UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Lucas Machado da Palma, Luiz Henrique Urias de Sousa

ADIÇÃO DE FUNÇÕES DE CRIPTOGRAFIA SIMÉTRICA EM UM APPLET DE CÓDIGO ABERTO COM SUPORTE VIA MIDDLEWARE OPENSC

Florianópolis

2017

Lucas Machado da Palma, Luiz Henrique Urias de Sousa

ADIÇÃO DE FUNÇÕES DE CRIPTOGRAFIA SIMÉTRICA EM UM APPLET DE CÓDIGO ABERTO COM SUPORTE VIA MIDDLEWARE OPENSC

Trabalho de Conclusão de Curso submetido ao Curso de Ciências da Computação para a obtenção do Grau de Bacharelado em Ciências da Computação.
Orientador: Prof. Dr. Jean Everson Martina
Coorientador: Me. Lucas Pandolfo Perin

Florianópolis

2017

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

Urias, Palma, Luiz, Lucas
Adição de funções de criptografia simétrica em um applet de código aberto com suporte via middleware OpenSC / Luiz, Lucas Urias, Palma ; orientador, Jean Everson Martina, coorientador, Lucas Perin, 2017.
133 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2017.

Inclui referências.

1. Ciências da Computação. 2. Java Cards. 3. Applets. 4. Criptografia Simétrica. 5. OpenSC. I. Martina, Jean Everson. II. Perin, Lucas. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

Lucas Machado da Palma, Luiz Henrique Urias de Sousa

ADIÇÃO DE FUNÇÕES DE CRIPTOGRAFIA SIMÉTRICA EM UM APPLET DE CÓDIGO ABERTO COM SUPORTE VIA MIDDLEWARE OPENSC

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

Florianópolis, 01 de junho 2017.

____

Prof. Dr. Rafael Luiz Cancian
Coordenador do Curso

Banca Examinadora:

____

Prof. Dr. Jean Everson Martina
Orientador

____

Me. Lucas Pandolfo Perin
Coorientador

____

Me. Rick Lopes de Souza

Este trabalho é dedicado aos bits e bytes
das nossas aplicações.

AGRADECIMENTOS

Em um momento como este a questão mais óbvia é: "A quem eu devo agradecer?". E o temor de esquecer nomes é um tormento. Irei correr o risco neste pequeno texto de agradecimento.

Primeiro, agradeço a minha família. Agradeço o meu pai João Alberto, a minha mãe Sirlei e meu irmão João Victor, pelo apoio incondicional durante todo o período de desenvolvimento deste trabalho, assim como em todos os momentos da minha vida. Agradeço a todos os meus amigos e colegas, que estiveram comigo até este momento, mas em especial a dois deles: André Weber e Vinícius Barcelos. Amigos de longa data, que sem sombra de dúvidas me auxiliaram mesmo com as atitudes mais simples, como um cafezinho antes das aulas.

Agradeço o apoio dos meus colegas do laboratório de segurança em computação (LabSEC). Ao professor Jean Martina e ao mestre Lucas Perin pela orientação. E por fim, mas não menos importante, agradeço ao meu amigo Luiz Urias, sem o qual este trabalho não estaria completo.

Lucas Machado da Palma

AGRADECIMENTOS

Primeiramente e especialmente, agradeço à minha falecida mãe Lucineide Jeski, por todos os bons momentos vividos e por toda a base que me tornou o que sou hoje, sei que em algum lugar você está observando tudo. Segundamente, agradeço ao meu pai Luiz Antônio e minha irmã Julia Urias, por toda a força após esta perda, sem esse apoio não terminaria este trabalho.

Agradeço também aos meus amigos Mateus, Anderson e Felipe, amigos de longa data que considero irmãos, e que, me ajudaram em momentos importantes da minha vida. Agradeço também à Marina Barufi, pela longa amizade mesmo que longe e por ter ajudado na revisão deste trabalho.

Por último, agradeço ao laboratório de segurança em computação por toda a oportunidade dada, mas em especial ao Jean Martina e Lucas Perin pela orientação e Lucas Palma por ter aceitado fazer este trabalho junto comigo, sem o qual este trabalho não estaria completo.

Luiz Henrique Urias de Sousa

"I'm not great at the advice. Can I interest you in a sarcastic comment?"

Chandler Bing

RESUMO

O sigilo dos dados é um dos principais pilares da segurança em computação. Dentre as diferentes formas para alcançá-lo, vale destacar o uso de chaves secretas. Com elas, pode-se cifrar informações sensíveis utilizando algoritmos de criptografia simétrica em que a chave é compartilhada entre o remetente e o destinatário. A mesma chave realiza as operações de cifragem e decifragem dos dados. Neste contexto, o maior problema é o armazenamento da chave de forma segura. Dentre as possíveis maneiras de armazená-la, pode-se citar: banco de dados local, servidores e *Smart Cards*. Sendo o enfoque deste trabalho, o último, que fornece portabilidade e segurança. Embora sua relevância, a maioria das aplicações que implementam criptografia simétrica em *Smart Cards*, são de código proprietário. Este trabalho busca uma alternativa utilizando *Java Cards*, OpenSC e o cifrador AES, implementando as principais funções de criptografia simétrica em um *applet* de código aberto. Como resultado, obteve-se um *applet open source* suportado pelo *middleware* OpenSC, capaz de realizar as principais funções de criptografia simétrica.

Palavras-chave: java card. applet. opensc. aes. open source.

ABSTRACT

Data confidentiality is one of the main pillars of computer security. Among the different ways to reach it, it is worth highlighting the use of secret keys. With them, sensitive information can be encrypted using symmetric encryption algorithms in which the key is shared between the sender and the recipient. The same key performs data encryption and decryption operations. In this context, the biggest problem is securely storing the key. Among the possible ways to store it, we can mention: local database, servers and Smart Cards. Being the focus of this work, the latter, which provides portability and security. Although its relevance, most applications that implement symmetric encryption on Smart Cards, are proprietary code. This work looks for an alternative using Java Cards, OpenSC and the AES cipher, implementing the main functions of symmetric cryptography in an open source applet. As a result, we obtained an open source applet supported by the OpenSC middleware, capable of performing the main symmetric cryptographic functions.
Keywords: java card. applet. opensc. aes. open source.

LISTA DE FIGURAS

Figura 1 Cifragem e decifragem simétrica. Fonte: (STALLINGS, 2014) 30
Figura 2 Parâmetros do AES. Fonte: (STALLINGS, 2014) 31
Figura 3 Estrutura de um Java Card. Fonte: (GRINGERI, 2008-2009) 33
Figura 4 Application Identifier. Fonte: (GRINGERI, 2008-2009) .. 34
Figura 5 Estrutura de um command APDU. Fonte: (GRINGERI, 2008-2009) 35
Figura 6 Estrutura de um response APDU. Fonte: (GRINGERI, 2008-2009) 35
Figura 7 *Command* APDU e *Response* APDU caso 1. Adaptado de: (GRINGERI, 2008-2009) 36
Figura 8 *Command* APDU e *Response* APDU caso 2. Adaptado de: (GRINGERI, 2008-2009) 36
Figura 9 *Command* APDU e *Response* APDU caso 3. Adaptado de: (GRINGERI, 2008-2009) 36
Figura 10 *Command* APDU e *Response* APDU caso 4. Adaptado de: (GRINGERI, 2008-2009) 36
Figura 11 Ciclo de vida de um applet. Fonte: (GRINGERI, 2008-2009) 38
Figura 12 Alguns padrões de INS. Fonte: (ISO/IEC, 2005) 41
Figura 13 Estrutura do formato TLV. Fonte: Wikipédia - A enciclopédia livre. 41
Figura 14 Hierarquia de dados PKCS. Fonte: (RSA, 2009) 43
Figura 15 Arquitetura do OpenSC. Fonte: (OPENSC, 2017) 46
Figura 16 Diagrama de classes UML simplificado do Iso Applet. Fonte: Criada pelos autores. 48
Figura 17 Criação da estrura PKCS#15. Fonte: Criada pelos autores. 49
Figura 18 Saída do comando *-D* para a estrutura PKCS#15 e PIN. Fonte: Criada pelos autores. 49
Figura 19 Saída do comando *-D* para o par de chaves RSA. Fonte: Criada pelos autores. 50
Figura 20 Fluxograma simplificado da geração de chaves simétricas.

Fonte: Criada pelos autores. . . . 61
Figura 21 Memória do *Java Card* antes do upload do applet. Fonte: Criada pelos autores. . . . 70
Figura 22 Memória do *Java Card* após o upload do applet. Fonte: Criada pelos autores. . . . 70
Figura 23 Memória do *Java Card* após a geração de uma chave AES 128 bits. Fonte: Criada pelos autores. . . . 72
Figura 24 Memória do *Java Card* após a geração de uma chave AES 256 bits. Fonte: Criada pelos autores. . . . 73
Figura 25 Resultado da cifragem da palavra "labsec". Fonte: Criada pelos autores. . . . 73
Figura 26 Resultado da decifragem do dado no passo anterior. . . . 73
Figura 27 Tela inicial do *Java Card* Manager. . . . 74
Figura 28 Exemplo de fluxo de telas no JCM. Fonte: Criada pelos autores. . . . 75

LISTA DE TABELAS

Tabela 1 Tabela de descrição dos códigos de um command APDU 35
Tabela 2 Tópicos da ISO 7816 39
Tabela 3 Tópicos do PKCS 42
Tabela 4 Exemplo de funções definidas no PKCS#11 44
Tabela 5 Comparação entre os *applets* 47
Tabela 6 Protótipo Iso Applet 69
Tabela 7 Protótipo OpenSC 69

LISTA DE ABREVIATURAS E SIGLAS

AET Advanced Encryption Technology 25
ISO International Organization for Standardization 25
PKCS Public-Key Cryptography Standards 25
API Application Programming Interface 25
DES Data Encryption Standard 26
AES Advanced Encryption Standard 26
RSA Rivest, Shamir e Adleman 27
EC Elliptic Curve 27
NIST National Institute of Standards and Technology 30
ROM Read-only memory 31
RAM Random Access Memory 31
EEPROM Electrically-Erasable Programmable Read-Only Memory 31
PIN Personal Identification Number 32
RE Runtime Enviroment 33
VM Virtual Machine 33
AID Application Identifier 33
RID Resource Identifier 33
PIX Proprietary Identifier Extension 33
CAP Converted Applet 34
APDU Application Protocol Data Unit 34
SW Status Word 35
DF Dedicated File 40
EF Elementary File 40
MF Mandatory File 40
MIT Massachusetts Institute of Technology 42
Cryptoki Cryptographic Token Interface 42
IC Integrated Circuit 44
ODF Object Directory File 45
PrKDFs Private Key Directory File 45
PuKDFs Public Key Directory File 45
SKDFs Secret Key Directory File 45

SUMÁRIO

1 INTRODUÇÃO 25
1.1 OBJETIVOS 27
1.1.1 Objetivos Específicos 27
1.2 METODOLOGIA 27
1.3 ORGANIZAÇÃO DO TRABALHO 28
2 FUNDAMENTAÇÃO TEÓRICA 29
2.1 CRIPTOGRAFIA SIMÉTRICA 29
2.1.1 Advanced Encryption Standard 30
2.2 SMART CARD 31
2.2.1 Java Card 32
2.3 JAVA CARD APPLETS 33
2.3.1 Protocolo de comunicação 34
2.3.2 Ciclo de vida de um applet 37
2.4 GLOBALPLATFORM 38
2.5 ISO 7816 39
2.5.1 ISO 7816-4 39
2.6 PKCS 41
2.6.1 PKCS#11 43
2.6.2 PKCS#15 44
2.7 PC/SC 45
2.8 OPENSC 45
3 DESENVOLVIMENTO 47
3.1 ESCOLHA DO APPLET 47
3.2 ISO APPLET 47
3.2.1 Adições no Iso Applet 50
3.2.1.1 Funções implementadas 55
3.3 SUPORTE VIA MIDDLEWARE OPENSC 60
4 RESULTADOS 69
4.1 TESTES COM OS PROTÓTIPOS 70
4.1.1 Instalação do applet 70
4.1.2 Geração das chaves 71
4.1.3 Outras funções oferecidas 72
4.2 JAVA CARD MANAGER 74
5 CONCLUSÕES 77
5.1 TRABALHOS FUTUROS 77
6 DIREITOS AUTORAIS 79
REFERÊNCIAS 81

APÊNDICE A – Comunicação com applet via APDU . . . 85
APÊNDICE B – Artigo 91
ANEXO A – Códigos Iso Applet 105
ANEXO B – Códigos OpenSC 117

1 INTRODUÇÃO

Ainda que muitas vezes despercebidos, os *Smart Cards* estão presentes na realidade da maioria das pessoas que vive nos grandes centros. Adotando, em alguns casos, os papeis de cartões de crédito ou cartões de autenticação, são equipamentos dotados de poder computacional de propósito específico.

No contexto de segurança em computação, são necessários cartões capazes de realizar alguma função criptográfica, como a criação de chaves ou cifragem e decifragem de dados. Pode-se encontrar no mercado, uma série de empresas especializadas neste tipo de aplicação, assim como projetos de código aberto mantidos por desenvolvedores especializados.

Empresas como a europeia AET, líder na área de segurança digital, oferecem soluções de autenticação com *Smart Cards* seguindo padrões internacionalmente estabelecidos, como a ISO 7816 (ISO/IEC, 2005) e o *PKCS* (Public-Key Cryptography Standards) (LABORATORIES, 2017). Uma variação importante dos *Smart Cards* são os *Java Cards* (ORACLE, 2017), que executam aplicações codificadas na linguagem de programação Java, denominadas *applets*. Para estes, pode-se encontrar uma coleção de *applets* criptográficos de código aberto, disponíveis em serviços de repositório como o GitHub. A importância destes projetos está na liberdade de acesso e contribuição por parte de pesquisadores ou empresas, que optam por soluções *Open Source*.

Utilizado pela maioria das Autoridades Certificadoras e *softwares* que acessam *Smart Cards*, o padrão PKCS#11 (RSA, 2009) define um compilado de regras e funções para *tokens* criptográficos. Em conjunto com outros padrões como GlobalPlatform (GLOBALPLATFORM, 2017) e ISO 7816, possibilita a criação de cartões capazes de realizar funções como: geração de chaves simétricas e assimétricas, cifragem e decifragem de dados e armazenamento de certificados digitais.

As aplicações contidas nos cartões, devem ser instruídas, quanto as funções a serem realizadas. O responsável por esse gerenciamento é conhecido como *middleware*, ou mediador. Um *middleware* bastante difundido é o OpenSC, projeto de código aberto, com mais de dez anos de história e dezenas de contribuidores, além de estar presente em repositórios oficiais de sistemas operacionais como *Red Hat* e *Ubuntu*.

O OpenSC implementa parte significativa da API PKCS#11, principalmente as funções relacionadas a autenticação, cifragem de correio eletrônico e assinatura digital. Porém não inclui àquelas ligadas a

criptografia simétrica, utilizando algoritmos como *DES* (Data Encryption Standard) e *AES* (Advanced Encryption Standard), como pode ser verificado na versão original do projeto (OPENSC, 2017). Sendo esse, um dos prováveis motivos pelos quais, a maioria dos *applets* de código aberto encontrados, também não suportam estas funções.

Busca-se selecionar, dentre diversas opções, um *applet* de código aberto que implemente o maior número de funções definidas pelo padrão PKCS#11 e que respeite outras normas internacionais, como a ISO 7816. Muscle Applet e Iso Applet são exemplo das aplicações cogitadas, mas que se diferem em determinados pontos.

Em uma comparação considerando parâmetros como: recorrência de atualizações, qualidade de código, documentação e respeito à normas internacionais, o Iso Applet se destaca das demais aplicações avaliadas. Desenvolvido originalmente por Philip Wendland (WENDLAND, 2016), o *applet* segue, com rigor, as normas estabelecidas na ISO 7816 e possui suporte no *middleware* OpenSC. Porém, como destacado anteriormente, não implementa funções de criptografia simétrica, que por sua vez não são suportadas no mediador.

Visando colaborar com a comunidade *Open Source* e conceber um *applet* com um maior número de funções definidas pelo padrão PKCS#11, este trabalho adiciona a geração, o carregamento e a exportação de chaves AES, nos tamanhos de 128 e 256 bits. Além da cifragem e decifragem, utilizando estas chaves no *applet* desenvolvido por Wendland. Mais do que isso, também é demostrado como inserir apoio à uma destas funcionalidades na arquitetura do OpenSC.

Busca-se compreender e executar o desenvolvimento de aplicações para *Java Cards*, considerando suas particularidades em relação ao desenvolvimento de *software* de propósito geral, por exemplo, a comunicação que se dá entre o cartão e o mediador.

Ao final, são resultados desta pesquisa: levantamento das funções PKCS#11 oferecidas pelo *middleware* OpenSC, seleção de um *applet* de código aberto a ser estendido, compreensão do desenvolvimento de *applets*, adição de funções de criptográfica simétrica no *applet* escolhido, suporte de uma das funções no OpenSC e o desenvolvimento de uma aplicação de auxílio na instalação de *applets* nos *Java Cards*. Não estão incluídos no projeto a geração e uso de chaves AES 192 bits, bem como o uso de diferentes modos de operação, além do CBC, na função de cifragem de dados.

1.1 OBJETIVOS

O objetivo deste trabalho é complementar um *Java Card applet* de código aberto, suportado pelo *middleware OpenSC*, adicionando funções de criptografia simétrica e seguindo padrões internacionais.

1.1.1 Objetivos Específicos

- Avaliação dos *applets* desenvolvidos pela comunidade *open source.*
- Adição das funções de geração, importação, exportação de chaves AES e cifragem e decifragem de dados no *applet.*
- Implementação do suporte à função de geração da chave secreta no *OpenSC.*
- Desenvolvimento de uma aplicação para instalação de *applets* nos cartões.

1.2 METODOLOGIA

Afim de alcançar os objetivos propostos, primeiramente fez-se um levantamento dos *Java Card Applets* de código aberto disponíveis nos serviços de repositório na *web*. Dentre as aplicações com foco em criptografia encontradas, foi selecionada aquela em maior conformidade com os requisitos estabelecidos.

Em seguida, realizou-se um estudo da implementação original do *applet* escolhido, afim de compreender o papel de cada uma das classes que o compõe. Testes foram realizados com esta versão, em cenários reais de segurança em computação. Finalmente, foram adicionadas as novas funções de criptografia simétrica propostas. Tem-se como resultado desta etapa, um protótipo de aplicação, completamente utilizável, capaz de armazenar chaves assimétricas (RSA e EC), chaves simétricas (AES) e certificados digitais. Além de realizar operações de cifragem e decifragem de dados, com ambas opções de chave.

Para as implementações no *middleware* OpenSC, fez-se um estudo do fluxo de processamento de uma função já implementada (geração de chaves RSA), utilizada como referência, dada a complexidade do projeto. Em seguida a geração de chaves AES nos tamanhos 128 e 256 bits, foi implementada.

1.3 ORGANIZAÇÃO DO TRABALHO

Os próximos capítulos deste trabalho estão organizados como segue: No capítulo 2 é feita uma apresentação dos principais conceitos envolvidos no problema abordado. O terceiro capítulo, desenvolvimento, apresenta: a adição das funções de criptografia simétrica no Iso Applet e a implementação do suporte de uma delas no OpenSC. No capítulo 4, busca-se mostrar os resultados do trabalho. Por fim, no quinto e último capítulo é feita a conclusão e listagem dos trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 CRIPTOGRAFIA SIMÉTRICA

A criptografia em geral pode ser caracterizada em três dimensões distintas (STALLINGS, 2014):

- O tipo de operação usada para transformação dos dados originais nos dados cifrados.
- O número de chaves utilizadas.
- A forma como os dados de entrada são processados.

No caso dos cifradores simétricos, pode-se assumir que o número de chaves usadas sempre será um, variando as operações e o tratamento dos dados entre uma técnica e outra.

Cifradores simétricos, também conhecidos como cifradores convencionais, são, ainda hoje, os mais utilizados em comparação a outras técnicas de criptografia, como os cifradores de chave pública (assimétricos). Mundialmente, os algoritmos de criptografia simétrica mais usados são: DES (Data Encryption Standard) e AES (Advanced Encryption Standard).

Historicamente pode-se particionar o desenvolvimento dos cifradores simétricos em duas fases distintas: antes e depois da era dos computadores. Técnicas de cifragem de dados são utilizadas há séculos pela humanidade, baseadas em técnicas de substituição, como os cifradores de Caesar e Playfair, ou ainda utilizando transposições, como é o caso do Rail Fence.

Com o advento dos primeiros mainframes — computadores de grande porte, dedicados ao processamento de grandes quantidade de informação — houve um ganho considerável no poder de processamento de dados, tornando as técnicas clássicas de criptografia ultrapassadas, uma vez que as cifras eram facilmente quebradas utilizando a força bruta (uma bateria de testes com todas as possibilidades).

A corrida pela criação de novos algoritmos resultou no desenvolvimento de diferentes cifradores em bloco, como o DES, o cifrador mais utilizado mundialmente, mas que vem sendo substituído pelo AES.

O modelo de cifragem simétrica pode ser dividido em cinco itens:

- Algoritmo Criptográfico - algoritmo que manipula os dados originais através de substituições e transformações.

- Chave Secreta - valor usualmente numérico, representado por uma sequência de n bits, onde n é o tamanho da chave, por exemplo, 56 bits no caso do DES.
- Texto Plano - conjunto de dados original, o qual se quer cifrar.
- Cifra - dados resultantes do final da cifragem. Deseja-se que os dados resultantes não aparentem ter qualquer relação com a informação original aos olhos de um potencial atacante.

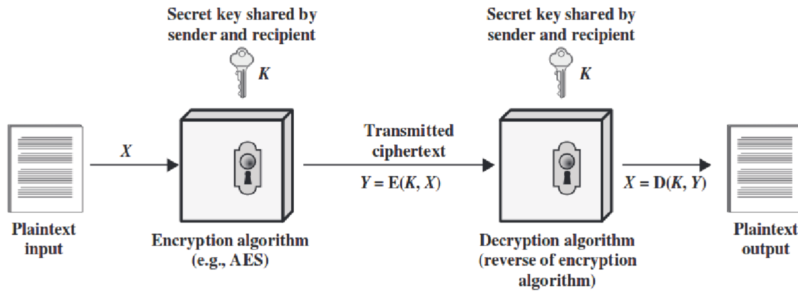

Figura 1 Cifragem e decifragem simétrica. Fonte: (STALLINGS, 2014)

Os processos de cifragem e decifragem de dados utilizam a mesma chave secreta, que deve ser compartilhada entre as partes no caso de trocas de mensagem. Os resultados da aplicação dos algoritmos estão fortemente relacionados à chave escolhida. Por esse motivo, a preocupação principal com relação a privacidade esta relacionada à chave, enquanto os algoritmos são conhecidos e abertos à criptoanálise.

Dentre as diferentes maneiras possíveis de armazenar as chaves secretas de forma segura, este trabalho explorará o uso de Smart Cards para tanto.

2.1.1 Advanced Encryption Standard

O padrão AES foi publicado em 2001 pelo NIST (National Institute of Standards and Technology), resultante da busca por um algoritmo que substituísse o padrão até então utilizado, DES, que, devido ao legado, é o algoritmo mais popular mundialmente.

O AES foi escolhido sob critérios de avaliação, que envolviam desde o nível de segurança do algoritmo até o custo de licenciamento e requisitos de memória para implementações em software e hardware.

Trata-se de um cifrador de blocos de 16 bytes (128 bits), ou seja, os dados de entrada são sempre divididos em porções de 128 bits. O padrão aceita três tamanhos distintos de chave: 16 bytes (128 bits), 24 bytes (192 bits) e 32 bytes (256 bits).

A cifragem e decifragem consistem em n rodadas, definidas pelo tamanho da chave escolhida, cada rodada efetua uma série de manipulações sobre o dado original.

Key Size (words/bytes/bits)	4/16/128	6/24/192	8/32/256
Plaintext Block Size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Number of Rounds	10	12	14
Round Key Size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Expanded Key Size (words/bytes)	44/176	52/208	60/240

Figura 2 Parâmetros do AES. Fonte: (STALLINGS, 2014)

2.2 SMART CARD

A maioria das pessoas não percebem, mas os *Smart Cards* estão presentes em seu cotidiano. Cartões de crédito e cartões SIM (chip para celulares) são exemplos de *Smart Cards*. Eles são equipados com um microprocessador e diferentes tipos de memória, podendo-se resumir à cartões com alguma capacidade computacional. (GRINGERI, 2008-2009) Existem três tipos de memória em um *Smart Card*:

- ROM - utilizada para armazenar programas fixos. Não podem ser escritas após a criação do cartão.
- RAM - não é persistente e é utilizada para armazenar dados que são modificáveis. Assim como no computador, os dados armazenados não são guardados após a fonte de energia desligar.
- EEPROM - junção das duas memórias anteriores, onde os dados são mantidos após um desligamento e é possível modificá-los.

Porém, capacidade computacional e possibilidade de armazenamento não são os únicos benefícios de um *Smart Card*. Fácil utilização, portabilidade e segurança são fatores que se encaixam ao uso de *Smart Cards*. Com isso, eles podem ser usados para guardar informações de um indivíduo, controlar acesso a determinados locais ou até mesmo armazenar dados financeiras. Em aplicações criptográficas, os *Smart*

Cards podem ser utilizados para memorizar uma chave privada e um certificado digital, dois componentes que verificam a autenticidade de um indivíduo. Neste trabalho, será utilizado para gerar e armazenar uma chave secreta.

Apesar das muitas vantagens que um *Smart Card* proporciona, ele possui uma grande desvantagem: o desenvolvimento de aplicações. Cada aplicação precisa ser compilada especificamente para cada *Smart Card* diferente, ou seja, necessita-se de um conhecimento do *hardware* em questão.

2.2.1 Java Card

Os *Java Cards* foram introduzidos em 1996 pela empresa Schlumberger (que mais tarde se tornou a Gemalto) com o intuito de solucionar o maior problema dos *Smart Cards*: desenvolvimento das aplicações. Provendo maior flexibilidade entre as aplicações, um mesmo *Java Card* pode executar diversos *applets* diferentes. Sendo os principais benefícios de um *Java Card*:

- Uso - os desenvolvedores de aplicações podem focar somente nas aplicações, deixando as camadas relacionadas ao hardware, a parte de mais abaixo nível, para o próprio *Java Card* tratar.

- Segurança - além da funcionalidade de acesso restrito ao uso do PIN (Personal Identification Number) , encontrada na maioria dos *Smart Cards*, os *Java Cards* possuem um *firewall* que impede que um objeto qualquer "A"de um determinado *applet* possa acessar um outro objeto "B", de outro *applet* armazenado no mesmo cartão.

- Independência de *hardware* - os *applets* são desenvolvidos na camada de mais alto nível da plataforma *Java Card*, sendo assim, não há necessidade de recompilá-los para cada Java Card diferente. Existe apenas uma dependência em relação à versão da API do Java Card.

- Múltiplos *applets* - é possível armazenar diversas instâncias de *applets* diferentes em um mesmo cartão. Cada *applet* possui acesso apenas a sua memória.

- Compatibilidade - Java Cards são baseados no padrão ISO 7816.

A figura 3 mostra como é a estrutura interna de um *Java Card*. É possível notar que existem três *applets* instalados neste cartão. Além disso, possui uma Java Card RE (Runtime Enviroment) que contém: uma Java Card VM (Virtual Machine) que define um subconjunto da linguagem Java e uma máquina virtual para as aplicações. O cartão também conta com uma Java Card API, que abstrai a complexidade do *hardware*, facilitando o desenvolvimento de *applets*. Além destes componentes, o cartão possui seu próprio sistema operacional.

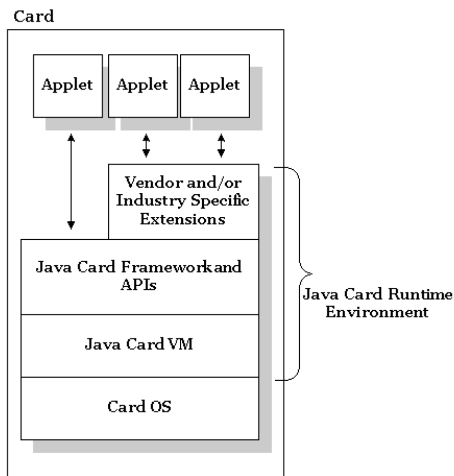

Figura 3 – Estrutura de um Java Card. Fonte: (GRINGERI, 2008-2009)

2.3 JAVA CARD APPLETS

Para executar programas em um *Java Card*, é necessário desenvolver um *Java Card Applet* e carregá-lo no cartão. Cada *applet* possui um identificador único AID (Application Identifier), que é separado em duas partes: RID (Resource Identifier) e PIX (Proprietary Identifier Extension). O RID possui 5 bytes e identifica o provedor do *applet* no cartão, enquanto o PIX possui de 0 à 11 bytes e identifica a aplicação propriamente dita.

Figura 4 Application Identifier. Fonte: (GRINGERI, 2008-2009)

Um programa *Java Card* não é muito diferente de um programa Java comum. Toda classe deve estender *javacard.framework.Applet* e assim como em um programa Java *desktop*, possui um conjunto de classes. O JCRE suporta múltiplas aplicações, ou seja, é possível executar várias instâncias do mesmo *applet* ou diferentes *applets*. Após o fim do desenvolvimento da aplicação, as classes são convertidas em um arquivo CAP (Converted Applet) que representa o *applet* propriamente dito. Após a conversão, é necessário carregar o CAP no cartão e instanciá-lo para deixá-lo em um estado selecionável e executável. O processo para carregá-lo consiste em transformar o arquivo CAP em um conjunto de comandos APDU's (Application Protocol Data Unit) e escrever o conteúdo destes APDU's na memória persistente do cartão (ROM), para, em seguida, instanciar e registar o *applet* ao JCRE. Feito isso, o *applet* pode ser selecionado e executado. Quando selecionado, a aplicação espera por comandos APDU's e quando os recebe, executa procedimentos e retorna resultados.

2.3.1 Protocolo de comunicação

Quando um *applet* está selecionado no cartão, ele aguarda comandos a serem executados. Estes comandos, obedecem um protocolo de comunicação: APDU. A comunicação entre o *host* e o *applet* segue o padrão mestre-escravo, sendo o *applet* sempre o escravo. Existem dois tipos de APDU definidos pela ISO7816-4: *command* APDU (C-APDU) executado pelo *host* e *response* APDU (R-APDU) executado pelo *applet*. A estrutura de um *command* APDU consiste em duas partes: *header* (obrigatória) e *body* (opcional).

Command APDU						
Header (required)				Body (optional)		
CLA	INS	P1	P2	Lc	Data Field	Le

Figura 5 – Estrutura de um command APDU. Fonte: (GRINGERI, 2008-2009)

Código	Descrição	Tamanho (bytes)
CLA	Identifica o tipo de instrução (proprietário ou fabricante)	1
INS	Identifica o o comando especificado	1
P1	Primeiro parâmetro, usado para prover uma caracterização adicional à instrução	1
P2	Segundo parâmetro, usado para prover uma caracterização adicional à instrução	1
Lc	Identifica o tamanho do dado que será transmitido	0 ou 1
Data	Dado que será transmitido ao applet	variável
Le	Identifica o tamanho do dado de resposta	0 ou 1

Tabela 1 – Tabela de descrição dos códigos de um command APDU

Todo *command* APDU é seguido de um *response* APDU. A estrutura de um *response* APDU consiste também em duas partes: *body* (opcional) e *trailer* (obrigatória).

Response APDU		
Body (optional)	Trailer (required)	
Data Field	SW1	SW2

Figura 6 – Estrutura de um response APDU. Fonte: (GRINGERI, 2008-2009)

As *Status words* (SW) fornecem informação sobre a execução do *command* APDU. No corpo opcional, o *data field*, contém os dados de resposta para o *host*.

Os APDU's podem ser categorizados em diversos casos diferentes, baseados na possibilidade do *command* APDU conter o campo de

dados ou o *response* APDU conter os dados.

- no primero caso, ambos, o *command* APDU e *response* APDU contém apenas os campos obrigatórios.

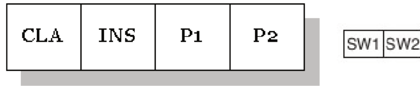

Figura 7 *Command* APDU e *Response* APDU caso 1. Adaptado de: (GRINGERI, 2008-2009)

- no segundo caso, nenhum dado é transferido ao cartão, porém o *applet* envia dados ao *host*.

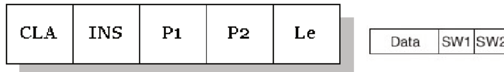

Figura 8 *Command* APDU e *Response* APDU caso 2. Adaptado de: (GRINGERI, 2008-2009)

- no terceiro caso, o dado é transferido para o cartão, porém o *applet* não retorna nenhum dado.

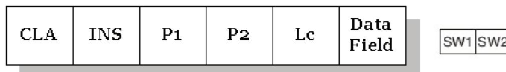

Figura 9 *Command* APDU e *Response* APDU caso 3. Adaptado de: (GRINGERI, 2008-2009)

- no quarto caso, dados são transferidos nas duas direções, *host* e *applet*.

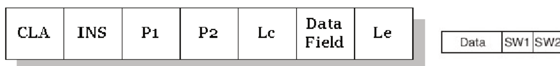

Figura 10 *Command* APDU e *Response* APDU caso 4. Adaptado de: (GRINGERI, 2008-2009)

2.3.2 Ciclo de vida de um applet

Todo *applet* possui um ciclo de vida, resumido em: instalar, selecionar, executar e deselecionar. Cada um destes estados deve ser implementado obrigatoriamente pelas aplicações.

Método de instalação

Basicamente, o método de instalação (*install*) utiliza o operador *new* para instanciar um objeto da classe principal, chamando seu construtor. É altamente recomendado que toda variável e/ou objeto que é utilizado pelo *applet* seja instanciado e inicializado em seu construtor. Ou seja, um construtor de um *Java Card Applet* deve conter:

- Criação dos objetos utilizados pelo *applet* em seu tempo de vida.
- Inicialização de objetos e variáveis.
- Registro do *applet* ao JCRE.

A aplicação aguarda comandos a serem executados, o ciclo de vida do *applet* é iniciado, consequentemente, é altamente recomendável que último comando de um construtor seja o método de registro. O método de registro possui duas funções: armazenar uma referência do *applet* ao JCRE e atribuir um AID ao mesmo. Caso aconteça alguma falha durante o processo, o JCRE realiza uma limpeza dos dados previamente alocados na memória.

Método de seleção

A seleção (*select*) ocorre quando o *SELECT APDU* — o único comando padronizado *Java Card* - é acionado. Até que isto ocorra, o *applet* fica em um estado de suspensão. A consequência deste comando é a invocação do método de seleção no *applet*. Se alguma falha ocorrer, o JCRE retorna uma resposta em APDU padrão: 0x6999.

Método de execução

Após a seleção do *applet* e até sua deseleção, todo comando APDU é examinado no método de processamento (*process*). Toda aplicação Java Card deve conter este método. O método examina o APDU recebido e executa a função correspondente.

Método de deseleção

O método de deseleção (*deselect*) realiza uma limpeza na memória e permite que outro *applet* possa ser selecionado. Toda aplicação deve sobrescrever este método, que fornece as operações para limpeza da memória. Caso ocorra alguma falha, ou, alguma exceção seja gerada, o *applet* sempre será deselecionado, ignorando todas as exceções. Desinstalar o *applet* ou resetar o cartão, causa deseleção do *applet* sem a necessidade de chamar o método.

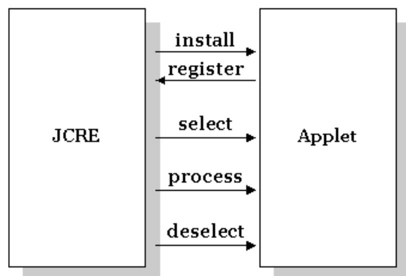

Figura 11 – Ciclo de vida de um applet. Fonte: (GRINGERI, 2008-2009)

2.4 GLOBALPLATFORM

As definições de Java Cards não preveem como serão instaladas e mantidas as aplicações no cartão. A GlobalPlatform específica padrões para a administração da infraestrutura dos *Smart Cards*. Essa administração consiste na instalação e remoção de *applets*. Normalmente, um Java Card possui um *applet* para gerenciar essa administração, este *applet* é conhecido como *Card Manager*.

O padrão específica que toda aplicação será identificada pelo seu AID e serão alternadas entre si utilizando o comando SELECT definido pelo padrão Java Card. Como exemplo de programas que implementam a especificação GlobalPlatform pode-se citar: GPShell (desenvolvido em C) e GlobalPlatformPro (desenvolvido em Java).

2.5 ISO 7816

A ISO 7816 é um conjunto de padrões relacionados à cartões de identificação digital, os *Smart Cards*. O padrão foi estabelecido pela *International Organization for Standardization* (ISO), uma organização internacional não governamental com membros em mais de 160 países (STANDARDIZATION, 2016).

O padrão foi dividido em quatorze diferentes tópicos, relacionados às distintas características de um cartão, desde princípios físicos até formatos para as informações criptográficas.

Este trabalho discutirá com mais detalhes o tópico 7816-4, que normatiza os comandos de comunicação com os *Smart Cards*. Tanto a 7816-4 quanto a 7816-3, que define os protocolos de transmissão de dados, estão encapsulados na interface ISO7816 do Java Card *Framework*, encontrada nas diferentes versões de *Java Cards*.

Tópico	**Escopo**
7816-1	Physical characteristics
7816-2	Dimensions and location of the contacts
7816-3	Electrical interface and transmission protocols
7816-4	Organization, security and commands for interchange
7816-5	Registration of application providers
7816-6	Interindustry data elements for interchange
7816-7	Interindustry commands for Structured Card Query Language
7816-8	Commands for security operations
7816-9	Commands for card management
7816-10	Electronic signals and answer to reset for synchronous cards
7816-11	Personal verification through biometric methods
7816-12	USB electrical interface and operating procedures
7816-13	Application management in multi-application environments
7816-15	Cryptographic information application

Tabela 2 – Tópicos da ISO 7816.

2.5.1 ISO 7816-4

Estão definidos no escopo da 7816-4: *Interindustry Commands for Interchange*:

- O conteúdo das mensagens, os comandos e as respostas transmi-

tidas do dispositivo de interface para o cartão e vice-versa.

- A estrutura e o conteúdo dos bytes enviados pelo cartão durante a *answer to reset.*
- A estrutura dos arquivos e dados, para o processamento de comandos de troca de informação.
- Métodos de acesso à arquivos e dados no cartão.
- Métodos para troca de mensagens seguras.
- Métodos de acesso a algoritmos processados pelo cartão.

Para a adição de novas funções em um *Java Card applet* dois aspectos importantes devem ser destacados: sistema de arquivos e padrões de APDU.

Sistema de arquivos

São suportados dois tipos de arquivo: Dedicated File (DF) e Elementary File (EF). Todo cartão possuí um arquivo obrigatório do tipo DF nomeado MF, ou Mandatory File, sendo os demais arquivos DF opcionais. Os arquivos DF são dedicados a dados interpretados pelo cartão, enquanto os Elementary Files são usados pelos dados utilizados exclusivamente pelo mundo exterior.

Os arquivos podem ser selecionados pelos seus identificadores, codificados em 2 bytes, ou pelo seu caminho (concatenação de identificadores), começando pelo identificador do arquivos mandatório (MF), seguido pelo DF corrente e finalmente o identificador do arquivo em si.

Padrões de APDU

A ISO define padrões para a codificação de cada um dos campos em um APDU, as combinações de byte muitas vezes são reservadas para casos recorrentes em implementações em *Smart Cards.* Para o INS, por exemplo, certos valores devem ser evitados e outros são reservados para os *Basic Interindustry Commands* e *Transmission Oriented Interindustry Commands.*

Value	Command name	Clause
'0E'	ERASE BINARY	6.4
'20'	VERIFY	6.12
'70'	MANAGE CHANNEL	6.16
'82'	EXTERNAL AUTHENTICATE	6.14
'84'	GET CHALLENGE	6.15
'88'	INTERNAL AUTHENTICATE	6.13
'A4'	SELECT FILE	6.11
'B0'	READ BINARY	6.1
'B2'	READ RECORD(S)	6.5
'C0'	GET RESPONSE	7.1
'C2'	ENVELOPE	7.2
'CA'	GET DATA	6.9
'D0'	WRITE BINARY	6.2
'D2'	WRITE RECORD	6.6
'D6'	UPDATE BINARY	6.3
'DA'	PUT DATA	6.10
'DC'	UPDATE DATA	6.8
'E2'	APPEND RECORD	6.7

Figura 12 Alguns padrões de INS. Fonte: (ISO/IEC, 2005)

Outro padrão importante definido pela norma é o formato de dados TLV (Tag-Lenght-Value). O TLV é uma estrutura simples: no primeiro campo um código binário define o tipo do dado processado, seguido pelo tamanho (usualmente em bytes) e uma série de bytes indicando o valor (DETTONI, 2015). Desta forma múltiplos pedaços de informação podem ser transmitidos em uma mesma mensagem, adicionando-se triplas encadeadas.

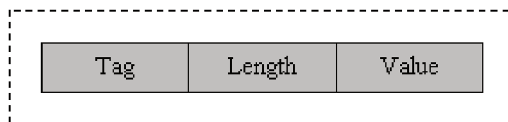

Figura 13 Estrutura do formato TLV. Fonte: Wikipédia - A enciclopédia livre.

2.6 PKCS

O padrão de criptografia de chaves públicas foi desenvolvido em uma parceria entre a *RSA Laboratories* e desenvolvedores na área de

segurança no mundo todo. Originalmente envolvia empresas, universidades e organizações, como: *Apple*, *Microsoft*, DEC, Lotus, Sun e MIT (Massachusetts Institute of Technology) .

Publicado em 1991, a norma vem sendo vastamente implementada e referenciada. As modificações na proposta original da década de 90 ocorrem, principalmente, em listas de discussões e *workshops* (oficinas de trabalho). Muitos algoritmos são suportados, como o RSA e a troca de chaves Diffie-Hellman. Semelhante à outros padrões, como a ISO 7816, o *PKCS* é dividido em diferentes tópicos:

Tópico	**Escopo**
#1	RSA Cryptography Standard
#2	Withdrawn
#3	Diffie–Hellman Key Agreement Standard
#4	Withdrawn
#5	Password-based Encryption Standard
#6	Extended-Certificate Syntax Standard
#7	Cryptographic Message Syntax Standard
#8	Private-Key Information Syntax Standard
#9	Selected Attribute Types
#10	Certification Request Standard
#11	Cryptographic Token Interface
#12	Personal Information Exchange Syntax Standard
#13	Elliptic Curve Cryptography Standard
#14	Pseudo-random Number Generation
#15	Cryptographic Token Information Format Standard

Tabela 3 – Tópicos do PKCS.

É importante frisar que alguns dos tópicos foram condensados em um, como os de número 1, 2 e 4 e outros foram abandonados ou estagnados como o referente à números pseudoaleatórios (PKCS#14). Para este trabalho são importantes aqueles padrões relacionados à *tokens* criptográficos, ou seja: PKCS#15, que permite que usuários destes *tokens* se identifiquem perante aplicações cientes do padrão e PKCS#11, que define uma API (Application Programming Interface), conhecida como *Cryptoki* (Cryptographic Token Interface), para dispositivos que armazenam informações criptográficas e/ou executam funções de criptografia.

2.6.1 PKCS#11

A *Cryptoki* possibilita a interoperabilidade, entre as aplicações e os *tokens*, que podem trocar informações independente da tecnologia e do tipo de dado compartilhado. Sendo assim, diferentes aplicações podem compartilhar um mesmo dispositivo e, uma aplicação em particular pode comunicar-se com vários dispositivos ao mesmo tempo.

Para o padrão, toda a leitora de dispositivos é denominada *slot*. E cada *slot*, pode conter um *token*. Os objetos armazenados nos *tokens* são divididos em: dados, certificados e chaves. Onde as chaves podem ser públicas, privadas ou secretas. Os dados são manipulados em sessões de leitura ou leitura e escrita.

O PKCS#11 também define dois tipos de usuários: Security Officer (SO) e o usuário comum. Apenas o usuário comum tem acesso ao seus dados privados, que são acessados e manipulados após um passo de autenticação, o requerimento de um PIN.

Em relação à segurança, além do acesso regulamentado pelo PIN, os dados podem ser registrados com *flags* especiais como: *sensitive*, que impede que uma chave seja exposta em um arquivo fora do *token* e *unextractable*, que impede a extração do objeto.

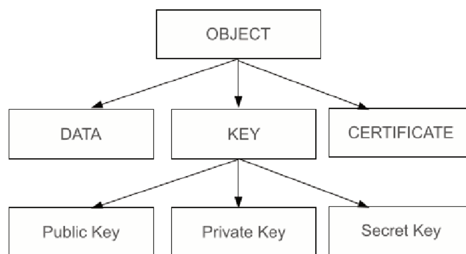

Figura 14 Hierarquia de dados PKCS. Fonte: (RSA, 2009)

As funções normatizadas pelo PKCS são definidas na linguagem de programação ANSI C e normalmente disponibilizadas via arquivos *header*, pelo provedor da biblioteca. Em especial são definidas aquelas para a geração e uso de chaves secretas, parte do tema deste trabalho.

Propósito Geral	*C_ Initialize*, *C_ Finalize*, *C_ GetFunctionList*, *C_ GetInfo*.
Gerenciamento de Slots e Tokens	*C_ GetSlotList*, *C_ GetSlotInfo*, *C_ GetTokenInfo*, *C_ InitToken*, *C_ InitPIN*, *C_ SetPIN*.
Gerenciamento de Sessão	*C_ OpenSession*, *C_ CloseSession*, *C_ Login*, *C_ CloseAllSessions*, *C_ GetSessionInfo*, *C_ Logout*.
Cifragem	*C_ EncryptIni*, *C_ Encrypt*, *C_ EncryptUpdate*, *C_ EncryptFinal*.
Decifragem	*C_ DecryptInit*, *C_ Decrypt*, *C_ DecryptUpdate*, *C_ DecryptFinal*.

Tabela 4 – Exemplo de funções definidas no PKCS#11.

2.6.2 PKCS#15

O padrão PKCS#15 possibilita que usuários de *tokens* criptográficos identifiquem-se perante aplicações cientes do padrão, sendo elas a *cryptoki* ou qualquer outra provedora de interface com *tokens*. Estão cobertos pela norma tanto dispositivos de *hardware*, por exemplo, cartões IC (Integrated Circuit) quanto *softwares*. Em 2004 foi desenvolvido um padrão para os *Smart Cards*, baseado no PKCS15, nomeado ISO/IEC 7816-15.

Principais Características do PKCS#15 (KNOSPE, 2006):

- Pode ser implementado em qualquer cartão que seja compatível com as normas básicas da ISO 7816.
- Possui mecanismos que possibilitam a integridade de dados, favorecendo os cartões que não suportam funções de criptografia.

- Para aplicações que não trabalham com a noção de arquivos, possibilita que as informações sejam armazenadas em um bloco contínuo na memória.

- Possui arquivos específicos para que as aplicações externas possam identificar os algoritmos, chaves e certificados presentes no cartão.

Sistema de Arquivos

A estrutura de arquivos adotada pelo padrão é relativamente dependente do tipo de cartão usado e dos dados armazenados. Porém, em geral, pode-se assumir uma estrutura comum, onde: o *Object Directory File* (ODF) , um arquivo obrigatório, serve como ponteiro para os demais arquivos (PrKDFs, PuKDFs, SKDFs, CDFs, DODFs e AODFs). Estes arquivos podem estar relacionados à chaves privadas (PrKDFs) , públicas (PuKDFs) e secretas (SKDFs) , contendo informações como: *labels*, restrições de uso, tipo de algoritmo, tamanho da chave e um ponteiro para a chave propriamente dita.

2.7 PC/SC

O PC/SC é uma especificação, mantida pela organização *PC/SC Workgroup*, que padroniza a integração dos *Smart Cards* com as leitoras (PC/SC, 2017). Os principais focos desta organização são:

- prover um padrão que permita que *Smart Cards*, leitoras e computadores integrem-se, independente dos fabricantes.

- facilitar o desenvolvimento de aplicações *Smart Cards* para computadores.

Em relação ao uso pelos sistemas operacionais. O *Windows* implementa sua própria API que segue o padrão PC/SC, chamada *winscard*, enquanto *Linux* e *Mac* utilizam o pacote *open source* pcsc-lite.

2.8 OPENSC

O OpenSC é um *middleware* localizado entre as aplicações e os *Smart Cards*. Ele prove um conjunto de bibliotecas e utilitários para trabalhar com *tokens* criptográficos, principalmente funções relacionadas a autenticação e assinatura digital. O foco do projeto pode ser a

motivação pela qual não houve um desejo por parte dos desenvolvedores em incluírem a criptografia simétrica no mesmo.

Uma das principais características deste conjunto de bibliotecas é a implementação dos padrões PKCS#11 e PKCS#15, que possibilitam a comunicar com diferentes *softwares*, como o navegador *web* Mozilla Firefox e a aplicação de *e-mails* Thunderbird.

O projeto é desenvolvido por uma equipe voluntária internacional e mantido sob a licença *open source* LGPL. Conta, atualmente, com 16 versões, sendo a mais recente de Junho de 2016.

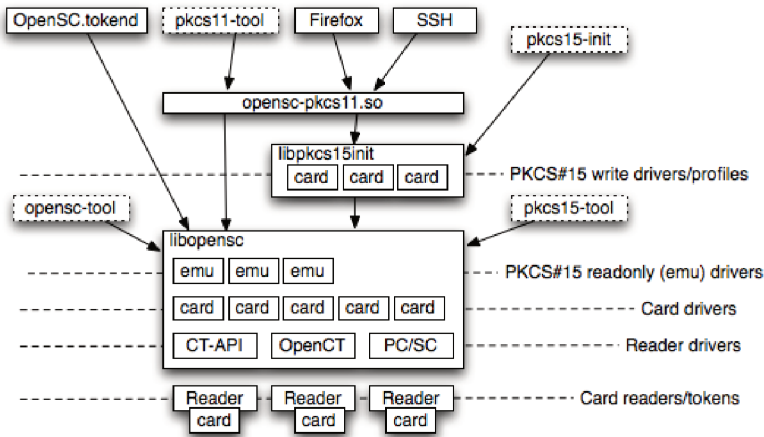

Figura 15 Arquitetura do OpenSC. Fonte: (OPENSC, 2017)

A figura 15 mostra a arquitetura do OpenSC. As aplicações de usuário podem se comunicar com o mediador por, pelo menos, dois caminhos distintos: através do módulo PKCS#11 (*opensc-pkcs11.so*), que implementa parte significativa das funções definidas pelo padrão, ou diretamente com a biblioteca OpenSC (*libopensc*). A comunicação com os *Smart Cards* se dá, principalmente, com o uso do PC/SC.

Qualquer *applet* em conformidade com o OpenSC possuí um drive desenvolvido e acoplado ao projeto, desta forma uma função requisitada por um utilitário qualquer pode ser traduzida, em forma de APDU's e transmitida ao cartão. Dentre os *applets* suportados pode-se citar: Muscle Apple e Iso Applet.

3 DESENVOLVIMENTO

3.1 ESCOLHA DO APPLET

Afim de implementar as funções de criptografia simétrica em um *applet* de código aberto, inicialmente se mostrou necessário a pesquisa e escolha de uma aplicação que melhor se adaptasse aos parâmetros estabelecidos. Julgou-se relevante avaliar: frequência de atualizações, padronizações, compatibilidade com o projeto OpenSC e implementação de funções criptográficas. A busca se deu no serviço de repositório GitHub, onde foram encontrados: Iso Applet, Muscle Applet e CoolKeyApplet.

	IsoApplet	**MuscleApplet**	**CoolKeyApplet**
Atualizações	Atualizado	Obsoleto	Atualizado
Padronização	ISO7816	Despadronizado	Despadronizado
Compatibilidade	OpenSC 0.16	OpenSC 0.13	OpenSC 0.16
Criptografia assimétrica	RSA e EC	RSA	RSA
Criptografia simétrica	Não implementa	Não implementa	Não implementa

Tabela 5 – Comparação entre os *applets*.

Como pode ser visto na tabela 5, nenhum *applet* implementa as funções de criptografia simétrica normatizadas pelo padrão PKCS#11. Além disso, o Iso Applet é o único que segue, com maior rigor, o padrão ISO 7816 e o que suporta o maior número de algoritmos criptográficos. Outro fator que vale a pena destacar é o fato das aplicações acima poderem ser utilizadas para autenticação de usuários. Por esses motivos optou-se pelo Iso Applet, como a aplicação a ser estendida.

3.2 ISO APPLET

Originalmente o *applet* é capaz de armazenar estruturas PKCS#15, bem como realizar operações com chaves públicas e privadas, como: assinaturas e decifragens. O par de chaves (RSA ou EC) pode ser gerado internamente, ou carregado no cartão através de uma função de *upload.* O Iso segue os padrões de codificação, uso dos APDU's e sistema de arquivos estabelecidos pela ISO 7816 e pode ser instalado nas versões mais modernas de *Java Cards*, 2.2.2 ou maiores.

O projeto se encontra hospedado no serviço de repositório GitHub

e conta com uma documentação detalhada sobre suas motivações, padrões e algoritmos suportados, além de explicações do uso do *applet* em conjunto à grandes bibliotecas *open source*, como: OpenSC e OpenSSL. Suas atualizações são frequentes e possui mais de uma dezena de *forks* de outros desenvolvedores da plataforma.

O software em Java é divido em 14 classes distintas, das quais: 8 implementam funções relacionadas ao sistema de arquivos, outras 2 implementam a troca de dados entre o *applet* e o mediador, uma delas é responsável pela centralização das funções, variáveis, atributos e padrões da aplicação e as demais tratam as possíveis exceções que podem ocorrer durante uma execução.

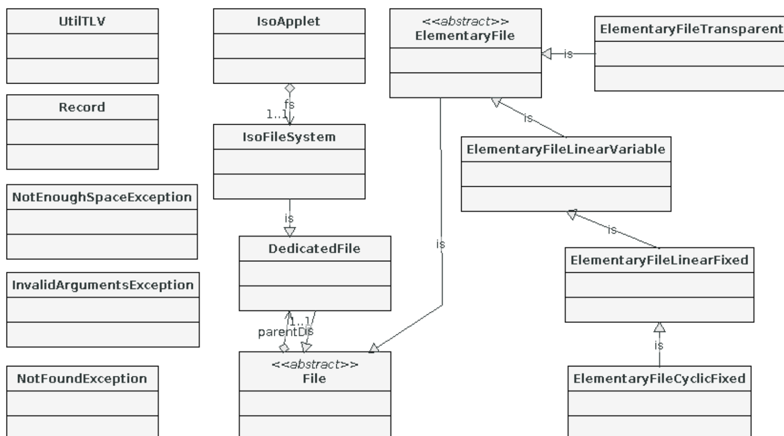

Figura 16 – Diagrama de classes UML simplificado do Iso Applet. Fonte: Criada pelos autores.

Baterias de testes foram realizadas afim de verificar o potencial da aplicação em cenários reais de segurança em computação. Verificou-se o uso do *applet* como meio de autenticação de usuários, explorando uma das mais poderosas utilidades da criptografia assimétrica. Os resultados foram positivos em um cenário onde uma aplicação utilitária, Ywapa — projeto desenvolvido pelo laboratório de segurança em computação (LABSEC, UFSC) para o governo brasileiro, com o objetivo de emitir certificados da Autoridade Certificadora (AC) raiz brasileira —, cria e autentica usuários através do par de chaves armazenado no cartão, com o apoio do *middleware* OpenSC.

Uma vez que o *applet* é instalado, o passo seguinte necessário é a geração de uma estrutura de arquivos PKCS#15 e um PIN associado,

que permitirão o armazenamento de dados, certificados ou chaves no *Java Card*. Para tanto, utiliza-se o utilitário *pkcs15-init* do mediador OpenSC.

```
:/$ pkcs15-init -C

New User PIN.
Please enter User PIN:
Please type again to verify:
Unblock Code for New User PIN (Optional - press return for no PIN).
Please enter User unblocking PIN (PUK):
```

Figura 17 Criação da estrura PKCS#15. Fonte: Criada pelos autores.

O resultado da criação deste e outros arquivos PKCS#15 podem ser verificados com a evocação de um segundo utilitário do OpenSC: *pkcs15-tool*. A opção *-D* faz acesso ao arquivo descritor de dados e é capaz de informar as chaves, certificados e outros dados armazenados no cartão.

```
:/$ pkcs15-tool -D

PKCS#15 Card [JavaCard isoApplet]:
        Version        : 0
        Serial number  : 0000
        Manufacturer ID: unknown
        Last update    : 20170516185456Z
        Flags          : EID compliant
PIN [User PIN]
        Object Flags   : [0x3], private, modifiable
        ID             : ff
        Flags          : [0x39], case-sensitive, unblock-disabled, initialized, needs-padding
        Length         : min_len:4, max_len:16, stored_len:16
        Pad char       : 0x00
        Reference      : 1 (0x01)
        Type           : ascii-numeric
```

Figura 18 Saída do comando *-D* para a estrutura PKCS#15 e PIN. Fonte: Criada pelos autores.

Conforme estabelecido no padrão de sistema de arquivos da norma PKCS, todas as chaves geradas ou carregadas no cartão adotam um dos formatos possíveis: PrKDFs, PuKDFs, SKDFs. Além de armazenarem informação relacionadas à identificação do objeto, tipo de algoritmo utilizado, tamanho (bits), seus possíveis usos e restrições de acesso. No Iso Applet, a geração de um par de chaves RSA de tamanho 2048 bits (o único tamanho permitido pela aplicação) se dá pela execução de variações do comando: *pkcs15-init generate-key rsa:2048 key-usage sign,decrypt id 01 auth-id FF label Key*.

Importante frisar que testes semelhantes foram executados utilizado Muscle Applet, mas para chaves RSA 1024 bits e em uma versão

menos atualizado do mediador (OpenSC 0.13). Os resultados também foram positivos para autenticação de usuários, mas devido à sua obsolescência e as recomendações de não uso do próprio desenvolvedor, o *applet* não se mostrou uma boa opção para este trabalho.

```
:~$ pkcs15-tool -D

PKCS#15 Card [JavaCard isoApplet]:
        Version        : 0
        Serial number  : 0000
        Manufacturer ID: unknown
        Last update    : 20170516210042Z
        Flags          : EID compliant
PIN [User PIN]
        Object Flags   : [0x3], private, modifiable
        ID             : ff
        Flags          : [0x39], case-sensitive, unblock-disabled, initialized, needs-padding
        Length         : min_len:4, max_len:16, stored_len:16
        Pad char       : 0x00
        Reference      : 1 (0x01)
        Type           : ascii-numeric

Private RSA Key [Key]
        Object Flags   : [0x3], private, modifiable
        Usage          : [0x2E], decrypt, sign, signRecover, unwrap
        Access Flags   : [0x1D], sensitive, alwaysSensitive, neverExtract, local
        ModLength      : 2048
        Key ref        : 0 (0x0)
        Native         : yes
        Path           : 3f005015
        Auth ID        : ff
        ID             : 01
        MD:guid        : 9508e905-48b0-440a-4a61-e5743b76c1e3

Public RSA Key [Key]
        Object Flags   : [0x2], modifiable
        Usage          : [0xD1], encrypt, wrap, verify, verifyRecover
        Access Flags   : [0x0]
        ModLength      : 2048
        Key ref        : 0 (0x0)
        Native         : no
        Path           : 3f0050153300
        ID             : 01
```

Figura 19 Saída do comando *-D* para o par de chaves RSA. Fonte: Criada pelos autores.

3.2.1 Adições no Iso Applet

A adição de novas funcionalidades no Iso Applet segue uma estrutura bastante uniforme e reproduzida como um padrão na maioria das aplicações de mesma finalidade. Na classe principal, aquela que representa uma instância da aplicação propriamente dita, encontram-se:

1. Inclusão de bibliotecas:

   Assim como em muitas aplicações desenvolvidas na linguagem de programação Java, para os *applets* também pode ser necessário a adição de uma série de bibliotecas externas, que ofereçam uma gama de funcionalidades pré-desenvolvidas. Para um *applet* que

implementa funções criptográficas com chaves assimétricas, seguindo o padrão ISO 7816, como é caso do Iso, são relevantes importações como as que seguem:

```
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.framework.APDU;
import javacard.framework.JCSystem;
import javacard.framework.Util;
import javacard.framework.OwnerPIN;
import javacard.security.KeyBuilder;
import javacard.security.KeyPair;
import javacard.security.Key;
import javacard.security.RSAPublicKey;
import javacard.security.RSAPrivateCrtKey;
import javacard.security.ECKey;
import javacard.security.ECPublicKey;
import javacard.security.ECPrivateKey;
import javacardx.crypto.Cipher;
import javacardx.apdu.ExtendedLength;
import javacard.security.CryptoException;
import javacard.security.Signature;
import javacard.security.RandomData;
```

Além destas, encontradas na versão original da aplicação. Afim de abranger a criptográfica simétrica, mais especificamente o algoritmo AES, é necessário a adição de pelo menos mais uma biblioteca:

```
import javacard.security.AESKey;
```

2. Declaração dos atributos da classe:

   Existem uma série de atributos importantes declarados neste trecho de código, porém os mais relevantes à este trabalho são os relacionados aos campos do APDU, que identificam as funções oferecidas pelo *software* (campos CLA e INS), além dos atributos da classe que armazenam os objetos do tipo chave. Para a adição das funções de geração, importação, exportação, cifragem e decifragem de dados com chaves secretas, foram adicionados 14 novos atributos:

```
// INS
public static final byte INS_GEN_SKEY = (byte) 0x30;
public static final byte INS_GET_SKEY = (byte) 0x32;
public static final byte INS_CIPH_SKEY = (byte) 0x34;
```

```
5  public static final byte INS_DECIPH_SKEY = (byte) 0x36;
6  public static final byte INS_LOAD_SKEY = (byte) 0x38;
7
8  // Key Size
9  byte ALG_GEN_AES_128 = (byte) 0xA1;
10 byte ALG_GEN_AES_256 = (byte) 0xA2;
11
12 private static AESKey sKey = null;
13 private Key[] sKeys = null;
14 private short [] currentSKeyRef = null;
15
16 private static byte[] sKey_array = null;
17 private static byte[] encrypted = null;
18 private static byte[] holder = null;
19 private static short sKeySize = 16; // default size
```

Os valores para o campo INS foram escolhidos seguindo as recomendações da norma ISO 7816. Evitando valores impares, INSs reservados e variações dos hexadecimais *0x9X* e *0x6X*, que são restritos ao *response* APDU.

3. Métodos de instalação, seleção, deseleção e construção da classe:

   Funções necessária no desenvolvimento de qualquer *applet*, responsáveis principalmente pela instalação da aplicação nos *Java Cards* e pela inicialização dos atributos globais. No contexto deste trabalho, mostrou-se necessária a inicialização de dois novos atributos no construtor da classe: *currentSKeyRef*, que identifica a chave em uso e *sKeys*, um arranjo que armazenas as chaves secretas geradas ou carregadas no cartão.

```
1 protected IsoApplet() {
2     // ...
3     currentSKeyRef = JCSystem.makeTransientShortArray((short)1,
          JCSystem.CLEAR_ON_DESELECT);
4     sKeys = new Key[KEY_MAX_COUNT];
5     // ...
6 }
```

4. Método de processamento:

   O método *process* filtra as requisições feitas ao *Java Card* e executa as funções necessárias, através do valor do campo INS contido nos APDU's. Cinco novos casos foram adicionados ao método:

```
1 @Override
```

```
2       public void process(APDU apdu) {
3           // ...
4           switch (ins) {
5               // ...
6               case INS_GEN_SKEY:
7                   processGenerateSKey(apdu);
8                   break;
9               case INS_GET_SKEY:
10                  processGetSKey(apdu);
11                  break;
12              case INS_CIPH_SKEY:
13                  processCiphSKey(apdu);
14                  break;
15              case INS_DECIPH_SKEY:
16                  processDeciphSKey(apdu);
17                  break;
18              case INS_LOAD_SKEY:
19                  processLoadSKey(apdu);
20                  break;
21              // ...
22          }
23          // ...
24  }
```

Para cada um dos casos existe uma função associada e todas elas recebem como parâmetro o valor do APDU, para que possam extrair dele as informações necessárias ao processamento.

5. Métodos de propósito específicos:

   Neste bloco são implementadas as funções oferecidas pela aplicação. No Iso Applet pode-se encontrar algumas das principais funções estabelecidas pelo item 4 do padrão ISO 7816: *processReadBinary*, *processVerify*, *processPutData*, *processManageSecurityEnviroment*, entre outras.

   Vale destacar a função *processManageSecurityEnviroment*, a qual realiza uma pré-configuração das variáveis, algoritmos e referências, isto é: define a posição no *array* de chaves para o uso ou criação no processamento seguinte, bem como seus parâmetros e algoritmo. Algumas modificações foram necessárias neste método, afim de adaptá-lo para as funções de criptografia simétrica. Para a escolha do parâmetro P1 da APDU, adicionou-se um novo caso:

```
1   switch(p1) {
2       case (byte) 0x42:
3           try {
```

```
4              pos = UtilTLV.findTag(buf, offset_cdata, (byte) lc, (byte)
                   0x80);
5          } catch (NotFoundException e) {
6              ISOException.throwIt(ISO7816.SW_DATA_INVALID);
7          } catch (InvalidArgumentsException e) {
8              ISOException.throwIt(ISO7816.SW_DATA_INVALID);
9          }
10
11         if(buf[++pos] != (byte) 0x01) {
12             ISOException.throwIt(ISO7816.SW_DATA_INVALID);
13         }
14
15         algRef = buf[++pos];
16
17         try {
18             pos = UtilTLV.findTag(buf, offset_cdata, (byte) lc, (byte)
                   0x84);
19         } catch (NotFoundException e) {
20             ISOException.throwIt(ISO7816.SW_DATA_INVALID);
21         } catch (InvalidArgumentsException e) {
22             ISOException.throwIt(ISO7816.SW_DATA_INVALID);
23         }
24
25         if(buf[++pos] != (byte) 0x01 || buf[++pos] >=
               KEY_MAX_COUNT) {
26             ISOException.throwIt(ISO7816.SW_DATA_INVALID);
27         }
28
29         sKeyRef = buf[pos];
30         break;
31 }
```

O trecho de código acima configura o algoritmo e a referencia da chave para a sua geração. Entretanto, para definir seus parâmetros e verificar os padrões, utiliza-se o quarto *byte* da APDU, P2, no qual também foram adicionados novos casos, semelhantes à:

```
1  switch(p2) {
2      case (byte) 0x01:
3              if(algRef != ALG_GEN_AES_128 && algRef !=
                   ALG_GEN_AES_256) {
4                  ISOException.throwIt(
5                  ISO7816.SW_FUNC_NOT_SUPPORTED);
6              }
7              switch(algRef){
8                case ALG_GEN_AES_128:
9                  sKey_size = 16;
10                 break;
11               case ALG_GEN_AES_256:
12                 sKey_size = 32;
```

```
13                break;
14              default:
15                ISOException.throwIt(
16                ISO7816.SW_FUNC_NOT_SUPPORTED);
17            }
18            break;
19    }
```

3.2.1.1 Funções implementadas

- *Generate Secret Key*:

  Primeiramente faz-se necessário a verificação do PIN, dada a sensibilidade da operação executada. Caso o usuário falhe em se autenticar, a execução é finalizada em uma mensagem de exceção.

```
1  if( ! pin.isValidated() ) {
2      ISOException.throwIt(
3          ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
4  }
```

  Como a solução permite que até quinze chaves sejam armazenadas, a posição no *array* de chaves é atualizada com o valor configurado em um passo anterior. Além disso, é realizada a geração do valor aleatório que compõe a chave secreta.

```
1  short sKeyRef = currentSKeyRef[0];
2  RandomData randomData = RandomData.getInstance(
3      RandomData.ALG_PSEUDO_RANDOM);
4  sKey_array = JCSystem.makeTransientByteArray(
5      sKey_size, JCSystem.CLEAR_ON_RESET);
6  randomData.generateData(
7      sKey_array, (short)0, (short)sKey_array.length);
```

  Se houver uma chave armazenada na posição selecionada, o algoritmo executa uma limpeza preliminar.

```
1  if(sKeys[sKeyRef] != null) {
2      sKeys[sKeyRef].clearKey();
3  }
```

  Um *switch* define o tamanho de chave a ser gerado, com base no valor armazenado na variável *currentAlgorithmRef*. São dois os

valores possíveis de chave: 128 bits ou 256 bits. Tenta-se executar o método *buildKey* oferecido pela classe *KeyBuilder*, afim de criar uma estrutura básica para a chave.

```
1  switch(currentAlgorithmRef[0]){
2      case ALG_GEN_AES_128:
3          try{
4              sKey = (AESKey)KeyBuilder.buildKey(
5                  KeyBuilder.TYPE_AES,
                       KeyBuilder.LENGTH_AES_128, false);
6          }catch(CryptoException e) {
7              if (e.getReason() ==
                       CryptoException.NO_SUCH_ALGORITHM) {
8                  ISOException.throwIt(
9                      ISO7816.SW_FUNC_NOT_SUPPORTED);
10             }
11             ISOException.throwIt(ISO7816.SW_UNKNOWN);
12         }
13         break;
14     case ALG_GEN_AES_256:
15         try{
16             sKey = (AESKey)KeyBuilder.buildKey(
17                 KeyBuilder.TYPE_AES,
                       KeyBuilder.LENGTH_AES_256, false);
18         }catch(CryptoException e) {
19             if (e.getReason() ==
                       CryptoException.NO_SUCH_ALGORITHM) {
20                 ISOException.throwIt(
21                     ISO7816.SW_FUNC_NOT_SUPPORTED);
22             }
23             ISOException.throwIt(ISO7816.SW_UNKNOWN);
24         }
25         break;
26     default:
27         ISOException.throwIt(
28             ISO7816.SW_CONDITIONS_NOT_SATISFIED);
29         break;
30 }
```

Finalmente o valor aleatório gerado anteriormente é adicionado à estrutura básica e a chave resultante é salva no arranjo de chaves secretas.

```
1  sKey.setKey(sKey_array, (short)0);
2  sKeys[sKeyRef] = sKey;
```

- *Get Secret Key*:

Em um contexto onde seja necessário a extração da chave secreta, para compartilhamento com terceiros ou para o uso em aplicações externas, a função *processGetSKey* é acionada. Para que a execução ocorra corretamente o usuário deve, primeiramente, estar corretamente autenticado através do PIN.

```
1  if( ! pin.isValidated() ) {
2      ISOException.throwIt(
3          ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
4  }
```

Busca-se a chave escolhida no arranjo de chaves, com base em um parâmetro configurado no método *processManageSecurityEnviroment*, para em sequência exportá-la para o mundo externo.

```
1  byte [] buffer = apdu.getBuffer();
2  short sKeyRef = currentSKeyRef[0];
3  AESKey key = (AESKey) sKeys[sKeyRef];
4  byte [] sendKey = new byte[key.getSize()];
5  key.getKey(sendKey, (short)0);
6
7  short le = apdu.setOutgoing();
8  short totalBytes = (short) sendKey.length;
9  Util.arrayCopyNonAtomic(sendKey, (short)0, buffer, (short)0, totalBytes);
10 apdu.setOutgoingLength(totalBytes);
11 apdu.sendBytes((short) 0, (short) ((sendKey.length)/(short)8));
```

- *Ciph*:

  Para os algoritmos de cifragem e decifragem de dados, optou-se pela separação das funções que tratam dos dados e as funções que realizam as operações criptográficas. Assim, primeiro faz-se uma avaliação do PIN, configuração dos parâmetros de saída e cópia da informação a ser cifrada para uma variável auxiliar:

```
1  public void processCiphSKey(APDU apdu) {
2          if( ! pin.isValidated() ) {
3              ISOException.throwIt(
4              ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
5          }
6          byte [] buffer = apdu.getBuffer();
7          short bytesReadCount = apdu.setIncomingAndReceive();
8          short echoOffset = (short)0;
9          short sKeyRef = currentSKeyRef[0];
10         AESKey key = (AESKey) sKeys[sKeyRef];
11         holder = new byte[16];
12         while ( bytesReadCount > 0 ) {
13           echoOffset = Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA,
```

```
14            holder, echoOffset, bytesReadCount);
15            bytesReadCount =
                  apdu.receiveBytes(ISO7816.OFFSET_CDATA);
16        }
17        encrypted = cipherS(holder, key);
18        // ...
```

Em seguida, realiza-se a chamada de uma segunda função, *cipherS*, que realiza a cifragem dos dados com o auxilio da classe *Cipher*, utilizando a política *ALG_AES_BLOCK_128_CBC_NOPAD*.

```
1   public byte[] cipherS(byte [] data, AESKey key){
2       Cipher cipher = Cipher.getInstance(
3           Cipher.ALG_AES_BLOCK_128_CBC_NOPAD, false);
4       try{
5           cipher.init(key, Cipher.MODE_ENCRYPT);
6       }catch (CryptoException e) {
7           if(e.getReason() == CryptoException.ILLEGAL_VALUE ||
                e.getReason() ==
                CryptoException.UNINITIALIZED_KEY){
8             ISOException.throwIt(
9               ISO7816.SW_CONDITIONS_NOT_SATISFIED);
10          }
11          ISOException.throwIt(ISO7816.SW_UNKNOWN);
12      }
13
14      byte [] result = new byte[data.length];
15      short encLength = cipher.doFinal(data, (short)0, (short)data.length,
              result, (short)0);
16      return result;
17  }
```

Finalmente, encaminha-se o resultado para a aplicação externa. O valor de saída possuí o mesmo tamanho da entrada, já que o bloco de 128 bits se mantém.

```
1       // ...
2       short le = apdu.setOutgoing();
3       short totalBytes = (short) encrypted.length;
4       Util.arrayCopyNonAtomic(encrypted, (short)0, buffer, (short)0,
              totalBytes);
5       apdu.setOutgoingLength(totalBytes);
6       apdu.sendBytes((short) 0, (short) encrypted.length);
7   }
```

- *Deciph*:

  O método de decifragem de dados segue uma estrutura bastante

similar à sua operação inversa, com uma diferença crucial no uso da classe Cipher, utilizando o modo *DECRYPT*. Em ambas as operações é possível selecionar a chave secreta a ser utilizada, dentre as disponíveis no *Java Card*.

- *Load*:

  Para o processo de importação de uma chave simétrica, inicialmente realiza-se uma checagem do PIN, devido a gravidade da operação, e configura-se os parâmetros da chave (algoritmo e referência).

```
1  public void processLoadSKey(APDU apdu){
2     if( ! pin.isValidated() ) {
3         ISOException.throwIt(
4         ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
5     }
6     byte[] buffer = apdu.getBuffer();
7     byte algRef = currentAlgorithmRef[0];
8     short sKeyRef = currentSKeyRef[0];
9     int size = 0;
10    AESKey key = null;
```

  Após esta configuração inicial, é criado uma instância de uma chave, baseando-se no tamanho que foi pré-definido.

```
1  switch(algRef){
2       case ALG_GEN_AES_128:
3         size = 16;
4         key = (AESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_AES,
                KeyBuilder.LENGTH_AES_128, false);
5         break;
6       case ALG_GEN_AES_256:
7         size = 32;
8         key = (AESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_AES,
                KeyBuilder.LENGTH_AES_256, false);
9         break;
10      default:
11        ISOException.throwIt(ISO7816.SW_FUNC_NOT_SUPPORTED);
12   }
```

  Finalmente, caso já exista alguma chave com a mesma referência, ela é deletada. Os *bytes* enviados para o cartão são armazenados na instância criada anteriormente.

```
1  if(sKeys[sKeyRef] != null) {
2       sKeys[sKeyRef].clearKey();
3  }
```

```
4  byte[] local_data = new byte[size];
5  Util.arrayCopy(buffer,ISO7816.OFFSET_CDATA, local_data, (short)0,
       (short) size);
6  key.setKey(local_data, (short)0);
7  sKeys[sKeyRef] = key;
```

O apêndice A1 demonstra como estas funções podem ser executadas sem o uso de uma camada de abstração na comunicação com o *Java Card*, ou seja, transmitindo diretamente as sequências de APDU necessárias ao *applet*. Todas as modificações no código original, podem ser verificadas no anexo A.1 Iso Applet.

3.3 SUPORTE VIA MIDDLEWARE OPENSC

Entre as classes que compõem a estrutura do OpenSC, *pkcs11-object* é a responsável por implementar as assinaturas definidas no padrão PKCS#11. Embora todas as funções estejam presentes na classe, parte delas não possuí implementação. Em especial verifica-se que, originalmente, o método de geração de chaves secretas é dito não suportado pela biblioteca, como pode ser verificado pelo trecho de código a seguir:

```
1  CK_RV C_GenerateKey(CK_SESSION_HANDLE hSession,
2              CK_MECHANISM_PTR pMechanism,
3              CK_ATTRIBUTE_PTR pTemplate,
4              CK_ULONG ulCount,
5              CK_OBJECT_HANDLE_PTR phKey)
6  {
7     return CKR_FUNCTION_NOT_SUPPORTED;
8  }
```

Porém, nota-se a preocupação do projeto em englobar, em um momento futuro, essa e outras funções PKCS#11, uma vez que, no código original, encontram-se as assinaturas corretamente alinhadas com a norma vigente.

Antes da descrição propriamente dita das alterações realizadas no *middleware*, é importante que se entenda o fluxo de processamento que foi criado, iniciado na requisição da função por meio de um utilitário até o envio dos APDU's para o cartão. Semelhante ao que acontece para outras funções encontradas na versão original do OpenSC, optou-se pelo seguinte caminho:

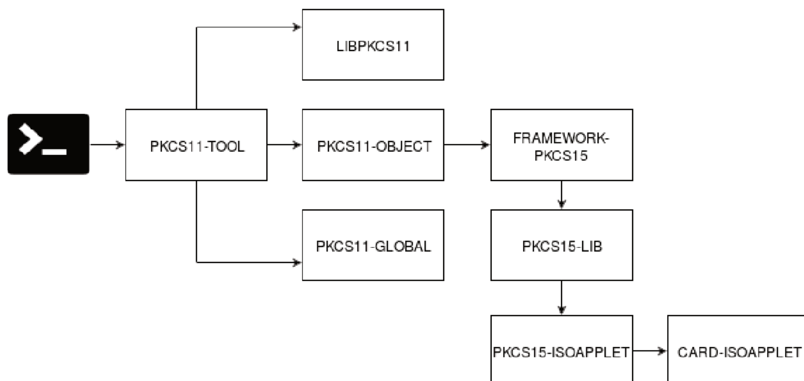

Figura 20 Fluxograma simplificado da geração de chaves simétricas. Fonte: Criada pelos autores.

Embora tenha-se optado pela implementação via utilitário *pkcs11-tool*, o OpenSC provê outra maneira de gerar chaves (assimétricas), que também poderia ter sido escolhido, através do *pkcs15-init*. Porém, como pode ser visto na figura 15, em ambas situações as requisições culminam em funções PKCS#15, que são encarregadas da criação das estruturas no cartão.

- *PKCS11-TOOLS*

  A primeira etapa no fluxograma. Funciona como uma interface entre o usuário e a biblioteca, responsável, neste caso, por intermediar os parâmetros selecionados para a geração de uma chave secreta. Antes de realizar qualquer operação, o método que responde à opção -k, salva em uma estrutura auxiliar as principais características do objeto: tipo de chave (AES, DES ou DES3), tamanho em bytes (16 ou 32 para AES), *flags* de acesso, rótulo e identificador. Finalmente a interface PKCS#11 (*pkcs11-object*) é acionada com a chamada da função *C_GenerateKey*.

```
1  static int
2  gen_key(CK_SLOT_ID slot, CK_SESSION_HANDLE session,
       CK_OBJECT_HANDLE *hSecretKey,
3     const char *type, char *label){
4      // ...
5      rv = p11->C_GenerateKey(session, &mechanism, keyTemplate,
          n_attr, hSecretKey);
6     if (rv != CKR_OK)
7        p11_fatal("C_GenerateKey", rv);
```

```
8   // ... }
```

- *PKCS11-OBJECT*

  Classe onde se iniciam as adições propostas, mais especificamente na função *C_GenerateKey*. Para tanto, buscou-se entender o funcionamento da geração de chaves assimétricas, como um padrão a ser seguido.

  Antes de mais nada, realiza-se uma verificação do *template* gerado no passo anterior. Inicia-se uma sessão de comunicação com o *Java Card*, seguida da verificação do suporte da função no *framework* PKCS#15 (modificações apresentadas mais a frente). O comando *lock* reserva o uso do *Java Card*, para que nenhuma outra aplicação possa interferir no processamento. Se todas as etapas anteriores forem um sucesso, invoca-se a função *gen_key* do *framework*.

```
1   CK_RV C_GenerateKey(CK_SESSION_HANDLE hSession,
2               CK_MECHANISM_PTR pMechanism,
3               CK_ATTRIBUTE_PTR pTemplate,
4               CK_ULONG ulCount,
5               CK_OBJECT_HANDLE_PTR phKey)
6   {
7       CK_RV rv;
8       struct sc_pkcs11_session *session;
9       struct sc_pkcs11_slot *slot;
10
11      if (pMechanism == NULL_PTR || (pTemplate == NULL_PTR
            && ulCount > 0))
12          return CKR_ARGUMENTS_BAD;
13
14      rv = sc_pkcs11_lock();
15      if (rv != CKR_OK)
16          return rv;
17
18        dump_template(SC_LOG_DEBUG_NORMAL,
              "C_GenerateKey(), Secret Key attrs", pTemplate, ulCount);
19
20      rv = get_session(hSession, &session);
21      if (rv != CKR_OK)
22          goto out;
23
24      if (!(session->flags & CKF_RW_SESSION)) {
25          rv = CKR_SESSION_READ_ONLY;
26          goto out;
27      }
28
29      slot  = session->slot;
```

```
30         if (slot->p11card->framework->gen_key == NULL)
31             rv = CKR_FUNCTION_NOT_SUPPORTED;
32         else {
33             rv = restore_login_state(slot);
34             if (rv == CKR_OK)
35                 rv = slot->p11card->framework->gen_key(slot,
                       pMechanism,
36                      pTemplate, ulCount, phKey);
37                 rv = reset_login_state(session->slot, rv);
38         }
39 
40 out:
41     sc_pkcs11_unlock();
42     return rv;
43 }
```

- *FRAMEWORK-PKCS15*

  Como já mencionado, a implementação de *pkcs11* do OpenSC utiliza as definições do pdrão *pkcs11* para os atributos. É nesta classe que os parâmetros *pkcs11* são convertidos para *pkcs15*, afim de criar uma estrutura padronizada para inserção nos cartões. Além disso, definem-se os parâmetros de uso (*usage*) e acesso (*access_flags*) da chave. Assim, como no passo anterior, sessões e *locks* são criados para a comunicação. Tudo isto culmina na chamada de método que tenta gerar a chave no cartão propriamente dito:

```
1  static CK_RV
2  pkcs15_gen_key(struct sc_pkcs11_slot *slot,CK_MECHANISM_PTR
       pMechanism,
3          CK_ATTRIBUTE_PTR pTemplate, CK_ULONG ulCount,
4          CK_OBJECT_HANDLE_PTR phKey)
5  {
6       // ...
7       rc = sc_pkcs15init_generate_symmetric(fw_data->p15_card,
            profile, &keyargs, keybits, &key_obj);
8      if (rc < 0) {
9          sc_log(context, "sc_pkcs15init_generate_key returned %d", rc);
10         rv = sc_to_cryptoki_error(rc, "C_GenerateKey");
11         goto kpgen_done;
12     }
13     // ...
14 }
```

- *PKCS15-LIB*

  A escolha da assinatura *generate_symmetric* se deu, principal-

mente, pelo fato de *generate_key* ser utilizado na classe, para geração de chaves assimétricas. Na função, inicialmente é feita uma verificação, a fim de identificar se a geração de chaves simétricas é suportada pelo *applet*, também é realizada uma checagem do identificador escolhido em passos anteriores.

```
1   int sc_pkcs15init_generate_symmetric(struct sc_pkcs15_card *p15card,
2          struct sc_profile * profile ,
3          struct sc_pkcs15init_skeyargs *keygen_args, unsigned int keybits,
4          struct sc_pkcs15_object **res_obj)
5   {
6       // ..
7       if ( profile ->ops->generate_symmetric == NULL)
8          LOG_TEST_RET(ctx, SC_ERROR_NOT_SUPPORTED,
9                "Secret Key generation not supported");
10
11      if (keygen_args->id.len) {
12         r = sc_pkcs15_find_skey_by_id(p15card, &keygen_args->id,
                NULL);
13         if (!r)
14            LOG_TEST_RET(ctx, SC_ERROR_NON_UNIQUE_ID,
15                  "Non unique ID of the secret key object");
16         else if (r != SC_ERROR_OBJECT_NOT_FOUND)
17            LOG_TEST_RET(ctx, r, "Find secret key error");
18      }
```

A versão original do OpenSC não implementa as definições para os arquivos de chave secreta (SKDF), bem como a criação de seu objeto. Para tanto, modificou-se o arquivo *isoApplet.profile* — que reúne informações de *drivers* e sistemas de arquivos do *applet* — com as adições necessárias.

```
1   # PKCS15 profile, generic information.
2   # This profile is loaded before any card specific profile .
3   # ...
4   # Default settings.
5   # This option block will always be processed.
6   option default {
7       macros {
8           # ...
9           skdf-size = 256;
10      }
11  }
12  # ...
13  filesystem {
14      # ...
15      # Here comes the application DF
16      DF PKCS15-AppDF {
17          # ...
```

```
18            EF PKCS15-SKDF {
19             file-id = 4406;
20             size     = $skdf-size;
21             acl      = $protected;
22            }
23        }
24 }
```

Desta forma, pode-se criar um objeto SKDF. Para que em seguida o *driver* do Iso Applet seja chamado.

```
1     // ...
2     r = sc_pkcs15init_init_skdf(p15card, profile, keygen_args, keybits,
          &object);
3     // ...
4     r = profile->ops->generate_symmetric(profile, p15card, object);
5     //...
```

- *PKCS15-ISOAPPLET*

  Esta classe funciona como uma ponte entre o mediador e o *driver* do *applet*. Nela é onde, entre outras coisas, configuram-se os APDU's para o tamanho da chave e o algoritmo a ser enviado pelo *driver card-isoApplet*.

```
1  static int
2  isoApplet_generate_key_aes(sc_pkcs15_skey_info_t *key_info,
        sc_card_t *card)
3  {
4      int rv;
5      size_t keybits;
6      struct sc_cardctl_isoApplet_gen_symmetric args;
7
8      LOG_FUNC_CALLED(card->ctx);
9
10     /* Check key size: */
11     keybits = key_info->value_len;
12     if (keybits != 256 && keybits != 128)
13     {
14         rv = SC_ERROR_INVALID_ARGUMENTS;
15         sc_log(card->ctx, "%s: AES secret key length is unsupported,
               correct length is 128 or 256", sc_strerror(rv));
16         goto err;
17     }
18
19     memset(&args, 0, sizeof(args));
20     switch(keybits){
21         case 128:
22             args.algorithm_ref = SC_ISOAPPLET_ALG_REF_AES_128;
```

```
23              break;
24          case 256:
25              args.algorithm_ref = SC_ISOAPPLET_ALG_REF_AES_256;
26              break;
27      default:
28          args.algorithm_ref = SC_ISOAPPLET_ALG_REF_AES_128;
29      }
30      args.s_key_ref = key_info->key_reference;
31
32      rv = sc_card_ctl(card,
            SC_CARDCTL_ISOAPPLET_GENERATE_SYMMETRIC,
            &args);
33      if (rv < 0)
34      {
35          sc_log(card->ctx, "%s: Error in card_ctl", sc_strerror(rv));
36          goto err;
37      }
38
39  err:
40      LOG_FUNC_RETURN(card->ctx, rv);
41  }
```

- *CARD-ISOAPPLET*

  Trata-se da última classe a ser consultada. Onde os APDU's necessários à geração da chave são enviados ao cartão. Havendo a necessidade da adição de um novo caso ao método *card_ctl*, que originalmente suportava apenas geração de chaves assimétricas:

```
1   isoApplet_card_ctl(sc_card_t *card, unsigned long cmd, void *ptr)
2   {
3        // ...
4       switch (cmd)
5       {
6       //...
7         case SC_CARDCTL_ISOAPPLET_GENERATE_SYMMETRIC:
8               r = isoApplet_ctl_generate_symmetric(card,
9                       (sc_cardctl_isoApplet_gen_symmetric_t *) ptr);
10      break;
11      //...
12  }
```

  Em conclusão, o método *isoApplet_ctl_generate_symmetric* realiza a configuração e o envio das APDU's necessárias para a geração da chave. Primeiramente com a execução da função *manageSecurityEnviroment*, seguida pela função de geração propriamente dita, como explicado na sessão 3.2.1.1.

```
static int
isoApplet_ctl_generate_symmetric(sc_card_t *card,
        sc_cardctl_isoApplet_gen_symmetric_t *args)
{
    int r;
    sc_apdu_t apdu;
    u8 sbuf[SC_MAX_EXT_APDU_BUFFER_SIZE];
    u8 *p;
    char buff_sw[80];

    LOG_FUNC_CALLED(card->ctx);

    sc_format_apdu(card, &apdu, SC_APDU_CASE_3_SHORT, 0x22,
        0x42, 0x01);

    p = sbuf;
    *p++ = 0x80;
    *p++ = 0x01;
    *p++ = args->algorithm_ref;
    *p++ = 0x84;
    *p++ = 0x01;
    *p++ = args->s_key_ref;
    r = p - sbuf;
    p = NULL;

    apdu.lc = r;
    apdu.datalen = r;
    apdu.data = sbuf;

    r = sc_transmit_apdu(card, &apdu);
    LOG_TEST_RET(card->ctx, r, "APDU transmit failed");
    r = sc_check_sw(card, apdu.sw1, apdu.sw2);
    LOG_TEST_RET(card->ctx, r, "Card returned error");

    /* GENERATE SYMMETRIC KEY*/
    sc_format_apdu(card, &apdu, SC_APDU_CASE_1, 0x30, 0x00,
        0x00);

    r = sc_transmit_apdu(card, &apdu);
    LOG_TEST_RET(card->ctx, r, "APDU transmit failed");

    r = sc_check_sw(card, apdu.sw1, apdu.sw2);
    if(apdu.sw1 == 0x6A && apdu.sw2 == 0x81){
        sc_log(card->ctx, "Key generation not supported by the card with
            that particular key type. "
            "Your card may not support the specified algorithm used by
                the applet / specified  by you. ");
    }
    LOG_TEST_RET(card->ctx, r, "Card returned error");
    LOG_FUNC_RETURN(card->ctx, SC_SUCCESS);
}
```

Todas as alterações necessárias à geração de chaves AES, nos tamanhos 128 e 256 bits, aqui propostas, podem ser verificadas no anexo A2. Parte delas não foram apresentadas neste capítulo, afim de condensar o conhecimento e simplificar as explicações.

4 RESULTADOS

Os principais resultados alcançados neste trabalho estão descritos nas tabela 6 e 7. Um *applet* de código aberto capaz de realizar funções de criptografia simétrica e assimétrica, seguindo as recomendações da norma ISO 7816. E uma versão do *middleware* OpenSC, que abstrai a geração de chaves AES em *Java Cards*.

Neste ponto vale ressaltar um cenário relevante de uso dos protótipos gerados: envelopamento de chave. Como descrito na introdução deste trabalho, empresas e pesquisadores ao redor do mundo optam, em determinados casos, pelo uso de aplicações *open source*, um exemplo real disto é a produção de *software* e pesquisa realizada pelo laboratório de segurança em computação da Universidade Federal de Santa Catarina, parte da motivação pela busca dos resultados aqui alcançados. E o envelopamento de chaves é uma das necessidade do laboratório.

Sabe-se que a cifragem de dados com uma chave AES é relativamente mais eficiente do que o uso de, por exemplo, um par de chaves RSA. Com o envelopamento, pode-se cifrar primeiramente os dados sigilosos com uma chave AES, em seguida cifrar a mesma com criptografia assimétrica e enviar ambos resultados, ou seja, o dado cifrado e a chave cifrada, ao destinatário, que terá apenas o trabalho de decifrar a chave AES para usá-la na operação de decifragem do dado sigiloso.

	ISO
Adições	gen_skey get_skey ciph_skey deciph_skey load_skey
Norma	ISO 7816

Tabela 6 – Protótipo Iso Applet.

	OPENSC
Adição	C_GenerateKey
Normas	PKCS#11 e PKCS#15

Tabela 7 – Protótipo OpenSC.

4.1 TESTES COM OS PROTÓTIPOS

Através dos teste descritos nesta sessão, pode-se verificar a utilidade das adições realizadas. Para tanto, foram utilizados *Java Cards* NXP J2A080 2.2.2 (3b:f5:13:00:00:81:31:fe:45:73:74:64:31:30:8f), as versões modificadas das aplicações e o sistema operacional Ubuntu. Bem como a biblioteca PC/SC para a comunicação com os cartões e a leitora de *Smart Cards*: SCM Microsystems, Inc. SCR331-LC1 / SCR3310 SmartCard Reader.

4.1.1 Instalação do applet

Utilizando implementações do padrão GlobalPlatform nota-se que, inicialmente, os *Java Cards* contam com apenas uma aplicação instalada, responsável pela gerência do *hardware*.

```
# Mode: GP22
ISD: A000000151000000 (OP_READY)
     Privs:   SecurityDomain, CardLock, CardTerminate, CardReset, CVMManagement
```

Figura 21 Memória do *Java Card* antes do upload do applet. Fonte: Criada pelos autores.

Para realizar o *upload* da versão modificado do Iso applet, pode-se executar a função *install* da aplicação GlobalPlatformPro, em conjunto à um arquivo XML, que descreve os parâmetros de conversão das classes Java em um arquivo CAP.

```
# Mode: GP22
ISD: A000000151000000 (OP_READY)
     Privs:   SecurityDomain, CardLock, CardTerminate, CardReset, CVMManagement

APP: F276A288BCFBA69D34F31001 (SELECTABLE)

PKG: F276A288BCFBA69D34F310 (LOADED)
     Applet:  F276A288BCFBA69D34F31001
```

Figura 22 Memória do *Java Card* após o upload do applet. Fonte: Criada pelos autores.

Os arquivos XML contém informações descritivas da aplicação, como: AID, local dos arquivos fonte e nome do arquivo resultante da conversão.

```
1  <?xml version="1.0" encoding="UTF−8"?>
2  <project name="JavaCard PKI isoApplet" default="dist" basedir=".">
3    <description>Builds the project. </description>
4    <target name="dist" description="generate the distribution">
5      <tstamp/>
6      <taskdef name="javacard" classname="pro.javacard.ant.JavaCard"
           classpath="ext/ant/ant−javacard.jar"/>
7      <javacard>
8        <cap jckit="java_card_kit−2_2_2/"
             aid="f2:76:a2:88:bc:fb:a6:9d:34:f3:10" output="IsoApplet.cap"
             sources="src" version="1.0">
9          <applet class="net.pwendland.javacard.pki.isoapplet.IsoApplet"
               aid=" f2:76:a2:88:bc:fb:a6:9d:34:f3:10:01 "/>
10       </cap>
11     </javacard>
12   </target>
13 </project>
```

O ultimo passo antes da geração de chaves secretas é a criação de uma estrutura de arquivos PKCS#15, que será responsável pelo armazenamento dos objetos de forma padronizada. Para tanto, pode-se executar a opção "-*C*"do utilitário *pkcs15-init* da biblioteca OpenSC. O comando também armazena um PIN associado à estrutura, requisitado ao usuário. A execução e o resultado desta operação, podem ser verificados nas figuras 17 e 18 do capítulo 3.

4.1.2 Geração das chaves

Neste ponto, o *Java Card* está pronto para o armazenamento de dados, chaves ou certificados. As figuras 23 e 24 demonstram como chaves AES (128 ou 256 bits) podem ser geradas com as adições propostas na sessão 3.3. Nas linhas de comando deve-se indicar o módulo PKCS#11 a ser utilizado, neste caso *opensc-pkcs11.so*, um rótulo para a chave (caso não seja informado, será utilizado um rótulo padrão), o tamanho em bytes e um identificador numérico.

- 128 bits:

  pkcs11-tool –module /usr/local/lib/pkcs11/opensc-pkcs11.so -l –pin 123456 –keygen –label AES128 –key-type "aes:16--id 01

```
PKCS#15 Card [JavaCard isoApplet]:
        Version         : 0
        Serial number   : 0000
        Manufacturer ID: unknown
        Last update     : 20170523225108Z
        Flags           : EID compliant
PIN [User PIN]
        Object Flags    : [0x3], private, modifiable
        ID              : ff
        Flags           : [0x39], case-sensitive, unblock-disabled, initialized, needs-padding
        Length          : min_len:4, max_len:16, stored_len:16
        Pad char        : 0x00
        Reference       : 1 (0x01)
        Type            : ascii-numeric

Secret AES Key [AES128]
        Object Flags    : [0x2], modifiable
        Usage           : [0x33], encrypt, decrypt, wrap, unwrap
        Access Flags    : [0x11], sensitive, local
        Size            : 128 bits
        ID              : 01
        Native          : no
        Key ref         : 0 (0x0)
        Path            : 3f0050153600
```

Figura 23 Memória do *Java Card* após a geração de uma chave AES 128 bits. Fonte: Criada pelos autores.

- 256 bits:

  pkcs11-tool module /usr/local/lib/pkcs11/opensc-pkcs11.so -l pin 123456 keygen label AES256 key-type "aes:32--id 02

Analisando as duas chaves gerados anteriormente, nota-se que além dos dados propositalmente diferentes, como ID e *label*, o valor de *Key ref*, também é modificado, mas a nível de aplicação.

4.1.3 Outras funções oferecidas

Pode-se demonstrar que as chaves geradas através do OpenSC podem ser aproveitas nas demais funções adicionadas ao *applet*. Ainda que a cifragem de dados dentro dos cartões não seja uma operação usual, devido a sua limitação de hardware e velocidade na transferência de dados, pode-se cifrar por exemplo blocos de 128 bits, como a palavra "labsec"(6C6162736563), utilizando uma chave do mesmo tamanho, cujo o *Key ref* é igual a zero:

```
PKCS#15 Card [JavaCard isoApplet]:
        Version        : 0
        Serial number  : 0000
        Manufacturer ID: unknown
        Last update    : 20170523225951Z
        Flags          : EID compliant
PIN [User PIN]
        Object Flags   : [0x3], private, modifiable
        ID             : ff
        Flags          : [0x39], case-sensitive, unblock-disabled, initialized, needs-padding
        Length         : min_len:4, max_len:16, stored_len:16
        Pad char       : 0x00
        Reference      : 1 (0x01)
        Type           : ascii-numeric

Secret AES Key [AES128]
        Object Flags   : [0x2], modifiable
        Usage          : [0x33], encrypt, decrypt, wrap, unwrap
        Access Flags   : [0x11], sensitive, local
        Size           : 128 bits
        ID             : 01
        Native         : no
        Key ref        : 0 (0x0)
        Path           : 3f0050153600

Secret AES Key [AES256]
        Object Flags   : [0x2], modifiable
        Usage          : [0x33], encrypt, decrypt, wrap, unwrap
        Access Flags   : [0x11], sensitive, local
        Size           : 256 bits
        ID             : 02
        Native         : no
        Key ref        : 1 (0x1)
        Path           : 3f0050153601
```

Figura 24 Memória do *Java Card* após a geração de uma chave AES 256 bits. Fonte: Criada pelos autores.

```
Received (SW1=0x90, SW2=0x00):
8E 41 89 56 91 47 DB 74 23 0E 3C 8B D3 EE 3F E8 .A.V.G.t#.<...?.
```

Figura 25 Resultado da cifragem da palavra "labsec". Fonte: Criada pelos autores.

Em seguida, pode-se usar o valor resultante como entrada da função *deciph*, para uma prova real:

```
Received (SW1=0x90, SW2=0x00):
6C 61 62 73 65 63 00 00 00 00 00 00 00 00 00 00 labsec..........
```

Figura 26 Resultado da decifragem do dado no passo anterior.

Assim, três das funções adicionadas no *applet* foram executadas. Já as funções de extração e carregamento podem ser testadas nos seguintes passos: extrai-se a chave de 128 bits, armazenando-a localmente, seguido por uma limpeza do cartão e a inserção da chave novamente no *Java Card*, através do *load*. Porém as chaves carregadas sem o uso do OpenSC, ou seja, apenas com o envio dos APDUS, não

são apontadas pela estrutura PKCS#15 e por esse motivo não podem ser visualizadas com o utilitário *pkcs15-init*.

4.2 JAVA CARD MANAGER

Embora as aplicações desenvolvidas pela comunidade de *software* livre, que utilizam interface por linhas de comando, possam ser bastante intuitivas com a prática, pensando em um ambiente de produção, ou até mesmo de desenvolvimento, nos casos de uma equipe não especializada, no desenrolar deste trabalho foi desenvolvida uma aplicação codificada na linguagem de programação C++ e com interface gráfica intuitiva em QT (biblioteca multiplataforma para desenvolvimento de aplicações *desktop*), capaz de realizar a instalação dos *applets* aqui estudados, bem como a geração e armazenamento de chaves e certificados. Como pode ser visto na figura 27, a aplicação conta com 4 distintas funções:

- *Install Applet*: Primordialmente estão disponíveis os *applets* Muscle e Iso. Porém é oferecida a opção de *upload* de novos arquivos CAP, para a instalação.
- *Delete Applet*: Feita a escolha da aplicação a ser deletada do *Java Card*, o software executa a completa remoção dos dados relacionados ao *applet* da memória do cartão.
- *Upload Keys*: Para o *upload* de chaves existem uma gama de opções. Pode-se gerar as chaves de forma externa ao cartão, através da biblioteca OpenSSL, para que em seguida o objeto seja carregado no *Java Card*. Mas também é possível gera-las internamente com o auxilio do OpenSC.
- *Informations*: Informações relevantes são mostradas ao usuário, como: *applets* em memória e arquivos PKCS#15 armazenados.

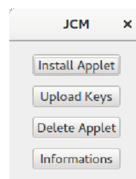

Figura 27 Tela inicial do *Java Card* Manager.

Em cada uma destas situações o usuário é apresentado à um *wizzard*, responsável pela colheita dos dados ou opções necessária à execução das funções. A figura 28 mostra um exemplo de fluxo de telas, neste caso, a partir da opção *Install Applet*.

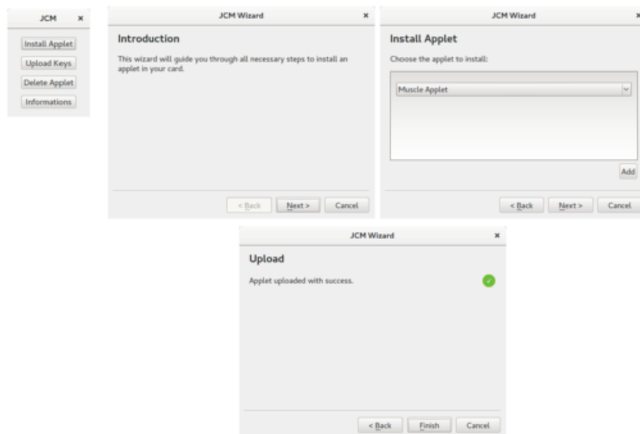

Figura 28 Exemplo de fluxo de telas no JCM. Fonte: Criada pelos autores.

5 CONCLUSÕES

O problema descrito no capítulo de introdução deste trabalho, onde é apontada a falta de um *applet* de código aberto que ofereça funções de criptografia simétrica, foi resolvido, como demonstrado nas sessões 3.2.1.1 e 4.1.2, em que foram adicionadas e testadas cinco novas funções no Iso Applet, um projeto de código aberto em conformidade com o padrão ISO 7816. O mesmo se aplica ao objetivo traçado com relação ao OpenSC, relacionado ao suporte na geração de chaves AES, contemplado nas sessões 3.3 e 4.1.2,

Embora os principais objetivos tenham sido alcançados no desenrolar do desenvolvimento, vale ressaltar alguns pontos de fragilidade e limitações das soluções apresentadas. Na geração de chaves AES no *applet*, não são oferecidos todos os tamanhos de chave descritos pelo padrão, a ser adicionado o tamanho 192 bits. Além disso, as funções de cifragem e decifragem poderiam ter sido englobadas como um novo caso (um caso de chaves simétricas) nos métodos já existentes na aplicação, no lugar de possuírem assinaturas próprias.

No que diz respeito ao OpenSC, além do suporte à geração das chaves, poderiam ter sido implementadas as demais funções de criptografia simétrica da sessão 3.2.1.1.

Pôde-se observar que os materiais relacionados a *Smart Cards*, desenvolvimento de *Java Card applets* e sobre os padrões ISO 7816 e PKCS, são bastante limitados e muitas vezes confusos. Para o entendimento do funcionamento básico dos cartões vale a leitura dos trabalhos de mestrado e graduação citados aqui como referência. Já o desenvolvimento das aplicações é mais facilmente compreendido, com uma análise cuidadosa de casos conhecidos, como o próprio Iso Applet ou Muscle Applet. No caso dos padrões mencionados, as descrições do PKCS podem ser encontradas facilmente nos *web* sites da *RSA Laboratories*, o que não acontece para as normas ISO. Os diferentes tópicos da ISO 7816 são patenteados, com poucas acessões que podem ser encontradas fragmentadas em sites especializados.

5.1 TRABALHOS FUTUROS

Com base nos resultados aqui apresentados, podem-se destacar alguns temas para trabalhos futuros, tanto relacionados à pesquisa, quanto ao desenvolvimento. Como por exemplo, um estudo das di-

ferentes funções de criptografia oferecidas por aplicações privadas em *Smart Cards* e as oferecidas pelos *applets* de código aberto como Iso Applet. Adição do algoritmo DES nas opções de chaves secretas no Iso Applet, pois mesmo sendo um algoritmo em fase de substituição, ainda é o mais usado mundialmente. Ainda sobre o Iso Applet, um estudo do nível de segurança das implementações propostas neste trabalho, no que diz respeito à ataques externos.

Para o mediador OpenSC, pode-se implementar o suporte as demais funções adicionadas ao *applet*, que não foram cobertas por este trabalho: extração da chave, carregamento de uma chave gerada externamente, cifragem e decifragem de dados.

6 DIREITOS AUTORAIS

Ainda que as aplicações utilizadas no desenvolvimento deste trabalho sejam de código aberto e estejam disponíveis em serviços de repositório de livre acesso na *internet*. Vale lembrar os nomes ou equipes que as desenvolveram e possibilitaram, a existência das adições e melhorias propostas.

Em destaque o nome do principal desenvolvedor ou equipe responsável, seguido pelo endereço *url* do projeto:

- Iso Applet: Philip Wendland.
  https://github.com/philipWendland/IsoApplet
- Muscle Applet: Martin Paljak.
  https://github.com/martinpaljak/MuscleApplet
- OpenSC: Equipe OpenSC.
  https://github.com/OpenSC/OpenSC
- GlobalPlatformPro: Martin Paljak.
  https://github.com/martinpaljak/GlobalPlatformPro
- ant-javacard: Martin Paljak.
  https://github.com/martinpaljak/ant-javacard
- OpenSSL: Equipe OpenSSL
  https://www.openssl.org/source/

REFERÊNCIAS

DETTONI, C. L. Implementação de cartão de identificação acadêmico baseado no padrão icao-9303. *Universidade Federal de Santa Catarina, Bacharelado em Ciências da Computação*, 2015.

GLOBALPLATFORM. *GlobalPlatform*. 2017. <https://www.globalplatform.org/>. Acessado em 10/08/2016.

GRINGERI, E. *Improving security and memory management in the muscle card framework*. Tese (Doutorado) — Universitá degli studi di Pisa, Pisa, 2008–2009.

ISO/IEC. *International Standart ISO/IEC 7816-4*. Bedford, EUA, 2005.

KNOSPE, H. Smart card technology and linux integration. *Cologne University of Applied Sciences*, 2006.

LABORATORIES, R. *Public-key Crytography Standards (PKCS)*. 2017. <https://brazil.emc.com/emc-plus/rsa-labs/standards-initiatives/public-key-cryptography-standards.htm>. Acessado em 21/02/2017.

OPENSC. *Overview OpenSC*. Janeiro 2017. <https://github.com/OpenSC/OpenSC/wiki/Overview>. Acessado em 5/02/2017.

ORACLE. *Java Card Documentation*. 2017. <http://www.oracle.com/technetwork/java/embedded/javacard/>. Acessado em 15/06/2016.

PC/SC, W. *PCSC*. 2017. <https://www.pcscworkgroup.com/>. Acessado em 1/03/2017.

RSA, L. *PKCS#11 Base Functionality v2.30: Criptoki - Draft 4*. Bedford, EUA, 2009.

STALLINGS, W. *Cryptography and Network Security*. New Jersey, EUA: Ed. Pearson, 2014. 63-74,132-136 p.

STANDARDIZATION, I. O. for. *About ISO*. 2016. <https://www.iso.org/about-us.html>. Acessado em 14/11/2016.

WENDLAND, P. *General Information of IsoApplet*. 2016. <https://github.com/philipWendland/IsoApplet>. Acessado em 19/05/2016.

APÊNDICE A – Comunicação com applet via APDU

A.1 COMUNICAÇÃO COM APPLET VIA APDU

Sabe-se que a comunicação entre os *Java Cards* e as aplicações externas se da pela troca de APDU's. Nesta sessão, pode-se verificar a execução das funções explanadas no capítulo 3, utilizando o comando *-s* do utilitário *opensc-tool*. Outros projetos abertos como o *software* em Java *apdu4j*, oferecem a mesma funcionalidade.

- *Generate Secret Key*:

  Para a geração de uma chave AES de 256 bits, por exemplo, são necessários os seguintes comandos realizados em sequências:

```
1 opensc-tool -s 00:A4:08:00:04:50:15:44:01:00
2 -s 00:B0:00:00:00
3 -s  00:20:00:01:10:31:32:33:34:35:36:00:00:00:00:00:00:00:00:00:00
4 -s 00:A4:08:0C:02:50:15
5 -s 00:22:42:01:06:80:01:A2:84:01:00
6 -s 00:30:00:00
```

  Os quatro primeiros APDU's são responsáveis pela seleção, leitura e verificação do PIN, que neste exemplo é "123456". Seguidos pelo comando *Manage Security Enviroment*, que configura 256 bits para o tamanho da chave e zero para a posição no arranjo de chaves secretas. Finalmente a última combinação de hexadecimais caracteriza o comando de geração propriamente dito, com o valor de INS igual a 30.

  A mesma ideia se aplica a chaves 128 bits, com uma sutil diferença no comando de configuração. O hexadecimal A2 passa a ser A1.

```
1 opensc-tool -s 00:A4:08:00:04:50:15:44:01:00
2  -s 00:B0:00:00:00
3  -s  00:20:00:01:10:31:32:33:34:35:36:00:00:00:00:00:00:00:00:00:00
4  -s 00:A4:08:0C:02:50:15
5  -s 00:22:42:01:06:80:01:A1:84:01:00
6  -s 00:30:00:00
```

- *Get Secret Key*:

  Uma vez gerada ou carregada uma chave AES no Iso Applet, pode-se extraí-la com a função *processGetSKey*. Semelhante ao comando de geração de chaves, os primeiros APDU's realizam a verificação do PIN, seguidos pela configuração da referência da chave e do comando de extração propriamente dito:

```
1  opensc−tool −s 00:A4:08:00:04:50:15:44:01:00
2  −s 00:B0:00:00:00
3  −s  00:20:00:01:10:31:32:33:34:35:36:00:00:00:00:00:00:00:00:00:00
4  −s 00:A4:08:0C:02:50:15
5  −s 00:22:42:02:03:84:01:00
6  −s 00:32:00:00
```

- *Ciph*:

  Para a cifragem de dados, utilizando qualquer um dos dois tamanhos possíveis de chaves AES necessita-se de: comandos para a autenticação do PIN, configuração da chave a ser utilizada na operação e envio do dado a ser cifrado. O *plaintext* pode assumir valores de até 128 bits, uma vez que o cifrador AES funciona em blocos deste tamanho.

```
1  opensc−tool −s 00:A4:08:00:04:50:15:44:01:00
2  −s 00:B0:00:00:00
3  −s  00:20:00:01:10:31:32:33:34:35:36:00:00:00:00:00:00:00:00:00:00
4  −s 00:A4:08:0C:02:50:15
5  −s 00:22:42:02:03:84:01:00
6  −s 00:34:00:00:08:70:69:7A:7A:61:73:65:63
```

  No último comando pode-se notar o valor de INS 34, que representa o comando de cifragem, seguido por dois campos 00 e pelo campo LC, que indica o tamanho do dado a ser cifrado. Neste exemplo oito bytes são enviados e representam o texto "pizzasec".

- *Deciph*:

  Pode-se decifrar o item anterior, utilizando-se a mesmo valor de chave, porém com diferentes comandos:

```
1  opensc−tool −s 00:A4:08:00:04:50:15:44:01:00
2  −s 00:B0:00:00:00
3  −s  00:20:00:01:10:31:32:33:34:35:36:00:00:00:00:00:00:00:00:00:00
4  −s 00:A4:08:0C:02:50:15
5  −s 00:22:42:02:03:84:01:00
6  −s 00:36:00:00:10:F2:5B:8F:A1:9D:1E:B9:A0:61:BA:42:E6:A2:4A:4A:13
```

- *Load*:

  Finalmente, para *upload* de chaves secretas basta que seja executado uma sequência semelhante à:

```
1 opensc−tool −s 00:A4:08:00:04:50:15:44:01:00
2 −s 00:B0:00:00:00
3 −s  00:20:00:01:10:31:32:33:34:35:36:00:00:00:00:00:00:00:00:00:00
4 −s 00:A4:08:0C:02:50:15
5 −s 00:22:42:02:06:80:01: A1:84:01:01
6 −s 00:38:00:00:10:59:7C:5A:D0:46:28:69:CD:14:2D:75:7C:EF:7A:AA:E1
```

A chave acima, de 128 bits, é armazenada na posição "01"do *array* de chaves secretas, seu valor está representado no último comando após o quinto *byte*.

Conclui-se deste apêndice, que a execução de funções em um *applet* através do enviando comandos APDU, utilizando as bibliotecas OpenSC e PCSC, funciona de forma bastante satisfatória. Porém em um ambiente de produção, em que necessitam-se de soluções com maior abstração, utilitários como *pkcs11-tools* podem ser mais interessantes.

APÊNDICE B – Artigo

Adição de funções de criptografia simétrica em um applet de código aberto com suporte via middleware OpenSC

Lucas M. Palma[1], **Luiz H. U. Sousa**[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina
Florianópolis, Santa Catarina, Brasil

{lucas.palma,luiz.urias}@grad.ufsc.br

Abstract. *The relevant role of symmetric cryptography can be followed from its birth in the 1970s, through the creation of other techniques, such as public key cryptography, until the 21st century. With it, we can guarantee, mainly, the secrecy of the data. It is known that in this context, only one key is generated and has the dual power to encrypt and decipher information. The results of these operations are strongly related to the chosen key. For this reason, while the algorithms are known and analyzed exhaustively, the key must be stored securely. This article deals with the use of Java Cards as a means of storing them, through applets supported by the OpenSC middleware, developed according to international standards. It is proposed to add symmetric encryption functions in an open-source applet using the Advanced Encryption Standard (AES) algorithm for 128-bit and 256-bit keys.*

Resumo. *O papel relevante da criptografia simétrica pode ser acompanhado desde seu nascimento na década de 1970, passando pela criação de outras técnicas, como a criptografia de chaves públicas, até o século XXI. Com ela, pode-se garantir, principalmente, o sigilo dos dados. Sabe-se que neste contexto, apenas uma chave é gerada e possui o poder dual, de cifrar e decifrar informações. Os resultados destas operações estão fortemente relacionados à chave escolhida. Por esse motivo, enquanto os algoritmos são conhecidos e analisados exaustivamente, a chave deve ser armazenada de forma segura. Este artigo trata do uso de Java Cards como meio de armazená-las, através de applets suportados pelo middleware OpenSC, desenvolvidos seguindo padrões internacionais. Propõe-se a adição de funções de criptografia simétrica, em um applet de código aberto, utilizando o algoritmo AES (Advanced Encryption Standard) para chaves de 128 e 256 bits.*

1. Introdução

Empresas como a europeia [A.E.T 2016], líder na área de segurança digital, oferecem soluções de autenticação com *Smart Cards* seguindo padrões internacionalmente estabelecidos, como as normas ISO 7816 [ISO/IEC 2005] e PKCS (*Public-Key Cryptography Standards*). Uma variação importante dos *Smart Cards* são os *Java Cards*, que executam aplicações codificadas na linguagem de programação Java, denominadas *applets*. Para estes, pode-se encontrar uma coleção de *applets* criptográficos de código aberto, disponíveis em serviços de repositório como o GitHub. A importância destes projetos está na liberdade de acesso e contribuição por parte de pesquisadores ou empresas, que optam por soluções *Open Source*.

Utilizado pela maioria das Autoridades Certificadoras e *softwares* que acessam *Smart Cards*, o padrão PKCS#11 [RSA 2009] define um compilado de regras e funções para *tokens* criptográficos. Em conjunto com outros padrões como [GlobalPlatform 2016] e ISO 7816, possibilita a criação de cartões capazes de realizar funções como: geração de chaves simétricas e assimétricas, cifragem e decifragem de dados e armazenamento de certificados digitais.

As aplicações contidas nos cartões, devem ser instruídas, quanto as funções a serem realizadas. O responsável por esse gerenciamento é conhecido como *middleware*, ou mediador. Um *middleware* bastante difundido é o [OpenSC 2017], projeto de código aberto, com mais de dez anos de história e dezenas de contribuidores.

O OpenSC implementa parte significativa da API PKCS#11, principalmente as funções relacionadas a autenticação, cifragem de correio eletrônico e assinatura digital. Porém não inclui àquelas ligadas a criptografia simétrica, que utilizam algoritmos como *DES* (Data Encryption Standard) e *AES* (Advanced Encryption Standard). Sendo esse, um dos prováveis motivos pelos quais, a maioria dos *applets* de código aberto encontrados, também não suportam estas funções.

Busca-se selecionar, dentre diversas opções, um *applet* de código aberto que implemente o maior número de funções definidas pelo padrão PKCS#11 e que respeite outras normas internacionais, como a ISO 7816. Visando colaborar com a comunidade *Open Source* e conceber um *applet* mais condizente com a gama de funções definidas pelo padrão PKCS#11, este trabalho prototipa uma solução capaz de gerar, carregar, exportar, cifrar e decifrar, utilizando chaves AES, nos tamanhos de 128 e 256 bits. Mais do que isso, demonstra-se uma alternativa de como adicionar suporte à uma destas funções no *middleware* OpenSC. Não estão incluídas a geração e uso de chaves AES 192 bits, bem como o uso de diferentes modos de operação, além do CBC, na função de cifragem de dados.

2. Conceitos

2.1. Advanced Encryption Standard

Cifradores simétricos, também conhecidos como cifradores convencionais, são, ainda hoje, os mais utilizados em comparação a outras técnicas de criptografia, como os cifradores de chave pública (assimétricos). Mundialmente, os algoritmos de criptografia simétrica mais usados são: DES (Data Encryption Standard) e AES (Advanced Encryption Standard).

Publicado em 2001 pelo NIST (National Institute of Standards and Technology), resultante da busca por um algoritmo que substituísse o padrão até então utilizado, DES. O AES é um cifrador de blocos de 16 bytes (128 bits), ou seja, os dados de entrada são sempre divididos em porções de 128 bits. O padrão aceita três tamanhos distintos de chave: 16 bytes (128 bits), 24 bytes (192 bits) e 32 bytes (256 bits). A cifragem e decifragem consistem em n rodadas, definidas pelo tamanho da chave escolhida, cada rodada efetua uma série de manipulações sobre o dado original.

Key Size (words/bytes/bits)	4/16/128	6/24/192	8/32/256
Plaintext Block Size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Number of Rounds	10	12	14
Round Key Size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Expanded Key Size (words/bytes)	44/176	52/208	60/240

Figura 1. Parâmetros do AES. Fonte: [Stallings 2014]

2.2. Java Card

Pode-se notar o uso de *Smart Cards* em uma série de situações do cotidiano das grande cidades, adotando papeis como cartões de autenticação ou cartões de crédito. São equipados com microprocessadores e diferentes tipos de memória, capazes de realizar funções computacionais. Além destas características, os *Java Cards* contam com o suporte de uma Java Card RE (Runtime Enviroment), que define um subconjunto da linguagem Java e uma máquina virtual para a execução das aplicações.

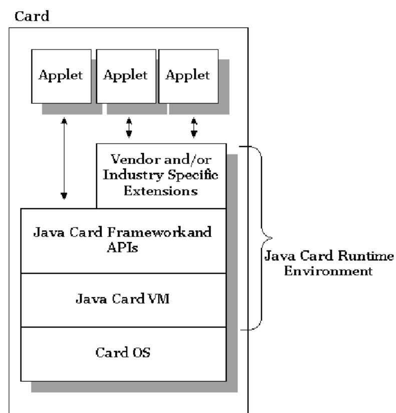

Figura 2. Estrutura de um Java Card. Fonte: [Gringeri 2009]

2.2.1. Applet

As aplicações que executam em um *Java Card* são denominadas *Java Card Applets*. Semelhantes à aplicações Java *desktop*, possuem um conjunto de classes, mas que posteriormente são convertidas para um formato específico a ser carregado no cartão, CAP (Converted Applet). Toda aplicação instalada é identificada por um AID e possui uma região de memória exclusiva.

2.2.2. Comunicação

A comunicação com os *Java Cards* se dá pelo envio de APDUs, uma sequência de bytes estruturados, que possibilitam a troca de dados e informações, entre os cartões e as aplicações externas. Existem diferentes configurações para as sequência de bytes, dependendo principalmente, se há troca ou não de dados.

Command APDU						
Header (required)				Body (optional)		
CLA	INS	P1	P2	Lc	Data Field	Le

Figura 3. Exemplo da estrutura de um APDU. Fonte: [Gringeri 2009]

2.3. ISO 7816

A ISO 7816 é um conjunto de padrões relacionados à cartões de identificação digital. O padrão foi estabelecido pela ISO (International Organization for Standardization) e conta com quatorze diferentes tópicos, relacionados às distintas características de um cartão, desde princípios físicos até formatos para as informações criptográficas. Dentre estes, o tópico 7816-4 é o de maior destaque para este artigo, o qual normatiza os comandos de comunicação com os *Smart Cards*.

Para a adição de novas funções em um *Java Card applet* dois aspectos importantes devem ser destacados: sistema de arquivos e os padrões de APDU.

2.3.1. Sistema de arquivos

São suportados dois tipos de arquivo: Dedicated File (DF) e Elementary File (EF). Todo cartão possuí um arquivo obrigatório do tipo DF nomeado MF, ou Mandatory File, sendo os demais arquivos DF opcionais. Os arquivos DF são dedicados a dados interpretados pelo cartão, enquanto os Elementary Files são usados pelos dados utilizados exclusivamente pelo mundo exterior.

2.3.2. Padrões de APDU

A ISO defini padrões para a codificação de cada um dos campos em um APDU, as combinações de byte muitas vezes são reservadas para casos recorrentes em implementações em *Smart Cards*. Para o INS, por exemplo, certos valores devem ser evitados e outros são reservados.

2.4. PKCS

O padrão de criptografia de chaves públicas foi desenvolvido em uma parceria entre a *RSA Laboratories* e desenvolvedores na área de segurança no mundo todo. Semelhante à outros padrões, como a ISO 7816, o *PKCS* é dividido em diferentes tópicos, dando-se maior importância para este trabalho os tópicos relacionados à *tokens* criptográficos: PKCS#15, que defini uma estrutura de arquivos, permitindo que usuários de *tokens* se

Value	Command name	Clause
'0E'	ERASE BINARY	6.4
'20'	VERIFY	6.12
'70'	MANAGE CHANNEL	6.16
'82'	EXTERNAL AUTHENTICATE	6.14
'84'	GET CHALLENGE	6.15
'88'	INTERNAL AUTHENTICATE	6.13
'A4'	SELECT FILE	6.11
'B0'	READ BINARY	6.1
'B2'	READ RECORD(S)	6.5
'C0'	GET RESPONSE	7.1
'C2'	ENVELOPE	7.2
'CA'	GET DATA	6.9
'D0'	WRITE BINARY	6.2
'D2'	WRITE RECORD	6.6
'D6'	UPDATE BINARY	6.3
'DA'	PUT DATA	6.10
'DC'	UPDATE DATA	6.8
'E2'	APPEND RECORD	6.7

Figura 4. Alguns padrões de INS. Fonte: [ISO/IEC 2005]

identifiquem perante aplicações cientes do padrão e PKCS#11, que define uma API (Application Programming Interface), conhecida como *Cryptoki*, para dispositivos que armazenam informações criptográficas e/ou executam funções de criptografia.

3. Adições no applet

Como proposta de solução, optou-se pela adição das funções em um *applet* de código aberto que cumprisse os requisitos: implementação do maior número de funções criptográficas, conformidades com o padrão ISO 7816, qualidade na documentação, frequência de atualizações e suporte no *middleware* OpenSC.

A aplicação escolhida, Iso Applet, conta com uma documentação detalhada, disponível no serviço de repositórios GitHub. Além disso, suporta, originalmente, a geração de chaves assimétricas, utilizando RSA e EC e possui um *driver* implementado na última versão do mediador (OpenSC 0.16).

```
1  case INS_GEN_SKEY:
2      processGenerateSKey(apdu);
3      break;
4  case INS_GET_SKEY:
5      processGetSKey(apdu);
6      break;
7  case INS_CIPH_SKEY:
8      processCiphSKey(apdu);
9      break;
10 case INS_DECIPH_SKEY:
11     processDeciphSKey(apdu);
12     break;
13 case INS_LOAD_SKEY:
14     processLoadSKey(apdu);
15     break;
```

Código 1: Novos casos no método process.

```
1  byte ALG_GEN_AES_128 = (byte) 0xA1;
2  byte ALG_GEN_AES_256 = (byte) 0xA2;
3
4  private Key[] sKeys = null;
5  private short[] currentSKeyRef = null;
6  private static short sKeySize = 16;
```

Código 2: Atributos adicionados.

O código fonte 2 mostra parte dos novos atributos necessários às funções de criptografia simétrica. A variável *sKeys* é o arranjos responsável pelo armazenamento das

chaves AES, geradas ou carregadas no cartão. Além disso, são definidos atributos de referência e tamanho das chaves, *currentSKeyRef* e *sKeySize*, respectivamente.

O método padrão *process*, um filtro das requisição feitas ao cartão, foi acrescido em cinco novas funções, que geram, extraem e carregam chaves AES, além de cifrarem e decifrarem informações com as mesmas. Todas as funções podem ser implementadas utilizando duas interfaces e duas classes disponíveis no *java card framework*: Interface ISO7816, Interface AESKey, Class Cipher e Class KeyBuilder.

Em todos os casos é necessária a execução de uma função preliminar de configuração, definida na ISO 7816 como *processManageSecurityEnviroment*. Ela defini, entre outras coisas, qual a posição no arranjo de chaves será escrito, atualizado ou utilizado nas funções subsequentes, ou ainda, qual é o tamanho da chave AES em questão (128 ou 256 bits).

O método de geração de chaves, exemplifica a estrutura das funções de propósito específico em um *java card applet*. As funções em geral, recebem o valor do APDU, realizam os passos de computação, preparam os dados de resposta e em seguida os enviam.

```
if( ! pin.isValidated() )
 ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
```

Código 3: Verificação do PIN.

Dada a sensibilidade da função, primeiro faz-se uma verificação do PIN (Personal Indentification Number). Case o usuário falhe em se autenticar, a execução é finalizada em uma mensagem de erro.

```
short sKeyRef = currentSKeyRef[0];
RandomData randomData = RandomData.getInstance(RandomData.ALG_PSEUDO_RANDOM);
sKey_array = JCSystem.makeTransientByteArray(sKey_size, JCSystem.CLEAR_ON_RESET);
randomData.generateData(sKey_array, (short)0, (short)sKey_array.length);
```

Código 4: Geração do número pseudo-aleatório.

A posição no arranjo de chaves, definida em um passo anterior, é salva em uma variável auxiliar local e o valor pseudo-aleatório que compõem a chave é gerado.

```
if(sKeys[sKeyRef] != null)
  sKeys[sKeyRef].clearKey();
```

Código 5: Verificação da posição no arranjo de chaves.

Caso haja uma chave na posição selecionada, faz-se uma limpeza preliminar, para que em seguida seja criado, dependendo do atributo *currentAlgorithmRef*, uma chave AES não inicializada.

Finalmente o valor aleatório gerado anteriormente é adicionado à estrutura básica e a chave resultante é salva no arranjo de chaves secretas. Não há retorno nesta função, a não ser as *response* APDUs, que expõem o resultado positivo ou não da execução.

As demais operações são implementadas de maneira semelhante. As funções que cifram de decifram os dados, retornam ao usuário, além do *status* da execução, os dados

```
case ALG_GEN_AES_128:
  try{
    sKey = (AESKey)KeyBuilder.buildKey(
    KeyBuilder.TYPE_AES,
    KeyBuilder.LENGTH_AES_128, false);
  }catch(CryptoException e) {
    if(e.getReason() ==
      CryptoException.NO_SUCH_ALGORITHM) {
      ISOException.throwIt(
      ISO7816.SW_FUNC_NOT_SUPPORTED);
    }
  ISOException.throwIt(
  ISO7816.SW_UNKNOWN);
  }
  break;
```

Código 6: AES 128 bits.

```
case ALG_GEN_AES_256:
  try{
    sKey = (AESKey)KeyBuilder.buildKey(
      KeyBuilder.TYPE_AES,
      KeyBuilder.LENGTH_AES_256, false);
  }catch(CryptoException e) {
     if(e.getReason() ==
       CryptoException.NO_SUCH_ALGORITHM) {
       ISOException.throwIt(
         ISO7816.SW_FUNC_NOT_SUPPORTED);
     }
      ISOException.throwIt(
      ISO7816.SW_UNKNOWN);
  }
break;
```

Código 7: AES 256 bits.

```
sKey.setKey(sKey_array, (short)0);
sKeys[sKeyRef] = sKey;
```

Código 8: Armazenamento da chave gerada.

requeridos. Os valores de INS escolhidos para as funções, seguem as recomendações da norma ISO 7816, evitando valores impares, *bytes* reservados e variações dos valores de resposta *0x9X* e *0x6X*.

Tabela 1. Valores de INS.

30	Geração da chave
32	Extração da chave
34	Cifragem de dados
36	Decifragem de dados
38	Carregamento de chave

4. Suporte via OpenSC

Pode-se dizer que o *middleware* OpenSC é implementado de forma bastante modular. Nele, como pode ser verificado na figura 5, são oferecidos uma série de utilitários, que se comunicam com a aplicação de forma direta ou através do módulo PKCS#11 e da biblioteca PKCS#15.

No módulo estão implementadas as assinaturas das funções normatizadas pelo PKCS. Parte delas não é suportada pela biblioteca, como por exemplo a geração de chaves simétricas. A solução proposta traça um fluxo de implementação, com base na função de geração de chaves RSA, oferecida na versão original da aplicação.

Primeiro, a requisição é tratada no utilitário *pkcs11_tool*, atributos de chave como *flags* de acesso, tamanho, identificador e *label* são analisados e salvos em estruturas auxiliares. Na classe *pkcs11-object*, onde a interface PKCS#11 é implementada e as alterações propostas se iniciam, confere-se, através de canais de comunicação, o suporte da funções por parte do *applet*, para que em seguida seja criado um objeto PKCS#11.

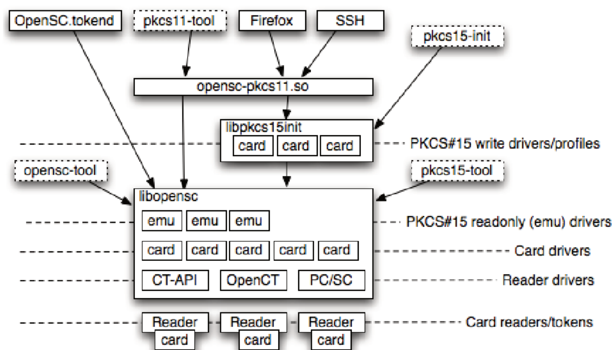

Figura 5. Arquitetura do OpenSC. Fonte: [OpenSC 2017]

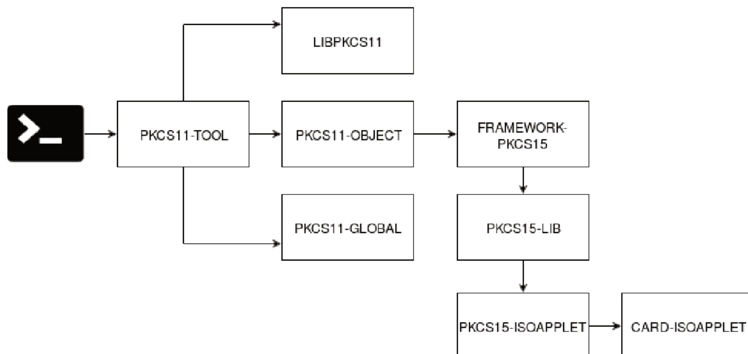

Figura 6. luxograma simplificado da geração de chaves simétricas.

O *framework* converte o objeto criado anteriormente em uma estrutura no padrão PKCS#15 e culmina na chamada da biblioteca PKCS. Os três últimos passos são os reesposáveis pela seleção do tipo de arquivo a ser gerado, neste caso SKDF, e pelo envio dos APDUs para o cartão.

5. Resultados

Os principais resultados alcançados neste trabalho estão descritos nas tabela 6 e 7. Um *applet* de código aberto capaz de realizar funções de criptografia simétrica e assimétrica, seguindo as recomendações da norma ISO 7816. E uma versão do *middleware* OpenSC, que abstrai a geração de chaves AES em *Java Cards*.

5.1. Testes com os protótipos

Uma vez que o *applet* está instalado no cartão, cria-se uma estrutura PKCS#15, responsável pelo armazenamento dos objetos de forma padronizada. Para tanto, pode-se exe-

Tabela 2. Protótipo Iso Applet

	ISO
Adições	gen_skey get_skey ciph_skey deciph_skey load_skey
Norma	ISO 7816

Tabela 3. Protótipo OpenSC

	OPENSC
Adição	C_GenerateKey
Normas	PKCS#11 e PKCS#15

cutar a opção "$-C$"do utilitário *pkcs15-init* da biblioteca OpenSC. O comando também armazena um PIN associado à estrutura, requisitado ao usuário.

Feito isto, o cartão está pronto para o armazenamento de dados, chaves ou certificados. As figuras 7 e 8 demonstram como chaves AES (128 e 256 bits) podem ser geradas com as adições propostas na sessão 4. Nas linhas de comando deve-se indicar o módulo PKCS#11 a ser utilizado, neste caso *opensc-pkcs11.so*, um rótulo para a chave, o tamanho em bytes e um identificador numérico.

- 128 bits:
  pkcs11-tool –module /usr/local/lib/pkcs11/opensc-pkcs11.so -l –pin 123456 –keygen –label AES128 –key-type "aes:16--id 01

```
Secret AES Key [AES128]
        Object Flags   : [0x2], modifiable
        Usage          : [0x33], encrypt, decrypt, wrap, unwrap
        Access Flags   : [0x11], sensitive, local
        Size           : 128 bits
        ID             : 01
        Native         : no
        Key ref        : 0 (0x0)
        Path           : 3f0050153600
```

Figura 7. Memória do *Java Card* após a geração de uma chave AES 128 bits.

- 256 bits:
  pkcs11-tool –module /usr/local/lib/pkcs11/opensc-pkcs11.so -l –pin 123456 –keygen –label AES256 –key-type "aes:32--id 02

```
Secret AES Key [AES256]
        Object Flags   : [0x2], modifiable
        Usage          : [0x33], encrypt, decrypt, wrap, unwrap
        Access Flags   : [0x11], sensitive, local
        Size           : 256 bits
        ID             : 02
        Native         : no
        Key ref        : 1 (0x1)
        Path           : 3f0050153601
```

Figura 8. Memória do *Java Card* após a geração de uma chave AES 256 bits.

O apêndice A1 demonstra como estas e outras funções podem ser executadas sem o uso de uma camada de abstração na comunicação com o *Java Card*, ou seja, transmitindo diretamente as sequências de APDU necessárias ao *applet*.

6. Conclusões

Este artigo sugeriu uma serie de adições e modificação, sobre um *applet* de código aberto e a biblioteca OpenSC. Buscando uma solução, para a escassez de aplicações de código aberto com suporte à criptografia simétrica. Para tanto foram sugeridas as alterações descritas nas sessões 3 e 4.

Embora os principais objetivos tenham sido alcançados no desenrolar do desenvolvimento, vale ressaltar alguns pontos de fragilidade e limitações das soluções apresentadas. Na geração de chaves AES no *applet*, não são oferecidos todos os tamanhos de chave descritos pelo padrão, a ser adicionado o tamanho 192 bits. Além disso, as funções de cifragem e decifragem poderiam ter sido englobadas como um novo caso (um caso de chaves simétricas) nos métodos já existentes na aplicação, ao invés de possuírem assinaturas próprias.

Pôde-se observar que os materiais relacionados a *Smart Cards*, desenvolvimento de *Java Card applets* e sobre os padrões ISO 7816 e PKCS, são bastante limitados e muitas vezes confusos. Para o entendimento do funcionamento básico dos cartões vale a leitura dos trabalhos de mestrado e graduação citados aqui como referência. Já o desenvolvimento das aplicações é mais facilmente compreendido, com uma analise cuidadosa de casos conhecidos, como o próprio Iso Applet ou Muscle Applet. No caso dos padrões mencionados, as descrições do PKCS podem ser encontradas facilmente nos *web* sites da *RSA Laboratories*, o que não acontece para as normas ISO. Os diferentes tópicos da ISO 7816 são patenteados, com poucas acessões que podem ser encontradas fragmentadas em sites especializados.

Referências

A.E.T (2016). Digital trust & security solutions by aet europe.

GlobalPlatform (2016). The standard for secure digital services and devices.

Gringeri, E. (2008-2009). *Improving security and memory management in the muscle card framework.* PhD thesis, Universitá degli studi di Pisa, Pisa.

ISO/IEC (2005). *International Standart ISO/IEC 7816-4.* International Organization for Standardization, Bedford, EUA.

OpenSC (2017). Overview opensc.

RSA, L. (2009). *PKCS#11 Base Functionality v2.30: Criptoki - Draft 4.* Laboratories RSA, Bedford, EUA.

Stallings, W. (2014). *Cryptography and Network Security.* Ed. Pearson, New Jersey, EUA.

Apêndice A1: Comunicação com o applet via APDU

Sabe-se que a comunicação entre os *Java Cards* e as aplicações externas se da pela troca de APDU's. Nesta sessão, pode-se verificar a execução das funções da tabela 1, utilizando o comando *-s* do utilitário *opensc-tool*.

- *Generate Secret Key*:
  Para a geração de uma chave AES de 256 bits, por exemplo, são necessários os seguintes comandos realizados em sequências:

```
1 opensc-tool -s 00:A4:08:00:04:50:15:44:01:00
2 -s 00:B0:00:00:00
3 -s 00:20:00:01:10:31:32:33:34:35:36:00:00:00:00:00:00:00:00:00:00
4 -s 00:A4:08:0C:02:50:15
5 -s 00:22:42:01:06:80:01:A2:84:01:00
6 -s 00:30:00:00
```

  Os quatro primeiros APDU's são responsáveis pela seleção, leitura e verificação do PIN, que neste exemplo é "123456". Seguidos pelo comando *Manage Security Enviroment*, que configura 256 bits para o tamanho da chave e zero para a posição no arranjo de chaves secretas. Finalmente a última combinação de hexadecimais caracteriza o comando de geração propriamente dito, com o valor de INS igual a 30.
  A mesma ideia se aplica a chaves 128 bits, com uma sutil diferença no comando de configuração. O hexadecimal A2 passa a ser A1.

```
1 opensc-tool -s 00:A4:08:00:04:50:15:44:01:00
2  -s 00:B0:00:00:00
3  -s 00:20:00:01:10:31:32:33:34:35:36:00:00:00:00:00:00:00:00:00:00
4  -s 00:A4:08:0C:02:50:15
5  -s 00:22:42:01:06:80:01:A1:84:01:00
6  -s 00:30:00:00
```

- *Get Secret Key*:
  Uma vez gerada ou carregada uma chave AES no Iso Applet, pode-se extraí-la com a função *processGetSKey*. Semelhante ao comando de geração de chaves, os primeiros APDU's realizam a verificação do PIN, seguidos pela configuração da referência da chave e do comando de extração propriamente dito:

```
1 opensc-tool -s 00:A4:08:00:04:50:15:44:01:00
2 -s 00:B0:00:00:00
3 -s 00:20:00:01:10:31:32:33:34:35:36:00:00:00:00:00:00:00:00:00:00
4 -s 00:A4:08:0C:02:50:15
5 -s 00:22:42:02:03:84:01:00
6 -s 00:32:00:00
```

- *Ciph*:
  Para a cifragem de dados, utilizando qualquer um dos dois tamanhos possíveis de chaves AES necessita-se de: comandos para a autenticação do PIN, configuração da chave a ser utilizada na operação e envio do dado a ser cifrado. O *plaintext* pode assumir valores de até 128 bits, uma vez que o cifrador AES funciona em blocos deste tamanho.

```
1 opensc-tool -s 00:A4:08:00:04:50:15:44:01:00
2 -s 00:B0:00:00:00
```

```
-s 00:20:00:01:10:31:32:33:34:35:36:00:00:00:00:00:00:00:00:00:00
-s 00:A4:08:0C:02:50:15
-s 00:22:42:02:03:84:01:00
-s 00:34:00:00:08:70:69:7A:7A:61:73:65:63
```

No último comando pode-se notar o valor de INS 34, que representa o comando de cifragem, seguido por dois campos 00 e pelo campo LC, que indica o tamanho do dado a ser cifrado. Neste exemplo oito bytes são enviados e representam o texto ”pizzasec”.

- *Deciph*:
  Pode-se decifrar o item anterior, utilizando-se a mesmo valor de chave, porém com diferentes comandos:

```
opensc-tool -s 00:A4:08:00:04:50:15:44:01:00
-s 00:B0:00:00:00
-s 00:20:00:01:10:31:32:33:34:35:36:00:00:00:00:00:00:00:00:00:00
-s 00:A4:08:0C:02:50:15
-s 00:22:42:02:03:84:01:00
-s 00:36:00:00:10:F2:5B:8F:A1:9D:1E:B9:A0:61:BA:42:E6:A2:4A:4A:13
```

- *Load*:
  Finalmente, para *upload* de chaves secretas basta que seja executado uma sequência semelhante à:

```
opensc-tool -s 00:A4:08:00:04:50:15:44:01:00
-s 00:B0:00:00:00
-s 00:20:00:01:10:31:32:33:34:35:36:00:00:00:00:00:00:00:00:00:00
-s 00:A4:08:0C:02:50:15
-s 00:22:42:02:06:80:01:A1:84:01:01
-s 00:38:00:00:10:59:7C:5A:D0:46:28:69:CD:14:2D:75:7C:EF:7A:AA:E1
```

A chave acima, de 128 bits, é armazenada na posição ”01”do *array* de chaves secretas, seu valor está representado no último comando, após o quinto *byte*.

ANEXO A – Códigos Iso Applet

A.1 ISO APPLET

Alterações realizadas no arquivo "IsoApplet.java".

```
[...]

import javacard.security.AESKey;

[...]

// Secret key INS
public static final byte INS_GEN_SKEY = (byte) 0x30;
public static final byte INS_GET_SKEY = (byte) 0x32;
public static final byte INS_CIPH_SKEY = (byte) 0x34;
public static final byte INS_DECIPH_SKEY = (byte) 0x36;
public static final byte INS_LOAD_SKEY = (byte) 0x38;

[...]

// Secret key size
private static final byte ALG_GEN_AES_128 = (byte) 0xA1;
private static final byte ALG_GEN_AES_256 = (byte) 0xA2;

[..]

// Secret Key
private static byte[] sKey_array = null;
private static AESKey sKey = null;
private static byte[] encrypted = null;
private static byte[] holder = null;
private static short sKey_size = 16; // default size
private static short size = 0;
private Key[] sKeys = null;
private short[] currentSKeyRef = null;

[...]

protected IsoApplet() {

    [...]

    currentSKeyRef = JCSystem.makeTransientShortArray((short)1,
        JCSystem.CLEAR_ON_DESELECT);
    sKeys = new Key[KEY_MAX_COUNT];
```

```
40
41      [...]
42  }
43
44  [...]
45
46  @Override
47      public void process(APDU apdu) {
48
49      [...]
50
51      case INS_GEN_SKEY:
52                  processGenerateSKey(apdu);
53                  break;
54              case INS_GET_SKEY:
55                  processGetSKey(apdu);
56                  break;
57              case INS_CIPH_SKEY:
58                  processCiphSKey(apdu);
59                  break;
60              case INS_DECIPH_SKEY:
61                  processDeciphSKey(apdu);
62                  break;
63              case INS_LOAD_SKEY:
64                  processLoadSKey(apdu);
65                  break;
66      [...]
67
68  }
69
70  [...]
71
72  /**
73       * \brief Process the CIPH SECRET KEY apdu (INS = 34).
74       *
75       * This apdu is used to ciph a data with a secret key.
76       *
77       * \param apdu The apdu.
78       */
79      public void processCiphSKey(APDU apdu) {
80        if( ! pin.isValidated() ) {
81            ISOException.throwIt(
82            ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
83        }
84
```

```
85        byte [] buffer = apdu.getBuffer();
86        short bytesReadCount = apdu.setIncomingAndReceive();
87        short echoOffset = (short)0;
88
89        short sKeyRef = currentSKeyRef[0];
90        AESKey key = (AESKey) sKeys[sKeyRef];
91
92        holder = new byte[16];
93
94        while ( bytesReadCount > 0 ) {
95          echoOffset = Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA,
96          holder, echoOffset, bytesReadCount);
97          bytesReadCount = apdu.receiveBytes(ISO7816.OFFSET_CDATA);
98        }
99
100       encrypted = cipherS(holder, key);
101
102       short le = apdu.setOutgoing();
103       short totalBytes = (short) encrypted.length;
104       Util.arrayCopyNonAtomic(encrypted, (short)0, buffer,
              (short)0, totalBytes);
105       apdu.setOutgoingLength(totalBytes);
106       apdu.sendBytes((short) 0, (short) encrypted.length);
107     }
108
109     public byte[] cipherS(byte [] data, AESKey key){
110       Cipher cipher =
              Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
              false);
111
112       try{
113         cipher.init(key, Cipher.MODE_ENCRYPT);
114       }catch (CryptoException e) {
115         if(e.getReason() == CryptoException.ILLEGAL_VALUE ||
                e.getReason() == CryptoException.UNINITIALIZED_KEY){
116           ISOException.throwIt(
117           ISO7816.SW_CONDITIONS_NOT_SATISFIED);
118         }
119         ISOException.throwIt(ISO7816.SW_UNKNOWN);
120       }
121
122       byte [] result = new byte[data.length];
123       short encLength = cipher.doFinal(data, (short)0,
              (short)data.length, result, (short)0);
124       return result;
```

```
125     }
126
127 /**
128      * \brief Process the DECIPH SECRET KEY apdu (INS = 36).
129      *
130      * This apdu is used to deciph a data with a secret key.
131      *
132      * \param apdu The apdu.
133      */
134     public void processDeciphSKey(APDU apdu) {
135       if( ! pin.isValidated() ) {
136           ISOException.throwIt(
137           ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
138       }
139
140       byte [] buffer = apdu.getBuffer();
141       short bytesReadCount = apdu.setIncomingAndReceive();
142       short echoOffset = (short)0;
143
144       short sKeyRef = currentSKeyRef[0];
145       AESKey key = (AESKey) sKeys[sKeyRef];
146
147       holder = new byte[16];
148
149       while ( bytesReadCount > 0 ) {
150         echoOffset = Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA,
151         holder, echoOffset, bytesReadCount);
152         bytesReadCount = apdu.receiveBytes(ISO7816.OFFSET_CDATA);
153       }
154
155       byte [] decrypted = decipherS(holder, key);
156
157       short le = apdu.setOutgoing();
158       short totalBytes = (short) decrypted.length;
159
160       Util.arrayCopyNonAtomic(decrypted, (short)0, buffer,
              (short)0, totalBytes);
161       apdu.setOutgoingLength(totalBytes);
162       apdu.sendBytes((short) 0, (short) decrypted.length);
163     }
164
165     static byte[] decipherS(byte[] data, AESKey key) {
166       Cipher decipher =
              Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD,
              false);
```

```
167
168      try{
169        decipher.init(key, Cipher.MODE_DECRYPT);
170      }catch (CryptoException e) {
171        if(e.getReason() == CryptoException.ILLEGAL_VALUE ||
               e.getReason() == CryptoException.UNINITIALIZED_KEY){
172          ISOException.throwIt(
173          ISO7816.SW_CONDITIONS_NOT_SATISFIED);
174        }
175        ISOException.throwIt(ISO7816.SW_UNKNOWN);
176      };
177
178      byte [] result = new byte [data.length];
179      short encLength = decipher.doFinal(data, (short)0,
             (short)data.length, result, (short)0);
180      return result;
181    }
182
183    /**
184     * \brief Process the GET SECRET KEY apdu (INS = 32).
185     *
186     * This apdu is used to get the secret key bytes.
187     *
188     * \param apdu The apdu.
189     */
190    public void processGetSKey(APDU apdu){
191        if( ! pin.isValidated() ) {
192          ISOException.throwIt(
193          ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
194        }
195
196      byte [] buffer = apdu.getBuffer();
197        short sKeyRef = currentSKeyRef[0];
198        AESKey key = (AESKey) sKeys[sKeyRef];
199      byte [] sendKey = new byte[key.getSize()];
200      key.getKey(sendKey, (short)0);
201
202      short le = apdu.setOutgoing();
203      short totalBytes = (short) sendKey.length;
204      Util.arrayCopyNonAtomic(sendKey, (short)0, buffer,
             (short)0, totalBytes);
205       apdu.setOutgoingLength(totalBytes);
206      apdu.sendBytes((short) 0, (short)
             ((sendKey.length)/(short)8));
207    }
```

```
208
209     /**
210      * \brief Process the GENERATE SECRET KEY apdu (INS = 30).
211      *
212      * This apdu is used to generate a secret key.
213      *
214      * \param apdu The apdu.
215      */
216     public void processGenerateSKey(APDU apdu){
217       if( ! pin.isValidated() ) {
218           ISOException.throwIt(
219           ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
220       }
221
222       short sKeyRef = currentSKeyRef[0];
223       RandomData randomData =
              RandomData.getInstance(RandomData.ALG_PSEUDO_RANDOM);
224       sKey_array = JCSystem.makeTransientByteArray(sKey_size,
              JCSystem.CLEAR_ON_RESET);
225       randomData.generateData(sKey_array, (short)0,
              (short)sKey_array.length);
226
227       if(sKeys[sKeyRef] != null) {
228         sKeys[sKeyRef].clearKey();
229       }
230
231       switch(currentAlgorithmRef[0]){
232         case ALG_GEN_AES_128:
233           try{
234             sKey =
                    (AESKey)KeyBuilder.buildKey(KeyBuilder.TYPE_AES,
                    KeyBuilder.LENGTH_AES_128, false);
235
236           }catch(CryptoException e) {
237               if(e.getReason() ==
                      CryptoException.NO_SUCH_ALGORITHM) {
238                   ISOException.throwIt(
239                   ISO7816.SW_FUNC_NOT_SUPPORTED);
240               }
241               ISOException.throwIt(ISO7816.SW_UNKNOWN);
242           }
243           break;
244         case ALG_GEN_AES_256:
245           try{
246             sKey =
```

```
                (AESKey)KeyBuilder.buildKey(KeyBuilder.TYPE_AES,
                KeyBuilder.LENGTH_AES_256, false);
247          }catch(CryptoException e) {
248              if(e.getReason() ==
                     CryptoException.NO_SUCH_ALGORITHM) {
249                  ISOException.throwIt(
250                  ISO7816.SW_FUNC_NOT_SUPPORTED);
251              }
252              ISOException.throwIt(ISO7816.SW_UNKNOWN);
253          }
254          break;
255        default:
256          ISOException.throwIt(
257          ISO7816.SW_CONDITIONS_NOT_SATISFIED);
258          break;
259      }
260
261      sKey.setKey(sKey_array, (short)0);
262      sKeys[sKeyRef] = sKey;
263    }
264
265    /**
266     * \brief Process the LOAD SECRET KEY apdu (INS = 38).
267     *
268     * This apdu is used to load a secret key.
269     *
270     * \param apdu The apdu.
271     */
272    public void processLoadSKey(APDU apdu){
273      if( ! pin.isValidated() ) {
274          ISOException.throwIt(
275          ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
276      }
277
278      byte[] buffer = apdu.getBuffer();
279      byte algRef = currentAlgorithmRef[0];
280      short sKeyRef = currentSKeyRef[0];
281      int size = 0;
282      AESKey key = null;
283
284      switch(algRef){
285            case ALG_GEN_AES_128:
286              size = 16;
287              key = (AESKey)
                   KeyBuilder.buildKey(KeyBuilder.TYPE_AES,
```

```
                    KeyBuilder.LENGTH_AES_128, false);
288               break;
289             case ALG_GEN_AES_256:
290               size = 32;
291               key = (AESKey)
                    KeyBuilder.buildKey(KeyBuilder.TYPE_AES,
                    KeyBuilder.LENGTH_AES_256, false);
292               break;
293             default:
294               ISOException.throwIt(
295               ISO7816.SW_FUNC_NOT_SUPPORTED);
296       }
297
298       if(sKeys[sKeyRef] != null) {
299                 sKeys[sKeyRef].clearKey();
300       }
301
302       byte[] local_data = new byte[size];
303       Util.arrayCopy(buffer,ISO7816.OFFSET_CDATA, local_data,
               (short)0, (short) size);
304
305       key.setKey(local_data, (short)0);
306       sKeys[sKeyRef] = key;
307     }
308
309 [...]
310
311 public void processManageSecurityEnvironment(APDU apdu) throws
         ISOException {
312
313     [...]
314
315     case (byte) 0x42:
316             try {
317                 pos = UtilTLV.findTag(buf, offset_cdata, (byte)
                        lc, (byte) 0x80);
318             } catch (NotFoundException e) {
319                 ISOException.throwIt(ISO7816.SW_DATA_INVALID);
320             } catch (InvalidArgumentsException e) {
321                 ISOException.throwIt(ISO7816.SW_DATA_INVALID);
322             }
323             if(buf[++pos] != (byte) 0x01) { // Length must be 1.
324                 ISOException.throwIt(ISO7816.SW_DATA_INVALID);
325             }
326             // Set the current algorithm reference.
```

```
327             algRef = buf[++pos];
328 
329             // Secret key reference (Index in keys[]-array).
330             try {
331                 pos = UtilTLV.findTag(buf, offset_cdata, (byte)
                        lc, (byte) 0x84);
332             } catch (NotFoundException e) {
333                 ISOException.throwIt(ISO7816.SW_DATA_INVALID);
334             } catch (InvalidArgumentsException e) {
335                 ISOException.throwIt(ISO7816.SW_DATA_INVALID);
336             }
337             if(buf[++pos] != (byte) 0x01 // Length: must be 1 -
                    only one key reference (byte) provided.
338                     || buf[++pos] >= KEY_MAX_COUNT) { // Value:
                        KEY_MAX_COUNT may not be exceeded. Valid
                        key references are from 0..KEY_MAX_COUNT.
339                 ISOException.throwIt(ISO7816.SW_DATA_INVALID);
340             }
341             sKeyRef = buf[pos];
342             break;
343 
344     [...]
345 
346     case (byte) 0x01:
347         if(algRef != ALG_GEN_AES_128 && algRef !=
                ALG_GEN_AES_256) {
348             ISOException.throwIt(ISO7816.SW_FUNC_NOT_SUPPORTED);
349         }
350         switch(algRef){
351           case ALG_GEN_AES_128:
352             sKey_size = 16;
353             break;
354           case ALG_GEN_AES_256:
355             sKey_size = 32;
356             break;
357           default:
358             ISOException.throwIt(ISO7816.SW_FUNC_NOT_SUPPORTED);
359         }
360         break;
361       case (byte) 0x02:
362           if(sKeyRef == -1) {
363               ISOException.throwIt(ISO7816.SW_DATA_INVALID);
364           }
365           break;
366       case (byte) 0x03:
```

```
367             if(sKeyRef == -1) {
368                 ISOException.throwIt(ISO7816.SW_DATA_INVALID);
369             }
370             if(algRef == ALG_GEN_AES_128
371                 && algRef == ALG_GEN_AES_256) {
372                 ISOException.throwIt(
373                 ISO7816.SW_FUNC_NOT_SUPPORTED);
374             }
375             break;
376 
377         [...]
378 
379     }
380 
381     [...]
382 
383 }
```

ANEXO B – Códigos OpenSC

B.1 OPENSC

Alterações realizadas no arquivo "pkcs11-object.c".

```
 [...]

CK_RV C_GenerateKey(CK_SESSION_HANDLE hSession,
            CK_MECHANISM_PTR pMechanism,
            CK_ATTRIBUTE_PTR pTemplate,
            CK_ULONG ulCount,
            CK_OBJECT_HANDLE_PTR phKey)
{
        CK_RV rv;
        struct sc_pkcs11_session *session;
        struct sc_pkcs11_slot *slot;

        if (pMechanism == NULL_PTR || (pTemplate == NULL_PTR &&
                ulCount > 0))
            return CKR_ARGUMENTS_BAD;

        rv = sc_pkcs11_lock();
        if (rv != CKR_OK)
            return rv;

        dump_template(SC_LOG_DEBUG_NORMAL, "C_GenerateKey(),
                Secret Key attrs", pTemplate, ulCount);

        rv = get_session(hSession, &session);
        if (rv != CKR_OK)
            goto out;

        if (!(session->flags & CKF_RW_SESSION)) {
            rv = CKR_SESSION_READ_ONLY;
            goto out;
        }

        slot = session->slot;
        if (slot->p11card->framework->gen_key == NULL)
            rv = CKR_FUNCTION_NOT_SUPPORTED;
        else {
            rv = restore_login_state(slot);
            if (rv == CKR_OK)
                rv = slot->p11card->framework->gen_key(slot, pMechanism,
                        pTemplate, ulCount, phKey);
                rv = reset_login_state(session->slot, rv);
        }

out:
    sc_pkcs11_unlock();
    return rv;
}
```

```
[...]
```

Alterações realizadas no arquivo "pkcs15-lib.c".

```
/*
 * Generate a new symmetric-key
 */
int sc_pkcs15init_generate_symmetric(struct sc_pkcs15_card *p15card,
        struct sc_profile *profile,
        struct sc_pkcs15init_skeyargs *keygen_args, unsigned int keybits,
        struct sc_pkcs15_object **res_obj)
{

    struct sc_context *ctx = p15card->card->ctx;
    struct sc_pkcs15_object *object = NULL;
    struct sc_pkcs15_skey_info *key_info = NULL;
    int r;

    printf("### BEGIN PKCS15-LIB GEN SYMMETRIC\n");

    LOG_FUNC_CALLED(ctx);

    if (profile->ops->generate_symmetric == NULL)
        LOG_TEST_RET(ctx, SC_ERROR_NOT_SUPPORTED,
                "Secret Key generation not supported");

    if (keygen_args->id.len) {
        /* Make sure that secret key's ID is the unique inside the PKCS#15
             application */
        r = sc_pkcs15_find_skey_by_id(p15card, &keygen_args->id, NULL);
        if (!r)
            LOG_TEST_RET(ctx, SC_ERROR_NON_UNIQUE_ID,
                    "Non unique ID of the secret key object");
        else if (r != SC_ERROR_OBJECT_NOT_FOUND)
            LOG_TEST_RET(ctx, r, "Find secret key error");
    }

    /* Set up the SkKDF object */
    r = sc_pkcs15init_init_skdf(p15card, profile, keygen_args, keybits, &object);
    LOG_TEST_RET(ctx, r, "Set up secret key object error");

    key_info = (struct sc_pkcs15_skey_info *) object->data;

    /* Generate the secret key on card */
    r = profile->ops->create_key(profile, p15card, object);
    LOG_TEST_RET(ctx, r, "Cannot generate key: create key failed");

    r = profile->ops->generate_symmetric(profile, p15card, object);
    LOG_TEST_RET(ctx, r, "Failed to generate secret key");

```

```
46      r = sc_pkcs15init_add_object(p15card, profile, SC_PKCS15_SKDF,
            object);
47      LOG_TEST_RET(ctx, r, "Failed to add generated secret key object");
48
49      if (!r && profile->ops->emu_store_data) {
50          r = profile->ops->emu_store_data(p15card, profile, object, NULL,
                NULL);
51          if (r == SC_ERROR_NOT_IMPLEMENTED)
52              r = SC_SUCCESS;
53          LOG_TEST_RET(ctx, r, "Card specific 'store data' failed");
54      }
55
56      if (res_obj)
57          *res_obj = object;
58
59      profile->dirty = 1;
60
61      printf("### BEGIN PKCS15-LIB GEN SYMMETRIC\n");
62      LOG_FUNC_RETURN(ctx, r);
63  }
```

Alterações realizadas no arquivo "pkcs15-isoApplet.c".

```
1   [...]
2   static int isoApplet_generate_symmetric(sc_profile_t *profile,
        sc_pkcs15_card_t *p15card,
3       sc_pkcs15_object_t *obj)
4   {
5       int r;
6       sc_pkcs15_skey_info_t *key_info = (sc_pkcs15_skey_info_t *)
            obj->data;
7       sc_file_t *sKeyFile = NULL;
8       sc_card_t *card = p15card->card;
9
10      printf("### BEGIN PKCS15-ISOAPPLET\n");
11
12      LOG_FUNC_CALLED(card->ctx);
13
14      /* Authentication stuff. */
15      r = sc_profile_get_file_by_path(profile, &key_info->path, &sKeyFile);
16      if (!sKeyFile)
17      {
18          SC_FUNC_RETURN(card->ctx, SC_LOG_DEBUG_VERBOSE,
                SC_ERROR_NOT_SUPPORTED);
19      }
20      sc_file_free(sKeyFile);
21
22      /* Generate the key. */
23      switch(obj->type)
24      {
25      case SC_PKCS15_TYPE_SKEY_AES:
```

```
26          r = isoApplet_generate_key_aes(key_info, card);
27          break;
28
29      default:
30          r = SC_ERROR_NOT_SUPPORTED;
31          sc_log(card->ctx, "%s: Secret Key generation failed:
                Unknown/unsupported key type.", strerror(r));
32      }
33
34      printf("### END PKCS15-ISOAPPLET\n");
35      LOG_FUNC_RETURN(card->ctx, r);
36  }
37
38   [...]
39
40  static int
41  isoApplet_generate_key_aes(sc_pkcs15_skey_info_t *key_info, sc_card_t
        *card)
42  {
43      int rv;
44      size_t keybits;
45      struct sc_cardctl_isoApplet_gen_symmetric args;
46
47      LOG_FUNC_CALLED(card->ctx);
48
49      /* Check key size: */
50      keybits = key_info->value_len;
51      if (keybits != 256 && keybits != 128)
52      {
53          rv = SC_ERROR_INVALID_ARGUMENTS;
54          sc_log(card->ctx, "%s: AES secret key length is unsupported, correct
                length is 128 or 256", sc_strerror(rv));
55          goto err;
56      }
57
58      /* Generate the key.
59       * Note: key size is not explicitly passed to the card.
60       * It assumes 32 along with the algorithm reference. */
61      memset(&args, 0, sizeof(args));
62      switch(keybits){
63          case 128:
64              args.algorithm_ref = SC_ISOAPPLET_ALG_REF_AES_128;
65              break;
66          case 256:
67              args.algorithm_ref = SC_ISOAPPLET_ALG_REF_AES_256;
68              break;
69      default:
70          args.algorithm_ref = SC_ISOAPPLET_ALG_REF_AES_128;
71      }
72      args.s_key_ref = key_info->key_reference;
73
74      rv = sc_card_ctl(card,
```

```
        SC_CARDCTL_ISOAPPLET_GENERATE_SYMMETRIC, &args);
75    if (rv < 0)
76    {
77        sc_log(card->ctx, "%s: Error in card_ctl", sc_strerror(rv));
78        goto err;
79    }
80
81 err:
82    LOG_FUNC_RETURN(card->ctx, rv);
83 }
84
85  [...]
```

Alterações realizadas no arquivo "pkcs15-init.h".

```
1  [...]
2 #define DEFAULT_SECRET_KEY_LABEL "Secret Key"
3  [...]
4 int (*generate_symmetric)(struct sc_profile *, struct sc_pkcs15_card *,
       sc_pkcs15_object_t *obj);
5  [...]
6 extern int sc_pkcs15init_generate_symmetric();
7  [...]
```

Alterações realizadas no arquivo "framework-pkcs15.c".

```
1  static CK_RV
2 pkcs15_gen_key(struct sc_pkcs11_slot *slot,CK_MECHANISM_PTR
       pMechanism,
3          CK_ATTRIBUTE_PTR pTemplate, CK_ULONG ulCount,
4          CK_OBJECT_HANDLE_PTR phKey)
5 {
6
7     struct sc_profile *profile = NULL;
8     struct sc_pkcs11_card *p11card = slot->p11card;
9     struct sc_pkcs15_auth_info *pin = NULL;
10    struct sc_aid *aid = NULL;
11    struct pkcs15_fw_data *fw_data = NULL;
12    struct sc_pkcs15init_skeyargs keyargs;
13    struct sc_pkcs15_object *key_obj = NULL;
14    struct pkcs15_skey_object *skeyobj = NULL;
15    struct pkcs15_any_object *key_any_obj = NULL;
16    struct sc_pkcs15_id id;
17    size_t      len;
18    CK_KEY_TYPE keytype;
19    CK_ULONG key_length = 0;
20    CK_ULONG i = 0;
21    CK_ATTRIBUTE_PTR attr;
22    CK_ULONG keybits = 0;
23    char      label[SC_PKCS15_MAX_LABEL_SIZE];
24    int rc, rv = CKR_OK;
```

```
25
26
27      printf("### BEGIN FREAMEWORK-PKCS15\n");
28
29      sc_log(context, "Secret key generation, mech = 0x%0x",
            pMechanism->mechanism);
30
31      if (pMechanism->mechanism != CKM_AES_KEY_GEN
              && pMechanism->mechanism != CKM_DES_KEY_GEN
33            && pMechanism->mechanism != CKM_DES3_KEY_GEN)
34          return CKR_MECHANISM_INVALID;
35
36      fw_data = (struct pkcs15_fw_data *)
            p11card->fws_data[slot->fw_data_idx];
37      if (!fw_data)
38          return sc_to_cryptoki_error(SC_ERROR_INTERNAL,
                "C_GenerateKey");
39
40      rc = sc_lock(p11card->card);
41      if (rc < 0)
42          return sc_to_cryptoki_error(rc, "C_GenerateKey");
43
44      rc = sc_pkcs15init_bind(p11card->card, "pkcs15", NULL,
            slot->app_info, &profile);
45      if (rc < 0) {
46          sc_unlock(p11card->card);
47          return sc_to_cryptoki_error(rc, "C_GenerateKey");
48      }
49
50      if(slot->app_info)
51          aid = &slot->app_info->aid;
52
53      rc = sc_pkcs15init_finalize_profile(p11card->card, profile, aid);
54      if (rc != CKR_OK) {
55          sc_log(context, "Cannot finalize profile: %i", rc);
56          return sc_to_cryptoki_error(rc, "C_GenerateKey");
57      }
58
59      memset(&keyargs, 0, sizeof(keyargs));
60
61      /* 1. Convert the pkcs11 attributes to pkcs15init args */
62
63      keyargs.access_flags |=
64              SC_PKCS15_PRKEY_ACCESS_SENSITIVE
65            | SC_PKCS15_PRKEY_ACCESS_LOCAL;
66
67      attr = pTemplate;
68      for (i = ulCount; i > 0; i--) {
69          attr++;
70
71          switch (attr->type) {
72          case CKA_DECRYPT:
```

```
73              keyargs.usage |= pkcs15_check_bool_cka(attr,
                    SC_PKCS15_PRKEY_USAGE_DECRYPT);
74              break;
75          case CKA_ENCRYPT:
76              keyargs.usage |= pkcs15_check_bool_cka(attr,
                    SC_PKCS15_PRKEY_USAGE_ENCRYPT);
77              break;
78          case CKA_WRAP:
79              keyargs.usage |= pkcs15_check_bool_cka(attr,
                    SC_PKCS15_PRKEY_USAGE_WRAP);
80              break;
81          case CKA_UNWRAP:
82              keyargs.usage |= pkcs15_check_bool_cka(attr,
                    SC_PKCS15_PRKEY_USAGE_UNWRAP);
83              break;
84          default:
85              continue;
86          }
87      }
88
89      if ((pin = slot_data_auth_info(slot->fw_data)) != NULL)
90          keyargs.auth_id = pin->auth_id;
91
92      rv = attr_find(pTemplate, ulCount, CKA_KEY_TYPE, &keytype, NULL);
93      if (rv != CKR_OK && pMechanism->mechanism ==
            CKM_AES_KEY_GEN)
94          keytype = CKK_AES;
95      else if(rv != CKR_OK && pMechanism->mechanism ==
            CKM_DES_KEY_GEN)
96          keytype = CKK_DES;
97      else if(rv != CKR_OK && pMechanism->mechanism ==
            CKM_DES3_KEY_GEN)
98          keytype = CKK_DES3;
99      else if(rv != CKR_OK)
100         goto kpgen_done;
101
102     keyargs.type = keytype; // saving key type at new attr
103
104     if(keytype == CKK_AES){
105         keyargs.algorithm = SC_ALGORITHM_AES;
106         rv = attr_find(pTemplate, ulCount, CKA_VALUE_LEN, &key_length,
                NULL);
107         if (rv != CKR_OK)
108             keybits = 256; /* Default key size */
109         else
110             keybits = key_length * 8;
111     }
112     else if(keytype == CKK_DES)
113         keyargs.algorithm = SC_ALGORITHM_DES;
114     else if(keytype == CKK_DES3)
115         keyargs.algorithm = SC_ALGORITHM_3DES;
116     else    {
```

```
117         rv = CKR_ATTRIBUTE_VALUE_INVALID;
118         goto kpgen_done;
119     }
120 
121     id.len = SC_PKCS15_MAX_ID_SIZE;
122     rv = attr_find(pTemplate, ulCount, CKA_ID, &id.value, &id.len);
123     if (rv == CKR_OK)
124         keyargs.id = id;
125 
126     len = sizeof(label) - 1;
127     rv = attr_find(pTemplate, ulCount, CKA_LABEL, label, &len);
128     if (rv == CKR_OK) {
129             label[len] = '\0';
130             keyargs.label = label;
131     }
132 
133     /* 3.a Try on-card key generation */
134 
135     sc_pkcs15init_set_p15card(profile, fw_data->p15_card);
136 
137     sc_log(context, "Try on-card key pair generation");
138     rc = sc_pkcs15init_generate_symmetric(fw_data->p15_card, profile,
            &keyargs, keybits, &key_obj);
139     if (rc < 0) {
140         sc_log(context, "sc_pkcs15init_generate_key returned %d", rc);
141         rv = sc_to_cryptoki_error(rc, "C_GenerateKey");
142         goto kpgen_done;
143     }
144 
145    /* 4. Create new pkcs11 secret key object */
146 
147     rc = __pkcs15_create_secret_key_object(fw_data, key_obj,
            &key_any_obj);
148     if (rc != 0) {
149         sc_log(context, "__pkcs15_create_secret_key_object returned %d", rc);
150         rv = sc_to_cryptoki_error(rc, "C_GenerateKey");
151         goto kpgen_done;
152     }
153 
154     pkcs15_add_object(slot, key_any_obj, phKey);
155     skeyobj = (struct pkcs15_skey_object *) key_any_obj;
156 
157 kpgen_done:
158     sc_pkcs15init_unbind(profile);
159     sc_unlock(p11card->card);
160     printf("### END FREAMEWORK-PKCS15\n");
161     return rv;
162 }
```

Alterações realizadas no arquivo "card-isoApplet.c".

```
[...]

/*
 * @brief Generate a secret key on the card.
 */
static int
isoApplet_ctl_generate_symmetric(sc_card_t *card,
      sc_cardctl_isoApplet_gen_symmetric_t *args)
{
    int r;
    sc_apdu_t apdu;
    u8 sbuf[SC_MAX_EXT_APDU_BUFFER_SIZE];
    u8 *p;
    char buff_sw[80];

    LOG_FUNC_CALLED(card->ctx);

    /* MANAGE SECURITY ENVIRONMENT (SET). Set the algorithm and
        key references. */
    sc_format_apdu(card, &apdu, SC_APDU_CASE_3_SHORT, 0x22, 0x42,
        0x01);

    p = sbuf;
    *p++ = 0x80; /* algorithm reference */
    *p++ = 0x01;
    *p++ = args->algorithm_ref;

    *p++ = 0x84; /* Private key reference */
    *p++ = 0x01;
    *p++ = args->s_key_ref;

    r = p - sbuf;
    p = NULL;

    apdu.lc = r;
    apdu.datalen = r;
    apdu.data = sbuf;

    r = sc_transmit_apdu(card, &apdu);
    LOG_TEST_RET(card->ctx, r, "APDU transmit failed");

    r = sc_check_sw(card, apdu.sw1, apdu.sw2);
    LOG_TEST_RET(card->ctx, r, "Card returned error");

    /* GENERATE SYMMETRIC KEY*/

    sc_format_apdu(card, &apdu, SC_APDU_CASE_1, 0x30, 0x00, 0x00);

    r = sc_transmit_apdu(card, &apdu);
    LOG_TEST_RET(card->ctx, r, "APDU transmit failed");

    r = sc_check_sw(card, apdu.sw1, apdu.sw2);
```

```
50      if (apdu.sw1 == 0x6A && apdu.sw2 == 0x81)
51      {
52          sc_log(card->ctx, "Key generation not supported by the card with that
                particular key type. "
53                  "Your card may not support the specified algorithm used by the
                        applet / specified  by you. ");
54      }
55      LOG_TEST_RET(card->ctx, r, "Card returned error");
56
57      LOG_FUNC_RETURN(card->ctx, SC_SUCCESS);
58  }
59
60   [...]
61
62  static  int
63  isoApplet_card_ctl(sc_card_t *card, unsigned long cmd, void *ptr)
64  {
65
66   [...]
67  case SC_CARDCTL_ISOAPPLET_GENERATE_SYMMETRIC:
68          r = isoApplet_ctl_generate_symmetric(card,
69                                                  (sc_cardctl_isoApplet_gen_symmetric_t *)
                                                        ptr);
70          break;
71
72   [...]
73  }
```

Alterações realizadas no arquivo "pkcs15-westcos.c".

```
1  static  struct  sc_pkcs15init_operations sc_pkcs15init_westcos_operations = {
2          [...]
3      NULL, // symmetric-key
4  };
```

Alterações realizadas no arquivo "pkcs15-starcos.c".

```
1  static  struct  sc_pkcs15init_operations sc_pkcs15init_starcos_operations = {
2          [...]
3      NULL, // symmetric-key
4  };
```

Alterações realizadas no arquivo "pkcs15-setcos.c".

```
1  static  struct  sc_pkcs15init_operations sc_pkcs15init_setcos_operations = {
2          [...]
3      NULL, // symmetric-key
4  };
```

Alterações realizadas no arquivo "pkcs15-sc-hsm.c".

```
static struct sc_pkcs15init_operations
sc_pkcs15init_sc_hsm_operations = {
        [...]
    NULL, // symmetric-key
};
```

Alterações realizadas no arquivo "pkcs15-rutoken.c".

```
static struct sc_pkcs15init_operations sc_pkcs15init_rutoken_operations = {
        [...]
    NULL, // symmetric-key
};
```

Alterações realizadas no arquivo "pkcs15-rtecp.c".

```
static struct sc_pkcs15init_operations sc_pkcs15init_rtecp_operations = {
        [...]
    NULL, // symmetric-key
};
```

Alterações realizadas no arquivo "pkcs15-openpgp.c".

```
static struct sc_pkcs15init_operations sc_pkcs15init_openpgp_operations = {
        [...]
    NULL, // symmetric-key
};
```

Alterações realizadas no arquivo "pkcs15-oberthur.c".

```
static struct sc_pkcs15init_operations
sc_pkcs15init_oberthur_operations = {
      [...]
    NULL, // symmetric-key
};
```

Alterações realizadas no arquivo "pkcs15-myeid.c".

```
static struct sc_pkcs15init_operations sc_pkcs15init_myeid_operations = {
        [...]
    NULL, // symmetric-key
};
```

Alterações realizadas no arquivo "pkcs15-muscle.c".

```
static struct sc_pkcs15init_operations sc_pkcs15init_muscle_operations = {
        [...]
    NULL, // symmetric-key
};
```

Alterações realizadas no arquivo "pkcs15-miocos.c".

```
1 static struct sc_pkcs15init_operations sc_pkcs15init_miocos_operations = {
2     [...]
3   NULL, // symmetric-key
4 };
```

Alterações realizadas no arquivo "pkcs15-jcop.c".

```
1 static struct sc_pkcs15init_operations sc_pkcs15init_jcop_operations = {
2    [...]
3  NULL, // symmetric-key
4 };
```

Alterações realizadas no arquivo "pkcs15-incrypto34.c".

```
1 static struct sc_pkcs15init_operations sc_pkcs15init_incrypto34_operations =
      {
2      [...]
3   NULL, // symmetric-key
4 };
```

Alterações realizadas no arquivo "pkcs15-iasecc.c".

```
1 static struct sc_pkcs15init_operations
2 sc_pkcs15init_iasecc_operations = {
3     [...]
4   NULL, // symmetric-key
5 };
```

Alterações realizadas no arquivo "pkcs15-gpk.c".

```
1 static struct sc_pkcs15init_operations sc_pkcs15init_gpk_operations = {
2      [...]
3   NULL, // symmetric-key
4 };
```

Alterações realizadas no arquivo "pkcs15-gids.c".

```
1 static struct sc_pkcs15init_operations sc_pkcs15init_gids_operations =
2 {
3    [...]
4  NULL, // symmetric-key
5 };
```

Alterações realizadas no arquivo "pkcs15-entersafe.c".

```
1 static struct sc_pkcs15init_operations sc_pkcs15init_entersafe_operations = {
2      [...]
```

```
3    NULL, // symmetric-key
4 };
```

Alterações realizadas no arquivo "pkcs15-cflex.c".

```
1 static struct sc_pkcs15init_operations sc_pkcs15init_cryptoflex_operations =
      {
2     [...]
3    NULL, // symmetric-key
4 };
```

Alterações realizadas no arquivo "pkcs15-cardos.c".

```
1 static struct sc_pkcs15init_operations sc_pkcs15init_cardos_operations = {
2     [...]
3    NULL, // symmetric-key
4 };
```

Alterações realizadas no arquivo "pkcs15-authentic.c".

```
1 static struct sc_pkcs15init_operations
2 sc_pkcs15init_authentic_operations = {
3     [...]
4    NULL, // symmetric-key
5 };
```

Alterações realizadas no arquivo "pkcs15-asepcos.c".

```
1 static struct sc_pkcs15init_operations sc_pkcs15init_asepcos_operations = {
2     [...]
3    NULL, // symmetric-key
4 };
```

Alterações realizadas no arquivo "pkcs15.profile".

```
1  [...]
2 
3 # Default settings.
4 # This option block will always be processed.
5 option default {
6      macros {
7       [...]
8      skdf-size = 256;
9      }
10 }
11 
12  [...]
13 
14 filesystem {
```

```
15      [...]
16
17       EF PKCS15-SKDF {
18          file -id = 4406;
19          size     = $skdf-size;
20          acl      = $protected;
21       }
22
23      [...]
24  }
```

Alterações realizadas no arquivo "pkcs15-skey.c".

```
1    [...]
2
3   #define C_ASN1_SKEY_CHOICE_SIZE 6
4   static const struct sc_asn1_entry
        c_asn1_skey_choice[C_ASN1_SKEY_CHOICE_SIZE] = {
5           { "genericSecretKey", SC_ASN1_PKCS15_OBJECT,
                SC_ASN1_TAG_SEQUENCE | SC_ASN1_CONS,
                SC_ASN1_OPTIONAL, NULL, NULL },
6           { "desKey", SC_ASN1_PKCS15_OBJECT, SC_ASN1_CTX | 2 |
                SC_ASN1_CONS, SC_ASN1_OPTIONAL, NULL, NULL },
7           { "des2Key", SC_ASN1_PKCS15_OBJECT, SC_ASN1_CTX | 3 |
                SC_ASN1_CONS, SC_ASN1_OPTIONAL, NULL, NULL },
8           { "des3Key", SC_ASN1_PKCS15_OBJECT, SC_ASN1_CTX | 4 |
                SC_ASN1_CONS, SC_ASN1_OPTIONAL, NULL, NULL },
9         { "aesKey", SC_ASN1_PKCS15_OBJECT, SC_ASN1_CTX | 5 |
              SC_ASN1_CONS, SC_ASN1_OPTIONAL, NULL, NULL },
10        { NULL, 0, 0, 0, NULL, NULL }
11  };
12
13  #define C_ASN1_SKEY_SIZE 2
14  static const struct sc_asn1_entry c_asn1_skey[C_ASN1_SKEY_SIZE] = {
15     { "secretKey", SC_ASN1_CHOICE, 0, 0, NULL, NULL},
16     { NULL, 0, 0, 0, NULL, NULL }
17  };
18
19   [...]
20
21  int sc_pkcs15_encode_skdf_entry(struct sc_context *ctx, const struct
        sc_pkcs15_object *obj,
22                 u8 **buf, size_t *buflen)
23  {
24
25     struct sc_asn1_entry
           asn1_com_key_attr[C_ASN1_COM_KEY_ATTR_SIZE];
26     struct sc_asn1_entry
           asn1_com_skey_attr[C_ASN1_COM_SKEY_ATTR_SIZE];
27     struct sc_asn1_entry asn1_skey_choice[C_ASN1_SKEY_CHOICE_SIZE];
28     struct sc_asn1_entry asn1_generic_skey_attr[
```

```
29                  C_ASN1_COM_GENERIC_SKEY_ATTR_SIZE];
30      struct sc_asn1_entry asn1_sKey[C_ASN1_SKEY_SIZE];
31
32      struct sc_asn1_pkcs15_object sk_obj = {
33          (struct sc_pkcs15_object *) obj, asn1_com_key_attr,
34           asn1_com_skey_attr, asn1_generic_skey_attr
35          };
36
37      struct sc_pkcs15_skey_info *sKey = (struct sc_pkcs15_skey_info *)
            obj->data;
38      int r;
39      size_t af_len, usage_len;
40
41      sc_copy_asn1_entry(c_asn1_skey, asn1_sKey);
42      sc_copy_asn1_entry(c_asn1_com_skey_attr, asn1_com_skey_attr);
43      sc_copy_asn1_entry(c_asn1_generic_skey_attr, asn1_generic_skey_attr);
44      sc_copy_asn1_entry(c_asn1_com_key_attr, asn1_com_key_attr);
45      sc_copy_asn1_entry(c_asn1_skey_choice, asn1_skey_choice);
46
47
48      switch(obj->type){
49      case SC_PKCS15_TYPE_SKEY_AES:
50          sc_format_asn1_entry(asn1_skey_choice + 4, &sk_obj, NULL, 1);
51          sc_format_asn1_entry(asn1_generic_skey_attr + 0, &sKey->path,
                NULL, 1);
52          break;
53      default:
54          sc_log(ctx,  "Unsupported secret key type: %X", obj->type);
55          LOG_FUNC_RETURN(ctx, SC_ERROR_INTERNAL);
56          break;
57      }
58
59      sc_format_asn1_entry(asn1_com_key_attr + 0, &sKey->id, NULL, 1);
60      usage_len = sizeof(sKey->usage);
61      sc_format_asn1_entry(asn1_com_key_attr + 1, &sKey->usage,
            &usage_len, 1);
62      if (sKey->native == 0)
63          sc_format_asn1_entry(asn1_com_key_attr + 2, &sKey->native,
                NULL, 1);
64      if (sKey->access_flags) {
65          af_len = sizeof(sKey->access_flags);
66          sc_format_asn1_entry(asn1_com_key_attr + 3, &sKey->access_flags,
                &af_len, 1);
67      }
68      if (sKey->key_reference >= 0)
69          sc_format_asn1_entry(asn1_com_key_attr + 4,
                &sKey->key_reference, NULL, 1);
70
71      sc_format_asn1_entry(asn1_com_skey_attr + 0, &sKey->value_len,
            NULL, 1);
72      sc_format_asn1_entry(asn1_sKey + 0, asn1_skey_choice, NULL, 1);
73
```

```
74      r = sc_asn1_encode(ctx, asn1_sKey, buf, buflen);
75      sc_log(ctx, "Key path %s", sc_print_path(&sKey->path));
76
77      return r;
78  }
79
80   [...]
```

Alterações realizadas no arquivo "pkcs15.h".

```
81   [...]
82
83  #define SC_PKCS15_TYPE_SKEY_AES    0x305
84
85   [...]
86
87  int sc_pkcs15_encode_skdf_entry(struct sc_context *ctx,
88                  const struct sc_pkcs15_object *obj,
89                  u8 **buf, size_t *buflen);
90
91   [...]
```