FELIPE MICHELS FONTOURA

UMA API CRIPTOGRÁFICA PARA APLICAÇÕES EMBARCADAS

Dissertação submetida ao Programa de Pós-Graduação em Computação Aplicada da Universidade Tecnológica Federal do Paraná como requisito parcial para a obtenção do título de Mestre em Computação Aplicada.

Curitiba PR
Agosto de 2016

FELIPE MICHELS FONTOURA

UMA API CRIPTOGRÁFICA PARA APLICAÇÕES EMBARCADAS

Dissertação submetida ao Programa de Pós-Graduação em Computação Aplicada da Universidade Tecnológica Federal do Paraná como requisito parcial para a obtenção do título de Mestre em Computação Aplicada.

Área de concentração: *Engenharia de Sistemas Computacionais*

Orientador: Carlos Alberto Maziero
Co-orientador: Marco Aurélio Wehrmeister

Curitiba PR
Agosto de 2016

Dados Internacionais de Catalogação na Publicação

F684a 2016

Fontoura, Felipe Michels

Uma API criptográfica para aplicações embarcadas / Felipe Michels Fontoura.-- 2016.

155 f. : il. ; 30 cm

Texto em português, com resumo em inglês

Dissertação (Mestrado) - Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Computação Aplicada, Curitiba, 2016

Bibliografia: p. 109-117

1. Criptografia. 2. Criptografia de dados (Computação). 3. Sistemas embarcados (Computadores). 4. Computação – Dissertações. I. Maziero, Carlos Alberto, orient. II. Wehrmeister, Marco Aurelio, coorient. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Computação Aplicada. IV. Título.

CDD: Ed. 22 -- 621.39

Biblioteca Central da UTFPR, Câmpus Curitiba

ATA DE DEFESA DE DISSERTAÇÃO DE MESTRADO Nº 48

Aos 31 dias do mês de agosto de 2016 realizou-se na sala C-301 a sessão pública de Defesa da Dissertação de Mestrado intitulada "Uma API Criptográfica para Aplicações Embarcadas", apresentado pelo aluno **Felipe Michels Fontoura** como requisito parcial para a obtenção do título de Mestre em Computação Aplicada, na área de concentração "Engenharia de Sistemas Computacionais", linha de pesquisa "Redes e Sistemas Distribuídos".

Constituição da Banca Examinadora:

Profº. Drº. Carlos Alberto Maziero (Presidente) UTFPR ___________________________

Profº. Drº. Douglas Paulo B. Renaux UTFPR ____________________________

Profº. Drº. Diego de Freitas Aranha UNICAMP_____________________________

Em conformidade com os regulamentos do Programa de Pós-Graduação em Computação aplicada e da Universidade Tecnológica Federal do Paraná, o trabalho apresentado foi considerado ____________________________ (aprovado/reprovado) pela banca examinadora. No caso de aprovação, a mesma está condicionada ao cumprimento integral das exigências da banca examinadora, registradas no verso desta ata, da entrega da versão final da dissertação em conformidade com as normas da UTFPR e da entrega da documentação necessária à elaboração do diploma, em até _____________ dias desta data.

Ciente (assinatura do aluno): ____

(para uso da coordenação)

A Coordenação do PPGCA/UTFPR declara que foram cumpridos todos os requisitos exigidos pelo programa para a obtenção do título de Mestre.

Curitiba PR, _____/_____/____________ ________________________________

"A Ata de Defesa original está arquivada na Secretaria do PPGCA".

Agradecimentos

Agradeço primeiramente ao orientador, Prof. Dr. Carlos Alberto Maziero, e ao co-orientador, Prof. Dr. Marco Aurélio Wehrmeister, pelo auxílio na elaboração deste trabalho. Agradeço também a outros professores que me guiaram durante a minha vida acadêmica, como a Prof. Dr.ª Tânia Mezzadri Centeno, o Prof. Dr. Douglas Paulo Bertrand Renaux, entre tantos outros.

Agradeço aos meus pais, João Maria Fontoura Junior e Rosa Maria Michels Fontoura, e à minha irmã, Mariana Michels Fontoura, pelo apoio e pela compreensão durante a elaboração deste trabalho.

Agradeço à minha namorada, Thayse Marques Solis, que teve muita paciência, me deu ideias e me auxiliou na revisão desta dissertação e do artigo.

Agradeço aos meus amigos e colegas de profissão, que me deram sugestões referentes a este trabalho e tiveram muita paciência para me ouvir falar repetidas vezes sobre ele: Eduardo Cromack Lippmann, Juliana Schwartz e Cauan Boss.

Agradeço aos amigos que fiz durante o curso de graduação, sem os quais esta jornada seria sem dúvida mais árdua: Lucas Longen Gioppo, Leandro Piekarski do Nascimento, William Hitoshi Tsunoda Meira e Felipe Luiz Bill.

Agradeço também aos demais familiares e amigos.

Resumo

Neste documento, está apresentada a *GEmSysC*, uma *API* criptográfica unificada para aplicações embarcadas. Camadas de abstração compatíveis com esta *API* podem ser construídas sobre bibliotecas existentes, de forma que as funcionalidades criptográficas podem ser acessadas pelo *software* de alto nível de forma consistente e independente da implementação. As características da *API* foram definidas com base em boas práticas de construção de *APIs*, práticas indicadas em *software* embarcado e também com base em outras bibliotecas e padrões criptográficos existentes. A principal inspiração para este projeto foi o padrão *CMSIS-RTOS*, que também busca unificar interfaces para *software* embarcado de forma independente da implementação, mas é voltado a sistemas operacionais, não a funcionalidades criptográficas. A estrutura da *GEmSysC* é modular, sendo composta de um *core* genérico e módulos acopláveis, um para cada algoritmo criptográfico. Nesta dissertação, está apresentada a especificação do *core* e de três módulos: *AES*, *RSA* e *SHA-256*. Ainda que a *GEmSysC* tenha sido elaborada para utilização em sistemas embarcados, ela também poderia ser utilizada em computadores computacionais, já que, em última instância, sistemas embarcados são sistemas computacionais. Como provas de conceito, foram feitas duas implementações da *GEmSysC*: uma sobre a biblioteca *wolfSSL*, que é de código aberto e voltada a sistemas embarcados, e outra sobre a *OpenSSL*, que é amplamente utilizada e de código aberto, mas não é voltada especificamente a sistemas embarcados. A primeira implementação foi testada em um processador *Cortex-M3* sem sistema operacional, enquanto a segunda foi testada em um *PC* com sistema operacional *Windows* 10. Mostrou-se que a *GEmSysC* é, sob alguns aspectos, mais simples que outras bibliotecas. Mostrou-se também que o *overhead* da camada de abstração é pequeno, ficando entre pouco mais de 0% e 0,17% na implementação voltada a sistemas embarcados e entre 0,03% e 1,40% na implementação para *PC*. Apresentaram-se ainda os valores dos custos de memória de programa e de *RAM* de cada uma das implementações.

Palavras-chave: interface de *software*, criptografia, sistemas embarcados.

Abstract

This document presents GEmSysC, an unified cryptographic API for embedded systems. Software layers implementing this API can be built over existing libraries, allowing embedded software to access cryptographic functions in a consistent way that does not depend on the underlying library. The API complies to good practices for API design and good practices for embedded software development and took its inspiration from other cryptographic libraries and standards. The main inspiration for creating GEmSysC was the CMSIS-RTOS standard, which defines an unified API for embedded software in an implementation-independent way, but targets operating systems instead of cryptographic functions. GEmSysC is made of a generic core and attachable modules, one for each cryptographic algorithm. This document contains the specification of the core of GEmSysC and three of its modules: AES, RSA and SHA-256. GEmSysC was built targeting embedded systems, but this does not restrict its use only in such systems – after all, embedded systems are just very limited computing devices. As a proof of concept, two implementations of GEmSysC were made. One of them was built over wolfSSL, which is an open source library for embedded systems. The other was built over OpenSSL, which is open source and a *de facto* standard. Unlike wolfSSL, OpenSSL does not specifically target embedded systems. The implementation built over wolfSSL was evaluated in a Cortex-M3 processor with no operating system while the implementation built over OpenSSL was evaluated on a personal computer with Windows 10 operating system. This document displays test results showing GEmSysC to be simpler than other libraries in some aspects. These results have shown that both implementations incur in little overhead in computation time compared to the cryptographic libraries themselves. The overhead of the implementation has been measured for each cryptographic algorithm and is between around 0% and 0.17% for the implementation over wolfSSL and between 0.03% and 1.40% for the one over OpenSSL. This document also presents the memory costs for each implementation.

Keywords: software interfaces, cryptography, embedded systems.

Sumário

Resumo ix

Abstract xi

Lista de Figuras xvii

Lista de Tabelas xx

Lista de Códigos xxi

Lista de Abreviações xxii

1 Introdução 1
- 1.1 Desafio 1
- 1.2 Motivação e problemática 2
- 1.3 Objetivos 2
- 1.4 Organização do documento 3

2 Segurança em sistemas embarcados 5
- 2.1 Sistemas embarcados 5
  - 2.1.1 Sistemas móveis 6
- 2.2 Segurança de sistemas computacionais 6
- 2.3 Segurança em sistemas embarcados 8
  - 2.3.1 Ataques a sistemas embarcados 10
  - 2.3.2 Estudos de caso 15
- 2.4 Análise de domínio 21
  - 2.4.1 Requisitos de segurança 21
  - 2.4.2 Modelagem de sistema 22
  - 2.4.3 Funcionalidades de uma biblioteca criptográfica 24
- 2.5 Análise 25

3 Interfaces de *software* 27
- 3.1 Definição 27
- 3.2 Qualidade de *APIs* 28
  - 3.2.1 Boas práticas 28
  - 3.2.2 Formas de avaliação 31
  - 3.2.3 Discussão 34

3.3 Drivers . . . 34
3.3.1 Organização de código-fonte . . . 36
3.3.2 *Device Object Model* . . . 36
3.3.3 *Multi tiered architecture* . . . 37
3.3.4 *Abstraction layers* . . . 38
3.3.5 Discussão . . . 38
3.4 Sistemas operacionais . . . 39
3.4.1 *CMSIS-RTOS* . . . 40
3.4.2 *OSEK/VDX* . . . 41
3.4.3 *FreeRTOS* . . . 42
3.4.4 *ChibiOS* . . . 43
3.4.5 Discussão . . . 44
3.5 Bibliotecas criptográficas . . . 45
3.5.1 *Cryptoki* . . . 46
3.5.2 *OpenSSL* . . . 49
3.5.3 *wolfSSL* . . . 51
3.5.4 Discussão . . . 52
3.6 Análise . . . 53

4 *GEmSysC* **55**
4.1 Justificativa . . . 55
4.2 Requisitos . . . 56
4.3 Características . . . 57
4.3.1 Sistemas embarcados . . . 57
4.3.2 Convenções . . . 58
4.3.3 Discussão . . . 61
4.4 Especificação . . . 61
4.4.1 *Core* da biblioteca . . . 64
4.4.2 Módulo de *AES* . . . 68
4.4.3 Módulo de *RSA* . . . 73
4.4.4 Módulo de *SHA*-256 . . . 79
4.4.5 Discussão . . . 84
4.5 Análise . . . 84

5 Avaliação e Resultados **87**
5.1 Implementações . . . 87
5.1.1 *GEmSysC* sobre *OpenSSL* . . . 88
5.1.2 *GEmSysC* sobre *wolfSSL* . . . 89
5.1.3 Comparação . . . 89
5.2 Testes e avaliação . . . 90
5.2.1 Complexidade de código . . . 90
5.2.2 Desempenho . . . 96
5.2.3 Memória . . . 100
5.2.4 Imprecisões . . . 101
5.3 Análise . . . 103

6 Conclusão **105**
6.1 Trabalhos futuros 107

A Especificação completa da *GEmSysC* **119**

B Códigos da análise de complexidade **135**
B.1 *AES* em modo *ECB* 135
B.2 *AES* em modo *CBC* 136
B.3 *RSA* com preenchimento *PKCS* #1 v1.5 140
B.4 *RSA* com preenchimento *OAEP* 143
B.5 *SHA-256* 144
B.6 Todos os códigos 146
B.7 Visão geral 148

C Resultados das métricas de complexidade **149**

D Resultados dos testes de desempenho **151**

Lista de Figuras

2.1 Representação visual da ontologia utilizada para descrever o conhecimento específico de domínio [Vasilevskaya, 2015]. 23
2.2 Representação visual da ontologia utilizada para descrever a avaliação dos mecanismos [Vasilevskaya, 2015]. 24

4.1 Tecnologias estudadas agrupadas por suas características (autoria própria). . . . 56
4.2 Relação entre uma camada de abstração utilizando a *API* desenvolvida, a aplicação de alto nível e a implementação da biblioteca criptográfica (autoria própria). 59
4.3 Exemplo de como incluir a biblioteca criptográfica e o módulo com suporte a *AES* em um arquivo de código (autoria própria). 60
4.4 Diagrama de módulos da *GEmSysC* (autoria própria). 62

5.1 Número de tipos de dados definidos na *API* para cada funcionalidade avaliada (autoria própria). 93
5.2 Número de constantes definidas na *API* para cada funcionalidade avaliada. O *SHA-256* foi omitido pois é igual a zero para todas as *APIs* (autoria própria). . . 94
5.3 Número de funções e *macros* parametrizadas para cada funcionalidade avaliada (autoria própria). 94
5.4 Número total de parâmetros das funções e *macros* parametrizadas para cada funcionalidade avaliada (autoria própria). 95
5.5 Número de linhas de código para cada funcionalidade avaliada (autoria própria). 95

Lista de Tabelas

3.1 Operações com objetos da *Cryptoki*. 48
3.2 Operações com chaves da *Cryptoki*. 48
3.3 Operações de cifragem de dados da *Cryptoki*. 49
3.4 Operações de decifragem de dados da *Cryptoki*. 49
3.5 Operações de cálculo de resumo criptográfico da *Cryptoki*. 49
3.6 Operações de assinatura da *Cryptoki*. 50

4.1 Lista de arquivos básicos presentes nas implementações da *GEmSysC*. 63
4.2 Comparação das *APIs* com relação à forma de definir e utilizar instâncias de mecanismos criptográficos. 65
4.3 Lista de funções e *macros* que parecem funções do *core* da *GEmSysC*. 68
4.4 Como são definidos os parâmetros do *AES* na *Cryptoki*, na *OpenSSL* e na *wolfSSL*. 72
4.5 Lista de funções e *macros* que parecem funções do módulo de *AES* da *GEmSysC*. 73
4.6 Como são definidos os parâmetros do *RSA* na *Cryptoki*, na *OpenSSL* e na *wolfSSL*. 78
4.7 Lista de funções e *macros* que parecem funções do módulo de *RSA* da *GEmSysC*. 79
4.8 Lista de funções e *macros* que parecem funções do módulo de *SHA-256* da *GEmSysC*. 83

5.1 Funções da camada de abstração de testes. 97
5.2 Medidas de desempenho da *wolfSSL* e da *GemSysC* em plataforma *Cortex-M3*. 99
5.3 Medidas de desempenho da *OpenSSL* e da *GemSysC* em plataforma *x86*. 99
5.4 Memória de código e dados constantes ocupada pela *GEmSysC* sobre *wolfSSL*. 100
5.5 Memória de código e dados constantes ocupada pela *GEmSysC* sobre *OpenSSL*. 100
5.6 Estruturas que representam instâncias de algoritmos criptográficos na *wolfSSL*, na *OpenSSL* e nas duas implementações da *GEmSysC*. 101
5.7 Memória ocupada por instâncias de algoritmo criptográfico da *wolfSSL* e da *GEmSysC* sobre *wolfSSL*. 101
5.8 Memória ocupada por instâncias de algoritmo criptográfico da *OpenSSL* e da *GEmSysC* sobre *OpenSSL*. 102

C.1 Comparação entre códigos com *AES* em modo *ECB*. 149
C.2 Comparação entre códigos com *AES* em modo *CBC*. 149
C.3 Comparação entre códigos com *RSA* com preenchimento *PKCS* #1 v1.5. 150
C.4 Comparação entre códigos com *RSA* com preenchimento *OAEP*. 150
C.5 Comparação entre códigos com *SHA-256*. 150
C.6 Comparação entre todos os códigos. 150

D.1 Medidas de desempenho da *wolfSSL* e da *GemSysC* em plataforma *Cortex-M3*. 152

D.2 Medidas de desempenho da *OpenSSL* e da *GemSysC* em plataforma *x86*. 153
D.3 Média, desvio padrão e coeficiente de variação das medidas de desempenho da *wolfSSL* e da *GemSysC* em plataforma *Cortex-M3*. 154
D.4 Média, desvio padrão e coeficiente de variação das medidas de desempenho da *OpenSSL* e da *GemSysC* em plataforma *x86*. 155

Lista de Códigos

4.1 Cifragem simétrica de dados com *AES* usando a *Cryptoki*. 69
4.2 Cifragem simétrica de dados com *AES* usando a *OpenSSL*. 70
4.3 Cifragem simétrica de dados com *AES* usando a *wolfSSL*. 71
4.4 Cifragem simétrica de dados com *AES* usando a *GEmSysC*. 73
4.5 Cifragem assimétrica de dados com *RSA* usando a *Cryptoki*. 75
4.6 Cifragem assimétrica de dados com *RSA* usando a *OpenSSL*. 76
4.7 Cifragem assimétrica de dados com *RSA* usando a *wolfSSL*. 77
4.8 Cifragem assimétrica de dados com *RSA* usando a *GEmSysC*. 80
4.9 Resumo criptográfico de dados com *SHA*-256 usando a *Cryptoki*. 81
4.10 Resumo criptográfico de dados com *SHA*-256 usando a *OpenSSL*. 82
4.11 Resumo criptográfico de dados com *SHA*-256 usando a *wolfSSL*. 83
4.12 Resumo criptográfico de dados com *SHA*-256 usando a *GEmSysC*. 84
B.1 Cifragem e decifragem usando *AES* em modo *ECB* pela *GEmSysC*. 136
B.2 Cifragem e decifragem usando *AES* em modo *ECB* pela *OpenSSL*. 137
B.3 Cifragem e decifragem usando *AES* em modo *ECB* pela *wolfSSL*. 137
B.4 Cifragem e decifragem usando *AES* em modo *CBC* pela *GEmSysC*. 138
B.5 Cifragem e decifragem usando *AES* em modo *CBC* pela *OpenSSL*. 139
B.6 Cifragem e decifragem usando *AES* em modo *CBC* pela *wolfSSL*. 139
B.7 Cifragem e decifragem usando *RSA* com preenchimento *PKCS* #1 v1.5 pela *GEmSysC*. 140
B.8 Cifragem e decifragem usando *RSA* com preenchimento *PKCS* #1 v1.5 pela *OpenSSL*. 141
B.9 Cifragem e decifragem usando *RSA* com preenchimento *PKCS* #1 v1.5 pela *wolfSSL*. 142
B.10 Cifragem e decifragem usando *RSA* com preenchimento *OAEP* pela *GEmSysC*. 143
B.11 Cifragem e decifragem usando *RSA* com preenchimento *OAEP* pela *OpenSSL*. . 144
B.12 Cifragem e decifragem usando *RSA* com preenchimento *OAEP* pela *wolfSSL*. . 145
B.13 Resumo criptográfico usando *SHA-256* pela *GEmSysC*. 145
B.14 Resumo criptográfico usando *SHA-256* pela *OpenSSL*. 146
B.15 Resumo criptográfico usando *SHA-256* pela *wolfSSL*. 146

Lista de Abreviações

API	*Application Programming Interface*
ASLR	*Address Space Layout Randomization*
CMSIS	*Cortex Microcontroller Software Interface Standard*
CPU	*Central Processing Unit*
DEP	*Data Execution Prevention*
DoS	*Denial of Service*
DRM	*Digital Rights Management*
FIPS	*Federal Information Processing Standard*
GPS	*Global Positioning System*
HAL	*Hardware Abstraction Layer*
IoT	*Internet of Things*
LBS	*Location-based Services*
NIST	*National Institute of Standards and Technology*
POSIX	*Portable Operating System Interface*
RAM	*Random-access Memory*
ROM	*Read-only Memory*
RTC	*Real-time Clock*
RTOS	*Real Time Operating System*
SoC	*System on a Chip*
SSL	*Secure Socket Layer*

Capítulo 1

Introdução

O uso ubíquo de sistemas embarcados e outros mais limitados [Wangham et al., 2013], como redes de sensores, torna estes sistemas cada vez mais responsáveis por atividades críticas, tornando essencial desenvolver soluções adequadas de segurança para eles [Kermani et al., 2013]. É comum que estes sistemas possam ser operados remotamente, o que só é possível devido à sua crescente conectividade com redes como a *Internet*. Tanto a localização física quanto a sua conectividade fazem com que estes sistemas sejam vulneráveis a ataques intencionais [Parameswaran and Wolf, 2008], que podem variar tanto em relação à forma como são executados quanto em relação aos objetivos finais. Por exemplo, um atacante pode ter como alvo um sistema de alarmes conectado à *Internet* com o objetivo de realizar um furto, o que pode ser feito danificando os dispositivos físicos do alarme (sensores de presença, câmeras, etc) ou prejudicando seu funcionamento interno (tomando o controle do sistema ou afetando sua disponibilidade).

Ao criar soluções seguras, é comum que os desenvolvedores utilizem como "blocos de construção" bibliotecas de *software*, como as que fornecem funcionalidades relacionadas à criptografia [Bernstein et al., 2012], e *hardware* intrinsecamente seguro, como por exemplo sistemas em um único *chip* [Jyostna and Padmaja, 2011]. Utilizar estes blocos facilita o projeto de sistemas, pois permite que se abstraiam detalhes de implementação, sejam de *software* ou de *hardware*.

1.1 Desafio

Existem diversas bibliotecas criptográficas disponíveis para sistemas de computação geral, mas o número de bibliotecas criptográficas voltadas para sistemas embarcados é consideravelmente menor. As bibliotecas voltadas a sistemas de computação geral muitas vezes não consideram limitações específicas de sistemas embarcados [Hatzivasilis et al., 2014]. Por exemplo, a biblioteca *OpenSSL*, que é amplamente utilizada em servidores [Netcraft, 2014], tem um código executável grande demais, o que pode impedir que seja utilizada em sistemas embarcados [Hatzivasilis et al., 2014].

Muitas destas bibliotecas, como por exemplo *wolfSSL* e *OpenSSL*, fornecem ao *software* principal funcionalidades semelhantes, como *AES*, *RSA* e *SHA-256*. Entretanto, em geral, a forma como as aplicações acessam as funcionalidades varia de acordo com a biblioteca utilizada [wolfSSL, 2016, OpenSSL, 2015]. Como as interfaces são diferentes, espera-se que o

código que realiza certa operação através de uma biblioteca seja diferente do que utiliza outra biblioteca para a realizar a mesma operação.

Cada biblioteca criptográfica é disponibilizada para um conjunto de arquiteturas computacionais. Se um desenvolvedor de sistemas elaborar aplicações para plataformas embarcadas variadas, é possível que tenha de utilizar bibliotecas criptográficas diferentes entre um projeto e outro. Como geralmente bibliotecas expõem funcionalidades às aplicações de maneiras diferentes umas das outras, desenvolvedores têm de aprender a utilizar as *APIs* de diversas bibliotecas criptográficas. Este fator também dificulta que se aproveite código entre projetos.

1.2 Motivação e problemática

Não foi encontrado um padrão amplamente adotado para as *APIs* das bibliotecas criptográficas utilizadas em sistemas embarcados. Entretanto, existem bibliotecas multiplataforma para estes sistemas, como *wolfSSL* [wolfSSL, 2016], bibliotecas multiplataforma para aplicações de computação geral, como *OpenSSL* [OpenSSL, 2015], e *APIs* criptográficas padronizadas para aplicações de computação geral, como *Cryptoki* (definida no padrão *PKCS* #11) [Griffin and Fenwick, 2015a].

A existência de padrões facilita o desenvolvimento de *software*, pois com ela, a curva de aprendizado é menor e há maior possibilidade de reaproveitamento de código entre projetos [Johnson, 1997, Renaux and Pottker, 2014]. Segundo a pesquisa em [UBM Tech, 2015], em 2015, 39% dos desenvolvedores de sistemas embarcados evitavam trocar de sistema operacional para manter a compatibilidade do *software*, 35% evitaram trocar de sistema operacional pois já tinham boa experiência com um sistema operacional, e 22% evitavam trocar de sistema operacional devido ao impacto financeiro e de tempo que isto pode causar. Em todos estes casos, as razões apresentadas dizem respeito às interfaces de *software*: no primeiro caso, o problema é a compatibilidade em si, no segundo, o problema é o aprendizado de nova *API*, enquanto no terceiro, o problema é a dificuldade de adaptar o *software* às novas interfaces.

1.3 Objetivos

O projeto descrito nesta dissertação é uma *API* criptográfica unificada para sistemas embarcados. Ela não acompanha uma implementação específica, pois foi concebida para ser utilizada, sob a forma de uma camada de abstração, sobre bibliotecas criptográficas existentes. Com isso, é possível ter as vantagens de uma *API* unificada e ainda utilizar a biblioteca criptográfica de preferência.

A ideia de construir uma *API* unificada foi inspirada no padrão *CMSIS-RTOS*, que serve a uma finalidade semelhante, mas é voltado a sistemas operacionais embarcados [CMSIS, 2012]. Não foi encontrada nenhuma pesquisa de mercado especificamente sobre *APIs* criptográficas para sistemas embarcados, mas os resultados da pesquisa em [UBM Tech, 2015] mostram que a falta de padronização de *APIs* de *software* é um problema para os desenvolvedores, tal qual discutido na seção 1.2. Isso também é corroborado pelas práticas sugeridas em [Henning, 2007] e [Blanchette, 2008]. Por estes motivos, pareceu relevante sugerir uma *API* unificada para fornecer funcionalidades criptográficas às aplicações embarcadas.

Segundo a pesquisa em [UBM Tech, 2015], em 2015, a grande maioria dos desenvolvedores de sistemas embarcados utilizavam as linguagens *C* e *C++* em seus projetos principais,

totalizando 66% e 19%, respectivamente. Por comparação, o terceiro item da lista é a linguagem *assembly*, que totalizou 3% dos entrevistados. A *API* criptográfica foi elaborada na linguagem *C*, pois ela é a mais utilizada pelos desenvolvedores segundo a pesquisa. As linguagens *C* e *C++* têm uma mesma origem, e muitos códigos feitos em uma também são válidos na outra. Se as incompatibilidades forem consideradas durante o desenvolvimento, é possível garantir que códigos tenham compatibilidade tanto com *C* quanto com *C++* [Tribble, 2001]. Uma vez que é possível desenvolver códigos em *C* que também são válidos em *C++*, a *API* projetada pode alcançar mais de 80% dos desenvolvedores de sistemas embarcados, segundo dados da pesquisa.

Para elaborar a *API* apresentada neste documento, baseou-se em práticas consideradas "boas" por outros autores. Além de práticas com relação a *APIs* em geral [Henning, 2007, Blanchette, 2008], estudaram-se também convenções para melhorar *software* voltado a sistemas embarcados [Barry and Hartnett, 2007, Gradinaru, 2010, Graves, 2011, MacMillan, 2011]. Estas práticas e convenções foram utilizadas como referência para elaborar a *API* criptográfica.

1.4 Organização do documento

No capítulo 2, está apresentada uma análise sobre segurança de sistemas embarcados em geral, o que explica a motivação deste trabalho. No capítulo 3, expõe-se uma análise mais detalhada de algumas *APIs* criptográficas existentes e outras *APIs* voltadas para sistemas embarcados (como sistemas operacionais e *drivers*). No capítulo 4, apresenta-se a *API* elaborada, e no capítulo 5, sua validação (procedimento de avaliação e resultados). No capítulo 6, está a conclusão deste trabalho e também se propõem sugestões para trabalhos futuros.

Capítulo 2

Segurança em sistemas embarcados

Neste capítulo, estão apresentados estudos sobre segurança em dispositivos embarcados e sobre os ataques a que eles estão sujeitos. O objetivo desta pesquisa foi verificar que problemas de segurança existem em sistemas embarcados e como eles podem ser solucionados. Também se apresenta um estudo de domínio, com uma análise de requisitos comuns de segurança e dos blocos de construção necessários para implementá-los.

2.1 Sistemas embarcados

Sistemas embarcados[1] são dispositivos computacionais integrados a outros sistemas maiores [Lee and Seshia, 2011]. Estes dispositivos são mais especializados que computadores convencionais [Parameswaran and Wolf, 2008] e também menos visíveis [Lee and Seshia, 2011]. Dispositivos como estes são utilizados em uma grande variedade de áreas, que vão desde o controle de sistemas críticos até a captura de dados em ambientes hostis [Parameswaran and Wolf, 2008], e suas aplicações incluem: monitoramento ambiental, rastreio de animais selvagens, ferramentas de auxílio nas áreas de saúde e medicina [Jyostna and Padmaja, 2011], dispositivos móveis, cartões inteligentes, infra-estruturas automotivas, controle automatizado de aeronaves e indústrias [Kermani et al., 2013], entre outros.

Sistemas embarcados devem ser capazes de realizar uma variedade de funções sem necessidade de supervisão humana [Jyostna and Padmaja, 2011] ou com supervisão remota, pois muitas vezes são colocados em ambientes inacessíveis [Parameswaran and Wolf, 2008]. Com o tempo, espera-se que a quantidade de dispositivos embarcados por humano cresça, impossibilitando a intervenção humana para realizar atividades administrativas, como por exemplo aplicar atualizações de segurança [Kermani et al., 2013]. Apesar de serem cada vez mais responsáveis por atividades críticas, tais dispositivos são muito mais limitados que computadores convencionais em diversos aspectos, como custo, consumo energético, poder de processamento, tamanho e peso [Parameswaran and Wolf, 2008].

Para que possam ser operados sem necessidade de proximidade física, muitos dispositivos embarcados têm a capacidade de comunicar-se com outros dispositivos ou sistemas. Esta conectividade permite tanto o envio de comandos e captura de dados por parte de operadores humanos [Parameswaran and Wolf, 2008] quanto a atuação em conjunto de dispositivos distintos para um mesmo fim [Atzori et al., 2010]. Esta possibilidade de operação remota dos

[1] *“Embedded systems”* em inglês.

dispositivos abre a possibilidade de que eles se tornem ubíquos, isto é, presentes por todo o ambiente, interagindo com humanos de forma transparente [Wangham et al., 2013]. Esta tendência está relacionada à "*Internet* das Coisas"[2], um paradigma que envolve a presença pervasiva de sistemas conectados à *Internet* nos mais diversos ambientes [Atzori et al., 2010].

2.1.1 Sistemas móveis

Muitos sistemas embarcados são também sistemas móveis, ou seja, aqueles cuja localização pode mudar. A "computação móvel" é o estudo de tais sistemas. Cada componente de um sistema móvel é chamado de "unidade móvel". Para garantir entrega de dados a uma unidade móvel, é necessário continuar lhe enviando dados à medida que ela se move. Para que isso seja possível, é preciso detectar mudanças em sua localização [Roman et al., 2000].

No contexto de sistemas móveis, "localização" é um determinado ponto no espaço, e há dois tipos de espaços: físicos e lógicos. Mobilidade física é a mudança da posição física, concreta, das unidades móveis, seja em pequenas ou grandes distâncias. Mobilidade lógica, por outro lado, refere-se à mobilidade de unidades móveis de *software* entre diferentes *hosts* [Roman et al., 2000]. Através da mobilidade lógica, é possível transportar partes de uma aplicação (como, por exemplo, enviar atualizações para um programa anti-vírus) ou migrar processos complexos de um dispositivo para o outro [Zachariadis et al., 2004]. Sistemas com mobilidade física dividem-se em dois grandes grupos: sistemas de computação nômade e redes *ad hoc*. Sistemas de computação nômade são sistemas que dependem de uma rede principal fixa e são compostos de uma variedade de dispositivos móveis que conectam-se a suas estações-base, enquanto as redes *ad hoc* são compostas apenas de dispositivos móveis, que comunicam-se apenas enquanto estiverem alcançáveis uns aos outros [Roman et al., 2000].

2.2 Segurança de sistemas computacionais

No contexto de sistemas computacionais, a palavra "segurança" tem ao menos dois significados diferentes. Enquanto em língua inglesa há termos distintos para cada um desses significados, eles são representados pela mesma palavra em língua portuguesa. Para contornar essa ambiguidade, alguns autores sugeriram terminologias em língua portuguesa, como por exemplo [Verissimo and de Lemos, 1989] e [Rela, 2003], mas não há consenso sobre sua utilização. Nesta dissertação, escolheu-se utilizar os termos em inglês para resolver ambiguidades.

A segurança traduzida como *"security"* refere-se à prevenção de acessos indevidos e modificação de informações e recursos que afetariam a confidencialidade, a integridade e a disponibilidade do sistema [Farkhani and Razzazi, 2006]. A segurança traduzida como *"safety"* refere-se à garantia de que certo sistema executa corretamente suas funções sem causar danos a outros sistemas ou pessoas [Weber, 2003], mesmo quando ocorrem acidentes ou algum mau funcionamento do próprio sistema [Constantinescu and Vladoiu, 2013]. Neste documento, o termo "segurança" será utilizado no sentido de *"security"*, exceto quando o contrário for indicado explicitamente. A maioria dos termos relacionados à segurança no sentido de *"security"* utilizados neste documento foram traduzidos para o português de acordo com a terminolo-

[2] *"Internet of Things"* ou *IoT* em inglês.

gia sugerida pelo ITI[3] [ITI, 2007], exceto o termo "resumo criptográfico", que foi retirado de [Maziero, 2014].

A dependabilidade[4] é um conceito muito semelhante à segurança, e refere-se à garantia de bom funcionamento em situações adversas. Quando um sistema tem segurança e dependabilidade, diz-se que ele tem confiabilidade[5] [Burleson and Carrara, 2013]. A confiabilidade de um sistema também pode aumentar quando seus componentes são replicados por inteiro. Neste caso, as réplicas devem assumir as funções dos originais enquanto estes estiverem fora do ar [Emmerich, 2000].

Um sistema seguro deve ter um comportamento correto e bem definido mesmo na presença de adversários [Burleson and Carrara, 2013] e deve ser protegido contra atividades maliciosas, isto é, com objetivo de afetar a confidencialidade, a autenticidade, a integridade ou a irretratabilidade dos dados [Weber, 2003]. A confidencialidade[6] refere-se à garantia de que informações presentes nos sistemas não possam ser acessadas por agentes que não são autorizados a acessá-las. A integridade[7] refere-se à garantia de que dados ou partes do sistema não possam ser manipulados por agentes não autorizados a manipulá-los. A autenticação[8] refere-se à garantia de que apenas os agentes autorizados a agir junto ao sistema possam fazê-lo, ou, de forma equivalente, que os não autorizados não possam agir como se fossem autorizados. A irretratabilidade, também chamada de "irrefutabilidade" ou de "não-repudiação"[9], refere-se à garantia de que agentes sejam incapazes de negar sua relação com dados inseridos ou alterados por eles e também com atividades realizadas por eles. A disponibilidade[10] refere-se à garantia de que um sistema pode ser utilizado quando necessário [Jyostna and Padmaja, 2011], ou seja, à manutenção do funcionamento do sistema e dos serviços providos por ele sem interrupção devido a ações danosas de atores externos [Ravi et al., 2004], sejam elas intencionais ou não. Um sistema pode ser composto de vários componentes de *hardware* e *software*, e o sistema como um todo é tão seguro quanto seu elo mais fraco [Clulow, 2015].

As ferramentas de engenharia contemporâneas são adequadas para desenvolver sistemas complexos, mas não para garantir sua segurança. Metodologias de testes geralmente tentam garantir que não há falhas funcionais em sistemas, mas são ineficazes em verificar falhas de segurança. Abstrações de engenharia geralmente separam aplicações em diversas camadas, de forma que falhas de segurança das camadas inferiores costumam ser invisíveis para os desenvolvedores das camadas superiores [Ravi et al., 2004].

Atividades computacionais que são realizadas para garantir a segurança de um sistema são chamadas de "processamento de segurança"[11] [Ravi et al., 2004]. O processamento de segurança muitas vezes requer mais recursos computacionais do que um sistema tem a oferecer, levando a uma diferença entre o que se pode implementar e o que seria desejável em termos de segurança, chamada "disparidade de processamento de segurança"[12]. A disparidade de processamento de segurança é mais evidente em sistemas que precisam realizar um grande

[3] Instituto Nacional de Tecnologia da Informação, autarquia do governo federal brasileiro.
[4] *"Dependability"* em inglês.
[5] *"Thrustworthyness"* em inglês.
[6] *"Confidentiality"* em inglês.
[7] *"Integrity"* em inglês.
[8] *"Authentication"* em inglês.
[9] *"Non-repudiation"* em inglês.
[10] *"Availability"* em inglês.
[11] *"Security processing"* em inglês
[12] *"Security processing gap"* em inglês.

número de transações (como roteadores e servidores) e sistemas com pouca capacidade de processamento ou memória (como sistemas embarcados). A disparidade entre a energia necessária para realizar o processamento de segurança e a energia que um sistema pode fornecer chama-se “disparidade de bateria”[13]. Há uma disparidade entre as contramedidas de segurança e a forma com que são implementadas nos sistemas, formando a “disparidade de garantia”[14]. A disparidade de garantia é maior à medida que sistemas se tornam mais complexos, pois os desenvolvedores passam a ter mais dificuldade para garantir que não há vulnerabilidades em suas implementações [Ravi et al., 2004].

Quando vários sistemas precisam operar em conjunto, é comum que tenham de estabelecer alguma forma de comunicação entre si. A espionagem do tráfego na rede[15] ocorre quando um atacante intercepta a comunicação não-criptografada entre dois dispositivos, como mensagens de *e-mail* e mensagens instantâneas [Parameswaran and Wolf, 2008]. Um protocolo seguro ou protocolo de segurança[16] é uma sequência de passos seguida por dois ou mais sistemas para garantir confidencialidade, integridade, autenticação, irretratabilidade e disponibilidade durante a comunicação entre eles [Jyostna and Padmaja, 2011]. Algoritmos de criptografia são utilizados como parte destes protocolos seguros.

Um princípio básico da criptografia, conhecido como princípio de Kerckhoff, determina que um sistema deve ser seguro mesmo quando um atacante conhece tudo sobre o sistema, exceto as chaves de criptografia. Em outros termos, o sistema de criptografia não deve depender da obscuridade, pois ela é facilmente contornada, então é geralmente melhor empregar algoritmos amplamente utilizados e testados do que desenvolver seu próprio [Burleson and Carrara, 2013]. A criptografia provê segurança robusta contra ataques específicos, geralmente de natureza matemática, como é o caso de ataques de força bruta. Quando a criptografia começou a ser amplamente utilizada, atacantes passaram a focar-se em aspectos mais sutis e complexos dos sistemas para realizar seus ataques [Ravi et al., 2004].

2.3 Segurança em sistemas embarcados

Sistemas embarcados devem cada vez mais ser responsáveis por atividades críticas, tornando essencial desenvolver para eles soluções adequadas de segurança [Kermani et al., 2013]. Sistemas embarcados interagem com entes externos, e falhas de segurança podem causar danos a estes entes, como perda de propriedades ou mesmo morte [Koopman, 2004]. A segurança de sistemas computacionais convencionais é estudada e aplicada há bastante tempo, enquanto a segurança de sistemas embarcados é um tema que só passou a ser estudado com mais afinco recentemente. As fontes limitadas de energia e a baixa disponibilidade de memória fazem com que seja desafiador desenvolver soluções de segurança para tais sistemas [Ravi et al., 2004].

Sistemas embarcados são comumente dispostos em ambientes físicos inseguros, onde são mais expostos a ataques. Os desafios encontrados em sistemas embarcados podem incluir a implementação de mecanismos eficientes de criptografia e processamento de segurança e o desenvolvimento de *hardware* e *software* que resistam a ataques dos mais diversos tipos. Mecanismos funcionais (criptografia, protocolos seguros, biometria, ferramentas anti-*malware*, de-

[13] *“Battery gap”* em inglês.
[14] *“Assurance gap”* em inglês.
[15] *“Eavesdropping”* em inglês.
[16] *“Security protocol”* em inglês.

tecção de intrusão) desenvolvidos originalmente para computação de uso geral geralmente não são viáveis em sistemas embarcados [Kermani et al., 2013]. Como as soluções de segurança convencionais requerem muitos recursos, soluções híbridas de *software* e *hardware* foram propostas [Jyostna and Padmaja, 2011]. Devido à forma como alguns dispositivos são utilizados, simplesmente não faz sentido adicionar a eles certas soluções convencionais de segurança. Por exemplo, dispositivos médicos implantáveis devem permitir acesso de usuários ou sistemas não previamente autorizados em caso de emergências, o que dificulta ou impossibilita o uso de criptografia [Kermani et al., 2013].

O custo influencia a arquitetura de segurança em sistemas embarcados. O balanço entre os requisitos desejáveis de segurança e o custo de implementar soluções que os atendem é definido pelos desenvolvedores do sistema [Ravi et al., 2004]. Também há custos (geralmente elevados) quando é necessário intervir ativamente em sistemas devido a brechas de segurança, então é ideal que estes sistemas consigam reagir a ataques de forma autônoma [Kermani et al., 2013]. Por outro lado, projetos muito ambiciosos em termos de segurança podem sofrer com altos custos, tempos elevados de desenvolvimento e maior esforço de implementação [Ravi et al., 2004].

As melhores práticas usadas em computadores convencionais muitas vezes não são praticáveis em sistemas embarcados [Koopman, 2004]. As capacidades de processamento dos dispositivos embarcados podem ser sobrepujadas pelo processamento exigido pelas soluções tecnológicas escolhidas, reduzindo a vazão de dados do sistema [Ravi et al., 2004]. Em outros termos, um sistema embarcado pode não ser capaz de processar dados na velocidade desejada se gastar muito tempo de processamento realizando atividades relativas à segurança, como criptografia ou detecção de anomalias.

Dispositivos embarcados podem ser operados remotamente, e os mecanismos que permitem esta operação são um alvo potencial de ataques. Se os dispositivos forem conectados à *Internet*, por exemplo, os ataques que se aproveitam de vulnerabilidades da comunicação podem ser realizados a partir de qualquer lugar [Parameswaran and Wolf, 2008]. Um indivíduo mal-intencionado não precisa ter acesso físico a um sistema conectado à *Internet* para ser capaz de atacá-lo, então sistemas embarcados conectados à *Internet* são mais sujeitos a ataques [Jyostna and Padmaja, 2011]. Protocolos de segurança são utilizados para garantir comunicação segura com sistemas embarcados e a partir deles [Ravi et al., 2004].

Com a evolução da *Internet*, foram elaboradas diversas soluções para garantir autenticidade, confidencialidade, integridade e irretratabilidade em sistemas computacionais. Os algoritmos criptográficos e protocolos de segurança resolvem problemas de segurança de forma funcional, mas ignoram o fato de que sistemas embarcados são limitados pelo ambiente em que se encontram e pelos recursos que possuem [Ravi et al., 2004]. Em outros termos, as tecnologias desenvolvidas para computadores convencionais podem não satisfazer os requisitos de sistemas embarcados [Koopman, 2004]. Por exemplo, utilizar criptografia assimétrica requer muito processamento dedicado à cifragem e decifragem de dados, aumentando o consumo energético, o que nem sempre é possível devido a limitações do sistema embarcado [Kermani et al., 2013]. Como o crescimento de demandas energéticas é mais rápido que o crescimento de capacidades das baterias, torna-se necessário considerar o consumo energético ao elaborar soluções de segurança (como, por exemplo, protocolos de segurança otimizados ou *hardware* especializado para segurança) [Ravi et al., 2004].

Para a comunicação entre dispositivos, estabelecer chaves de criptografia individuais *a priori* é viável em redes pequenas, mas não é viável quando uma rede tem muitos disposi-

tivos. Para estabelecimento das chaves *a posteriori*, é possível utilizar um protocolo de troca de chaves, em que os participantes trocam informações públicas com as quais podem gerar as chaves, mas com as quais um terceiro não consegue [Jyostna and Padmaja, 2011].

A comunicação entre dispositivos pode passar por diversos pontos intermediários durante o roteamento. Para evitar que indivíduos mal-intencionados interfiram na comunicação, especialmente na troca de chaves, é possível assinar digitalmente os dados enviados. Através da assinatura digital, o destinatário de um dado pode confirmar a autenticidade das informações. A assinatura digital é gerada com base em uma chave privada de assinatura, e pode ser verificada com base na chave pública. Para utilizar assinatura digital, é preciso que o destinatário das informações saiba a chave pública do remetente. A chave pública do destinatário pode ser recebida de outros dispositivos, sempre assinada por autoridades mais confiáveis, em um esquema de certificação digital. A certificação digital envolve a assinatura das chaves por parte de autoridades certificadoras, cuja autenticidade pode ser certificada por outras autoridades [Jyostna and Padmaja, 2011].

A identificação de usuários é um requisito comum de segurança em sistemas embarcados, e refere-se à necessidade de validar usuários antes de permitir que usem o sistema. Tecnologias biométricas como identificação de voz ou impressões digitais podem ser utilizadas para verificar a identidade de usuários. Certificados digitais podem ser usados para identificar sistemas, enquanto assinaturas digitais podem ser utilizadas para autenticar a origem de dados e verificar sua identidade [Ravi et al., 2004]. O acesso seguro a redes é outro requisito de segurança, e refere-se a prover acesso a redes (como a *Internet*) apenas a dispositivos autorizados a fazê-lo [Ravi et al., 2004].

A susceptibilidade de sistemas embarcados a ataques requer que preservem a segurança mesmo quando são acessíveis logicamente ou fisicamente por indivíduos maliciosos [Ravi et al., 2004]. Sistemas embarcados podem armazenar dados sensíveis, que devem ser protegidos contra leitura e alteração por terceiros, o que resulta em um tipo armazenamento de dados denominado “armazenamento seguro”[17] [Jyostna and Padmaja, 2011]. O armazenamento seguro refere-se à garantia de confidencialidade e integridade dos dados armazenados no sistema. O armazenamento seguro de dados e a execução segura de códigos podem ser garantidas por diversas técnicas de *hardware* e *software*. A segurança de conteúdo refere-se a preservar as restrições determinadas para acesso aos conteúdos armazenados ou acessados pelo sistema embarcado, e pode ser um requisito de segurança. Protocolos de gestão de direitos digitais[18] podem ser utilizados para proteger conteúdos contra usos não-autorizados [Ravi et al., 2004].

2.3.1 Ataques a sistemas embarcados

Ataques a sistemas embarcados podem ser classificados pelo objetivo do ataque ou pela forma com que ele é realizado. Os objetivos de um ataque podem ser ferir a privacidade, a integridade ou a disponibilidade do sistema [Parameswaran and Wolf, 2008]. Ataques a sistemas embarcados podem pertencer a dois grandes tipos: ataques lógicos e ataques físicos [Ravi et al., 2004].

Um atacante pode querer roubar informações armazenadas em um sistema embarcado ou inserir nele informações falsas, o que é facilitado se ele estiver ao alcance do atacante

[17] *“Secure storage”* em inglês.
[18] *“Digital rights management”* ou *DRM* em inglês.

[Parameswaran and Wolf, 2008]. A resistência contra adulterações[19] refere-se à proteção de um sistema contra ataques lógicos e físicos [Ravi et al., 2004].

Há outro grupo de ataques denominados “ataques de canal lateral”. Apesar de alguns autores considerarem que estes e os ataques físicos pertencem ao mesmo grupo [Ravi et al., 2004], ataques de canal lateral também podem ser feitos unicamente com *software* [Lawson, 2009]. Assim, pareceu mais adequado considerar que ataques de canal lateral são uma classificação à parte (isto é, ataques físicos e lógicos podem ser também ataques de canal lateral). Em linhas gerais, estes ataques são baseados na observação de “canais laterais” por parte dos atacantes. Canais laterais são efeitos colaterais observáveis de algum processo computacional, que podem ser medidos e talvez influenciados por atacantes [Lawson, 2009].

Ataques lógicos

Ataques lógicos podem ser de *software*, em que se exploram falhas na arquitetura de *software*, e de criptografia, em que se exploram falhas no desenvolvimento de algoritmos de criptografia ou de protocolos de segurança. Alguns problemas de implementação que levam a ataques lógicos são falhas nos mecanismos de atualização de código e estouros de *buffer*[20], falhas nos protocolos criptográficos, fraquezas nas negociações de parâmetos de segurança, entre outros [Ravi et al., 2004].

Ataques de *software* aproveitam-se de fraquezas de códigos supostamente confiáveis, como sistema operacional, *middleware* e aplicações [Jyostna and Padmaja, 2011]. Ataques de intrusão pela rede são ataques de *software* feitos através da rede de comunicação à qual o sistema embarcado está conectado, como por exemplo ataques de estouro de *buffer* [Parameswaran and Wolf, 2008]. Ataques criptográficos aproveitam-se de fraquezas de um sistema criptográfico para, por exemplo, obter senhas que podem ser utilizadas para acessar algum sistema [Parameswaran and Wolf, 2008].

Ataques de *software* geralmente buscam fazer com que um dispositivo execute códigos maliciosos ou revele dados sensíveis, como chaves de criptografia [Ravi et al., 2004]. Ataques de *software* podem corromper o código e os dados do sistema [Jyostna and Padmaja, 2011]. Em sistemas que permitem executar aplicações recebidas a partir da rede, é possível haver ataques lógicos em que aplicações maliciosas exploram vulnerabilidades dos sistemas operacionais e prejudicam sua integridade, o que pode levar ao roubo de informações confidenciais e à negação de serviço[21] [Ravi et al., 2004]. Os ataques de injeção de código são ataques de *software* que buscam acabar com a integridade da aplicação sendo executada, forçando alterações no fluxo de execução e nas instruções que formam o programa. Outros tipos de ataques de *software* são baseados em estouros de *buffer* na pilha, estouros de *buffer* na *heap*, dupla liberação de memória, estouros de inteiro e exploração de vulnerabilidades em *strings* de formatação [Parameswaran and Wolf, 2008].

Segundo a pesquisa em [UBM Tech, 2015], entre 2010 e 2015, as linguagens de programação mais utilizadas em sistemas embarcados eram *C*, *C++* e *assembly*, totalizando juntas aproximadamente 85% dos projetos. Linguagens deste tipo operam quase diretamente no *hardware*, e portanto não implementam mecanismos implícitos de segurança, como verificação automática de índices em vetores. As aplicações desenvolvidas nessas linguagens costumam

[19] *“Tamper resistance”* em inglês.
[20] *“Buffer overflows”* em inglês.
[21] *“Denial of Service”* ou *DoS* em inglês.

ser suscetíveis a ataques de *software*, que exploram a falta desses mecanismos. Tais ataques são divididos em dois grandes grupos: ataques ao sigilo de informação e ataques à integridade. Vulnerabilidade a ataques de *software* não é uma exclusividade de sistemas embarcados, mas é um problema muito comum neles [Silva et al., 2013].

Ataques ao sigilo da informação podem ocorrer de duas formas: através de vazamento de endereços[22] e de vazamento de dados[23]. Ataques de vazamento de dados são aqueles em que o atacante consegue acessar dados da aplicação, aos quais presumivelmente não deveria ter acesso. O vazamento é forçado para obter alguma informação que possa utilizar no futuro para comprometer o funcionamento do sistema ou para obter dados sigilosos [Silva et al., 2013].

Os ataques de vazamento de endereços ocorrem quando um atacante descobre em que regiões da memória estão carregados dados ou código. Os mecanismos de proteção adotados para evitar este problema são geralmente impostos pelo sistema operacional, como aleatorização de espaço de endereços[24] e prevenção contra a execução de dados[25] [Silva et al., 2013]. Estes mecanismos costumam ter um custo computacional elevado, então não são comumente utilizados em sistemas embarcados.

Em ataques à integridade, um atacante manipula os dados em regiões específicas de memória, com o objetivo de modificá-los ou alterar o fluxo de execução do programa. Ataques à integridade são bastante populares e comuns, e podem ser de dois tipos: ataque de estouro de inteiro[26] e de estouro de *buffer*[27]. [Silva et al., 2013].

Os ataques de estouro de *buffer* são causados pela falta de controle dos índices dos elementos nos *buffers*. Nas linguagens de baixo nível, *buffers* de dados são representados apenas como ponteiros na memória, então não há controle explícito de onde os *buffers* começam e terminam. Desta forma, é possível escrever em regiões de memória anteriores ou posteriores ao *buffer* de dados se forem utilizados índices inválidos, isto é, negativos ou superiores ao tamanho do *buffer*. Ataques de estouro de *buffer* na pilha, por exemplo, podem permitir que o atacante altere o endereço de retorno de uma função, dando a ele domínio sobre o fluxo de controle do programa. Uma forma de tentar evitar problemas de estouro de *buffer* na pilha é através de "canários de pilha"[28], que são valores inseridos pelo compilador e verificados internamente para garantir a integridade [Silva et al., 2013]. Um *buffer* pode estar na pilha, na memória estática ou na *heap*. Em qualquer destes casos, é possível que nas regiões próximas a ele estejam armazenadas outras variáveis. Portanto, conhecimento prévio sobre o *layout* da memória antes do ataque permite que um atacante sobrescreva variáveis específicas de um programa.

Os ataques de estouro de inteiro são causados pela combinação de dois aspectos importantes das variáveis inteiras em linguagens de programação tradicionais: as variáveis têm tamanho fixo e operam de forma modular. A aritmética modular em variáveis de tamanho fixo significa que o maior valor representável acrescido de um resulta no menor valor representável. Por exemplo, em uma variável inteira de 16 *bits* sinalizada, a operação $32767+1$ resulta em -32768. O comportamento modular é útil em algumas situações, como para implementar funções de resumo criptográfico (*hash*). Por outro lado, esta característica também pode ser utili-

[22] *"Address leak"* ou *"program data leak"* em inglês.
[23] *"Data leak"* em inglês.
[24] *"Address space layout randomization"* ou *ASLR* em inglês.
[25] *"Data execution prevention"* ou *DEP* em inglês.
[26] *"Integer overflow"* em inglês.
[27] *"Buffer overflow"* em inglês.
[28] *"Stack canaries"* em inglês.

zada para realizar ataques de estouro de *buffer*, como por exemplo quando se altera uma variável que indica o tamanho de um bloco de memória alocado dinamicamente [Silva et al., 2013].

Proteção

Técnicas como revisão de código e *sandboxing* ajudam a evitar ataques lógicos [Ravi et al., 2004]. Como os endereços de retorno são os alvos mais comuns de ataques de estouro de *buffer*, foram desenvolvidas técnicas que usam o *hardware* para proteger estes dados [Parameswaran and Wolf, 2008]. Para proteger um sistema contra ataques lógicos, é necessário garantir a privacidade e a integridade de códigos sensíveis durante todas as etapas da execução [Ravi et al., 2004].

Algumas linguagens de programação de alto nível como *Java* e *ML* garantem a integridade dos códigos durante a execução, mas não é assim com as linguagens de mais baixo nível. Entretanto, há dialetos de *C* e *C++* que evitam os problemas descritos anteriormente utilizando restrições no gerenciamento de memória [Parameswaran and Wolf, 2008].

Para proteger um sistema contra ataques lógicos, é preciso identificar e remover *bugs* e falhas arquiteturais que tornam o sistema suscetível a ataques [Ravi et al., 2004]. Analisadores estáticos de código[29], são ferramentas que buscam encontrar falhas em *software* através de análise do código antes da execução. Analisadores estáticos simples apenas verificam linhas individuais de código e declarações, mas há analisadores mais complexos, cujas saídas podem ser utilizadas para provar matematicamente propriedades de programas [Parameswaran and Wolf, 2008].

Em oposição aos analisadores estáticos, que verificam códigos apenas antes da execução, existem os analisadores dinâmicos, que verificam códigos durante a execução. Para que isto seja possível, estes analisadores fazem instrumentação dos códigos durante a compilação e realizam testes práticos para detectar vulnerabilidades. Enquanto é possível obter mais informações a partir de analisadores dinâmicos do que a partir de analisadores estáticos, é possível também que alguns erros não sejam detectados por não serem contemplados nos casos de teste [Parameswaran and Wolf, 2008].

As análises tradicionais examinam blocos individuais de código, ignorando as estruturas dinâmicas presentes em sistemas embarcados conectados à *Internet*. Em [Silva et al., 2013] sugere-se um terceiro tipo, a análise distribuída, para verificar sistemas em que há diversos dispositivos envolvidos [Silva et al., 2013].

Alguns sistemas analisam o comportamento de aplicações em execução para detectar atividades maliciosas, num procedimento denominado "detecção de anomalias baseada no comportamento"[30]. Uma maneira de realizar esta atividade é definir o que são comportamentos corretos, e então tentar detectar situações diferentes do definido. A desvantagem deste modelo é o grande número de falsos positivos, isto é, casos em que ações legítimas são classificadas como maliciosas. Este problema é mais evidente quando há modificações no *software* da aplicação, que pode incluir atividades legítimas mas que não haviam sido previstas anteriormente. Neste caso, essas atividades serão provavelmente detectadas como incorretas, quando na verdade não são [Parameswaran and Wolf, 2008].

É possível adicionar mecanismos de segurança transparentemente a programas com ajuda do compilador, que pode inserir mecanismos de proteção de pilha, proteção a ponteiros de

[29] *"Static code analyzers"* em inglês.

[30] *"Behavior-based anomaly detection"* em inglês.

programa e até checagem de limites dos *buffers*. Uma alternativa também é utilizar bibliotecas que realizem atividades diversas (como manipulação de *strings*) de forma mais segura. Outra solução para dificultar a realização de ataques lógicos é separar código e dados em setores diferentes da memória, e impedir escritas no setor em que fica o código executável, o que depende de suporte de *hardware* a estas funcionalidades [Parameswaran and Wolf, 2008].

Ataques físicos

Ataques físicos aproveitam-se de características de implementações de *hardware* ou buscam obter informações e dados sigilosos a partir de suas características [Ravi et al., 2004]. Quando um atacante tem acesso físico ao sistema embarcado, por exemplo, pode interferir na comunicação entre a unidade central de processamento[31] e seus periféricos para lançar ataques [Jyostna and Padmaja, 2011]. Sistemas embarcados são sujeitos a ataques físicos como ataques pelo barramento, análises temporais, indução de falhas, análises energéticas, análises eletromagnéticas [Ravi et al., 2004], *microprobing*, engenharia reversa e espionagem de tráfego.

Ataques físicos são geralmente divididos em dois grandes grupos: ataques invasivos e não invasivos. Ataques invasivos envolvem observação, manipulação e interferência nos aspectos internos do *hardware*, e podem ser difíceis de realizar. Exemplos são *microprobing* e engenharia reversa. Ataques não invasivos não requerem a abertura dos sistemas por parte dos atacantes [Ravi et al., 2004] e envolvem observar características de um sistema enquanto são realizados procedimentos críticos, como operações criptográficas [Parameswaran and Wolf, 2008]. Exemplos de ataques não invasivos são análises temporais, indução de falhas, análises energéticas, análises eletromagnéticas [Ravi et al., 2004].

Um atacante pode danificar ou confundir os sensores ou outros periféricos (como memória ou atuadores) do sistema embarcado, o que faz com que sejam operados de forma incorreta. Ataques de drenagem de energia, também chamados de "ataques de exaustão"[32] são ataques nos quais um atacante busca forçar o equipamento embarcado a consumir mais energia que o necessário. Uma forma de realizar ataques deste tipo é forçar um aumento da carga computacional, reduzir ciclos de *clock* ou aumentar o uso de sensores e outros periféricos. Aquecimento ou resfriamento excessivos do sistema embarcado também podem danificá-lo [Parameswaran and Wolf, 2008].

A exposição de dados através de barramentos de memória é uma vulnerabilidade de sistemas embarcados. Uma forma óbvia de dificultar ataques físicos é melhorar os invólucros do sistema [Jyostna and Padmaja, 2011].

Uma alternativa de *hardware* para aumentar a segurança é a utilização de sistemas seguros em um único *chip*[33], em que há uma *ROM* e uma *RAM* internas no próprio *chip* do sistema, em que são armazenados códigos seguros, dados sensíveis e valores intermediários de cálculos criptográficos. Em sistemas seguros em um único *chip*, não são expostos dados ou códigos críticos nos barramentos externos, dificultando o acesso a informações pelos atacantes. Geralmente apenas processos com privilégios especiais podem ter acesso à *RAM* interna do sistema seguro em um único *chip*, já que nela são armazenados dados sensíveis. Por exemplo, dados armazenados nas *ROMs* externas, como chaves de criptografia utilizadas na comunicação, podem ser cifrados utilizando chaves armazenadas na *ROM* interna [Jyostna and Padmaja, 2011].

[31] *"Central processing unit"* ou *CPU* em inglês.

[32] *"Energy drainage"* ou *"exhaustion attack"* em inglês.

[33] *"Secure System on a Chip"* ou *"Secure SoC"* em inglês.

Ataques não invasivos permitem a atacantes roubar informações secretas sem deixar traços. Estes ataques são feitos inserindo diversas entradas no sistema e capturando canais laterais (medidas de consumo energético, tempo de processamento ou emissões eletromagnéticas) durante processamentos importantes, em particular o processamento criptográfico. Um atacante pode correlacionar estas medidas com as computações internas, o que pode fornecer informações sobre as chaves de criptografia [Parameswaran and Wolf, 2008].

Ataques baseados em análise temporal[34] permitem que atacantes determinem chaves (ou características das chaves) de criptografia analisando variações de tempo de um sistema para realizar computações de segurança. Contramedidas comuns para lidar com ataques de análise temporal, como quantizar o tempo utilizado para processamento, geralmente não previnem o ataque, apenas dificultam-no [Ravi et al., 2004].

Também é possível obter informações sobre chaves criptográficas analisando o consumo energético, que varia de acordo com as operações realizadas pelo processador nos chamados "ataques de análise energética"[35] [Ravi et al., 2004]. Os ataques de análise energética são os ataques não invasivos mais utilizados [Parameswaran and Wolf, 2008] e podem ser de dois tipos: estáticos e dinâmicos. Ataques de análise energética estáticos tentam determinar as chaves de criptografia diretamente a partir do perfil energético das computações criptográficas [Ravi et al., 2004]. Em ataques de análise energética dinâmica, tenta-se extrair informações das chaves de criptografia com base em leituras de consumo energético. Neste tipo de ataque. são fornecidas diversas entradas para o dispositivo e analisado o consumo energético para as diversas computações criptográficas, então são utilizados métodos estatísticos para determinar as chaves. As contramedidas para lidar com ataques de análise energética dinâmica muitas vezes são matematicamente complexas e pouco intuitivas, e também podem ser patenteadas [Ravi et al., 2004].

Uma contramedida utilizada para evitar ataques não invasivos é injetar ruído durante a execução de código, fazendo as medidas do atacante oscilarem artificialmente. Outra técnica utilizada é separar a exponenciação modular (quando este cálculo é utilizado nos algoritmos criptográficos) em janelas, de forma que o tamanho das janelas seja definido aleatoriamente [Parameswaran and Wolf, 2008].

Também é possível utilizar circuitos de supressão de sinais, reduzindo a razão sinal-ruído e dificultando que o atacante diferencie o ruído dos dados em processamento. Outras técnicas podem ser usadas para evitar estes ataques, como execução fora de ordem, aleatorização do sinal de *clock* e uso de instruções com assinatura energética independente dos dados [Parameswaran and Wolf, 2008].

2.3.2 Estudos de caso

Nesta seção, estão apresentados estudos de caso referentes à segurança de classes específicas de dispositivos embarcados. O principal objetivo desta pesquisa foi mostrar como os ataques e técnicas de segurança descritas anteriormente ocorrem em aplicações práticas. Para a pesquisa, consideraram-se duas classes de sistemas embarcados: dispositivos médicos e dispositivos integrados a serviços baseados na localização. Estas duas classes de dispositivos foram selecionadas pois executam em ambientes bastante diferentes entre si, e apresentam desafios

[34] *"Timing analysis attacks"* em inglês.
[35] *"Power analysis attacks"* em inglês.

diferentes aos projetistas. Muitos dos desafios encontrados nestes dois tipos de sistemas podem ser encontrados em outras áreas de aplicação.

Dispositivos médicos implantáveis

Dispositivos médicos implantáveis e trajáveis[36] são equipamentos utilizados para diagnosticar, monitorar e tratar condições médicas [Kermani et al., 2013]. Dispositivos deste tipo podem realizar uma variedade de funções terapêuticas e salva-vidas, desde infusão de drogas e captura de batimentos cardíacos até neuro-estimulação direta [Burleson and Carrara, 2013]. Os sensores presentes em equipamentos deste tipo podem capturar sinais extremamente ruidosos, pois os pacientes que os utilizam podem mover-se livremente. Assim, muitas vezes é necessário utilizar algoritmos sofisticados para pré-processamento e filtragem das medições [Konstantas, 2007]. Alguns tipos de marca-passos, desfibriladores cardíacos, bombas de insulina e neuroestimuladores são exemplos de dispositivos médicos implantáveis [Gollakota et al., 2011].

Tais equipamentos surgiram devido ao interesse de monitorar condições de pacientes por períodos extensos de tempo [Bonato, 2003] e fora do hospital [Konstantas, 2007]. Enquanto acompanhar condições de um paciente por um curto período pode ser feito sem sistemas deste tipo, eles permitem que o monitoramento seja realizado de forma economicamente viável por semanas ou meses, o que pode ser necessário em diversos casos [Bonato, 2003]. Os dispositivos médicos implantáveis e trajáveis são alternativas aos sistemas convencionais de *home-care* [Konstantas, 2007].

Dispositivos médicos implantáveis viabilizam modalidades de medicina personalizada que eram idealizadas há muito tempo, mas não eram possíveis devido a limitações tecnológicas. Ao contrário de diversos objetos na *Internet* das Coisas, os dispositivos médicos implantáveis realizam operações sensíveis com seus sensores e atuadores, então requerem segurança no sentido de *security* com mais ênfase que outros tipos de sistemas devido a questões de saúde e segurança no sentido de *safety* [Burleson and Carrara, 2013].

Dispositivos médicos trajáveis podem ser dispositivos independentes com elementos para fixação ao corpo ou podem ser colocados em peças de roupa como camisetas, camisas, vestidos ou calças. Dispositivos médicos implantáveis são inseridos sob a pele ou em locais específicos dentro do corpo, como na boca ou no joelho. Como nos dispositivos implantáveis há contato direto com o corpo, eles geralmente realizam leituras de sinais fisiológicos de forma mais precisa que os dispositivos trajáveis [Konstantas, 2007].

Os dispositivos médicos muitas vezes precisam transmitir os dados coletados para sistemas de controle ou armazenagem, que podem estar no mesmo equipamento (o que é menos comum) ou em equipamentos separados (o que é mais comum). Quando há um dispositivo implantado ou trajado e um sistema separado de controle ou armazenagem, é necessário que estes dois sistemas comuniquem-se [Konstantas, 2007]. É comum que dispositivos deste tipo tenham interfaces de conexão sem fio usadas para comunicação com equipamentos externos de monitoramento ou diagnóstico e para a reprogramação das terapias realizadas [Kermani et al., 2013]. É possível que as leituras dos dispositivos médicos sejam transmitidas para sistemas externos de bancos de dados, nos quais estes dados são armazenados para consulta futura [Bonato, 2003].

Devido às funções críticas que realizam, os dispositivos implantáveis passam por um processo rigoroso de avaliação para verificar se atendem aos requisitos regulatórios de efetivi-

[36] *"Implantable and wearable medical devices"* ou *IWMD* em inglês.

dade e segurança no sentido de *safety*. Em contrapartida, a segurança no sentido de *security* é um tema relativamente novo em termos de requisitos regulatórios [Burleson and Carrara, 2013]. À medida que estes equipamentos tornam-se mais complexos, tornam-se cada vez mais sujeitos a ataques [Kermani et al., 2013].

Problemas de *software* (ou *"bugs"*) são comuns em sistemas em geral, mas são especialmente problemáticos em sistemas implantáveis, já que podem fazer com que *recalls* sejam necessários. No caso de um *recall*, pode ser necessário trocar de equipamentos que já foram implantados em pacientes, o que pode requerer intervenção cirúrgica. Para evitar a ocorrência de *bugs*, é comum que a implementação de dispositivos médicos implantáveis seja limitada às funcionalidades estritamente necessárias [Gollakota et al., 2011].

Apesar disto, equipamentos deste tipo tornam-se cada vez mais complexos, com conectividade e reprogramabilidade, o que melhora o monitoramento e a terapia que realizam, além de trazer maior comodidade para os pacientes [Kermani et al., 2013]. Espera-se que futuramente equipamentos deste tipo conectem-se formando uma rede de equipamentos ligados ao corpo, as chamadas "redes de área de corpo"[37] [Kermani et al., 2013]. A implementação destes equipamentos requer uma miniaturização dos sensores e do restante do *hardware* [Konstantas, 2007].

Alguns dispositivos médicos implantáveis podem funcionar por até 10 anos depois de implantados, e trocá-los requer um procedimento cirúrgico que pode levar a inúmeras complicações [Gollakota et al., 2011]. Como a troca das fontes energéticas pode requerer intervenção cirúrgica, espera-se que tais equipamentos fiquem anos operando com baterias de tamanho limitado. A menor disponibilidade energética impõe restrições severas à memória e ao processamento destes sistemas. Assim, utilizar soluções convencionais de segurança em tais equipamentos é praticamente impossível. Mesmo em relação a outras classes de dispositivos embarcados, os dispositivos médicos implantáveis têm restrições muito severas de recursos [Kermani et al., 2013].

Dispositivos médicos implantáveis e trajáveis são um dos tipos de sistemas embarcados mais críticos em termos de segurança, pois executam tarefas que envolvem riscos de vida ou de saúde. Ataques bem-sucedidos a sistemas deste tipo podem ter consequências severas, desde perder privacidade até arriscar a vida dos pacientes [Kermani et al., 2013]. Dispositivos médicos implantáveis, como outros dispositivos embarcados, são sujeitos a ataques de exaustão de bateria [Burleson and Carrara, 2013]. Um ataque que force um consumo maior de bateria faz com que seja necessário trocar cirurgicamente o equipamento antes do necessário [Kermani et al., 2013].

O canal de comunicação sem fio de dispositivos médicos implantáveis comumente não utiliza criptografia, então está sujeito a ataques que comprometem a confidencialidade e a segurança no sentido de *safety* destes equipamentos [Gollakota et al., 2011]. A comunicação segura entre equipamentos embarcados comumente utiliza criptografia simétrica (para preservar confidencialidade), autenticação de mensagens (para garantir a integridade) e criptografia assimétrica (para autenticação e troca de chaves). Em particular, a criptografia assimétrica requer muito processamento e memória, e portanto muita energia, o que é um fator proibitivo para seu uso [Kermani et al., 2013]. Outro fator complicador para adicionar segurança a dispositivos deste tipo é o grande número de dispositivos médicos já implantados. Modificar equipamentos já implantados para adicionar suporte a criptografia é inviável na maioria dos casos devido a limitações de processamento ou memória, então seria necessário trocá-los para adici-

[37] *"Body area networks"* ou *BANs* em inglês.

onar suporte a criptografia. Outro motivo que inviabiliza ou dificulta o uso de criptografia é a eventual necessidade de profissionais da saúde terem acesso imediato ao dispositivo implantado ao atender o paciente. Este acesso imediato pode ser impossibilitado se o médico ou hospital não tiverem acesso à chave de criptografia, o que pode ocorrer em diversos casos, como por exemplo quando o paciente está inconsciente ou quando vai a um hospital diferente do habitual [Gollakota et al., 2011]. Uma solução para o problema de a criptografia impedir o acesso do dispositivo em situações emergenciais é fazer com que os pacientes carreguem cartões ou braceletes com as chaves dos dispositivos, ou que elas sejam gravadas sobre a pele usando pigmentos ultravioletas, que normalmente são invisíveis [Kermani et al., 2013].

As consequências de ataques a sistemas médicos implantáveis e trajáveis podem ser muito mais severas que em outras classes de sistemas embarcados. Dispositivos médicos são sujeitos a ataques pelo canal de comunicação sem fio, que podem expor dados confidenciais, prover entradas incorretas e até reconfigurar o dispositivo médico para fins maliciosos. A reconfiguração de um equipamento médico por um atacante com a intenção de causar dano ao paciente pode ser feita aplicando algum tratamento quando não for necessário ou não o aplicando quando necessário. Com conhecimento do protocolo utilizado pelos equipamentos, ainda é possível realizar ataques de negação de serviço para evitar a comunicação entre o equipamento e outros sistemas ou para forçar um consumo maior de bateria. Em qualquer destes casos, o atacante consegue comprometer a disponibilidade do sistema, já que este pode não ser capaz de atuar quando necessário [Kermani et al., 2013].

Uma solução de segurança proposta para sistemas médicos é limitar a comunicação entre equipamentos para que seja de baixo alcance. Esta solução é efetiva em evitar grande parte dos ataques que usam o canal sem fio, já que é necessária proximidade física ou usar antenas de alto ganho. Outra solução semelhante seria utilizar de alguma forma o próprio corpo humano como canal de comunicação. Em qualquer dos casos, a proximidade física de um atacante continua representando perigo para o sistema [Kermani et al., 2013].

Serviços baseados na localização

Com os avanços tecnológicos em tecnologias de posicionamento, é possível desenvolver equipamentos que determinam a posição de indivíduos [Gruteser and Liu, 2004]. Equipamentos que utilizam tecnologias de posicionamento como *GPS* para determinar sua localização atual são denominados "dispositivos detectores de localização"[38] [Mokbel, 2007]. Equipamentos deste tipo podem prover serviços personalizados de acordo com sua localização atual, formando os chamados "serviços baseados na localização"[39] [Gruteser and Liu, 2004].

Quanto ao tratamento dos posicionamentos determinados pelos dispositivos detectores de localização, os sistemas baseados em localização são divididos em três categorias: sistemas cientes da localização, sistemas de consultas esporádicas e sistemas de rastreio. Sistemas cientes da localização apenas têm ciência de sua localização atual[40], o que é utilizado internamente para provimento de algum serviço sem ser enviado para servidores ou outros equipamentos. Exemplos de sistemas deste tipo são navegadores *GPS off-line*. Sistemas de consultas esporádicas utilizam a localização apenas quando necessário, como referência para obtenção de informações junto a outros sistemas. Por exemplo, aplicações que procuram pontos de inte-

[38] *"Location-detection devices"* em inglês.
[39] *"Location-based services"* ou *LBS* em inglês.
[40] *"Position awareness"* em inglês.

resse próximos utilizando a *Internet* são serviços de consultas esporádicas. Sistemas de rastreio são aqueles que fazem consultas frequentes à localização atual e armazenam estas informações localmente ou enviam-nas para servidores externos [Gruteser and Liu, 2004]. Exemplos de sistemas deste tipo são rastreadores de pessoas [Bancroft et al., 2012] e rastreadores veiculares [Gruteser and Liu, 2004].

Alguns sistemas de rastreio armazenam as informações localmente, na memória do próprio equipamento embarcado, no chamado "rastreio passivo". Os dados armazenados por sistemas de rastreio passivo podem ser analisados após sua transferência para um computador. Outros sistemas enviam autonomamente as informações de rastreio para servidores centrais, no chamado "rastreio ativo". Sistemas de rastreio ativo são normalmente utilizados para rastreio de veículos (monitoramento de frota) e de pessoas [Bertagna, 2010].

A preocupação com privacidade dificulta a aceitação de tais sistemas por parte dos usuários [Gruteser and Liu, 2004]. Tais sistemas são vistos como inseguros, devido ao risco de indivíduos não autorizados terem acesso à localização dos dispositivos em tempo real. Se um usuário utilizar um sistema deste tipo e desejar manter sua localização privativa, em geral é preciso desligar o dispositivo detector de localização ou desabilitar os serviços baseados em localização, mas nem sempre isto é possível [Mokbel, 2007]. No caso de sistemas de rastreio, os usuários têm preocupação com questões de privacidade, o que varia de acordo com a situação: em certas ocasiões, os usuários não desejam que terceiros saibam onde eles estão [Gruteser and Liu, 2004].

O principal mecanismo de posicionamento utilizado em dispositivos detectores de localização é o *GPS*[41], originalmente desenvolvido pelo Departamento de Defesa dos Estados Unidos da América[42]. Nestes sistemas, é usado o princípio da trilateração para determinar a posição na superfície da terra. Aparelhos celulares costumam ter módulos de *GPS* integrados [Bertagna, 2010], então podem ser utilizados como plataforma para serviços baseados em localização.

Há uma infinidade de sistemas baseados na localização disponíveis, como é o caso de diversos aplicativos para telefones celulares. Dois tipos em particular serão discutidos em maior detalhes a seguir: sistemas de rastreio veicular e sistemas de rastreio de sentenciados.

A redução de custos de produtos e serviços e a crescente capacidade computacional e miniaturização do *hardware* embarcado tornou possível o desenvolvimento de sistemas de rastreio veicular. Geralmente tais sistemas são conectados a outros dispositivos e sensores do próprio veículo, tornando possível a coleta de uma variedade de dados a respeito de seu funcionamento [Constantinescu and Vladoiu, 2013].

Rastreadores veiculares são formados por um módulo de rastreio, um *hardware* embarcado especializado que é conectado a outros sistemas veiculares, um módulo opcional de armazenamento e um módulo de comunicação. Exemplos de módulos de comunicação utilizados são modems *GSM/GPRS*, módulos *Bluetooth* ou outros rádios operando nas bandas *ISM*[43]. Geralmente os dados coletados por rastreadores veiculares são enviados a servidores de monitoramento para armazenamento e análise. Rastreadores veiculares costumam utilizar *GPS* como tecnologia para provimento de localizações [Constantinescu and Vladoiu, 2013].

---

[41] *"Global Positioning System"*, que significa "sistema de posicionamento global".

[42] *"United States Department of Defense"* ou *U.S. D.o.D.* em inglês.

[43] Abreviação de *"Industrial, Scientific and Medical radio"*, que significa "rádio Industrial, Científico e Médico".

Há diversos tipos de rastreadores veiculares. Os sistemas de localização automática de veículos, a localização geográfica de veículos é transmitida constantemente a servidores de monitoramento através de redes móveis como *GSM* ou *GPRS*, e às vezes aos usuários através de mensagens *SMS*[44]. Neste tipo de rastreador, outras tecnologias de posicionamento são utilizadas quando a localização por *GPS* é imprecisa ou não está disponível, como por exemplo a navegação inercial. Nos sistemas de localização veicular ativada por eventos, as posições do veículo só são capturadas e armazenadas quando a segurança do veículo ou do sistema de rastreamento são violadas. Estes sistemas enviam notificações (isto é, alarmes) a servidores centrais quando ações suspeitas acontecem para que reações sejam tomadas, como por exemplo seguir o veículo ou alertar as autoridades. Sistemas gravadores de dados de eventos são aqueles em que diversas informações são armazenadas para auditoria futura, como ocorre em caixas-pretas de aviões. Estes sistemas guardam informações detalhadas para análise futura em caso de acidentes, e geralmente utilizam suas memórias de forma cíclica (isto é, quando a memória fica cheia, dados mais antigos começam a ser substituídos pelos mais recentes) [Constantinescu and Vladoiu, 2013].

A detecção e a notificação de atividades indesejadas ou maliciosas com o veículo monitorado é uma característica desejável a todos os tipos rastreadores veiculares. Geralmente, indivíduos mal intencionados tentam danificar os rastreadores para que não operem corretamente. No caso de rastreadores colocados em um carro de uso pessoal pelo proprietário, não se espera que o dono danifique intencionalmente o sistema de rastreio, mas é possível que outros indivíduos (por exemplo, ladrões) façam-no. Já no caso de rastreadores colocados pelo proprietário para monitorar e punir ações indesejadas de outros motoristas, como é o caso em frotas corporativas, é possível que o próprio condutor tente prejudicar a operação do dispositivo de rastreio (por exemplo, se quiser usar o carro da empresa para fins pessoais). Uma preocupação neste tipo de dispositivo também é proteger a privacidade dos dados transmitidos, para evitar que outros tenham acesso à localização dos veículos, o que pode ser perigoso [Constantinescu and Vladoiu, 2013], por exemplo, se a pessoa que tiver acesso a esta informação quiser causar algum dano ao veículo, a seus condutores ou a seus proprietários.

Outra tecnologia de rastreio que ganhou popularidade recentemente foi o monitoramento eletrônico de sentenciados, como uma alternativa à prisão em regime fechado [Danielsson and Makipaa, 2012]. O monitoramento é feito através de equipamentos fixados ao corpo do sentenciado, geralmente ao tornozelo, nas denominadas “tornozeleiras de monitoramento eletrônico” ou apenas “tornozeleiras eletrônicas”[45]. Equipamentos semelhantes podem ser utilizados também por militares, policiais ou bombeiros para auxiliar na coordenação de esforços em situações críticas [Bancroft et al., 2012].

O uso destes equipamentos varia bastante de um lugar para o outro, sendo usado desde casos em que indivíduos perigosos ganham liberdade até casos em que indivíduos pouco perigosos recebem penas alternativas em que o aprisionamento não é necessário. O objetivo destes equipamentos é reduzir os riscos à segurança pública, e seu uso é eficaz em reduzir o número de violações a restrições judiciais e a reincidência de crimes, oferecendo portanto um potencial de reabilitação. Da mesma forma que rastreadores veiculares, equipamentos de monitoramento de sentenciados costumam utilizar *GPS* e tecnologias de rádio e telefonia celular [Danielsson and Makipaa, 2012]. Como no caso de rastreadores veiculares cujos condutores

[44] Abreviação de *“Short Messaging System”*, que significa “sistema de envio de mensagens curtas”.
[45] *“Ankle monitors”* em inglês.

não querem ser rastreados, as tornozeleiras eletrônicas são sujeitas a violações intencionais por parte dos usuários, e devem ser resistentes a danos físicos [Hoshen et al., 1995].

2.4 Análise de domínio

Nesta seção, apresenta-se uma análise de aspectos relacionados à segurança de sistemas e sua aplicação em sistemas embarcados. Inicialmente, está apresentada uma discussão sobre requisitos de segurança em geral, que deve facilitar a compreensão dos dois tópicos seguintes. O segundo tópico abordado é a modelagem de segurança em sistemas embarcados. Finalmente, apresentam-se os "blocos" que podem ser utilizados para implementar segurança em sistemas embarcados.

2.4.1 Requisitos de segurança

Existem diversas formas de determinar se um sistema satisfaz seus requisitos, e a qualidade de um sistema é relacionada diretamente ao quão bem os requisitos foram atendidos [Farkhani and Razzazi, 2006]. Enquanto requisitos funcionais têm soluções concretas, requisitos de segurança são subjetivos e dependem grandemente do ambiente de operação [Cheng et al., 2003].

Cada aplicação de *software* e *hardware* desenvolvida tem um conjunto de funcionalidades, e portanto um conjunto de requisitos. Entre diferentes aplicações, as funcionalidades variam, e também os respectivos requisitos funcionais. Isto é ainda mais evidente quando as aplicações são de domínios diferentes, como por exemplo, dispositivos rastreadores de carga e servidores de alto desempenho. Por outro lado, os requisitos de segurança têm uma variabilidade muito menor, pois praticamente todos os sistemas têm de especificar níveis de identificação, autenticação, integridade, privacidade, entre outros [Firesmith, 2003].

De forma bastante abstrata, é possível dizer que todas as aplicações apresentam os mesmos tipos de elementos vulneráveis, como dados, comunicação, serviços, componentes de *hardware* e operadores. Estes elementos vulneráveis estão sujeitos ao mesmo tipo de ameaças (como roubos, vandalismo, divulgação de informações sensíveis, espionagem, invasão, entre outros) por parte do mesmo tipo de atacante (como *hackers*, *crackers*, espiões, entre outros) [Firesmith, 2003].

Os tipos de ataques a estes elementos variam de acordo com a arquitetura do sistema atacado, mas as similaridades das ameaças e dos atacantes faz com que os mecanismos de segurança empregados nos mais diversos sistemas sejam semelhantes (como identificadores de usuário, senhas, criptografia, *firewalls*, antivírus, sistemas de detecção de intrusão, entre outros). De um sistema para o outro, os requisitos relacionados à segurança são ainda mais semelhantes que os mecanismos utilizados para implementá-los. Existem várias formas de atender um mesmo requisito de segurança [Firesmith, 2003].

Segurança é um fator de qualidade de um sistema, e pode ser decomposto em subfatores. Estes subfatores são: autenticação, autorização, imunidade, integridade, detecção de intrusão, irretratabilidade, privacidade (ou confidencialidade), auditoria, capacidade de sobrevivência e proteção física. Enquanto os requisitos funcionais podem variar bastante, os requisitos de segurança devem especificar uma certa quantidade de um subfator de segurança. Em [Firesmith, 2003], define-se requisito de segurança como uma especificação de uma quantidade mínima de algum subfator de segurança em relação a algum critério e de acordo com alguma

medida. Neste contexto, entende-se por "critério" um parâmetro que pode envolver: o elemento vulnerável, a ameaça, o atacante e a situação. Ainda em [Firesmith, 2003], sugere-se que todos os requisitos de segurança podem ser escritos com base em um modelo padronizado, o que facilitaria sua análise e verificação.

2.4.2 Modelagem de sistema

Em [Vasilevskaya, 2015] está apresentada uma metodologia denominada *SEED*[46], que pode ser utilizada para modelar sistemas embarcados e seus aspectos de segurança. O *SEED* tem dois níveis de abstração: a base do *SEED*[47] e a realização do *SEED*[48]. A base do *SEED* é a definição genérica do processo de modelagem, enquanto a realização do *SEED* é a aplicação da base do *SEED* com tecnologias específicas. No documento em questão, aplicou-se o *SEED* com *SPACE*, *MARTE* (um perfil *UML* para modelar sistemas embarcados de tempo real [OMG, 2011]) e ontologias.

A base do *SEED* consiste de três atividades: criação de um modelo de sistema, captura do conhecimento de segurança específico do domínio e desenvolvimento de um sistema embarcado com mais segurança. O modelo do sistema é dividido em três partes: o modelo funcional, o modelo da plataforma e a informação de alocação. O modelo funcional descreve aspectos da estrutura e do funcionamento da aplicação, o que pode ser feito, por exemplo, através de diagramas *UML* de classe e de atividade. O modelo da plataforma descreve os recursos disponíveis para a implementação da solução, como elementos de *hardware*, bibliotecas de *software*, memória, entre outros. Finalmente, a informação de alocação descreve quais elementos do modelo funcional estão relacionados a que elementos do modelo de plataforma. O modelo de sistema contém informação relevante para estimar a capacidade do sistema de lidar com aspectos relativos à segurança, mas não contém informações específicas sobre segurança do sistema.

O conhecimento de segurança específico do domínio envolve a listagem de mecanismos de segurança disponíveis para o domínio em particular. Cada mecanismo é composto de implementação, avaliação de desempenho e informação a respeito das propriedades de segurança. Um mecanismo de segurança é criado em diversas etapas. A primeira etapa envolve a criação de um modelo funcional do mecanismo, como por exemplo sua definição matemática. A etapa seguinte é a implementação do mecanismo, o que gera restrições com relação às plataformas de execução em que este mecanismo pode ser executado. Em seguida, é feita a avaliação de desempenho do mecanismo implementado em um conjunto de plataformas, dada uma certa carga de trabalho. Nesta avaliação, busca-se medir quanto os recursos são sobrecarregados pelo mecanismo. Estas informações podem ser utilizadas, por exemplo, para avaliar se um mecanismo se encaixa nas restrições de um sistema em termos de recursos, se o mecanismo será eficaz quando integrado ao sistema e se as propriedades do mecanismo atendem aos requisitos de segurança do sistema. As informações específicas de domínio podem ser armazenadas em um repositório para serem reutilizadas futuramente.

O domínio de aplicação é uma área de interesse para desenvolvimento. Ele pode ser visto como uma fonte de informação específica sobre componentes que são usados para cons-

[46] *"Security-Enhanced Embedded System Design"*, que significa "Desenvolvimento de Sistema Embarcado com Mais Segurança".
[47] *"SEED foundation"* em inglês.
[48] *"SEED realisation"* em inglês.

trução de plataformas e sobre cargas de trabalho típica. Um desenvolvedor utiliza componentes e restrições associadas ao domínio para projetar um sistema. Os mecanismos de segurança utilizam os componentes e restrições do domínio de aplicação.

A última etapa do *SEED*, o desenvolvimento de um sistema embarcado com mais segurança, corresponde a três etapas. A primeira é a análise do modelo de sistema (tanto do modelo funcional quando do modelo de plataforma) para identificar que elementos necessitam de proteção. Em seguida, buscam-se propriedades de segurança que estes elementos devem ter e os mecanismos que as garantem, o que é feito a partir do conhecimento de segurança específico de domínio. Finalmente, avalia-se a viabilidade de utilizar estes mecanismos no sistema integrado.

Em [Vasilevskaya, 2015], apresenta-se uma realização do *SEED* em que utiliza-se uma ontologia para descrever o conhecimento específico de domínio. Nesta ontologia, um mecanismo de segurança é denominado "bloco de construção de segurança" ou BCS[49]. O modelo funcional e a implementação do mecanismo correspondem ao "BCS concreto"[50], enquanto o "BCS abstrato"[51] define a natureza do mecanismo (por exemplo, cifra, resumo criptográfico, entre outros). Nesta ontologia, considera-se que um BCS abstrato pertence a um domínio e protege dados, sejam estacionários ou em trânsito, para atingir um objetivo de segurança através de uma estratégia de segurança. Um objetivo de segurança está relacionado a garantir uma certa propriedade de segurança. Um BCS concreto tem uma especificação funcional e atende a algum padrão, além de implementar um BCS abstrato e satisfazer alguma propriedade de segurança. Esta ontologia está representada na figura 2.1.

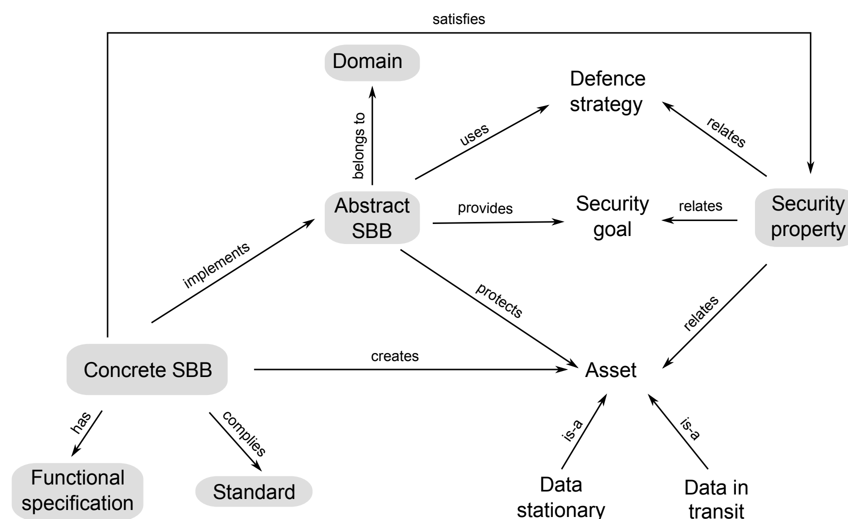

Figura 2.1: Representação visual da ontologia utilizada para descrever o conhecimento específico de domínio [Vasilevskaya, 2015].

A realização do *SEED* em [Vasilevskaya, 2015] também utiliza uma ontologia para descrever a avaliação dos blocos de construção de segurança. Na verdade, a ontologia sugerida é capaz de descrever a avaliação de qualquer tipo de bloco de construção, então utiliza-se neste contexto o termo "bloco de construção reutilizável" ou BCR[52]. Um conceito central desta

[49] *"Security building block"* ou *SBB* em inglês.
[50] *"Concrete SBB"* em inglês.
[51] *"Abstract SBB"* em inglês.
[52] *"Reusable building block"* ou *BRB* em inglês.

ontologia é o "alvo de avaliação" ou AdA[53], que pode corresponder ao BCR inteiro ou a uma funcionalidade dele. Segundo a ontologia, toda avaliação avalia um alvo de avaliação, o que é feito executando uma certa carga de trabalho através de alguma metodologia, para obter algumas métricas. O alvo de avaliação executa sobre alguma plataforma que contém uma série de componentes necessários. Esta ontologia está apresentada na figura 2.2.

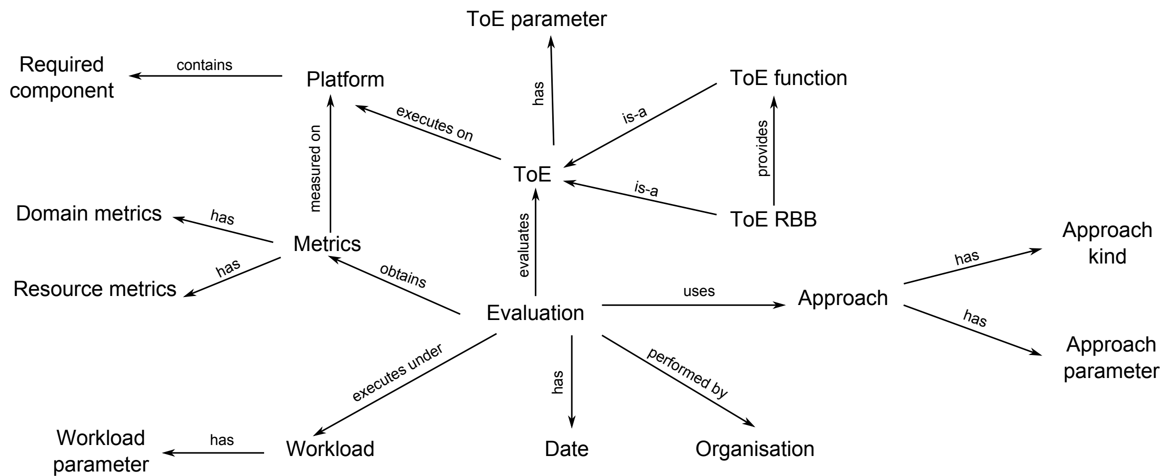

Figura 2.2: Representação visual da ontologia utilizada para descrever a avaliação dos mecanismos [Vasilevskaya, 2015].

2.4.3 Funcionalidades de uma biblioteca criptográfica

Em [Clulow, 2015], está apresentado um estudo das vulnerabilidades existentes nas *APIs* de acesso a dispositivos seguros. No trabalho, apresentaram-se e analisaram-se as funcionalidades básicas expostas por algumas *APIs* criptográficas, como por exemplo a *Cryptoki*. Uma grande contribuição do trabalho em questão é a enumeração das funcionalidades básicas que uma *API* criptográfica fornece.

Segundo o princípio de Kerckhoff, as chaves são o elemento mais crítico de sistemas criptográficos [Burleson and Carrara, 2013]. De acordo com este princípio, em *APIs* criptográficas, a grande maioria das operações recebem como entrada as chaves (exceto operações como o resumo criptográfico, que não utilizam chaves). Entre uma *API* e outra, a forma de fornecer chaves para as operações criptográficas pode variar: as chaves podem ser fornecidas diretamente à função criptográfica (a cada chamada), podem ser armazenadas de alguma forma e referenciadas a partir de *handles* ou ambas as opções anteriores podem estar disponíveis. Quando as chaves criptográficas têm de ser armazenadas em algum meio físico, elas são cifradas utilizando outras chaves [Clulow, 2015].

Em um sistema criptográfico, diferenciam-se dois tipos principais de chaves: as operacionais e as externas. As chaves operacionais são utilizadas internamente no sistema, e muitas vezes não são conhecidas a partir do *software* de alto nível. As chaves externas são importadas para o sistema (ou são exportadas a partir dele), e podem necessitar de uma transformação para convertê-las em chaves funcionais. As chaves externas podem ser cifradas, de forma que é necessário decifrá-las (ou cifrá-las), o que é feito através de uma chave mestra, que é uma chave

[53] *"Target of evaluation"* ou *ToE* em inglês.

operacional, ou de chaves de cifragem de chaves[54], que são chaves externas [Clulow, 2015]. Em [Clulow, 2015], também reforça-se a necessidade de diferenciar as funções de cifragem e decifragem de dados das funções de cifragem e decifragem de chaves, para evitar que seja possível recuperar maliciosamente o valor das chaves. As operações básicas referentes a chaves de uma *API* criptográfica são: criação de chaves, derivação de chaves, estabelecimento de chaves, troca de chaves e verificação de chaves [Clulow, 2015].

A criação de chaves é o procedimento de gerar chaves aleatórias com certas características. Esta operação deve receber como entrada o tipo da chave, seu tamanho e outras metainformações sobre ela. A saída desta operação deve ser um indicador de erro e a chave criada (ou um *token* que a referencia). A derivação de chaves é o procedimento de criar uma chave a partir de outra já existente. Este procedimento é bastante semelhante à criação de chaves, mas utiliza algum algoritmo de derivação e a chave original em lugar de um algoritmo de criação e números aleatórios [Clulow, 2015].

O estabelecimento de chaves refere-se ao procedimento de combinar de duas ou mais partes de chave para formar uma chave criptográfica. Esta combinação pode ser, por exemplo, uma operação de "ou exclusivo". Esta operação pode funcionar, por exemplo, de forma incremental. Neste caso, como entrada, fornece-se a chave combinada até a iteração atual, a parte de chave a ser processada e opções, e recebe-se como saída uma versão mais atualizada da chave final. A troca de chaves refere-se ao procedimento de cifrar chaves para transporte entre dispositivos e também ao procedimento de decifrá-las. A operação de exportar chave refere-se à cifagem da chave, e deve receber uma referência à chave que deve ser cifrada e outra à chave de cifragem de chaves, e gera como saída um código de erro e a chave exportada (cifrada). A operação de importar chave refere-se à decifragem da chave e deve receber como entrada a chave exportada e a chave de cifragem de chaves, e gera como saída um código de erro e a chave importada (decifrada) [Clulow, 2015].

Segundo [Clulow, 2015], as operações com dados básicas de uma API criptográfica são cifragem e decifragem de dados e geração e verificação de códigos de autenticação de mensagem. A cifragem de dados é uma operação que processa dados abertos e gera dados cifrados, enquanto a decifragem processa os dados cifrados e gera novamente os dados abertos (decifrados). As entradas de uma operação de cifragem são os dados abertos e a chave, enquanto suas saídas são um código de erro e os dados cifrados. As entradas de uma operação de cifragem são os dados cifrados e a chave, enquanto suas saídas são um código de erro e os dados abertos. A geração de códigos de autenticação de mensagem é o procedimento de gerar um conjunto adicional de dados que, em conjunto com os dados originais, certifica a autenticidade dos dados. Esta operação recebe como entrada os dados e a chave, e gera como saída um código de erro e o código de autenticação. A verificação de códigos de autenticação de mensagem é o procedimento de verificar se um código de autenticação é válido para um conjunto de dados fornecido [Clulow, 2015].

2.5 Análise

Neste capítulo, apresentou-se um panorama da segurança em sistemas embarcados. Na primeira parte, discutiram-se aspectos genéricos de segurança em sistemas computacionais, como as características que definem a segurança (confidencialidade, integridade, não-

[54] *"Key encripting key"* em inglês.

refutabilidade e autenticidade), os tipos de ataques que tentam prejudicar estas características e as estratégias utilizadas para evitar estes ataques. Para exemplificar como estes conceitos são aplicados em situações reais, analisaram-se dois grupos de sistemas embarcados: os sistemas médicos implantáveis e trajáveis e os sistemas utilizados para prover serviços baseados na localização. Enfim, foi feita uma análise ais detalhada da modelagem de segurança em sistemas embarcados, finalizada com uma discussão sobre quais funcionalidades uma biblioteca criptográfica deve ter.

Foi possível perceber a heterogeneidade de sistemas embarcados e ataques que estes sistemas podem sofrer. Apesar desta heterogeneidade, notou-se que ataques semelhantes podem ser enfrentados através do emprego de estratégias convencionais para proteger sistemas embarcados. Exemplos de tais estratégias são invólucros físicos resistentes, circuitos intrinsecamente seguros, criptografia de dados e detecção de intrusão. A criptografia merece destaque como uma estratégia de proteção de dados utilizada amplamente, e não somente em sistemas embarcados. Apesar de não ser possível utilizá-la em todas as áreas (por exemplo, alguns tipos de sistemas implantáveis), percebeu-se que ela está presente em uma variedade muito ampla de aplicações.

Capítulo 3

Interfaces de *software*

Neste capítulo, estão apresentados os resultados de estudos com o objetivo de direcionar decisões tomadas no projeto da *API* criptográfica para sistemas embarcados. Primeiramente, apontam-se práticas e convenções indicadas por outros autores para elaboração de *APIs*. Em seguida, estão expostas algumas *APIs* existentes. Consideraram-se nesta segunda parte dos estudos as *APIs* de *drivers* e de sistemas operacionais de tempo real, com o objetivo de observar como são construídas e utilizadas as interfaces de *software* em sistemas embarcados. Consideraram-se também as *APIs* de bibliotecas criptográficas, com o objetivo de conhecer de que maneira as aplicações de alto nível comumente utilizam operações criptográficas.

3.1 Definição

Uma *API*[1] é a interface entre sistemas de *software*, isto é, a forma pela qual dois elementos de *software* interagem entre si. No caso de uma biblioteca, a *API* é o conjunto de símbolos exportados pela biblioteca que permitem sua utilização por outras aplicações. Seu formato é muitas vezes considerado a etapa mais importante do desenvolvimento de uma biblioteca, já que afeta o desenvolvimento de todas as aplicações baseadas nela [Blanchette, 2008], e pois ela geralmente não pode ser modificada por um longo tempo, para garantir a retrocompatibilidade [Rama and Kak, 2015]. *APIs* permitem a criação de abstrações através das quais detalhes são omitidos das aplicações de alto nível, reduzindo sua complexidade [Henning, 2007].

Segundo [Fowler, 2014], quando *APIs* são abertas, elas encorajam o desenvolvimento de novas tecnologias a custos menores; por exemplo, elas foram vitais no desenvolvimento da *Internet*. *APIs* abertas facilitam a integração de sistemas, o que é valorizado por seus usuários. Ainda segundo [Fowler, 2014], não deveria ser possível proteger *APIs* por direitos autorais, pois isto prejudica os usuários e os desenvolvedores de outros sistemas.

Em geral, uma certa *API* pode ser considerada "boa" se for fácil de usar e memorizar, se for bem documentada e se operar bem mesmo em casos limítrofes. Segundo [Henning, 2007], é mais fácil criar uma *API* ruim que uma *API* boa. Pequenos erros tendem a ser amplificados, já que são implementados uma vez e utilizados muitas vezes [Henning, 2007].

Grande parte do desenvolvimento de *software* é criar abstrações, que são expostas através das *APIs*. Abstrações ajudam a reduzir a complexidade do *software* final, pois omitem detalhes irrelevantes e preservam apenas os mais relevantes. É comum que abstrações sejam

[1]Sigla em inglês de *"Application Programming Interface"*, que significa "Interface de Programação de Aplicações".

construídas umas sobre as outras, formando as chamadas "camadas de abstração". Quanto mais baixa a camada de abstração em que ocorre uma falha, mais sérias podem ser suas consequências [Henning, 2007].

Camadas de abstração melhor projetadas são comumente utilizadas para minimizar o impacto de *APIs* ruins. Isto é feito envolvendo as funções mal projetadas da *API* ruim dentro de funções melhor projetadas na camada de abstração superior. Esta camada de código torna programas maiores e menos eficientes, o que muitas vezes é causado por cópias de dados desnecessárias. A camada de código adicional é também mais um local para ocorrência de *bugs* e mais um código a ser testado, o que pode aumentar o tempo e o custo do desenvolvimento. Como esta camada não faz parte da *API* original, códigos envoltórios[2] podem acabar sendo programado várias vezes por projetistas diferentes, o que vai na contramão da reusabilidade. Em suma, apesar de os envoltórios melhorarem o uso de *APIs* ruins, eles não resolvem o problema [Henning, 2007].

A padronização das interfaces de *software* simplifica o processo de desenvolvimento, já que ela torna possível o reaproveitamento de código entre projetos e reduz a curva de aprendizado [Johnson, 1997, Renaux and Pottker, 2014]. Sistemas operacionais de uso geral, como *Linux*, e *frameworks*, como as máquinas virtuais *Java*, provêm aos sistemas que os utilizam uma gama ampla de estruturas e funcionalidades através de interfaces padronizadas ou parcialmente padronizadas. Já no escopo de sistemas embarcados mais limitados, a padronização das *APIs* é menos geral. Há padrões localizados, isto é, para certas arquiteturas ou sistemas, mas entre estas arquiteturas e sistemas há enormes variações nas estruturas de *hardware*, na forma e funcionamento de *drivers* e nas plataformas de *software* utilizadas.

3.2 Qualidade de *APIs*

Nesta seção, estão apresentados princípios propostos por outros autores para tornar uma *API* melhor. Em seguida, estão apresentadas formas de avaliar as interfaces de *software*, tais quais propostas por outros autores.

3.2.1 Boas práticas

Não há uma metodologia perfeita para elaboração de uma boa *API*. Na verdade, não existe sequer uma forma universalmente aceita de determinar se uma *API* é "boa" ou não. A seguir são apresentadas metodologias sugeridas para elaboração de *APIs*.

Princípios de Henning

Em [Henning, 2007], está apresentada uma série de princípios que podem melhorar a qualidade de uma *API*. Segundo o autor, enquanto estes princípios não garantem a boa qualidade do *software* resultante, eles podem ajudar a melhorá-la.

Uma boa *API* deve ser mínima e não causar inconvenientes para seus utilizadores. Funções de conveniência são úteis, mas é importante usá-las com cautela. Por exemplo, a classe *"string"* da biblioteca padrão do *C++* contém mais de 100 métodos e funções, o que dificulta sua memorização [Henning, 2007].

[2] *"Wrapper codes"* em inglês.

APIs não devem ser desenvolvidas sem conhecimento do contexto na qual serão utilizadas: enquanto as *APIs* de uso geral devem impor poucas limitações nas entradas aceitáveis, as *APIs* de uso específico devem impor limitações mais estritas. Decidir o que deve e o que não deve ser considerado como erro também é um fator delicado, que se feito incorretamente pode dificultar o trabalho dos utilizadores [Henning, 2007].

APIs devem ser desenvolvidas pela perspectiva de seus utilizadores. Em outros termos, as funções de uma *API* devem ser feitas de forma a serem simples de utilizar, e não de forma a serem simples de implementar. É ideal que *APIs* sejam documentadas antes de serem implementadas, e não o contrário. Quando a documentação é feita após a implementação, ela tende a ser incompleta e conter os mesmos vícios que existem no código. O ideal é que seja feita por algum utilizador, de forma que sua forma final seja mais simples e adequada da perspectiva dos utilizadores [Henning, 2007].

É comum que os desenvolvedores de uma *API* deixem muitos parâmetros configuráveis. Enquanto esta configurabilidade é necessária em alguns casos, em outros pode aumentar desnecessariamente a complexidade da utilização. Uma boa *API* deve também ser "ergonômica" para os utilizadores, no sentido que deve ter consistência. Funções semelhantes que comportam-se de forma diferente podem levar os utilizadores a conclusões erradas. Em geral, devem ser estabelecidas e seguidas convenções simples e uniformes para nomes de funções e de estruturas de dados, para ordenação de parâmetros e para tratamento de erros [Henning, 2007].

Princípios de Blanchette

Em [Blanchette, 2008], estão apresentadas metodologias adotadas na elaboração da *API* da biblioteca *Qt* na empresa *Trolltech*, que veio mais tarde a ser parte da *Nokia*. A autora alerta que, apesar de a *API* ser a parte mais importante do desenvolvimento de uma biblioteca, a complexidade da implementação também deve ser levada em conta ao elaborar novas funções.

Enquanto o conceito de "boa *API*" é subjetivo, há algumas características que são geralmente consideradas importantes para tornar uma *API* boa. Estas características são: facilidade de aprender e memorizar, legibilidade da utilização, dificuldade de usar incorretamente, facilidade de estender e completude. Compactação e consistência não foram colocadas na lista pois são consequência das características que fazem parte dela [Blanchette, 2008].

A facilidade de aprender e memorizar diz respeito à existência de convenções e padrões para nomes, a redução ao máximo de conceitos que devem ser aprendidos e a previsibilidade de comportamento. Elementos parecidos da *API* devem ter nomes parecidos, enquanto elementos diferentes devem ter nomes diferentes. Métodos de conveniência devem ser utilizados com parcimônia, pois podem tornar o código da *API* muito extenso, e devem sempre ser documentados como "de conveniência" [Blanchette, 2008].

A legibilidade do código que utiliza a biblioteca é importante pois um certo código é escrito uma única vez, mas utilizado e alterado repetidas vezes. Quão mais legível for o código, menor a chance de ele conter *bugs*, pois a legibilidade ajuda a tornar os erros mais evidentes [Blanchette, 2008].

A dificuldade de utilizar uma biblioteca incorretamente refere-se à maior facilidade de escrever códigos que utilizam a biblioteca corretamente do que o contrário. Em geral, esta característica é garantida evitando exigir que métodos sejam invocados em ordens específicas e que seja necessário tratar efeitos colaterais inesperados [Blanchette, 2008].

Bibliotecas recebem novas funcionalidades com o tempo, o que inclui novas classes, métodos e funções. Ao adicionar estas novas funcionalidades, é importante levar em conta compatibilidade com as funcionalidades existentes anteriormente, tanto a nível conceitual quando a nível binário [Blanchette, 2008].

Uma *API* é considerada completa se permite aos usuários fazerem tudo o que desejam fazer. Em geral, uma *API* aproxima-se da completude com o tempo, à medida que novas funcionalidades vão sendo adicionadas. Quando uma *API* não é completa, espera-se que seja ao menos extensível para que os utilizadores possam estendê-la ou adaptá-la [Blanchette, 2008].

Antes de desenvolver uma *API*, é preciso saber seus requisitos. Alguns requisitos são claros, enquanto outros não tanto, o que requer um processo de análise de requisitos. Uma metodologia menos adequada é adotada às vezes, que envolve implementar as funcionalidades para depois especificar as *APIs*. Quando esta metodologia é adotada, a *API* tende a refletir características do código subjacente. Como uma *API* pode ser utilizada por muito mais tempo que uma implementação em particular (por exemplo, o padrão *POSIX*), refletir características da implementação em sua forma não é uma boa prática. Sugere-se projetar a *API* e inclusive validá-la criando códigos de exemplo antes de implementá-la. Após a implementação, sugere-se escrever ainda mais códigos de exemplo, o que pode ajudar outros desenvolvedores quando forem utilizar a *API* [Blanchette, 2008].

Quando uma nova funcionalidade é adicionada a uma biblioteca, é ideal que ela tenha uma semântica coerente com as funções semelhantes que já existem na biblioteca. Uma biblioteca deve suportar extensões tanto da parte de seus próprios desenvolvedores quanto por parte de seus utilizadores [Blanchette, 2008].

Aspectos internos da *API* não devem ser expostos para utilização externa sem uma revisão prévia. Quando não há certeza se certas funcionalidades devem ser expostas, ou como devem ser expostas, é indicado não adicioná-las à *API* ou torná-las internas [Blanchette, 2008].

Utilizar opiniões de outros desenvolvedores a respeito da *API* pode ajudar na identificação de problemas existentes e de melhorias que podem ser aplicadas. Opiniões a respeito dos códigos de exemplo também podem ser úteis para melhorar a *API* [Blanchette, 2008].

Em geral, quando é preciso haver distinção entre dois conceitos, seus nomes na *API* devem ser escolhidos de tal forma que seja possível indicá-los sem ambiguidade. De forma semelhante, espera-se que nomes da *API* demonstrem simetria, ou seja, aquilo que é funcionalmente semelhante deve ser nomeado de forma semelhante. Por exemplo, se em algum momento convencionou-se iniciar *setters* com *"set"*, todos devem conter este prefixo, que também não deve ser utilizado em funções e métodos que não sejam *setters*. Ainda sugere-se evitar abreviações (exceto quando a abreviação é convencional, como "min" e "max" para "mínimo" e "máximo", respectivamente) e preferir nomes mais específicos em lugar de nomes genéricos. Nomes mal escolhidos não podem ser trocados, já que *APIs* geralmente têm de preservar compatibilidade reversa. Para implementar uma certa *API*, é possível que utilizem-se outras bibliotecas. Quando isto é feito, é comum que adotem-se as convenções de nomenclatura ou formato da biblioteca utilizada. Isto deve ser evitado, pois muitas vezes a biblioteca usa convenções diferentes das escolhidas para a *API* [Blanchette, 2008].

O comportamento de objetos e estruturas de dados deve ser simples e previsível, então é importante escolher um bom comportamento "padrão". Funções e métodos da biblioteca não devem fazer muito mais do que seria razoável esperar deles: funções que implementam muitas funcionalidades tendem a ter efeitos colaterais pouco compreensíveis por parte dos utilizadores.

Os casos limítrofes devem ser levados em conta, pois podem levar a *"bugs"* nas aplicações construídas com base na biblioteca [Blanchette, 2008].

A extensibilidade de uma biblioteca pode ser garantida pela presença de métodos virtuais nas suas classes. Decidir que métodos serão virtuais é crítico, pois não deve ser possível alterar o funcionamento de todos os métodos e também porque métodos virtuais são menos eficientes que métodos não-virtuais. Sugere-se que métodos virtuais sejam protegidos, não públicos [Blanchette, 2008].

Em geral, a inicialização de objetos deve ser feita por meio de propriedades, e não por meio de parâmetros. Apesar de esta característica tornar o código dos utilizadores levemente maior, ela os torna mais compreensíveis e fáceis de manter [Blanchette, 2008].

3.2.2 Formas de avaliação

A interface entre elementos de *software* é uma etapa importante do desenvolvimento. Na seção anterior, mostraram-se "boas práticas" de desenvolvimento de *APIs* segundo diversos autores. Nesta seção, estão exibidos métodos utilizados para avaliar a qualidade de interfaces de *software* existentes.

A *API* proposta neste trabalho foi especificada em linguagem *C*, que é estruturada. Assim, não é possível aplicar diretamente métricas voltadas a sistemas orientadas a objetos para a avaliá-la. Mesmo não sendo "estruturalmente" orientada a objetos, como explicado em mais detalhes no capítulo 4, é possível correlacionar conceitos da *API* especificada com conceitos de orientação a objetos. Assim, sob certa ótica, é possível considerá-la como orientada a objetos. Por este motivo, considerou-se relevante estudar metodologias de avaliação de *APIs* orientadas a objetos.

Diversos fatores definem a qualidade de uma *API*, como corretude, completude, entre outros. Dentre estes fatores, merece destaque especial a usabilidade. A usabilidade é a característica que determina a simplicidade com que desenvolvedores utilizam a *API* [Rama and Kak, 2015]. Grande parte dos mecanismos adotados para avaliar a usabilidade de *APIs* é qualitativa, mas existem diversos trabalhos que buscam utilizar métricas quantitativas [Bandi et al., 2003, Doucette, 2008, de Souza and Bentolila, 2009, Rama and Kak, 2015]. As "boas práticas" apresentadas na seção anterior são heurísticas que buscam principalmente melhorar a usabilidade de *APIs*. Da mesma forma, as métricas utilizadas para avaliação quantitativa de *APIs* também são métodos heurísticos.

Métricas de complexidade orientadas a objetos

Em [Bandi et al., 2003], estão apresentada métricas para avaliar a complexidade de interfaces de *software*. No trabalho, consideraram-se interfaces orientadas a objetos. A complexidade é uma fator que geralmente está associado à menor usabilidade, então a medição de complexidade serve como uma medição de quão ruim é a usabilidade de uma *API*. Portanto, uma *API* com menor métrica de complexidade deve ter uma melhor usabilidade.

No estudo em [Bandi et al., 2003], foram sugeridas três métricas de complexidade: nível de interação, tamanho de interface e complexidade dos argumentos de operação. A complexidade de uma classe é o somatório da complexidade de seus métodos, e a complexidade de uma API é o somatório da complexidade de suas classes.

O nível de interação de um método é a soma ponderada de dois valores: o número de interações e a soma da força das interações, tal qual apresentado na equação 3.1. O número de

interações é o produto do número de atributos da classe com o número de parâmetros do método. A força de uma interação é produto do tamanho de um atributo de dados com o tamanho do parâmetro correspondente. Para estes cálculos, também se considera o valor de retorno como um parâmetro [Bandi et al., 2003].

$$IL = K_1 \times (\text{number of interactions}) + K_2 \times \sum (\text{strength of interactions}) \tag{3.1}$$

O tamanho da interface é a soma ponderada de dois valores: o número de parâmetros e a soma dos tamanhos dos parâmetros, tal qual apresentado na equação 3.2. O tamanho de cada parâmetro é um valor pré-fixado com base no seu tipo primitivo (não é o tamanho do parâmetro em memória); por exemplo, valores booleanos têm tamanho igual a 0, valores inteiros têm tamanho igual a 1 e ponteiros têm tamanho igual a 5 [Bandi et al., 2003].

$$IS = K_3 \times (\text{number of parameters}) + K_4 \times \sum (\text{sizes of parameters}) \tag{3.2}$$

A complexidade dos argumentos de operação é a soma dos tamanhos dos parâmetros e do valor de retorno, tal qual apresentado na equação 3.3. Nesta equação, o valor $P(i)$ indica o tamanho do i-ésimo parâmetro [Bandi et al., 2003].

$$IS = \sum P(i) \tag{3.3}$$

As avaliações propostas no estudo consideram métodos com mais parâmetros e classes com mais atributos como mais complexas. Também se considera que parâmetros de tipos mais complexos tornam a *API* mais complexa. Por exemplo, métodos com parâmetros inteiros terão uma métrica de complexidade maior que um método com parâmetros booleanos.

O trabalho em [de Souza and Bentolila, 2009] usa as mesmas métricas propostas em [Bandi et al., 2003], mas apresenta os resultados de forma gráfica, através de uma representação denominada *"tree map"*. Segundo o artigo, com os *tree maps* é possível, de forma visual, avaliar a complexidade de uma API ou comparar duas ou mais *APIs*. No artigo também se apresenta um segundo tipo de visualização, o *"starplot"*, que permite visualizar a complexidade de classes individualmente.

Ferramenta de análise e avaliação de usabilidade de *API*

Em [Doucette, 2008], é descrita uma ferramenta de análise e avaliação da usabilidade de *APIs* de forma automática. Esta ferramenta analisa *bytecode Java* compilado através da *"Java Reflection API"*.

As métricas utilizadas por esta ferramenta são as seguintes:

1. Número de classes.
2. Número de classes que não são de exceção.
3. Tamanho da cadeia de herança de cada classe.
4. Número de construtores por classe.
5. Número de métodos de cada classe, e uma média de todas as classes.

6. Número de métodos sobrecarregados.
7. Número de parâmetros por método.
8. Número de atributos públicos por classe, e se estes métodos seguem as convenções de nomeação da linguagem *Java* e se são de somente leitura.
9. Número de métodos e atributos estáticos por classe.
10. Quantos métodos estáticos retornam um objeto definido na *API*.
11. Número de parâmetros separados por tipos primitivos e tipos complextos, e estes últimos separados em objetos da biblioteca padrão e objetos definidos na *API*.
12. Número de *setters* que retornam *void*, *boolean*, *int* ou outras *flags*.

Diversos critérios de avaliação levam em conta convenções da linguagem *Java*, como por exemplo a convenção de usar atributos públicos como constantes e de capitalizar todas as letras de contantes.

Medidas estruturais da usabilidade de *API*

Em [Rama and Kak, 2015], está apresentado um estudo de características que tornam uma *API* ruim, e métricas projetadas para avaliar estas características. Os autores elencaram uma lista de "defeitos" que *APIs* comumente apresentam, justificando por que cada um deles é problemático.

No estudo, considera-se que métodos sobrecarregados com diferentes valores de retorno são problemáticos, pois isso confunde desenvolvedores que utilizam a *API* e dificulta o entendimento de programas em que ela é utilizada. Métodos com sequências de parâmetros do mesmo tipo são considerados problemáticos pois, neste caso, é fácil para os desenvolvedores se confundirem com relação à ordem dos parâmetros. Métodos com muitos parâmetros são ruins pois isso dificulta a memorização do método por parte dos desenvolvedores. Um outro problema existente em *APIs* é a existência de métodos que recebem parâmetros de mesma natureza em ordens diferentes, ou seja, inconsistências no ordenamento de parâmetros entre métodos. Essa característica é considerada problemática pois dificulta a memorização e o entendimento do código resultante. A existência de muitos métodos com finalidades diferentes, mas com nomes semelhantes, é ruim pois confunde desenvolvedores e dificulta a compreensão do código final. Outro problema em potencial é não agrupar métodos semelhantes, o que pode fazer com que desenvolvedores, em lugar de usar um método adequado para resolver algum problema, acabem utilizando métodos diferentes de forma menos eficiente, por não encontrarem o método desejado. Considera-se como ruim não indicar quando um método é seguro ou não para uso concorrente (*thread-safe*), pois sem esse conhecimento, desenvolvedores podem cometer erros. Disparar exceções de tipos genéricos também é problemático, pois dificulta o tratamento de exceções a partir do código principal. o último problema documentado é a má qualidade da documentação, o que dificulta o aprendizado e pode levar a erros [Rama and Kak, 2015].

As métricas elaboradas no trabalho buscaram quantificar individualmente cada um destes parâmetros através de cálculos simples, que podem ser executados de forma automática. Os autores implementaram um *software* que calcula estas métricas a partir de *software* feito em *Java* e avaliaram várias bibliotecas, entre elas a biblioteca padrão de *Java* [Rama and Kak, 2015].

3.2.3 Discussão

Nesta seção, foram discutidas as "boas práticas" de elaboração de *APIs* propostas em [Henning, 2007] e [Blanchette, 2008], além das formas de avaliação propostas em [Bandi et al., 2003], [Doucette, 2008] e [Rama and Kak, 2015]. Notou-se certa convergência entre os trabalhos citados em relação a diversas características desejáveis em *APIs*.

Em geral, os autores citados consideram que *APIs* são melhores à medida que são mais simples de utilizar, à medida que têm mais consistentência na forma de utilização e à medida que são mais simples de serem aprendidas. Em [Henning, 2007], considera-se que especificar corretamente os casos de erro é importante, e sugere-se que uma *API* seja sempre elaborada do ponto de vista de quem a utiliza, não de quem a implementa. Em [Blanchette, 2008], defende-se a adoção de convenções na forma e no nome das estruturas de código.

As métricas de avaliação estudadas buscaram quantificar as características tidas como "más práticas". Por exemplo, a simplicidade de uso é um fator tido como "bom" em uma *API*, e funções ou métodos com menos parâmetros são menos complexos de se utilizar, então nos três trabalhos estudados, há métricas de complexidade em função do número de parâmetros que cada função ou método recebe.

Selecionaram-se algumas práticas sugeridas por estes autores para embasar a construção da *API* sugerida nesta dissertação. A *API* proposta tem uma especificação aberta e pode ser utilizada livremente, como sugerido em [Fowler, 2014]. Ela não apresenta funções de conveniência, que, segundo [Henning, 2007], poderiam torná-la mais complexa. As funções definidas na *API* devem ter comportamentos bem definidos, evitando assim consequências além das esperadas, como sugerido por [Blanchette, 2008]. Segundo [Henning, 2007], é ideal que *APIs* sejam desenvolvidas com a forma de utilização em mente, não com a forma de implementação. Por este motivo, foi feito um estudo de algumas *APIs* criptográficas e *APIs* para sistemas embarcados já existentes, para ter contato com as formas de utilização mais convencionais nestes nichos. A *API* elaborada também é extensível e modular, como sugerido em [Blanchette, 2008]. Em seu projeto, também se objetivou ter poucas funções e minimizar o número de parâmetros necessários, pois estes fatores podem aumentar a complexidade e diminuir a usabilidade [Henning, 2007, Bandi et al., 2003, Doucette, 2008, Rama and Kak, 2015].

3.3 Drivers

Nesta seção, está apresentada uma análise de algumas "boas práticas" de elaboração de *drivers* para sistemas embarcados. Este estudo, como os demais apresentados neste capítulo, foi realizado com o objetivo de encontrar características comuns às *APIs* utilizadas em sistemas embarcados, o que norteou as decisões a respeito da *API* proposta neste documento.

Para que aplicações de *software* tenham acesso a recursos externos ao processador, são necessários *drivers*. Segundo [Khyo, 2011], *"driver"* é um componente de *software* que controla a operação de um dispositivo físico ou virtual. Dispositivos físicos são componentes de *hardware* que executam operações para o usuário ou sistema operacional, enquanto dispositivos virtuais são componentes de *software* sem representação física. Em [Microsoft, 2016b], *drivers* que controlam dispositivos virtuais são chamados de "*drivers* de *software*". Ainda em [Microsoft, 2016b], diferenciam-se os "*drivers* funcionais", que se comunicam diretamente com o dispositivo, dos "*drivers* de filtragem", que intermedeiam a comunicação entre a aplicação e outro *driver*.

Dispositivos que fornecem o mesmo conjunto de funcionalidades podem ser agrupados em classes de dispositivos. É comum *drivers* não se restringirem ao tratamento de um único dispositivo, mas sim serem capazes de tratar diversos dispositivos pertencentes à mesma classe. Quando vários dispositivos de uma classe são tratados pelo mesmo *driver*, o sistema operacional pode interagir com eles da mesma forma. Desta forma, as aplicações e outras partes do sistema operacional que interagem com algum dispositivo através da interface da classe de dispositivos não precisam saber detalhes de *hardware* do dispositivo utilizado. Este *driver* pode ser entendido como um tradutor entre comandos referentes à classe de dispositivos e comandos reais específicos do dispositivo [Khyo, 2011].

Muitas vezes, a implementação de *drivers* é feita de forma que a lógica de alto nível e o código específico de acesso ao *hardware* são combinados nas mesmas funções. Quando este procedimento é adotado e é necessário suportar o dispositivo (ou classe de dispositivos) em uma nova plataforma, é preciso desenvolver um *driver* inteiramente novo, o que aumenta o tempo de desenvolvimento [Amar et al., 2008]. Durante o desenvolvimento de um produto, o tempo gasto na elaboração de *drivers* pode ser muito grande, tempo no qual os desenvolvedores perdem foco da implementação do produto em si [Beningo, 2012].

Um *driver* qualquer inclui código que interage com o sistema operacional e acessa registradores de *hardware*, código que interage com o resto da aplicação embarcada e código que realiza suas operações internas [Bammi, 2006]. *Drivers* devem implementar abstrações de *hardware* e esconder detalhes internos dos dispositivos [Khyo, 2011].

Uma das etapas mais críticas da elaboração de um driver é a especificação de como suas funcionalidades serão expostas para as aplicações, ou seja, sua *API*. Uma *API* bem projetada deve permitir que o acesso ao *hardware* seja feito da mesma forma em diferentes plataformas e projetos. Via de regra, as *APIs* devem incluir rotinas para acessar funcionalidades comuns e úteis de cada periférico [Beningo, 2012].

Não existem padrões universalmente aceitos relacionados às *APIs* de *drivers*. A padronização destas interfaces encorajaria a criação de *software* padrão para controlar e gerenciar os dispositivos [Amar et al., 2008]. Em [Bammi, 2006], sugere-se que sejam utilizadas *APIs* existentes como ponto de partida para elaboração de novas *APIs*, tais como as do *kernel* do *Linux* e das bibliotecas do *Arduino*, por exemplo.

Existe uma tentativa de padronização da *API* de *drivers* que merece destaque: o *CMSIS*. O *CMSIS* é uma camada de abstração de *hardware* padronizada para arquiteturas da família *Cortex-M*, criada pela *ARM* [Renaux and Pottker, 2014]. Apesar desta padronização, mesmo em arquitetura *Cortex-M*, os *drivers* de dispositivos externos fornecidos por fabricantes têm estruturas e fluxos de funcionamento diferentes entre si.

Um requisito básico de grande parte dos *drivers* é a garantia de que apenas uma *thread* da aplicação acesse o periférico em um dado instante, o que geralmente é resolvido utilizando semáforos ou estruturas semelhantes. Desta forma, a cada *driver* é associado um semáforo, que deve ser decrementado ao início e incrementado ao final do uso. A forma de funcionamento de diversos *drivers* requer que *threads* tenham acesso exclusivo a eles durante sequências longas de operações, como é o caso de impressoras. Neste caso, as operações de controle de concorrência por semáforos ou outras estruturas deve ser feito de forma explícita no código das *threads*. Quando o acesso exclusivo não precisa ocorrer em sequências de operações, mas apenas durante a execução das operações em si, o controle de concorrência pode ocorrer dentro do próprio *driver* [Kalinsky, 1999].

Drivers podem funcionar de forma síncrona ou assíncrona. Em *drivers* síncronos, quando uma *thread* de aplicação requisita uma operação de leitura ou escrita no periférico, sua execução não continua até a operação ser terminada. Em *drivers* assíncronos, quando uma *thread* de aplicação requisita uma operação de leitura ou escrita no periférico, sua execução continua normalmente. *Drivers* assíncronos permitem um maior paralelismo, mas geralmente são mais complexos de desenvolver e utilizar [Kalinsky, 1999]. Em [Kalinsky, 1999], discutem-se diversos problemas encontrados ao programar *drivers* assíncronos e como resolvê-los.

3.3.1 Organização de código-fonte

Em [Beningo, 2012], está apresentada uma metodologia de desenvolvimento do código-fonte de *drivers* de periféricos mapeados em memória que tem como objetivo maximizar sua reusabilidade. A metodologia proposta inclui a utilização de duas estruturas de dados que auxiliam na estruturação do código: *arrays* de ponteiros e tabelas de configuração.

Arrays de ponteiros são estruturas de dados que agrupam logicamente múltiplos periféricos mapeados em memória que funcionem da mesma forma. Por exemplo, em um processador que tenha múltiplos *timers*, um *array* de ponteiros agruparia os ponteiros para os registradores equivalentes de cada um dos *timers* em estruturas de adjacentes na memória [Beningo, 2012].

As tabelas de configuração são estruturas de dados que informam quais os valores iniciais dos registradores de periféricos. Em conjunto com *arrays* de ponteiros, as tabelas de configuração simplificam a inicialização de *drivers*, que passam a consistir principalmente da cópia de valores das tabelas de inicialização para os registradores dos periféricos [Beningo, 2012].

3.3.2 *Device Object Model*

Em [Amar et al., 2008], é descrito um método de desenvolvimento de *drivers* denominado *Device Object Model*[3]. Segundo [Amar et al., 2008], a adoção desta metodologia foi responsável por reduzir o custo de desenvolvimento e manutenção de *software* desenvolvido pela *Cisco* em aproximadamente 30% e por diminuir o tempo de lançamento[4] de novos produtos.

Por esta metodologia, *drivers* são divididos em quatro componentes fundamentais: o *driver* de interface, o *driver* de *plug-in*, o mapeador interface-dispositivo e os *drivers* de dispositivo. O *driver* de interface implementa a *API* de acesso ao componente, e é estruturado de forma a ser independente de aspectos específicos de cada plataforma ou dispositivo. O *driver* de *plug-in* abstrai a plataforma subjacente, permitindo que novas plataformas sejam adotadas sem alterar o código genérico. O mapeador interface-dispositivo é uma pequena camada que adapta as *APIs* do *driver* de interface às *APIs* de um ou mais *drivers* de dispositivo. Os *drivers* de dispositivo implementam *software* para controlar de fato os dispositivos. Estes *drivers* devem utilizar uma estrutura que expõe um conjunto de *APIs* padronizadas e, opcionalmente, um conjunto de *APIs* estendidas. Cada dispositivo é controlado por uma instância diferente do *driver* de dispositivo [Amar et al., 2008].

Os *drivers* de dispositivo são divididos em uma parte dependente e outra independente da plataforma e do sistema operacional, com *APIs* padronizadas na interface entre estas cama-

[3]Traduzido como “Modelo de Objeto de Dispositivo”.

[4]*“Time to market”* ou *TTM* em inglês.

das. A camada dependente de plataforma inclui código responsável pela interface entre o *driver*, o *hardware* e o sistema operacional. Como esta camada é integrada à plataforma subjacente, ela deve ser implementada de forma diferente em cada sistema operacional e plataforma que deve suportar o *driver*. Mesmo assim, é possível haver reaproveitamento de código de partes significativas desta camada entre plataformas. A camada independente de plataforma (chamada de camada *"Dev Object"*) é responsável por implementar acesso aos registradores do dispositivo, coletar estatísticas, manter estruturas de controle, entre outros. Busca-se manter esta camada simples e leve, para facilitar o reaproveitamento de *software* entre diferentes plataformas. Apesar de esta camada ser logicamente separada do sistema operacional, ela é compilada junto dele [Amar et al., 2008].

A camada independente de plataforma é ainda dividida em três partes: objetos básicos de dispositivo, extensões comuns da classe de dispositivo e extensões específicas de dispositivo. Os objetos básicos de dispositivo definem funções padronizadas que se aplicam a dispositivos de qualquer tipo (criação, destruição, inicialização, habilitação e desabilitação de interrupções, etc). Como estas funções estão presentes em todos os dispositivos, podem ser invocadas a partir de código comum sem necessidade de conhecer o tipo do dispositivo. As operações comuns ao todos os dispositivos suportados pelo *hardware* são parte das extensões comuns da classe de dispositivo. Nesta parte, são especificadas interfaces para configurar, coletar estatísticas e monitorar cada tipo de dispositivo. Esta camada não deve se limitar a uma implementação em particular do dispositivo. As extensões específicas de dispositivo incluem funções específicas de cada variação do dispositivo pertencente à classe. O uso destas extensões deve ser minimizado pois diminui a reusabilidade do código resultante [Amar et al., 2008].

Para isolamento entre o sistema operacional e os *drivers*, limitam-se os tipos de dados utilizados, exigindo a utilização de tipos *POSIX*. Todos os tipos e estruturas de dados mais complexas devem ser construídos a partir destes tipos. Também limitam-se as primitivas de sistema operacional que podem ser utilizadas (`malloc`, `free`, `printf`, entre outros), de forma a maximizar a independência em relação ao sistema operacional. Desta forma, do *driver* de dispositivo independente de plataforma fica completamente isolado do sistema operacional [Amar et al., 2008].

3.3.3 *Multi tiered architecture*

Em [Bammi, 2006], está sugerida uma arquitetura de *drivers* denominada *"multi tiered architecture"*. A utilização desta arquitetura envolve a divisão de *drivers* em três camadas: o nível de aplicação, o nível de sistema e o baixo nível. O nível de aplicação é acessado pelas *threads* de aplicação e nunca interage diretamente com o código do baixo nível. O nível de sistema provê acesso às funções de baixo nível para o nível de aplicação. Para cada dispositivo em particular, deve haver um único objeto de *driver* que fornece acesso sincronizado ao dispositivo. Segundo [Bammi, 2006], muitos *drivers* desenvolvidos para aplicações de tempo real da *Schlumberger* usam uma arquitetura semelhante a esta.

No nível de sistema, é preciso gerenciar o acesso ao *hardware* para que não haja mais de uma *thread* de aplicação tentando manipulá-lo ao mesmo tempo. Este gerenciamento de acesso pode ocorrer tanto se o *driver* for síncrono quanto se ele for assíncrono. Em geral, estruturas de dados e *buffers* internos do *driver* não são acessíveis à aplicação para evitar que sejam corrompidos maliciosamente ou acidentalmente. Encapsulamento garante o isolamento entre as estruturas internas do *driver* e a aplicação, o que às vezes tem um impacto negativo

no desempenho. Para solucionar este problema, utiliza-se um mecanismo que permite acesso direto às estruturas e *buffers* internos do *driver* a um grupo seleto de funções e classes. No caso da linguagem *C++*, este comportamento é obtido através do modificador *"friend"*, que permite acesso a campos privados de uma certa classe por parte de funções específicas ou outras classes [Bammi, 2006].

3.3.4 *Abstraction layers*

Em [Finseth, 2004], é apresentada uma sugestão de metodologia para elaboração de *drivers* para computadores de placa única. No texto, afirma-se que esta metodologia diminui o custo de tempo e dinheiro gastos na integração entre dispositivos, simplifica o processo de compilação e viabiliza economicamente a avaliação de múltiplas placas, sistemas operacionais e periféricos.

A metodologia proposta envolve a criação de uma camada de abstração entre as aplicações e o *hardware*. Através da camada, as aplicações interagem com o *hardware* de forma mais genérica do que se utilizassem um *driver* específico. As chamadas de função mais genéricas provenientes da aplicação são delegadas para uma camada de *software* que lida de forma mais específica com o *hardware*. Esta camada isola o desenvolvedor de aplicações de todos os detalhes referentes à placa utilizada e permite que o *driver* seja utilizada em diversas placas diferentes sem necessidade de intervenção dos desenvolvedores [Finseth, 2004].

A camada de abstração mais baixa é a camada de leitura e escrita (*"I/O abstraction layer"*), que lida com aspectos de baixíssimo nível, como programação de registradores, atributos de barramento e temporização. A camada de abstração de *driver* para placa (*"device-driver-to-BSB abstraction layer"*) define rotinas parão para tratar interrupções, traduzir endereços, ler e escrever na memória e controlar *clocks*. Esta camada tem o objetivo de remover dos *drivers* todas as rotinas dependentes de algum *hardware* específico, de forma que eles ficam completamente independentes da placa utilizada [Finseth, 2004].

Em essência, a camada de abstração define um conjunto comum de rotinas que permitem ao *driver* comunicar-se com a placa. Estas rotinas permitem que o *driver* seja escrito sem nenhum conhecimento específico a respeito da placa ou do processador [Finseth, 2004].

3.3.5 Discussão

Nesta seção, foram apresentadas metodologias de desenvolvimento de *drivers* para sistemas embarcados. Segundo [Khyo, 2011], a definição de "*driver* virtual" é bastante semelhante à de uma camada de abstração. Por esta definição, é possível considerar que a *API* criptográfica proposta nesta dissertação é uma espécie um *driver* virtual de acesso à biblioteca criptográfica. Com este conceito em mente, as metodologias de desenvolvimento de *drivers* podem ser entendidas como metodologias para construção de camadas de abstração de *software* de um modo geral.

Na metodologia de desenvolvimento de *drivers* proposta em [Amar et al., 2008], *drivers* são divididos em quatro componentes. Estes componentes são divididos de forma a que apenas um deles dependa de aspectos específicos da plataforma (como chamadas de sistema, entre outros) e que apenas um deles faça a comunicação com os dispositivos de *hardware*. Esta divisão tem como objetivo maximizar o reaproveitamento de código entre *drivers* e plataformas.

A *API* proposta nesta dissertação não depende de uma implementação em particular. As metodologias propostas em [Amar et al., 2008] dizem respeito ao código dos *drivers*, e portanto não são diretamente aplicáveis à *API*, mas aproveitou-se a ideia de separar aspectos dependentes dos independentes do dispositivo (no caso, a própria biblioteca criptográfica). Conceitos similares são defendidos em [Bammi, 2006] e [Finseth, 2004].

3.4 Sistemas operacionais

A área de sistemas operacionais de tempo real que são utilizados em sistemas mais limitados é pouco padronizada. Há diferenças na forma com a qual diversos sistemas operacionais de tempo real são implementados e não existe convenção para forma e nomeação das funções e estruturas expostas em suas *APIs*. Ainda assim, diferentes sistemas operacionais de tempo real costumam incluir um conjunto limitado e geralmente similar de funcionalidades, de forma que é possível traçar analogias entre estruturas e funções de diversos sistemas. Os estudos em [Tan and Tran Nguyen, 2009] e [Renaux and Pottker, 2014] apresentam comparações das funcionalidades presentes em diferentes sistemas operacionais de tempo real comerciais e gratuitos. Em [Klaus et al., 2014], apresentou-se uma comparação entre dois sistemas operacionais de tempo real e dois padrões de *API* para sistemas deste tipo, demonstrando diversas similaridades e algumas diferenças entre eles.

Para verificar as funcionalidades existentes em sistemas operacionais de tempo real, foram escolhidas *APIs* de sistemas operacionais para serem estudadas em maior detalhe. A pesquisa referente a sistemas operacionais focou-se em duas especificações, *CMSIS-RTOS* e *OSEK/VDX*, e em dois sistemas de código aberto, *FreeRTOS* e o *ChibiOS*. Os dois padrões foram escolhidos pois são tentativas de centralizar em uma única *API* os pontos comuns a sistemas operacionais de tempo real utilizados em um nicho específico, seja um conjunto de arquiteturas semelhantes (*CMSIS-RTOS*) ou uma mesma gama de aplicações (*OSEK/VDX*). Os dois sistemas operacionais foram escolhidos por serem de código aberto, estarem disponíveis sob licenças gratuitas (e também sob licenças comerciais quando for interessante manter fechado o código da aplicação) e terem implementações para diversas plataformas, compartilhando em todas elas a mesma *API*. Os estudos a respeito dessas *APIs* estão apresentados a seguir.

Considerou-se inicialmente estudar também o padrão *ARINC-653*, usado em *software* embarcado da aviônica. A especificação *ARINC*-653 descreve o *kernel APEX*[5] e serviços relacionados. Esta especificação refere-se a sistemas operacionais de tempo real utilizados na aviônica. Além de especificar serviços, ele exige que sejam providas separação temporal e espacial fortes entre as aplicações, para contenção de falhas de *software*. A separação espacial é feita por *hardware* de gerenciamento de memória (como é feito, por exemplo, pela *MMU*[6]), enquanto a separação temporal é garantida por um escalonamento temporal periódico pré-estabelecido para cada uma das partições do sistema [Dubey et al., 2010]. Este padrão foi descartado da pesquisa pois trabalha com modelos de operação muito mais complexos que os demais, e foge do escopo de sistemas embarcados em geral.

[5] Abreviação de *"Application Executive"*.

[6] Abreviação de *"Memory Management Unit"*, que significa "Unidade de Gerenciamento de Memória".

3.4.1 *CMSIS-RTOS*

Como parte do padrão *CMSIS*, também há uma proposta de padronização das interfaces de sistemas operacionais embarcados: o *CMSIS-RTOS*. A especificação propõe utilizar uma camada de abstração padronizada acima do sistema operacional, com a qual diferentes fabricantes podem fazer *drivers* que acessam funcionalidades de sistemas operacionais sem necessitar que um sistema operacional específico seja utilizado [CMSIS, 2012]. Em [Renaux, 2014], considera-se a possibilidade de a camada *CMSIS-RTOS* ser utilizada também em arquiteturas diferentes de *Cortex-M*. Os resultados de um estudo apresentado no mesmo artigo mostram que não há impactos significativos no desempenho devidos à utilização da camada *CMSIS-RTOS* sobre o sistema operacional em lugar de utilizar o sistema operacional isoladamente [Renaux, 2014].

A *API* do *CMSIS-RTOS* especifica uma interface genérica de sistema operacional de tempo real para sistemas baseados em processadores *Cortex-M*. Através desta *API*, as funcionalidades relacionadas ao sistema operacional são padronizadas, o que diminui o esforço de aprendizado e simplifica o compartilhamento de componentes de *software*. Uma implementação do *CMSIS-RTOS* é uma camada acima de outro sistema operacional de tempo real. No *CMSIS-RTOS*, as seguintes funcionalidades estão disponíveis: gerenciamento de *threads*, espera, temporizadores[7], duas formas de controle de acesso (semáforos e *mutexes*) e três formas de comunicação entre *threads* (sinais, mensagens e *mail*) [CMSIS, 2012].

Em praticamente todas as funções da *API* do *CMSIS-RTOS*, é fornecido um parâmetro de tempo limite[8]. Este tempo limite indica qual o tempo máximo que deve-se esperar para tentar executar a operação. Por exemplo, em um *mutex*, este tempo indica qual o tempo máximo que se deve esperar para obter o acesso ao recurso compartilhado. Quando o tempo limite fornecido é zero, as funções devem retornar imediatamente, e se o tempo limite for `osWaitForever`, as funções devem esperar indefinidamente. A partir de uma interrupção, as funções só podem ser executadas com tempo limite igual a zero [CMSIS, 2012].

Há dois tipos de dados que referenciam cada tipo de objeto no *CMSIS-RTOS*, a “definição” e o “identificador”. O identificador de um objeto é uma forma de referenciá-lo, enquanto a definição é uma forma de determinar seus parâmetros e, no caso de alocação estática, reservar memória para ele. Todas as funções de inicialização recebem a definição e devolvem o identificador, enquanto todas as funções de manipulação recebem apenas o identificador [CMSIS, 2012].

Todas as funções e tipos de dados no *CMSIS-RTOS* começam com `os`. Tipos internos têm um *underscore* após o prefixo `os`, enquanto tipos públicos são nomeados em *Pascal case*. Os tipos que definem estruturas do sistema operacional recebem o sufixo `Def_t`, enquanto os tipos que definem os identificadores de objetos recebem o sufixo `Id`. As estruturas de definição não são criadas manualmente, mas sim através de *macros* com o mesmo nome da estrutura, mas sem o sufixo `_t`. Da mesma forma, as definições também não são acessadas diretamente, mas sim através de uma *macro* cujo nome é o prefixo `os` seguido do tipo do objeto. Por exemplo, o identificador de um semáforo usa o tipo `osSemaphoreId`, enquanto a definição usa o tipo `osSemaphoreDef_t`, é criada através da *macro* `osSemaphoreDef` e é acessada através da *macro* `osSemaphore` [CMSIS, 2012].

[7] *“Timers”* em inglês.
[8] *“Timeout”* em inglês.

As funções que operam sobre objetos recebem o prefixo `os`, seguido do tipo da estrutura que a função manipula (`Mutex`, `Mail`, `Semaphore`, entre outros) e terminado com o nome da operação (`Create`, `Wait`, `Release`). Por exemplo, a função que espera pela liberação de um recurso compartilhado por um *mutex* chama-se `osMutexWait`. As funções que não manipulam estruturas de dados em particular recebem apenas o sufixo `os` seguido do nome da operação. Por exemplo, a função que faz a *thread* atual esperar por alguns milissegundos chama-se `osDelay` [CMSIS, 2012].

3.4.2 *OSEK/VDX*

O padrão *OSEK/VDX* consiste em uma série de padrões para unidades de controle distribuídas em veículos desenvolvidos em conjunto por diversas empresas da indústria automotiva [OSEK Group, 2004a]. O projeto *OSEK*[9] foi um projeto iniciado por diversas empresas da indústria alemã, como *BMW*, *Bosch*, *DaimlerChrysler*, *Opel*, *Siemens* e *Volkswagen*, sob coordenação do Instituto de Tecnologia da Informação Industrial[10] da Universidade de *Karlsruhe*[11]. Um projeto semelhante da indústria automotiva francesa, o VDX[12] das empresas *PSA* e *Renault*, juntou-se ao grupo do *OSEK* em 1994, formando o *OSEK/VDX* [OSEK Group, 2004b].

Os fatores motivadores do *OSEK/VDX* foram o grande custo de desenvolvimento devido a aspectos não-funcionais do *software* das unidades de controle e a incompatibilidade entre unidade de controle desenvolvidas por fabricantes distintos devido a diferenças nas interfaces de *hardware* e nos protocolos de comunicação. Os objetivos do *OSEK/VDX* eram facilitar a portabilidade e reusabilidade de *software*. Este padrão inclui uma especificação de interfaces abstratas e independentes de aplicação para os sistemas operacionais de tempo real, para comunicação e para gerenciamento de redes e a especificação de *hardware* e da rede independentes das interfaces com o usuário. Apesar de especificar as interfaces de *software* e *hardware*, o padrão *OSEK/VDX* não inclui detalhes de implementação, então pode-se dizer que este padrão independe da implementação realizada [OSEK Group, 2004a].

Há quatro níveis de conformidade ao padrão *OSEK/VDX*. Isto permite que diferentes sistemas operacionais tenham certificação, mesmo que utilizem modelos de execução levemente diferentes. Os quatro níveis referem-se às combinações de duas variantes no funcionamento do sistema operacional. O primeiro aspecto que pode variar é o número de tarefas por nível de prioridade e o número de vezes que uma mesma tarefa pode ser ativada, o que pode ser limitado a uma única tarefa (níveis *BCC1* e *ECC1*) ou a mais de uma (níveis *BCC2* e *ECC2*). O segundo aspecto de funcionamento que define o nível de conformidade é o suporte a tarefas básicas (níveis *BCC1* e *BCC2*) ou a tarefas estendidas (níveis *ECC1* e *ECC2*) [OSEK Group, 2005].

O escalonamento no *OSEK-VDX* pode ser temporizado ou cooperativo: em ambos os casos, a tarefa em execução é sempre uma tarefa pronta com maior prioridade. No *OSEK-VDX*, há dois tipos de tarefas: básicas e estendidas. A principal diferença é que as tarefas básicas sempre executam enquanto estiverem prontas e forem prioritárias, enquanto as tarefas estendidas podem ficar em um estado de espera, em que aguardam a ocorrência de eventos externos para voltarem a ficar prontas para execução [OSEK Group, 2005].

[9]Abreviação em alemão de *"Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug"*, que significa "Sistemas Abertos e Interfaces Correspondentes para Eletrônica Automotiva".

[10]Em alemão: *"Intitut für Industrielle Informationstechnik"*.

[11]*"Universität Karlsruhe"* em alemão.

[12]Abreviação em inglês de *"Vehicle Distributed Executive"*.

As estruturas disponíveis em sistemas *OSEK-VDX* são: tarefas, eventos e alarmes. Alarmes são utilizados para ativar tarefas após intervalos de tempo, com o objetivo de coordenar operações periódicas. Eventos são associados a tarefas estendidas e podem ser definidos por outras tarefas ou tratadores de interrupção. São utilizados para acordar tarefas no estado de espera. O único mecanismo de sincronização presente no *OSEK-VDX* são os eventos: sua especificação não inclui outros mecanismos tais quais mensagens, semáforos e *mutexes* [OSEK Group, 2005].

No *OSEK-VDX*, todas as funções e tipos de dados são nomeados em *Pascal case*, e geralmente são compostas de um nome de operação seguido do tipo do objeto que deve ser manipulado. Por exemplo, a função que ativa uma tarefa é denominada `ActivateTask` e a função que espera por um evento é denominada `WaitEvent`. Há funções para inicializar e manipular as estruturas, além de funções para controlar o próprio sistemas operacional. A especificação do *OSEK-VDX* define que funções podem ser invocadas em que situação [OSEK Group, 2005].

3.4.3 *FreeRTOS*

O *FreeRTOS* é um sistema operacional de tempo real desenvolvido pela empresa *Real Time Engineers Ltd.*. Segundo [FreeRTOS, 2016], o *FreeRTOS* é líder de mercado, suporta 35 arquiteturas diferentes e é referência em termos de sistemas operacionais de tempo real. Segundo a pesquisa em [UBM Tech, 2015], em 2015, 22% dos desenvolvedores de sistemas embarcados utilizam *FreeRTOS* em seus projetos, sendo apenas menos utilizado que sistemas proprietários e *Android*.

No *FreeRTOS*, diversas estruturas do sistema operacional como tarefas, filas, semáforos, temporizadores, *mutexes* ou grupos de eventos são alocadas dinamicamente. A alocação dinâmica geralmente não é uma operação determinista, então é indesejável em operações de tempo real. Para garantir o determinismo, é possível alocar todas as estruturas necessárias quando a aplicação embarcada começa a executar. Se isto for feito, garante-se o mesmo determinismo com relação à memória que seria observado se fosse utilizada apenas alocação estática. Para que os desenvolvedores de aplicações possam ter controle fino sobre o gerenciamento da memória, é possível definir qual a implementação das funções de alocação e liberação de memória dinâmica, que comportam-se como as funções `malloc` e `free` da biblioteca padrão [Barry, 2009, FreeRTOS, 2016].

O *FreeRTOS* é desenvolvido com base nas normas de codificação da *MISRA* (*Motor Industry Software Reliability Association*) com algumas exceções. Durante seu desenvolvimento, foram estabelecidas também regras referentes ao estilo do código e à nomeação de variáveis, funções, tipos e *macros*. As convenções adotadas para funções, *macros* e tipos afetam diretamente a *API* do *FreeRTOS*. Em geral, nomes de funções e variáveis são prefixadas com uma sequência de caracteres que indica o tipo de retorno: `ul` (`uint32_t`), `us` (`uint16_t`), `uc` (`uint8_t`), `x` (`BaseType_t`, `TickType_t`, `size_t`, e outros inteiros sinalizados não definidos na *stdint*), `ux` (`UBaseType_t` e outros inteiros não sinalizados não definidos na *stdint*), `v` (`void`) e `c` (`char`). Adicionalmente, o prefixo `p` é adicionado às funções ou variáveis que indicam ponteiros. Funções locais são prefixadas com `prv`. Após o prefixo, no caso das funções, segue o nome do arquivo e o nome da função em *Pascal case*. As *macros* são prefixadas com o nome do arquivo em letras minúsculas seguidas pelo nome da *macro* em letras maiúsculas e palavras separadas por *underscores* [Barry, 2009, FreeRTOS, 2016].

Todos os objetos de sistema operacional são manipulados através de *handles*, que são referências abstratas aos objetos e cujo formato é irrelevante para as aplicações de alto nível.

A criação de estruturas do *FreeRTOS* corresponde a duas operações em sequência: alocação de memória e inicialização da estrutura. As funções que criam estruturas podem receber diferentes tipos de parâmetros e têm como valor de retorno o *handle* da estrutura criada. As funções que operam com as estruturas geralmente recebem como primeiro parâmetro o *handle* e retornam códigos de erro [Barry, 2009, FreeRTOS, 2016].

O *FreeRTOS* apresenta uma extensão, denominada *FreeRTOS+IO*, que permite o acesso a periféricos através de uma *API* semelhante à *API POSIX*. O *FreeRTOS+IO* não acompanha a implementação dos *drivers*, é apenas uma camada de abstração que fornece uma forma padronizada para acesso aos periféricos. Como na *API POSIX*, o acesso ao periférico é feito através de quatro funções básicas: `FreeRTOS_open` (abertura do periférico e obtenção do descritor de arquivo), `FreeRTOS_read` (realização de leitura a partir do periférico), `FreeRTOS_write` (realização de escrita no periférico) e `FreeRTOS_ioctl` (configuração do periférico) [FreeRTOS, 2016, FreeRTOS, 2016].

3.4.4 *ChibiOS*

O *ChibiOS* é um sistema operacional de tempo real desenvolvido por Giovanni Di Sirio. Segundo [Di Sirio, 2015], o sistema operacional ocupa pouco espaço e é extremamente eficiente. Oficialmente, o *ChibiOS* tem versões para diversas plataformas, como *Cortex-M* (0, 0+, 3, 4, 7), *ARM7*, *ARM9*, entre outros. Há ainda versões não oficiais para plataformas como *MSP430*, *ARM11* e *AVR*. Este sistema é distribuído sob três licenças diferentes. Ele pode ser utilizado sob licença *GPLv3* para aplicações de código aberto ou sob licenças comerciais pagas ou gratuitas para aplicações de código fechado. A camada de abstração de acesso ao *hardware* (*HAL*) do *ChibiOS* é distribuída sob a licença Apache. O *ChibiOS* para a plataforma *Cortex-M* acompanha uma camada compatível com *CMSIS-RTOS*.

No *ChibiOS*, as estruturas internas do sistema operacional e as utilizadas pelas aplicações como semáforos e filas de mensagens são estaticamente alocadas. Com a alocação estática, há verificação do uso de memória em tempo de compilação e não há aumento do custo computacional devido à alocação e à liberação de memória. Estas estruturas funcionam internamente como listas encadeadas, então todas as operações relacionadas à sua listagem (como a manutenção da lista de *threads* ativas) ocorrem em tempo linear. Também é possível utilizar alocação dinâmica com o *ChibiOS*. As funções do sistema operacional são divididas de acordo com o estado do sistema operacional em que podem ser invocadas. Por exemplo, funções diferentes devem ser chamadas para subir um semáforo em *threads* de aplicação e em códigos internos do *kernel* [Di Sirio, 2015].

Algumas convenções são adotadas na *API* do *ChibiOS*. As funções são geralmente nomeadas começando com `ch` (iniciais de *"Chibi"*), seguido do grupo da função (que indica a que tipo de estrutura do sistema operacional a função se refere), do nome da operação (que indica que tipo de operação é realizada sobre a estrutura) e do sufixo (que indica o estado do sistema operacional no qual a função pode ser invocada). Por exemplo, a função que sobe um semáforo a partir de uma *thread* de aplicação é denominada `chSemSignal`, enquanto a função que faz a mesma operação em tratadores de interrupção ou dentro de *locks* de sistema é `chSemSignalI`. Praticamente todas as funções do sistema operacional recebem como primeiro parâmetro o ponteiro para a estrutura que devem manipular, à exceção de funções que dizem respeito ao próprio sistema operacional (neste caso, não há estrutura a manipular) [ChibiOS, 2015b].

O *ChibiOS* ainda apresenta uma camada de abstração de *hardware*[13] padronizada para as diversas arquiteturas nas quais foi implementado, o que inclui acesso padronizado a periféricos seriais, *RTC*, entre outros [Di Sirio, 2015]. A *HAL* do *ChibiOS* foi projetada para ser utilizável com outros sistemas operacionais, então a forma que interage com o sistema operacional foi projetada para ser desacoplável do *ChibiOS*. As chamadas da *HAL* para o sistema operacional são feitas através de uma camada de abstração denominada *OSAL*[14]. Para utilizar um sistema operacional diferente do *ChibiOS*, basta implementar a *OSAL* sobre o outro sistema, o que pode ser feito através de funções ou *macros* [ChibiOS, 2015a].

Na *API* da *HAL* do *ChibiOS*, convenções diferentes são adotadas para nomear as funções. As funções são nomeadas começando com o nome do dispositivo (por exemplo, `adc`), seguidas do nome do comando a ser executado. Em alguns lugares da *HAL*, as convenções da *API* do sistema operacional referentes aos sufixos dos nomes das funções são adotados. Na *HAL*, cada dispositivo é representado por uma estrutura de dados estaticamente alocada. Da mesma forma que as funções do sistema operacional em si, as funções da *HAL* geralmente recebem como primeiro parâmetro o ponteiro para a estrutura de dados que representa o dispositivo [ChibiOS, 2015a].

Na *HAL* do *ChibiOS*, diversos *drivers* podem ser abstraídos na forma de canais de entrada e saída (*"I/O Channels"*), arquivos (*"Files"*), dispositivos de blocos de entrada e saída (*"I/O Block Devices"*) e fluxos (*"Streams"*). Em todos os casos, a leitura e a escrita de dados pode ser realizada através de funções genéricas e independentes do *driver*. Por exemplo, o *driver* de serial da *HAL* do *"ChibiOS"* implementa um canal de entrada e saída, então pode ser utilizado tanto através de funções específicas do *driver* quando através de funções genéricas que operam sobre canais [ChibiOS, 2015a].

3.4.5 Discussão

Sistemas operacionais embarcados foram estudados com o objetivo de verificar aspectos comuns de *APIs* para sistemas deste tipo. O *CMSIS-RTOS* é uma definição de *API* e pode ser construído como uma camada de abstração sobre sistemas operacionais embarcados existentes. Este conceito é similar ao sugerido nesta dissertação, mas direcionado a sistemas operacionais, não a bibliotecas criptográficas. Foram consideradas muitas das estratégias do *CMSIS-RTOS* no projeto da *API* criptográfica, como:

- Padronização de nomes de funções, tipos de dados, entre outros.
- Separação entre definição de estruturas de dados e identificadores das estruturas inicializadas. Isto dá maior liberdade às implementações para utilizarem as estratégias que acharem mais adequadas.
- Utilização de prefixo único em todas as funções e estruturas.
- Uso de tipos de dados opacos que escondem detalhes de implementação.

O *OSEK/VDX* também é uma *API* criptográfica unificada, mas direcionado a sistemas veiculares. No *OSEK/VDX*, também ha padronização dos nomes de funções.

[13] *"Hardware abstraction layer"* em inglês, abreviado como *HAL*.

[14] Sigla de *"Operating System Abstraction Layer"*, que significa "Camada de Abstração do Sistema Operacional".

Com relação às *APIs* do *FreeRTOS* e do *ChibiOS*, notou-se que expõem funcionalidades muito semelhantes (semáforos, *mutexes*, filas de mensagens, *threads*, entre outros), mas utilizam estratégias um pouco diferentes. Enquanto o *FreeRTOS* depende constantemente do uso de alocação dinâmica de memória, o *ChibiOS* utiliza memória estática alocada em tempo de compilação. Como no caso do *CMSIS/RTOS*, a *API* elaborada foi elaborada de forma agnóstica à estratégia de alocação de memória utilizada pelas implementações.

3.5 Bibliotecas criptográficas

É comum utilizar bibliotecas que implementam primitivas criptográficas, como *AES* para criptografia simétrica e *RSA* para criptografia assimétrica, na construção de outros sistemas seguros [Bernstein et al., 2012]. Dois exemplos de bibliotecas amplamente utilizadas são *OpenSSL*, que é mais adequada para sistemas de computação geral, e *wolfSSL*, anteriormente chamada *cyaSSL* [wolfSSL, 2016], que é mais adequada a sistemas embarcados e de tempo real [Hatzivasilis et al., 2014].

A maioria das bibliotecas criptográficas são muito grandes para serem utilizadas em dispositivos limitados ou muito limitados, como nós de redes de sensores. Aplicações leves e ultra leves precisam de apenas um subconjunto pequeno das funcionalidades que uma biblioteca criptográfica completa é capaz de fornecer. Bibliotecas projetadas para uso em sistemas embarcados devem suportar primitivas criptográficas otimizadas em termos de tamanho, velocidade ou consumo energético [Hatzivasilis et al., 2014].

No padrão *PKCS* #11, sugeriu-se uma *API* padronizada denominada *Cryptoki*, para acesso a dispositivos criptográficos. Este padrão é definido para a linguagem *ANSI* C e tenta ser o mais genérico possível. Dispositivos criptográficos são entendidos como elementos que armazenam informações relevantes às operações criptográficas e realizam-nas, e podem ser dispositivos externos ou mesmo módulos de *software*. Este padrão é bastante extenso e inclui funções para diversas finalidades, como cifragem, decifragem, autenticação de mensagens (por *MAC*), assinatura digital e geração de números aleatórios, entre outros. Implementações do padrão *Cryptoki* devem isolar as aplicações de alto nível da implementação das primitivas criptográficas [Griffin and Fenwick, 2015a].

A biblioteca criptográfica *OpenSSL* tem código aberto e foi desenvolvida para utilização em aplicações de uso geral, apresentando grandes desempenho e segurança. Ela contém um conjunto quase completo de funcionalidades criptográficas, mas não incorpora novos padrões [Hatzivasilis et al., 2014] como *SHA*-3, pelo menos até sua versão 1.0.2 [OpenSSL, 2015]. Apesar de ser bastante eficiente, seu uso em sistemas embarcados é muitas vezes proibitivo, já que o tamanho de seu código executável tende a ser grande [Hatzivasilis et al., 2014].

A biblioteca *wolfSSL* é de código aberto, sob licença *GPLv2*, mas tem uma opção comercial que pode ser utilizada em aplicações de código fechado. Seu objetivo é uso em sistemas embarcados que utilizam sistemas operacionais de tempo real. É uma biblioteca leve com alto desempenho e suporta diversos padrões criptográficos mais novos. Sua *API* é bastante simples e bem documentada, e acompanha uma camada que permite compatibilidade com a *API* da *OpenSSL* [Hatzivasilis et al., 2014]. Segundo [wolfSSL, 2016], a *wolfSSL* é aproximadamente 20 vezes menor que *OpenSSL* e é também mais eficiente para a vasta maioria das operações relacionadas a *SSL*.

A biblioteca criptográfica *ULCL*[15], foi projetada para uso em sistemas embarcados, sendo otimizada tanto em termos de tamanho de código executável quanto em termos de desempenho. Esta biblioteca pode ser configurada para que em tempo de compilação apenas os módulos necessários sejam considerados, diminuindo o tamanho do *software* final. As funções criptográficas que compõem esta biblioteca são na verdade retiradas de diversas outras implementações de código aberto [Hatzivasilis et al., 2014].

A biblioteca criptográfica *Edon*, implementada em C, é extremamente leve e foi projetada para uso em sistemas embarcados. Ela utiliza quasigrupos para implementar primitivas criptográficas e contém uma cifra de bloco, uma cifra de fluxo, uma função de resumo criptográfico (*hash*) e um gerador de números pseudoaleatórios [Hatzivasilis et al., 2014].

A biblioteca criptográfica *NaCl*[16], chamada pelos autores de "sal" (pois seu nome corresponde à fórmula química para o sal de cozinha), é de domínio público. Enquanto outras bibliotecas fornecem operações elementares, as quais são combinadas, por exemplo, para cifrar e decifrar mensagens ou verificar autenticidade de dados, a biblioteca *NaCl* fornece funções complexas mais específicas, que realizam com uma só chamada de função operações como cifragem ou decifragem de dados e assinatura digital de dados [Bernstein et al., 2012].

Na *RFC* 2628 publicada pela *IETF*, foi apresentada uma *API* de criptografia. Esta *API* é sugerida como uma interface padronizada de acesso a serviços criptográficos a partir de aplicações conectadas à *Internet*. A *RFC* inclui um apêndice contendo o código de um arquivo de cabeçalho em linguagem *C* com as funções da *API* [Smyslov, 1999].

Na *RFC* 2743 da *IETF*, sugere-se uma *API* padronizada de acesso a serviços de segurança, a *GSSAPI*. A versão atual deste especificação tornou obsoleta a versão anterior, que havia sido definida na *RFC* 2078 de 1997, que por sua vez substituiu a *RFC* 1508 de 1993. Seu objetivo é abstrair o acesso a serviços de segurança de forma independente dos mecanismos subjacentes, permitindo uma maior portabilidade no que diz respeito ao código-fonte. Apesar de prover diversas funções para proteger comunicação garantindo autenticação, integridade e confiabilidade, esta *API* não inclui funções específicas para cifragem e decifragem [Linn, 2000].

Dentre as bibliotecas e padrões apresentados, selecionaram-se três que merecem destaque especial: *wolfSSL*, *OpenSSL* e *Cryptoki*. A *wolfSSL* é de código aberto e é amplamente utilizada. A *OpenSSL* é disponível para uso em aplicações de código aberto e comerciais, e sua especificação é aberta. A *Cryptoki* foi escolhida pois o *PKCS* #11 também é um padrão disponível publicamente. Estudos detalhados a respeito destas três *APIs* estão apresentados a seguir.

3.5.1 *Cryptoki*

A *API Cryptoki*, definida no padrão *PKCS* #11, permite o acesso a "dispositivos criptográficos" abstratos, que geralmente são dispositivos externos de *hardware*, como *smart cards*, mas também podem ser compostos exclusivamente de *software*. A *Cryptoki* foi elaborada de forma a ser independente do sistema operacional utilizado na aplicação. A representação de um dispositivo criptográfico na *Cryptoki* é denominada de *"token"*, e a conexão entre a aplicação e o *token* é denominada de "sessão". Para utilizar um dispositivo criptográfico usando a *Cryptoki*, é sempre necessário estabelecer uma sessão com o respectivo *token* [Griffin and Fenwick, 2015a].

[15]Sigla de *"Ultra-Lightweight Cryptographic Library"*, que significa "Biblioteca Criptográfica Ultra-Leve".

[16]Abreviação de *"Networking and Cryptography Library"*, que significa "Biblioteca de Redes e Criptografia".

Cada dispositivo com o qual a *Cryptoki* trabalha expõe uma ou mais operações criptográficas, denominadas de "mecanismos". É possível enumerar todos os mecanismos de um *token* antes de estabelecer uma sessão com ele. Na biblioteca, entendem-se como dados básicos a respeito de um mecanismo criptográfico: o tamanho das chaves e um grupo de *flags* que indicam suas capacidades básicas (por exemplo, se o mecanismo é capaz de realizar criptografia simétrica, criptografia assimétrica, cálculo de resumo criptográfico, entre outros). Os dados básicos a respeito do mecanismo também podem ser lidos antes de a aplicação estabelecer uma sessão com o *token* [Griffin and Fenwick, 2015a].

A biblioteca *Cryptoki* trabalha com objetos que representam diversas entidades envolvidas em operações criptográficas. Os objetos da *Cryptoki* são sempre consistentes e não necessitam de nenhum tipo de inicialização após a criação. A criação de objetos é feita através de funções especializadas que recebem como parâmetros estruturas de dados denominadas *"templates"*, que armazenam atributos. Após a criação, cada objeto guarda uma série de atributos que podem ser lidos e alterados a partir do código da aplicação [Griffin and Fenwick, 2015a].

Há três grandes classes às quais os objetos utilizados pela *Cryptoki* podem pertencer. A primeira grande classe agrupa objetos que representam características de componentes de *hardware*. Cada uma de suas subclasses representa o conjunto de características de um tipo diferente de componente de *hardware*, como *RTC*, contador monotônico e a interface gráfica. A segunda grande classe da *Cryptoki* agrupa todos os objetos que são relacionadas a armazenamento de dados, com subclasses especializadas para cada tipo de dado ou conjunto de dados. A terceira grande classe da *Cryptoki* refere-se aos objetos de mecanismo, que armazenam dados adicionais sobre os mecanismos criptográficos, isto é, além dos dados básicos [Griffin and Fenwick, 2015a].

A classe de objetos que são relacionados a armazenamento de dados é dividida em uma série de subclasses. A primeira subclasse corresponde aos objetos de dados, que armazenam dados binários cujo formato é arbitrário e definido pela aplicação. A segunda subclasse se refere a certificados digitais, que podem ter diferentes formatos, cada qual implementado por uma subclasse diferente. A terceira subclasse refere-se aos objetos de chave, que representam chaves criptográficas. As chaves de cada tipo (privada, pública, secreta) e algoritmo têm atributos diferentes. A quarta subclasse refere-se aos objetos que armazenam parâmetros de domínio, isto é, parâmetros adicionais de alguns algoritmos de criptografia [Griffin and Fenwick, 2015a].

Com relação à *API*, a biblioteca segue algumas convenções de nomeação e forma. Na *Cryptoki*, todas as funções da biblioteca começam com o prefixo `C_` (inicial de *"Cryptoki"*). A maior parte das funções retorna um código indicando sucesso ou um número indicativo da falha. Os objetos são manipulados através de *handles*, tipos abstratos que referenciam objetos e cujo formato interno é irrelevante para a aplicação. Em geral, as funções que criam objetos devem receber como parâmetros o ponteiro para a variável que deve armazenar o *handle* e ponteiros para estruturas com os dados para inicialização. Todas as funções que necessitam de interação com o dispositivo criptográfico também devem receber como parâmetro o *handle* da sessão estabelecida com o dispositivo [Griffin and Fenwick, 2015a].

Outra convenção existente na *Cryptoki* diz respeito às funções que operam sobre conjunto de dados, como as que devolvem os dados processados resultantes de operações criptográficas. Todas as funções deste tipo devem receber como parâmetro o ponteiro para o *buffer* que deve conter os dados resultantes e um ponteiro para uma variável que deve armazenar o tamanho do *buffer*. O comportamento comum destas funções é o seguinte: a variável de tamanho contém inicialmente o tamanho máximo do *buffer* de saída, e, quando a operação é realizada

com sucesso, a variável recebe a quantidade de dados efetivamente preenchidos no *buffer*. O outro comportamento destas funções é observado quando o *buffer* de saída fornecido é nulo; neste caso, a função retorna com sucesso mas não faz nada além de armazenar na variável com o tamanho a quantidade mínima de dados serão escritos no *buffer* se a operação for realizada com sucesso [Griffin and Fenwick, 2015a].

A *API* tem uma série de funções de manipulação de objetos. Há funções para inicialização da própria *Cryptoki*, para manutenção de seções, para gerenciamento de objetos, para cifragem e decifragem, para assinatura, entre outros. Como a biblioteca é independente do sistema operacional utilizado, não é possível realizar diretamente a partir dela operações como criação de semáforos e *mutexes*. Para que a biblioteca possa acessar tais recursos, é possível fornecer para ela a implementação de diversas operações relacionadas ao sistema operacional. Isto é feito nas funções de inicialização da biblioteca, em que é possível fornecer ponteiros para funções que implementam operações que dependem do sistema operacional, como controle de concorrência [Griffin and Fenwick, 2015a].

Com relação às operações fundamentais da biblioteca, merecem destaque as funções de gerenciamento de objetos, de gerenciamento de chaves e de realização das operações de cifragem, decifragem, resumo criptográfico e assinatura. As funções de manutenção de sessão e para outras finalidades não serão descritas neste documento pois dizem pouco respeito à realização em si das operações criptográficas [Griffin and Fenwick, 2015a].

As funções de gerenciamento de objetos estão apresentadas na tabela 3.1. As responsáveis por criação de objetos e por manipulação de atributos são fundamentais para todas as operações envolvendo objetos, e são provavelmente as funções mais importantes desta lista [Griffin and Fenwick, 2015a].

Tabela 3.1: Operações com objetos da *Cryptoki*.

`C_CreateObject`	Cria um objeto.
`C_CopyObject`	Cria um objeto que é cópia de outro.
`C_DestroyObject`	Destrói um objeto.
`C_GetObjectSize`	Obtém o tamanho de um objeto em bytes.
`C_GetAttributeValue`	Obtém o valor de um atributo de um objeto.
`C_SetAttributeValue`	Altera o valor de um atributo de um objeto.
`C_FindObjectsInit`	Inicializa uma operação de busca de objetos.
`C_FindObjects`	Continua uma operação de busca de objetos.
`C_FindObjectsFinal`	Finaliza uma operação de busca de objetos.

As funções de gerenciamento de chaves estão apresentadas na tabela 3.2. Estas funções representam as operações fundamentais realizáveis com chaves de criptografia simétrica e assimétrica [Griffin and Fenwick, 2015a].

Tabela 3.2: Operações com chaves da *Cryptoki*.

`C_GenerateKey`	Gera uma chave criptografia simétrica.
`C_GenerateKeyPair`	Gera um par de chaves de criptografia assimétrica.
`C_WrapKey`	Cifra uma chave.
`C_UnwrapKey`	Decifra uma chave.
`C_DeriveKey`	Cria uma chave derivada a partir de outra chave.

As funções que realizam operações de cifragem de dados estão apresentadas na tabela 3.3, as de decifragem, na tabela 3.4, as de cálculo de resumo criptográfico (*hash*), na tabela 3.5, e as de assinatura, na tabela 3.6. Nas quatro listas de funções, as formas de realizar a entrada e de obter a saída de dados seguem as seguintes formas: as funções podem receber os dados e processá-los através de uma única chamada (como no caso de `C_Encrypt`), ou recebê-los por partes, em diversas chamadas, e então terminar a operação (como no caso de `C_EncryptUpdate` e `C_EncryptFinal`, respectivamente) [Griffin and Fenwick, 2015a].

Tabela 3.3: Operações de cifragem de dados da *Cryptoki*.

`C_EncryptInit`	Inicializa uma operação de cifragem.
`C_Encrypt`	Realiza a cifragem de um conjunto de dados.
`C_EncryptUpdate`	Continua a entrada de dados para cifragem.
`C_EncryptFinal`	Termina a entrada de dados e realiza a cifragem dos dados.

Tabela 3.4: Operações de decifragem de dados da *Cryptoki*.

`C_DecryptInit`	Inicializa uma operação de decifragem.
`C_Decrypt`	Realiza a decifragem de um conjunto de dados.
`C_DecryptUpdate`	Continua a entrada de dados para decifragem.
`C_DecryptFinal`	Termina a entrada de dados e realiza a decifragem dos dados.

Tabela 3.5: Operações de cálculo de resumo criptográfico da *Cryptoki*.

`C_DigestInit`	Inicializa uma operação de cálculo de resumo criptográfico.
`C_Digest`	Calcula o resumo criptográfico de um conjunto de dados.
`C_DigestUpdate`	Continua a entrada de dados para cálculo de resumo criptográfico. A entrada são dados binários.
`C_DigestKey`	Continua a entrada de dados para cálculo de resumo criptográfico. A entrada é uma chave.
`C_DigestFinal`	Termina a entrada de dados e calcula o resumo criptográfico dos dados.

3.5.2 *OpenSSL*

OpenSSL é uma implementação de código aberto dos protocolos *TLS* (*"Transport Layer Security"*) e *SSL* (*"Secure Sockets Layer"*). Uma das partes do *OpenSSL* é sua biblioteca criptográfica, denominada *"OpenSSL crypto library"*. Esta biblioteca implementa uma variedade de algoritmos utilizados em padrões da *Internet* [OpenSSL, 2015]. Uma pesquisa feita em 2014, que considerou quase 100 milhões de *sites*, mostrou que mais de dois terços dos *sites* (avaliados) utilizam *OpenSSL* como motor criptográfico [Netcraft, 2014].

Na biblioteca criptográfica do *OpenSSL*, há uma série de convenções relativas à forma de funções e ordem de chamadas que facilitam enormemente seu uso. O subconjunto de funções que seguem essas convenções é denominado *"EVP"*. As funções de *EVP* operam sobre objetos que representam algoritmos criptográficos, que podem ser dos tipos `EVP_CIPHER` (cifras

Tabela 3.6: Operações de assinatura da *Cryptoki*.

C_SignInit	Inicia uma operação de geração de assinatura.
C_Sign	Gera a assinatura de um conjunto de dados.
C_SignUpdate	Continua a entrada de dados para geração de assinatura.
C_SignFinal	Termina a entrada de dados e gera a assinatura dos dados.
C_SignRecoverInit	Inicia uma operação de geração de assinatura em que os dados podem ser recuperados a partir da assinatura.
C_SignRecover	Gera a assinatura de um conjunto de dados em que os dados podem ser recuperados a partir da assinatura.
C_VerifyInit	Inicia uma operação de verificação de assinatura.
C_Verify	Verifica a assinatura de um conjunto de dados.
C_VerifyUpdate	Continua a entrada de dados para verificação de assinatura.
C_VerifyFinal	Termina a entrada de dados e verifica a assinatura dos dados.
C_VerifyRecoverInit	Inicia uma operação de verificação de assinatura em que os dados podem ser recuperados a partir da assinatura.
C_VerifyRecover	Verifica a assinatura de um conjunto de dados em que os dados podem ser recuperados a partir da assinatura.

simétricas ou assimétricas) e EVP_MD (resumo criptográfico). Estes objetos podem ser obtidos através de funções dedicadas cujos nomes seguem a seguinte convenção: EVP, seguido de um *underscore*, seguido do nome do algoritmo em letras minúsculas, e opcionalmente seguido de outros parâmetros. Por exemplo, a função que obtém o objeto EVP_CIPHER que representa o *AES* em modo *CBC* com chave de 192 *bits* é EVP_aes_192_cbc, e a função que obtém o objeto EVP_MD que representa o *MD5* é EVP_md5 [OpenSSL, 2013].

O estado interno de um algoritmo criptográfico é armazenado em objetos dos tipos EVP_CIPHER_CTX e EVP_MD_CTX. Assim, antes de realizar cada operação criptográfica, uma instância de EVP_CIPHER_CTX ou de EVP_MD_CTX deve ser criada. Esta instância deve ser utilizada em todas as chamadas de funções que utilizam este algoritmo criptográfico. Ao final do cálculo, esta instância deve ser destruída [OpenSSL, 2013].

Os objetos EVP_MD e EVP_CIPHER são manipulados através de funções cujos nomes também seguem certas convenções. Geralmente, as funções são nomeadas começando com EVP, seguido de um *underscore*, seguido do nome da operação em *Pascal case* (por exemplo, Encrypt ou Digest), seguido da etapa da operação. A primeira chamada de uma operação criptográfica recebe o sufixo Init, as etapas intermediárias recebem o sufixo Update e a etapa final recebe o sufixo Final [OpenSSL, 2013].

É possível acessar diretamente os mecanismos criptográficos, sem utilizar a camada *EVP*. Há inclusive mecanismos que só podem ser acessados diretamente, pois não há abstração para suas funcionalidades na camada *EVP*, como é o caso do *RSA*. Ao acessar diretamente um algoritmo, usam-se funções especializadas e estruturas de dados específicas. Neste caso, os nomes das estruturas e das funções não seguem normas de nomeação, mas geralmente começam com o nome do algoritmo. Quando há suporte ao algorítimo na camada *EVP*, o uso da camada é mais indicado que o acesso direto aos algoritmos [OpenSSL, 2015].

3.5.3 *wolfSSL*

A biblioteca *wolfSSL*, desenvolvida pela empresa homônima, é voltada para aplicações embarcadas e implementa desde operações simples de cifragem e decifragem até funções para utilização de *SSL* sobre *sockets*. O subconjunto da *wolfSSL* que implementa as funções criptográficas básicas denomina-se de *wolfCrypt*. A *wolfSSL* acompanha um conjunto de *wrappers* que é compatível com a *API* da biblioteca *OpenSSL*. Com isto, a adaptação de aplicações que utilizam *OpenSSL* para utilizarem a *wolfSSL* torna-se muito mais simples. A *wolfSSL* foi desenvolvida de forma a ser independente do sistema operacional utilizado [wolfSSL, 2016].

Em algumas arquiteturas, a implementação da *wolfCrypt* utiliza aceleração de *hardware* para implementar suas operações. Por exemplo, em processadores *Intel* da família *x86* que suportam a teconologia *AES-NI*, que inclui instruções dedicadas à realização de etapas intensivas do *AES*, a *wolfSSL* pode utilizar as instruções dedicadas para acelerar o cálculo. Em processadores *STM32* (da família *Cortex-M3*) que suportam tecnologias de aceleração de *hardware* para operações criptográficas e geração de números aleatórios, a *wolfSSL* pode utilizar estes recursos.

Para implementar as funções referentes a *SSL*, a *wolfSSL* utiliza a abstração de *sockets* como descritores de arquivo. O sistema operacional subjacente pode abstrair operações com arquivos de diferentes formas ou mesmo não apresentar os conceitos de "arquivo" e "descritor de arquivo". Como a *wolfSSL* deve ser independente do sistema operacional, é necessário informar à biblioteca como são feitas leituras e escritas de dados a partir dos descritores de arquivo. Isto é feito fornecendo à biblioteca ponteiros para as funções que devem ser utilizadas para realizar estas operações. Este mecanismo permite a utilização da *wolfSSL* junto a sistemas operacionais ou pilhas que não utilizam descritores de arquivo para representar os *sockets* ou mesmo para realizar comunicação por *SSL* em canais de comunicação diferente de *sockets*. Para os sistemas operacionais e pilhas oficialmente suportados pela *wolfSSL*, são fornecidas na própria bilbioteca implementações básicas das funções de leitura e escrita.

Além das funções de leitura e escrita sobre descritores de arquivo, outras funções podem ser fornecidas pela aplicação para personalizar e adequar o funcionamento da biblioteca. Entre as funções customizáveis estão as de alocação e liberação de memória, cópia de dados e outras relacionadas ao sistema operacional. É possível também fornecer à biblioteca funções que são notificadas de alguns eventos internos da *wolfSSL*, como término de etapas do cálculo criptográfico e estouro de limites de tempo.

Na *wolfCrypt* há funções específicas para implementar cada operação relacionada a cada algoritmo de criptografia, então sua lista de funções é bastante extensa. O conjunto completo de funções da *wolfCrypt* pode ser encontrado em [wolfSSL, 2016]. Apesar de haver um número imenso de funções, é possível observar que o formato, o nome e a forma de utilizá-las seguem algumas regras.

Para cada algoritmo criptográfico (cifragem, decifragem, resumo criptográfico, etc) há um tipo de estrutura de dados que armazena dados relevantes para o cálculo. O formato destas estruturas não é relevante para as aplicações de alto nível. Cada estrutura deste tipo tem o mesmo nome do algoritmo correspondente então, por exemplo, a estrutura que armazena dados para cálculo de *AES* denomina-se `Aes`. Grande parte as funções referentes ao algoritmo são precedidas com `wc_` (iniciais de *"wolfCrypt"*) seguido do nome do algoritmo então, por exemplo, as funções referentes a *AES* são geralmente iniciadas com `wc_Aes`. Há exceções a esta regra, como a função `wc_InitRsaKey`, em que o nome do algoritmo (no caso, *RSA*)

não é imediatamente posterior ao prefixo `wc_`. O restante dos nomes das funções é escrito em *Pascal case*.

Em geral, as funções da *wolfCrypt* que realizam criptografia por blocos recebem blocos completos de dados e escreve blocos processados completos nos *buffers* de saída, através de funções com nomes contendo `Encrypt` e `Decrypt`. Por outro lado, as funções que realizam criptografia em fluxo recebem conjuntos de dados de qualquer tamanho como entrada e salvam os resultados imediatamente nos *buffers* de saída, através de funções com nomes contendo `Process`. As funções de cálculo de resumo criptográfico (*hash*) recebem dados em partes e fornecer o resultado do cálculo ao final, através de funções com nomes contendo `Update` e `Final`.

3.5.4 Discussão

Para obter um conjunto mínimo de funcionalidades que uma biblioteca criptográfica deve conter, foram estudadas as *APIs* das bibliotecas criptográficas *OpenSSL* e *wolfSSL* e do padrão *Cryptoki*. Também foi considerado o estudo em [Clulow, 2015], em que está listado um conjunto reduzido de funções que uma *API* criptográfica possui. O conjunto em [Clulow, 2015] não contempla resumo criptográfico como parte deste conjunto mínimo, mas esta função está presente nas duas bibliotecas estudadas e na *Cryptoki*. Naquele trabalho também considerou-se cifragem e decifragem, independente da natureza das chaves, como uma operação única. No conjunto listado para a *GEmSysC*, entretanto, pareceu mais adequado separar cifragem e decifragem (pois são operações diferentes) e criptografia simétrica de assimétrica (já que os algoritmos de cifragem e decifragem usam chaves diferentes). Este conjunto está condensado a seguir:

1. Cifragem de dados com chaves simétricas.
2. Decifragem de dados com chaves simétricas.
3. Cifragem de dados com chaves assimétricas.
4. Decifragem de dados com chaves assimétricas.
5. Cálculo de resumo criptográfico (*hash*).

Inicialmente, considerou-se um conjunto maior de funcionalidades, que também englobava as operações de verificação de resumo criptográfico, geração e verificação de códigos de autenticação de mensagem[17] e geração e verificação de assinaturas digitais. A verificação de resumo criptográfico pode ser construída a partir da operação de geração de resumo criptográfico e da comparação do resultado com o valor esperado, então não foi incluída na lista reduzida. Apesar de operação de geração e verificação de código de autenticação de mensagem ter várias variantes, sendo que nem todas podem ser implementadas com composição das operações da lista, formas populares destas operações podem ser compostas a partir de resumo criptográfico [NIST, 2008] ou cifragem/decifragem [ISO/IEC, 2011]. Assim, a geração e a verificação de códigos de autenticação de mensagem também foram excluídas da lista. As assinaturas digitais podem ser geradas e verificadas de diversas formas, mas no caso de assinatura com *RSA*,

[17]Em inglês, *"message authentication code"* ou *MAC*.

elas são geradas em função de resumo criptográfico, cifragem e decifragem. Como a assinatura com *RSA* é uma das técnicas mais populares em uso [Bos et al., 2013], operações relacionadas a assinaturas digitais não foram incluídas no conjunto reduzido. As operações de geração e verificação de assinaturas digitais também foram excluídas. Assim, o conjunto reduzido apresentado anteriormente foi utilizado como base na especificação da *GEmSysC*. A exclusão de certas funcionalidades do conjunto mínimo de funções para a primeira versão não impede que sejam posteriormente incorporadas à *GEmSysC*, já que mecanismos especializados para realizar estas operações permitirão que ela tenha suporte a outros tipos de código de autenticação de mensagem e de assinatura digital.

3.6 Análise

A *API* criptográfica proposta neste documento refere-se a uma interface unificada de acesso a funcionalidades criptográficas. Esta *API* foi elaborada para utilização em sistemas embarcados, então características destes sistemas foram consideradas ao especificar a *API*. Neste capítulo, foi apresentado o estudo que embasou as decisões relativas a esta *API*, apresentadas no capítulo 4.

Foram apresentados neste capítulo estudos sobre interfaces de *softare*. Nestes estudos, notou-se que alguns autores ([Blanchette, 2008], [Rama and Kak, 2015]) consideram importante que uma *API* seja consistente. Tentou-se garantir a consistência de comportamento na especificação da *API* criptográfica. Isto foi feito utilizando-se sempre formas semelhantes para declarar estruturas, inicializá-las, utilizá-las e destruí-las.

As bibliotecas e *APIs* voltadas para sistemas embarcados consideradas foram todas relativas a sistemas operacionais de tempo real. Notou-se uma certa similaridade entre elas, pois todos manipulam estruturas semelhantes, como *mutexes*, semáforos e filas de mensagens, e pois suas chamadas são geralmente bloqueantes com *timeout.* exceto com relação às estratégias de gerenciamento de memória utilizadas. Entre as especificações analisadas, são assumidas e utilizadas duas formas distintas de gerenciamento de memória: alocação estática (*ChibiOS*) e alocação dinâmica (*FreeRTOS*). O padrão *CMSIS-RTOS* é a especificação de uma camada de abstração acima de um outro sistema operacional. Neste padrão, a forma de declarar as estruturas de dados é uniforme e não depende da forma que o operacional subjacente utiliza alocação de memória (estática ou dinâmica). A forma de declarar estruturas do *CMSIS-RTOS* foi utilizada como inspiração na criação da *API* criptográfica.

Ao contrário do que se observou sobre sistemas operacionais embarcados, notaram-se apenas similaridades superficiais entre as diferentes bibliotecas e *API* criptográficas analisadas. Em todas elas, é possível instanciar mecanismos criptográficos, operar sobre as instâncias e finalizá-las assim que não forem mais ser utilizadas. Por outro lado, em cada uma delas, usa-se uma forma diferente para declarar e utilizar instâncias dos mecanismos criptográficos.

A lista reduzida de operações sugerida para a *API* criptográfica não inclui operações relacionadas a códigos de autenticação de mensagem e a assinaturas digitais, pois elas muitas vezes podem ser feitas através de combinações de resumo criptográfico, cifragem e decifragem. O *NIST*[18], em seu padrão *FIPS*[19] 198-1, especifica formas de gerar códigos de autenticação

[18] *"National Institute of Standards and Technology"*, que significa "Instituto Nacional de Padrões e Tecnologia", um órgão do governo dos Estados Unidos da América.

[19] *"Federal Information Processing Standard"*, que significa "Padrão de Processamento de Informação Federal".

de mensagem baseados em resumo criptográfico [NIST, 2008], e a *ISO*[20] e a *IEC*[21], no padrão *ISO/IEC* 9797-1 definem formas de gerar códigos de autenticação de mensagem baseados em cifragem em blocos [ISO/IEC, 2011]. Ainda assim, há formas de gerar e verificar códigos de autenticação de mensagem que não podem ser construídas a partir de resumo criptográfico, cifragem ou decifragem. O *NIST*, em seu padrão *FIPS* 186-4, especifica apenas três técnicas de geração de assinaturas digitais: *DSA*[22], assinatura com *RSA* e *ECDSA*[23][NIST, 2013]. Destes, apenas a assinatura com *RSA* usa criptografia assimétrica, enquanto os demais não podem ser construídos simplesmente como composição de outras operações. Em [Bos et al., 2013], o *RSA* é citado como um dos métodos mais utilizados de assinatura digital. O fato de operações explícitas de geração e verificação de códigos de autenticação de mensagem e assinaturas digitais não terem sido incluídos como parte da *API* genérica impede que sejam realizadas operações destes tipos que não sejam meras composições de cifragem, decifragem e resumo criptográfico. Estes tipos de operação devem ser incluídos em versões posteriores da *API*, de forma a aumentar sua abrangência.

[20] *"International Organization for Standardization"*, que significa "Organização Internacional para Padronização".

[21] *"International Electrotechnical Commission"*, que significa "Comissão Eletrotécnica Internacional".

[22] *"Digital Signature Algorithm"*, que significa "Algoritmo de Assinatura Digital".

[23] *"Elliptic Curve Digital Signature Algorithm"*, que significa "Algoritmo de Assinatura Digital com Curvas Elípticas".

Capítulo 4

GEmSysC

Esta dissertação descreve o processo de elaboração de uma *API* criptográfica unificada para sistemas embarcados. Neste capítulo, está descrita esta *API*, batizada de "API Criptográfica Genérica para Sistemas Embarcados" (em inglês, *"Generic Embedded Systems Cryptographic API"* ou *GEmSysC*).

4.1 Justificativa

Na seção de trabalhos correlatos, apresentaram-se *APIs* existentes na área de sistemas embarcados, tanto de sistemas operacionais quanto de bibliotecas criptográficas. A pesquisa mostrou que o escopo destas *APIs* é limitado: ou elas são voltadas para um nicho específico (como *CMSIS* e *OSEK-VDX*) ou referem-se a implementações em particular (como *FreeRTOS*, *ChibiOS* e *wolfSSL*). A relação de tecnologias consideradas na pesquisa está apresentada na figura 4.1. Nesta figura, as *APIs* estão agrupadas de acordo com três critérios: se são direcionadas a sistemas embarcados (e portanto se consideram as limitações destes sistemas), se são independentes da arquitetura computacional e se são independentes da implementação.

Na figura 4.1, observa-se que apenas a *API OSEK/VDX* atende aos três critérios considerados. Entretanto, segundo [Renaux, 2014], o padrão *CMSIS-RTOS*, apesar de ter sido elaborado exclusivamente para a plataforma *Cortex-M*, poderia ser utilizado em outras plataformas. Se esta possibilidade for considerada, o padrão *CMSIS-RTOS* passa a atender a todos os critérios considerados. A *API* desse padrão somente incorpora interfaces funcionais entre aplicações e o sistema operacional, e portanto não inclui especificações de interfaces relacionadas a funcionalidades criptográficas. Assim, por mais que se considere que o *CMSIS-RTOS* atende aos três critérios considerados na figura 4.1, tanto ele quanto o *OSEK/VDX* não resolvem a falta de *APIs* criptográficas padronizadas para sistemas embarcados.

Segundo a pesquisa em [UBM Tech, 2015], no ano de 2015, 39% dos desenvolvedores de sistemas embarcados afirmam utilizar o mesmo sistema operacional em diferentes projetos para manter a compatibilidade do *software*, 35% afirmam que o fizeram para aproveitar o conhecimento a respeito de um outro sistema operacional e 22% afirmaram que o fizeram pois trocar de sistema operacional pode ser muito caro ou tomar muito tempo. Não foi encontrada nenhuma pesquisa a respeito das dificuldades encontradas por desenvolvedores com relação à criptografia em sistemas embarcados. As três razões dizem respeito às interfaces de *software*: no primeiro caso, o problema é a compatibilidade por si só, no segundo, os problemas são a perda de tempo aprendendo a nova *API* e a falta de experiência com ela, enquanto no terceiro, o

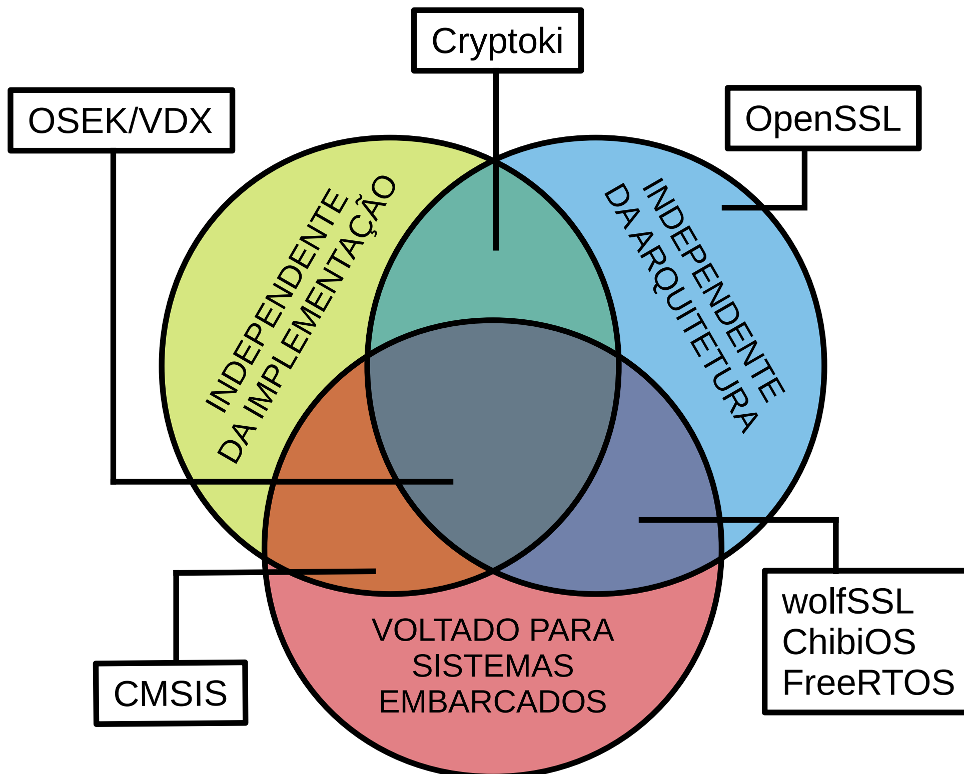

Figura 4.1: Tecnologias estudadas agrupadas por suas características (autoria própria).

problema é o custo de adaptar o *software* às novas interfaces. A compatibilidade de *software* e a redução de tempo de desenvolvimento são características desejáveis em todos os projetos, portanto, assumiu-se que as razões dadas pelos desenvolvedores para evitar migrar entre sistemas operacionais também evitam que eles migrem entre bibliotecas de software em geral, como é o caso de bibliotecas criptográficas.

Segundo [Johnson, 1997] e [Renaux and Pottker, 2014], a existência de padrões simplifica o desenvolvimento de *software*, já que a padronização diminui a curva de aprendizado (que tem de ocorrer uma única vez) e permite maior reaproveitamento de código (já que todo *software* construído sobre estruturas padronizadas utiliza as mesmas interfaces). A existência de uma *API* criptográfica unificada para sistemas embarcados tem o potencial de trazer estes benefícios a todas as aplicações embarcadas.

4.2 Requisitos

A primeira etapa da elaboração da *GEmSysC* envolveu sua especificação. A especificação foi feita com base em estudos do domínio de sistemas embarcados, nos quais observaram-se as necessidades que estes sistemas têm em termos de segurança. Em seguida, estudaram-se algumas *APIs* criptográficas existentes, para investigar que funcionalidades são mais relevantes para implementar soluções seguras e como elas são expostas ao código de aplicação. Nesta seção, estão condensadas as conclusões obtidas durante estes estudos e durante a fase inicial da especificação da *API*.

Segundo [Firesmith, 2003], os requisitos de segurança podem ser resumidos como especificações de quantidades mínimas de subfatores de segurança. No mesmo trabalho, constata-se que os mecanismos de segurança de *software* e *hardware* variam enormemente, mas protegem os mesmos elementos básicos (dados, comunicação, serviços) dos mesmos tipos de ameaças. Uma biblioteca criptográfica fornece mecanismos para proteger integridade, autenticidade, confidencialidade e irretratabilidade dos dados [Firesmith, 2003]. Neste sentido, ela fornece mecanismos que podem ser utilizados para atender aos requisitos de segurança de um sistema de *software*.

Em [Vasilevskaya, 2015], apresentou-se uma metodologia de projeto de sistemas embarcados denominada *SEED*. Segundo a realização desta metodologia no documento em questão, os mecanismos de segurança utilizados em aplicações formam os chamados "blocos de construção de segurança". Os "blocos de construção de segurança" podem ser utilizados na metodologia *SEED*, mas seu conceito é bastante genérico: a implementação de funcionalidades em particular utilizadas para levar segurança a sistemas. Assim, considera-se que este conceito é relevante mesmo fora do contexto do *SEED*. Os blocos de construção de segurança no *software* são funções criptográficas fornecidas pela biblioteca criptográfica ou outros serviços mais complexos construídos a partir destas funções. Portanto, para elaborar uma biblioteca criptográfica é primeiro necessário saber quais são as funções criptográficas que devem compor os "blocos de construção" necessários para construir aplicações de alto nível.

Na seção 3.5.4, foram listadas as funcionalidades básicas de uma biblioteca criptográfica. A lista abaixo, idêntica à apresentada em 3.5.4, foi utilizada como base para a elaboração da *GEmSysC*.

1. Cifragem de dados com chaves simétricas.
2. Decifragem de dados com chaves simétricas.
3. Cifragem de dados com chaves assimétricas.
4. Decifragem de dados com chaves assimétricas.
5. Cálculo de resumo criptográfico (*hash*).

4.3 Características

Nesta seção, estão apresentadas as características principais da *GEmSysC*. Esta seção está organizada da seguinte forma: primeiramente são discutidos aspectos da *API* que buscaram torná-la mais adequada para utilização em sistemas embarcados, e em seguida são discutidas outras convenções adotadas em seu projeto.

4.3.1 Sistemas embarcados

Como sistemas embarcados têm restrições quanto a memória, consumo energético e poder de processamento [Parameswaran and Wolf, 2008], nem sempre é possível utilizar soluções convencionais de *software* junto a eles [Kermani et al., 2013]. Nesta seção, estão condensadas algumas práticas consideradas "boas" quanto ao funcionamento de *software* embarcado. Estas características serviram como base para definir a forma da *GEmSysC*.

Segundo [Graves, 2011], a alocação dinâmica de memória é uma estratégia muito popular em desenvolvimento de *software*, mas que deve ser evitada em sistemas embarcados. A alocação dinâmica pode levar a vazamentos ou fragmentação de memória, que, segundo o autor, podem ser desastrosos. O autor ainda cita o padrão *DO-178B*, que define características de *software* crítico utilizado em aviônica e proíbe o uso de alocação dinâmica nestes sistemas.

Em [Graves, 2011], apresentam-se alocadores personalizados de memória como uma alternativa à alocação dinâmica de memória na *heap*. Alocadores de memória são estruturas e funções que funcionam de forma semelhante às chamadas convencionais `malloc` e `free`, mas levam em conta limitações e características da aplicação. Por exemplo, alocadores podem usar *pools* de memória ou a pilha. Em [Gradinaru, 2010] sugere-se o uso de *pools* de memória como alternativa à alocação dinâmica. Em [MacMillan, 2011], sugere-se evitar o uso de alocação dinâmica, e em seu lugar usar alocação de pilha ou, preferencialmente, alocação estática. A vantagem de se utilizar alocação estática é permitir que se saiba de quanta memória o *software* precisa em tempo de compilação [MacMillan, 2011].

A *GEmSysC* deve ser apenas uma camada de abstração sobre implementações existentes. Ela especificada de forma que, no nível da *API*, não é necessário utilizar alocação dinâmica. Para evitar alocações dinâmicas, tal qual sugerido em [Graves, 2011], em [Gradinaru, 2010] e em [MacMillan, 2011], poderiam ser utilizadas *pools* de memória, alocação estática e alocação na pilha. A alocação na pilha não foi utilizada pois limitaria o tempo de vida das estruturas ao escopo local (dados alocados na pilha são liberados assim que a função retorna). Entre *pools* de memória e alocação estática, escolheu-se utilizar alocação estática pois isto não depende de nenhum código adicional. A *GEmSysC* exige a declaração estática e a definição de todas as estruturas antes de serem utilizadas. Pelo mesmo motivo, todas as funções que geram blocos de dados como saída não devem alocar os *buffers* que armazenarão estes dados, mas sim receber ponteiros para *buffers*, que não são alocados pela *GEmSysC*.

Segundo [Barry and Hartnett, 2007], parâmetros grandes devem ser passados por referência, para evitar o custo computacional da cópia de dados. O autor também sugere que se utilizem ponteiros diretos em lugar de índices de *array*.

Para evitar cópia de parâmetros e valores de retorno, tal qual sugerido em [Barry and Hartnett, 2007], foram adotas práticas de passar parâmetros por referência na *GEmSysC*. Os identificadores de algoritmo criptográfico passados às funções, tais quais apresentados na seção 4.4.1, são na verdade referências às estruturas estaticamente alocadas que representam cada instância de algoritmo criptográfico. Os *buffers* de dados (com dados de entrada e de saída, chaves, etc) das funções da *API* são sempre passados por ponteiro. Finalmente, com relação aos dados, nunca passam-se *buffers* e índices, mas sim o ponteiro direto para a posição de memória de cada *buffer* em que os dados devem ser escritos, tal qual sugerido em [Barry and Hartnett, 2007].

4.3.2 Convenções

Nesta seção, estão apresentadas as convenções estabelecidas quanto a: definição e declaração de estruturas de dados e definição, declaração e utilização de funções, além da nomenclatura utilizada. Cada convenção está justificada com base nas “boas práticas” de elaboração de *API* apresentadas na seção 3.2.1 e com base em outras convenções e *APIs* existentes.

Da mesma forma que ocorre com a *API* do *CMSIS-RTOS* [CMSIS, 2012], a *GEmSysC* é independente da implementação. A forma como suas estruturas e funções foram elaboradas

visou facilitar o processo de construir uma camada compatível com ela sobre implementações existentes. Na figura 4.2, apresenta-se a relação entre a camada de abstração com *API* elaborada, a biblioteca que implementa as funcionalidades criptográficas e uma aplicação de alto nível.

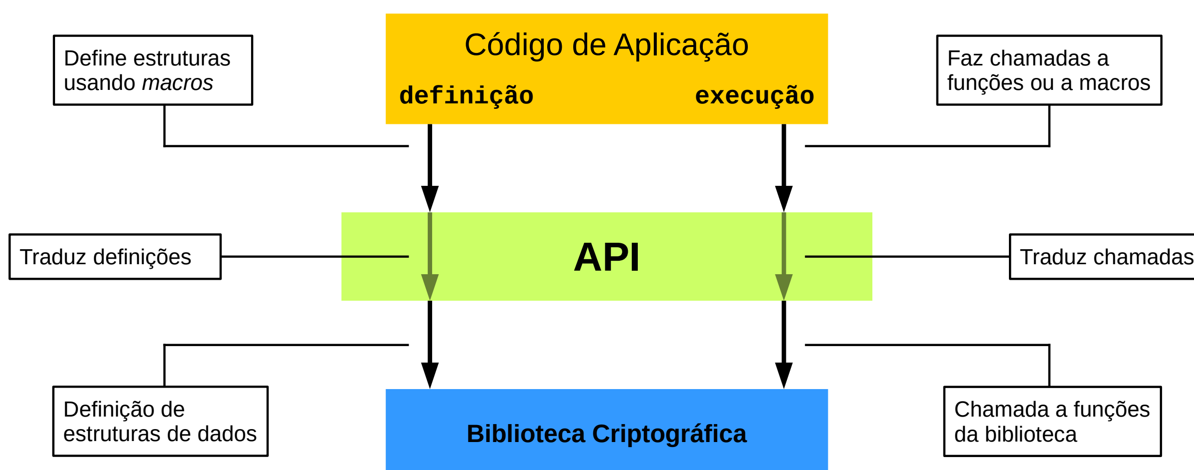

Figura 4.2: Relação entre uma camada de abstração utilizando a *API* desenvolvida, a aplicação de alto nível e a implementação da biblioteca criptográfica (autoria própria).

Em todas as *APIs* criptográficas estudadas, há uma forma de se definir estruturas de dados que armazenam o estado interno de operações criptográficas e funções que realizam as operações, tendo como entrada os dados a serem processados e estas estruturas. Cada *API* utiliza uma nomenclatura própria para se referir a diferentes elementos (funções, estruturas, módulos, entre outros) e conceitos (algoritmos, implementação, entre outros) que a compõem. Adotou-se neste documento uma nomenclatura única para referir-se aos elementos e conceitos relacionados à *GEmSysC*. Segue a nomenclatura adotada.

- Um "**algoritmo criptográfico**" é a especificação de como se realizam uma ou mais operações criptográficas de natureza semelhante. Exemplos de algoritmos criptográficos são *AES* e *RSA*;

- Operações criptográficas são agrupadas em "**tipos de operações**", de acordo com o tipo de processamento de dados realizado. Exemplos de tipos de operações são cifragem, decifragem, geração de resumo criptográfico;

- Um "**mecanismo criptográfico**" é a implementação do algoritmo criptográfico, e engloba o conjunto de funções e estruturas de uma biblioteca ou *API*, que combinadas realizam as operações criptográficas relacionadas ao algoritmo. Um mesmo mecanismo criptográfico pode realizar mais de um tipo de operação criptográfica. Por exemplo, é comum que o mesmo mecanismo realize cifragem e decifragem de dados.

- A ***API* do mecanismo criptográfico** é a especificação de como são suas estruturas e funções, enquanto o código que o compõe é sua **implementação**;

- O mecanismo criptográfico opera sobre estruturas de dados que armazenam o estado do algoritmo correspondente, que são coletivamente chamadas de "**instâncias de mecanismo criptográfico**".

- A camada *GEmSysC* é composta de duas partes: a **especificação** de como são suas estruturas e funções (a *API*), e sua **implementação** sobre a biblioteca criptográfica. Enquanto a especificação da *GEmSysC* é única, as implementações construídas sobre duas bibliotecas diferentes podem variar bastante. A especificação da *GEmSysC* relativa a cada algoritmo criptográfico também é denominada mecanismo criptográfico.

Um sistema operacional embarcado expõe para as aplicações um conjunto limitado de primitivas (semáforos, *mutexes*, filas de mensagens, *threads*, etc) [Tan and Tran Nguyen, 2009]. Por este motivo, espera-se que as *APIs* de padrões como o *CMSIS-RTOS* e de implementações como o *FreeRTOS* e o *ChibiOS* sofram poucas alterações com o tempo. O mesmo não se aplica a bibliotecas criptográficas, que podem não suportar certos algoritmos, e vir a suportá-los à medida que estes forem criados ou tornarem-se relevantes. Por este motivo, não é possível estabelecer uma especificação fechada e definitiva para a *API* da *GEmSysC*. Optou-se por adotar uma estrutura modular na especificação de mecanismos criptográficos suportados pela *GEmSysC*.

A estrutura da *GEmSysC* é composta de um *core* genérico, em que estão inclusas todas as funcionalidades e estruturas genéricas que independem do mecanismo criptográfico utilizado, e de módulos adicionais para cada mecanismo em particular. Os módulos referentes a cada algoritmo criptográfico são especificados e implementados separadamente e podem ser incluídos ou não de acordo com a necessidade da aplicação de alto nível e com a disponibilidade da biblioteca criptográfica. Em termos de organização de código, isto é feito da seguinte forma: o *core* genérico deve ser incluído no código de aplicação através de um arquivo de cabeçalho, enquanto cada módulo deve ser incluído a partir de um arquivo de cabeçalho individuais. Um exemplo desta estrutura está apresentado na figura 4.3.

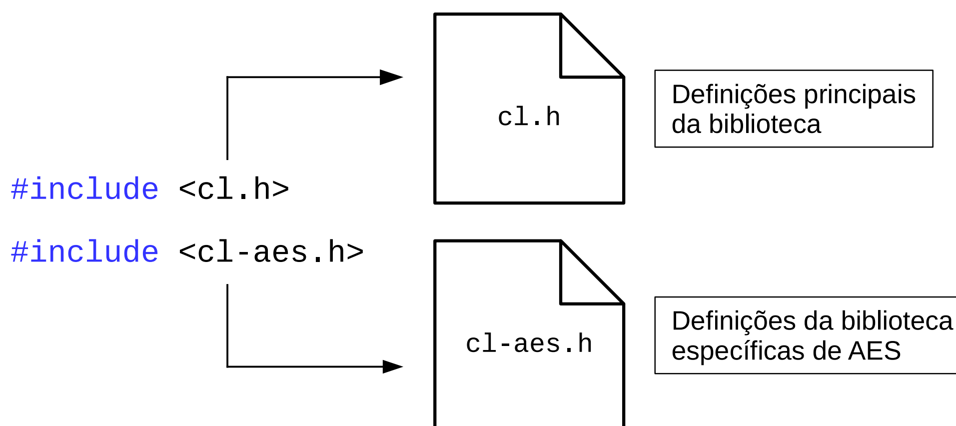

Figura 4.3: Exemplo de como incluir a biblioteca criptográfica e o módulo com suporte a *AES* em um arquivo de código (autoria própria).

Com relação à nomeação de elementos de código, decidiu-se adotar convenções semelhantes às adotadas no *CMSIS-RTOS* [CMSIS, 2012], mas trocando-se o prefixo `os` por `cl`, abreviação de *"cryphographic library"* ("biblioteca criptográfica" em inglês). Da mesma forma que no *CMSIS-RTOS*, as estruturas, funções e *macros* de uso global são nomeadas em *Camel case* com o prefixo `cl`, enquanto as internas são nomeadas em caracteres minúsculos separados por *underscores*, também com o prefixo `cl`. Esse prefixo é utilizado para evitar colisões de nomes entre estruturas, funções e *macros* da *GEmSysC* e aquelas declaradas ou definidas no código de usuário, já que a linguagem *C* não tem mecanismos nativos para separar espaços de nomes (*namespaces*).

4.3.3 Discussão

A *GEmSysC* foi concebida de forma a não necessitar, no nível da *API*, que seja utilizada alocação dinâmica. A biblioteca utilizada em cada implementação da camada de abstração pode ou não aderir à mesma prática. Se a biblioteca também evitar este tipo de gerenciamento memória, um código que utilize a *GEmSysC* sobre esta biblioteca não necessitará de alocação dinâmica, o que é recomendado no desenvolvimento de sistemas embarcados [Gradinaru, 2010, MacMillan, 2011, Graves, 2011].

Com relação aos parâmetros das funções, adotaram-se as práticas de passar estruturas grandes por referência e de utilizar *buffers* de memória alocados no código de usuário. Da mesma forma que ocorre com a metodologia de alocação de memória, a biblioteca utilizada em cada implementação pode ou não aderir a estas práticas. Esta estratégia só será realmente vantajosa se a biblioteca utilizada também evitar o uso de alocação dinâmica, pois, do contrário, a aplicação como um todo (incluindo a biblioteca) continuará dependendo de alocação dinâmica.

A *GEmSysC* foi concebida de forma a ser modular. Com modularidade, não é necessário alterar a especificação como um todo quando for interessante adicionar suporte a um novo algoritmo criptográfico. Em lugar disso, é necessário apenas especificar um novo módulo. Outra vantagem da estrutura modular é permitir que sejam feitas implementações parciais: a camada de abstração pode ser implementada sobre bibliotecas mais simples, que não têm suporte a todos os módulos previstos na *GEmSysC*.

4.4 Especificação

Foi feita uma especificação da *GEmSysC* e dos módulos de *AES*, *RSA* e *SHA-256*, o que está apresentado no apêndice A. Estes algoritmos são exemplares, respectivamente, de criptografia simétrica, criptografia assimétrica e resumo criptográfico. O *AES* foi escolhido pois é um padrão de criptografia simétrica indicado para uso pelas agências e departamentos dos Estados Unidos da América para proteger informações sensíveis [NIST, 2001]. O *RSA*, que é um algoritmo de criptografia assimétrica, foi escolhido pois é o algoritmo recomendado para trocas de chaves entre sistemas pelo *NIST* [Barker et al., 2014]. O *SHA*-256 foi escolhido pois é um padrão elaborado pelo *NIST*. Os algoritmos da família *SHA*-3 são sucessores da família *SHA*-2, mas não foram escolhidos para a especificação inicial da *GEmSysC* pois eles não estão presentes em bibliotecas amplamente utilizadas como a *OpenSSL* [OpenSSL, 2015].

Na figura 4.4, estão representados graficamente os elementos da *GEmSysC*. Nesta figura, cada elemento está representada como uma peça de quebra-cabeça, em que cada "encaixe fêmea" representa uma *API* e cada "encaixe macho" representa o acesso a uma a *API*. No diagrama, está evidenciada uma *API* interna do core da *GEmSysC* que é utilizada pelos demais módulos. Esta *API* interna não faz parte da especificação da *GEmSysC*, o que é intencional, pois permite que a implementação seja elaborada da forma mais eficiente possível de acordo com a biblioteca criptográfica utilizada. A *API* pública da *GEmSysC* está representada no diagrama pelo encaixe entre a aplicação e *GEmSysC*. Este diagrama foi feito com foco na relação entre os módulos da *GEmSysC* e não representa adequadamente a forma que a aplicação inclui cada um dos módulos; a aplicação só foi inserida neste diagrama para evidenciar sua relação com a *GEmSysC*.

Para facilitar a inclusão da *GEmSysC* em projetos e sua implementação sobre bibliotecas criptográficas existentes, elaborou-se uma convenção sobre como criar e utilizar os arquivos

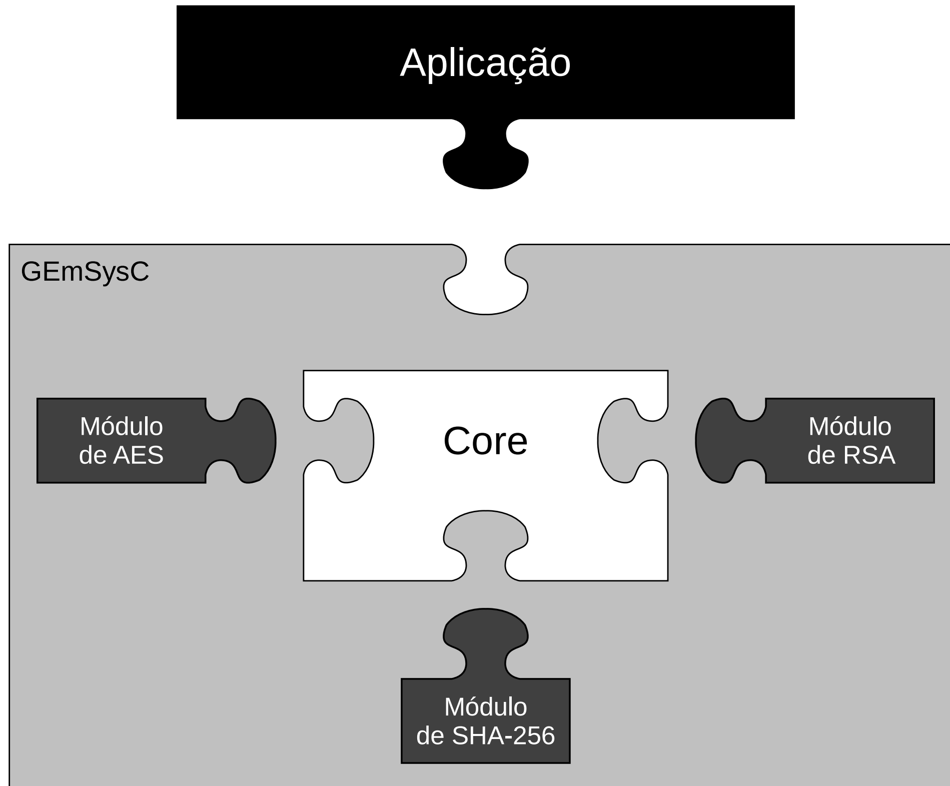

Figura 4.4: Diagrama de módulos da *GEmSysC* (autoria própria).

de cabeçalho. A convenção utilizada na *GEmSysC* é inspirada na utilizada com o *CMSIS-RTOS*. Decidiu-se tomar o *CMSIS-RTOS* como referência pois este é um padrão que permite que diversas implementações diferentes sejam realizadas preservando uma *API* consistente.

Quando se utiliza o *CMSIS-RTOS*, é necessário incluir o arquivo *"cmsis_os.h"*. Segundo a especificação, este arquivo tem definições específicas de acordo com a implementação, mas devem necessariamente conter um conjunto mínimo de estruturas básicas [CMSIS, 2012]. O problema deste procedimento é que cada implementador do *CMSIS-RTOS* deve necessariamente fazer a sua própria versão do *"cmsis_os.h"*. Este processo é passível de erros e torna mais demorada a atividade de atualizar a implementação quando a especificação do *CMSIS-RTOS* é atualizada.

Do ponto de vista das aplicações, há uma diferença marcante entre utilizar a *GEmSysC* e o *CMSIS-RTOS*: na *GEmSysC*, em lugar de incluir um único arquivo de cabeçalho, é necessário incluir o arquivo de cabeçalho que representa o *core* da *API* e os arquivos de cabeçalho com cada um dos módulos utilizados. Esta diferença se deve à modularidade da *GEmSysC*, como discutido no começo deste capítulo.

Do ponto de vista dos implementadores da *API*, foi feita uma simplificação em relação ao que se faz no *CMSIS-RTOS* [CMSIS, 2012]. Em lugar de os implementadores terem de criar sua própria versão dos arquivos de cabeçalho referentes ao core e a cada um dos módulos, com todas as declarações e definições referentes à *API*, decidiu-se dividir cada arquivo de cabeçalho em duas partes: uma parte genérica (independente da implementação) e uma parte específica (dependente da implementação). O arquivo de cabeçalho genérico é fornecido na especificação da *GEmSysC*, enquanto o arquivo de cabeçalho específico é fornecido pela implementação.

O arquivo de cabeçalho específico é incluído automaticamente dentro do arquivo de cabeçalho genérico (através de uma diretiva `#include`), e o código de aplicação inclui somente o arquivo de cabeçalho genérico. Por exemplo, o arquivo de cabeçalho do módulo de *AES* é o *"cl-aes.h"*, que é parte da especificação da *GEmSysC* e não muda de uma implementação para a outra. Este arquivo inclui necessariamente o arquivo *"cl-aes-internal.h"*, que não faz parte da especificação e é definido de forma diferente em cada uma das implementações. Neste exemplo, o código de aplicação inclui somente o *"cl-aes.h"*. Na tabela 4.1, estão apresentados os arquivos básicos que cada implementação da *GEmSysC* deve conter.

Tabela 4.1: Lista de arquivos básicos presentes nas implementações da *GEmSysC*.

	Módulo	Definido em:	Conteúdo
cl.h	*Core*	Especificação	Definições do *core*.
cl-internal.h	*Core*	Implementação	Definições do *core* específicas da implementação.
cl.c	*Core*	Implementação	Implementação do *core*.
cl-aes.h	*AES*	Especificação	Definições do módulo de *AES*.
cl-aes-internal.h	*AES*	Implementação	Definições do módulo de *AES* específicas da implementação.
cl-aes.c	*AES*	Implementação	Implementação do módulo de *AES*.
cl-rsa.h	*RSA*	Especificação	Definições do módulo de *RSA*.
cl-rsa-internal.h	*RSA*	Implementação	Definições do módulo de *RSA* específicas da implementação.
cl-rsa.c	*RSA*	Implementação	Implementação do módulo de *RSA*.
cl-sha256.h	*SHA-256*	Especificação	Definições do módulo de *SHA256*.
cl-sha256-internal.h	*SHA-256*	Implementação	Definições do módulo de *SHA256* específicas da implementação.
cl-sha256.c	*SHA-256*	Implementação	Implementação do módulo de *SHA256*.

Os arquivos de cabeçalho foram documentados com a ferramenta *Doxygen* [Doxygen, 2015]. Segundo [Doxygen, 2015], *Doxygen* é o padrão de fato de documentação em *C++*. Com *Doxygen*, é possível fazer a documentação como parte do código e exportá-la para diversos formatos, como *HTML* e *PDF*. No apêndice A, está apresentada a documentação tal qual exportada em formato *PDF*.

Na subseção seguinte, estão justificadas as decisões tomadas na especificação da *API*, e nas subseções subsequentes, as decisões tomadas na especificação dos módulos de *AES*, *RSA* e *SHA-256*. Para cada um dos módulos, apresenta-se uma breve introdução do algoritmo criptográfico correspondente, analisa-se a forma que é representado na *Cryptoki*, na *OpenSSL* e na *wolfSSL*, então discutem-se quais são os parâmetros de entrada do algoritmo e apresenta-se a forma do mecanismo criptográfico da *GEmSysC*.

A biblioteca *OpenSSL* foi escolhida para esta comparação pois é de fato um padrão em sistemas computacionais. Isto é evidenciado por uma pesquisa feita em 2014, em que demonstrou-se que, dentre 100 milhões de *sites* avaliados, mais de dois terços utilizam *OpenSSL* como motor criptográfico [Netcraft, 2014]. A biblioteca *wolfSSL* foi escolhida por ser de código aberto e voltada a sistemas embarcados [wolfSSL, 2016]. A *API Cryptoki* foi escolhida pois é um padrão aberto e disponível publicamente [Griffin and Fenwick, 2015a].

Nesta seção, utilizou-se somente a nomenclatura da *GEmSysC*. Como esta nomenclatura não é exatamente igual à adotada na especificação de cada uma das *APIs* analisadas, é possível que pareça haver discrepâncias entre este documento e a especificação da *API*. Quando cabível, os conceitos tais quais definidos na especificação da *API* analisada serão comparados aos da *GEmSysC*. Cada um dos módulos da *GEmSysC* declara seus próprios tipos de dados, funções, constantes e *macros*. A especificação completa do *core* e dos módulos, tal qual exportada pela ferramenta *Doxygen*, pode ser encontrada no apêndice A.

4.4.1 *Core* da biblioteca

A forma de abstrair os acesso aos algoritmos criptográficas difere entre a *Cryptoki*, a *OpenSSL* e *wolfSSL*. Nesta seção, estão discutidos aspectos gerais das *API Cryptoki* e das *APIs* das bibliotecas *OpenSSL* e *wolfSSL*, e, em seguida, estão justificadas as escolhas feitas no projeto da *GEmSysC*.

Na *Cryptoki*, o acesso aos mecanismos criptográficos é feito através de *seções*, que são estruturas que representam a conexão entre a aplicação de alto nível e algum dispositivo externo ou elemento de *software* que realiza efetivamente as operações criptográficas. Cada sessão pode ser inicializada com algum mecanismo criptográfico, para então poder realizar processamento de dados. De acordo com esta definição, pode-se considerar que uma sessão torna-se uma "instância de mecanismo criptográfico" enquanto estiver inicializada para utilizar algum mecanismo criptográfico. O tipo de dados que representa uma sessão é opaco, o que significa que seu conteúdo é de uso interno da biblioteca e não deve ser manipulado diretamente a partir do código de alto nível. As operações relacionadas à sessão são genéricas em relação ao mecanismo criptográfico, mas variam de acordo com o tipo de operação criptográfica [Griffin and Fenwick, 2015a].

Na *OpenSSL*, o acesso a funcionalidades criptográficas pode ser feito através de estruturas de dados e funções especializadas ou através de uma camada de abstração denominada *EVP* [OpenSSL, 2013]. Para utilizar a camada *EVP*, cria-se um "contexto *EVP*", que então é inicializado para utilização com algum mecanismo criptográfico. A partir daí, as operações criptográficas são feitas através do contexto *EVP*. Por este conceito, considera-se que o contexto *EVP* torna-se uma "instância de mecanismo criptográfico" quando é inicializado. As operações sobre o contexto *EVP* são genéricas em relação ao mecanismo criptográfico, mas variam de acordo com o tipo de operação criptográfica. Nem todos os mecanismos suportados pela *OpenSSL* estão disponíveis através da camada *EVP*, como por exemplo o algoritmo *RSA*. Neste caso, o acesso ao mecanismo criptográfico se dá através de funções e estruturas específicas. Assim, apesar de haver a abstração *EVP* para alguns mecanismos, na *API* como um todo há mais de um tipo de dado que representa instâncias de mecanismos criptográficos. Em geral, estes tipos de dados devem ser tratados como se fossem opacos [OpenSSL, 2015].

Na *wolfSSL*, cada mecanismo criptográfico é acessado por funções diferentes e utiliza estruturas de dados próprias. Por exemplo, as estruturas de dados e funções utilizadas para realizar criptografia em *AES* são diferentes das utilizadas com *RSA*. Assim, na *wolfSSL*, há vários tipos que representam instâncias de algoritmos criptográficos. As funções utilizadas para processamento de dados variam de acordo com o mecanismo criptográfico, com o tipo de operação criptográfica, e às vezes até com outros parâmetros. Por exemplo, variações do *AES* com diferentes modos de operação são realizados por funções diferentes. Em geral, os tipos de

dados que representam as instâncias de mecanismos criptográficos devem ser tratados como se fossem opacos [wolfSSL, 2016].

Nas três *APIs*, sempre é necessário definir estruturas que representam a instância de algoritmo criptográfico, as quais devem ser inicializadas antes do uso. Após inicializadas, as instâncias de mecanismo podem ser utilizadas para processar dados e para gerar novos dados como resultado. Em alguns casos, após o processamento de dados, é necessário liberar recursos computacionais tomados durante a inicialização.

Na *Criyptoki*, usa-se um único tipo de dados que representa genericamente a instância de mecanismo criptográfico. Na parte da *OpenSSL* que utiliza a camada *EVP*, há dois tipos de dados que representam a instância, um para mecanismos que fazem cifragem ou decifragem e outro para mecanismos que fazem resumo criptográfico. Tanto na *Cryptoki* quanto na *OpenSSL*, as funções utilizadas para inicializar as instâncias dos mecanismos e para processar dados variam apenas de acordo com o tipo de operação criptográfica. Em ambos os casos, o mecanismo é determinado por um parâmetro da função de inicialização. Na *wolfSSL* e na parte da *OpenSSL* que não suporta a camada *EVP*, há diversos tipos de dados que representam instâncias de algoritmos criptográficos, e as funções utilizadas para inicializar estas instâncias e para processar dados variam de acordo com diversos parâmetros, como o mecanismo criptográfico, o tipo de operação criptográfica e outros aspectos de funcionamento. Na tabela 4.2, está apresentada uma comparação entre as três *APIs* consideradas.

Tabela 4.2: Comparação das *APIs* com relação à forma de definir e utilizar instâncias de mecanismos criptográficos.

	Cryptoki	*OpenSSL*	*wolfSSL*
Tipos de dados variam por:			
Mecanismo	Não varia	Pode ser um contexto *EVP* (`EVP_CIPHER_CTX` e `EVP_MD_CTX`) ou outro tipo especializado.	Um tipo diferente por mecanismo.
Tipo de operação	Não varia	O contexto *EVP* é diferente para resumo e para cifragem/decifragem. Tipos especializados são diferentes naturalmente.	Não varia.
Função para inicializar varia por			
Mecanismo	Não varia	No caso de *EVP*, não varia, mas em outros casos, há funções diferentes por mecanismo.	Funções diferentes de acordo com o mecanismo.
Tipo de operação	Uma função diferente para cada tipo de operação.	Geralmente variam com o tipo de operação.	Funções diferentes de acordo com o tipo de operação.
Parâmetros do mecanismo	Não varia	No caso de *EVP*, não varia, mas em outros casos, pode haver variação.	Dependendo do mecanismo, as funções podem variar.

Na *GEmSysC*, as estruturas de dados que representam cada instância de mecanismo criptográfico são definidas através de *macros*. Existe uma única *macro* que define estruturas de dados de cada mecanismo em particular. Em cada implementação, estas *macros* são traduzidas pelo compilador em definições de estruturas utilizadas pela biblioteca subjacente. A definição destas *macros* depende da implementação, então o código de alto nível não pode depender de seus aspectos internos. Por este motivo, os mecanismos criptográficos são considerados tipos

opacos na *GEmSysC*. Esse modo de definir estruturas foi inspirado na utilizado na *API* do *CMSIS-RTOS*, em que se usa a mesma estratégia.

A declaração utilizando *macros* diferentes de acordo com o mecanismo foi escolhida em detrimento de utilizar uma forma única, que não variasse em função do mecanismo, como se faz na *OpenSSL* (com *EVP*) e na *Cryptoki*, pois tem vantagem de permitir que se saiba *a priori* o tamanho das estruturas de dados. Esta liberdade na implementação das estruturas permite que se leve em conta o tipo de alocação de memória utilizado pela biblioteca subjacente (estático, ou seja, em tempo de compilação e dinâmico, ou seja, em tempo de execução). Cada instância é definida com uma *macro* que então é inicializada por uma função específica que retorna um identificador. No caso de alocação dinâmica, a *macro* que define a instância é traduzida em uma definição de dados de somente leitura alocados em tempo de compilação, enquanto o identificador é o ponteiro para uma estrutura alocada dinamicamente inicializada com base nos dados constantes. No caso de alocação estática, a *macro* que define a instância é traduzida como duas definições: dados de somente leitura e dados voláteis que serão alterados durante a execução, enquanto o identificador é o ponteiro para os dados voláteis.

As funções de inicialização diferem em função do mecanismo e do tipo de operação, e retornam um "identificador". O "identificador" é de um tipo opaco, que não varia em função de nenhum parâmetro. Cada identificador é um *handle* para a instância de um mecanismo criptográfico. Todas as demais operações relacionadas ao mecanismo devem ser realizadas em função deste identificador. Escolheu-se usar funções diferentes para inicialização de acordo com o mecanismo e com o tipo de operação pois esta variação facilita a entrada de parâmetros, que podem variar em forma, tamanho e número de uma função de inicialização para a outra.

Além das *macros* que definem mecanismos criptográficos, que variam de acordo com o algoritmo criptográfico, e do identificador, que é um tipo único, há outro elemento que integra a parte genérica da *GEmSysC*: a declaração de motor criptográfico. Para usar um motor criptográfico definido em outro arquivo, utiliza-se outra *macro*, diferente da utilizada para definir a estrutura. Esta *macro* é traduzida pelo compilador na declaração das estruturas usando na palavra-chave `extern` da linguagem *C*.

A entrada e a saída de dados do mecanismo criptográfico são feitas através de uma única função, a função de processamento de dados. Esta função independe do mecanismo criptográfico e do tipo de operação utilizados, e recebe como entrada: o identificador do algoritmo criptográfico, um *buffer* de entrada e seu tamanho, um *buffer* de saída e seu tamanho e um parâmetro genérico adicional. No caso de funções de criptografia em blocos, esta função geralmente espera receber blocos inteiros de dados como entrada e gerar blocos correspondentes como saída. No caso de resumo criptográfico, a entrada geralmente pode ser em partes: a função pode ser invocada repetidamente recebendo dados como entrada, mas sem *buffer* de saída, e então, para gerar o resultado, é utilizada sem dados de entrada e com *buffer* de saída. A função que processa (cifra, decifra ou resume) dados na *GEmSysC* é a `clEngineProcess`.

A função que faz a liberação dos recursos utilizados pelos mecanismos criptográficos (como memória, periféricos, etc) é também única, e opera somente com o identificador do algoritmo criptográfico. Assim que o algoritmo tiver seus recursos liberados, o identificador não deve ser utilizado novamente, pois tornou-se inválido. A função que libera recursos na *GEmSysC* é a `clEngineFinalize`.

Existem certas operações que só se aplicam a certos algoritmos criptográficos. Neste caso, são declaradas e definidas funções especializadas como parte dos módulos referentes a estes algoritmos. Estas funções devem receber como parâmetro o identificador do algoritmo

criptográfico, como ocorre com as funções genéricas. Um exemplo deste tipo de função é a que verifica o tamanho máximo do bloco de dados de entrada em *RSA*, `clRSAMaxInputLength`.

Apesar de ser especificada em *C*, que não é uma linguagem de programação orientada a objetos, a camada *GEmSysC* conceitualmente pode ser vista como orientada a objetos. É possível considerar que cada mecanismo criptográfico suportado representa uma classe concreta que implementa a classe abstrata “mecanismo criptográfico”. Como as funções de processamento de dados e liberação de recursos são independentes do mecanismo, é possível entendê-las como métodos polimórficos definidos para os objetos desta classe abstrata. Nesta mesma linha de raciocínio, é possível compreender cada instância de mecanismo criptográfico como um objeto da classe concreta que implementa o mecanismo criptográfico.

Os valores de retorno das funções da *GEmSysC* seguem uma convenção: quando a função é executada com sucesso, o valor de retorno é não-negativo, e quando a função falha por algum motivo, o valor de retorno é negativo. O valor de retorno negativo é um código de erro cujos valores são geralmente compatíveis com os especificados no padrão *POSIX* [IEEE, 2013]. Nas funções de processamento de dados, o valor de retorno positivo indica o número de *bytes* escritos no vetor de destino.

A escolha de códigos de erro *POSIX* se deveu à ampla utilização deste padrão, que é a base de sistemas tipo *UNIX* [IEEE, 2013], de forma que se espera que desenvolvedores tenham certa familiaridade com estes códigos de erro. Os códigos de erro especificados no padrão *POSIX* são definidos no arquivo `errno.h` da biblioteca padrão. Nem todas as plataformas suportam o padrão *POSIX*, então, de acordo com a plataforma, o arquivo `errno.h` pode conter apenas os códigos de erro requeridos pela especificação da linguagem *C* [IEEE, 2013]. Para garantir a independência da plataforma, os códigos de erro utilizados na *GEmSysC* foram definidos em arquivos de cabeçalho próprios, acrescidos do prefixo `CL_`, segundo as convenções de nomeação adotadas. Por exemplo, o código de erro que indica sucesso é `CL_ENOERR`.

Não foi especificada a maneira que as estruturas internas das implementações da *GEmSysC* se relacionam, isto é, a *API* interna de cada implementação. A forma destas interfaces depende grandemente das características da biblioteca utilizada, e pode ser elaborada e finamente ajustada de forma a tornar a camada *GEmSysC* menor e mais eficiente.

Adotaram-se certas convenções para nomenclatura de *macros*, além de utilizar *Pascal case* e o prefixo `cl`, tal qual explicado na seção anterior. A definição de uma instância de mecanismo criptográfico (isto é, o que se insere no arquivo de código) é sempre feita por uma *macro* cujo nome começa com `cl`, seguido do nome do algoritmo, seguido do sufixo `Def`. A declaração da instância do motor (isto é, o que se insere no arquivo de cabeçalho) segue a mesma forma, mas com sufixo `Decl`. Para referenciar a instância do motor criptográfico no código, utiliza-se uma *macro* cujo nome tem o prefixo `cl` seguido somente do nome do algoritmo. Os identificadores de instâncias motores criptográficos são do tipo `clEngineInstanceId`, e os nomes de todas as funções que operam sobre motores criptográficos começam com o prefixo `clEngine`, seguido do nome da operação. Todas as funções específicas de algum mecanismo têm nomes começando com prefixo `cl`, seguido do nome do algoritmo, seguido do nome da operação.

As funções definidas para o core da *GEmSysC* estão apresentadas na tabela 4.3. Detalhes sobre seu funcionamento podem ser encontrados no apêndice A.

Tabela 4.3: Lista de funções e *macros* que parecem funções do *core* da *GEmSysC*.

`clEngineProcess`	Processa dados através de uma instância de mecanismo criptográfico.
`clEngineFinalize`	Libera recursos utilizados por uma instância de mecanismo criptográfico.

4.4.2 Módulo de *AES*

O *AES*[1] é um algoritmo de criptografia simétrica em blocos. O *AES* é um padrão aberto, definido no *FIPS* 197 publicado pelo *NIST*. O *NIST* indica que agências e departamentos do governo dos Estados Unidos da América utilizem este algoritmo para proteger informações sensíveis [NIST, 2001].

Em janeiro de 1997, o *NIST* realizou um processo seletivo em que diversos algoritmos criptográficos foram submetidos para apreciação. O objetivo deste processo era criar um algoritmo criptográfico aberto, que seria denominado *AES* [NIST, 1997]. Um dos algoritmos apresentados neste processo foi o *Rijndael*, que foi elaborado com três objetivos: resistir a todos os tipos de ataques conhecidos até então, ter um código pequeno sem perder a eficiência e ter *design* simples. O tamanho do bloco e o tamanho da chave usados com o *Rijndael* podem ser de 128, 192 ou 256 *bits* e não precisam ser iguais [Daemen and Rijmen, 1999]. Em 2000, o *NIST* anunciou que o *Rijndael* seria adotado como *AES* sem modificações em relação ao funcionamento, mas com uma limitação: o tamanho de bloco foi fixado em 128 *bits*. Portanto, o *AES* é o *Rijndael* com blocos de 128 *bits* [Daemen and Rijmen, 2002].

O *AES* é um algoritmo de criptografia em blocos, o que significa que ele processa blocos de tamanho fixo. Para processar mensagens maiores que o bloco, o algoritmo pode ser aplicado de diversas formas, denominadas "modos de operação". O modo de operação mais simples existente envolve dividir a mensagem em trechos do tamanho do bloco e aplicar o algoritmo criptográfico em cada um dos trechos individualmente, sem nenhum pré-processamento dos dados. Este modo de operação é denominado *ECB*[2]. A desvantagem do *ECB* é que a cifragem de dois blocos com o mesmo conteúdo gerará sempre a mesma saída. Outra opção é o *CBC*, em que cada bloco de dados é "aleatorizado" antes da cifragem, o que se faz através de sua combinação por operador ou-exclusivo (*XOR*) com o resultado da cifragem do bloco anterior [Daemen and Rijmen, 2002]. Em bibliotecas criptográficas, denomina-se o valor com o qual se faz a operação de ou-exclusivo de "vetor de inicialização"[3] [OpenSSL, 2013, wolfSSL, 2016]. Há diversos outros modos de operação disponíveis. Quando o tamanho da mensagem não é múltiplo do tamanho do bloco, é possível adicionar um preenchimento[4] à mensagem. Com preenchimento, o resultado da operação de cifragem fica maior que a os dados originais [Daemen and Rijmen, 2002].

Na *Cryptoki*, para instanciar um mecanismo criptográfico, é necessário estabelecer uma sessão representada por estruturas do tipo `CK_SESSION_HANDLE`. A inicialização da sessão deve ser feita com um dispositivo que tenha suporte a *AES*. Uma vez estabelecida a sessão, ela deve ser preparada para processamento de dados, o que se faz através da função `C_EncryptInit` no caso de cifragem ou da função `C_DecryptInit`

[1] *"Advanced Encryption Standard"*, que significa "Padrão Criptográfico Avançado".
[2] Sigla de *"Eletronic Code Book"*.
[3] *"Initialization vector"* ou *IV* em inglês.
[4] *"Padding"* em inglês.

Código 4.1: Cifragem simétrica de dados com *AES* usando a *Cryptoki*.

```
#define BLOCK_LENGTH ( 16 ) /* 128 bits */

CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;

/* omitido trecho em que hSession e hKey são inicializadas */

CK_MECHANISM mechanism = {CKM_AES_CBC, (CK_BYTE*) iv, BLOCK_LENGTH};

rc = C_EncryptInit(hSession, &mechanism, hKey);
if (rc != CKR_OK) {
    printf("Erro inicializando cifraegm: 0x%X\n", rc);
    return rc;
}

CK_ULONG encrypt_len = output_len;
rc = C_Encrypt(hSession, (CK_BYTE_PTR) input, (CK_ULONG) input_len, (
   CK_BYTE_PTR) output, &encrypt_len);
if (rc != CKR_OK) {
    printf("Erro durante cifragem: %x\n", rc);
    return rc;
}

return CKR_OK;
```

no caso de decifragem. O processamento em si é feito através de chamadas às funções `C_EncryptUpdate` ou `C_DecryptUpdate` e terminado através de chamadas às funções `C_EncryptFinal` e `C_DecryptFinal` [Griffin and Fenwick, 2015a]. Um dos parâmetros das funções `C_EncryptInit` e `C_DecryptInit` é uma estrutura do tipo `CK_MECHANISM`, na qual se define o modo de operação, através do campo `mechanism`. Exemplos de valores para este campo são `CKM_AES_ECB`, `CKM_AES_CBC` e `CKM_AES_OFB`. O tamanho da chave não é definido diretamente na sessão, mas sim através do objeto que representa a chave [Griffin and Fenwick, 2015b]. Um exemplo de como é feita cifragem de dados com *AES* em modo *CBC* usando a *Cryptoki* está apresentado no código 4.1. Neste código, `input` e `input_len` são respectivamente o *buffer* de entrada e seu tamanho, `output` e `output_len` são respectivamente o *buffer* de saída e seu tamanho, `iv` é o *buffer* com o vetor de inicialização e `encrypt_len` é o número de bytes realmente escritos no *buffer* de saída.

Na biblioteca *OpenSSL*, o *AES* é usado através de uma abstração denominada *EVP*. O *EVP* é uma representação genérica de uma instância de um mecanismo criptográfico [OpenSSL, 2013]. Para inicializar a instância do *AES*, é fornecido o valor da chave e do vetor de inicialização, através das funções `EVP_EncryptInit_ex` e `EVP_DecryptInit_ex`. É possível definir o tipo de preenchimento utilizado através da função `EVP_CIPHER_CTX_set_padding`. O processamento de dados é feito de forma contínua, sem necessariamente respeitar o tamanho dos blocos, através das funções `EVP_EncryptUpdate` e `EVP_DecryptUpdate`. O processamento é terminado pelas funções `EVP_EncryptFinal_ex` e `EVP_DecryptFinal_ex`. A *OpenSSL* suporta diversos modos de operação do *AES*, como *ECB*, *CBC*, *CFB*, *OFB*, *GCM* e *CCM*. A definição do tamanho da chave e do modo de operação é feita através de estruturas do tipo *EVP_CIPHER*, que

Código 4.2: Cifragem simétrica de dados com *AES* usando a *OpenSSL*.

```
EVP_CIPHER_CTX * ctx = EVP_CIPHER_CTX_new();
if (ctx == NULL) {
    printf("Erro criando contexto\n");
    return -1;
}

ret = EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv)
if (!ret) {
    printf("Erro inicializando cifragem: 0x%X\n", ret);
    return -1;
}
ret = EVP_CIPHER_CTX_set_padding(ctx, 0);
if (!ret) {
    printf("Erro definindo preenchimento: 0x%X\n", ret);
    return -1;
}

int encrypt_length = output_len;
ret = EVP_EncryptUpdate(ctx, output, &encrypt_length, (const unsigned char
    *) input, input_len);
if (!ret) {
    printf("Erro durante cifragem: 0x%X\n", ret);
    return -1;
}

EVP_CIPHER_CTX_free(ctx);

return encrypt_length;
```

são obtidas através de funções especializadas presentes na biblioteca [OpenSSL, 2015]. Um exemplo de como é feita cifragem de dados com *AES* em modo *CBC* usando a *OpenSSL* está apresentado no código 4.2. Neste código, `input` e `input_len` são respectivamente o *buffer* de entrada e seu tamanho, `output` e `output_len` são respectivamente o *buffer* de saída e seu tamanho, `iv` é o *buffer* com o vetor de inicialização, `key` é o *buffer* com a chave de criptografia e `encrypt_len` é o número de bytes realmente escritos no *buffer* de saída.

Na biblioteca *wolfSSL*, o *AES* é utilizado através de funções que manipulam uma estrutura do tipo `Aes`. Esta estrutura deve ser definida antes do uso e inicializada através da função `wc_AesSetKey` para modo *CBC*, `wc_AesSetKeyDirect` para modos *CTR* e *ECB*, `wc_AesGcmSetKey` para modo *GCM* e `wc_AesCcmSetKey` para modo *CCM*. A estrutura `Aes` pode ser inicializada para cifragem ou decifragem de dados e com qualquer tamanho de chave suportado, o que é definido através de um parâmetro da função de inicialização. A cifragem de dados é feita através das funções `wc_AesCbcEncrypt`, `wc_AesEncryptDirect`, `wc_AesGcmEncrypt` e `wc_AesCcmEncrypt`, de acordo com o modo de operação utilizado. A decifragem dos dados é feita através das funções `wc_AesCbcDecrypt`, `wc_AesDecryptDirect`, `wc_AesGcmDecrypt` e `wc_AesCcmDecrypt`, de acordo com o modo de operação utilizado. Dependendo da função utilizada, a entrada pode ser um único bloco de dados ou um conjunto de blocos de dados, e a saída é composta do bloco ou dos blocos de dados processados correspondentes [wolfSSL, 2016]. Um exemplo de como é feita

Código 4.3: Cifragem simétrica de dados com *AES* usando a *wolfSSL*.

```
#define KEY_LENGTH ( 32 ) /* 256 bits */

Aes aes;

ret = wc_AesSetKey(&aes, (const byte*) key, KEY_LENGTH, (const byte*) iv,
    AES_ENCRYPTION);
if (ret != 0) {
    printf("Erro inicializando cifragem: 0x%X\n", ret);
    return -1;
}

ret = wc_AesCbcEncrypt(&aes, (byte*) output, (const byte*) input, in_length
    );
if (ret != 0) {
    printf("Erro durante cifragem: 0x%X\n", ret);
    return -1;
}

return 0;
```

cifragem de dados com *AES* em modo *CBC* usando a *wolfSSL* está apresentado no código 4.3. Neste código, `input` e `input_len` são respectivamente o *buffer* de entrada e seu tamanho, `output` é o *buffer* de saída e seu tamanho, `iv` é o *buffer* com o vetor de inicialização e `key` é o *buffer* com a chave de criptografia.

Pelas características do *AES*, observa-se que há pelo menos cinco parâmetros de configuração deste algoritmo:

- Tamanho da chave (que pode ser 128, 192 ou 256 *bits*);
- Valor da chave;
- Vetor de inicialização;
- Modo de operação;
- Direção da operação criptográfica (cifragem ou decifragem).

Entre a *Cryptoki*, a *OpenSSL* e a *wolfSSL*, a forma de determinar estes parâmetros varia imensamente. O tamanho da chave é determinado pelo tamanho da chave fornecida na *Cryptoki* e na *wolfSSL*, mas é determinado junto à definição do motor criptográfico utilizado na *OpenSSL* (há um motor diferente para cada tamanho de chave). O vetor de inicialização pode ser fornecido na inicialização em todas as *APIs*. O modo de operação é determinado na *Cryptoki* e na *OpenSSL* através do mecanismo criptográfico indicado durante a inicialização (há um motor criptográfico para cada modo de operação), enquanto na *wolfSSL* é determinado pelas funções utilizadas (há funções diferentes para cada modo de operação). A direção do algoritmo é determinada na *wolfSSL* como parâmetro das funções de inicialização e também pelas funções utilizadas, enquanto na *OpenSSL* e na *wolfCrypt*, ela é determinada unicamente pelas funções utilizadas. Estas características estão resumidas na tabela 4.4.

Tabela 4.4: Como são definidos os parâmetros do *AES* na *Cryptoki*, na *OpenSSL* e na *wolfSSL*.

Parâmetro	*Cryptoki*	*OpenSSL*	*wolfSSL*
Tamanho da chave	Parâmetro da função de inicialização	Parâmetro da função de inicialização (implicitamente pelo tipo do motor criptográfico)	Parâmetro da função de inicialização
Valor da chave	Objeto da chave	Parâmetro da função de inicialização	Parâmetro da função de inicialização
Vetor de inicialização	Objeto da chave	Parâmetro da função de inicialização	Parâmetro da função de inicialização
Modo de operação	Parâmetro da função de inicialização	Parâmetro da função de inicialização	Funções utilizadas para inicialização e processamento de dados
Direção	Parâmetro da função de inicialização e função utilizada para processamento de dados	Função utilizada para processamento de dados	Função utilizada para processamento de dados

A forma do módulo que contém o mecanismo de *AES* da *GEmSysC* foi especificada com base nas características deste algoritmo e na forma que outras bibliotecas abstraem seu funcionamento. Dentre os parâmetros de configuração elencados no parágrafo anterior, apenas o tamanho da chave tem um efeito direto sobre a quantidade de memória que o algoritmo pode utilizar. A *GEmSysC* deve abstrair da forma mais genérica possível os detalhes de implementação da biblioteca subjacente, o que inclui a estratégia de alocação de memória. Assim, ao definir uma estrutura de memória referente a *AES* na *GEmSysC*, estabeleceu-se que é necessário fornecer o tamanho da chave. Se a implementação subjacente não usar alocação estática ou não diferenciar a forma de definir instâncias do motor de *AES* com base no tamanho das chaves (como ocorre nas duas bibliotecas consideradas), este parâmetro não afeta o gasto de memória, só é utilizado para validar o tamanho da chave. Como a diferença entre a cifragem e a decifragem *AES* é a ordem na qual as operações internas são realizadas [Kak, 2016], a direção do algoritmo também não deve afetar o tamanho dos dados. Como estes parâmetros não devem afetar o gasto de memória, podem ser definidos ao inicializar a instância do mecanismo criptográfico. Para utilizar convenções semelhantes às das bibliotecas e *APIs* utilizadas, determinou-se que a direção do algoritmo determina qual função deve ser utilizada para inicializar a instância do mecanismo de *AES*, enquanto os demais parâmetros do *AES* são definidos através de parâmetros desta função. As funções de inicialização disponíveis são: `clAESEncryptCreate` (cifragem) e `clAESDecryptCreate` (decifragem).

As funções definidas para o módulo de *AES* da *GEmSysC* estão apresentadas na tabela 4.5. Detalhes sobre seu funcionamento podem ser encontrados no apêndice A.

Um exemplo de como é feita cifragem de dados com *AES* em modo *CBC* usando a *GEmSysC* está apresentado no código 4.4. Neste código, `input` e `input_len` são respectivamente o *buffer* de entrada e seu tamanho, `output` e `output_len` são respectivamente o

Tabela 4.5: Lista de funções e *macros* que parecem funções do módulo de *AES* da *GEmSysC*.

`clAES`	Faz referência à estrutura que armazena o estado interno da instância de um mecanismo criptográfico de *AES*.
`clAESEncryptCreate`	Inicializa uma instância do mecanismo criptográfico de *AES* para cifragem de dados.
`clAESDecryptCreate`	Inicializa uma instância do mecanismo criptográfico de *AES* para decifragem de dados.

Código 4.4: Cifragem simétrica de dados com *AES* usando a *GEmSysC*.

```
1  #define BLOCK_LENGTH ( 16 ) /* 128 bits */
2  #define KEY_LENGTH ( 32 ) /* 256 bits */
3
4  clAESDef(aes_engine, CL_AES_KEYLENGTH_256);
5
6  clEngineInstanceId id;
7  ret = clAESEncryptCreate(&id, clAES(aes_engine), CL_BLOCK_CIPHER_MODE_CBC,
       (const void*) key, KEY_LENGTH, (const void*) iv, BLOCK_LENGTH);
8  if (ret < 0) {
9      printf("Erro inicializando cifragem: 0x%X\n", -ret);
10     return -1;
11 }
12
13 ret = clEngineProcess(id, (const void*) input, input_len, (void*) output,
       output_len, NULL);
14 if (ret < 0) {
15     printf("Erro durante cifragem: 0x%X\n", -ret);
16     return -1;
17 }
18
19 clEngineFinalize(id);
```

buffer de saída e seu tamanho, `iv` é o *buffer* com o vetor de inicialização e `key` é o *buffer* com a chave de criptografia.

4.4.3 Módulo de *RSA*

O *RSA* é um algoritmo de criptografia assimétrica criado por Ronald Linn Rivest, Adi Shamir e Leonard Max Adleman e publicado pela primeira vez em 1977 [Rivest et al., 1983]. O nome do algoritmo é a combinação das iniciais de seus criadores. Este algoritmo é utilizado para garantir privacidade na comunicação, gerar assinaturas digitais, entre outras finalidades [Rivest et al., 1978]. O *RSA* é recomendado pelo *NIST* para trocas de chaves entre sistemas [Barker et al., 2014].

O *RSA* utiliza propriedades da aritmética modular para realizar a cifragem e a decifragem dos dados. No *RSA*, a cifragem envolve representar os dados como números, elevá-los a um certo exponente, e computar o resto de divisão por um valor numérico. A decifragem envolve representar os dados cifrados como números, elevá-los a um expoente diferente ao utilizado na cifragem e computar o resto de divisão pelo mesmo número utilizado na cifragem. Os valores dos expoentes e do módulo têm de atender a certas propriedades para que a operação seja possí-

vel. Cada uma das chaves utilizadas no *RSA* é composta de um par expoente-módulo. Antes da cifragem, os dados a serem codificados em *RSA* geralmente recebem um preenchimento (*padding*) de pelo menos 11 *bytes*, denominado preenchimento *PKCS* #1 v1.5, ou de pelo menos 42 *bytes*, denominado preenchimento *OAEP* (com *SHA*-1 e *label* vazia) [RSA, 2012].

As chaves *RSA* podem ser representadas de duas formas: *PKCS* #1 e *PKCS* #8. Tanto uma representação quanto a outra podem ser codificadas em dois formatos: *DER* e *PEM*. O formato *DER*[5], definido no padrão *ASN.1*, é um formato binário no qual os números que compõem a chave são armazenados. O formato *PEM*[6] contém os mesmos dados do formato *DER*, mas codificados em *base64* para facilitar sua transmissão em meios que usem texto, como *e-mail* [Bakker, 2014].

Para utilizar *RSA* na *API Cryptoki*, é necessário estabelecer uma sessão representada por estruturas do tipo `CK_SESSION_HANDLE` com um *token* que tenha suporte a *RSA*. A sessão deve ser preparada para processamento de dados através da função `C_EncryptInit` para cifragem ou da função `C_DecryptInit` para decifragem. O processamento de dados é feito através das funções `C_EncryptUpdate` ou `C_DecryptUpdate` e terminado através das funções `C_EncryptFinal` e `C_DecryptFinal` [Griffin and Fenwick, 2015a]. Uma estrutura do tipo `CK_MECHANISM` com o campo `mechanism` igual a `CKM_RSA_PKCS` deve ser fornecida às funções `C_EncryptInit` e `C_DecryptInit`, O tamanho da chave não é definido diretamente na sessão, mas sim através do objeto que representa a chave. Os objetos que representam as chaves públicas e as chaves privadas são criados a partir dos números (módulos, expoentes, primos e coeficientes) que as compõem. As chaves privadas pertencem à classe `CKO_PRIVATE_KEY`, com tipo `CKK_RSA`. As chaves públicas pertencem à classe `CKO_PUBLIC_KEY`, com tipo `CKK_RSA` [Griffin and Fenwick, 2015b]. Um exemplo de como é feita cifragem de dados com *RSA* com preenchimento *OAEP* com *SHA*-1 e mensagem vazia usando a *Cryptoki* está apresentado no código 4.5. Neste código, `input` e `input_len` são respectivamente o *buffer* de entrada e seu tamanho, `output` e `output_len` são respectivamente o *buffer* de saída e seu tamanho e `encrypt_len` é o número de bytes realmente escritos no *buffer* de saída.

Para ter acesso às funcionalidades de *RSA* pela *OpenSSL*, ao contrário do que ocorre com o *AES*, não é possível utilizar a camada de abstração *EVP*, então é necessário empregar estruturas e funções especializadas. A instância do mecanismo de *RSA* é representado pelo ponteiro para uma estrutura denominada `RSA`. Para a maioria das finalidades, a estrutura `RSA` pode ser tratada como uma estrutura opaca. Para obter um ponteiro para uma estrutura `RSA`, são utilizadas funções que leem chaves *RSA* no formato *PKCS* #1, como `d2i_RSA_PUBKEY` e `d2i_RSAPrivateKey`, que aceitam codificação *DER*, e `PEM_read_bio_RSA_PUBKEY` e `PEM_read_bio_RSAPrivateKey`, que aceitam codificação *PEM*. O processamento de dados é feito através das funções `RSA_public_encrypt`, `RSA_public_decrypt`, `RSA_private_encrypt` e `RSA_private_decrypt`, de acordo com o tipo de chave (privada ou pública) e direção da operação de criptografia (cifragem ou decifragem). Estas funções recebem como entrada os blocos inteiros de dados e fornecem os blocos cifrados ou decifrados correspondentes como saída. As funções de processamento de dados recebem também informação sobre o tipo de preenchimento a ser utilizado. Na *OpenSSL*, é possível utilizar diferentes tipos de preenchimentos, como o *PKCS* #1 v1.5, o *OAEP* e o modo alternativo do *PKCS* #1 v1.5 utilizado no *SSL* 3.0 (durante conexões *SSL* 2.0, este tipo de preenchimento é utilizado

[5]Sigla de *"Distinguished Encoding Rules"*.

[6]Sigla de *"Privacy-enhanced Electronic Mail"*.

Código 4.5: Cifragem assimétrica de dados com *RSA* usando a *Cryptoki*.

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hPublicKey;

/* omitido trecho em que hSession e hPublicKey são inicializadas */

CK_RSA_PKCS_OAEP_PARAMS oaep_params = {CKM_SHA_1, CKG_MGF1_SHA1,
    CKZ_DATA_SPECIFIED, "", 0};
CK_MECHANISM mechanism = {CKM_RSA_PKCS_OAEP, &oaep_params, sizeof(
    CK_RSA_PKCS_OAEP_PARAMS)};

rc = C_EncryptInit(hSession, &mechanism, hPublicKey);
if (rc != CKR_OK) {
    printf("Erro inicializando cifragem: 0x%X\n", rc);
    return rc;
}

CK_ULONG encrypt_len = output_len;
rc = C_Encrypt(hSession, (CK_BYTE_PTR) input, (CK_ULONG) input_len, (
    CK_BYTE_PTR) output, &encrypt_len);
if (rc != CKR_OK) {
    printf("Erro durante cifragem: %x\n", rc);
    return rc;
}

return CKR_OK;
```

para indicar compatibilidade com *SSL* 3.0 [Freier et al., 2011]). Uma estrutura `RSA` é finalizada (isto é, libera os recursos dinâmicos obtidos na inicialização) através da função `RSA_free` [OpenSSL, 2015]. Um exemplo de como é feita cifragem de dados com *RSA* com preenchimento *OAEP* com *SHA*-1 e mensagem vazia usando a *OpenSSL* está apresentado no código 4.6. Neste código, `input` e `input_len` são respectivamente o *buffer* de entrada e seu tamanho, `key` e `key_len` são respectivamente o *buffer* com a chave codificada em *DER* e seu tamanho, `output` é o *buffer* de saída e `encrypt_len` é o número de bytes realmente escritos no *buffer* de saída.

Na *wolfSSL*, as instâncias do mecanismo de *RSA* são presentadas por estruturas do tipo `RsaKey`. Praticamente todas as funções da biblioteca que operam com a estrutura `RsaKey` começam com `wc_Rsa`, exceto a função `wc_InitRsaKey` que é responsável por inicializar esta estrutura. Há duas funções diferentes para leitura de chaves *RSA*: `wc_RsaPublicKeyDecode` para chaves públicas e `wc_RsaPrivateKeyDecode` para chaves privadas, que aceitam apenas formato *PKCS* #1 com codificação *DER*. Na *wolfSSL*, só é possível cifrar dados com a chave pública (função `wc_RsaPublicEncrypt_ex`) e decifrar com a chave privada (função `wc_RsaPrivateDecrypt_ex`). A biblioteca *wolfSSL* suporta dois tipos diferentes de preenchimento com o *RSA*: *PKCS* #1 v1.5 e *OAEP*. Para liberar recursos utilizados pela estrutura `RsaKey`, usa-se a função `wc_FreeRsaKey`. A operação cifragem também tem de receber como parâmetro um gerador de números aleatórios, representado por um ponteiro para uma estrutura do tipo `RNG`. As estruturas do tipo `RNG` são inicializadas pela função `wc_InitRng` e finalizadas pela função `wc_FreeRng` [wolfSSL, 2016]. Um exemplo de como é feita cifragem de dados com *RSA* com preenchimento *OAEP* com *SHA*-1 e mensagem

Código 4.6: Cifragem assimétrica de dados com *RSA* usando a *OpenSSL*.

```
1  const unsigned char * key_ptr = (const unsigned char*) key;
2  RSA * rsa = d2i_RSA_PUBKEY(NULL, &key_ptr, key_len);
3  if (rsa == NULL) {
4      printf("Erro lendo chave\n");
5      return -1;
6  }
7
8  encrypt_len = RSA_public_encrypt(input_len, (const unsigned char*)input, (
       unsigned char*)output, rsa, RSA_PKCS1_OAEP_PADDING);
9  if (encrypt_len < 0) {
10     printf("Erro durante cifragem\n");
11     return -1;
12 }
13
14 RSA_free(rsa);
15
16 return encrypt_len;
```

vazia usando a *wolfSSL* está apresentado no código 4.7. Neste código, `input` e `input_len` são respectivamente o *buffer* de entrada e seu tamanho, `key` e `key_len` são respectivamente o *buffer* com a chave codificada em *DER* e seu tamanho, `output` e `output_len` são respectivamente o *buffer* de saída e seu tamanho e `encrypt_len` é o número de bytes realmente escritos no *buffer* de saída.

Pelas características do *RSA*, observa-se que há pelo menos cinco parâmetros de configuração:

- Tamanho da chave;
- Tipo da chave (privada ou pública);
- Valor da chave;
- Preenchimento utilizado;
- Direção da operação criptográfica (cifragem ou decifragem).

Na *Cryptoki*, as chaves são inicializadas a partir dos valores numéricos de seus parâmetros e o tipo da chave é definido pelo tipo do objeto de chave. Assim, na *Cryptoki*, o tamanho, o tipo e o valor das chaves são definidos ao criar o objeto de chave. Por outro lado, na *OpenSSL* e a *wolfSSL* não é comum acessar diretamente o tamanho das chaves nem seu valor, mas é possível lê-las de formato *DER* (e também formato *PEM*, no caso da *OpenSSL*) com codificação *PKCS* #1. As funções utilizadas para carregar as chaves a partir do formato *DER* (ou *PEM*) definem o tipo da chave. Enquanto não existe influência direta sobre o tamanho e o valor das chaves, estes valores estão implícitos nos dados em formato *DER* (ou *PEM*) utilizados. Na *OpenSSL*, as instâncias do motor de *RSA* são criadas e então inicializadas com chaves. Na *wolfSSL*, a inicialização do motor de *RSA* já envolve a leitura da chave. Nestas bibliotecas o tamanho da chave, seu tipo e seu valor são definidos pela função utilizada para inicializar o motor criptográfico. Nelas, o tipo da chave e a direção do algoritmo também definem que

Código 4.7: Cifragem assimétrica de dados com *RSA* usando a *wolfSSL*.

```
RsaKey rsa;
RNG rng;

ret = wc_InitRsaKey(&rsa, NULL);
if (ret != 0)
{
    printf("Erro inicializando chave: 0x%X\n", rc);
    return ret;
}

ret = wc_InitRng(&rng);
if (ret != 0)
{
    printf("Erro inicializando gerador de numeros aleatorios: 0x%X\n", rc);
    return ret;
}

ret = wc_RsaPublicKeyDecode((const byte*)key, &index, &rsa, key_len);
if (ret != 0)
{
    printf("Erro lendo chave: 0x%X\n", rc);
    return ret;
}

encrypt_len = wc_RsaPublicEncrypt_ex((const byte*)input, input_len, (byte*)
    output, output_len, &rsa, &rng, WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA,
    WC_MGF1SHA1, NULL, 0);
if (ret < 0)
{
    printf("Erro durante cifragem: %x\n", rc);
    return ret;
}

wc_FreeRsaKey(&rsa);
wc_FreeRng(&rng);

return 0;
```

função deve ser utilizada para processar dados. Na *Cryptoki*, o tipo de preenchimento utilizado é definido como parâmetro ao inicializar a sessão para a operação de *RSA*, enquanto na *OpenSSL* e na *wolfSSL*, o tipo de preenchimento é parâmetro das funções que processam dados. Na *Cryptoki*, a direção da operação criptográfica é definida pelo tipo da função utilizada para inicializar a operação de *RSA* e também pelas funções utilizadas para processar dados, enquanto na *OpenSSL* e na *wolfSSL*, é definida pelas funções utilizadas para processar dados, que também variam de acordo com o tipo da chave. Estas características estão resumidas na tabela 4.6.

Tabela 4.6: Como são definidos os parâmetros do *RSA* na *Cryptoki*, na *OpenSSL* e na *wolfSSL*.

Parâmetro	*Cryptoki*	*OpenSSL*	*wolfSSL*
Tamanho da chave	Tipo do objeto da chave	Parâmetro da função de inicialização	Parâmetro da função de inicialização
Tipo da chave	Objeto da chave	Função utilizada para inicialização	Função utilizada para inicialização
Valor da chave	Objeto da chave	Parâmetro da função de inicialização	Parâmetro da função de inicialização
Preenchimento	Parâmetro da função de inicialização	Parâmetro da função de processamento de dados	Parâmetro da função de processamento de dados
Direção	Função utilizada para inicialização	Função utilizada para inicialização	Função utilizada para inicialização

O módulo da *GEmSysC* que contém o motor de *RSA* foi especificado com base nas características do algoritmo e nas características da *APIs* estudadas. De todas as características do *RSA* consideradas no parágrafo anterior, a única que tem influência direta sobre a quantidade de memória utilizada pelo motor criptográfico é o tamanho da chave, já que quanto maior a chave, mais memória tem de ser reservada para armazená-la. Por este motivo, o tamanho máximo da chave é o único parâmetro que deve ser fornecido ao definir a estrutura. Quando a implementação subjacente utiliza alocação dinâmica para alocar memória para as chaves (como é o caso das duas bibliotecas fornecidas), este parâmetro não tem efeito prático e é ignorado. Como nas bibliotecas utilizadas, o tipo da chave e a direção da operação determinam que função deverá ser utilizada para inicializar a instância do mecanismo de *RSA*. Os demais parâmetros são definidos como parâmetros da função de inicialização utilizada. As funções de inicialização disponíveis são: `clRSAPrivateEncryptCreate` (cifragem com chave privada), `clRSAPrivateDecryptCreate` (decifragem com chave privada), `clRSAPublicEncryptCreate` (cifragem com chave pública), `clRSAPublicDecryptCreate` (decifragem com chave pública). Neste módulo, também há quatro funções específicas: `clRSAMinInputLength` (verifica o tamanho mínimo da entrada), `clRSAMaxInputLength` (verifica o tamanho máximo da entrada), `clRSAMinOutputLength` (verifica o tamanho mínimo da saída) e `clRSAMaxOutputLength` (verifica o tamanho máximo da saída).

As funções definidas para o módulo de *RSA* da *GEmSysC* estão apresentadas na tabela 4.7. Detalhes sobre seu funcionamento podem ser encontrados no apêndice A.

Um exemplo de como é feita cifragem de dados com *RSA* com preenchimento *OAEP* com *SHA*-1 e mensagem vazia usando a *GEmSysC* está apresentado no código 4.8. Neste código, `input` e `input_len` são respectivamente o *buffer* de entrada e seu tamanho, `output`

Tabela 4.7: Lista de funções e *macros* que parecem funções do módulo de *RSA* da *GEmSysC*.

`clRSA`	Faz referência à estrutura que armazena o estado interno da instância de um mecanismo criptográfico de *RSA*.
`clRSAPrivateEncryptCreate`	Inicializa uma instância do mecanismo criptográfico de *RSA* para cifragem de dados com chave privada.
`clRSAPrivateDecryptCreate`	Inicializa uma instância do mecanismo criptográfico de *RSA* para decifragem de dados com chave privada.
`clRSAPublicEncryptCreate`	Inicializa uma instância do mecanismo criptográfico de *RSA* para cifragem de dados com chave pública.
`clRSAPublicDecryptCreate`	Inicializa uma instância do mecanismo criptográfico de *RSA* para decifragem de dados com chave pública.
`clRSAMinInputLength`	Verifica o tamanho mínimo do *buffer* de entrada que deve ser fornecido.
`clRSAMaxInputLength`	Verifica o tamanho máximo do *buffer* de entrada que deve ser fornecido.
`clRSAMinOutputLength`	Verifica o tamanho mínimo do *buffer* de saída que deve ser fornecido.
`clRSAMaxOutputLength`	Verifica o tamanho máximo do *buffer* de saída que deve ser fornecido.

e `output_len` são respectivamente o *buffer* de saída e seu tamanho e `key` e `key_len` são respectivamente o *buffer* com a chave de criptografia e seu tamanho.

4.4.4 Módulo de *SHA*-256

O *SHA*-256 é um algoritmo de resumo criptográfico (*hash*) definido no *FIPS* 180-2, publicado pelo *NIST* [NIST, 2002]. No *FIPS* 180-2, tal qual foi originalmente publicado em 2002, estão definidos o algoritmo *SHA*-1 e a família de algoritmos *SHA*-2 (*SHA*-128, *SHA*-256, *SHA*-384, e *SHA*-512). Em 2004, foi feita uma alteração neste documento incluindo na família *SHA*-2 o *SHA*-224 [NIST, 2004].

No final de 2015, o *NIST* publicou o *FIPS* 180-4, em que está definida uma nova família de funções de resumo criptográfico, *SHA*-3, que deve substituir os algoritmos da família *SHA*-2 [NIST, 2015]. Ainda assim, não há suporte a *SHA*-3 nas versões mais recentes das bibliotecas *wolfSSL* (versão 3.9.0) [wolfSSL, 2016] e *OpenSSL* (versão 1.0.2) [OpenSSL, 2015], nem na versão mais recente da especificação de mecanismos da *Cryptoki* (versão 2.40) [Griffin and Fenwick, 2015b]. Assim, decidiu-se utilizar o *SHA*-2, particularmente o *SHA*-256, na prova de conceito da *GEmSysC*.

Para utilizar *SHA*-256 na *Cryptoki*, é necessário estabelecer uma sessão representada por estruturas do tipo `CK_SESSION_HANDLE` com um *token* que tenha suporte a *SHA*-256. A sessão deve ser preparada para processamento de dados através da função `C_DigestInit`, então o processamento de dados deve ser feito através de chamadas à função `C_DigestUpdate` e terminado através de uma chamada à função `C_DigestFinal`

Código 4.8: Cifragem assimétrica de dados com *RSA* usando a *GEmSysC*.

```
clRSADef(rsa_engine, 4096);

clEngineInstanceId id;
ret = clRSAPublicEncryptCreate(&id, clRSA(rsa_engine),
    CL_RSA_PADDING_MODE_OAEP, CL_RSA_KEY_ENCODING_DER,
    CL_RSA_KEY_FORMAT_PKCS1, (const void*) key, key_len);
if (ret < 0) {
    printf("Erro inicializando cifragem: 0x%X\n", -ret);
    return -1;
}

ret = clEngineProcess(id, (const void*) input, input_len, (void*) output,
    output_len, NULL);
if (ret < 0) {
    printf("Erro durante cifragem: 0x%X\n", -ret);
    return -1;
}

clEngineFinalize(id);
```

[Griffin and Fenwick, 2015a]. Uma estrutura do tipo `CK_MECHANISM` com o campo `mechanism` igual a `CKM_SHA256` deve ser fornecida à função `C_DigestInit` [Griffin and Fenwick, 2015b]. Um exemplo de como é gerado o resumo criptográfico com *SHA*-256 usando a *Cryptoki* está apresentado no código 4.9. Neste código, `input` e `input_len` são respectivamente o *buffer* de entrada e seu tamanho, `output` e `output_len` são respectivamente o *buffer* de saída e seu tamanho e `hash_len` é o número de bytes realmente escritos no *buffer* de saída.

Na *OpenSSL*, o algoritmo *SHA*-256 é usado através da abstração *EVP*. O *EVP* é uma representação genérica de um algoritmo de criptografia [OpenSSL, 2013]. Para inicializar o *SHA*-256, é utilizada a função `EVP_DigestInit_ex`. A entrada de dados é feita de forma contínua através da função `EVP_DigestUpdate` e é terminada pela função `EVP_DigestFinal_ex`. O valor do resumo criptográfico só é obtido ao terminar a entrada de dados [OpenSSL, 2015]. Um exemplo de como é gerado o resumo criptográfico com *SHA*-256 usando a *OpenSSL* está apresentado no código 4.10. Neste código, `input` e `input_len` são respectivamente o *buffer* de entrada e seu tamanho, `output` e `output_len` são respectivamente o *buffer* de saída e seu tamanho e `hash_len` é o número de bytes realmente escritos no *buffer* de saída.

Na *wolfSSL*, o *SHA*-256 é utilizado através de funções e estruturas especializadas. A estrutura de dados que armazena o estado interno do *SHA*-256 denomina-se `Sha256`. Esta estrutura é inicializada através da função `wc_InitSha256`. Após a inicialização, é possível realizar a entrada de dados através da função `wc_Sha256Update`. Assim que todos os dados tiverem sido fornecidos, é possível obter o resultado da operação pela função `wc_Sha256Final`. Só é possível obter o valor do resumo criptográfico após terminar a entrada de dados [wolfSSL, 2016]. Um exemplo de como é gerado o resumo criptográfico com *SHA*-256 usando a *wolfSSL* está apresentado no código 4.11. Neste código, `input` e `input_len` são respectivamente o *buffer* de entrada e seu tamanho e `output` é o *buffer* de saída.

Código 4.9: Resumo criptográfico de dados com *SHA*-256 usando a *Cryptoki*.

```
CK_SESSION_HANDLE hSession;

/* omitido trecho em que hSession é inicializada */

CK_MECHANISM mechanism = {CKM_SHA256, 0, 0};

rc = C_DigestInit(hSession, &mechanism);
if (rc != CKR_OK) {
    printf("Erro inicializando resumo criptográfico: 0x%X\n", rc);
    return rc;
}

rc = C_DigestUpdate(hSession, input, input_len);
if (rc != CKR_OK) {
    printf("Erro resumindo dados: 0x%X\n", rc);
    return rc;
}

CK_ULONG hash_len = output_len;
rc = C_DigestFinal(hSession, output, &hash_len);
if (rc != CKR_OK) {
    printf("Erro gerando resumo criptográfico: 0x%X\n", rc);
    return rc;
}

return CKR_OK;
```

Código 4.10: Resumo criptográfico de dados com *SHA*-256 usando a *OpenSSL*.

```
EVP_MD_CTX * ctx = EVP_MD_CTX_create();
if (ctx == NULL)
{
    printf("Erro criando contexto\n");
    return -1;
}

ret = EVP_DigestInit_ex(ctx, EVP_sha256(), NULL);
if (!ret)
{
    printf("Erro inicializando resumo criptográfico\n");
    return -1;
}

ret = EVP_DigestUpdate(ctx, input, input_len);
if (!ret)
{
    printf("Erro resumindo dados\n",);
    return -1;
}

hash_len = output_len;
ret = EVP_DigestFinal_ex(ctx, output, &hash_len);
if (!ret)
{
    printf("Erro gerando resumo criptográfico\n");
    return -1;
}

EVP_MD_CTX_destroy(ctx);

return hash_len;
```

Código 4.11: Resumo criptográfico de dados com *SHA*-256 usando a *wolfSSL*.

```
Sha256 sha256;

ret = wc_InitSha256(&sha256);
if (ret != 0)
{
    printf("Erro inicializando resumo criptográfico: 0x%X\n", ret);
    return ret;
}

ret = wc_Sha256Update(&sha256, (const byte*)input, (word32)input_len);
if (ret != 0)
{
    printf("Erro resumindo dados: 0x%X\n", ret);
    return ret;
}

ret = wc_Sha256Final(&sha256, (byte*)output);
if (ret != 0)
{
    printf("Erro gerando resumo criptográfico: 0x%X\n", ret);
    return ret;
}

return 0;
```

Na *wolfSSL*, usam-se funções e estruturas exclusivas para acessar funcionalidades referentes a *SHA*-256, enquanto na *Cryptoki* e na *OpenSSL*, usam-se estruturas genéricas. Entretanto, a forma de usar o algoritmo é igual nas três bibliotecas. Dados são fornecidos ao algoritmo por repetidas chamadas a uma função, e então o resultado do processamento é obtido através da chamada a uma outra função.

Não há parâmetros de configuração para a operação de *SHA*-256. Assim, a *macro* que define uma instância do mecanismo de *SHA*-256 não recebe nenhum parâmetro adicional. A função genérica de processamento de dados da *GEmSysC*, que também é utilizada com *SHA*-256, recebe um *buffer* de dados como entrada e outro como saída. O comportamento desta função, no caso do *SHA*-256, é a seguinte: quando houver *buffer* de entrada, seu conteúdo é fornecido como entrada, e quando houver *buffer* de saída, os dados resultantes do processamento são escritos nele. Se ambos os *buffers* forem fornecidos, o processamento é feito e então o resultado é gerado. A única função de inicialização disponível é `clSHA256Create`.

As funções definidas para o módulo de *SHA*-256 da *GEmSysC* estão apresentadas na tabela 4.8. Detalhes sobre seu funcionamento podem ser encontrados no apêndice A.

Tabela 4.8: Lista de funções e *macros* que parecem funções do módulo de *SHA-256* da *GEmSysC*.

`clSHA256`	Faz referência à estrutura que armazena o estado interno de uma instância do mecanismo criptográfico de *SHA-256*.
`clSHA256Create`	Inicializa uma instância do mecanismo criptográfico de *SHA-256* para resumo criptográfico de dados.

Código 4.12: Resumo criptográfico de dados com *SHA*-256 usando a *GEmSysC*.

```
clSHA256Def(sha256_engine);

clEngineInstanceId id;
ret = clSHA256Create(&id, clSHA256(sha256_engine));
if (ret < 0) {
    printf("Erro inicializando resumo criptográfico: 0x%X\n", -ret);
    return -1;
}

ret = clEngineProcess(id, (const void*) input, input_len, (void*) output,
   output_len, NULL);
if (ret < 0) {
    printf("Erro gerando resumo criptográfico: 0x%X\n", -ret);
    return -1;
}

clEngineFinalize(id);
```

Um exemplo de como é gerado o resumo criptográfico com *SHA*-256 usando a *GEmSysC* está apresentado no código 4.12. Neste código, `input` e `input_len` são respectivamente o *buffer* de entrada e seu tamanho, `output` e `output_len` são respectivamente o *buffer* de saída e seu tamanho e `key` e `key_len` são respectivamente o *buffer* com a chave de criptografia e seu tamanho.

4.4.5 Discussão

Nesta seção, apresentaram-se as justificativas das decisões acerca da forma do core e dos módulos da *GEmSysC*. Para especificar estes módulos, verificaram-se os parâmetros de configuração de cada um os módulos e a forma que estes parâmetros foram definidos na *Cryptoki*, na *OpenSSL* e na *wolfSSL*.

Na *GEmSysC* buscou-se garantir que os parâmetros que afetam o gasto de memória sejam definidos na declaração das estruturas de dados, enquanto os parâmetros que não afetam o tamanho podem ser definidos nas funções de inicialização. Esta decisão possibilita que a implementação subjacente use alocação estática de memória, pois o tamanho das estruturas é conhecido em tempo de compilação. O esforço para garantir que seja possível utilizar alocação estática está alinhado com o discutido na seção 4.3.1.

4.5 Análise

Neste capítulo, apresentou-se uma visão geral da estrutura da *API GEmSysC* e de três de seus módulos: *AES*, *RSA* e *SHA*-256. Explicaram-se e justificaram-se as decisões de projeto feitas para cada um destes itens.

A avaliação de funcionalidades de outras bibliotecas e *APIs* criptográficas levou em conta o *Cryptoki*, que é um padrão de *API* criptográfica, a *OpenSSL*, que é uma biblioteca criptográfica que é de fato padrão, e a *wolfSSL*, que é de código aberto e voltada a sistemas

embarcados. Com isto, se tinha um exemplar de biblioteca criptográfica de uso geral, de biblioteca criptográfica para sistemas embarcados e de *API* criptográfica genérica. Estes estudos demonstraram as características em cada um destes "nichos", mas a avaliação poderia ser mais rica se tivessem sido levadas em conta outras bibliotecas e *APIs*.

A *GEmSysC* é extensível e modular, como sugerido em [Blanchette, 2008]. Isto garante que suporte a novos mecanismos criptográficos possa ser adicionado no futuro sem alterar outras partes da *GEmSysC*. Outra vantagem da modularidade é permitir que haja implementações da *GEmSysC* com apenas alguns módulos, disponíveis de acordo com a biblioteca utilizada.

Como sugerido por diversos autores como "boa prática" ao elaborar uma *APIs* [Henning, 2007, Bandi et al., 2003, Doucette, 2008, Rama and Kak, 2015], a *GEmSysC* tem um conjunto reduzido de funções. Isto é evidente pelo fato de haver uma única função de processamento de dados, em lugar de funções diferentes, como há nas outras *APIs* estudadas.

A forma de declarar estruturas na *GEmSysC* foi elaborada de forma a permitir que a implementação da camada de abstração possa utilizar somente alocação estática. Isto não garante que o sistema como um todo vai utilizar somente alocação estática, pois a biblioteca subjacente pode depender de alocação dinâmica. Por outro lado, se a biblioteca utilizar alocação estática, a *GEmSysC* preservará estas características. A ênfase no uso de alocação estática é devido a esta estratégia de alocação ser mais adequada em sistemas embarcados [Gradinaru, 2010, MacMillan, 2011, Graves, 2011]. Em geral, a *API* também foi elaborada de forma que parâmetros complexos são passados por referência, como sugerido em [Barry and Hartnett, 2007].

Não faz parte da especificação a *API* interna da *GEmSysC*. Isto poderia ser considerado como uma fraqueza da especificação, pois permite que códigos internos de uma implementação não sejam compatíveis com outras implementações. Por outro lado, esta liberdade permite que os desenvolvedores tenham liberdade para implementar a *GEmSysC* da forma mais eficiente possível com base nas características da biblioteca utilizada.

Capítulo 5

Avaliação e Resultados

Neste capítulo, estão apresentados os procedimentos de avaliação aos quais se submeteu a *GEmSysC* e os resultados destes testes. Inicialmente, buscou-se avaliar o quanto a *GEmSysC* simplifica o desenvolvimento de sistemas em comparação com o uso de outras bibliotecas. Avaliou-se também o custo em termos de tempo de processamento e memória de duas implementações sobre bibliotecas criptográficas em plataformas computacionais diferentes. Neste capítulo também estão discutidas as implementações realizadas. O capítulo está organizado da seguinte forma: primeiramente estão apresentadas as duas implementações da *GEmSysC*, e então estão expostas as três formas de avaliação às quais a *GEmSysC* e suas implementações foram submetidas.

5.1 Implementações

A *GEmSysC* é uma *API* criptográfica genérica para sistemas embarcados, implementada como uma camada de abstração sobre bibliotecas criptográficas existentes. Há apenas uma especificação da *GEmSysC*, mas pode haver múltiplas implementações.

A *GEmSysC* foi elaborada com sistemas embarcados como foco, mas isto não limita sua utilização a tais sistemas. Em última instância, todo sistema embarcado é apenas um sistema computacional com muitas limitações. Uma aplicação para computador, por exemplo, pode ser desenvolvida com o uso desta *API*, ainda que este não seja seu objetivo principal.

Para a avaliação, foram desenvolvidas duas implementações da *GEmSysC*: uma sobre a biblioteca *wolfSSL* [wolfSSL, 2016] e outra sobre a *OpenSSL* [OpenSSL, 2015]. Estas bibliotecas foram escolhidas por serem bem documentadas, bem utilizadas e de código aberto. As duas bibliotecas são multiplataforma, então as duas implementações podem ser utilizadas em diversas arquiteturas.

A *wolfSSL* é voltada para uso em sistemas embarcados, enquanto a *OpenSSL* é uma biblioteca mais utilizada em sistemas como computadores pessoais e servidores . Assim, decidiu-se utilizar a *GEmSysC* sobre *wolfSSL* para avaliar a *API* em uma plataforma embarcada (*Cortex-M3*), e a *GEmSysC* sobre *OpenSSL* para avaliar a *API* em um computador convencional (plataforma *x86*). Como tanto a *OpenSSL* quanto a *wolfSSL* têm suporte à plataforma *x86*, tanto o desenvolvimento quanto os testes funcionais das duas implementações foram realizadas em um computador pessoal, e apenas os testes de avaliação da implementação sobre *wolfSSL* precisaram ser realizados sobre um processador *Cortex-M3*.

Apesar de a definição da *GEmSysC* e de seus módulos de *AES*, *RSA* e *SHA*-256 terem sido elaboradas de forma independente da implementação, certas funcionalidades não estão disponíveis em todas as bibliotecas, como discutido nas subseções seguintes. Por exemplo, na implementação de *RSA* da *wolfSSL*, só é possível cifrar dados com a chave pública e decifrar com a chave privada. Assim, a implementação da *GEmSysC* sobre a *wolfSSL* não é capaz de cifrar dados com a chave privada nem de decifrar dados com a chave pública, pois a biblioteca subjacente não implementa estas funcionalidades. Neste caso, as funções da *GEmSysC* que realizam estas operações retornam à aplicação indicando que tais funcionalidades não estão indisponíveis.

As implementações da *GEmSysC* foram feitas como provas de conceito, então nem todos os mecanismos englobam todas as funcionalidades possíveis referentes a cada algoritmo criptográfico. Para o módulo de *AES*, foram suportados apenas dois modos de operação, *ECB* e *CBC*. Como tanto a *wolfSSL* quanto a *OpenSSL* suportam os dois modos de operação, não há limitações em nenhuma das implementações. É importante notar, entretanto, que o *ECB* é considerado inseguro [Arcieri, 2012], e só foi adicionado à *GEmSysC* pois é o modo mais simples de operação, e para manter compatibilidade com sistemas que utilizam esta forma de preenchimento. Para o módulo de *RSA*, foram suportados dois tipos de preenchimento: *PKCS* #1 v1.5 e *OAEP* com *SHA*-1. Não foi adicionado suporte a outras variantes do preenchimento *OAEP* na prova de conceito. O preenchimento *PKCS* #1 v1.5 é considerado inseguro [Bleichenbacher et al., 1998], mas foi adicionado à *GEmSysC* para manter compatibilidade com sistemas que utilizam esta forma de preenchimento. No módulo de *RSA*, foi adicionado suporte às codificações *DER* e *PEM* e ao formato *PKCS* #1. Na prova de conceito, não adicionou-se suporte ao formato *PKCS* #8, pois nem a *wolfSSL* nem a *OpenSSL* têm suporte a este formato. A função da camada *GEmSysC* que inicializa a instância do mecanismo criptográfico também decodifica as chaves, então, como a *wolfSSL* não suporta formato *PEM*, esta função na implementação sobre *wolfSSL* retorna código de erro se houver uma tentativa de usá-la com formato *PEM*. O módulo de *SHA*-256 suporta geração de resumo criptográfico sem restrições em ambas as implementações.

5.1.1 *GEmSysC* sobre *OpenSSL*

Todos os módulos da *GEmSysC* implementada sobre *OpenSSL* têm todas as funcionalidades previstas na especificação sem limitações. Nesta implementação, as estruturas que representam as instâncias de algoritmos criptográficos são estaticamente alocadas. Entretanto, como parte destas estruturas estaticamente alocadas, há campos que são dinamicamente alocados. Por exemplo, na estrutura de dados estaticamente alocada pode haver um ponteiro para um contexto *EVP*, que deve ser alocado dinamicamente quando o mecanismo for inicializado.

A camada de abstração sobre *OpenSSL* implementa o polimorfismo do mecanismo criptográfico através de ponteiros de funções. Estes ponteiros de funções não são definidos por instância de algoritmo criptográfico, pois isto seria ineficiente em termos de memória: haveria por instância um ponteiro de função para cada método virtual. Assim, adotou-se o uso de tabela de métodos virtuais[1], em que há uma tabela de métodos para cada tipo de mecanismo (classe), e cada instância tem apenas um ponteiro para esta tabela.

Quando utiliza-se a abstração *EVP*, é necessário criar dinamicamente um "contexto *EVP*" e inicializá-lo para uso de um mecanismo criptográfico. Nas implementações dos mó-

[1] *"Virtual method table"* em inglês.

dulos de *AES* e *SHA*-256, que utilizam a abstração *EVP*, a estrutura de dados estaticamente alocada que representa a instância de algoritmo criptográfico armazena um ponteiro para um contexto *EVP*, que é alocado dinamicamente.

No caso do *RSA*, a implementação utiliza estruturas de tipo específico, em contrate com tipos genéricos (EVP) utilizados com *AES* e *SHA*-256. A estrutura que armazena o estado do mecanismo de *RSA* da *OpenSSL*, e as funções que realizam as operações de cifragem, decifragem, estimativa e tamanho e liberação de recursos são todas diferentes das utilizadas com a abstração *EVP*.

5.1.2 *GEmSysC* sobre *wolfSSL*

A *wolfSSL* não tem todas as funcionalidades previstas na especificação da *GEmSysC*, então algumas funções da implementação da *GEmSysC* sobre wolfSSL retornam ao código principal indicando erro de funcionalidade não disponível. Nesta implementação, as estruturas que representam as instâncias de algoritmos criptográficos são estaticamente alocadas, mas podem ter campos dinamicamente alocados. Como ocorre na camada sobre *OpenSSL*, a camada de abstração sobre *wolfSSL* utiliza tabelas de métodos virtuais para implementar o polimorfismo.

No módulo de *AES*, a instância do algoritmo criptográfico contém uma estrutura do tipo `Aes`, cujos campos são todos alocados de forma estática. Neste módulo, estão implementadas todas as funcionalidades previstas na especificação. No módulo de *RSA*, a instância do algoritmo criptográfico contém uma estrutura do tipo `RsaKey`, que contém campos alocados dinamicamente com números que compõem a chave *RSA*. Neste módulo, não é suportada a leitura de chaves com codificação PEM, Também não são suportadas as operações de cifrar dados com a chave privada e decifrar dados com a chave pública. O módulo de *SHA*-256 contém uma estrutura do tipo `Sha256`, cujos campos são todos alocados de forma estática. Este módulo não tem limitações em relação à especificação.

5.1.3 Comparação

Foram feitas duas implementações da *GEmSysC*, uma sobre *OpenSSL* e outra sobre *wolfSSL*. Apesar de forneceram às aplicações as mesmas funcionalidades, estas bibliotecas são utilizadas de formas bastante diferentes. Na *OpenSSL*, as estruturas de dados são geralmente declaradas como ponteiros e alocadas dinamicamente. Na *wolfSSL*, as estruturas de dados são geralmente alocadas estaticamente ou na pilha, e operadas através do ponteiro. Esta diferença entre as bibliotecas é totalmente abstraída pela camada *GEmSysC*.

Na *wolfSSL*, grande parte das estruturas é alocada estaticamente. Algumas poucas exceções incluem as chaves RSA, que são alocadas dinamicamente. Como a implementação sobre *wolfSSL* utiliza menos alocação dinâmica, ela é mais adequada à utilização em ambientes embarcados, tal qual defendido em [Gradinaru, 2010, MacMillan, 2011, Graves, 2011]. Por outro lado, a camada sobre *OpenSSL* tem mais funcionalidades, então, se houvesse mais módulos, ela seria preferível nas plataformas em que não há tanta limitação quanto ao poder de processamento e à memória.

5.2 Testes e avaliação

Para avaliar a *GEmSysC*, foram feitos três tipos de avaliação. A primeira buscou avaliar o quanto uma *API* unificada deve afetar a complexidade de código e por consequência a facilidade com que se faz o desenvolvimento de aplicações. Esta avaliação é qualitativa e subjetiva, mas foi construída uma metodologia para fazê-la de forma quantitativa. Para tal, elaborou-se um conjunto de métricas para avaliar a complexidade de aprendizado das *APIs*. Como a maioria das técnicas para realizar este tipo de avaliação são heurísticas, é possível que resultados diferentes dos apresentados neste documento sejam obtidos se forem utilizadas outras metodologias.

No segundo tipo de avaliação realizado, buscou-se demonstrar quanto o uso de uma camada *GEmSysC* afeta o desempenho de programas. Para tal, foram feitos testes comparativos de desempenho entre as implementações da *GEmSysC* e as bibliotecas utilizadas nas implementações sem a camada de abstração. Estes testes dependem da eficiência da implementação da camada *GEmSysC*, então resultados diferentes podem ser obtidos com outras implementações.

A opção por comparar a *GEmSysC* com a *wolfSSL* e com a *OpenSSL* na etapa de avaliação foi tomada por dois motivos. O primeiro motivo é devido a estas bibliotecas serem exemplares de código aberto e publicamente acessíveis de bibliotecas criptográficas voltadas, respectivamente, a sistemas embarcados [wolfSSL, 2016] e a sistemas de computação geral [OpenSSL, 2015]. O segundo motivo desta opção é o fato de estas bibliotecas terem sido amplamente estudadas para realizar as implementações da *GEmSysC*, o que facilitou o processo de criação de códigos comparativos ou de aferimento da adequação das métricas escolhidas.

O terceiro tipo de avaliação a que se submeteu a *GEmSysC* buscou demonstrar o quanto esta camada afeta o tamanho dos programas. Para isso, mediu-se o espaço de memória utilizado pelas funções, constantes e variáveis das mesmas duas implementações utilizadas no segundo tipo de avaliação.

5.2.1 Complexidade de código

Um dos objetivos da *GEmSysC* é simplificar o desenvolvimento, por permitir que uma mesma *API* seja utilizada em diferentes projetos. Diferentes projetos podem ser desenvolvidos em diferentes arquiteturas e utilizar diferentes bibliotecas criptográficas. Uma *API* unificada, como é o caso da *GEmSysC*, simplifica o desenvolvimento, pois diminui a curva de aprendizado e permite reaproveitamento de código entre projetos [Johnson, 1997, Renaux and Pottker, 2014], mesmo que sejam desenvolvidos em plataformas computacionais diferentes. Também é desejável que uma *API* seja mais simples, pois isso melhora sua usabilidade [Henning, 2007].

As avaliações de usabilidade de *APIs* são geralmente feitas de forma qualitativa, o que tem um custo temporal muito elevado [Doucette, 2008]. Como alternativa a estas avaliações qualitativas, usam-se métricas heurísticas que permitem avaliação quantitativa da usabilidade. Em geral, estas métricas são projetadas com vistas em modelos orientados a objetos [Bandi et al., 2003, Doucette, 2008, de Souza and Bentolila, 2009, Rama and Kak, 2015], mas elas podem ser adaptadas para utilização em paradigmas estruturados baseados em procedimentos e funções. Diversas métricas existem, as quais podem considerar, de diferentes formas: o número de funções ou métodos, o número de parâmetros de funções ou métodos, o tipo dos parâmetros, o número de classes, entre outros parâmetros. Estas avaliações são heurísticas e

geralmente são utilizadas para medir a complexidade de uma única *API* ou para comparar duas *APIs* diferentes.

No estudo em [Bandi et al., 2003], são apresentadas métricas de complexidade de *software*. Apesar de os autores do referido artigo não considerarem suas métricas suficientes para modelar a complexidade de qualquer *API*, eles consideram que elas ajudam a estimar o esforço de manutenção de código que a utiliza. O problema desta métrica é que ela assume um modelo orientado a objetos, então não pode ser utilizada diretamente com bibliotecas que não são orientadas a objetos. Apesar de ser possível considerar que a *GEmSysC* é orientada a objetos, não há como utilizar esta métrica, pois tanto a *wolfSSL* quanto a *OpenSSL* não são orientadas a objetos.

No estudo em [Rama and Kak, 2015], também foram apresentadas métricas para avaliar a usabilidade de *software*. No artigo em questão, foram apresentados diversos testes evidenciando a adequação das métricas propostas. As métricas do artigo também foram elaboradas para modelos orientados a objetos, então não podem ser aplicadas diretamente para comparar a *GEmSysC* à *wolfSSL* e à *OpenSSL*.

Como as métricas de ambos os artigos já foram validadas e inclusive aplicadas por outros autores [de Souza and Bentolila, 2009], considerou-se que elas são adequadas para avaliar *APIs*. Inspirou-se nestas métricas para elaborar as métricas utilizadas para avaliar a *GEmSysC*. Aplicaram-se também outras métricas próprias para comparar o quanto a *API* unificada pode simplificar o desenvolvimento. Estas métricas estão apresentadas nos parágrafos seguintes.

A avaliação da *GEmSysC* teve como objetivo verificar os benefícios de uma *API* unificada em relação às *APIs* individuais. Ao estudar as metodologias de [Bandi et al., 2003] e [Rama and Kak, 2015], notou-se que a maioria das métricas não são aplicáveis à *GEmSysC*. Este é o caso das métricas que levam em conta o tamanho dos parâmetros, pois os tipos opacos da *GEmSysC* podem ser traduzidos em tipos diferentes de acordo com a implementação, ou seja, seu tamanho não é determinado pela especificação da *API*, mas sim pela implementação. Outras métricas que não são aplicáveis diretamente são as que a avaliam classes como um todo ou as que avaliam métodos sobrecarregados, pois não existem tais conceitos na linguagem *C*. Entretanto, métricas envolvendo métodos podem ser aplicadas às funções da *API*, se funções foram tratadas como métodos.

Tanto a *wolfSSL* quanto a *OpenSSL* têm suporte a diversas funcionalidades, enquanto a *GEmSysC* suporta apenas um conjunto muito reduzido. Por esse motivo, não se considerou adequado comparar os módulos inteiros das duas bibliotecas aos módulos das *GEmSysC*. A abordagem adotada comparou apenas as estruturas de dados e funções que realizam as mesmas funcionalidades suportadas pela *GEmSysC*. Para esta finalidade, foram enumeradas as funcionalidades que a *GEmSysC* suporta, e então foram feitos códigos com a *GEmSysC*, com a *OpenSSL* e com a *wolfSSL* para cada uma destas funcionalidades. Estes códigos foram utilizados para comparar as três *APIs* e podem ser encontrados no apêndice B. As funcionalidades avaliadas foram:

- Cifragem e decifragem de dados usando *AES* em modo *ECB*.
- Cifragem e decifragem de dados usando *AES* em modo *CBC*.
- Cifragem e decifragem de dados usando *RSA* com preenchimento *PKCS* #1 v1.5.
- Cifragem e decifragem de dados usando *RSA* com preenchimento *OAEP*.

- Geração de resumo criptográfico por *SHA-256*.

Tanto o estudo em [Bandi et al., 2003] quanto o estudo em [Rama and Kak, 2015] utilizam métricas para avaliar quantos parâmetros as funções das *APIs* recebem. Em ambos os estudos, considera-se que listas muito extensas de parâmetros são ruins, então quanto maior o valor desta métrica, maior a complexidade da *API*. Decidiu-se adotar uma versão adaptada da métrica de "tamanho de interface" apresentada em [Bandi et al., 2003]. O tamanho de interface avalia o número de parâmetros e seus tipos. Como avaliações que consideram o tipo dos parâmetros não são adequadas para avaliar a *GEmSysC*, avaliou-se somente o número de parâmetros das funções utilizadas. Como o objetivo principal deste estudo é comparar a *GEmSysC* com as outras bibliotecas, é ideal que haja um único valor de complexidade a ser comparado, não um valor por função. Como não é possível calcular adequadamente a complexidade por classe (pois não há classes), decidiu-se adotar uma abordagem diferente. A referência desta avaliação foram os códigos de exemplo do apêndice B, então considerou-se que a complexidade referente ao número de parâmetros para realizar cada uma das operações consideradas é a soma dos números de parâmetros de cada uma das funções utilizadas. Decidiu-se utilizar esta métrica para avaliar a *API* em quanto ela afeta a complexidade de códigos que a utilizam, e não em quanto ela afeta a de facilidade de aprendizado. Por esse motivo, considerou-se o número total de parâmetros utilizados no código, não o total de parâmetros de cada uma das funções. Em outros termos, se uma função com x parâmetros aparecer duas vezes em um determinado código, o valor da métrica de parâmetros desta função neste código é de $2x$. Para esta avaliação, consideraram-se *macros* parametrizadas como funções, pois são utilizadas da mesma forma. Outra métrica adotada para avaliar a complexidade de códigos resultantes foi o total de linhas que cada um dos códigos de exemplo ocupa. Com estas métricas, é possível observar quanto a *API* afeta a complexidade horizontalmente, isto é, a cada linha de código (total de parâmetros), e verticalmente, isto é, na extensão do código (número de linhas).

Em [Henning, 2007], afirma-se que *APIs* são melhores se forem mais simples, o que inclui o número de funções ou métodos distintos utilizados. Portanto, parece adequado haver uma métrica de complexidade que considere uma *API* mais complexa à medida que tem mais métodos (ou funções). Assim, mediu-se o número de funções diferentes utilizadas para implementar cada uma das funcionalidades. Decidiu-se utilizar essa métrica para avaliar o quanto a *API* por si só é complexa, o que afeta sua usabilidade e sua facilidade de aprender. Para esta avaliação, consideraram-se macros parametrizadas como funções. O número de constantes e tipos de dados diferentes utilizados pode afetar a complexidade pelo mesmo motivo que o número de funções, então a quantidade destes elementos também foi considerada.

Além de se comparar o número de parâmetros, métodos e constantes da *GEmSysC* com cada uma das bibliotecas, esta avaliação demonstra outro fator: quanto seria o custo em termos de complexidade de utilizar duas bibliotecas diferentes em um mesmo projeto. Por exemplo, em um mesmo projeto, a *wolfSSL* poderia ser utilizada no *firmware* de um dispositivo embarcado e a *OpenSSL* poderia ser utilizada no *software* executado em um servidor. Se a mesma *API* for utilizada em ambos os sistemas, o número de construções que os desenvolvedores têm de aprender reduz-se apenas ao conjunto da *GEmSysC*, não à soma dos conjuntos da *wolfSSL* e da *OpenSSL*.

As métricas calculadas para cada uma das funcionalidades estão apresentadas no apêndice C. O resumo destes dados está apresentado nos gráficos 5.1, 5.2, 5.3, 5.4 e 5.5. Além de serem apresentadas as métricas para cada uma das funcionalidades, também foram apresentadas métricas calculadas para todas as funcionalidades em conjunto. No gráfico 5.1, estão

apresentadas as quantidades de tipos de dados diferentes definidos dentro das *APIs* que são utilizados em cada um dos códigos de testes. No gráfico 5.2, estão apresentadas as quantidades de constantes diferentes definidas dentro das *APIs* que são utilizadas em cada um dos códigos de testes. No gráfico 5.3, estão apresentadas as quantidades totais de funções e de macros parametrizadas de dados diferentes definidos dentro das *APIs* que são utilizados em cada um dos códigos de testes. No gráfico 5.4, está apresentado o total de parâmetros de funções utilizados em cada um dos códigos de exemplo. No gráfico 5.5, está apresentado o total de linhas de cada um dos códigos de exemplo, desconsiderando linhas vazias e comentários de código.

As métricas em conjunto apresentadas não servem para avaliar o quanto a simplicidade de um *software* em particular é afetada por cada uma das *APIs*, pois considera que se utilizam o *AES* e o *RSA* com mais de uma configuração diferente, o que dificilmente aconteceria em uma aplicação real. Por outro lado, estas métricas conjuntas evidenciam o quão complexa é cada uma das *APIs*, se delimitadas ao subconjunto de funcionalidades consideradas na avaliação. Nas métricas conjuntas, não se considerou o número de linhas nem o somatório dos parâmetros. Isto foi pois em um código que realizasse todas as operações, decisões sobre a forma de programar poderiam afetar enormemente os resultados. Por exemplo, seria possível utilizar apenas duas declarações de mecanismo criptográfico para todos os códigos de *AES* e de *RSA* da *GEmSysC*, o que reduziria o número de linhas de código e de *macros* parametrizadas (a declaração de algoritmo criptográfico da *GEmSysC* é uma *macro* parametrizada).

Para cada funcionalidade em particular, a *GEmSysC* é mais simples que a *OpenSSL*. Isto é evidenciado pelo fato de, no geral, a *GEmSysC* necessitar que se usem menos funções ou macros parametrizadas e permita que se implementem as funcionalidades com menos linhas de código, como apresentado nos gráficos 5.3 e 5.5. A única métrica através da qual a *GEmSysC* é significativamente mais complexa que a *OpenSSL* é em relação ao número de parâmetros de funções ou macros parametrizadas, apresentada no gráfico 5.4. A mesma análise pode ser

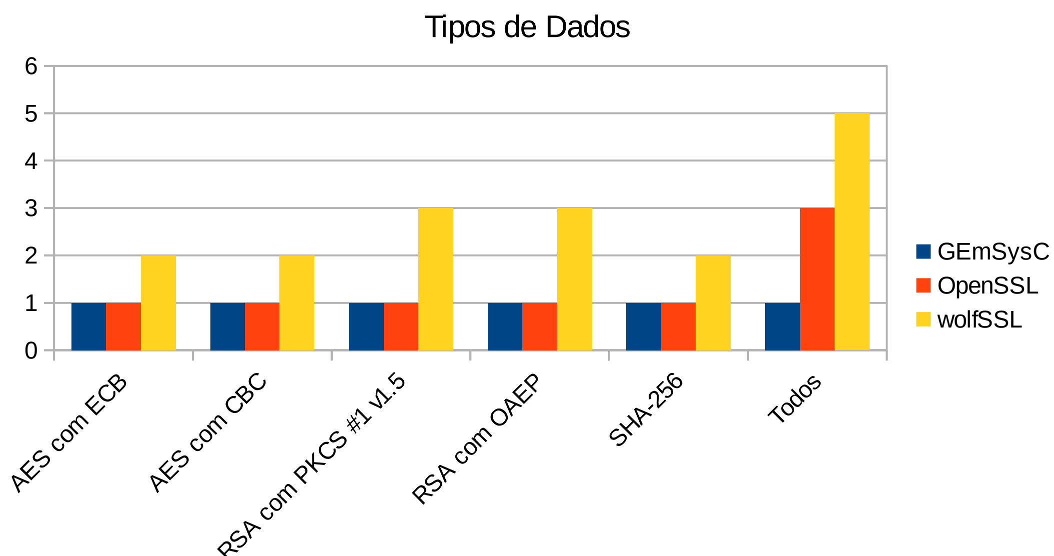

Figura 5.1: Número de tipos de dados definidos na *API* para cada funcionalidade avaliada (autoria própria).

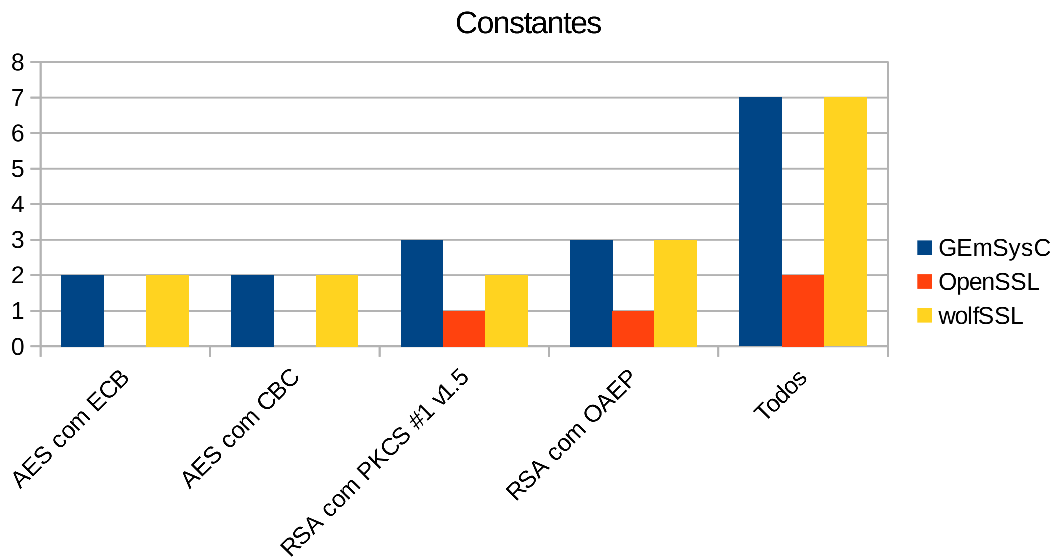

Figura 5.2: Número de constantes definidas na *API* para cada funcionalidade avaliada. O *SHA-256* foi omitido pois é igual a zero para todas as *APIs* (autoria própria).

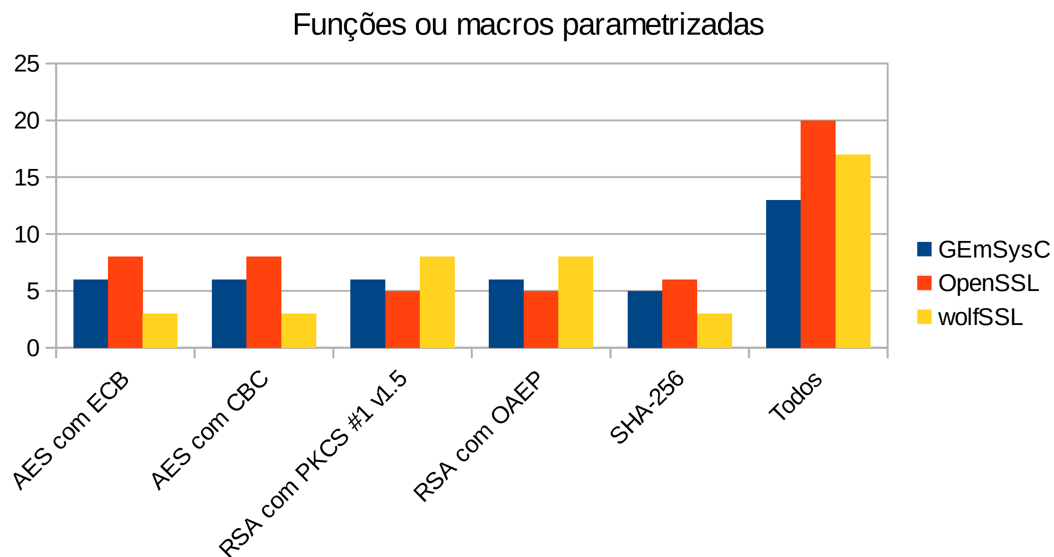

Figura 5.3: Número de funções e *macros* parametrizadas para cada funcionalidade avaliada (autoria própria).

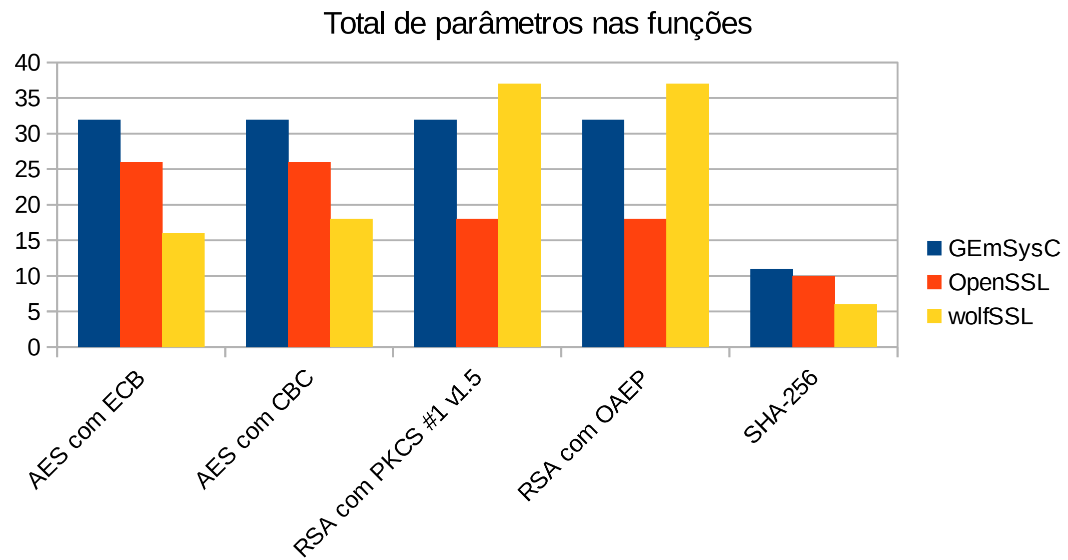

Figura 5.4: Número total de parâmetros das funções e *macros* parametrizadas para cada funcionalidade avaliada (autoria própria).

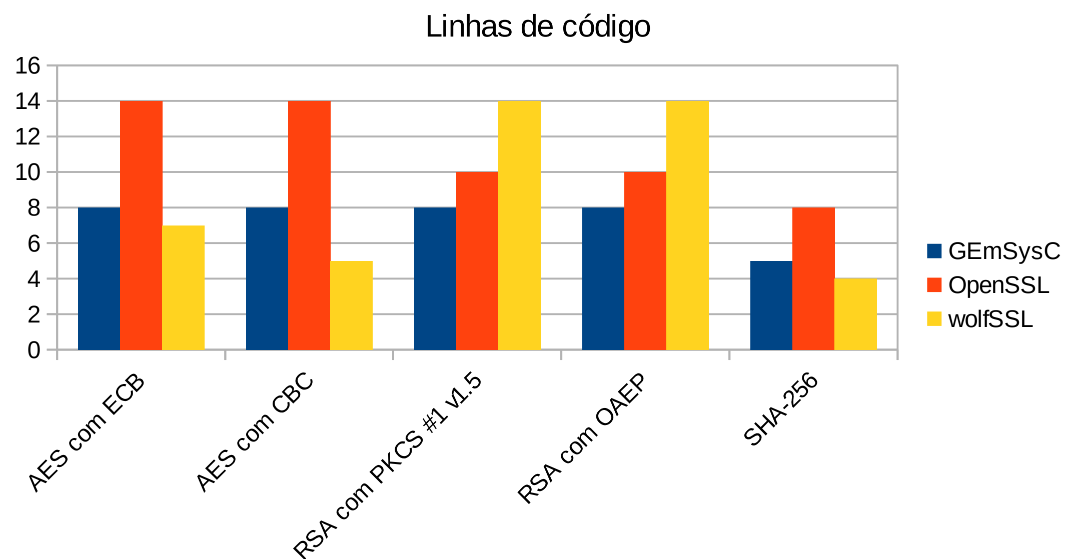

Figura 5.5: Número de linhas de código para cada funcionalidade avaliada (autoria própria).

aplicada para afirmar que a *wolfSSl* é mais simples que a *GEmSysC* para cada funcionalidade em particular.

Por outro lado, observando-se a *API* como um todo, nota-se que a *GEmSysC* é mais simples que a *OpenSSL* e a *wolfSSL*. Isto é evidenciado pelo fato de que a *GEmSysC* tem um conjunto total de funções muito menor e menos tipos de dados, como apresentado nos gráficos 5.3 e 5.1. O único fator em que a *GEmSysC* como um todo é mais complexa que a *OpenSSL* é em relação ao número total de constantes utilizadas nos códigos de exemplo. A maior simplicidade da *GEmSysC* na avaliação total se deve ao fato de a *GEmSysC* empregar sempre as mesmas funções para processamento de dados e liberação de recursos de memória, sem depender do algoritmo criptográfico utilizado. Em outros termos, esta mais simplicidade se deve à sua maior consistência.

Na avaliação, consideraram-se as *macros* da *GEmSysC* que declaram estruturas de dados e que as referenciam na contagem de funções ou macros parametrizadas. Nas outras bibliotecas, as declarações de estruturas de dados e a referência a elas não foram contabilizadas na avaliação de complexidade, pois são apenas declarações de variáveis, então é possível considerar que houve certa tendenciosidade na avaliação. Se estas *macros* em particular fossem excluídas da contagem total, seria diminuída a contagem de funções ou macros parametrizadas e de total de parâmetros de todas as tabelas anteriores com as métricas. Nesta hipótese, a *GEmSysC* apresentaria resultados melhores que os presentes nesta avaliação.

5.2.2 Desempenho

Uma camada *GEmSysC* construída sobre uma biblioteca criptográfica representa um custo computacional adicional nas aplicações, já que quando esta camada é utilizada, há um código adicional entre a aplicação e as funções da biblioteca. Cada um dos módulos da *GEmSysC* foi projetado de forma a refletir diretamente as características dos algoritmos criptográficos correspondentes, e geralmente, o mesmo se aplica aos módulos criptográficos das implementações subjacentes. Por este motivo, geralmente cada implementação da *GEmSysC* deve definir estruturas e funções análogas às da biblioteca, de forma que a camada *GEmSysC* tem apenas de fazer a "tradução" entre declarações, definições e invocações de funções, ou seja, na maioria dos casos, espera-se que esta camada seja computacionalmente leve.

Ambas as implementações da *GEmSysC* foram avaliadas com relação ao quanto afetam o tempo de execução. Apesar de estes testes terem fornecido evidências de quanto o emprego da *GEmSysC* pode afetar aplicações em termos de desempenho, seu resultado dependeu diretamente da maneira que se utilizou para implementar a camada *GEmSysC*. Assim, é possível que outras implementações da camada sobre as mesmas bibliotecas apresentem resultados um pouco diferentes, mas não muito diferentes, já que em todos os casos ela deve ser computacionalmente leve.

Para os testes comparativos, escolheu-se um conjunto de operações criptográficas suportadas integralmente por ambas as implementações. Para cada uma destas operações, comparou-se o tempo de execução sem e com a camada *GEmSysC*. Isto foi feito executando a mesma operação repetidamente, e dividindo-se o tempo total de execução pelo número de repetições. Por opção, o tempo total não inclui a inicialização e a finalização das instâncias de mecanismos criptográficos, apenas o tempo de processamento de dados. Isto foi feito pois medições de desempenho apresentadas em outros trabalhos costumeiramente conside-

Tabela 5.1: Funções da camada de abstração de testes.

IMPL_PRINTF	Imprime um texto no console. Tem a mesma semântica da `printf` da biblioteca padrão de *C*.
IMPL_GETC	Lê um carácter do console. Tem a mesma semântica da `getchar` da biblioteca padrão de *C*.
IMPL_START_TIMER	Inicia uma contagem de tempo.
IMPL_END_TIMER	Interrompe uma contagem de tempo.
IMPL_READ_TIMER	Lê a última contagem de tempo.

ram a vazão de dados[2] como métrica principal [Lin and Huang, 2007, Jindal and Singh, 2012, Papagiannopoulos, 2013].

Como descrito no início desta seção, os testes foram realizados em duas plataformas: *x86* e *Cortex-M*. O código em *x86* foi testado em uma máquina com sistema operacional *Windows* 10 *Home*. Esta máquina foi escolhida para os testes pois foi a utilizada para desenvolvimento. O computador utilizado tinha 4 GB de *RAM* e um processador *Core 2 Quad Q8300*. Os testes em plataforma *Cortex-M3* foram feitos em uma placa de prototipação da *mbed* com um processador *NXP LPC1768* [mbed, 2016]. Devido à simplicidade dos testes a serem realizados na placa embarcada, que não necessitam de nenhum mecanismos como semáforos, filas de mensagens, entre outros, não se achou necessário utilizar um sistema operacional embarcado. Esta placa foi escolhida devido à sua facilidade de uso.

Em ambiente *Windows*, é possível utilizar diretamente chamadas convencionais da biblioteca padrão de *C* como `printf`, para imprimir dados no *console* [Microsoft, 2015a, IEEE, 2013], ou chamadas da *API* do *Windows*. Devido à simplicidade dos testes a serem realizados, decidiu-se não utilizar um sistema operacional embarcado no código em *Cortex-M3*. Com isso, não se gastou tempo de desenvolvimento para inserir o sistema operacional no projeto para a plataforma embarcada, mas deixou de ser possível utilizar chamadas como `printf`, que dependem do sistema operacional. Em lugar do sistema operacional, utilizou-se a biblioteca de desenvolvimento da *mbed*, que tem *drivers* que permitem, entre outras coisas, comunicação pela *UART* (para exibição dos resultados em um computador) e medição de tempo (utilizando *timers* de *hardware* ou o *RTC*). Esta biblioteca é liberada sob licença *Apache* 2.0, que permite sua utilização em projetos comerciais [mbed, 2014].

As únicas partes dependentes de plataforma dos códigos de teste são a impressão de valores no terminal e a medição de tempo. Para evitar que fosse necessário fazer dois códigos de teste, fez-se uma camada de abstração com cinco funções: iniciar medição de tempo, parar a medição de tempo, obter a medição de tempo, imprimir um texto no *console* e ler um carácter do *console*. Na plataforma *x86*, estas funções foram implementadas como chamadas diretas à biblioteca padrão de *C* ou à *API* do *Windows*, enquanto em plataforma *Cortex-M3*, foram implementadas como chamadas à biblioteca da *mbed*. Estas funções estão apresentadas na tabela 5.1.

Durante os testes, houve um problema no mecanismo de medição de tempo na placa da *mbed*. A biblioteca da *mbed* fornece uma classe denominada `Timer`, que permite fazer medições de tempo em microssegundos com o auxílio de um *timer* de *hardware*. Objetos desta classe conseguem fazer medições de até 30 minutos, devido à precisão temporal (microssegundos) e do tipo de dados utilizado (inteiros de 32 bits não sinalizados). Inicialmente utilizou-

[2] *Throughput* em inglês.

se um objeto desta classe para implementar `IMPL_START_TIMER`, `IMPL_END_TIMER` e `IMPL_READ_TIMER`. Isto mostrou-se problemático, pois quando foram realizados os testes de desempenho, houve testes que duraram mais de 30 minutos. Para solucionar este problema, foi feito outro esquema de medição de tempo. Na biblioteca de desenvolvimento da *mbed* há uma classe denominada `Ticker` que permite disparar um código repetidamente com o auxílio de *timers* de *hardware* [mbed, 2014]. Usando este código, fez-se um contador de tempo em milissegundos, cujas leituras serviram como base para implementar as funções de medição de tempo. Este contador causou um *overhead* pequeno na execução, pois a cada milissegundo de execução, o processador é interrompido para incrementar o contador de tempo.

No caso da implementação para *PC*, o programa de testes foi executado em modo de segurança, em que apenas *drivers* e serviços essenciais do sistema operacional são executados [Microsoft, 2016a], para diminuir a interferência de outros processos nas medições. O processo que executou testes foi executado com prioridade máxima e afinidade com um único processador. A medição de tempo foi feita através da função `QueryPerformanceCounter` da *API* do *Windows*, cuja finalidade é medir intervalos de tempo com grande precisão [Microsoft, 2016c]. Para diminuir o efeito de falhas de *cache* no tempo de execução, antes de realizar os testes de desempenho, as operações criptográficas são realizadas ao menos uma vez sobre os mesmos dados, de forma que os dados devem estar na memória *cache* quando os testes forem iniciados. Em plataforma *Windows*, adicionou-se um pequeno tempo de *sleep* antes de realizar cada teste para garantir que a medição dos tempos tenha início no começo de um *quantum*.

O código para *Cortex-M3* foi compilado com a *IDE* online da *mbed*, usando o compilador *ARMCC*. Esta plataforma de compilação foi escolhida pois é a sugerida pela *mbed* para uso com a placa de prototipação e pode ser utilizado para construir aplicações comerciais [mbed, 2013]. O código em *x86* foi compilado utilizando o *MSVC* na *IDE Visual Studio 2013 Community*, que é gratuita para pesquisas acadêmicas [Microsoft, 2013].

Em ambos os casos, desabilitaram-se as otimizações de código por parte do compilador. Isto foi feito pois compiladores distintos fazem otimizações diferentes, que podem afetar os resultados obtidos. Em outros termos, a avaliação de desempenho poderia ser afetada pela qualidade do otimizador do compilador, se otimizações estivessem habilitadas.

Para o módulo de *AES*, foram testadas as operações de cifragem e decifragem de um bloco de dados usando os modos de operação *ECB* e *CBC*. O tamanho dos dados fornecidos para cifragem foi de 1024 *bytes*, valor este que foi escolhido arbitrariamente. Para o módulo de *RSA*, foram testadas as operações de cifragem e decifragem usando preenchimentos *PKCS*#1 v1.5 e *OAEP* com *SHA*-1. As chaves *RSA* utilizadas têm 2048 *bits* e os dados fornecidos têm o maior tamanho permitido, o que também foi escolhido de forma arbitrária. Para o módulo de *SHA*-256, foi testada a operação de geração de resumo criptográfico. Os dados fornecidos para este algoritmo têm 1000 *bytes*.

O número de repetições de cada operação criptográfica foi diferente no *PC* e no sistema embarcado. Isto foi feito pois o sistema embarcado processa dados a uma velocidade muito inferior ao *PC*. Se o número de repetições fossem iguais em ambas as plataformas, ou os testes demorariam muito no sistema embarcado ou a medidas no *PC* teriam poucos dígitos significativos. O total de repetições foi de 1000 no sistema embarcado no e 50000 no *PC*. Com estes números, uma batelada de testes (uma medição de cada operação criptográfica, com e sem a *GEmSysC*) levou em torno de 3 horas e 45 minutos no sistema embarcado e 35 minutos no *PC*. Cada batelada de testes foi repetida três vezes.

Tabela 5.2: Medidas de desempenho da *wolfSSL* e da *GemSysC* em plataforma *Cortex-M3*.

	wolfSSL		*GemSysC*		*Overhead*
	$\bar{t}_1$	σ_1	$\bar{t}_2$	σ_2	$\frac{\bar{t}_2-\bar{t}_1}{\bar{t}_1}$
Cifragem *AES* com *ECB*	1,478 s	0 ms	1,478 s	0 ms	>0,00%
Decifragem *AES* com *ECB*	1,495 s	0 ms	1,495 s	0 ms	>0,00%
Cifragem *AES* com *CBC*	1,515 s	0 ms	1,517 s	0 ms	0,13%
Decifragem *AES* com *CBC*	1,536 s	0 ms	1,538 s	0 ms	0,13%
Cifragem *RSA* com *PKCS* #1 v1.5	250,424 s	0 ms	250,426 s	0 ms	>0,00%
Decifragem *RSA* com *PKCS* #1 v1.5	3057,225 s	0 ms	3057,227 s	0 ms	>0,00%
Cifragem *RSA* com *OAEP*	251,337 s	0 ms	251,339 s	0 ms	>0,00%
Decifragem *RSA* com *OAEP*	3058,734 s	0 ms	3058,736 s	0 ms	>0,00%
Resumo com *SHA-256*	0,595 s	0 ms	0,596 s	0 ms	0,17%

Tabela 5.3: Medidas de desempenho da *OpenSSL* e da *GemSysC* em plataforma *x86*.

	OpenSSL		*GemSysC*		*Overhead*
	$\bar{t}_3$	σ_3	$\bar{t}_4$	σ_4	$\frac{\bar{t}_4-\bar{t}_3}{\bar{t}_3}$
Cifragem *AES* com *ECB*	0,340 s	0 ms	0,342 s	0 ms	0,59%
Decifragem *AES* com *ECB*	0,422 s	1 ms	0,423 s	0 ms	0,32%
Cifragem *AES* com *CBC*	0,315 s	0 ms	0,317 s	1 ms	0,53%
Decifragem *AES* com *CBC*	0,402 s	0 ms	0,403 s	0 ms	0,25%
Cifragem *RSA* com *PKCS* #1 v1.5	12,461 s	1 ms	12,467 s	2 ms	0,05%
Decifragem *RSA* com *PKCS* #1 v1.5	482,458 s	38 ms	482,851 s	37 ms	0,08%
Cifragem *RSA* com *OAEP*	12,805 s	1 ms	12,808 s	2 ms	0,03%
Decifragem *RSA* com *OAEP*	483,047 s	163 ms	483,379 s	150 ms	0,07%
Resumo com *SHA-256*	0,285 s	1 ms	0,289 s	1 ms	1,40%

Os resultados detalhados dos testes estão apresentados no apêndice D. Os resultados dos testes comparativos entre a *wolfSSL* e a *GemSysC* em um processador *Cortex-M3* estão resumidos na tabela 5.2. Os resultados dos testes comparativos entre a *OpenSSL* e a *GemSysC* em um computador convencional estão resumidos na tabela 5.3. Nestas tabelas, $\bar{t}_1$, $\bar{t}_2$, $\bar{t}_3$ e $\bar{t}_4$ representam as médias de tempo dos testes com *wolfSSL*, *GEmSysC* sobre *wolfSSL*, *OpenSSL* e *GEmSysC* sobre *OpenSSL*, respectivamente, e σ representa o desvio padrão.

Os resultados de desempenho no *PC* demonstraram desvios padrão muito grandes comparados com os obtidos no dispositivo embarcado. Apesar de o programa que realizou os testes ter sido executado com prioridade de tempo real no *PC*, seu tempo de execução foi afetado por outros processos executando em paralelo. Fatores como perdas de cache e trocas de página podem ter afetado estes resultados.

Com base nos resultados apresentados no apêndice D, observa-se que a *GemSysC* sobre *OpenSSL*, em comparação com a *OpenSSL* por si só, tem um *overhead* (acréscimo no tempo de consumo) entre 0,03% a 1,40%, de acordo com a funcionalidade testada. Por outro lado, a *GemSysC* sobre *wolfSSL* tem um *overhead* entre pouco mais de 0% e 0,17%. Enquanto em ambos os testes os valores de *overhead* são baixos, nota-se que são maiores no caso da *OpenSSL*.

Ambas as implementações da *GemSysC* foram realizadas com o objetivo de gerar em um código bem documentado e simples, sem foco em otimização. Com esse fato em mente,

Tabela 5.4: Memória de código e dados constantes ocupada pela *GEmSysC* sobre *wolfSSL*.

	Código	Dados constantes	Total
Core	18 B	0 B	18 B
Módulo de *AES*	384 B	16 B	400 B
Módulo de *RSA*	596 B	48 B	644 B
Módulo de *SHA-256*	130 B	8 B	138 B

Tabela 5.5: Memória de código e dados constantes ocupada pela *GEmSysC* sobre *OpenSSL*.

	Código	Dados constantes	Total
Core	70 B	0 B	70 B
Módulo de *AES*	944 B	16 B	960 B
Módulo de *RSA*	1440 B	88 B	1528 B
Módulo de *SHA-256*	432 B	8 B	440 B

considera-se que seria possível implementar a *GemSysC* sobre a *OpenSSL* de forma mais eficiente que a realizada.

5.2.3 Memória

As duas implementações da *GEmSysC* também foram avaliadas em função da quantidade de memória que ocupam. Para tal, fizeram-se duas avaliações diferentes. Primeiramente, avaliou-se quanta memória a própria *GEmSysC* ocupa, isto é, quanto cada um de seus módulos representa em termos de memória de programa e dados. Em seguida, avaliou-se quanta memória as instâncias de algoritmos criptográfico ocupam a mais em relação ao que ocupam as mesmas estruturas da biblioteca subjacente por si só.

Para avaliar o quanto a própria *GEmSysC* ocupa em termos de memória, usaram-se os arquivos de mapa de memória produzidos pelos *linkers*. Como a *IDE online* da *mbded* não permite acesso a este arquivo, foi necessário compilar o código para *Cortex-M3* utilizando outro compilador. Utilizou-se para esta finalidade o *GCC* distribuído como parte das *GNU Tools ARM Embedded* [ARM, 2016]. Este compilador foi escolhido por ser gratuito, de código aberto e bem documentado. Além disto, a *IDE online* da *mbed* exporta o projeto completo para uso com este compilador, tornando o trabalho muito mais simples. Como explicado na seção 5.2.2, os códigos foram compilados sem otimizações.

Para medir quanto o core da *GEmSysC* e cada um de seus módulos ocupa, somou-se o tamanho dos símbolos exportados presentes no arquivo de mapa de memória. A implementação sobre a *OpenSSL* foi compilada utilizando o compilador *MSVC* com *Visual Studio* 2013. No arquivo de mapa de memória gerado por ele, observou-se que parte dos símbolos ficaram na seção `.text` e outra parte ficou na seção `.rdata`. Segundo a especificação em [Microsoft, 2015b], os símbolos em `.text` representam código executável, enquanto os símbolos em `.rdata` são dados constantes. Na implementação sobre *wolfSSL*, compilada pelo *GCC*, observou-se que parte dos símbolos ficaram na seção `.text`, enquanto outra parte ficou na seção `.rodata`; Segundo a especificação em [SCO, 2013], os símbolos em `.text` representam código executável, enquanto os símbolos em `.rodata` são dados constantes. Os tamanhos de cada módulo em termos de código e dados constantes estão apresentados nas tabelas 5.4 e 5.5.

Tabela 5.6: Estruturas que representam instâncias de algoritmos criptográficos na *wolfSSL*, na *OpenSSL* e nas duas implementações da *GEmSysC*.

	OpenSSL	*wolfSSL*	*GEmSysC*
AES	`EVP_CIPHER_CTX *`	`Aes`	`clAESDef_t` e `clAESData_t`
RSA	`RSA *`	`RsaKey` e `RNG` (na cifragem)	`clRSADef_t` e `clRSAData_t`
SHA-256	`EVP_MD_CTX *`	`Sha256`	`clSHA256Def_t` e `clSHA256Data_t`

Tabela 5.7: Memória ocupada por instâncias de algoritmo criptográfico da *wolfSSL* e da *GEmSysC* sobre *wolfSSL*.

	wolfSSL	*GEmSysC*	Diferença
Módulo de *AES*	276 B	296 B	20 B
Módulo de *RSA*	136 B a 148 B	164 B	16 a 28 B
Módulo de *SHA-256*	108 B	120 B	12 B

Para medir quanto cada instância de algoritmo criptográfico da *GEmSysC* ocupa a mais que as mesmas estruturas da biblioteca por si só, usaram-se medições com o operador `sizeof` da linguagem *C*. Mediram-se os tamanhos em *bytes* das estruturas que implementam as instâncias de algoritmos criptográficos na *GEmSysC*, e as estruturas equivalentes sem a *GEmSysC*. As estruturas que representam cada instância de algoritmo criptográfico na *wolfSSL*, na *OpenSSL* e nas duas implementações da *GEmSysC* estão apresentadas na tabela 5.6.

As medidas de quanto cada instância ocupa em cada implementação estão apresentadas nas tabelas 5.7 e 5.8. No caso da *wolfSSL*, as estruturas indicadas (`Aes`, `RsaKey`, `RNG` e `Sha256`) utilizam alocação estática, armazenando localmente os dados do algoritmo criptográfico. No caso da *OpenSSL*, por outro lado, utiliza-se alocação dinâmica e as estruturas utilizadas são apenas ponteiros para estruturas alocadas dinamicamente. No caso da *OpenSSL*, não é possível saber quanta memória o algoritmo realmente utiliza, então não é possível saber quanto é o custo da *GEmSysC* proporcionalmente à biblioteca.

5.2.4 Imprecisões

Nesta seção, estão discutidas imprecisões das avaliações realizadas. Na avaliação da *GEmSysC* quanto à sua simplicidade, foram feitos códigos de exemplo que realizam cada uma das operações suportadas, e então foram calculadas métricas a partir destes códigos. Buscou-se fazer, com cada uma das *APIs*, o código mais simples possível para a mesma finalidade. Os códigos de exemplo utilizados afetam diretamente os resultados, e os resultados poderiam ser diferentes se estes códigos fossem feitos de forma diferente da apresentada. As métricas utilizadas na avaliação de simplicidade são heurísticas. Se heurísticas diferentes fossem utilizadas, resultados diferentes poderiam ter sido obtidos.

Neste capítulo, foram apresentadas duas implementações da *GemSysC*. Estas implementações foram feitas com dois objetivos: o primeiro foi demonstrar que a especificação da *GemSysC* é suficientemente genérica para que possa ser implementada sobre outras bibliotecas criptográficas. O outro objetivo das implementações foi quantificar o *overhead* desta camada de abstração, tanto em relação ao desempenho quanto em relação ao gasto de memória. Estas avaliações foram feitas com implementações de exemplo das bibliotecas, nas quais não se focou em eficiência, mas sim em simplicidade. Assim, acredita-se que seja possível fazer implementações mais eficientes que as propostas.

Tabela 5.8: Memória ocupada por instâncias de algoritmo criptográfico da *OpenSSL* e da *GEmSysC* sobre *OpenSSL*.

	OpenSSL	*GEmSysC*	Diferença
Módulo de *AES*	4 B	24 B	20 B
Módulo de *RSA*	4 B	20 B	16 B
Módulo de *SHA-256*	4 B	16 B	12 B

Nos testes de desempenho, foi medido o tempo que diversas operações criptográficas demoram com e sem a camada *GEmSysC*. Estas avaliações foram feitas em plataformas *x86* e *Cortex-M3*, com duas implementações da *GEmSysC*, sobre *OpenSSL* e sobre *wolfSSL* respectivamente. A metodologia de medição adotada avaliou apenas o tempo de processamento de dados, sem considerar o tempo de instanciação e liberação de recursos das instâncias de algoritmos criptográficos. Assim, os resultados destes testes são de pouca utilidade nos casos em que as instâncias de algoritmos criptográficos forem criadas e destruídas com frequência, pois os tempos que estas operações demoram para executar não foram avaliados. As implementações da *GEmSysC* não foram feitas com o objetivo de torná-las mais eficientes, mas sim com o objetivo de torná-las mais simples. A eficiência das implementações afetou diretamente os resultados dos testes. Então, se fossem feitas implementações mais eficientes, resultados melhores poderiam ser obtidos.

As avaliações dos módulos e do *core* da *GEmSysC* com relação à quantidade de memória que utilizam também foram fortemente afetados pelas implementações das camadas de abstração, pois implementações mais eficientes poderiam trazer resultados melhores. Devido à forma utilizada pela *OpenSSL* para alocar estruturas de dados, não foi possível avaliar quanto a *GEmSysC* implementada sobre esta biblioteca representa de *overhead* de forma proporcional, só foi possível avaliar este fator em termos absolutos.

5.3 Análise

Quanto à comparação de complexidade, observou-se que a *GemSysC* é mais simples que cada uma das bibliotecas se observada como um todo, já que suas métricas de complexidades foram geralmente menores. Como discutido no final da seção 5.2.1, isto se deve à sua *API* ser mais consistente entre algoritmos criptográficos. Por outro lado, se for observado cada um dos algoritmos individualmente, a *GemSysC* é levemente mais complexa que as outras bibliotecas pois, neste caso, as métricas referentes a ela são maiores que as das outras bibliotecas.

Quanto à comparação de desempenho, notou-se que o *overhead* de tempo de processamento da implementação sobre *wolfSSL* está entre perto de 0% e 0,17%, enquanto, o *overhead* da implementação sobre *OpenSSL* está entre 0,03% e 1,40%. Não se considerou adequado fazer uma análise mais aprofundada destes resultados, pois a significância das medições para o desempenho da aplicação final pode variar entre uma aplicação e outra.

Com relação aos testes de gasto de memória, observou-se que, em ambas as implementações, o custo de utilizar as definições de algoritmo criptográfico variou entre 12 e 28 *bytes*, o que corresponde a entre 3 e 7 inteiros de 32 *bits*. Já com relação ao tamanho dos trechos de código, notou-se que a implementação sobre a *wolfSSL* é menor (isto é, ocupa menos *bytes*) para o *core* e para todos os módulos. Com estas medidas e com as medidas de *overhead* de tempo da *GEmSysC*, deve ser possível aferir a viabilidade de utilizar a *GEmSysC* em outros projetos de *software*.

Capítulo 6

Conclusão

Neste documento, foi apresentada uma *API* para acesso a funcionalidades criptográficas voltada a aplicações embarcadas. Esta *API* permite que funcionalidades criptográficas sejam acessadas por aplicações de alto nível sem atrelá-las a uma biblioteca criptográfica em particular. O uso de uma única *API*, independente da biblioteca, permite maior reaproveitamento de código entre projetos e simplifica o processo de aprendizado. A principal inspiração para a elaboração deste trabalho foi o padrão *CMSIS-RTOS*, que abstrai as funcionalidades de um sistema operacional de tempo real através de uma *API* única e independente da implementação.

Para construir a *API* criptográfica, foi feita uma pesquisa sobre segurança em sistemas embarcados na qual ficou evidente que criptografia é uma ferramenta necessária para garantir segurança na maioria dos sistemas, mas muitas vezes ela não é suficiente. Foi feita também uma pesquisa sobre boas práticas de construção de *APIs* e métodos para avaliar e comparar *APIs*. Buscaram-se finalmente exemplos de *APIs* criptográficas (como é o caso de *Cryptoki*, *OpenSSL* e *wolfSSL*) ou para outras finalidades mais voltadas a sistemas embarcados (como é o caso do *CMSIS-RTOS*). O principal objetivo foi conhecer as formas convencionais com as quais as funcionalidades criptográficas são expostas às aplicações. A procura por outras *APIs* voltadas a sistemas embarcados teve como meta reconhecer convenções de forma e estilo adotadas em *software* embarcado. Com base na pesquisa realizada, foi possível tomar decisões embasadas no projeto da *API* criptográfica.

Reduziu-se um conjunto mínimo de funcionalidades que uma *API* criptográfica deve ter, a saber: cifragem e decifragem de dados por criptografia simétrica, cifragem e decifragem de dados por criptografia assimétrica e geração de resumo criptográfico. Com esta lista reduzida em mente, projetou-se a "API Criptográfica Genérica para Sistemas Embarcados" (em inglês, *"Generic Embedded Systems Cryptographic API"* ou *GEmSysC*).

A *GEmSysC* é somente a especificação de uma *API*, que pode ser construída como camada de abstração sobre outras bibliotecas existentes. Diferentes bibliotecas suportam diferentes algoritmos criptográficos e podem passar a suportar mais algoritmos com o tempo. Por esse motivo, escolheu-se adotar uma estrutura modular para a *GEmSysC*. Cada algoritmo criptográfico suportado pela biblioteca e pela *GEmSysC* é denominado "mecanismo criptográfico". Para cada mecanismo, a *GEmSysC* apresenta um módulo distinto. Assim, é possível implementar sobre cada biblioteca criptográfica apenas os módulos da *GEmSysC* correspondentes aos mecanismos criptográficos presentes na biblioteca. Além dos módulos, a *API* tem um *core* independente do mecanismo, o que inclui funções, constantes e tipos de dados.

A *GEmSysC* foi elaborada em linguagem *C*, que não é orientada a objetos. Ainda assim, a forma com que se desenvolveu a *API* pode ser entendida como orientada a objetos. Nesta acepção, todos os mecanismos criptográficos da *GEmSysC* são classes e estendem uma classe que representa um mecanismo criptográfico genérico.

Para a prova de conceito da *GEmSysC*, foi feito o projeto do *core* e de três módulos. Cada um dos módulos escolhidos é um exemplar de uma das funcionalidades mínimas necessárias a uma *API* criptográfica: criptografia simétrica, criptografia assimétrica e resumo criptográfico. Escolheu-se como exemplar da criptografia simétrica o *AES*, que é amplamente utilizado e é mantido pelo *NIST*, um órgão do governo dos Estados Unidos da América. Como modelo da criptografia assimétrica, escolheu-se o *RSA*. que também é amplamente utilizado e é indicado pelo *NIST* para, entre outros usos, realizar trocas de chaves. Como padrão do resumo criptográfico, escolheu-se o *SHA-256*, um algoritmo da família *SHA-2*. A família *SHA-2* foi proposta pelo *NIST*, mas já foi suplantada pela família *SHA-3*. Ainda assim, a família *SHA-3* não é muito utilizada nem é suportada por diversas bibliotecas criptográficas. Por isso, escolheu-se especificar o mecanismo relativo a um algoritmo da família *SHA-2*.

Apesar de a *GEmSysC* ter sido elaborada com foco em sistemas embarcados, nada impede que seja utilizada também em computadores convencionais. Na verdade, isso traria o benefício de se utilizar a mesma sintaxe no código embarcado e no *software* da estação-base. Para validar a prova de conceito da *GEmSysC*, foram feitas duas implementações, uma sobre a biblioteca *OpenSSL* e outra sobre a biblioteca *wolfSSL*. A *wolfSSL* é voltada para sistemas embarcados, então a implementação sobre ela foi utilizada para avaliar o desempenho da *GEmSysC* sobre um sistema embarcado. A *OpenSSL* é mais utilizada em computadores pessoais, então a implementação sobre esta biblioteca foi utilizada para avaliar o desempenho da *GEmSysC* em um computador pessoal. A *GEmSysC* foi avaliada quanto às características de sua *API* em comparação com as características das *APIs* criptográficas utilizadas (*OpenSSL* e *wolfSSL*).

Na etapa de avaliação, observou-se que a *GEmSysC* é levemente mais complexa (em termos de uso) que cada uma das bibliotecas consideradas, se for considerada cada operação criptográfica individualmente. Isto foi constatado pela avaliação das métricas apresentadas na seção 5.2.2. Por outro lado, evidenciou-se também que a *GEmSysC* é mais simples, se observada como um todo, pois tem uma *API* mais consistente entre mecanismos criptográficos que as duas bibliotecas consideradas. Isto é evidenciado pelo fato de haver um único tipo de dados para os identificadores de instâncias de mecanismos criptográficos, uma única função para processamento de dados e uma única função para liberação de recursos computacionais.

Nos testes comparativos de desempenho computacional entre as bibliotecas por si só e a *GEmSysC* construída sobre elas, observou-se que a *GEmSysC* representa um *overhead* um pouco maior quando construída sobre a *OpenSSL* (entre 0,03% e 1,40%), enquanto sobre a *wolfSSL*, o *overhead* ficou um pouco menor (entre quase 0% e 0,17%). Com relação ao gasto de memória, avaliou-se o tanto de memória constante e de memória variável que cada uma das implementações da *GEmSysC* traz à aplicação. Na implementação sobre *Cortex-M3* para *PC*, os módulos da *GEmSysC* combinados ocupam aproximadamente 3 kB, enquanto na implementação sobre *wolfSSL* para o sistema embarcado, eles ocupam aproximadamente 1,2 kB. Com relação às estruturas de dados variáveis, a *GEmSysC* sobre *OpenSSL* necessita de 12 e 20 *bytes* adicionais por algoritmo criptográfico (além dos já utilizados na biblioteca), enquanto a *GEmSysC* sobre *wolfSSL* necessita de 12 a 28 *bytes*.

6.1 Trabalhos futuros

A *GEmSysC* tem apenas a especificação de três de seus módulos (*AES*, *RSA* e *SHA-256*). Trabalhos futuros poderiam envolver a especificação de mais módulos, como por exemplo *DES*, outros algoritmos da família *SHA-2* e da família *SHA-1*, entre outros. Para que esses trabalhos sejam possíveis, será preciso estudar cada um destes algoritmos e a forma com que são apresentados em outras bibliotecas criptográficas. Também sugere-se para trabalhos futuros que se revisem as especificações realizadas. Por exemplo, a especificação do módulo de *RSA* não inclui mecanismos para utilizar outras formas de preenchimento *OAEP*, o que poderia ser previsto em uma versão revisada deste módulo.

A lista reduzida de operações criptográficas da *GEmSysC* não inclui operações relacionadas a códigos de autenticação de mensagem nem a assinaturas digitais, pois as formas mais populares destas operações podem ser construídas como composição de cifragem, decifragem e resumo criptográfico. Assim, a *GEmSysC* não tem suporte a mecanismos que não podem ser construídos inteiramente como composição de outras operações. Assim, para que a abrangência da *GEmSysC* seja maior, sugere-se adicionar à lista de tipos de operações outras quatro: geração e verificação de códigos de autenticação de mensagem e geração e verificação de assinaturas digitais. Além de prever a possibilidade de utilizar estes tipos de operação, sugere-se especificar pelo menos três módulos associados a assinaturas digitais: *DSA*, assinatura com *RSA* e *ECDSA*, pois estes são os algoritmos sugeridos pelo *NIST* [NIST, 2013].

Foi feita uma especificação da *GEmSysC* em linguagem *C* voltada para ambientes embarcados. Se uma *API* semelhante pudesse ser utilizada em outros universos, como por exemplo máquinas virtuais *Java* e ambientes como o *node.js*, as vantagens da *API* unificada ficariam ainda mais evidentes. Por exemplo, o código utilizado em um servidor feito em *Java* ou *Java Script* poderia ser semelhante ao utilizado em um *firmware* desenvolvido em linguagem *C*. Para tal, primeiramente seria necessário especificar uma variante da *GEmSysC* em outra linguagem de programação. O maior desafio deste trabalho seria manter uma consistência tanto com a *GEmSysC* quanto com as convenções da linguagem utilizada, sem comprometer a eficiência nem a facilidade de uso.

Foram feitas duas implementações da *GEmSysC* sobre duas bibliotecas (*OpenSSL* e *wolfSSL*). Seria ideal fazer estudos de outras formas de implementar e buscar a implementação mais eficiente possível para a camada de abstração, especialmente no caso da *OpenSSL*. Também seria interessante se houvesse implementações sobre outras bibliotecas criptográficas.

Outro trabalho futuro seria criar outras bibliotecas análogas à *GEmSysC* e ao *CMSIS-RTOS* para unificar o acesso a outras funcionalidades. Por exemplo, poderia haver uma *API* unificada para certos tipos de processamento e filtragem de sinais, para conversão entre codificações de dados, entre outros.

Referências Bibliográficas

[Amar et al., 2008] Amar, A., Joshi, S., and Wallwork, D. (2008). Generic Driver Model using hardware abstraction and standard APIs. Design & Reuse, URL: http://www.design-reuse.com/articles/18584/generic-driver-model.html.

[Arcieri, 2012] Arcieri, T. (2012). All the crypto code you've ever written is probably broken. Tony Arcieri's Web Site, URL: https://tonyarcieri.com/all-the-crypto-code-youve-ever-written-is-probably-broken.

[ARM, 2016] ARM (2016). GNU ARM Embedded Toolchain. ARM Developer. URL: https://developer.arm.com/open-source/gnu-toolchain/gnu-rm.

[Atzori et al., 2010] Atzori, L., Iera, A., and Morabito, G. (2010). The Internet of Things: A Survey. *Computer Networks*, 54(15):2787–2805.

[Bakker, 2014] Bakker, P. (2014). ASN.1 key structures in DER and PEM. ARM mbed TLS, Knowledge Base, URL: https://tls.mbed.org/kb/cryptography/asn1-key-structures-in-der-and-pem.

[Bammi, 2006] Bammi, S. (2006). Patterns for Designing a Generic Device Driver for Interrupt Driven I/O. In *13th Conference on Pattern Languages of Programs*, PLoP '06.

[Bancroft et al., 2012] Bancroft, J. B., Garrett, D., and Lachapelle, G. (2012). Activity and Environment Classification using Foot Mounted Navigation Sensors. In *2012 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 1–10.

[Bandi et al., 2003] Bandi, R. K., Vaishnavi, V. K., and Turk, D. E. (2003). Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics. *IEEE Transactions on Software Engineering*, 29(1):77–87.

[Barker et al., 2014] Barker, E., Chen, L., and Moody, D. (2014). *Recommendation for Pair-wise Key Establishment Schemes: Using Integer Factorization Cryptography*. National Institute of Standards and Technology. NIST Special Publication 800-56B – Revision 1.

[Barry and Hartnett, 2007] Barry, P. and Hartnett, G. (2007). Tricks and techniques for performance tuning your embedded system using patterns: Part 3. Embedded, UBM Electronics, URL: http://www.embedded.com/design/mcus-processors-and-socs/4007110/Tricks-and-techniques-for-performance-tuning-your-embedded-system-using-patterns-Part-3.

[Barry, 2009] Barry, R. (2009). *Using the FreeRTOS Real Time Kernel: A Practical Guide*. Real Time Engineers Ltd.

[Beningo, 2012] Beningo, J. (2012). Developing reusable device drivers for MCUs. EDN Network, UBM Electronics, URL: `http://www.edn.com/design/systems-design/4402641/Developing-reusable-device-drivers-for-MCUs`.

[Bernstein et al., 2012] Bernstein, D. J., Lange, T., and Schwabe, P. (2012). The Security Impact of a New Cryptographic Library. In Hevia, A. and Neven, G., editors, *Progress in Cryptology – LATINCRYPT 2012*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer Berlin Heidelberg.

[Bertagna, 2010] Bertagna, P. (2010). How does a GPS tracking system work? EETimes, `http://www.eetimes.com/document.asp?doc_id=1278363`.

[Blanchette, 2008] Blanchette, J. (2008). *The Little Manual of API Design*. Trolltech, a Nokia company. URL: `http://chaos.troll.no/\~{}shausman/api-design/api-design.pdf`.

[Bleichenbacher et al., 1998] Bleichenbacher, D., Kaliski, B., and Staddon, J. (1998). Recent Results on PKCS #1: RSA Encryption Standard. RSA Laboratories' Bulletin. Number 7.

[Bonato, 2003] Bonato, P. (2003). Wearable Sensors/Systems and Their Impact on Biomedical Engineering. *Engineering in Medicine and Biology Magazine, IEEE*, 22(3):18–20.

[Bos et al., 2013] Bos, J., Halderman, A., Heninger, N., Moore, J., Naehrig, M., and Wustrow, E. (2013). Elliptic Curve Cryptography in Practice. Technical report, Microsoft.

[Burleson and Carrara, 2013] Burleson, W. and Carrara, S. (2013). *Security and Privacy for Implantable Medical Devices*. Springer New York.

[Cheng et al., 2003] Cheng, B. H. C., Konrad, S., Campbell, L. A., and Wassermann, R. (2003). Using Security Patterns to Model and Analyze Security Requirements. In *Proceedings of the 2nd International Workshop on Requirements Engineering for High Assurance Systems 2013 (RHAS'03)*, pages 13–22. The 2nd International Workshop on Requirements Engineering for High Assurance Systems was part of the 11th IEEE International Requirements Engineering Conference (RE'03).

[ChibiOS, 2015a] ChibiOS (2015a). *ChibiOS/HAL Reference Manual*. ChibiOS.

[ChibiOS, 2015b] ChibiOS (2015b). *ChibiOS/RT Reference Manual*. ChibiOS.

[Clulow, 2015] Clulow, J. (2015). The Design and Analysis of Cryptographic Application Programming Interfaces for Security Devices. Master's thesis, University of Natal, Durban, South Africa.

[CMSIS, 2012] CMSIS (2012). *CMSIS Documentation*. ARM Ltd. Version 3.01.

[Constantinescu and Vladoiu, 2013] Constantinescu, Z. and Vladoiu, M. (2013). Challenges in Safety, Security, and Privacy in the Development of Vehicle Tracking Systems. In *System Theory, Control and Computing (ICSTCC), 2013 17th International Conference*, pages 607–612.

[Daemen and Rijmen, 1999] Daemen, J. and Rijmen, V. (1999). AES Proposal: Rijndael. National Institute of Standards and Technology.

[Daemen and Rijmen, 2002] Daemen, J. and Rijmen, V. (2002). *The Design of Rijndael*. Springer-Verlag Berlin Heidelberg.

[Danielsson and Makipaa, 2012] Danielsson, P. and Makipaa, L. (2012). A Systematic Literature Review of Electronic Monitoring of Offenders. Literature review, National Research Institute of Legal Policy. Research Communications No. 114.

[de Souza and Bentolila, 2009] de Souza, C. R. B. and Bentolila, D. L. M. (2009). Automatic Evaluation of API Usability Using Complexity Metrics and Visualizations. In *31st International Conference on Software Engineering - Companion Volume, 2009. ICSE-Companion 2009*, pages 299–302.

[Di Sirio, 2015] Di Sirio, G. (2015). ChibiOS/RT. `http://www.chibios.org/dokuwiki/doku.php?id=chibios:product:rt:start`.

[Doucette, 2008] Doucette, A. (2008). On API usability: An analysis and an evaluation tool. Technical report, University of Saskatchewan, Saskatoon, Saskatchewan, Canada.

[Doxygen, 2015] Doxygen (2015). Doxygen. Generated on Wed Dec 30 2015 11:12:50. URL: `http://www.stack.nl/~dimitri/doxygen/index.html`.

[Dubey et al., 2010] Dubey, A., Karsai, G., Kereskenyi, R., and Mahadevan, N. (2010). A real-time component framework: experience with CCM and ARINC-653. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 143–150. IEEE.

[Emmerich, 2000] Emmerich, W. (2000). Software Engineering and Middleware: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 117–129, New York, NY, USA. ACM.

[Farkhani and Razzazi, 2006] Farkhani, T. R. and Razzazi, M. R. (2006). Examination and Classification of Security Requirements of Software Systems. In *Proceedings of the 2nd International Conference on Information & Communication Technologies: From Theory to Applications 2006 (ICTTA'06)*, volume 2, pages 2778–2783. IEEE.

[Finseth, 2004] Finseth, S. (2004). Abstracting device-driver development. Embedded, UBM Electronics, URL: `http://www.embedded.com/design/configurable-systems/4024952/Abstracting-device-driver-development`.

[Firesmith, 2003] Firesmith, D. G. (2003). Analyzing and Specifying Reusable Security Requirements. In *Proceedings of the 2nd International Workshop on Requirements Engineering for High Assurance Systems 2013 (RHAS'03)*, pages 7–12. The 2nd International Workshop on Requirements Engineering for High Assurance Systems was part of the 11th IEEE International Requirements Engineering Conference (RE'03).

[Fowler, 2014] Fowler, M. (2014). APIs should not be copyrightable. Martin Fowler's Web Site, URL: `http://martinfowler.com/articles/copyrightapi.html`.

[FreeRTOS, 2016] FreeRTOS (2016). FreeRTOS API Reference.

[FreeRTOS, 2016] FreeRTOS (2016). FreeRTOS Official Site. FreeRTOS Official Site, URL: `http://www.freertos.org/index.html`.

[Freier et al., 2011] Freier, A. O., Karlton, P., and Kocher, P. C. (2011). The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), `http://www.ietf.org/rfc/rfc6101.txt`.

[Gollakota et al., 2011] Gollakota, S., Hassanieh, H., Ransford, B., Katabi, D., and Fu, K. (2011). They Can Hear Your Heartbeats: Non-invasive Security for Implantable Medical Devices. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 2–13, New York, NY, USA. ACM.

[Gradinaru, 2010] Gradinaru, B. (2010). Memory pools or best practice in dynamic allocation in embedded systems. Blog post, URL: `https://bogdangradinaru.wordpress.com/2010/02/17/memory-pools-or-best-practice-in-dynamic-allocation-in-embedded-systems/`.

[Graves, 2011] Graves, S. (2011). Justifiably taboo: Avoiding malloc()/free() APIs in military/aerospace embedded code. Military Embedded Systems, URL: `http://mil-embedded.com/articles/justifiably-apis-militaryaerospace-embedded-code/`.

[Griffin and Fenwick, 2015a] Griffin, R. and Fenwick, V. (2015a). PKCS #11 Cryptographic Token Interface Base Specification Version 2.40.

[Griffin and Fenwick, 2015b] Griffin, R. and Fenwick, V. (2015b). PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 2.40.

[Gruteser and Liu, 2004] Gruteser, M. and Liu, X. (2004). Protecting Privacy in Continuous Location-Tracking Applications. *Security Privacy, IEEE*, 2(2):28–34.

[Hatzivasilis et al., 2014] Hatzivasilis, G., Theodoridis, A., Gasparis, E., and Manifavas, C. (2014). ULCL - An Ultra-lightweight Cryptographic Library for Embedded Systems. In *PECCS 2014 - Proceedings of the 4th International Conference on Pervasive and Embedded Computing and Communication Systems*, pages 247–254. SciTePress.

[Henning, 2007] Henning, M. (2007). API Design Matters. *ACM Queue*, 5(4):24–36.

[Hoshen et al., 1995] Hoshen, J., Sennott, J., and Winkler, M. (1995). Keeping Tabs on Criminals. *Spectrum, IEEE*, 32(2):26–32.

[IEEE, 2013] IEEE (2013). *Standard for Information Technology Portable Operating System Interface (POSIX) Base Specifications, Issue 7*. The Open Group Base Specifications Issue 7. IEEE Std 1003.1, 2013 Edition. Incorporates IEEE Std 1003.1-2008 and IEEE Std 1003.1-2008/Cor 1-2013.

[ISO/IEC, 2011] ISO/IEC (2011). *ISO/IEC 9797-1:2011. Information technology – Security techniques – Message Authentication Codes (MACs) – Part 1: Mechanisms using a block cipher*, volume 1. ISO/IEC, 2 edition.

[ITI, 2007] ITI (2007). *Glossário ICP-Brasil*. Instituto Nacional de Tecnologia da Informação, 1.2 edition.

[Jindal and Singh, 2012] Jindal, P. and Singh, B. (2012). Study And Performance Evaluation Of Security-Throughput Tradeoff With Link Adaptive Encryption Scheme. *CoRR*, abs/1211.5080.

[Johnson, 1997] Johnson, R. E. (1997). Frameworks = Components + Patterns. *Communications of the ACM*, 40(10):39–42.

[Jyostna and Padmaja, 2011] Jyostna, K. and Padmaja, V. (2011). Secure Embedded System Networking: An Advanced Security Perspective. *International Journal of Engineering Science & Technology*, 3(5):3854–3862.

[Kak, 2016] Kak, A. (2016). Lecture 8: AES: The Advanced Encryption Standard. Lecture Notes on 'Computer and Network Security', Purdue University, URL: `https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture8.pdf`.

[Kalinsky, 1999] Kalinsky, D. (1999). Architecture of Device I/O Drivers. Lesson #202, The Embedded Systems Conference Europe 1999 (ESC Europe 1999, ESCE'99).

[Kermani et al., 2013] Kermani, M. M., Zhang, M., Raghunathan, A., and Jha, N. K. (2013). Emerging Frontiers in Embedded Security. In *26th International Conference on VLSI Design and 12th International Conference on Embedded Systems (VLSID)*, pages 203–208.

[Khyo, 2011] Khyo, G. A. (2011). Language Support for Linux Device Driver Programming. Trabalho de conclusão do curso de Engenharia de Software e Computação na Internet, Fakultät für Informatik der Technischen Universität Wien (Faculdade de Informática da Universidade Tecnológica de Viena).

[Klaus et al., 2014] Klaus, T., Franzmann, F., Engelhard, T., Scheler, F., and Schröder-Preikschat, W. (2014). Usable RTOS-APIs? In Brandenburg, B. B. and Kato, S., editors, *Proceedings of the 10th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2014)*, pages 61–66, Kaiserslautern, DE.

[Konstantas, 2007] Konstantas, D. (2007). An overview of wearable and implantable medical sensors. *IMIA Yearbook*, 2(1):66–69.

[Koopman, 2004] Koopman, P. (2004). Embedded System Security. *IEEE Computer*, 37(7):95–97.

[Lawson, 2009] Lawson, N. (2009). Side-Channel Attacks on Cryptographic Software. *IEEE Security & Privacy*, 7(6):65–68.

[Lee and Seshia, 2011] Lee, E. A. and Seshia, S. A. (2011). *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. Published by authors, 1st edition.

[Lin and Huang, 2007] Lin, S.-Y. and Huang, C.-T. (2007). A High-Throughput Low-Power AES Cipher for Network Applications. In *2007 Asia and South Pacific Design Automation Conference*, pages 595–600.

[Linn, 2000] Linn, J. (2000). Generic Security Service Application Program Interface Version 2, Update 1). RFC 2743 (Proposed Standard), http://www.ietf.org/rfc/rfc2743.txt.

[MacMillan, 2011] MacMillan, N. (2011). Lab Guide: Embedded Memory Management. Blog post, URL: http://nrqm.ca/mechatronics-lab-guide/lab-guide-embedded-memory-management/.

[Maziero, 2014] Maziero, C. A. (2014). Sistemas Operacionais: Conceitos e Mecanismos. Livro aberto. Acessível em: http://wiki.inf.ufpr.br/maziero/lib/exe/fetch.php?media=so:so-livro.pdf. Versão de 8 de agosto de 2014.

[mbed, 2013] mbed (2013). mbed Compiler. ARM mbed Handbook. URL: https://developer.mbed.org/handbook/mbed-Compiler?action=view&revision=15856.

[mbed, 2014] mbed (2014). mbed SDK. ARM mbed Handbook. URL: https://developer.mbed.org/handbook/mbed-SDK?action=view&revision=31773.

[mbed, 2016] mbed (2016). mbed LPC1768. ARM mbed. URL: https://developer.mbed.org/platforms/mbed-LPC1768/.

[Microsoft, 2013] Microsoft (2013). Microsoft Software License Terms – Microsoft Visual Studio Community 2013. URL: https://www.visualstudio.com/en-us/dn877550.aspx.

[Microsoft, 2015a] Microsoft (2015a). C Run-Time Library Reference. Visual Studio 2015. Microsoft Developer Network. URL: https://msdn.microsoft.com/en-us/library/59ey50w6.aspx.

[Microsoft, 2015b] Microsoft (2015b). Microsoft Portable Executable and Common Object File Format Specification – Revision 9.3.

[Microsoft, 2016a] Microsoft (2016a). Advanced startup options (including safe mode). Revision 22. Microsoft Support Center. URL: https://support.microsoft.com/en-us/help/17419/windows-7-advanced-startup-options-safe-mode#start-computer-safe-mode=windows-7.

[Microsoft, 2016b] Microsoft (2016b). What is a driver? Microsoft Hardware Dev Center. URL: https://msdn.microsoft.com/en-us/library/windows/hardware/ff554678(v=vs.85).aspx.

[Microsoft, 2016c] Microsoft (2016c). Windows API Index. Microsoft Windows Dev Center. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx.

[Mokbel, 2007] Mokbel, M. F. (2007). Privacy in Location-Based Services: State-of-the-Art and Research Directions. In *8th International Conference on Mobile Data Management (MDM 2007)*, pages 228–228.

[Netcraft, 2014] Netcraft (2014). Half a million widely trusted websites vulnerable to Heartbleed bug. Netcraft, URL: http://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html.

[NIST, 1997] NIST (1997). Announcing Development of a Federal Information Processing Standard for Advanced Encryption Standard. National Institute of Standards and Technology. In Federal Register of January 2, 1997, Vol. 62, No. 1, Pages 93–94.

[NIST, 2001] NIST (2001). *Federal Information Processing Standards (FIPS) 197: Advanced encryption standard (AES)*. National Institute of Standards and Technology (NIST).

[NIST, 2002] NIST (2002). Announcing Approval of Federal Information Processing Standard (FIPS) 180-2, Secure Hash Standard; a Revision of FIPS 180-1. National Institute of Standards and Technology. In Federal Register of August 26, 2002, Vol. 67, No. 165, Pages 54786–54787.

[NIST, 2004] NIST (2004). *Federal Information Processing Standards (FIPS) 180-2: Announcing the Secure Hash Standard (+ Change Notice to include SHA-224)*. National Institute of Standards and Technology (NIST).

[NIST, 2008] NIST (2008). *Federal Information Processing Standards (FIPS) 198-1: The Keyed-Hash Message Authentication Code (HMAC)*. National Institute of Standards and Technology (NIST).

[NIST, 2013] NIST (2013). *Federal Information Processing Standards (FIPS) 186-4: Digital Signature Standard (DSS)*. National Institute of Standards and Technology (NIST).

[NIST, 2015] NIST (2015). *Federal Information Processing Standards (FIPS) 180-4: Announcing the Secure Hash Standard*. National Institute of Standards and Technology (NIST).

[OMG, 2011] OMG (2011). UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems – Version 1.1.

[OpenSSL, 2013] OpenSSL (2013). EVP. OpenSSL Wiki: https://wiki.openssl.org/index.php?title=EVP&oldid=992.

[OpenSSL, 2015] OpenSSL (2015). OpenSSL – Manpages for 1.0.2. URL: https://www.openssl.org/docs/man1.0.2.

[OSEK Group, 2004a] OSEK Group (2004a). OSEK/VDX Binding Specification, Version 1.4.2.

[OSEK Group, 2004b] OSEK Group (2004b). OSEK/VDX Portal - Goals And Motivation. http://portal.osek-vdx.org/index.php?option=com_content&task=view&id=4&Itemid=4.

[OSEK Group, 2005] OSEK Group (2005). OSEK/VDX Operating System, Version 2.2.3.

[Papagiannopoulos, 2013] Papagiannopoulos, K. (2013). High-throughput implementations of lightweight ciphers in the AVR ATtiny architecture. Tese de mestrado, Radboud University Nijmegen.

[Parameswaran and Wolf, 2008] Parameswaran, S. and Wolf, T. (2008). Embedded Systems Security — An Overview. *Design Automation for Embedded Systems*, 12(3):173–183.

[Rama and Kak, 2015] Rama, G. M. and Kak, A. (2015). Some Structural Measures of API Usability. *Software – Practice & Experience*, 45(1):75–110.

[Ravi et al., 2004] Ravi, S., Raghunathan, A., Kocher, P., and Hattangady, S. (2004). Security in Embedded Systems: Design Challenges. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(3):461–491.

[Rela, 2003] Rela, M. A. d. C. Z. (2003). *Detecção de Erros Baseada na Semântica dos Dados*. PhD thesis, Universidade de Coimbra.

[Renaux, 2014] Renaux, D. P. B. (2014). Comparative Performance Evaluation of CMSIS-RTOS. In *Anais da edição de 2014 do Simpósio Brasileiro de Engenharia de Sistemas Computacionais*, Manaus.

[Renaux and Pottker, 2014] Renaux, D. P. B. and Pottker, F. (2014). Applicability of the CMSIS-RTOS Standard to the Internet of Things. In *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 284–291.

[Rivest et al., 1978] Rivest, R. L., Shamir, A., and Adleman, L. M. (1978). A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communications of the ACM*, 21(2):120–126.

[Rivest et al., 1983] Rivest, R. L., Shamir, A., and Adleman, L. M. (1983). Cryptographic communications system and method. US Patent 4,405,829. Originally deposited in 1997.

[Roman et al., 2000] Roman, G.-C., Murphy, A. L., and Picco, G. P. (2000). A Software Engineering Perspective on Mobility. In *Future of Software Engineering*. ACM Press.

[RSA, 2012] RSA (2012). *PKCS #1 v2.2: RSA Cryptography Standard*. RSA Laboratories. EMC Corporation Public-Key Cryptography Standards (PKCS).

[SCO, 2013] SCO (2013). System V Application Binary Interface - DRAFT - 10 June 2013.

[Silva et al., 2013] Silva, B., da Silva Jr, D. C., Souza, E. M., Pereira, F., Teixeira, F. A., Wong, H. C., Nazaré, H., Maffra, I., Freire, J., Santos, W. F., and Oliveira, L. B. (2013). Segurança de Software em Sistemas Embarcados: Ataques e Defesas. In *Minicursos do XIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais – SBSeg 2013*, pages 101–155. SBC.

[Smyslov, 1999] Smyslov, V. (1999). Simple Cryptographic Program Interface (Crypto API). RFC 2628 (Informational), `http://www.ietf.org/rfc/rfc2628.txt`.

[Tan and Tran Nguyen, 2009] Tan, S.-L. and Tran Nguyen, B. A. (2009). Survey and performance evaluation of real-time operating systems (RTOS) for small microcontrollers. *IEEE Micro*, PP(99).

[Tribble, 2001] Tribble, D. R. (2001). Incompatibilities Between ISO C and ISO C++. URL: http://david.tribble.com/text/cdiffs.htm.

[UBM Tech, 2015] UBM Tech (2015). 2015 Embedded Markets Study. Slide show by UBM Tech.

[Vasilevskaya, 2015] Vasilevskaya, M. (2015). Security in Embedded Systems: A Model-Based Approach with Risk Metrics. Master's thesis, Department of Computer and Information Science, Linköping University.

[Verissimo and de Lemos, 1989] Verissimo, P. and de Lemos, R. (1989). Confiança no funcionamento: Proposta para uma terminologia em português. Publicação conjunta INESC e LCMI/UFSC.

[Wangham et al., 2013] Wangham, M. S., Domenech, M. C., and Mello, E. R. d. (2013). Infraestruturas de Autenticação e de Autorização para Internet das Coisas. In *Minicursos do XIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais —- SBSeg 2013*, pages 156–205. SBC.

[Weber, 2003] Weber, T. S. (2003). Tolerância a falhas: conceitos e exemplos. http://www.inf.ufrgs.br/~taisy/disciplinas/textos/ConceitosDependabilidade.PDF. Apostila do Programa de Pós-Graduação – Instituto de Informática – UFRGS. Porto Alegre.

[wolfSSL, 2016] wolfSSL (2016). *wolfSSL User Manual – Version 3.9.0*. wolfSSL. URL: https://www.wolfssl.com/documentation/wolfSSL-Manual.pdf.

[Zachariadis et al., 2004] Zachariadis, S., Mascolo, C., and Emmerich, W. (2004). satin: A Component Model for Mobile Self Organisation. In Meersman, R. and Tari, Z., editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, volume 3291 of *Lecture Notes in Computer Science*, pages 1303–1321. Springer Berlin Heidelberg.

Apêndice A

Especificação completa da *GEmSysC*

Neste apêndice, está apresentada a especificação completa da *GEmSysC*. Esta especificação foi gerada com a ferramenta *Doxygen* [Doxygen, 2015] a partir dos arquivos de cabeçalho genéricos da biblioteca. Com base nessa documentação, é possível utilizar qualquer implementação da *GEmSysC*.

Sumário

1 Arquivos **1**

1.1 Referência do Arquivo cl/cl-aes.h 1

1.1.1 Descrição Detalhada 2

1.1.2 Definições e macros 2

1.1.3 Enumerações 3

1.1.4 Funções 3

1.2 Referência do Arquivo cl/cl-rsa.h 5

1.2.1 Descrição Detalhada 6

1.2.2 Definições e macros 6

1.2.3 Enumerações 7

1.2.4 Funções 7

1.3 Referência do Arquivo cl/cl-sha256.h 11

1.3.1 Descrição Detalhada 12

1.3.2 Definições e macros 12

1.3.3 Funções 13

1.4 Referência do Arquivo cl/cl.h 13

1.4.1 Descrição Detalhada 14

1.4.2 Enumerações 14

1.4.3 Funções 14

Índice **17**

1 Arquivos

1.1 Referência do Arquivo cl/cl-aes.h

Arquivo de cabeçalho principal do módulo de AES da GEmSysC. Neste arquivo, estão declaradas todas as construções relativas ao módulo de AES da GEmSysC.

```
#include <cl/cl.h>
#include "cl/cl-aes-internal.h"
```

Definições e Macros

- #define **clAESDef**(name, keyLength)

  Define a estrutura que armazena o estado interno da instância do mecanismo criptográfico de AES denominada `name` *com uma chave de tamanho igual a* `keyLength`*. Esta macro deve ser incluída em arquivos de código. De acordo com a implementação, esta macro pode ser traduzida em uma ou mais definições.*
- #define **clAESDecl**(name)

  Declara a estrutura que armazena o estado interno da instância do mecanismo criptográfico de AES denominada `name`*. Esta macro deve ser incluída em arquivos de cabeçalho, pois declara a estrutura mas não a define.*
- #define **clAES**(name)

  Faz referência à estrutura que armazena o estado interno da instância do mecanismo criptográfico de AES denominada `name`*. Esta macro deve ser utilizada no código de usuário quando for necessário obter uma referência a esta estrutura.*

Definições de Tipos

- typedef struct cl_aes_def_s **clAESDef_t**

  Tipo de dados da estrutura que armazena o estado interno de instâncias do mecanismo criptográfico de AES. Implementações são livres para utilizar mais de uma estrutura de dados para armazenar o estado interno das instâncias. Este é o tipo da estrutura de dados principal.

Enumerações

Funções

- **clStatus clAESEncryptCreate** (**clEngineInstanceId** ∗id, const **clAESDef_t** ∗aes_def, **clBlockCipherMode** mode, const void ∗key, uint32_t key_len, const void ∗iv, uint32_t iv_len)

  Inicializa uma instância do mecanismo criptográfico de AES para cifragem de dados. O comportamento desta função é indefinido se a estrutura que armazena o estado interno já estiver associada a uma instância.
- **clStatus clAESDecryptCreate** (**clEngineInstanceId** ∗id, const **clAESDef_t** ∗aes_def, **clBlockCipherMode** mode, const void ∗key, uint32_t key_len, const void ∗iv, uint32_t iv_len)

  Inicializa uma instância do mecanismo criptográfico de AES para decifragem de dados. O comportamento desta função é indefinido se a estrutura que armazena o estado interno já estiver associada a uma instância.

1.1.1 Descrição Detalhada

Arquivo de cabeçalho principal do módulo de AES da GEmSysC. Neste arquivo, estão declaradas todas as construções relativas ao módulo de AES da GEmSysC.

1.1.2 Definições e macros

1.1.2.1 #define clAES(*name*)

Faz referência à estrutura que armazena o estado interno da instância do mecanismo criptográfico de AES denominada `name`. Esta macro deve ser utilizada no código de usuário quando for necessário obter uma referência a esta estrutura.

Parâmetros

name	Nome da instância do mecanismo criptográfico de AES.

1.1.2.2 #define clAESDecl(*name*)

Declara a estrutura que armazena o estado interno da instância do mecanismo criptográfico de AES denominada `name`. Esta macro deve ser incluída em arquivos de cabeçalho, pois declara a estrutura mas não a define.

Parâmetros

name	Nome da instância do mecanismo criptográfico de AES.

1.1.2.3 #define clAESDef(*name, keyLength*)

Define a estrutura que armazena o estado interno da instância do mecanismo criptográfico de AES denominada `name` com uma chave de tamanho igual a `keyLength`. Esta macro deve ser incluída em arquivos de código. De acordo com a implementação, esta macro pode ser traduzida em uma ou mais definições.

Parâmetros

name	Nome da instância do mecanismo criptográfico de AES.
keyLength	Tamanho da chave de AES utilizada por este mecanismo.

1.1.3 Enumerações

1.1.3.1 enum clAESKeyLength_t

Tipo de dados que define os tamanhos possíveis para a chave do AES. Cada constante desta enumeração é nomeada com base no tamanho em bits da chave, mas tem como valor o tamanho em bytes da chave.

Valores de enumerações

CL_AES_KEYLENGTH_128 Chave AES com 128 bits (16 bytes).

CL_AES_KEYLENGTH_192 Chave AES com 192 bits (24 bytes).

CL_AES_KEYLENGTH_256 Chave AES com 256 bits (32 bytes).

1.1.4 Funções

1.1.4.1 clAESDecryptCreate (clEngineInstanceId ∗ *id,* const clAESDef_t ∗ *aes_def,* clBlockCipherMode *mode,* const void ∗ *key,* uint32_t *key_len,* const void ∗ *iv,* uint32_t *iv_len*)

Inicializa uma instância do mecanismo criptográfico de AES para decifragem de dados. O comportamento desta função é indefinido se a estrutura que armazena o estado interno já estiver associada a uma instância.

Parâmetros

out	*id*	Variável que deve receber o identificador da instância do mecanismo criptográfico.
in	*aes_def*	Referência à estrutura que armazena o estado interno da instância do mecanismo criptográfico.
in	*mode*	Modo de operação do mecanismo de criptografia em blocos.
in	*key*	Ponteiro para o buffer que contém a chave.
in	*key_len*	Tamanho do buffer que contém a chave.
in	*iv*	Ponteiro para o buffer que contém o vetor de inicialização.
in	*iv_len*	- Tamanho do buffer que contém o vetor de inicialização.

Retorna

Código de erro indicando o resultado da operação.

- `CL_ENOERR` - Indica sucesso na operação.
- `-CL_ENOSYS` - Esta implementação não suporta decifragem de dados ou algum dos parâmetros indica funcionalidade não suportada por esta implementação.
- `-CL_EINVAL` - Algum dos parâmetros é inválido ou inconsistente.
- `-CL_EAGAIN` - Houve algum erro interno, e talvez seja possível tentar mais tarde.

1.1.4.2 clAESEncryptCreate (clEngineInstanceId * *id*, const clAESDef_t * *aes_def*, clBlockCipherMode *mode*, const void * *key*, uint32_t *key_len*, const void * *iv*, uint32_t *iv_len*)

Inicializa uma instância do mecanismo criptográfico de AES para cifragem de dados. O comportamento desta função é indefinido se a estrutura que armazena o estado interno já estiver associada a uma instância.

Parâmetros

out	*id*	Variável que deve receber o identificador da instância do mecanismo criptográfico.
in	*aes_def*	Referência à estrutura que armazena o estado interno da instância do mecanismo criptográfico.
in	*mode*	Modo de operação do mecanismo de criptografia em blocos.
in	*key*	Ponteiro para o buffer que contém a chave.
in	*key_len*	Tamanho do buffer que contém a chave.
in	*iv*	Ponteiro para o buffer que contém o vetor de inicialização.
in	*iv_len*	Tamanho do buffer que contém o vetor de inicialização.

Retorna

Código de erro indicando o resultado da operação.

- `CL_ENOERR` - Indica sucesso na operação.
- `-CL_ENOSYS` - Esta implementação não suporta cifragem de dados ou algum dos parâmetros indica funcionalidade não suportada por esta implementação.
- `-CL_EINVAL` - Algum dos parâmetros é inválido ou inconsistente.
- `-CL_EAGAIN` - Houve algum erro interno, e talvez seja possível tentar mais tarde.

1.2 Referência do Arquivo cl/cl-rsa.h

Arquivo de cabeçalho principal do módulo de RSA da GEmSysC. Neste arquivo, estão declaradas todas as construções relativas ao módulo de RSA da GEmSysC.

```
#include <cl/cl.h>
#include "cl/cl-rsa-internal.h"
```

Definições e Macros

- #define **clRSADef**(name, maxKeyBytes)

  Define a estrutura que armazena o estado interno da instância do mecanismo criptográfico de RSA denominada `name` com uma chave de tamanho máximo igual a `maxKeyBytes`. Esta macro deve ser incluída em arquivos de código. De acordo com a implementação, esta macro pode ser traduzida em uma ou mais definições. O valor fornecido em `maxKeyBytes` pode ou não ser respeitado pela implementação.
- #define **clRSADecl**(name)

  Declara a estrutura que armazena o estado interno da instância do mecanismo criptográfico de RSA denominada `name`. Esta macro deve ser incluída em arquivos de cabeçalho, pois declara a estrutura mas não a define.
- #define **clRSA**(name)

  Faz referência à estrutura que armazena o estado interno da instância do mecanismo criptográfico de RSA denominada `name`. Esta macro deve ser utilizada no código de usuário quando for necessário obter uma referência a esta estrutura.

Definições de Tipos

- typedef struct cl_rsa_def_s **clRSADef_t**

  Tipo de dados da estrutura que armazena o estado interno de instâncias do mecanismo criptográfico de RSA. Implementações são livres para utilizar mais de uma estrutura de dados para armazenar o estado interno das instâncias. Este é o tipo da estrutura de dados principal.

Enumerações

Funções

- **clStatus clRSAPrivateEncryptCreate** (**clEngineInstanceId** ∗id, const **clRSADef_t** ∗rsa_def, **clRSAPadding↩ Mode_t** padding_mode, **clRSAKeyEncoding_t** key_encoding, **clRSAKeyFormat_t** key_format, const void ∗key, uint32_t key_len)

  Inicializa uma instância do mecanismo criptográfico de RSA para cifragem de dados com chave privada. O comportamento desta função é indefinido se a estrutura que armazena o estado interno já estiver associada a uma instância.
- **clStatus clRSAPrivateDecryptCreate** (**clEngineInstanceId** ∗id, const **clRSADef_t** ∗rsa_def, **clRSAPadding↩ Mode_t** padding_mode, **clRSAKeyEncoding_t** key_encoding, **clRSAKeyFormat_t** key_format, const void ∗key, uint32_t key_len)

  Inicializa uma instância do mecanismo criptográfico de RSA para decifragem de dados com chave privada. O comportamento desta função é indefinido se a estrutura que armazena o estado interno já estiver associada a uma instância.
- **clStatus clRSAPublicEncryptCreate** (**clEngineInstanceId** ∗id, const **clRSADef_t** ∗rsa_def, **clRSAPadding↩ Mode_t** padding_mode, **clRSAKeyEncoding_t** key_encoding, **clRSAKeyFormat_t** key_format, const void ∗key, uint32_t key_len)

Inicializa uma instância do mecanismo criptográfico de RSA para cifragem de dados com chave pública. O comportamento desta função é indefinido se a estrutura que armazena o estado interno já estiver associada a uma instância.

- **clStatus clRSAMinInputLength** (**clEngineInstanceId** id)

  Verifica o tamanho mínimo do buffer de dados de entrada que deve ser fornecido a uma instância de mecanismo criptográfico. Este tamanho varia de acordo com o tipo de enchimento utilizado e com a direção da operação de criptografia. O comportamento desta função é indefinido se a instância de mecanismo criptográfico não for de RSA.

- **clStatus clRSAMaxInputLength** (**clEngineInstanceId** id)

  Verifica o tamanho máximo do buffer de dados de entrada que deve ser fornecido a uma instância de mecanismo criptográfico. Este tamanho varia de acordo com o tipo de enchimento utilizado e com a direção da operação de criptografia. O comportamento desta função é indefinido se a instância de mecanismo criptográfico não for de RSA.

- **clStatus clRSAMinOutputLength** (**clEngineInstanceId** id)

  Verifica o tamanho mínimo do buffer de dados de saída que deve ser fornecido a uma instância de mecanismo criptográfico. Este tamanho varia de acordo com o tipo de enchimento utilizado e com a direção da operação de criptografia. O comportamento desta função é indefinido se a instância de mecanismo criptográfico não for de RSA.

- **clStatus clRSAMaxOutputLength** (**clEngineInstanceId** id)

  Verifica o tamanho máximo do buffer de dados de entrada que deve ser fornecido a uma instância de mecanismo criptográfico. Este tamanho varia de acordo com o tipo de enchimento utilizado e com a direção da operação de criptografia. O comportamento desta função é indefinido se a instância de mecanismo criptográfico não for de RSA.

1.2.1 Descrição Detalhada

Arquivo de cabeçalho principal do módulo de RSA da GEmSysC. Neste arquivo, estão declaradas todas as construções relativas ao módulo de RSA da GEmSysC.

1.2.2 Definições e macros

1.2.2.1 #define clRSA(*name*)

Faz referência à estrutura que armazena o estado interno da instância do mecanismo criptográfico de RSA denominada `name`. Esta macro deve ser utilizada no código de usuário quando for necessário obter uma referência a esta estrutura.

Parâmetros

name	Nome da instância do mecanismo criptográfico de RSA.

1.2.2.2 #define clRSADecl(*name*)

Declara a estrutura que armazena o estado interno da instância do mecanismo criptográfico de RSA denominada `name`. Esta macro deve ser incluída em arquivos de cabeçalho, pois declara a estrutura mas não a define.

Parâmetros

name	Nome da instância do mecanismo criptográfico de RSA.

1.2.2.3 #define clRSADef(*name*, *maxKeyBytes*)

Define a estrutura que armazena o estado interno da instância do mecanismo criptográfico de RSA denominada `name` com uma chave de tamanho máximo igual a `maxKeyBytes`. Esta macro deve ser incluída em arquivos de código. De acordo com a implementação, esta macro pode ser traduzida em uma ou mais definições. O valor fornecido em `maxKeyBytes` pode ou não ser respeitado pela implementação.

Parâmetros

name	Nome da instância do mecanismo criptográfico de RSA.
maxKeyBytes	Tamanho máximo da chave de RSA utilizada por este mecanismo. Este valor pode ou não ser respeitado pela implementação.

1.2.3 Enumerações

1.2.3.1 enum clRSAKeyEncoding_t

Tipo de dados que define as codificações possíveis para as chaves RSA.

Valores de enumerações

CL_RSA_KEY_ENCODING_DER Codificação DER.
CL_RSA_KEY_ENCODING_PEM Codificação PEM.

1.2.3.2 enum clRSAKeyFormat_t

Tipo de dados que define os formatos possíveis para as chaves RSA.

Valores de enumerações

CL_RSA_KEY_FORMAT_PKCS1 Formato PKCS #1.

1.2.3.3 enum clRSAPaddingMode_t

Tipo de dados que define os enchimentos possíveis com o mecanismo de RSA.

Valores de enumerações

CL_RSA_PADDING_MODE_PKCS1v15 Enchimento PKCS#1 v1.5.
CL_RSA_PADDING_MODE_OAEP Enchimento OAEP.

1.2.4 Funções

1.2.4.1 clRSAMaxInputLength (clEngineInstanceId *id*)

Verifica o tamanho máximo do buffer de dados de entrada que deve ser fornecido a uma instância de mecanismo criptográfico. Este tamanho varia de acordo com o tipo de enchimento utilizado e com a direção da operação de criptografia. O comportamento desta função é indefinido se a instância de mecanismo criptográfico não for de RSA.

Parâmetros

in	*id*	Identificador da instância do mecanismo criptográfico.

Retorna

Tamanho máximo do buffer ou código de erro indicando o resultado da operação.

- Valor positivo - Indica o tamanho máximo do buffer.
- `−CL_EINVAL` - O identificador não é válido.

1.2.4.2 clRSAMaxOutputLength (clEngineInstanceId *id*)

Verifica o tamanho máximo do buffer de dados de entrada que deve ser fornecido a uma instância de mecanismo criptográfico. Este tamanho varia de acordo com o tipo de enchimento utilizado e com a direção da operação de criptografia. O comportamento desta função é indefinido se a instância de mecanismo criptográfico não for de RSA.

Parâmetros

in	*id*	Identificador da instância do mecanismo criptográfico.

Retorna

Tamanho máximo do buffer ou código de erro indicando o resultado da operação.

- Valor positivo - Indica o tamanho máximo do buffer.
- `−CL_EINVAL` - O identificador não é válido.

1.2.4.3 clRSAMinInputLength (clEngineInstanceId *id*)

Verifica o tamanho mínimo do buffer de dados de entrada que deve ser fornecido a uma instância de mecanismo criptográfico. Este tamanho varia de acordo com o tipo de enchimento utilizado e com a direção da operação de criptografia. O comportamento desta função é indefinido se a instância de mecanismo criptográfico não for de RSA.

Parâmetros

in	*id*	Identificador da instância do mecanismo criptográfico.

Retorna

Tamanho mínimo do buffer ou código de erro indicando o resultado da operação.

- Valor positivo - Indica o tamanho mínimo do buffer.
- `−CL_EINVAL` - O identificador não é válido.

1.2.4.4 clRSAMinOutputLength (**clEngineInstanceId** *id*)

Verifica o tamanho mínimo do buffer de dados de saída que deve ser fornecido a uma instância de mecanismo criptográfico. Este tamanho varia de acordo com o tipo de enchimento utilizado e com a direção da operação de criptografia. O comportamento desta função é indefinido se a instância de mecanismo criptográfico não for de RSA.

Parâmetros

`in`	*id*	Identificador da instância do mecanismo criptográfico.

Retorna

Tamanho mínimo do buffer ou código de erro indicando o resultado da operação.

- Valor positivo - Indica o tamanho mínimo do buffer.
- `−CL_EINVAL` - O identificador não é válido.

1.2.4.5 clRSAPrivateDecryptCreate (**clEngineInstanceId** ∗ *id*, const **clRSADef_t** ∗ *rsa_def*, **clRSAPaddingMode_t** *padding_mode*, **clRSAKeyEncoding_t** *key_encoding*, **clRSAKeyFormat_t** *key_format*, const void ∗ *key*, uint32_t *key_len*)

Inicializa uma instância do mecanismo criptográfico de RSA para decifragem de dados com chave privada. O comportamento desta função é indefinido se a estrutura que armazena o estado interno já estiver associada a uma instância.

Parâmetros

`out`	*id*	Variável que deve receber o identificador da instância do mecanismo criptográfico.
`in`	*rsa_de*	Referência à estrutura que armazena o estado interno da instância do mecanismo criptográfico.
`in`	*padding_mode*	Tipo de enchimento utilizado.
`in`	*key_encoding*	Codificação utilizada na chave.
`in`	*key_format*	Formato da chave.
`in`	*key*	- Ponteiro para o buffer que contém a chave privada.
`in`	*key_len*	- Tamanho do buffer que contém a chave privada.

Retorna

Código de erro indicando o resultado da operação.

- `CL_ENOERR` - Indica sucesso na operação.
- `−CL_ENOSYS` - Esta implementação não suporta decifragem de dados com chave privada ou algum dos parâmetros indica funcionalidade não suportada por esta implementação.
- `−CL_EINVAL` - Algum dos parâmetros é inválido ou inconsistente.
- `−CL_ENOMEM` - A chave fornecida é maior que o tamanho máximo suportado pela estrutura de dados.
- `−CL_EAGAIN` - Houve algum erro interno, e talvez seja possível tentar mais tarde.

1.2.4.6 clRSAPrivateEncryptCreate (**clEngineInstanceId** ∗ *id,* const **clRSADef_t** ∗ *rsa_def,* **clRSAPaddingMode_t** *padding_mode,* **clRSAKeyEncoding_t** *key_encoding,* **clRSAKeyFormat_t** *key_format,* const void ∗ *key,* uint32_t *key_len*)

Inicializa uma instância do mecanismo criptográfico de RSA para cifragem de dados com chave privada. O comportamento desta função é indefinido se a estrutura que armazena o estado interno já estiver associada a uma instância.

Parâmetros

out	*id*	Variável que deve receber o identificador da instância do mecanismo criptográfico.
in	*rsa_def*	Referência à estrutura que armazena o estado interno da instância do mecanismo criptográfico.
in	*padding_mode*	Tipo de enchimento utilizado.
in	*key_encoding*	Codificação utilizada na chave.
in	*key_format*	Formato da chave.
in	*key*	Ponteiro para o buffer que contém a chave privada.
in	*key_len*	Tamanho do buffer que contém a chave privada.

Retorna

Código de erro indicando o resultado da operação.

- `CL_ENOERR` - Indica sucesso na operação.
- `−CL_ENOSYS` - Esta implementação não suporta cifragem de dados com chave privada ou algum dos parâmetros indica funcionalidade não suportada por esta implementação.
- `−CL_EINVAL` - Algum dos parâmetros é inválido ou inconsistente.
- `−CL_ENOMEM` - A chave fornecida é maior que o tamanho máximo suportado pela estrutura de dados.
- `−CL_EAGAIN` - Houve algum erro interno, e talvez seja possível tentar mais tarde.

1.2.4.7 clRSAPublicEncryptCreate (**clEngineInstanceId** ∗ *id,* const **clRSADef_t** ∗ *rsa_def,* **clRSAPaddingMode_t** *padding_mode,* **clRSAKeyEncoding_t** *key_encoding,* **clRSAKeyFormat_t** *key_format,* const void ∗ *key,* uint32_t *key_len*)

Inicializa uma instância do mecanismo criptográfico de RSA para cifragem de dados com chave pública. O comportamento desta função é indefinido se a estrutura que armazena o estado interno já estiver associada a uma instância.

Inicializa uma instância do mecanismo criptográfico de RSA para decifragem de dados com chave pública. O comportamento desta função é indefinido se a estrutura que armazena o estado interno já estiver associada a uma instância.

Parâmetros

out	*id*	Variável que deve receber o identificador da instância do mecanismo criptográfico.
in	*rsa_def*	Referência à estrutura que armazena o estado interno da instância do mecanismo criptográfico.
in	*padding_mode*	Tipo de enchimento utilizado.
in	*key_encoding*	Codificação utilizada na chave.
in	*key_format*	Formato da chave.
in	*key*	Ponteiro para o buffer que contém a chave pública.
in	*key_len*	- Tamanho do buffer que contém a chave pública.

Retorna

Código de erro indicando o resultado da operação.

- `CL_ENOERR` - Indica sucesso na operação.
- `-CL_ENOSYS` - Esta implementação não suporta cifragem de dados com chave pública ou algum dos parâmetros indica funcionalidade não suportada por esta implementação.
- `-CL_EINVAL` - Algum dos parâmetros é inválido ou inconsistente.
- `-CL_ENOMEM` - A chave fornecida é maior que o tamanho máximo suportado pela estrutura de dados.
- `-CL_EAGAIN` - Houve algum erro interno, e talvez seja possível tentar mais tarde.

Parâmetros

`out`	*id*	Variável que deve receber o identificador da instância do mecanismo criptográfico.
`in`	*rsa_def*	Referência à estrutura que armazena o estado interno da instância do mecanismo criptográfico.
`in`	*padding_mode*	Tipo de enchimento utilizado.
`in`	*key_encoding*	Codificação utilizada na chave.
`in`	*key_format*	Formato da chave.
`in`	*key*	Ponteiro para o buffer que contém a chave pública.
`in`	*key_len*	Tamanho do buffer que contém a chave pública.

Retorna

Código de erro indicando o resultado da operação.

- `CL_ENOERR` - Indica sucesso na operação.
- `-CL_ENOSYS` - Esta implementação não suporta decifragem de dados com chave pública ou algum dos parâmetros indica funcionalidade não suportada por esta implementação.
- `-CL_EINVAL` - Algum dos parâmetros é inválido ou inconsistente.
- `-CL_ENOMEM` - A chave fornecida é maior que o tamanho máximo suportado pela estrutura de dados.
- `-CL_EAGAIN` - Houve algum erro interno, e talvez seja possível tentar mais tarde.

1.3 Referência do Arquivo cl/cl-sha256.h

Arquivo de cabeçalho principal do módulo de SHA-256 da GEmSysC. Neste arquivo, estão declaradas todas as construções relativas ao módulo de SHA-256 da GEmSysC.

```
#include <cl/cl.h>
#include "cl/cl-sha256-internal.h"
```

Definições e Macros

- #define **clSHA256Def**(name)

  Define a estrutura que armazena o estado interno da instância do mecanismo criptográfico de SHA-256 denominada `name`*. Esta macro deve ser incluída em arquivos de código. De acordo com a implementação, esta macro pode ser traduzida em uma ou mais definições.*

- #define **clSHA256Decl**(name)

  Declara a estrutura que armazena o estado interno da instância do mecanismo criptográfico de SHA-256 denominada `name`. Esta macro deve ser incluída em arquivos de cabeçalho, pois declara a estrutura mas não a define.
- #define **clSHA256**(name)

  Faz referência à estrutura que armazena o estado interno da instância do mecanismo criptográfico de SHA-256 denominada `name`. Esta macro deve ser utilizada no código de usuário quando for necessário obter uma referência a esta estrutura.

Definições de Tipos

- typedef struct cl_sha256_def_s **clSHA256Def_t**

  Tipo de dados da estrutura que armazena o estado interno de instâncias do mecanismo criptográfico de SHA-256. Implementações são livres para utilizar mais de uma estrutura de dados para armazenar o estado interno das instâncias. Este é o tipo da estrutura de dados principal.

Funções

- **clStatus clSHA256Create** (**clEngineInstanceId** ∗id, const **clSHA256Def_t** ∗sha256_def)

  Inicializa uma instância do mecanismo criptográfico de SHA-256 para resumo criptográfico de dados. O comportamento desta função é indefinido se a estrutura que armazena o estado interno já estiver associada a uma instância.

1.3.1 Descrição Detalhada

Arquivo de cabeçalho principal do módulo de SHA-256 da GEmSysC. Neste arquivo, estão declaradas todas as construções relativas ao módulo de SHA-256 da GEmSysC.

1.3.2 Definições e macros

1.3.2.1 #define clSHA256(*name*)

Faz referência à estrutura que armazena o estado interno da instância do mecanismo criptográfico de SHA-256 denominada `name`. Esta macro deve ser utilizada no código de usuário quando for necessário obter uma referência a esta estrutura.

Parâmetros

name	Nome da instância do mecanismo criptográfico de SHA-256.

1.3.2.2 #define clSHA256Decl(*name*)

Declara a estrutura que armazena o estado interno da instância do mecanismo criptográfico de SHA-256 denominada `name`. Esta macro deve ser incluída em arquivos de cabeçalho, pois declara a estrutura mas não a define.

Parâmetros

name	Nome da instância do mecanismo criptográfico de SHA-256.

1.3.2.3 #define clSHA256Def(*name*)

Define a estrutura que armazena o estado interno da instância do mecanismo criptográfico de SHA-256 denominada `name`. Esta macro deve ser incluída em arquivos de código. De acordo com a implementação, esta macro pode ser traduzida em uma ou mais definições.

Parâmetros

name	Nome da instância do mecanismo criptográfico de SHA-256.

1.3.3 Funções

1.3.3.1 clSHA256Create (clEngineInstanceId * *id*, const clSHA256Def_t * *sha256_def*)

Inicializa uma instância do mecanismo criptográfico de SHA-256 para resumo criptográfico de dados. O comportamento desta função é indefinido se a estrutura que armazena o estado interno já estiver associada a uma instância.

Parâmetros

`out`	*id*	Variável que deve receber o identificador da instância do mecanismo criptográfico.
`in`	*sha256_def*	Referência à estrutura que armazena o estado interno da instância do mecanismo criptográfico.

Retorna

Código de erro indicando o resultado da operação.

- `CL_ENOERR` - Indica sucesso na operação.
- `-CL_ENOSYS` - Esta implementação não suporta geração de resumo criptográfico ou algum dos parâmetros indica funcionalidade não suportada por esta implementação.
- `-CL_EINVAL` - Algum dos parâmetros é inválido ou inconsistente.
- `-CL_EAGAIN` - Houve algum erro interno, e talvez seja possível tentar mais tarde.

1.4 Referência do Arquivo cl/cl.h

Arquivo de cabeçalho principal da GEmSysC. Neste arquivo, estão declaradas todas as construções da GEmSysC que são independentes do mecanismo criptográfico utiizado.

```
#include <stdint.h>
#include <stdbool.h>
#include "cl/cl-internal.h"
```

Definições de Tipos

- typedef int32_t **clStatus**

  Tipo de dados que define um resultado numérico ou um código de erros.
- typedef const void * **clEngineInstanceId**

  Tipo de dados que define uma instância de mecanismo criptográfico.

Enumerações

Funções

- **clStatus clEngineProcess** (**clEngineInstanceId** id, const void *in, uint32_t in_len, void *out, uint32_t out_len, void *extra)

  Processa dados através de uma instância de mecanismo criptográfico.
- **clStatus clEngineFinalize** (**clEngineInstanceId** id)

  Libera recursos utilizados por uma instância de mecanismo criptográfico.

1.4.1 Descrição Detalhada

Arquivo de cabeçalho principal da GEmSysC. Neste arquivo, estão declaradas todas as construções da GEmSysC que são independentes do mecanismo criptográfico utiizado.

1.4.2 Enumerações

1.4.2.1 enum clBlockCipherMode

Tipo de dados que define os modos de operação de mecanismos de criptografia em blocos.

Valores de enumerações

CL_BLOCK_CIPHER_MODE_ECB Modo de operação ECB.

CL_BLOCK_CIPHER_MODE_CBC Modo de operação CBC.

1.4.3 Funções

1.4.3.1 clEngineFinalize (clEngineInstanceId *id*)

Libera recursos utilizados por uma instância de mecanismo criptográfico.

Parâmetros

`out`	*id*	- Identificador da instância do mecanismo criptográfico.

Retorna

Código de erro indicando o resultado da operação.

- `CL_ENOERR` - Indica sucesso na operação.
- `-CL_EAGAIN` - Houve algum erro interno, e talvez seja possível tentar mais tarde.

1.4.3.2 clEngineProcess (**clEngineInstanceId** *id*, const void ∗ *in*, uint32_t *in_len*, void ∗ *out*, uint32_t *out_len*, void ∗ *extra*)

Processa dados através de uma instância de mecanismo criptográfico.

Parâmetros

`out`	*id*	- Identificador da instância do mecanismo criptográfico.
`in`	*in*	- Ponteiro para o buffer de entrada.
`in`	*in_len*	- Tamanho do buffer de entrada.
`out`	*out*	- Ponteiro para o buffer de saída.
`in`	*out_len*	- Tamanho do buffer de saída.
`in,out`	*extra*	- Parâmetros extras para utilizar na cifragem. O formato destes dados varia de acordo com o mecanismo.

Retorna

Número de bytes escritos no buffer de saída ou código de erro indicando o resultado da operação.

- Valor positivo - Indica o número de bytes escritos no buffer de saída.
- `-CL_EINVAL` - Algum dos parâmetros é inválido ou inconsistente.
- `-CL_EAGAIN` - Houve algum erro interno, e talvez seja possível tentar mais tarde.

Apêndice B

Códigos da análise de complexidade

Neste apêndice. estão apresentados os códigos feitos para a análise comparativa de complexidade de código entre a *GEmSysC* e as bibliotecas *wolfSSL* e *OpenSSL*. Esta análise buscou comparar o número de "construções" de linguagem utilizadas para implementar algumas operações básicas. Seguem as operações consideradas:

- Cifragem e decifragem de dados usando *AES* em modo *ECB*.
- Cifragem e decifragem de dados usando *AES* em modo *CBC*.
- Cifragem e decifragem de dados usando *RSA* com preenchimento *PKCS* #1 v1.5.
- Cifragem e decifragem de dados usando *RSA* com preenchimento *OAEP*.
- Geração de resumo criptográfico por *SHA-256*.

B.1 *AES* em modo *ECB*

O código que realiza a cifragem e a decifragem dos dados usando *AES* em modo *ECB* com chave de 256 *bits* está apresentado no código B.1 para a *GEmSysC*, no código B.2 para a *OpenSSL* e no código B.3 para a *wolfSSL*. Nestes códigos, assume-se a existência das seguintes estruturas: `aes_key` (*buffer* com a chave criptográfica), `aes_iv` (*buffer* com o vetor de inicialização), `aes_in_buffer` (*buffer* de entrada), `aes_encrypted_buffer` (*buffer* de saída da cifragem) e `aes_decrypted_buffer` (*buffer* de saída da decifragem). Assume-se que foram declaradas no código as seguintes constantes: `AES_KEY_LENGTH` (tamanho da chave, igual a 16 bytes, ou seja, 256 bits), `AES_BLOCK_LENGTH` (tamanho do bloco, igual a 8 bytes, ou seja, 128 bits) e `AES_DATA_LENGTH` (tamanho dos dados, múltiplo do tamanho do bloco).

Seguem as métricas do código em B.1. É utilizado um tipo de dados da *API*: `clEngineInstanceId`. São utilizadas duas constantes da *API*: `CL_AES_KEYLENGTH_256` e `CL_BLOCK_CIPHER_MODE_ECB`. São utilizadas seis funções ou *macros* parametrizadas diferentes: `clAESDef` (com dois parâmetros e utilizada uma vez), `clAES` (com um parâmetro e utilizada duas vezes), `clAESEncryptCreate` (com sete parâmetros e utilizada uma vez), `clAESDecryptCreate` (com sete parâmetros e utilizada uma vez), `clEngineProcess` (com seis parâmetros e utilizada duas vezes) e

Código B.1: Cifragem e decifragem usando *AES* em modo *ECB* pela *GEmSysC*.

```
1  clAESDef(aes_engine, CL_AES_KEYLENGTH_256);
2  clEngineInstanceId id;
3
4  // cifra dados.
5  clAESEncryptCreate(&id, clAES(aes_engine), CL_BLOCK_CIPHER_MODE_ECB, (const
       void*)aes_key, AES_KEY_LENGTH, (const void*)aes_iv, AES_BLOCK_LENGTH);
6  clEngineProcess(id, (const void*)aes_in_buffer, AES_DATA_LENGTH, (void*)
      aes_encrypted_buffer, AES_DATA_LENGTH, NULL);
7  clEngineFinalize(id);
8
9  // decifra dados.
10 clAESDecryptCreate(&id, clAES(aes_engine), CL_BLOCK_CIPHER_MODE_ECB, (const
       void*)aes_key, AES_KEY_LENGTH, (const void*)aes_iv, AES_BLOCK_LENGTH);
11 clEngineProcess(id, (const void*)aes_encrypted_buffer, AES_DATA_LENGTH, (
      void*)aes_decrypted_buffer, AES_DATA_LENGTH, NULL);
12 clEngineFinalize(id);
```

`clEngineFinalize` (com um parâmetro e utilizada duas vezes). Considerando as declarações de variáveis, que ocupam duas linhas, este código tem um total de oito linhas (ignorando linhas vazias e comentários). O somatório dos números de parâmetros das de funções ou *macros* parametrizadas é de 32.

Seguem as métricas do código em B.2. É utilizado um tipo de dados da *API*: `EVP_CIPHER_CTX`. Não são utilizadas constantes da *API*. São utilizadas oito funções ou *macros* parametrizadas da *API*: `EVP_CIPHER_CTX_new` (sem parâmetros e utilizada duas vezes), `EVP_EncryptInit_ex` (com cinco parâmetros e utilizada uma vez), `EVP_DecryptInit_ex` (com cinco parâmetros e utilizada uma vez), `EVP_aes_256_ecb` (sem parâmetros e utilizada duas vezes), `EVP_CIPHER_CTX_set_padding` (com dois parâmetros e utilizada duas vezes), `EVP_EncryptUpdate` (com cinco parâmetros e utilizada uma vez), `EVP_DecryptUpdate` (com cinco parâmetros e utilizada uma vez) e `EVP_CIPHER_CTX_free` (com um parâmetro e utilizada duas vezes). Considerando as declarações de variáveis, que ocupam duas linhas, este código tem um total de 14 linhas (ignorando linhas vazias e comentários). O somatório dos números de parâmetros das de funções ou *macros* parametrizadas é de 26.

Seguem as métricas do código em B.3. São utilizados dois tipos de dados da *API*: `Aes` e `byte`. São utilizadas duas constantes da *API*: `AES_ENCRYPTION` e `AES_DECRYPTION`. São utilizadas três funções ou *macros* parametrizadas da *API*: `wc_AesSetKeyDirect` (com cinco parâmetros e utilizada duas vezes), `wc_AesEncryptDirect` (com três parâmetros e utilizada uma vez) e `wc_AesDecryptDirect` (com três parâmetros e utilizada uma vez). Considerando a declaração de variável, que ocupa uma linha, este código tem um total de sete linhas (ignorando linhas vazias e comentários). O somatório dos números de parâmetros das de funções ou *macros* parametrizadas é de 16.

B.2 *AES* em modo *CBC*

O código que realiza a cifragem e a decifragem dos dados usando *AES* em modo *CBC* está apresentado no código B.4 para a *GEmSysC*, no código B.5 para a *OpenSSL* e no

Código B.2: Cifragem e decifragem usando *AES* em modo *ECB* pela *OpenSSL*.

```
EVP_CIPHER_CTX * ctx;
int actual_length;

// cifra dados.
ctx = EVP_CIPHER_CTX_new();
EVP_EncryptInit_ex(ctx, EVP_aes_256_ecb(), NULL, aes_key, aes_iv);
EVP_CIPHER_CTX_set_padding(ctx, 0);
actual_length = AES_DATA_LENGTH;
EVP_EncryptUpdate(ctx, aes_encrypted_buffer, &actual_length, (const
    unsigned char*)aes_in_buffer, AES_DATA_LENGTH);
EVP_CIPHER_CTX_free(ctx);

// decifra dados.
ctx = EVP_CIPHER_CTX_new();
EVP_DecryptInit_ex(ctx, EVP_aes_256_ecb(), NULL, aes_key, aes_iv);
EVP_CIPHER_CTX_set_padding(ctx, 0);
actual_length = AES_DATA_LENGTH;
EVP_DecryptUpdate(ctx, aes_decrypted_buffer, &actual_length, (const
    unsigned char*)aes_encrypted_buffer, AES_DATA_LENGTH);
EVP_CIPHER_CTX_free(ctx);
```

Código B.3: Cifragem e decifragem usando *AES* em modo *ECB* pela *wolfSSL*.

```
Aes aes;

// cifra dados.
wc_AesSetKeyDirect(&aes, (const byte *)aes_key, AES_KEY_LENGTH, (const byte
     *)aes_iv, AES_ENCRYPTION);
for (int i = 0; i < AES_DATA_LENGTH; i += AES_BLOCK_LENGTH)
  wc_AesEncryptDirect(&aes, (byte *)&(aes_encrypted_buffer[i]), (const byte
     *)&(aes_in_buffer[i]));

// decifra dados.
wc_AesSetKeyDirect(&aes, (const byte *)aes_key, AES_KEY_LENGTH, (const byte
     *)aes_iv, AES_DECRYPTION);
for (int i = 0; i < AES_DATA_LENGTH; i += AES_BLOCK_LENGTH)
  wc_AesDecryptDirect(&aes, (byte *)&(aes_decrypted_buffer[i]), (const byte
     *)&(aes_encrypted_buffer[i]));
```

Código B.4: Cifragem e decifragem usando *AES* em modo *CBC* pela *GEmSysC*.

```
1  clAESDef(aes_engine, CL_AES_KEYLENGTH_256);
2  clEngineInstanceId id;
3
4  // cifra dados.
5  clAESEncryptCreate(&id, clAES(aes_engine), CL_BLOCK_CIPHER_MODE_CBC, (const
       void*)aes_key, AES_KEY_LENGTH, (const void*)aes_iv, AES_BLOCK_LENGTH);
6  clEngineProcess(id, (const void*)aes_in_buffer, AES_DATA_LENGTH, (void*)
      aes_encrypted_buffer, AES_DATA_LENGTH, NULL);
7  clEngineFinalize(id);
8
9  // decifra dados.
10 clAESDecryptCreate(&id, clAES(aes_engine), CL_BLOCK_CIPHER_MODE_CBC, (const
       void*)aes_key, AES_KEY_LENGTH, (const void*)aes_iv, AES_BLOCK_LENGTH);
11 clEngineProcess(id, (const void*)aes_encrypted_buffer, AES_DATA_LENGTH, (
      void*)aes_decrypted_buffer, AES_DATA_LENGTH, NULL);
12 clEngineFinalize(id);
```

código B.6 para a *wolfSSL*. Nestes códigos, assume-se a existência das seguintes estruturas: `aes_key` (*buffer* com a chave criptográfica), `aes_iv` (*buffer* com o vetor de inicialização), `aes_in_buffer` (*buffer* de entrada), `aes_encrypted_buffer` (*buffer* de saída da cifragem) e `aes_decrypted_buffer` (*buffer* de saída da decifragem). Assume-se que foram declaradas no código as seguintes constantes: `AES_KEY_LENGTH` (tamanho da chave, igual a 16 bytes, ou seja, 256 bits), `AES_BLOCK_LENGTH` (tamanho do bloco, igual a 8 bytes, ou seja, 128 bits) e `AES_DATA_LENGTH` (tamanho dos dados, múltiplo do tamanho do bloco).

Seguem as métricas do código em B.4. É utilizado um tipo de dados da *API*: `clEngineInstanceId`. São utilizadas duas constantes da *API*: `CL_AES_KEYLENGTH_256` e `CL_BLOCK_CIPHER_MODE_CBC`. São utilizadas seis funções ou *macros* parametrizadas da *API*: `clAESDef` (com dois parâmetros e utilizada uma vez), `clAES` (com um parâmetro e utilizada duas vezes), `clAESEncryptCreate` (com sete parâmetros e utilizada uma vez), `clAESDecryptCreate` (com sete parâmetros e utilizada uma vez), `clEngineProcess` (com seis parâmetros e utilizada duas vezes) e `clEngineFinalize` (com um parâmetro e utilizada duas vezes). Considerando as declarações de variáveis, que ocupam duas linhas, este código tem um total de oito linhas (ignorando linhas vazias e comentários). O somatório dos números de parâmetros das de funções ou *macros* parametrizadas é de 32.

Seguem as métricas do código em B.5. É utilizado um tipo de dados da *API*: `EVP_CIPHER_CTX`. Não são utilizadas constantes da *API*. São utilizadas oito funções ou *macros* parametrizadas da *API*: `EVP_CIPHER_CTX_new` (sem parâmetros e utilizada duas vezes), `EVP_EncryptInit_ex` (com cinco parâmetros e utilizada uma vez), `EVP_DecryptInit_ex` (com cinco parâmetros e utilizada uma vez), `EVP_aes_256_cbc` (sem parâmetros e utilizada duas vezes), `EVP_CIPHER_CTX_set_padding` (com dois parâmetros e utilizada duas vezes), `EVP_EncryptUpdate` (com cinco parâmetros e utilizada uma vez), `EVP_DecryptUpdate` (com cinco parâmetros e utilizada uma vez) e `EVP_CIPHER_CTX_free` (com um parâmetro e utilizada duas vezes). Considerando as declarações de variáveis, que ocupam duas linhas, este código tem um total de 14 linhas (igno-

Código B.5: Cifragem e decifragem usando *AES* em modo *CBC* pela *OpenSSL*.

```
1  EVP_CIPHER_CTX * ctx;
2  int actual_length;
3
4  // cifra dados.
5  ctx = EVP_CIPHER_CTX_new();
6  EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, aes_key, aes_iv);
7  EVP_CIPHER_CTX_set_padding(ctx, 0);
8  actual_length = AES_DATA_LENGTH;
9  EVP_EncryptUpdate(ctx, aes_encrypted_buffer, &actual_length, (const
       unsigned char*)aes_in_buffer, AES_DATA_LENGTH);
10 EVP_CIPHER_CTX_free(ctx);
11
12 // decifra dados.
13 ctx = EVP_CIPHER_CTX_new();
14 EVP_DecryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, aes_key, aes_iv);
15 EVP_CIPHER_CTX_set_padding(ctx, 0);
16 actual_length = AES_DATA_LENGTH;
17 EVP_EncryptUpdate(ctx, aes_decrypted_buffer, &actual_length, (const
       unsigned char*)aes_encrypted_buffer, AES_DATA_LENGTH);
18 EVP_CIPHER_CTX_free(ctx);
```

rando linhas vazias e comentários). O somatório dos números de parâmetros das de funções ou *macros* parametrizadas é de 26.

Seguem as métricas do código em B.6. São utilizados dois tipos de dados da *API*: `Aes` e `byte`. São utilizadas duas constantes da *API*: `AES_ENCRYPTION` e `AES_DECRYPTION`. São utilizadas três funções ou *macros* parametrizadas da *API*: `wc_AesSetKey` (com cinco parâmetros e utilizada duas vezes), `wc_AesCbcEncrypt` (com quatro parâmetros e utilizada uma vez) e `wc_AesCbcDecrypt` (com quatro parâmetros e utilizada uma vez). Considerando a declaração de variável, que ocupa uma linha, este código tem um total de cinco linhas (ignorando linhas vazias e comentários). O somatório dos números de parâmetros das de funções ou *macros* parametrizadas é de 18.

Código B.6: Cifragem e decifragem usando *AES* em modo *CBC* pela *wolfSSL*.

```
1  Aes aes;
2
3  // cifra dados.
4  wc_AesSetKey(&aes, (const byte *)aes_key, AES_KEY_LENGTH, (const byte *)
       aes_iv, AES_ENCRYPTION);
5  wc_AesCbcEncrypt(&aes, (byte *)aes_encrypted_buffer, (const byte *)
       aes_in_buffer, AES_DATA_LENGTH);
6
7  // decifra dados.
8  wc_AesSetKey(&aes, (const byte *)aes_key, AES_KEY_LENGTH, (const byte *)
       aes_iv, AES_DECRYPTION);
9  wc_AesCbcDecrypt(&aes, (byte *)aes_decrypted_buffer, (const byte *)
       aes_encrypted_buffer, AES_DATA_LENGTH);
```

Código B.7: Cifragem e decifragem usando *RSA* com preenchimento *PKCS* #1 v1.5 pela *GEmSysC*.

```
clRSADef(rsa_engine, RSA_LENGTH);
clEngineInstanceId id;

// cifra dados.
clRSAPublicEncryptCreate(&id, clRSA(rsa_engine),
    CL_RSA_PADDING_MODE_PKCS1v15, CL_RSA_KEY_ENCODING_DER,
    CL_RSA_KEY_FORMAT_PKCS1, rsa_public_key, RSA_PUBLIC_KEY_LENGTH);
clEngineProcess(id, (const void*)rsa_in_buffer,
    RSA_PKCS1V15_DECRYPTED_DATA_LENGTH, (void*)rsa_encrypted_buffer,
    RSA_LENGTH, NULL);
clEngineFinalize(id);

// decifra dados.
clRSAPrivateDecryptCreate(&id, clRSA(rsa_engine),
    CL_RSA_PADDING_MODE_PKCS1v15, CL_RSA_KEY_ENCODING_DER,
    CL_RSA_KEY_FORMAT_PKCS1, rsa_private_key, RSA_PRIVATE_KEY_LENGTH);
clEngineProcess(id, (const void*)rsa_encrypted_buffer,
    RSA_PKCS1V15_ENCRYPTED_DATA_LENGTH, (void*)rsa_decrypted_buffer,
    RSA_LENGTH, NULL);
clEngineFinalize(id);
```

B.3 *RSA* com preenchimento *PKCS* #1 v1.5

O código que realiza a cifragem e a decifragem dos dados usando *RSA* com preenchimento *PKCS* #1 v1.5 está apresentado no código B.7 para a *GEmSysC*, no código B.8 para a *OpenSSL* e no código B.9 para a *wolfSSL*. Nestes códigos, assume-se a existência das seguintes estruturas: `rsa_public_key` (*buffer* com a chave pública codificada em *DER*), `rsa_private_key` (*buffer* com a chave privada codificada em *DER*), `rsa_in_buffer` (*buffer* de entrada), `rsa_encrypted_buffer` (*buffer* de saída da cifragem) e `rsa_decrypted_buffer` (*buffer* de saída da decifragem). Assume-se que foram declaradas no código as seguintes constantes: `RSA_LENGTH` (tamanho máximo da chave), `RSA_PUBLIC_KEY_LENGTH` (tamanho da chave pública codificada em *DER*), `RSA_PRIVATE_KEY_LENGTH` (tamanho da chave privada codificada em *DER*), `RSA_PKCS1V15_DECRYPTED_DATA_LENGTH` (tamanho dos dados abertos) e `RSA_PKCS1V15_ENCRYPTED_DATA_LENGTH` (tamanho dos dados cifrados).

Seguem as métricas do código em B.7. É utilizado um tipo de dados da *API*: `clEngineInstanceId`. São utilizadas três constantes da *API*: `CL_RSA_PADDING_MODE_PKCS1v15`, `CL_RSA_KEY_ENCODING_DER`, `CL_RSA_KEY_FORMAT_PKCS1`. São utilizadas seis funções ou *macros* parametrizadas da *API*: `clRSADef` (com dois parâmetros e utilizada uma vez), `clRSA` (com um parâmetro e utilizada duas vezes), `clRSAPublicEncryptCreate` (com sete parâmetros e utilizada uma vez), `clRSAPrivateDecryptCreate` (com sete parâmetros e utilizada uma vez), `clEngineProcess` (com seis parâmetros e utilizada duas vezes) e `clEngineFinalize` (com um parâmetro e utilizada duas vezes). Considerando as declarações de variáveis, que ocupam duas linhas, este código tem um total de oito linhas (ignorando linhas vazias e comen-

Código B.8: Cifragem e decifragem usando *RSA* com preenchimento *PKCS* #1 v1.5 pela *OpenSSL*.

```
1  RSA * rsa;
2  const unsigned char * key_ptr = (const unsigned char *)rsa_key_ptr;
3
4  // cifra dados.
5  key_ptr = (const unsigned char *)rsa_public_key;
6  rsa = d2i_RSA_PUBKEY(NULL, &key_ptr, RSA_PUBLIC_KEY_LENGTH);
7  RSA_public_encrypt(RSA_PKCS1V15_DECRYPTED_DATA_LENGTH, (const unsigned char
       *)rsa_in_buffer, (unsigned char *)rsa_encrypted_buffer, rsa,
      RSA_PKCS1_PADDING);
8  RSA_free(rsa);
9
10 // decifra dados.
11 key_ptr = (const unsigned char *)rsa_private_key;
12 rsa = d2i_RSAPrivateKey(NULL, &key_ptr, RSA_PRIVATE_KEY_LENGTH);
13 RSA_private_decrypt(RSA_PKCS1V15_ENCRYPTED_DATA_LENGTH, (const unsigned
      char *)rsa_encrypted_buffer, (unsigned char *)rsa_decrypted_buffer, rsa,
       RSA_PKCS1_PADDING);
14 RSA_free(rsa);
```

tários). O somatório dos números de parâmetros das de funções ou *macros* parametrizadas é de 32.

Seguem as métricas do código em B.8. É utilizado um tipo de dados da *API*: `RSA`. É utilizada uma constante da *API*: `RSA_PKCS1_PADDING`. São utilizadas cinco funções ou *macros* parametrizadas da *API*: `d2i_RSA_PUBKEY` (com três parâmetros e utilizada uma vez), `d2i_RSAPrivateKey` (com três parâmetros e utilizada uma vez), `RSA_public_encrypt` (com cinco parâmetros e utilizada uma vez), `RSA_private_decrypt` (com cinco parâmetros e utilizada uma vez) e `RSA_free` (com um parâmetro e utilizada duas vezes). Considerando as declarações de variáveis, que ocupam duas linhas, este código tem um total de 10 linhas (ignorando linhas vazias e comentários). O somatório dos números de parâmetros das de funções ou *macros* parametrizadas é de 18.

Seguem as métricas do código em B.9. São utilizados três tipos de dados da *API*: `RsaKey`, `RNG` e `byte`. São utilizadas duas constantes da *API*: `WC_RSA_PKCSV15_PAD` e `WC_HASH_TYPE_NONE`. São utilizadas oito funções ou *macros* parametrizadas da *API*: `wc_InitRsaKey` (com dois parâmetros e utilizada duas vezes), `wc_InitRng` (com um parâmetro e utilizada uma vez), `wc_RsaPublicKeyDecode` (com quatro parâmetros e utilizada uma vez), `wc_RsaPrivateKeyDecode` (com quatro parâmetros e utilizada uma vez), `wc_RsaPublicEncrypt_ex` (com 11 parâmetros e utilizada uma vez), `wc_RsaPrivateDecrypt_ex` (com 10 parâmetros e utilizada uma vez), `wc_FreeRng` (com um parâmetro e utilizada uma vez) e `wc_FreeRsaKey` (com um parâmetro e utilizada duas vezes). Considerando as declarações de variáveis, que ocupam duas linhas, este código tem um total de 14 linhas (ignorando linhas vazias e comentários). O somatório dos números de parâmetros das de funções ou *macros* parametrizadas é de 37.

Código B.9: Cifragem e decifragem usando *RSA* com preenchimento *PKCS* #1 v1.5 pela *wolfSSL*.

```
1  RSA * rsa;
2  RNG rng;
3  RsaKey rsa_key;
4  word32 key_offset = 0;
5
6  // cifra dados.
7  wc_InitRng(&rng);
8  wc_InitRsaKey(&rsa_key, NULL);
9  wc_RsaPublicKeyDecode((const byte*)rsa_public_key, &key_offset, &rsa_key,
       RSA_PUBLIC_KEY_LENGTH);
10 wc_RsaPublicEncrypt_ex((const byte *)rsa_in_buffer,
       RSA_PKCS1V15_DECRYPTED_DATA_LENGTH, (byte*)rsa_encrypted_buffer,
       RSA_LENGTH, &rsa_key, &rng, WC_RSA_PKCSV15_PAD, WC_HASH_TYPE_NONE, 0,
       NULL, 0);
11 wc_FreeRng(&rng);
12 wc_FreeRsaKey(&rsa_key);
13
14 // decifra dados.
15 wc_InitRsaKey(&rsa_key, NULL);
16 wc_RsaPrivateKeyDecode((const byte*)rsa_private_key, &key_offset, &rsa_key,
        RSA_PRIVATE_KEY_LENGTH);
17 wc_RsaPrivateDecrypt_ex((const byte *)rsa_encrypted_buffer,
       RSA_PKCS1V15_ENCRYPTED_DATA_LENGTH, (byte*)rsa_decrypted_buffer,
       RSA_LENGTH, &rsa_key, WC_RSA_PKCSV15_PAD, WC_HASH_TYPE_NONE, 0, NULL, 0)
       ;
18 wc_FreeRsaKey(&rsa_key);
```

Código B.10: Cifragem e decifragem usando *RSA* com preenchimento *OAEP* pela *GEmSysC*.

```
clRSADef(rsa_engine, RSA_LENGTH);
clEngineInstanceId id;

// cifra dados.
clRSAPublicEncryptCreate(&id, clRSA(rsa_engine), CL_RSA_PADDING_MODE_OAEP,
    CL_RSA_KEY_ENCODING_DER, CL_RSA_KEY_FORMAT_PKCS1, rsa_public_key,
    RSA_PUBLIC_KEY_LENGTH);
clEngineProcess(id, (const void*)rsa_in_buffer,
    RSA_OAEP_DECRYPTED_DATA_LENGTH, (void*)rsa_encrypted_buffer, RSA_LENGTH,
     NULL);
clEngineFinalize(id);

// decifra dados.
clRSAPrivateDecryptCreate(&id, clRSA(rsa_engine), CL_RSA_PADDING_MODE_OAEP,
     CL_RSA_KEY_ENCODING_DER, CL_RSA_KEY_FORMAT_PKCS1, rsa_private_key,
    RSA_PRIVATE_KEY_LENGTH);
clEngineProcess(id, (const void*)rsa_encrypted_buffer,
    RSA_OAEP_ENCRYPTED_DATA_LENGTH, (void*)rsa_decrypted_buffer, RSA_LENGTH,
     NULL);
clEngineFinalize(id);
```

B.4 *RSA* com preenchimento *OAEP*

O código que realiza a cifragem e a decifragem dos dados usando *RSA* com preenchimento *OAEP* está apresentado no código B.10 para a *GEmSysC*, no código B.11 para a *OpenSSL* e no código B.12 para a *wolfSSL*. Nestes códigos, assume-se a existência das seguintes estruturas: `rsa_public_key` (*buffer* com a chave pública codificada em *DER*), `rsa_private_key` (*buffer* com a chave privada codificada em *DER*), `rsa_in_buffer` (*buffer* de entrada), `rsa_encrypted_buffer` (*buffer* de saída da cifragem) e `rsa_decrypted_buffer` (*buffer* de saída da decifragem). Assume-se que foram declaradas no código as seguintes constantes: `RSA_LENGTH` (tamanho máximo da chave), `RSA_PUBLIC_KEY_LENGTH` (tamanho da chave pública codificada em *DER*), `RSA_PRIVATE_KEY_LENGTH` (tamanho da chave privada codificada em *DER*), `RSA_OAEP_DECRYPTED_DATA_LENGTH` (tamanho dos dados abertos) e `RSA_OAEP_ENCRYPTED_DATA_LENGTH` (tamanho dos dados cifrados).

Seguem as métricas do código em B.10. É utilizado um tipo de dados da *API*: `clEngineInstanceId`. São utilizadas três constantes da *API*: `CL_RSA_PADDING_MODE_OAEP`, `CL_RSA_KEY_ENCODING_DER`, `CL_RSA_KEY_FORMAT_PKCS1`. São utilizadas seis funções ou *macros* parametrizadas da *API*: `clRSADef` (com dois parâmetros e utilizada uma vez), `clRSA` (com um parâmetro e utilizada duas vezes), `clRSAPublicEncryptCreate` (com sete parâmetros e utilizada uma vez), `clRSAPrivateDecryptCreate` (com sete parâmetros e utilizada uma vez), `clEngineProcess` (com seis parâmetros e utilizada duas vezes) e `clEngineFinalize` (com um parâmetro e utilizada duas vezes). Considerando as declarações de variáveis, que ocupam duas linhas, este código tem um total de oito linhas (ignorando linhas vazias e comentários). O somatório dos números de parâmetros das de funções ou *macros* parametrizadas é de 32.

Código B.11: Cifragem e decifragem usando *RSA* com preenchimento *OAEP* pela *OpenSSL*.

```
RSA * rsa;
const unsigned char * key_ptr;

// cifra dados.
key_ptr = (const unsigned char *)rsa_public_key;
rsa = d2i_RSA_PUBKEY(NULL, &key_ptr, RSA_key_ptr_LENGTH);
RSA_public_encrypt(RSA_OAEP_DECRYPTED_DATA_LENGTH, (const unsigned char *)
    rsa_in_buffer, (unsigned char *)rsa_encrypted_buffer, rsa,
    RSA_PKCS1_OAEP_PADDING);
RSA_free(rsa);

// decifra dados.
key_ptr = (const unsigned char *)rsa_key_ptr;
rsa = d2i_RSAPrivateKey(NULL, &key_ptr, RSA_key_ptr_LENGTH);
RSA_private_decrypt(RSA_OAEP_ENCRYPTED_DATA_LENGTH, (const unsigned char *)
    rsa_encrypted_buffer, (unsigned char *)rsa_decrypted_buffer, rsa,
    RSA_PKCS1_OAEP_PADDING);
RSA_free(rsa);
```

Seguem as métricas do código em B.11. É utilizado um tipo de dados da *API*: `RSA`. É utilizada uma constante da *API*: `RSA_PKCS1_OAEP_PADDING`. São utilizadas cinco funções ou *macros* parametrizadas da *API*: `d2i_RSA_PUBKEY` (com três parâmetros e utilizada uma vez), `d2i_RSAPrivateKey` (com três parâmetros e utilizada uma vez), `RSA_public_encrypt` (com cinco parâmetros e utilizada uma vez), `RSA_private_decrypt` (com cinco parâmetros e utilizada uma vez) e `RSA_free` (com um parâmetro e utilizada duas vezes). Considerando as declarações de variáveis, que ocupam duas linhas, este código tem um total de 10 linhas (ignorando linhas vazias e comentários). O somatório dos números de parâmetros das de funções ou *macros* parametrizadas é de 18.

Seguem as métricas do código em B.12. São utilizados três tipos de dados da *API*: `RsaKey`, `RNG` e `byte`. São utilizadas três constantes da *API*: `WC_RSA_OAEP_PAD`, `WC_HASH_TYPE_SHA`, `WC_MGF1SHA1`. São utilizadas oito funções ou *macros* parametrizadas da *API*: `wc_InitRsaKey` (com dois parâmetros e utilizada duas vezes), `wc_InitRng` (com um parâmetro e utilizada uma vez), `wc_RsaPublicKeyDecode` (com quatro parâmetros e utilizada uma vez), `wc_RsaPrivateKeyDecode` (com quatro parâmetros e utilizada uma vez), `wc_RsaPublicEncrypt_ex` (com 11 parâmetros e utilizada uma vez), `wc_RsaPrivateDecrypt_ex` (com 10 parâmetros e utilizada uma vez), `wc_FreeRng` (com um parâmetro e utilizada uma vez) e `wc_FreeRsaKey` (com um parâmetro e utilizada duas vezes). Considerando as declarações de variáveis, que ocupam duas linhas, este código tem um total de 14 linhas (ignorando linhas vazias e comentários). O somatório dos números de parâmetros das de funções ou *macros* parametrizadas é de 37.

B.5 *SHA-256*

O código que calcula o resumo criptográfico de dados usando *SHA-256* está apresentado no código B.13 para a *GEmSysC*, no código B.14 para a *OpenSSL* e no código B.15 para a *wolfSSL*. Nestes códigos, assume-se a existência das seguin-

Código B.12: Cifragem e decifragem usando *RSA* com preenchimento *OAEP* pela *wolfSSL*.

```
RsaKey rsa_key;
RNG rng;
word32 key_offset = 0;

// cifra dados.
wc_InitRsaKey(&rsa_key, NULL);
wc_InitRng(&rng);
wc_RsaPublicKeyDecode((const byte*)rsa_public_key, &key_offset, &rsa_key,
    RSA_PUBLIC_KEY_LENGTH);
wc_RsaPublicEncrypt_ex((const byte *)rsa_in_buffer,
    RSA_OAEP_DECRYPTED_DATA_LENGTH, (byte*)rsa_encrypted_buffer, RSA_LENGTH,
     &rsa_key, &rng, WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL,
    0);
wc_FreeRng(&rng);
wc_FreeRsaKey(&rsa_key);

// decifra dados.
wc_InitRsaKey(&rsa_key, NULL);
wc_RsaPrivateKeyDecode((const byte*)rsa_private_key, &key_offset, &rsa_key,
     RSA_PRIVATE_KEY_LENGTH);
wc_RsaPrivateDecrypt_ex((const byte *)rsa_encrypted_buffer,
    RSA_OAEP_ENCRYPTED_DATA_LENGTH, (byte*)rsa_decrypted_buffer, RSA_LENGTH,
     &rsa_key, WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);
wc_FreeRsaKey(&rsa_key);
```

Código B.13: Resumo criptográfico usando *SHA-256* pela *GEmSysC*.

```
clSHA256Def(sha256_engine);
clEngineInstanceId id;

// resume dados.
clSHA256Create(&id, clSHA256(sha256_engine));
clEngineProcess(id, (const void*)sha256_in_buffer, SHA256_DATA_LENGTH, (
    void*)sha256_hash_buffer, SHA256_HASH_LENGTH, NULL);
clEngineFinalize(id);
```

tes estruturas: `sha256_in_buffer` (*buffer* de entrada), `sha256_hash_buffer` (*buffer* de saída do resumo), `SHA256_DATA_LENGTH` (tamanho dos dados de entrada) e `SHA256_HASH_LENGTH` (tamanho do resumo criptográfico, igual a 16 bytes, ou seja, 256 bits).

Seguem as métricas do código em B.13. É utilizado um tipo de dados da *API*: `clEngineInstanceId`. Não são utilizadas constantes da *API*. São utilizadas cinco funções ou *macros* parametrizadas da *API*: `clSHA256Def` (com um parâmetro e utilizada uma vez), `clSHA256` (com um parâmetro e utilizada uma vez), `clSHA256Create` (com dois parâmetros e utilizada uma vez), `clEngineProcess` (com seis parâmetros e utilizada uma vez) e `clEngineFinalize` (com um parâmetro e utilizada uma vez). Considerando as declarações de variáveis, que ocupam duas linhas, este código tem um total de cinco linhas (ignorando linhas vazias e comentários). O somatório dos números de parâmetros das de funções ou *macros* parametrizadas é de 11.

Código B.14: Resumo criptográfico usando *SHA-256* pela *OpenSSL*.

```
EVP_MD_CTX * ctx;
unsigned int hash_len;

// resume dados.
ctx = EVP_MD_CTX_create();
EVP_DigestInit_ex(ctx, EVP_sha256(), NULL);
EVP_DigestUpdate(ctx, (const void*)sha256_in_buffer, SHA256_DATA_LENGTH);
hash_len = SHA256_HASH_LENGTH
EVP_DigestFinal_ex(ctx, (void*)sha256_hash_buffer, &hash_len);
EVP_MD_CTX_destroy(ctx);
```

Código B.15: Resumo criptográfico usando *SHA-256* pela *wolfSSL*.

```
Sha256 sha256;

// resume dados.
wc_InitSha256(&sha256);
wc_Sha256Update(&sha256, (const byte*)sha256_in_buffer, SHA256_DATA_LENGTH)
    ;
wc_Sha256Final(&sha256, (byte*)sha256_hash_buffer);
```

Seguem as métricas do código em B.14. É utilizado um tipo de dados da *API*: `EVP_MD_CTX`. Não são utilizadas constantes da *API*. São utilizadas seis funções ou *macros* parametrizadas da *API*: `EVP_MD_CTX_create` (sem parâmetros e utilizada uma vez), `EVP_DigestInit_ex` (com três parâmetros e utilizada uma vez), `EVP_sha256` (sem parâmetros e utilizada uma vez), `EVP_DigestUpdate` (com três parâmetros e utilizada uma vez), `EVP_DigestFinal_ex` (com três parâmetros e utilizada uma vez) e `EVP_MD_CTX_destroy` (com um parâmetro e utilizada uma vez). Considerando as declarações de variáveis, que ocupam duas linhas, este código tem um total de oito linhas (ignorando linhas vazias e comentários). O somatório dos números de parâmetros das de funções ou *macros* parametrizadas é de 10.

Seguem as métricas do código em B.15. São utilizados dois tipos de dados da *API*: `Sha256` e `byte`. Não são utilizadas constantes da *API*. São utilizadas três funções ou *macros* parametrizadas da *API*: `wc_InitSha256` (com um parâmetro e utilizada uma vez), `wc_Sha256Update` (com três parâmetros e utilizada uma vez) e `wc_Sha256Final` (com dois parâmetros e utilizada uma vez). Considerando a declaração de variável, que ocupa uma linha, este código tem um total de quatro linhas (ignorando linhas vazias e comentários). O somatório dos números de parâmetros das de funções ou *macros* parametrizadas é de 6.

B.6 Todos os códigos

Se forem considerados os códigos de todas as seções anteriores, é possível ter uma ideia da complexidade de cada uma das *APIs*, tendo em vista as funcionalidades suportadas pela *GEmSysC*. As métricas de cada uma das *APIs* está apresentada a seguir.

Os exemplos de utilização da *GEmSysC* estão apresentados nos códigos B.1, B.4, B.7, B.10 e B.13. Seguem as métricas de todos estes códigos em conjunto. É

utilizado um tipo de dados da *API*: clEngineInstanceId. São utilizadas sete constantes da *API*: CL_AES_KEYLENGTH_256, CL_BLOCK_CIPHER_MODE_ECB, CL_BLOCK_CIPHER_MODE_CBC, CL_RSA_PADDING_MODE_PKCS1v15, CL_RSA_PADDING_MODE_OAEP, CL_RSA_KEY_ENCODING_DER, CL_RSA_KEY_FORMAT_PKCS1. São utilizadas 13 funções ou *macros* parametrizadas diferentes: clAESDef (com dois parâmetros), clAES (com um parâmetro), clAESEncryptCreate (com sete parâmetros), clAESDecryptCreate (com sete parâmetros), clRSADef (com dois parâmetros), clRSA (com um parâmetro), clRSAPublicEncryptCreate (com sete parâmetros), clRSAPrivateDecryptCreate (com sete parâmetros), clSHA256Def (com um parâmetro), clSHA256 (com um parâmetro), clSHA256Create (com dois parâmetros), clEngineProcess (com seis parâmetros) e clEngineFinalize (com um parâmetro).

Os exemplos de utilização da *OpenSSL* estão apresentados nos códigos B.2, B.5, B.8, B.11 e B.14. Seguem as métricas de todos estes códigos em conjunto. São utilizados três tipos de dados da *API*: EVP_CIPHER_CTX, RSA e EVP_MD_CTX. São utilizadas duas constantes da *API*: RSA_PKCS1_PADDING e RSA_PKCS1_OAEP_PADDING. São utilizadas 20 funções ou *macros* parametrizadas da *API*: EVP_CIPHER_CTX_new (sem parâmetros), EVP_EncryptInit_ex (com cinco parâmetros), EVP_DecryptInit_ex (com cinco parâmetros), EVP_aes_256_ecb (sem parâmetros), EVP_aes_256_cbc (sem parâmetros), EVP_CIPHER_CTX_set_padding (com dois parâmetros), EVP_EncryptUpdate (com cinco parâmetros), EVP_DecryptUpdate (com cinco parâmetros), EVP_CIPHER_CTX_free (com um parâmetro), d2i_RSA_PUBKEY (com três parâmetros), d2i_RSAPrivateKey (com três parâmetros), RSA_public_encrypt (com cinco parâmetros), RSA_private_decrypt (com cinco parâmetros), RSA_free (com um parâmetro), EVP_MD_CTX_create (sem parâmetros), EVP_DigestInit_ex (com três parâmetros), EVP_sha256 (sem parâmetros), EVP_DigestUpdate (com três parâmetros), EVP_DigestFinal_ex (com três parâmetros) e EVP_MD_CTX_destroy (com um parâmetro).

Os exemplos de utilização da *wolfSSL* estão apresentados nos códigos B.3, B.6, B.9, B.12 e B.15. Seguem as métricas de todos estes códigos em conjunto. São utilizados cinco tipos de dados da *API*: Aes, RsaKey, RNG, Sha256 e byte. São utilizadas sete constantes da *API*: AES_ENCRYPTION e AES_DECRYPTION, WC_RSA_PKCSV15_PAD, WC_HASH_TYPE_NONE, WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1. São utilizadas 17 funções ou *macros* parametrizadas da *API*: wc_AesSetKeyDirect (com cinco parâmetros), wc_AesEncryptDirect (com três parâmetros), wc_AesDecryptDirect (com três parâmetros), wc_AesSetKey (com cinco parâmetros), wc_AesCbcEncrypt (com quatro parâmetros), wc_AesCbcDecrypt (com quatro parâmetros), wc_InitRsaKey (com dois parâmetros), wc_InitRng (com um parâmetro), wc_RsaPublicKeyDecode (com quatro parâmetros), wc_RsaPrivateKeyDecode (com quatro parâmetros), wc_RsaPublicEncrypt_ex (com 11 parâmetros), wc_RsaPrivateDecrypt_ex (com 10 parâmetros), wc_FreeRng (com um parâmetro), wc_FreeRsaKey (com um parâmetro), wc_InitSha256 (com um parâmetro), wc_Sha256Update (com três parâmetros) e wc_Sha256Final (com dois parâmetros).

B.7 Visão geral

Neste apêndice, foram apresentados os códigos de testes utilizados na avaliação de complexidade da *GEmSysC*. Estes códigos realizam operações simples de processamento de dados, para cada um dos módulos da *GEmSysC*, e códigos correspondentes utilizando as bibliotecas *OpenSSL* e *wolfSSL*. A avaliação dos dados apresentados neste apêndice está apresentada na seção 5.2.1 da dissertação.

Apêndice C

Resultados das métricas de complexidade

Neste apêndice, estão apresentados os resultados das métricas de complexidade calculadas para a *GEmSysC*, para a *OpenSSL* e para a *wolfSSL*. As foram apresentadas e discutidas na seção 5.2.1, e representam a quantidade de tipos de dados, o total de constantes, o número de funções e macros parametrizadas, o total de linhas de código e o total de parâmetros nas funções presentes nos códigos do apêndice B.

As métricas calculadas para cada uma das funcionalidades avaliadas estão apresentadas nas tabelas C.1, C.2, C.3, C.4 e C.5. Como discutido na seção 5.2.1, além de terem sido calculadas métricas para cada uma das funcionalidades, também calcularam-se métricas para todas em conjunto. Estas métricas estão apresentadas na tabela C.6.

Tabela C.1: Comparação entre códigos com *AES* em modo *ECB*.

	GEmSysC	*OpenSSL*	*wolfSSL*
Tipos de dados	1	1	2
Constantes	2	0	2
Funções ou *macros* parametrizadas	6	8	3
Linhas de código	8	14	7
Total de parâmetros nas funções	32	26	16

Tabela C.2: Comparação entre códigos com *AES* em modo *CBC*.

	GEmSysC	*OpenSSL*	*wolfSSL*
Tipos de dados	1	1	2
Constantes	2	0	2
Funções ou *macros* parametrizadas	6	8	3
Linhas de código	8	14	5
Total de parâmetros nas funções	32	26	18

Tabela C.3: Comparação entre códigos com *RSA* com preenchimento *PKCS* #1 v1.5.

	GEmSysC	*OpenSSL*	*wolfSSL*
Tipos de dados	1	1	3
Constantes	3	1	2
Funções ou *macros* parametrizadas	6	5	8
Linhas de código	8	10	14
Total de parâmetros nas funções	32	18	37

Tabela C.4: Comparação entre códigos com *RSA* com preenchimento *OAEP*.

	GEmSysC	*OpenSSL*	*wolfSSL*
Tipos de dados	1	1	3
Constantes	3	1	3
Funções ou *macros* parametrizadas	6	5	8
Linhas de código	8	10	14
Total de parâmetros nas funções	32	18	37

Tabela C.5: Comparação entre códigos com *SHA-256*.

	GEmSysC	*OpenSSL*	*wolfSSL*
Tipos de dados	1	1	2
Constantes	0	0	0
Funções ou *macros* parametrizadas	5	6	3
Linhas de código	5	8	4
Total de parâmetros nas funções	11	10	6

Tabela C.6: Comparação entre todos os códigos.

	GEmSysC	*OpenSSL*	*wolfSSL*
Tipos de dados	1	3	5
Constantes	7	2	7
Funções ou *macros* parametrizadas	13	20	17

Apêndice D

Resultados dos testes de desempenho

Neste apêndice. estão apresentados os resultados dos testes de desempenho descritos no capítulo 5. Na tabela D.1, estão apresentados os resultados dos testes no sistema embarcado, comparando o desempenho da implementação da GEmSysC sobre wolfSSL com a própria wolfSSL sem a camada GEmSysC. Na tabela D.2, estão apresentados os resultados dos testes no PC, comparando o desempenho da implementação da GEmSysC sobre OpenSSL com a própria OpenSSL sem a camada GEmSysC. Tanto em uma tabela quanto na outra, estão apresentados os resultados de três testes independentes. Nas tabelas D.3 e D.4, estão apresentadas as médias, os desvios padrões e os coeficientes de variação dos resultados destes testes.

Tabela D.1: Medidas de desempenho da *wolfSSL* e da *GemSysC* em plataforma *Cortex-M3*.

	Execução #1				Execução #3				Execução #3			
	wolfSSL	*GemSysC*	Diferença	*Overhead* (%)	*wolfSSL*	*GemSysC*	Diferença	*Overhead* (%)	*wolfSSL*	*GemSysC*	Diferença	*Overhead* (%)
Cifragem *AES* com *ECB*	1,478 s	1,478 s	0 ms	0,00%	1,478 s	1,478 s	0 ms	0,00%	1,478 s	1,478 s	0 ms	0,00%
Decifragem *AES* com *ECB*	1,495 s	1,495 s	0 ms	0,00%	1,495 s	1,495 s	0 ms	0,00%	1,495 s	1,495 s	0 ms	0,00%
Cifragem *AES* com *CBC*	1,515 s	1,517 s	2 ms	0,13%	1,515 s	1,517 s	2 ms	0,13%	1,515 s	1,517 s	2 ms	0,13%
Decifragem *AES* com *CBC*	1,536 s	1,538 s	2 ms	0,13%	1,536 s	1,538 s	2 ms	0,13%	1,536 s	1,538 s	2 ms	0,13%
Cifragem *RSA* com *PKCS* #1 v1.5	250,424 s	250,426 s	2 ms	0,00%	250,424 s	250,426 s	2 ms	0,00%	250,424 s	250,426 s	2 ms	0,00%
Decifragem *RSA* com *PKCS* #1 v1.5	3057,225 s	3057,227 s	2 ms	0,00%	3057,225 s	3057,227 s	2 ms	0,00%	3057,225 s	3057,227 s	2 ms	0,00%
Cifragem *RSA* com *OAEP*	251,337 s	251,339 s	2 ms	0,00%	251,337 s	251,339 s	2 ms	0,00%	251,337 s	251,339 s	2 ms	0,00%
Decifragem *RSA* com *OAEP*	3058,734 s	3058,736 s	2 ms	0,00%	3058,734 s	3058,736 s	2 ms	0,00%	3058,734 s	3058,736 s	2 ms	0,00%
Resumo com *SHA-256*	0,595 s	0,596 s	1 ms	0,17%	0,595 s	0,596 s	1 ms	0,17%	0,595 s	0,596 s	1 ms	0,17%

Tabela D.2: Medidas de desempenho da *OpenSSL* e da *GemSysC* em plataforma *x86*.

	Execução #1				Execução #3				Execução #3			
	OpenSSL	*GemSysC*	Diferença	*Overhead* (%)	*OpenSSL*	*GemSysC*	Diferença	*Overhead* (%)	*OpenSSL*	*GemSysC*	Diferença	*Overhead* (%)
Cifragem *AES* com *ECB*	0,340 s	0,342 s	2 ms	0,58%	0,340 s	0,342 s	2 ms	0,59%	0,340 s	0,342 s	2 ms	0,59%
Decifragem *AES* com *ECB*	0,422 s	0,423 s	1 ms	0,24%	0,422 s	0,423 s	1 ms	0,24%	0,421 s	0,423 s	2 ms	0,48%
Cifragem *AES* com *CBC*	0,315 s	0,316 s	1 ms	0,32%	0,315 s	0,317 s	2 ms	0,63%	0,315 s	0,317 s	2 ms	0,63%
Decifragem *AES* com *CBC*	0,402 s	0,403 s	1 ms	0,25%	0,402 s	0,403 s	1 ms	0,25%	0,402 s	0,403 s	1 ms	0,25%
Cifragem *RSA* com *PKCS* #1 v1.5	12,460 s	12,466 s	6 ms	0,05%	12,462 s	12,467 s	5 ms	0,04%	12,461 s	12,469 s	8 ms	0,06%
Decifragem *RSA* com *PKCS* #1 v1.5	482,414 s	482,830 s	416 ms	0,09%	482,485 s	482,894 s	409 ms	0,08%	482,474 s	482,829 s	355 ms	0,07%
Cifragem *RSA* com *OAEP*	12,804 s	12,810 s	6 ms	0,05%	12,805 s	12,807 s	2 ms	0,02%	12,806 s	12,808 s	2 ms	0,02%
Decifragem *RSA* com *OAEP*	482,861 s	483,214 s	353 ms	0,07%	483,116 s	483,414 s	298 ms	0,06%	483,163 s	483,508 s	345 ms	0,07%
Resumo com *SHA-256*	0,285 s	0,289 s	4 ms	1,38%	0,286 s	0,290 s	4 ms	1,40%	0,285 s	0,289 s	4 ms	1,40%

Tabela D.3: Média, desvio padrão e coeficiente de variação das medidas de desempenho da *wolfSSL* e da *GemSysC* em plataforma *Cortex-M3*.

	wolfSSL			*GemSysC*			*Diferença*			*Overhead* (%)
	Média	Desvio padrão	Coeficiente de variação	Média	Desvio padrão	Coeficiente de variação	Média	Desvio padrão	Coeficiente de variação	Média
Cifragem *AES* com *ECB*	1,478 s	0 ms	0,00%	1,478 s	0 ms	0,00%	0 ms	0 ms	-	>0,00%
Decifragem *AES* com *ECB*	1,495 s	0 ms	0,00%	1,495 s	0 ms	0,00%	0 ms	0 ms	-	>0,00%
Cifragem *AES* com *CBC*	1,515 s	0 ms	0,00%	1,517 s	0 ms	0,00%	2 ms	0 ms	0,00%	0,13%
Decifragem *AES* com *CBC*	1,536 s	0 ms	0,00%	1,538 s	0 ms	0,00%	2 ms	0 ms	0,00%	0,13%
Cifragem *RSA* com *PKCS* #1 v1.5	250,424 s	0 ms	0,00%	250,426 s	0 ms	0,00%	2 ms	0 ms	0,00%	>0,00%
Decifragem *RSA* com *PKCS* #1 v1.5	3057,225 s	0 ms	0,00%	3057,227 s	0 ms	0,00%	2 ms	0 ms	0,00%	>0,00%
Cifragem *RSA* com *OAEP*	251,337 s	0 ms	0,00%	251,339 s	0 ms	0,00%	2 ms	0 ms	0,00%	>0,00%
Decifragem *RSA* com *OAEP*	3058,734 s	0 ms	0,00%	3058,736 s	0 ms	0,00%	2 ms	0 ms	0,00%	>0,00%
Resumo com *SHA-256*	0,595 s	0 ms	0,00%	0,596 s	0 ms	0,00%	1 ms	0 ms	0,00%	0,17%

Tabela D.4: Média, desvio padrão e coeficiente de variação das medidas de desempenho da *OpenSSL* e da *GemSysC* em plataforma *x86*.

	OpenSSL			*GemSysC*			*Diferença*			*Overhead* (%)
	Média	Desvio padrão	Coeficiente de variação	Média	Desvio padrão	Coeficiente de variação	Média	Desvio padrão	Coeficiente de variação	Média
Cifragem *AES* com *ECB*	0,340 s	0 ms	0,00%	0,342 s	0 ms	0,00%	2 ms	0 ms	0,00%	0,59%
Decifragem *AES* com *ECB*	0,422 s	1 ms	0,14%	0,423 s	0 ms	0,00%	1 ms	1 ms	43,30%	0,32%
Cifragem *AES* com *CBC*	0,315 s	0 ms	0,00%	0,317 s	1 ms	0,18%	2 ms	1 ms	34,64%	0,53%
Decifragem *AES* com *CBC*	0,402 s	0 ms	0,00%	0,403 s	0 ms	0,00%	1 ms	0 ms	0,00%	0,25%
Cifragem *RSA* com *PKCS* #1 v1.5	12,461 s	1 ms	0,01%	12,467 s	2 ms	0,01%	6 ms	2 ms	24,12%	0,05%
Decifragem *RSA* com *PKCS* #1 v1.5	482,458 s	38 ms	0,01%	482,851 s	37 ms	0,01%	393 ms	33 ms	8,49%	0,08%
Cifragem *RSA* com *OAEP*	12,805 s	1 ms	0,01%	12,808 s	2 ms	0,01%	3 ms	2 ms	69,28%	0,03%
Decifragem *RSA* com *OAEP*	483,047 s	163 ms	0,03%	483,379 s	150 ms	0,03%	332 ms	30 ms	8,95%	0,07%
Resumo com *SHA-256*	0,285 s	1 ms	0,20%	0,289 s	1 ms	0,20%	4 ms	0 ms	0,00%	1,40%