             for(unsigned int j=0;j<=i;j++)
535             {
536                 unsigned int k = i - j;
537                 double_digit aj = q[j];
538                 double_digit bk = (k == size ? 0 : _mod[k]);
539                 double_digit prod = aj * bk;
540                 r0 += (digit)prod;
541                 r1 += (prod >> bits_in_digit) + (r0 >> bits_in_digit);
542                 r0 = (digit)r0;
543                 r2 += r1 >> bits_in_digit;
544                 r1 = (digit)r1;
545             }
546             r[i] = r0;
547             r0 = r1;
548             r1 = r2;
549             r2 = 0;
550         }
```

```
551
552         // ------------------------------------------------------------
553         // r = ((a % base^(size+1)) - r) % base^(size+1)
554         // ------------------------------------------------------------
555         simple_sub(r, a, r, size+1);
556
557         // ------------
558         // r = r % _mod
559         // ------------
560         while((r[size] > 0) || (cmp(r, _mod, size) >= 0))
561         {
562             if (simple_sub(r, r, _mod, size))
563                 r[size]--;
564         }
565
566         for(unsigned int i=0;i<size;i++)
567             res[i] = r[i];
568     }
569 };
570 __END_SYS;
571 #endif
```

Código C.3 – diffie_hellman.h

```
1  #ifndef diffie_hellman_h
2  #define diffie_hellman_h
3
4  #include <bignum.h>
5  #include <chronometer.h>
6
7  __BEGIN_SYS;
8
9  class Diffie_Hellman
10 {
11     public:
12     Chronometer dh_chrono;
13     static const unsigned int SECRET_SIZE = Traits<Diffie_Hellman>::SECRET_SIZE;
14     static const unsigned int PUBLIC_KEY_SIZE = Traits<Diffie_Hellman>::PUBLIC_KEY_SIZE;
15
16     static const unsigned char default_base_point_x[Bignum::word * Bignum::sz_digit];
17     static const unsigned char default_base_point_y[Bignum::word * Bignum::sz_digit];
18
19     struct ECC_Point : private Bignum
20     {
21         Bignum x, y, z;
22         ECC_Point()__attribute__((noinline)){}
23         void operator=(const ECC_Point &b)__attribute__((noinline))
24         {
25             x = b.x;
26             y = b.y;
27             z = b.z;
28         }
29         void operator*=(const Bignum &b);
30         friend OStream &operator<<(OStream &out, const ECC_Point &a)
31         {
32             out << "Point: " << endl;
33             out << a.x << endl;
34             out << a.y << endl;
35             out << a.z << endl;
36             return out;
```

```
37          }
38
39          private:
40          void jacobian_double();
41          void add_jacobian_affine(const ECC_Point &b);
42      };
43
44      Diffie_Hellman( const unsigned char base_point_data_x[Bignum::word * Bignum::sz_digit],
45                  const unsigned char base_point_data_y[Bignum::word * Bignum::sz_digit])
46          __attribute__((noinline))
47          : _base_point()
48      {
49          _base_point.x.set_bytes(base_point_data_x, Bignum::word * Bignum::sz_digit);
50          _base_point.y.set_bytes(base_point_data_y, Bignum::word * Bignum::sz_digit);
51          _base_point.z = 1;
52          generate_keypair();
53      }
54      Diffie_Hellman()__attribute__((noinline))
55      {
56          dh_chrono.stop();
57          dh_chrono.reset();
58          dh_chrono.start();
59          _base_point.x.set_bytes(default_base_point_x, Bignum::word * Bignum::sz_digit);
60          _base_point.y.set_bytes(default_base_point_y, Bignum::word * Bignum::sz_digit);
61          _base_point.z = 1;
62          generate_keypair();
63          dh_chrono.stop();
64      }
65
66      private:
67      Bignum _private;
68      //Bignum _key;
69      ECC_Point _base_point;
70      ECC_Point _public;
71
72      public:
73
74
75      int get_public(unsigned char * buffer, int buffer_size);
76      // int get_key(unsigned char * buffer, int buffer_size);
77      void generate_keypair()
78      {
79          db<Diffie_Hellman>(TRC) << "Diffie_Hellman::generate_keypair()" << endl;
80          _private.random();
81          _public = _base_point;
82          _public *= _private;
83          db<Diffie_Hellman>(INF) << "Diffie_Hellman Private: " << _private << endl;
84          db<Diffie_Hellman>(INF) << "Diffie_Hellman Public: " << _public << endl;
85      }
86      void calculate_key(unsigned char * key, int buffer_size, const unsigned char * Yb, int Yb_size);
87  };
88
89  __END_SYS;
90  #endif
```

Código C.4 – diffie_hellman.cc

```
1   #include <diffie_hellman.h>
2   //#include < utility /malloc.h>
3
```

```
4  __USING_SYS;
5
6  int Diffie_Hellman::get_public(unsigned char * buffer, int buffer_size)
7  {
8      int i, j;
9      for(i=0, j=0; i<Bignum::sz_word && j<buffer_size; i++, j++)
10         buffer[j] = _public.x.byte_data[i];
11     for(i=0; i<Bignum::sz_word && j<buffer_size; i++, j++)
12         buffer[j] = _public.y.byte_data[i];
13     return j;
14 }
15
16 void Diffie_Hellman::calculate_key(unsigned char * key, int buffer_size,
17         const unsigned char * Yb, int Yb_size)
18 {
19     dh_chrono.stop();
20     dh_chrono.reset();
21     dh_chrono.start();
22     ECC_Point _Y;
23     int i;
24     i = _Y.x.set_bytes(Yb, Yb_size);
25     _Y.y.set_bytes(Yb+i, Yb_size-i);
26     _Y.z = 1;
27
28     _Y *= _private;
29
30     _Y.x ^= _Y.y;
31
32     for(i=0; i<Bignum::sz_word && i<buffer_size; i++)
33         key[i] = _Y.x.byte_data[i];
34
35     dh_chrono.stop();
36     db<Diffie_Hellman>(INF) << "Diffie_Hellman - Key set: " << _Y.x << endl;
37 }
38
39 void Diffie_Hellman::ECC_Point::operator*=(const Bignum &b)
40 {
41     // Finding last '1' bit of k
42     int t = bits_in_digit ;
43     int b_len = word+1;
44     Bignum::digit now;
45     do
46     {
47         now = b.data[(--b_len)-1];
48     }while(now == 0);
49
50     bool bin[t]; // Binary representation of now
51     ECC_Point pp;
52
53     pp = *this;
54
55     for(int j= bits_in_digit -1;j>=0;j--)
56     {
57         if (now%2) t=j+1;
58         bin[j] = now%2;
59         now/=2;
60     }
61
62     for(int i=b_len-1;i>=0;i--)
63     {
```

```
64          for (; t< bits_in_digit ; t++)
65          {
66              jacobian_double();
67              if (bin[ t ])
68                  add_jacobian_affine(pp);
69          }
70          if ( i>0)
71          {
72              now = b.data[i-1];
73              for( int  j= bits_in_digit -1;j>=0;j--)
74              {
75                  bin[ j ] = now%2;
76                  now/=2;
77              }
78              t=0;
79          }
80      }
81
82      Bignum Z;
83      z.inverse ();
84      Z = z; Z *= z;
85      x *= Z; Z *= z;
86      y *= Z;
87      z = 1;
88  }
89
90  void Diffie_Hellman::ECC_Point::add_jacobian_affine(const ECC_Point &b)
91  {
92      Bignum A,B,C,X,Y,aux,aux2;
93
94      A = z; A *= z;
95
96      B = A;
97
98      A *= b.x;
99
100     B *= z; B *= b.y;
101
102     C = A; C -= x;
103
104     B -= y;
105
106     X = B; X *= B;
107     aux = C; aux *= C;
108
109     Y = aux;
110
111     aux2 = aux;
112     aux *= C;
113     aux2 *= 2; aux2 *= x;
114     aux += aux2;
115     X -= aux;
116
117     aux = Y;
118     Y *= x; Y -= X; Y *= B;
119     aux *= y; aux *= C;
120     Y -= aux;
121
122     z *= C;
123
```

```
124         x = X;
125         y = Y;
126     }
127
128     void Diffie_Hellman::ECC_Point::jacobian_double()
129     {
130         Bignum B, C, aux;
131
132         C = x;
133         aux = z; aux *= z;
134         C -= aux;
135         aux += x;
136         C *= aux; C *= 3;
137
138         z *= y; z *= 2;
139
140         y *= y;
141         B = y;
142
143         y *= x; y *= 4;
144
145         B *= B; B *= 8;
146
147         x = C; x *= x;
148         aux = y; aux *= 2;
149         x -= aux;
150
151         y -= x; y *= C; y -= B;
152     }
```

Código C.5 – poly1305.h

```
1   #ifndef __poly1305_h
2   #define __poly1305_h
3
4   #include < utility / string .h>
5   #include <bignum.h>
6   #include <cipher.h>
7
8   #include <system/config.h>
9
10  __BEGIN_SYS
11
12  class Poly1305 : private Bignum
13  {
14      public:
15
16      Poly1305(char __k[16], char __r[16], Cipher *c);
17      Poly1305(char __k[16], char __r[16]);
18      Poly1305(Cipher *c);
19      Poly1305();
20      ~Poly1305();
21
22      void poly1305(char *out, const char *r, const char *s, const char *m, unsigned int l);
23
24      void poly1305_bignum(char *out, const char *s, const char *m, unsigned int l);
25
26      void authenticate( char out[16], const char n[16], const char * m, unsigned int l);
27
28      bool verify (const char a[16], const char n[16], const char m[],unsigned int l);
```

```
29
30      static bool isequal(const char x[16], const char y[16]);
31
32      void aes(char outflip[16], const char n[16])
33      {
34          char kflip[16];
35          char nflip[16];
36          char out[16];
37          int i;
38          for(i=0;i<4;i++)
39          {
40              kflip[i] = _k[3-i];
41              kflip[4+i] = _k[7-i];
42              kflip[8+i] = _k[11-i];
43              kflip[12+i] = _k[15-i];
44
45              nflip[i] = n[3-i];
46              nflip[4+i] = n[7-i];
47              nflip[8+i] = n[11-i];
48              nflip[12+i] = n[15-i];
49          }
50          _cipher->encrypt(nflip, kflip, out);
51          for(i=0;i<4;i++)
52          {
53              outflip[i] = out[3-i];
54              outflip[4+i] = out[7-i];
55              outflip[8+i] = out[11-i];
56              outflip[12+i] = out[15-i];
57          }
58      }
59
60      bool cipher(Cipher *c);
61      void k(char k1[16]);
62      void r(char r1[16]);
63      void kr(char kr1[32]);
64
65      const static unsigned int word = (17 + (sz_digit - 1)) / sz_digit;
66      const static unsigned int sz_word = word * sz_digit;
67
68      unsigned char _k[sz_word];
69      unsigned char _r[sz_word];
70  private:
71      Cipher * _cipher;
72
73      // 2^(130) - 5
74      // = { 251,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,3 };
75      const static unsigned char p1305_data[17];
76
77      const static digit default_barrett_u[word + 1];
78      unsigned char p1305[sz_word];
79      char _mac[32];
80  };
81
82  __END_SYS
83  #endif
```

Código C.6 – poly1305.cc

```
1  #include <poly1305.h>
2  #include <aes.h>
```

```
3
4  __USING_SYS;
5
6  // 2^(130) - 5
7  const unsigned char Poly1305::p1305_data[17] =
8  { 251,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,3 };
9  const Poly1305::digit Poly1305::default_barrett_u[] =
10 {0, 1342177280U, 0, 0, 0, 1073741824U};
11
12 Poly1305::Poly1305(char __k[16], char __r[16], Cipher *c)
13 {
14     this->k(__k); this->r(__r); this->cipher(c);
15     int i;
16     for(i=0;i<17;i++)
17         p1305[i] = p1305_data[i];
18     for(;i<sz_word;i++)
19         p1305[i] = 0;
20     _mod = (digit *)p1305;
21     _barrett_u = default_barrett_u;
22 }
23
24 Poly1305::Poly1305(char __k[16], char __r[16])
25 {
26     this->k(__k); this->r(__r);
27     int i;
28     for(i=0;i<17;i++)
29         p1305[i] = p1305_data[i];
30     for(;i<sz_word;i++)
31         p1305[i] = 0;
32     _mod = (digit *)p1305;
33     _barrett_u = default_barrett_u;
34 }
35
36 Poly1305::Poly1305(Cipher *c)
37 {
38     this->cipher(c);
39     int i;
40     for(i=0;i<17;i++)
41         p1305[i] = p1305_data[i];
42     for(;i<sz_word;i++)
43         p1305[i] = 0;
44     _mod = (digit *)p1305;
45     _barrett_u = default_barrett_u;
46 }
47
48 Poly1305::Poly1305()
49 {
50     _cipher = new AES();
51     int i;
52     for(i=0;i<17;i++)
53         p1305[i] = p1305_data[i];
54     for(;i<sz_word;i++)
55         p1305[i] = 0;
56     _mod = (digit *)p1305;
57     _barrett_u = default_barrett_u;
58 }
59
60 Poly1305::~Poly1305(){}
61
62 void Poly1305::k(char k1[16])
```

```
63  {
64      int i;
65      for(i=0; i<16; i++)
66          _k[i] = k1[i];
67      for(; i<sz_word; i++)
68          _k[i] = 0;
69  }
70
71  void Poly1305::r(char r1[16])
72  {
73      int i;
74      for(i=0; i<16; i++)
75          _r[i] = r1[i];
76      for(; i<sz_word; i++)
77          _r[i] = 0;
78
79      _r[3] &= 15;
80      _r[7] &= 15;
81      _r[11] &= 15;
82      _r[15] &= 15;
83      _r[4] &= 252;
84      _r[8] &= 252;
85      _r[12] &= 252;
86  }
87
88  void Poly1305::kr(char kr[32])
89  {
90      this->k(kr);
91      this->r(kr+16);
92  }
93
94  bool Poly1305::isequal(const char x[16], const char y[16])
95  {
96      /* Copied from Bernstein's (the author of Poly) code */
97      register unsigned int d; register unsigned int x0;
98      register unsigned int x1; register unsigned int x2;
99      register unsigned int x3; register unsigned int x4;
100     register unsigned int x5; register unsigned int x6;
101     register unsigned int x7; register unsigned int x8;
102     register unsigned int x9; register unsigned int x10;
103     register unsigned int x11; register unsigned int x12;
104     register unsigned int x13; register unsigned int x14;
105     register unsigned int x15; register unsigned int y0;
106     register unsigned int y1; register unsigned int y2;
107     register unsigned int y3; register unsigned int y4;
108     register unsigned int y5; register unsigned int y6;
109     register unsigned int y7; register unsigned int y8;
110     register unsigned int y9; register unsigned int y10;
111     register unsigned int y11; register unsigned int y12;
112     register unsigned int y13; register unsigned int y14;
113     register unsigned int y15;
114
115     x0 = *(unsigned char *) (x + 0);
116     y0 = *(unsigned char *) (y + 0);
117     x1 = *(unsigned char *) (x + 1);
118     y1 = *(unsigned char *) (y + 1);
119     x2 = *(unsigned char *) (x + 2);
120     y2 = *(unsigned char *) (y + 2);
121     d = y0 ^ x0;
122     x3 = *(unsigned char *) (x + 3);
```

```
123         y1 ^= x1;
124         y3 = *(unsigned char *) (y + 3);
125         d |= y1;
126         x4 = *(unsigned char *) (x + 4);
127         y2 ^= x2;
128         y4 = *(unsigned char *) (y + 4);
129         d |= y2;
130         x5 = *(unsigned char *) (x + 5);
131         y3 ^= x3;
132         y5 = *(unsigned char *) (y + 5);
133         d |= y3;
134         x6 = *(unsigned char *) (x + 6);
135         y4 ^= x4;
136         y6 = *(unsigned char *) (y + 6);
137         d |= y4;
138         x7 = *(unsigned char *) (x + 7);
139         y5 ^= x5;
140         y7 = *(unsigned char *) (y + 7);
141         d |= y5;
142         x8 = *(unsigned char *) (x + 8);
143         y6 ^= x6;
144         y8 = *(unsigned char *) (y + 8);
145         d |= y6;
146         x9 = *(unsigned char *) (x + 9);
147         y7 ^= x7;
148         y9 = *(unsigned char *) (y + 9);
149         d |= y7;
150         x10 = *(unsigned char *) (x + 10);
151         y8 ^= x8;
152         y10 = *(unsigned char *) (y + 10);
153         d |= y8;
154         x11 = *(unsigned char *) (x + 11);
155         y9 ^= x9;
156         y11 = *(unsigned char *) (y + 11);
157         d |= y9;
158         x12 = *(unsigned char *) (x + 12);
159         y10 ^= x10;
160         y12 = *(unsigned char *) (y + 12);
161         d |= y10;
162         x13 = *(unsigned char *) (x + 13);
163         y11 ^= x11;
164         y13 = *(unsigned char *) (y + 13);
165         d |= y11;
166         x14 = *(unsigned char *) (x + 14);
167         y12 ^= x12;
168         y14 = *(unsigned char *) (y + 14);
169         d |= y12;
170         x15 = *(unsigned char *) (x + 15);
171         y13 ^= x13;
172         y15 = *(unsigned char *) (y + 15);
173         d |= y13;
174         y14 ^= x14;
175         d |= y14;
176         y15 ^= x15;
177         d |= y15;
178         d -= 1;
179         d >>= 8;
180
181         return (d != 0);
182     }
```

```
183
184 bool Poly1305::cipher(Cipher *c)
185 {
186     bool ok = (c->mode(Cipher::CBC));
187     if (ok) _cipher = c;
188     return ok;
189 }
190
191 bool Poly1305::verify(const char a[16],
192   const char n[16], const char m[],unsigned int l)
193 {
194   char aeskn[16];
195   char valid[16];
196   aes(aeskn, n);
197   poly1305_bignum(valid,aeskn,m,l);
198   return isequal(a, valid);
199 }
200
201 void Poly1305::authenticate(char out[16],
202   const char n[16], const char m[],unsigned int l)
203 {
204   char aeskn[16];
205   aes(aeskn, n);
206   poly1305_bignum(out,aeskn,m,l);
207 }
208
209 void Poly1305::poly1305_bignum(char *out, const char *s, const char *m, unsigned int l)
210 {
211  /* r = Poly's key
212     s = AES result
213     m = message
214     l = message length */
215     unsigned int j;
216     unsigned int end=16;
217     unsigned char c[sz_word], h[(sz_word+1)*2];
218
219     for(int i=0;i<sizeof(h);i++)
220         h[i] = 0;
221
222     while(l>0)
223     {
224         if (l<end) end = l;
225
226         int i;
227         for(i=0; i<end; i++)
228             c[i] = m[i];
229         c[end] = 1;
230         for(; i<sz_word;i++)
231             c[i] = 0;
232
233         simple_add((digit *)h, (digit *)h, (digit *)c, word);
234
235         m += end;
236         l -= end;
237
238         simple_mult((digit *)h, (digit *)h, (digit *)_r, word);
239
240         barrett_reduction((digit *)h, (digit *)h, word);
241     }
242
```

```
243     int i;
244     for(i=0; i<16; i++)
245         c[i] = s[i];
246     for(;i<sz_word;i++)
247         c[i] = 0;
248
249     simple_add((digit *)h, (digit *)h, (digit *)c, word);
250
251     for(j = 0;j < 16;j++)
252         out[j] = h[j];
253 }
```

Código C.7 – secure_nic.h

```
1  #ifndef __SECURE_NIC_H
2  #define __SECURE_NIC_H
3
4  #include <nic.h>
5  #include <thread.h>
6  #include <cpu.h>
7  #include <cipher.h>
8  #include <key_database.h>
9  #include <poly1305.h>
10 #include <diffie_hellman.h>
11
12 __BEGIN_SYS;
13
14 class Secure_NIC : public Diffie_Hellman, Conditional_Observer, public Conditionally_Observed
15 {
16     public:
17     const static unsigned int ID_SIZE = Traits<Secure_NIC>::ID_SIZE;
18
19     Chronometer derive_chrono;
20     Chronometer auth_chrono;
21
22     Secure_NIC(bool is_gateway, Cipher *c, Poly1305 *p, NIC *nic = new NIC(),
23             Key_Database *db = new Key_Database())
24         : Diffie_Hellman(), _has_received_data(false), _update_lock(false), _waiting_dh_response(false),
25         _waiting_auth_response(false), _authenticated(false), accepting_connections(false),
26         _is_gateway(is_gateway), _nic(nic), _keydb(db), _cipher(c), _poly(p)
27     {
28         _nic->attach(this, Traits<Secure_NIC>::PROTOCOL_ID);
29     }
30
31     // Waits for a secure message.
32     // The message is delivered already decrypted.
33     int receive(NIC::Address &from, char *data, unsigned int size)
34     {
35         while(!_has_received_data)
36             Thread::yield();
37         db<Secure_NIC>(INF) << "Secure_NIC Received data" << endl;
38         from = _received_from;
39         for(unsigned int i=0; i < _received_length && i < size; i++)
40             data[i] = _received_data[i];
41         _has_received_data = false;
42         return _received_length;
43     }
44
45     // Used by the sensor.
46     // Sets the ID and AUTH of this sensor.
```

```
47      void set_id(const char _id[ID_SIZE])
48      {
49          for(int i=0;i<ID_SIZE;i++)
50              _serial_number[i] = _id[i];
51          calculate_auth(_auth, _serial_number);
52      }
53      void get_id(char _id[ID_SIZE])
54      {
55          for(int i=0;i<ID_SIZE;i++)
56              _id[i] = _serial_number[i];
57      }
58
59      // Used by the server.
60      // Inserts an ID that should be trusted.
61      bool insert_trusted_id(const char *id)
62      {
63          calculate_auth(_auth, id);
64          return _keydb->insert_peer(id, _auth);
65      }
66      // Used by the server.
67      // This ID should not be trusted anymore.
68      bool remove_trusted_id(const char *id)
69      {
70          calculate_auth(_auth, id);
71          return _keydb->remove_peer(id, _auth);
72      }
73
74      bool next_not_validated_id(char * id)
75      {
76          return _keydb->next_not_validated_id(id);
77      }
78
79      // Used by the sensor.
80      // Requests to start the protocol and establish a key.
81      int send_key_request(NIC::Address dest);
82
83      // Sends a secure message to an already authenticated destination.
84      int send(const NIC::Address &dst, const char *data, unsigned int size);
85
86      // Used by the sensor.
87      // Am I authenticated yet?
88      inline bool authenticated(){ return _authenticated; }
89
90      // Used by the server.
91      // Set to true to start accepting authentication requests.
92      // Set to false to reject authentication requests.
93      volatile bool accepting_connections;
94
95      void update(Conditionally_Observed * o, int p);
96
97      private:
98      // Type of message
99      enum
100     {
101         USER_MSG,
102         DH_REQUEST,
103         DH_RESPONSE,
104         AUTH_REQUEST,
105         AUTH_OK,
106     };
```

```
107
108     // Calculate AUTH code for a given ID
109     void calculate_auth(char out[16], const char id[ID_SIZE])
110     {
111         char tmp[16];
112         char tmp2[16];
113         int i;
114         for(i=0; i<ID_SIZE && i<16; i++)
115         {
116             tmp[i] = id[i];
117             tmp2[i] = id[i];
118             out[i] = 0;
119         }
120         for(; i<16; i++)
121         {
122             tmp[i] = 0;
123             tmp2[i] = 0;
124             out[i] = 0;
125         }
126         _cipher->mode(Cipher::CBC);
127         _cipher->encrypt(tmp, tmp2, out);
128         _cipher->mode(Cipher::CTR);
129
130         db<Secure_NIC>(TRC) << "Secure_NIC::calculate_auth"<< endl;
131         db<Secure_NIC>(TRC) << "id = [";
132         for(i=0;i<ID_SIZE;i++)
133             db<Secure_NIC>(TRC) << ", " << (int)id[i];
134         db<Secure_NIC>(TRC) << "]" << endl;
135         db<Secure_NIC>(TRC) << "auth = [";
136         for(i=0;i<16;i++)
137             db<Secure_NIC>(TRC) << ", " << (int)out[i];
138         db<Secure_NIC>(TRC) << "]" << endl;
139     }
140
141     // Critical sections control
142     void _acquire_lock(volatile bool & lock) { while(CPU::tsl(lock)) Thread::yield(); }
143     void _release_lock(volatile bool & lock) { lock = false; }
144     volatile bool _update_lock;
145
146     // Internal functions
147     NIC * _nic;
148     Key_Database * _keydb;
149     Cipher * _cipher;
150     Poly1305 * _poly;
151     volatile bool _waiting_dh_response;
152     volatile bool _waiting_auth_response;
153     volatile bool _authenticated;
154     NIC::Address _waiting_dh_from;
155     NIC::Address _waiting_auth_from;
156     char _auth[16];
157     char _serial_number[ID_SIZE];
158     int _send_dh_response(NIC::Address dest);
159     int _send_otp(NIC::Address dest);
160     int _send(const NIC::Address &dst, char *data, unsigned int size)
161     {
162         db<Secure_NIC>(INF) << "Secure_NIC: Sending msg of size " << size <<
163             ", header " << (int)data[0] << endl;
164         int ret = _nic->send(dst, Traits<Secure_NIC>::PROTOCOL_ID, data, size);
165         db<Secure_NIC>(TRC) << "Secure_NIC: nic::send returned " << ret << endl;
166         return ret;
```

```
167         }
168         bool _encrypt(char *encrypted_msg, const char *msg, unsigned int size, NIC::Address dst);
169         bool _decrypt(char *decrypted_msg, const char *msg, unsigned int size, NIC::Address from);
170         bool _derive_key(char *key, char *ms, char *sn, bool add_cmac_offset = false);
171         bool _authenticate(const char *msg, NIC::Address from);
172
173         bool _is_gateway;
174
175         NIC::Address _received_from;
176         char _received_data[Traits<CMAC<Radio_Wrapper> >::MTU];
177         unsigned int _received_length;
178         int _receive(NIC::Address &from, char *data, unsigned int size);
179         bool _has_received_data;
180     };
181
182     __END_SYS;
183     #endif
```

Código C.8 – secure_nic.cc

```
1   #include <secure_nic.h>
2   //#include <clock.h>
3   #include <tsc.h>
4   #include <bignum.h>
5
6   __USING_SYS;
7
8   // Sends a secure message to an already authenticated destination.
9   int Secure_NIC::send(const NIC::Address &dst, const char *data, unsigned int size)
10  {
11      // Round up to a multiple of 16, for AES
12      unsigned int sz = size + 15 - (size - 1) % 16;
13
14      // Header + (size rounded up to multiple of 16)
15      char msg[1+sz];
16
17      // Message header
18      msg[0] = USER_MSG;
19
20      // Copy user data and zero-pad
21      unsigned int i;
22      for(i=1;i<=size;i++) msg[i] = data[i-1];
23      for(;i<sz;i++) msg[i]=0;
24
25      // Encrypt the message
26      if (!_encrypt(msg+1, msg+1, sz, dst))
27          return -3;
28
29      // Give the message to lower layer
30      return _send(dst, msg, 1+sz);
31  }
32
33  // Used by the sensor.
34  // Requests to start the protocol and establish a key.
35  // Sends the public Diffie-Hellman key.
36  int Secure_NIC::send_key_request(NIC::Address dest)
37  {
38      // Header + public key
39      char msg[1 + PUBLIC_KEY_SIZE];
40      msg[0] = DH_REQUEST;
```

```
41
42      // Fetch the public Diffie-Hellman key
43      get_public((unsigned char *)(msg+1), PUBLIC_KEY_SIZE);
44
45      _waiting_dh_response = true;
46      _waiting_dh_from = dest;
47
48      // Give the message to lower layer
49      return _send(dest, msg, sizeof(msg));
50  }
51
52  // Response to DH_REQUEST
53  // Sends the public Diffie-Hellman key
54  int Secure_NIC::_send_dh_response(NIC::Address dest)
55  {
56      // Header + public key
57      char msg[1 + PUBLIC_KEY_SIZE];
58      msg[0] = DH_RESPONSE;
59
60      // Fetch the public Diffie-Hellman key
61      get_public((unsigned char *)(msg+1), PUBLIC_KEY_SIZE);
62
63      // Give the message to lower layer
64      return _send(dest, msg, sizeof(msg));
65  }
66
67  // Calculates and sends AUTH + OTP
68  int Secure_NIC::_send_otp(NIC::Address dest)
69  {
70      //Header + AUTH + OTP
71      char msg[1 + 16 + 16];
72      msg[0] = AUTH_REQUEST;
73
74      // Fetch Master Secret
75      if (!_keydb->addr_to_ms(msg+1, dest))
76          return -1;
77
78      // Calculate and attach OTP
79      if (!_derive_key(msg+17, msg+1, _serial_number, true))
80          return -2;
81
82      // Attach AUTH
83      for(int i=0;i<16;i++)
84          msg[i+1] = _auth[i];
85
86      // Give the message to lower layer
87      return _send(dest, msg, sizeof(msg));
88  }
89
90  // Derive the current key and use it to encrypt the message
91  bool Secure_NIC::_encrypt(char *encrypted_msg, const char *msg, unsigned int size, NIC::Address dst)
92  {
93      // Master Secret
94      char ms[SECRET_SIZE];
95      // Current Key
96      char key[16];
97
98      // Fetch master secret
99      if (!_keydb->addr_to_ms(ms,dst))
100         return false;
```

```
101
102         if (_is_gateway)
103         {
104             // Fetch the ID associated to that master secret
105             char sn[ID_SIZE];
106             if (!_keydb->ms_to_sn(sn,ms))
107                 return false;
108             // Calculate the current key
109             if (!_derive_key(key, ms, sn, true))
110                 return false;
111         }
112         // Calculate the current key
113         else if (!_derive_key(key, ms, _serial_number, true))
114                 return false;
115
116         // Encrypt the message, 16 bytes at a time
117         char tmp[16];
118         char tmp2[16];
119         unsigned int i, j;
120         for(i=0;i<size;i+=16)
121         {
122             for(j=0;j<16;j++)
123             {
124                 if(j+i < size)
125                     tmp2[j] = msg[j+i];
126                 else
127                     tmp2[j] = 0;
128             }
129
130             if (!_cipher->encrypt(tmp2, key, tmp))
131                 return false;
132
133             for(j=0;(j<16) && (j+i < size); j++)
134                 encrypted_msg[j+i] = tmp[j];
135         }
136         return true;
137 }
138
139 // Derive the current key and use it to decrypt the message
140 bool Secure_NIC::_decrypt(char *decrypted_msg, const char *msg, unsigned int size, NIC::Address from)
141 {
142         // Master Secret
143         char ms[SECRET_SIZE];
144         // Current Key
145         char key[16];
146
147         // Fetch master secret
148         if (!_keydb->addr_to_ms(ms,from))
149             return false;
150
151         if (_is_gateway)
152         {
153             char sn[ID_SIZE];
154             // Fetch the ID associated to that master secret
155             if (!_keydb->ms_to_sn(sn,ms))
156                 return false;
157             // Calculate the current key
158             if (!_derive_key(key, ms, sn))
159                 return false;
160         }
```

```
161         // Calculate the current key
162         else if (!_derive_key(key, ms, _serial_number))
163                 return false;
164
165         // Decrypt the message, 16 bytes at a time
166         char tmp[16];
167         char tmp2[16];
168         unsigned int i, j;
169         for(i=0;i<size;i+=16)
170         {
171             for(j=0;j<16;j++)
172             {
173                 if(j+i < size)
174                     tmp2[j] = msg[j+i];
175                 else
176                     tmp2[j] = 0;
177             }
178
179             if (!_cipher->decrypt(tmp2, key, tmp))
180                 return false;
181
182             for(j=0;(j<16) && (j+i < size);j++)
183                 decrypted_msg[j+i] = tmp[j];
184         }
185         return true;
186 }
187
188 // Calculate OTP
189 bool Secure_NIC::_derive_key(char *key, char *ms, char *sn, bool add_cmac_offset/*=false*/)
190 {
191         derive_chrono.stop();
192         derive_chrono.reset();
193         derive_chrono.start();
194         char _sn[16];
195         char _ms[16];
196         int i;
197         for(i=0;i<ID_SIZE && i<16;i++)
198             _sn[i] = sn[i];
199         for(;i<16;i++)
200             _sn[i] = 0;
201         for(i=0;i<SECRET_SIZE && i<16;i++)
202             _ms[i] = ms[i];
203         for(;i<16;i++)
204             _ms[i] = 0;
205
206         // Calculating OTP
207         _poly->r(_ms);
208         _poly->k(_sn);
209
210         union
211         {
212             char time[16];
213             unsigned long long s;
214         };
215
216         for(int i=0;i<16;i++)
217             time[i] = 0;
218
219         s = TSC::getMicroseconds();
220         if(add_cmac_offset)
```

```
221             s += (CMAC<Radio_Wrapper>::alarm_event_time_left() +
222                     Traits<CMAC<Radio_Wrapper> >::PREAMBLE_TIME) * 1000;
223
224         // Round up to a multiple of the time window
225         s = s + Traits<Secure_NIC>::TIME_WINDOW - 1 - (s - 1) % Traits<Secure_NIC>::TIME_WINDOW;
226
227         const unsigned int maxsz = ID_SIZE > SECRET_SIZE ? ID_SIZE : SECRET_SIZE;
228         char m[maxsz];
229         for(unsigned int i=0; i<maxsz; i++)
230             m[i] = ms[i % SECRET_SIZE] ^ sn[i % ID_SIZE];
231
232         _poly->authenticate(key, time, m, maxsz);
233
234         derive_chrono.stop();
235         return true;
236     }
237
238     // signed_msg = AUTH + OTP.
239     // Calculate the OTP and check if it matches the one in the msg
240     bool Secure_NIC::_authenticate(const char *signed_msg, NIC::Address dst)
241     {
242         auth_chrono.stop();
243         auth_chrono.reset();
244         auth_chrono.start();
245         db<Secure_NIC>(TRC) << "Secure_NIC::_authenticate()" << endl;
246
247         char sn[ID_SIZE];
248         char ms[SECRET_SIZE];
249         char otp[16];
250
251         // Fetch the ID associated to this AUTH
252         if (!_keydb->auth_to_sn(sn, signed_msg, dst))
253         {
254             db<Secure_NIC>(ERR) << "Secure_NIC::_authenticate() - failed to get sn" << endl;
255             kout << "Secure_NIC::_authenticate() - failed to get sn" << endl;
256             auth_chrono.stop();
257             return false;
258         }
259
260         // Fetch the address associated to this master secret
261         if (!_keydb->addr_to_ms(ms, dst))
262         {
263             db<Secure_NIC>(ERR) << "Secure_NIC::_authenticate() - failed to get ms" << endl;
264             kout << "Secure_NIC::_authenticate() - failed to get ms" << endl;
265             auth_chrono.stop();
266             return false;
267         }
268
269         // Calculate OTP
270         if (!_derive_key(otp, ms, sn))
271         {
272             db<Secure_NIC>(ERR) << "Secure_NIC::_authenticate() - failed to get otp" << endl;
273             kout << "Secure_NIC::_authenticate() - failed to get otp" << endl;
274             auth_chrono.stop();
275             return false;
276         }
277
278         db<Secure_NIC>(INF) << "Secure_NIC::authenticating..." << endl;
279
280         // Check if calculated OTP matches the received one
```

```
281         if ( _poly->isequal(signed_msg+16, otp) )
282         {
283             db<Secure_NIC>(INF)<< "Secure_NIC::authentication granted" << endl;
284             _keydb->validate_peer(sn, ms, signed_msg, dst);
285 
286             // Tell the sensor that it is authenticated
287             char msg[1 + ID_SIZE];
288             if (!_encrypt(msg+1, sn, ID_SIZE, dst))
289             {
290                 db<Secure_NIC>(ERR)<< "Secure_NIC - encryption failed" << endl;
291                 auth_chrono.stop();
292                 return false;
293             }
294             msg[0] = AUTH_OK;
295 
296             auth_chrono.stop();
297             _send(dst, msg, 1 + ID_SIZE);
298 
299             return true;
300         }
301         else
302         {
303             db<Secure_NIC>(ERR) << "Secure_NIC::authentication failed" << endl;
304             kout << "Secure_NIC::authentication failed" << endl;
305             _keydb->remove_peer(ms, dst);
306             auth_chrono.stop();
307             return false;
308         }
309     }
310 
311     void Secure_NIC::update(Conditionally_Observed * o, int p)
312     {
313         _acquire_lock(_update_lock);
314         db<Secure_NIC>(TRC) << "Secure_NIC::update()" << endl;
315         if (p == Traits<Secure_NIC>::PROTOCOL_ID)
316         {
317             NIC::Protocol prot;
318             // Get the message from lower layer
319             _received_length = _nic->receive(&_received_from, &prot, _received_data,
320                     sizeof(_received_data));
321             if (!( prot == Traits<Secure_NIC>::PROTOCOL_ID))
322             {
323                 _release_lock(_update_lock);
324                 return;
325             }
326             if (_received_length <= 0)
327             {
328                 _release_lock(_update_lock);
329                 return;
330             }
331             db<Secure_NIC>(INF) << "Secure_NIC - Received msg of size " << _received_length <<
332                 ", header " << (int)_received_data[0] << endl;
333 
334             unsigned char aux[SECRET_SIZE];
335             switch(_received_data[0])
336             {
337                 case USER_MSG:
338                     // Decrypt the message
339                     if (!_decrypt(_received_data, _received_data+1, _received_length, _received_from))
340                         break;
```

```
341                 _has_received_data = true;
342                 // Notify upper layer
343                 notify (p);
344                 break;
345             case DH_REQUEST:
346                 if (_received_length != PUBLIC_KEY_SIZE + 1)
347                     break;
348                 // Send back Diffie_Hellman public key
349                 _send_dh_response(_received_from);
350                 // Set a Diffie-Hellman Master Secret
351                 calculate_key(aux, SECRET_SIZE, (unsigned char *)(_received_data+1),
352                         PUBLIC_KEY_SIZE);
353                 _keydb->insert_peer((char *)aux, _received_from);
354
355                 // Send authentication message
356                 _waiting_auth_response = true;
357                 _waiting_auth_from = _received_from;
358                 _send_otp(_received_from);
359                 break;
360             case DH_RESPONSE:
361                 if (!_waiting_dh_response || _received_length != PUBLIC_KEY_SIZE + 1)
362                     break;
363                 if (!( _waiting_dh_from == NIC::BROADCAST || _waiting_dh_from == _received_from))
364                     break;
365                 _waiting_dh_response = false;
366                 // Set a Diffie-Hellman Master Secret
367                 calculate_key(aux, SECRET_SIZE, (unsigned char *)(_received_data+1), PUBLIC_KEY_SIZE);
368                 _keydb->insert_peer((char *)aux, _received_from);
369                 break;
370             case AUTH_REQUEST:
371                 if (!accepting_connections || _received_length < 33)
372                     break;
373                 _authenticate(_received_data+1, _received_from);
374                 break;
375             case AUTH_OK:
376                 if (!_waiting_auth_response || _received_length != 1+ID_SIZE)
377                 {
378                     db<Secure_NIC>(ERR) << "Secure_NIC - unexpected msg" << endl;
379                     break;
380                 }
381                 if (!( _waiting_auth_from == _received_from))
382                 {
383                     db<Secure_NIC>(ERR) << "Secure_NIC - address mismatch" << endl;
384                     break;
385                 }
386
387                 char tmp[ID_SIZE];
388                 if (!_decrypt(tmp, (const char *)(_received_data+1), ID_SIZE, _received_from))
389                 {
390                     db<Secure_NIC>(ERR) << "Secure_NIC - decryption failed" << endl;
391                     break;
392                 }
393                 int i;
394                 for(i=0; i<ID_SIZE; i++)
395                     if (tmp[i] != _serial_number[i])
396                         break;
397                 if (i<ID_SIZE)
398                 {
399                     db<Secure_NIC>(ERR) << "Secure_NIC - AUTH_OK msg wrong" << endl;
400                     break;
```

```
401                     }
402                     _waiting_auth_response = false;
403                     _authenticated = true;
404                     // Sensor is authenticated
405                     db<Secure_NIC>(INF) << "Secure_NIC - Authenticated" << endl;
406                     break;
407                 default:
408                     break;
409             }
410         }
411         _release_lock(_update_lock);
412     }
```

Código C.9 – key_database.h

```
1   #ifndef _KEY_DATABASE_H
2   #define _KEY_DATABASE_H
3
4   #include <nic.h>
5   #include <diffie_hellman.h>
6
7   __USING_SYS;
8
9   class Key_Database
10  {
11      const static int  ID_SIZE = Traits<Secure_NIC>::ID_SIZE;
12      const static unsigned int KEY_SIZE = Traits<Diffie_Hellman>::SECRET_SIZE;
13      public:
14      Key_Database();
15      bool validate_peer(const char *sn, const char *ms, const char *auth, const NIC::Address a);
16      bool insert_peer(const char *ms, const NIC::Address a);
17      bool remove_peer(const char *ms, const NIC::Address a);
18      bool insert_peer(const char *sn, const char *auth);
19      bool remove_peer(const char *sn, const char *auth);
20
21      bool auth_to_sn(char *serial_number, const char *auth, const NIC::Address addr);
22  //  bool sn_to_ms(char *master_secret, const char *serial_number, NIC::Address addr);
23      bool addr_to_ms(char *master_secret, const NIC::Address addr);
24      bool ms_to_sn(char *serial_number, const char *master_secret);
25
26      bool next_not_validated_id(char * id);
27
28      private:
29      typedef struct
30      {
31          char serial_number[ID_SIZE];
32          char auth[16];
33          bool free;
34          bool validated;
35      } Known_Node;
36
37      typedef struct
38      {
39          char master_secret[KEY_SIZE];
40          NIC::Address addr;
41          Known_Node * node;
42          bool free;
43      } Authenticated_Peer;
44
45      typedef struct
```

```
46      {
47          char master_secret[KEY_SIZE];
48          NIC::Address addr;
49          bool free;
50      } Weak_Peer;
51
52      Weak_Peer _weak_peers[16];
53      Known_Node _known_nodes[16];
54      Authenticated_Peer _peers[16];
55
56      int _next_id;
57
58      inline bool equals(const char *a, int sza, const char *b, int szb)
59      {
60          while((sza > 0) && ((a[sza-1] == 0) || (a[sza-1] == '0'))) sza--;
61          while((szb > 0) && ((b[szb-1] == 0) || (b[szb-1] == '0'))) szb--;
62          if (sza != szb) return false;
63          for(int i=0;i<sza;i++)
64              if (a[i] != b[i]) return false;
65          return true;
66      }
67  };
68
69  #endif
```

Código C.10 – key_database.cc

```
1   #include <key_database.h>
2   #include <nic.h>
3
4   __USING_SYS;
5
6   Key_Database::Key_Database() : _next_id(0)
7   {
8       for(int i=0;i<16;i++)
9       {
10          _peers[i].free = true;
11          _weak_peers[i].free = true;
12          _known_nodes[i].free = true;
13          _known_nodes[i].validated = false;
14      }
15  }
16
17  bool Key_Database::next_not_validated_id(char * id)
18  {
19      int i = _next_id;
20      do
21      {
22          if (!_known_nodes[i].free && !_known_nodes[i].validated)
23          {
24              for(int j=0; j<ID_SIZE; j++)
25                  id[j] = _known_nodes[i].serial_number[j];
26              _next_id = (i+1) % 16;
27              return true;
28          }
29          i = (i + 1) % 16;
30      } while(i != _next_id);
31      return false;
32  }
33
```

```
34  bool Key_Database::insert_peer(const char *sn, const char *auth)
35  {
36      for(int i=0;i<16;i++)
37      {
38          if (_known_nodes[i].free)
39          {
40              _known_nodes[i].free = false;
41              for(int j=0;j<ID_SIZE;j++)
42                  _known_nodes[i].serial_number[j] = sn[j];
43              for(int j=0;j<16;j++)
44                  _known_nodes[i].auth[j] = auth[j];
45              return true;
46          }
47      }
48      return false;
49  }
50
51  bool Key_Database::remove_peer(const char *sn, const char *auth)
52  {
53      for(int i=0;i<16;i++)
54      {
55          if (!_known_nodes[i].free && equals(_known_nodes[i].serial_number, ID_SIZE, sn, ID_SIZE))
56          {
57              _known_nodes[i].free = true;
58              return true;
59          }
60      }
61      return false;
62  }
63
64  bool Key_Database::insert_peer(const char *ms, const NIC::Address a)
65  {
66      for(int i=0;i<16;i++)
67      {
68          if (_weak_peers[i].free)
69          {
70              _weak_peers[i].free = false;
71              _weak_peers[i].addr = a;
72              for(int j=0;j<KEY_SIZE;j++)
73              {
74                  _weak_peers[i].master_secret[j] = ms[j];
75              }
76              return true;
77          }
78      }
79      return false;
80  }
81
82  bool Key_Database::remove_peer(const char *ms, const NIC::Address a)
83  {
84      for(int i=0;i<16;i++)
85      {
86          if (!_weak_peers[i].free && (a == _weak_peers[i].addr))
87          {
88              _weak_peers[i].free = true;
89              return true;
90          }
91      }
92      return false;
93  }
```

```
94
95  bool Key_Database::auth_to_sn(char *serial_number, const char *auth, const NIC::Address addr)
96  {
97      for(int i=0;i<16;i++)
98      {
99          if (!_known_nodes[i].free && equals(_known_nodes[i].auth, 16, auth, 16))
100         {
101             for(int j=0;j<ID_SIZE;j++)
102                 serial_number[j] = _known_nodes[i].serial_number[j];
103             _peers[i].addr = addr;
104             return true;
105         }
106     }
107     return false;
108 }
109
110 bool Key_Database::addr_to_ms(char *master_secret, const NIC::Address addr)
111 {
112     for(int i=0;i<16;i++)
113     {
114         if (!_peers[i].free && (_peers[i].addr == addr))
115         {
116             for(int j=0;j<KEY_SIZE;j++)
117                 master_secret[j] = _peers[i].master_secret[j];
118             return true;
119         }
120         if (!_weak_peers[i].free && (_weak_peers[i].addr == addr))
121         {
122             for(int j=0;j<KEY_SIZE;j++)
123                 master_secret[j] = _weak_peers[i].master_secret[j];
124             return true;
125         }
126     }
127     return false;
128 }
129
130 bool Key_Database::ms_to_sn(char *serial_number, const char *master_secret)
131 {
132     for(int i=0;i<16;i++)
133         if (!_peers[i].free && equals(_peers[i].master_secret, KEY_SIZE, master_secret, KEY_SIZE))
134         {
135             for(int j=0;j<ID_SIZE;j++)
136                 serial_number[j] = _peers[i].node->serial_number[j];
137             return true;
138         }
139     return false;
140 }
141
142 bool Key_Database::validate_peer(const char *sn, const char *ms, const char *auth, const NIC::Address a)
143 {
144     for(int i=0;i<16;i++)
145     {
146         if (!_weak_peers[i].free && (_weak_peers[i].addr == a) &&
147                 (equals(_weak_peers[i].master_secret, KEY_SIZE, ms, KEY_SIZE)))
148         {
149             for(int j=0;j<16;j++)
150             {
151                 if (!_known_nodes[j].free && !_known_nodes[j].validated &&
152                         equals(_known_nodes[j].serial_number, ID_SIZE, sn, ID_SIZE))
153                 {
```

```
154                         for(int k=0;k<16;k++)
155                         {
156                             if(_peers[k].free)
157                             {
158                                 _peers[k].free = false;
159                                 _peers[k].addr = a;
160                                 _peers[k].node = &_known_nodes[j];
161                                 for(int l=0;l<KEY_SIZE;l++)
162                                     _peers[k].master_secret[l] = ms[l];
163                                 _weak_peers[i].free = true;
164                                 _known_nodes[j].validated = true;
165                                 return true;
166                             }
167                         }
168                     }
169                 }
170             }
171         }
172         return false;
173     }
```

Código C.11 – cipher.h

```
1  #ifndef __CIPHER_H
2  #define __CIPHER_H
3
4  class Cipher
5  {
6      public:
7      enum Mode
8      {
9          CBC,
10         CTR,
11         CTRMAC,
12     };
13     virtual bool encrypt(const char *data, const char *key, char *encrypted_data) = 0;
14     virtual bool decrypt(const char *data, const char *key, char *decrypted_data) = 0;
15     virtual bool mode(Mode m) { _mode = m; return true; };
16     virtual Mode mode() { return _mode; };
17     protected:
18     Mode _mode;
19 };
20
21 #endif
```

Código C.12 – secure_nic_master.cc

```
1  #include <periodic_thread.h>
2  #include <uart.h>
3  #include <gpio_pin.h>
4  #include <ptp.h>
5  #include < utility /observer.h>
6  #include <secure_nic.h>
7  #include <alarm.h>
8  #include <diffie_hellman.h>
9  #include <bignum.h>
10 #include <poly1305.h>
11 #include <mach/mc13224v/aes_controller.h>
12 #include <cipher.h>
13 #include <chronometer.h>
14
```

```
15 //−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
16 // Model application for the Secure_NIC
17 // For more information, refer to doc/security/howto.pdf
18 //−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
19
20 #define SERVER_ADDR NIC::Address(40)
21 #define DH_REQUEST_INTERVAL 12000000
22
23 __USING_SYS
24 OStream cout;
25
26 NIC * nic;
27 PTP * ptp;
28
29
30 // 128−bit key parameters
31 const unsigned char Diffie_Hellman::default_base_point_x[] =
32 {134, 91, 44, 165, 124, 96, 40, 12, 45, 155, 137, 139, 82, 247, 31, 22};
33 const unsigned char Diffie_Hellman::default_base_point_y[] =
34 {131, 122, 237, 221, 146, 162, 45, 192, 19, 235, 175, 91, 57, 200, 90, 207};
35 const unsigned int Bignum::default_mod[4] =
36 {4294967295U, 4294967295U, 4294967295U, 4294967293U};
37 const unsigned int Bignum::default_barrett_u[5] =
38 {17,8,4,2,1};
39 /*
40 // 160−bit key parameters
41 const unsigned char Diffie_Hellman::default_base_point_x[] = {
42     0x82, 0xFC, 0xCB, 0x13, 0xB9, 0x8B, 0xC3, 0x68,
43     0x89, 0x69, 0x64, 0x46, 0x28, 0x73, 0xF5, 0x8E,
44     0x68, 0xB5, 0x96, 0x4A,
45 };
46 const unsigned char Diffie_Hellman::default_base_point_y[] = {
47     0x32, 0xFB, 0xC5, 0x7A, 0x37, 0x51, 0x23, 0x04,
48     0x12, 0xC9, 0xDC, 0x59, 0x7D, 0x94, 0x68, 0x31,
49     0x55, 0x28, 0xA6, 0x23,
50 };
51 const unsigned int Bignum::default_mod[5] =
52 {2147483647U, 4294967295U, 4294967295U, 4294967295U, 4294967295U};
53 const unsigned int Bignum::default_barrett_u[6] =
54 {2147483649, 0, 0, 0, 0, 1};
55 */
56 /*
57 // 192−bit key parameters
58 const unsigned char Diffie_Hellman::default_base_point_x[] = {
59     0x12, 0x10, 0xFF, 0x82, 0xFD, 0x0A, 0xFF, 0xF4,
60     0x00, 0x88, 0xA1, 0x43, 0xEB, 0x20, 0xBF, 0x7C,
61     0xF6, 0x90, 0x30, 0xB0, 0x0E, 0xA8, 0x8D, 0x18,
62 };
63 const unsigned char Diffie_Hellman::default_base_point_y[] = {
64     0x11, 0x48, 0x79, 0x1E, 0xA1, 0x77, 0xF9, 0x73,
65     0xD5, 0xCD, 0x24, 0x6B, 0xED, 0x11, 0x10, 0x63,
66     0x78, 0xDA, 0xC8, 0xFF, 0x95, 0x2B, 0x19, 0x07,
67 };
68 const unsigned int Bignum::default_mod[6] =
69 {4294967295U, 4294967295U, 4294967294U, 4294967295U, 4294967295U, 4294967295U};
70 const unsigned int Bignum::default_barrett_u[7] = {1, 0, 1, 0, 0, 0, 1};
71 */
72 /*
73 // 256−bit key parameters
74 const unsigned char Diffie_Hellman::default_base_point_x[] = {
```

```
75      0x98, 0x17, 0xF8, 0x16, 0x5B, 0x81, 0xF2, 0x59,
76      0xD9, 0x28, 0xCE, 0x2D, 0xDB, 0xFC, 0x9B, 0x02,
77      0x07, 0x0B, 0x87, 0xCE, 0x95, 0x62, 0xA0, 0x55,
78      0xAC, 0xBB, 0xDC, 0xF9, 0x7E, 0x66, 0xBE, 0x79,
79  };
80  const unsigned char Diffie_Hellman::default_base_point_y[] = {
81      0xB8, 0xD4, 0x10, 0xFB, 0x8F, 0xD0, 0x47, 0x9C,
82      0x19, 0x54, 0x85, 0xA6, 0x48, 0xB4, 0x17, 0xFD,
83      0xA8, 0x08, 0x11, 0x0E, 0xFC, 0xFB, 0xA4, 0x5D,
84      0x65, 0xC4, 0xA3, 0x26, 0x77, 0xDA, 0x3A, 0x48,
85  };
86  const unsigned int Bignum::default_mod[8] = {
87      4294967295U, 4294967295U, 4294967295U, 4294967295U,
88      4294967295U, 4294967295U, 4294967294U, 4294966319U
89  };
90  const unsigned int Bignum::default_barrett_u[9] = {1, 0, 0, 0, 0, 0, 0, 1, 977};
91  */
92
93  class Secure_NIC_Listener : public Conditional_Observer
94  {
95      public:
96      Secure_NIC_Listener(Secure_NIC * s)
97      {
98          _s = s;
99          _s->attach(this, Traits<Secure_NIC>::PROTOCOL_ID);
100     }
101
102     // This method will be called when a secure message arrives.
103     // The message is delivered to the application already decrypted.
104     void update(Conditionally_Observed * o, int p)
105     {
106         int sz = _s->receive(from, msg, 32);
107         if (sz<=0) return;
108         cout << "Received encrypted: "<< msg << '\n';
109         cout<<"Sending hi\n";
110         // Reply securely
111         _s->send(from, "Hi Sensor!",11);
112     }
113
114     private:
115     Secure_NIC * _s;
116     NIC::Address from;
117     char msg[32];
118 };
119
120 void led_rgb_r(bool on_off=true)
121 {
122     GPIO_Pin led(10);
123     led.put(on_off);
124 }
125
126 // Send PTP and DH messages
127 int send_message(PTP * ptp, Secure_NIC * s)
128 {
129     char id[Secure_NIC::ID_SIZE];
130     UART uart;
131     char cmd, _id;
132
133     kout << " Init time: " << s->dh_chrono.read() << endl;
134
```

```
135     while(true)
136     {
137         switch(cmd = uart.get())
138         {
139             case 'a':
140             case 'A':
141                 _id = uart.get();
142                 cout << "Adding ID..." << endl;
143                 for(int i=0; i<Secure_NIC::ID_SIZE; i++)
144                     id[i] = _id;
145                 s->insert_trusted_id(id);
146                 break;
147             case 'r':
148             case 'R':
149                 _id = uart.get();
150                 cout << "Removing ID..." << endl;
151                 for(int i=0; i<Secure_NIC::ID_SIZE; i++)
152                     id[i] = _id;
153                 s->remove_trusted_id(id);
154                 break;
155             case 'd':
156             case 'D':
157                 cout << "Sending DH..." << endl;
158                 _id = uart.get();
159                 s->send_key_request(_id);
160                 break;
161             case 'p':
162             case 'P':
163                 cout << "Sending PTP..." << endl;
164                 ptp->doState();
165                 break;
166             case 'q':
167             case 'Q':
168                 goto quit;
169             case 't':
170             case 'T':
171                 cout << "Chronometers: " << endl;
172                 cout << "auth: " << s->auth_chrono.read() << endl;
173                 cout << "derive: " << s->derive_chrono.read() << endl;
174                 cout << "dh: " << s->dh_chrono.read() << endl;
175             default:
176                 ;
177         }
178     }
179 quit:
180     Chronometer c;
181     c.start();
182     while(c.read() > 1000);
183     unsigned int last_ptp = c.read();
184     unsigned int last_dh = c.read();
185     while(true)
186     {
187         if ((c.read() - last_ptp) >= ptp->_ptp_parameters._sync_interval)
188         {
189             cout << "Sending PTP..." << endl;
190             ptp->doState();
191             last_ptp = c.read();
192             cout << "PTP sent" << endl;
193         }
194         if ((c.read() - last_dh) >= DH_REQUEST_INTERVAL)
```

```
195             {
196                 cout << "Sending DH..." << endl;
197                 if (s->next_not_validated_id(id))
198                 {
199                     kout << (int)id[0]  << endl;
200                     s->send_key_request(id[0]);
201                 }
202                 last_dh = c.read();
203                 cout << "DH sent" << endl;
204             }
205         }
206 }
207 int sink()
208 {
209     cout << "Sink running \n";
210     cout << "Key size: "  << Secure_NIC::SECRET_SIZE << endl;
211     cout << "ID size: "  << Secure_NIC::ID_SIZE << endl;
212
213     // Initialize PTP
214     // A sync message will be sent every _sync_interval
215     ptp->_ptp_parameters._sync_interval = 30000000; //microsecond
216     // This is a master node, which will send sync messages
217     ptp->_ptp_parameters._clock_stratum = PTP::PTP_DataSet::STRATUM_MASTER;
218     ptp->_ptp_parameters._original_clock_stratum = PTP::PTP_DataSet::STRATUM_MASTER;
219     ptp->_ptp_parameters._state = PTP::INITIALIZING;
220     ptp->execute();
221
222     Pseudo_Random::seed(127 + Secure_NIC::ID_SIZE + Secure_NIC::SECRET_SIZE);
223
224     // Initialize the secure NIC
225     Secure_NIC * s = new Secure_NIC(true, new AES_Controller(),
226             new Poly1305(new AES_Controller(Cipher::CBC)), nic);
227     kout << "hoho" << endl;
228
229     // This ID is to be trusted
230     char id[Secure_NIC::ID_SIZE];
231     for(int i=0;i<sizeof(id);i++)
232         id[i] = '1';
233     s->insert_trusted_id(id);
234
235     // This ID is to be trusted
236     for(int i=0;i<sizeof(id);i++)
237         id[i] = '2';
238     s->insert_trusted_id(id);
239
240     // This ID is to be trusted
241     for(int i=0;i<sizeof(id);i++)
242         id[i] = '3';
243     s->insert_trusted_id(id);
244
245     // Start accepting authentication requests
246     s->accepting_connections = true;
247
248     // Listen to secure messages
249     Secure_NIC_Listener * l = new Secure_NIC_Listener(s);
250
251     // Thread to send PTP and DH messages
252     Thread * messenger = new Thread(&send_message, ptp, s);//, -1, Thread::READY, 1300);
253
254     cout << "Done!\n";
```

```
255     messenger->join();
256     return 0;
257 }
258
259 int main()
260 {
261     nic = new NIC();
262     ptp = new PTP();
263     ptp->setNIC(nic);
264
265     nic->address(SERVER_ADDR);
266     cout << "Address: " << nic->address() << '\n';
267     sink ();
268     return 0;
269 }
```

Código C.13 – secure_nic_slave.cc

```
1  #include <periodic_thread.h>
2  #include <gpio_pin.h>
3  #include <ptp.h>
4  #include < utility /observer.h>
5  #include <secure_nic.h>
6  #include <alarm.h>
7  #include <diffie_hellman.h>
8  #include <bignum.h>
9  #include <poly1305.h>
10 #include <mach/mc13224v/aes_controller.h>
11 #include <cipher.h>
12 #include <chronometer.h>
13
14 //-------------------------------------------------------------
15 // Model application for the Secure_NIC
16 // For more information, refer to doc/security/howto.pdf
17 //-------------------------------------------------------------
18
19 #define SERVER_ADDR NIC::Address(40)
20 #define MY_ID '1'
21
22 __USING_SYS
23 OStream cout;
24
25 NIC * nic;
26 PTP * ptp;
27
28 // 128-bit key parameters
29 const unsigned char Diffie_Hellman::default_base_point_x[] =
30 {134, 91, 44, 165, 124, 96, 40, 12, 45, 155, 137, 139, 82, 247, 31, 22};
31 const unsigned char Diffie_Hellman::default_base_point_y[] =
32 {131, 122, 237, 221, 146, 162, 45, 192, 19, 235, 175, 91, 57, 200, 90, 207};
33 const unsigned int Bignum::default_mod[4] =
34 {4294967295U, 4294967295U, 4294967295U, 4294967293U};
35 const unsigned int Bignum::default_barrett_u[5] = {17,8,4,2,1};
36 /*
37 // 160-bit key parameters
38 const unsigned char Diffie_Hellman::default_base_point_x[] = {
39     0x82, 0xFC, 0xCB, 0x13, 0xB9, 0x8B, 0xC3, 0x68,
40     0x89, 0x69, 0x64, 0x46, 0x28, 0x73, 0xF5, 0x8E,
41     0x68, 0xB5, 0x96, 0x4A,
42 };
```

```
43  const unsigned char Diffie_Hellman::default_base_point_y[] = {
44      0x32, 0xFB, 0xC5, 0x7A, 0x37, 0x51, 0x23, 0x04,
45      0x12, 0xC9, 0xDC, 0x59, 0x7D, 0x94, 0x68, 0x31,
46      0x55, 0x28, 0xA6, 0x23,
47  };
48  const unsigned int Bignum::default_mod[5] =
49  {2147483647U, 4294967295U, 4294967295U, 4294967295U, 4294967295U};
50  const unsigned int Bignum::default_barrett_u[6] =
51  {2147483649, 0, 0, 0, 0, 1};
52  */
53  /*
54  // 192-bit key parameters
55  const unsigned char Diffie_Hellman::default_base_point_x[] = {
56      0x12, 0x10, 0xFF, 0x82, 0xFD, 0x0A, 0xFF, 0xF4,
57      0x00, 0x88, 0xA1, 0x43, 0xEB, 0x20, 0xBF, 0x7C,
58      0xF6, 0x90, 0x30, 0xB0, 0x0E, 0xA8, 0x8D, 0x18,
59  };
60  const unsigned char Diffie_Hellman::default_base_point_y[] = {
61      0x11, 0x48, 0x79, 0x1E, 0xA1, 0x77, 0xF9, 0x73,
62      0xD5, 0xCD, 0x24, 0x6B, 0xED, 0x11, 0x10, 0x63,
63      0x78, 0xDA, 0xC8, 0xFF, 0x95, 0x2B, 0x19, 0x07,
64  };
65  const unsigned int Bignum::default_mod[6] =
66  {4294967295U, 4294967295U, 4294967294U, 4294967295U, 4294967295U, 4294967295U};
67  const unsigned int Bignum::default_barrett_u[7] = {1, 0, 1, 0, 0, 0, 1};
68  */
69  /*
70  // 256-bit key parameters
71  const unsigned char Diffie_Hellman::default_base_point_x[] = {
72      0x98, 0x17, 0xF8, 0x16, 0x5B, 0x81, 0xF2, 0x59,
73      0xD9, 0x28, 0xCE, 0x2D, 0xDB, 0xFC, 0x9B, 0x02,
74      0x07, 0x0B, 0x87, 0xCE, 0x95, 0x62, 0xA0, 0x55,
75      0xAC, 0xBB, 0xDC, 0xF9, 0x7E, 0x66, 0xBE, 0x79,
76  };
77  const unsigned char Diffie_Hellman::default_base_point_y[] = {
78      0xB8, 0xD4, 0x10, 0xFB, 0x8F, 0xD0, 0x47, 0x9C,
79      0x19, 0x54, 0x85, 0xA6, 0x48, 0xB4, 0x17, 0xFD,
80      0xA8, 0x08, 0x11, 0x0E, 0xFC, 0xFB, 0xA4, 0x5D,
81      0x65, 0xC4, 0xA3, 0x26, 0x77, 0xDA, 0x3A, 0x48,
82  };
83  const unsigned int Bignum::default_mod[8] = {
84      4294967295U, 4294967295U, 4294967295U, 4294967295U,
85      4294967295U, 4294967295U, 4294967294U, 4294966319U
86  };
87  const unsigned int Bignum::default_barrett_u[9] = {1, 0, 0, 0, 0, 0, 0, 1, 977};
88  */
89
90
91  class Secure_NIC_Listener : public Conditional_Observer
92  {
93      public:
94      Secure_NIC_Listener(Secure_NIC * s)
95      {
96          _s = s;
97          _s->attach(this, Traits<Secure_NIC>::PROTOCOL_ID);
98      }
99
100     // This method will be called when a secure message arrives.
101     // The message is delivered to the application already decrypted.
102     void update(Conditionally_Observed * o, int p)
```

```
103     {
104         int sz = _s->receive(from, msg, 32);
105         if (sz<=0) return;
106         cout << "Received encrypted: "<< msg << '\n';
107     }
108
109     private:
110     Secure_NIC * _s;
111     NIC::Address from;
112     char msg[32];
113 };
114 int sensor()
115 {
116     Chronometer c;
117     NIC::Address from;
118     cout << "Sensor running \n";
119     cout << "Key size: " << Secure_NIC::SECRET_SIZE << endl;
120     cout << "ID size: " << Secure_NIC::ID_SIZE << endl;
121
122     // Initialize PTP
123     // This is a slave node, which will respond to sync messages
124     ptp->_ptp_parameters._original_clock_stratum = PTP::PTP_DataSet::STRATUM_SLAVE;
125     ptp->_ptp_parameters._clock_stratum = PTP::PTP_DataSet::STRATUM_SLAVE;
126     ptp->_ptp_parameters._state = PTP::INITIALIZING;
127     ptp->execute();
128
129     // Initialize this sensor's ID
130     char id[Secure_NIC::ID_SIZE];
131     unsigned long long seed = 0;
132     for(int i=0;i<sizeof(id);i++)
133     {
134         id[i] = MY_ID;
135         seed += id[i];
136     }
137     Pseudo_Random::seed(seed);
138
139     c.start();
140
141     // Initialize the secure NIC
142     Secure_NIC * s = new Secure_NIC(false, new AES_Controller(),
143             new Poly1305(new AES_Controller(Cipher::CBC)), nic);
144     s->set_id(id);
145     unsigned int t = s->dh_chrono.read();
146
147     while(!s->authenticated());
148
149     c.stop();
150
151     cout << "Key set!\nTotal time: " << c.read() << endl;
152
153     kout << "Init time: " << t << endl;
154     kout << "Key establish time: " << s->dh_chrono.read() << endl;
155
156     /*
157     do
158     {
159         Alarm::delay((Pseudo_Random::random() % 10000000));
160         if (!s->authenticated())
161         {
162             cout << "Trying to negotiate key\n";
```

```
163                 // Try to authenticate
164                 s->send_key_request(SERVER_ADDR);
165             }
166         }
167         while(!s->authenticated());
168         */
169
170         // At this point, this sensor is authenticated and shares
171         // a symmetric key with the server
172
173         c.stop();
174
175         cout << "Key set!\nTotal time: " << c.read() << endl;
176
177         // Listen to secure messages
178         Secure_NIC_Listener listener(s);
179
180         UART uart;
181         char msg[16];
182         while(true)
183         {
184             int i;
185             for(i=0;i<16;i++)
186                 msg[i] = 0;
187
188             for(i=0;i<16;i++)
189             {
190                 msg[i] = uart.get();
191                 if(msg[i] == '\n')
192                     break;
193             }
194             while(uart.has_data())
195                 uart.get();
196             if(msg[0] == 't' && msg[1] == '\n')
197             {
198                 cout << "Chronometers: " << endl;
199                 cout << "auth: " << s->auth_chrono.read() << endl;
200                 cout << "derive: " << s->derive_chrono.read() << endl;
201                 cout << "dh: " << s->dh_chrono.read() << endl;
202             }
203             else
204             {
205                 //      Alarm::delay(10000000 + (Pseudo_Random::random() % 23370000));
206                 cout << "Sending message\n";
207                 // Send an encrypted message to the server
208                 s->send(SERVER_ADDR, msg, i);
209             }
210         }
211
212         cout << "Done!\n";
213         Thread::exit();
214         return 0;
215     }
216
217
218     void led_rgb_r(bool on_off=true)
219     {
220         GPIO_Pin led(10);
221         led.put(on_off);
222     }
```

```
223 int main()
224 {
225     nic = new NIC();
226     ptp = new PTP();
227     ptp->setNIC(nic);
228
229     nic->address(MY_ID);
230     cout << "Address: " << nic->address() << '\n';
231     sensor();
232     return 0;
233 }
```