# Capítulo 2

# Criptografia Assimétrica para Programadores – Evitando Outros Maus Usos da Criptografia em Sistemas de Software

Alexandre Braga (UNICAMP) e Ricardo Dahab (UNICAMP)

***Abstract***

*The widespread misuse of cryptography in software systems is the most frequent source of cryptography-related security problems. Several misuses of cryptography have been found to be recurrent in software in general, resulting in vulnerabilities exploitable in real attacks. There is a huge gap between what cryptologists see as misuses of cryptography and what developers see as unsafe use of cryptographic technology. This chapter contributes to fill this gap by addressing the programmatic use of asymmetric (public key) cryptography by software developers with little or no experience in information security and cryptography. The text is introductory and aims at showing to software programmers, through actual examples and code snippets, the gooduses and misuses of asymmetric cryptography and facilitate further studies.*

***Resumo***

*O mau uso generalizado da criptografia em sistemas software é a fonte mais frequente de problemas de segurança relacionados à criptografia. Diversos maus usos de criptografia já são considerados recorrentes em softwares em geral, resultando em vulnerabilidades exploráveis em ataques reais. Percebe-se uma grande lacuna entre o que os criptologistas veem como maus usos de criptografia e aquilo que os desenvolvedores veem como uso inseguro da tecnologia criptográfica. Este texto contribui para preencher essa lacuna, abordando a utilização programática de criptografia assimétrica (de chave pública) por desenvolvedores de software com pouca ou nenhuma experiência em segurança da informação e criptografia. O texto é introdutório e tem o objetivo de mostrar aos programadores de software, por meio de exemplos reais e trechos de código, os bons e maus usos da criptografia assimétrica e, assim, facilitar o aprofundamento em estudos futuros.*

## 2.1. Introdução

Ao longo dos anos, estudos [Anderson 1993, Schneier 1998, Gutmann 2002, Marlinspike 2009] têm revelado que vulnerabilidades em softwares criptográficos são causadas principalmente por defeitos de implementação e pela má gestão de parâmetros criptográficos. Assim, parece ser mais fácil e prático para os atacantes cibernéticos procurarem por falhas não apenas nas implementações em software de algoritmos criptográficos, mas também, e às vezes principalmente, nas camadas de software que encapsulam, circundam ou utilizam as implementações criptográficas, em vez de tentarem encontrar falhas nos algoritmos criptográficos.

Atualmente, o desenvolvimento de sistemas de software é caraterizado por ecossistemas de aplicativos maciçamente disponíveis, desenvolvidos, implantados e utilizados de modo distribuído, por uma população geograficamente dispersa, mas unida socialmente por interesses em comum. Neste ambiente, a criptografia surge muitas vezes como uma tecnologia habilitadora de relações (sociais e de negócios), influenciando a lógica das aplicações, resultando, por um lado, em aplicações criptograficamente seguras e, por outro lado, em maus usos criptográficos cada vez mais comuns. Sabe-se, por vários trabalhos relacionados [Akhawe et al. 2013, Alashwali 2013, Egele et al. 2013, Georgiev et al. 2012, Fahl et al. 2012, Shuai et al. 2014], da presença recorrente de diversas práticas ruins de criptografia em softwares diversos. Possivelmente, estas vulnerabilidades criptográficas foram incluídas sem intenção.

Hoje, percebe-se o aumento de interesse, tanto da indústria [Bleichenbacher et al. 2017] quanto da academia [IACR 2012, Braga 2017, Acar et al. 2017], pelos aspectos práticos, do mundo real, relacionados aos maus usos da tecnologia criptográfica que levam a vulnerabilidades exploráveis em sistemas de software. À medida que a segurança dos sistemas de software torna-se transparente para os usuários e a criptografia de qualidade está disponível para todos os desenvolvedores de software, a fonte mais comum de vulnerabilidades deixa de ser a infra-estrutura criptográfica, para se tornar o software em torno desta, escrito por desenvolvedores não especialistas no assunto. Por isto, o mau uso generalizado da criptografia em software é, possivelmente, a fonte mais frequente de problemas de segurança relacionados à criptografia [Braga 2017].

Quando estudamos [Braga 2017] como a criptografia pode ser mal utilizada em Java, mesmo com o uso correto da API criptográfica, resultando em vulnerabilidades identificáveis diretamente no código fonte, as seguintes observações emergiram empiricamente:

- Apenas um quarto das ferramentas de verificação de softwares criptográficos pode ser usado por programadores não especialistas em criptografia [Braga and Dahab 2015a].
- O mau uso de criptografia é muito frequente em comunidades de programação online, com padrões recorrentes de mau uso de APIs criptográficas [Braga and Dahab 2016].
- As ferramentas de análise estática de código fonte são capazes de detectar apenas pouco mais de um terço dos maus usos criptográficos catalogados [Braga et al. 2017a].
- Os desenvolvedores podem aprender a usar APIs criptográficas sem realmente aprender criptografia, enquanto alguns maus usos criptográficos persistem ao longo do tempo e independem da experiência dos desenvolvedores com as APIs [Braga and Dahab 2017].

Neste contexto, este novo capítulo dá continuidade ao texto anterior intitulado "Introdução à criptografia para programadores" [Braga and Dahab 2015b], desta vez com o objetivo de mostrar aos programadores de software os bons e maus usos da criptografia assimétrica, por meio de exemplos reais, contra-exemplos, trechos de código e programas ilustrativos em Java.

Este capítulo busca atender à demanda crescente por material didático voltado para a utilização correta de implementações criptográficas por programadores não especialistas em criptografia. O texto aborda a utilização programática de criptografia assimétrica por desenvolvedores de software com pouca experiência em segurança da informação e criptografia. Java é a linguagem de programação escolhida porque possui uma API padronizada e estável [Oracle a] que vem sendo usada por desenvolvedores de software por muito tempo, tendo sido adotada também pela plataforma Android [Google ].

O escopo deste texto é a utilização correta de implementações criptográficas prontas, não cobrindo a implementação de algoritmos criptográficos. Ainda, o texto não oferece um tratamento exaustivo ao tema da programação criptográfica; por outro lado, ele tem a finalidade de fomentar o interesse pelo assunto e ajudar na formação de profissionais qualificados tanto na academia quanto na indústria. Todo o código fonte utilizado neste texto e no texto anterior [Braga and Dahab 2015b] pode ser encontrado no repositório indicado pela URL:

`https://bitbucket.org/alexmbraga/crypto4developers`

O restante do texto está organizado da seguinte forma. A Seção 2.2 revisita os conceitos e serviços criptográficos relacionados à criptografia assimétrica. A Seção 2.3 ilustra, com programas em Java, casos de uso comuns da criptografia assimétrica. A Seção 2.4 explica de forma programática os maus hábitos de programação insegura associados à criptografia assimétrica. A Seção 2.5 aborda as configurações inseguras do TLS do ponto de vista dos maus usos criptográficos. A Seção 2.6 conclui o capítulo.

## 2.2. Conceitos de criptografia assimétrica

Esta seção revisita os conceitos de serviços criptográficos assimétricos necessários para o entendimento do restante do capítulo. Os seguintes assuntos são tratados: criptografia de chave pública, encriptação assimétrica para sigilo e não-repúdio (ou irrefutabilidade) de mensagens integras e autênticas com assinaturas digitais. A seção também oferece exemplos de sistemas criptográficos assimétricos como RSA, acordos de chaves com Diffie-Hellman (DH), Criptografia de Curvas Elípticas (*Elliptic Curve Cryptography - ECC*), distribuição de chaves públicas com certificação digital e noções gerais sobre o protocolo *Transport Layer Security* (TLS).

### 2.2.1. Criptografia de chave pública

Há dois tipos de sistemas criptográficos que são comumente conhecidos como criptografia de chave secreta (ou simétrica) e criptografia de chave pública (ou assimétrica). Este texto trata a criptografia assimétrica (de chave publica), enquanto que a criptografia simétrica (de chave secreta) foi discutida em um texto anterior [Braga and Dahab 2015b]. Brevemente, na criptografia de chave secreta, uma única chave (um segredo compartilhado) é usada para encriptar e decriptar a informação. A criptografia de chave pública utiliza duas chaves, que são relacionadas matematicamente e construídas para trabalharem juntas. Uma das chaves do par é dita a chave privada (pessoal) e é mantida em segredo, sendo conhecida apenas pelo dono do par de chaves. A outra chave do par é dita a chave pública, porque é conhecida publicamente.

Devido à maior complexidade algébrica das operações matemáticas envolvidas, a criptografia de chave pública tradicional (associada ao algoritmo RSA) possui em geral um desempenho inferior se comparada à criptografia de chave secreta.

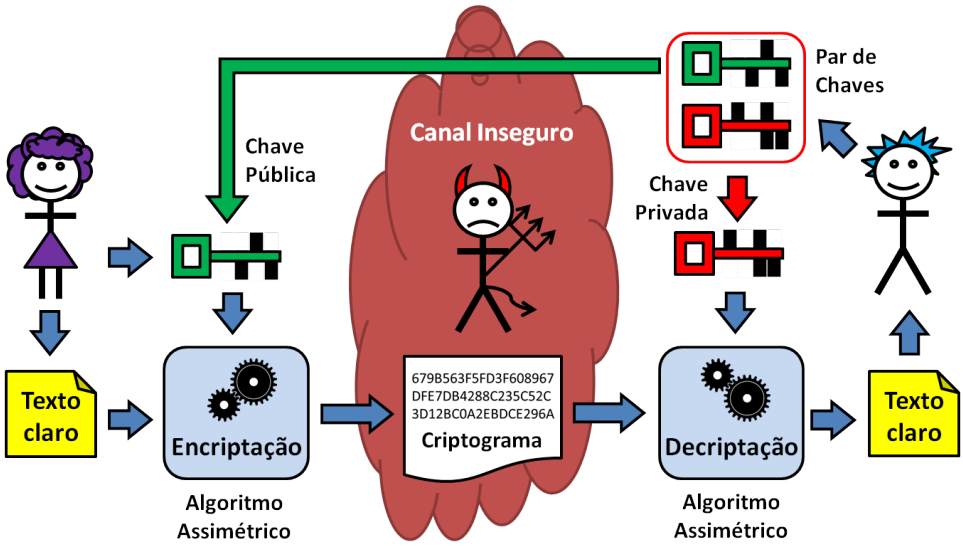


**Figura 2.1. Sistema criptográfico assimétrico para sigilo (de [Braga and Dahab 2015b]).**

### 2.2.2. Encriptação assimétrica para sigilo

A criptografia de chave pública pode ser usada para obter sigilo. Neste caso, a encriptação com a chave pública torna possível que qualquer um envie criptogramas (textos cifrados) para o dono da chave privada. A Figura 2.1 ilustra um sistema criptográfico assimétrico para sigilo e seus elementos básicos. Na figura, Ana, Beto e Ivo são os personagens. As mensagens de Ana para Beto são transmitidas por um canal inseguro, controlado por Ivo. Beto possui um par de chaves, uma chave pública e outra privada. Ana conhece a chave pública de Beto, mas somente o dono do par de chaves (Beto) conhece a chave privada (não há segredo compartilhado). A Figura 2.1 contém os passos a seguir:

1. Ana configura o algoritmo de encriptação com a chave pública de Beto;
2. Ana alimenta o algoritmo com a mensagem original (o texto claro);
3. O texto claro é encriptado e transmitido para Beto pelo canal inseguro;
4. Beto configura o algoritmo de decriptação com a sua própria chave privada;
5. O criptograma é decriptado e o texto claro original é finalmente obtido por Beto.

Analisando a Figura 2.1, observa-se que Ana envia uma mensagem privada para Beto. Para fazer isso, Ana encripta a mensagem usando a chave pública de Beto. Ana conhece a chave pública de Beto, porque ela foi divulgada por Beto. Porém, o criptograma só pode ser decriptado pela chave privada de Beto; nem Ana pode fazê-lo. Para obter comunicação segura bidirecional, basta acrescentar ao sistema criptográfico o mesmo processo no sentido oposto (de Beto para Ana), com outro par de chaves. Isto é, Beto usa a chave pública de Ana para enviar mensagens encriptadas para ela. Ana, ao receber a mensagem, usa sua própria chave privada pessoal para decriptar a mensagem enviada por Beto. A criptografia de chave pública é indispensável para o sigilo e a privacidade na Internet, pois torna possível a comunicação privada em uma rede pública.

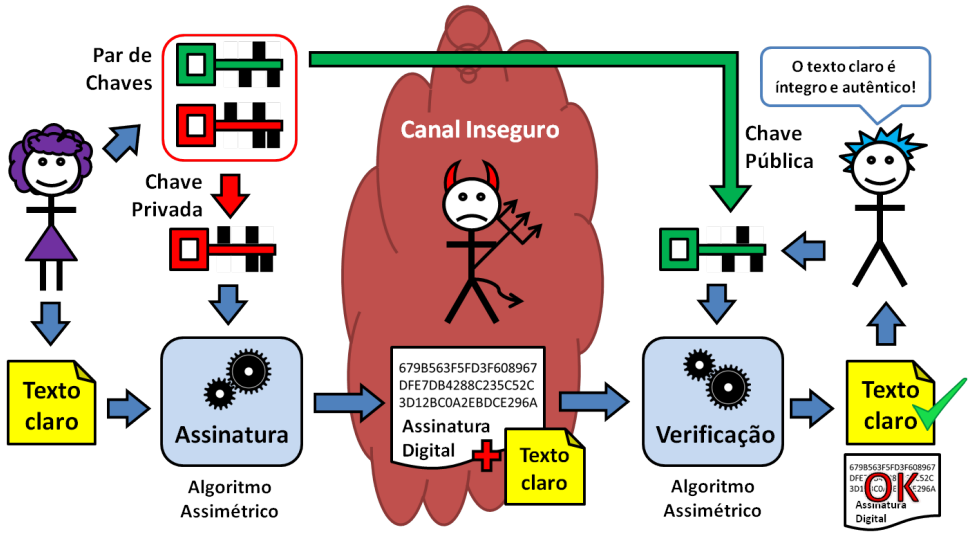


**Figura 2.2. Sist. criptográfico assimétrico para autenticação (de [Braga and Dahab 2015b]).**

### 2.2.3. Não-repúdio de mensagens integras e autênticas

A criptografia de chave pública é a base para outros dois serviços criptográficos: a autenticação das partes comunicantes e a verificação de integridade das mensagens. O procedimento operacional de utilização da criptografia de chave pública para autenticação de mensagens, em certa medida, é o oposto do uso para sigilo. A criptografia de chave pública para assinatura digital é usada para obter integridade, autenticidade e irrefutabilidade.

Chama-se assinatura digital ao resultado de uma certa operação criptográfica com a chave privada sobre o texto claro. Neste caso, o dono da chave privada pode gerar mensagens assinadas, que podem ser verificadas por qualquer um que conheça a chave pública correspondente, portanto, sendo capaz de verificar a autenticidade da assinatura digital. Do ponto de vista das implementações criptográficas subjacentes, nem sempre a operação de assinatura é uma encriptação e nem a sua verificação é sempre uma decriptação.

Visto que qualquer um de posse da chave pública pode "decriptar" a assinatura digital, ela não é mais secreta, mas possui outra propriedade: a irrefutabilidade. Isto é, quem quer que verifique a assinatura com a chave pública, sabe que ela foi produzida por uma chave privada exclusiva; logo, a mensagem não pode ter sido gerada por mais ninguém além do proprietário da chave privada.

Na Figura 2.2, Ana usa sua chave privada para assinar digitalmente uma mensagem para Beto. O texto claro e a assinatura digital são enviados por um canal inseguro (sem sigilo) e podem ser lidos por todos, por isso a mensagem não é secreta. Qualquer um que conheça a chave pública de Ana (todo mundo, inclusive Beto), pode verificar a assinatura digital. Ivo pode ler a mensagem, mas não consegue falsificá-la de maneira indetectável, pois não conhece a chave privada de Ana.

### 2.2.4. Exemplos de sistemas criptográficos assimétricos

Esta subseção descreve dois sistemas criptográficos assimétricos em suas formas clássicas, como descritos nos livros textos: o algoritmo RSA canônico e o acordo de chaves Diffie-Hellman sem autenticação. Além disso, a criptografia de curvas elípticas é tratada do ponto de vista funcional. A subseção termina mostrando uma comparação entre os níveis de segurança dos criptossistemas assimétricos.

#### 2.2.4.1. O algoritmo RSA canônico

Tradicionalmente, o algoritmo criptográfico assimétrico mais conhecido e utilizado para encriptação é o RSA, cujo nome é formado pelas letras iniciais dos sobrenomes dos autores Ron Rivest, Adi Shamir e Leonard Adleman. O RSA foi publicado em 1978 [Rivest et al. 1978]. Um par de chaves do RSA é gerado da seguinte forma: (*i*) Dois números primos muito grandes, $p$ e $q$, são escolhidos aleatoriamente, com $(p \neq q)$, para calcular $n = p \times q$. (*ii*) Um inteiro $e$ é escolhido tal que $1 < e < \phi(n)$, onde $e$ é coprimo de $\phi(n)$, com $\phi(n) = (p-1) \times (p-1)$. (*iii*) O inteiro $d$ é calculado tal que seja o inverso multiplicativo de $e$ ($d \times e \equiv 1 \bmod \phi(n)$). Assim, a chave pública é o par de números $(e,n)$ e a chave privada é o par $(d,n)$.

A encriptação com o RSA é realizada com a chave pública pela fórmula $c = m^e \bmod n$, em que $m$ é o texto claro e $c$ é o criptograma. A decriptação é obtida com a chave privada pela fórmula $m = c^d \bmod n$. A operação de assinatura digital é obtida com a chave privada pela fórmula $s = m^d \bmod n$, onde $m$ é a mensagem e $s$ é a assinatura. A verificação é realizada com a chave pública pela fórmula $m' = s^e \bmod n$, quando $m' = m$.

A segurança do RSA é baseada na dificuldade em se fatorar $n$ em seus fatores primos quando $p$ e $q$ são muito grandes, hoje em dia, com no mínimo 1024 bits cada (*Integer Factorization Problem* - IFP). Um valor comumente usado para $e$ é $2^(16)+1 = 65537$. Várias aplicações usam valores pequenos de $e$ (tais como 3, 5 ou 35) para aumentar o desempenho da cifração e a verificação de assinaturas. Na prática, esta implementação canônica nunca deve ser utilizada. Implementações com preenchimento e aleatorização devem ser preferidas.

Para promover segurança e interoperabilidade, o uso e a implementação do RSA devem obedecer aos padrões internacionais. O documento PKCS#1v2 [Jonsson and Burt Kaliski 2003] especifica o *Optimal Asymmetric Encryption Padding* (OAEP) como um mecanismo de preenchimento (*padding*), que transforma o RSA em um mecanismo de encriptação assimétrica aleatorizado chamado RSA-OAEP. Já o *Probabilistic Signature Scheme* (PSS) é um esquema de assinaturas digitais probabilísticas com RSA (RSA-PSS) também padronizado no PKCS#1v2 [Jonsson and Burt Kaliski 2003] para substituir o esquema de assinatura do PKCS#1v1, considerado inseguro.

#### 2.2.4.2. Acordos de chaves com Diffie-Hellman (DH)

Há ocasiões em que entidades que nunca tiveram a oportunidade de compartilhar chaves criptográficas (por exemplo, nunca se encontram, não se conhecem ou querem permanecer anônimas) precisam se comunicar em sigilo. Nestes casos, uma chave efêmera, usada apenas para algumas encriptações e decriptações decorrentes de uma conversa, pode ser gerada momentos antes do

início da conversa. Os métodos de acordo de chaves são utilizados para combinar ou negociar uma chave secreta entre dois ou mais participantes usando um canal público. Uma característica interessante destes métodos é que o segredo compartilhado (a partir do qual a chave será derivada) é combinado pela troca de informações públicas por meio de um canal inseguro.

O protocolo de acordo de chaves Diffie-Hellman (DH) foi publicado em 1976 no artigo que lançou a criptografia de chave pública [Diffie and Hellman 1976]. Com o DH, o acordo de um segredo compartilhado entre duas parte, Alice e Bob, ocorre da seguinte forma. (*i*) Alice escolhe um primo grande $p$ e um gerador $g$) do corpo finito $F_p$. (*ii*) Alice compartilha estes valores com Bob. (*iii*) Alice escolhe um número aleatório secreto $a$ e calcula $A = g^a \bmod p$. (*iv*) Bob também escolhe um número aleatório secreto $b$ e calcula $B = g^b \bmod p$. (*v*) Alice e Bob trocam entre si os valores $A$ e $B$. (*vi*) Alice calcula $B^a \bmod p$ enquanto Bob calcula $A^b \bmod p$. Pelas propriedades comutativas da exponenciação, $B^a = (g^b)^a = (g^a)^b = A^b = s$, onde $s$ é o segredo compartilhado entre Alice e Bob.

A segurança do DH vem da dificuldade do adversário em resolver o problema Diffie-Hellman, variação do problema do logaritmo discreto (*Discrete logarithm Problem* - DLP): calcular o valor $(g^a)^b$ para inteiros muitos grandes e sen conhecer $a$ e nem $b$. Atualmente, um primo $p$ seguro (*safe prime*) tem pelo menos 2048 bits e deve obedecer a propriedade $p = 2q - 1$, onde $q$ também é primo. Os parâmetros $g$ e $p$ podem ser estáticos (reusados em várias execuções do protocolo) ou dinâmicos (descartáveis ou efêmeros). Além disso, para evitar ataques de personificação causados por um homem no meio (*man-in-the-middle attack*) que se passa tanto por Alice quanto por Bob, apenas o DH autenticado deve ser usado.

Finalmente, o segredo compartilhado $s$ não deve ser usado diretamente como chave secreta, mas sim ser usado na derivação de uma chave secreta segura, por exemplo, com o auxilio de uma função de resumo criptográfico.

#### 2.2.4.3. Criptografia de Curvas Elípticas

Esta subseção oferece uma breve descrição sobre como a criptografia de curvas elíticas é usada na prática [Bos et al. 2014, Mart and Hern 2013], do ponto de vista de uma API criptográfica. O leitor interessado em aprofundar os estudos nos aspectos matemáticos e de implementação interna, deve procurar a literatura especializada [Menezes et al. 1996, Hankerson et al. 2004].

Os sistemas criptográficos baseados em curvas elípticas foram propostos por volta de 1985 de forma independente por dois pesquisadores [Miller 1985, Koblitz 1987], e se baseiam na dificuldade em se calcular o logaritmo discreto de um número inteiro longo sobre a estrutura algébrica de uma curva elíptica. Este problema é conhecido como *Elliptic Curve Discrete Logarithm Problem* (ECDLP). Em geral, a curva elíptica tem a forma $y^2 = x^3 + ax + b$ definida sobre um corpo finito primo $F_p$ ou binário $F_{2^m}$. O ECDLP é considerado mais difícil que o IFP e o DLP, associados ao RSA, ao DH e ao DSA. Por este motivo, o tamanho das chaves criptográficas nos criptossistemas de curvas elípticas é consideravelmente menor que no RSA ou no DH.

Aplicações da criptografia de curvas elípticas incluem troca de chaves, assinaturas digitais e encriptação. Os esquemas e protocolos criptográficos sobre curvas elípticas mais comumente utilizados atualmente são o protocolo de acordo de chaves *Elliptic Curve Diffie Hellman* (ECDH), o esquema de assinaturas digitais *Elliptic Curve Digital Signature Algo-*

**Tabela 2.1. Níveis de segurança e tamanhos de chave (de [BlueKrypt ]).**

| Nível de segurança | Fatoração (IFP) | DLP | | Curva Elíptica | Hash | |
|---|---|---|---|---|---|---|
| | | chave | Corpo | | Assinatura | KDF/HMAC/PRNG |
| 80 | 1024 | 160 | 1024 | 160–163 | SHA1 (160) | – |
| 112 | 2048 | 224 | 2048 | 224–233 | SHA-224/512 SHA3-224 | – |
| 128 | 3072 | 256 | 3072 | 256–283 | SHA-256/512 SHA3-256 | SHA1(160) |
| 192 | 7680 | 384 | 7680 | 384–409 | SHA-384 SHA3-384 | SHA-224 SHA-512 |
| 256 | 15360 | 512 | 15360 | 512–571 | SHA-512 SHA3-512 | SHA-256/384/512 SHA-512/SHA3-512 |

*rithm* (ECDSA) [NIST 2013] e a encriptação *Elliptic Curve Integrated Encryption Scheme* (ECIES) [Hankerson et al. 2004].

O NIST [NIST 2013] recomenda cinco curvas elípticas sobre corpos primos para serem utilizadas no ECDSA, em cinco níveis de segurança, a saber: P-192, P-224, P-256, P-384 e P-521 (também conhecidas [SEC 2010] como secp192r1, secp224r1, secp256r1, secp384r1, secp521r1). Existe ainda uma variedade de outras curvas definidas por diversas instituições. Muitas já são consideradas inseguras para os padrões atuais enquanto outras emergem como padrões de fato em substituição a padrões obsoletos.

Em se tratando de padrões emergentes, a curva 25519 [Bernstein 2006] tem sido considerada um novo padrão de fato na indústria de segurança criptográfica por ser uma boa substituta às curvas padronizadas pelo NIST [NIST 2013], porque é mais rápida, possivelmente mais segura, e alegadamente sem a influência de agências de governo sobre seu projeto. O seu autor, Daniel Bernstein, mantém um site sobre a curva `https://cr.yp.to/ecdh.html`. Infelizmente, nenhum dos provedores criptográficos Java adotados neste texto oferece a curva 25599 diretamente pela API.

#### 2.2.4.4. Comparação de tamanhos de chave e níveis de segurança

Sistemas criptográficos implementados em software geralmente combinam diversas funções criptográficas, cujas configurações devem ser escolhidas de modo a manter as funções no mesmo nível de segurança. A Tabela 2.1 mostra uma comparação entre os níveis de segurança de tipos de primitivas criptográficas e os tamanhos de chaves correspondentes. O nível de segurança é uma medida relativa e pode ser interpretado como sendo a força de um algoritmo criptográfico simétrico com chave de $n$ bits.

A tabela foi adaptada do web site `keylength.com` e considera as recomendações do NIST(2016) [BlueKrypt ]. Cada linha da tabela representa um nível de segurança, indo de 80 bits (para sistemas legados) até 256 bits. Por exemplo, no nível de 80 bits, atualmente útil apenas para sistemas legados, um RSA de 1024 bits é comparável ao DH de 1024 bits com chave de 160 bits, uma curva elíptica de 160 bits e hashes de 160 bits também. Já no nível de segurança mais alto, 256 bits, o RSA se torna inviável na prática com chaves de 15360 bits, enquanto as curvas elípticas despontam com chaves de 512 bits e hashes de 512 bits para assinaturas e de até 384 bits para geração de chaves.

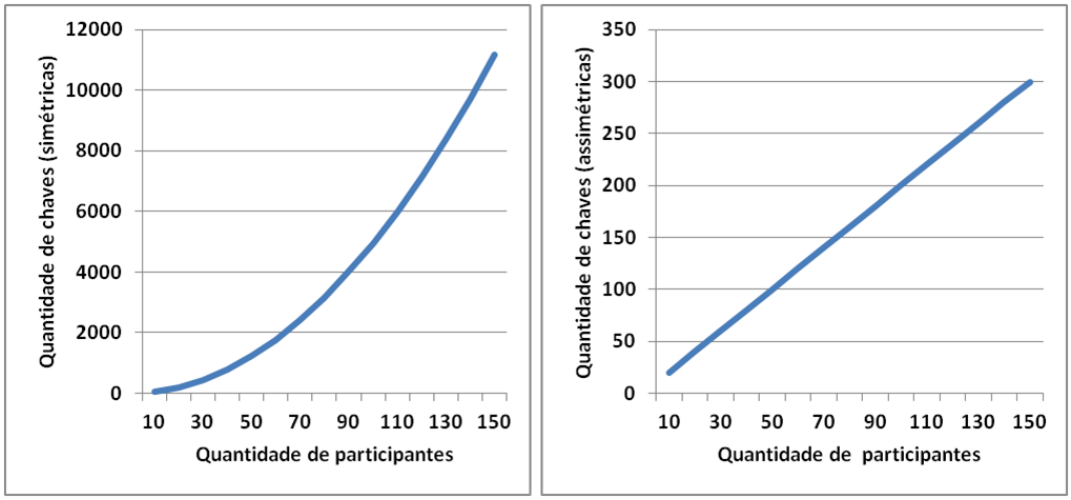


**Figura 2.3. Quantidades de chaves assimétricas e simétricas para até 150 participantes.**

### 2.2.5. Distribuição de chaves públicas com certificação digital

A criptografia assimétrica de chaves públicas tem a propriedade de simplificar o problema de distribuição de chaves criptográficas. Por exemplo, cada um dos participantes do sistema criptográfico só precisa ter o seu par de chaves, manter em segredo a sua chave privada e divulgar a chave pública. Pode haver um repositório global para todas as chaves públicas ou um mecanismo acessível para divulgá-las. A Figura 2.3 ilustra a distribuição de chaves públicas para um grupo de quatro indivíduos. O gráfico da direita na Figura 2.3 mostra o crescimento da quantidade de chaves (públicas e privadas) como uma função afim da quantidade de participantes. A quantidade de chaves é o dobro da quantidade de participantes. Já a distribuição de chaves simétricas (gráfico da esquerda) obedece a uma função quadrática.

O principal problema administrativo associado a distribuição de chaves públicas é justamente a confiança depositada na chave distribuída publicamente. Se a chave pública de alguém pode ser facilmente encontrada em qualquer lugar, então é difícil assegurar com alto grau de certeza que esta chave não foi corrompida ou substituída. O problema de garantir a integridade e a autenticidade da chave pública é muito importante e, se não for solucionado satisfatoriamente, pode comprometer a confiança no sistema criptográfico inteiro. Uma maneira de validar chaves públicas é fazer com que elas sejam emitidas por Autoridades Certificadoras (AC) de uma Infraestrutura de Chaves Públicas (ICP), do inglês *Public-Key Infrastrucuture* - PKI, que torne possível a verificação da autenticidade de tais chaves.

Certificação digital e os aspectos de validação de certificados digitais possuem atualmente boas práticas bem documentadas [Gutmann b, Gutmann a, NIST 2012] e literatura especializada na implantação de tais infraestruturas [Chandra et al. 2002]. Um certificado digital de chave pública, ou simplesmente certificado, é uma estrutura de dados que dá como verdadeiro o vínculo entre uma chave pública autêntica e uma entidade cujo nome está no certificado. A veracidade do certificado é garantida por uma terceira parte confiável, emissora do certificado, chamada de Autoridade Certificadora (CA). O certificado contém a assinatura digital da CA emissora e várias informações, tais como a chave pública, a identidade da entidade reconhecida pela CA, datas de início de uso e de validade (final de uso), etc. Por isto, o

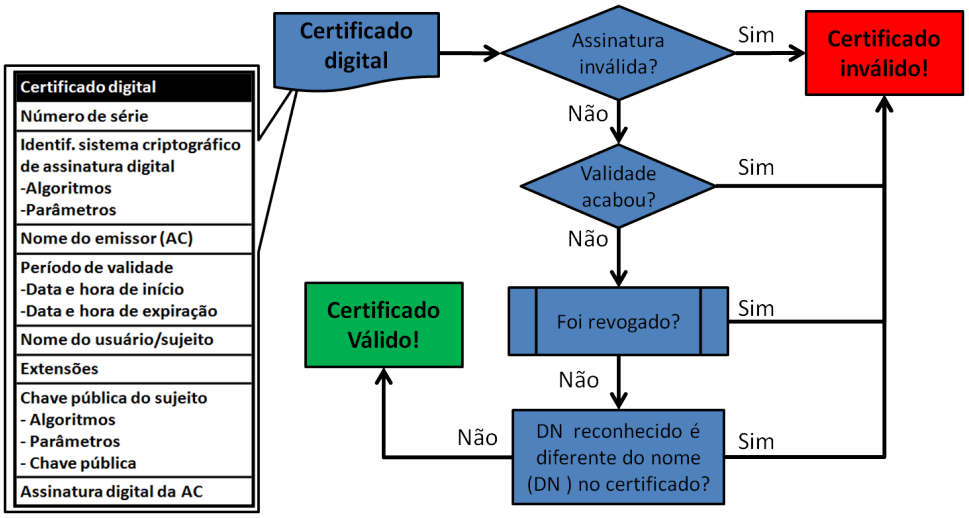


**Figura 2.4. Fluxograma de validação de um certificado digital (de [Braga and Dahab 2015b]).**

certificado é usado na verificação de que uma chave pública válida pertence a uma entidade e serve também como meio confiável para distribuição de chaves públicas.

A validação do certificado é realizada toda vez que a autenticidade da chave pública contida nele deve ser garantida. A assinatura da AC pode ser verificada por qualquer um com acesso à chave pública da AC, cujo certificado é amplamente disponível. Por exemplo, num caso bastante comum, uma AC pode emitir certificados SSL/TLS para servidores web que se comunicam por meio do protocolo HTTP sobre TLS (HTTPS). Quando um software cliente HTTPS (o navegador ou *browser*) faz uma requisição para um servidor web protegido, a resposta do servidor inclui o seu certificado digital. O software cliente HTTPS valida o certificado do servidor verificando a assinatura da AC emissora sobre a chave pública do servidor e outros parâmetros do certificado. Se o cliente já não possuir a chave pública da AC, ele vai buscá-la (em um repositório de chaves públicas da AC). Se o certificado é verificado como válido, então o cliente sabe que o servidor é autêntico.

A validação do certificado (e da chave pública contida nele) abrange mais etapas do que somente a verificação da assinatura da AC. Além da verificação da assinatura, o software cliente precisa verificar se o certificado não atingiu o final de seu período de validade, se não foi revogado e se o nome constante no certificado é o mesmo da parte que alega ser a dona da chave pública. O nome também deve ser obtido de um terceiro confiável, por exemplo, de um serviço de DNS seguro, no caso de nomes de domínio. A validação do certificado é ilustrada no fluxograma da Figura 2.4.

A verificação da assinatura da AC em um certificado digital exige a chave pública da AC. Para ser confiável, a chave pública da AC deve estar contida em um certificado assinado por outra AC ou autoassinado, se for uma CA raiz. As verificações sucessivas de uma sequência de assinaturas constroem uma cadeia ou hierarquia de certificados. Os certificados na base da hierarquia são assinados pelas ACs de mais baixo nível, cujos certificados são assinados pelas ACs intermediárias, que têm seus certificados assinados pelas ACs de alto nível, cujos certificados são assinados pela AC raiz, que tem seus certificados autoassinados. A Figura 2.5

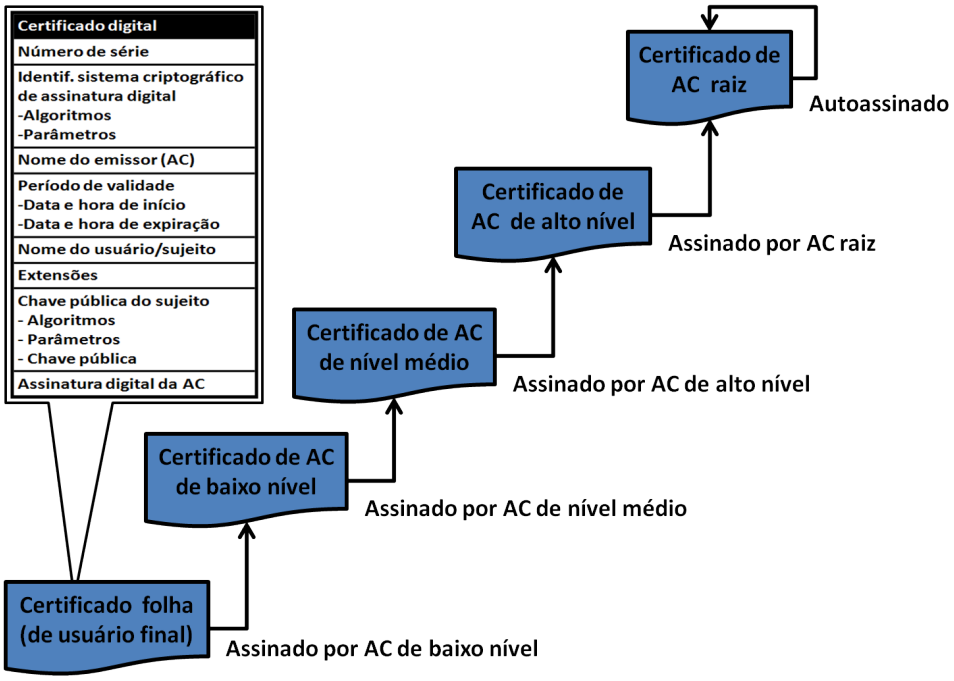


**Figura 2.5. Cadeia hierárquica de certificação digital (de [Braga and Dahab 2015b]).**

ilustra esta cadeia de certificação.

Uma AC revoga um certificado nas seguintes situações: quando ocorrem erros na emissão do certificado (nome grafado errado, por exemplo), ou o certificado foi emitido para uso de um serviço e o portador não tem mais acesso a ele (demissão de um funcionário), ou a chave privada do portador foi comprometida (um smartcard foi perdido), ou ainda a chave privada da AC foi comprometida (uma situação extrema). Uma Lista de Certificados Revogados (*Certificate Revocation List* - CRL) é o documento assinado digitalmente pela AC que lista o número de série de todos os certificados, ainda não expirados, que perderam a utilidade por algum dos motivos acima. Um software criptográfico pode consultar um serviço de CRL para receber atualizações periódicas, em intervalos regulares definidos por procedimentos, ou ainda consultar em tempo real se um certificado foi revogado ou não. Porém, a instabilidade do canal de comunicação pode causar indisponibilidade do serviço de validação em tempo real.

### 2.2.6. O protocolo SSL/TLS

*Security Sockets Layer* - SSL (camada de segurança de *sockets*, em tradução livre) é o protocolo para transporte de dados protegidos com criptografia sobre os *sockets* TCP. O SSL permite que a comunicação pela Internet entre aplicações cliente/servidor possa ser protegida contra monitoramento, adulteração de conteúdo e falsificação de mensagens. O objetivo principal do SSL é proporcionar um canal de comunicação seguro entre partes comunicantes. Tradicionalmente, livros de segurança em redes [Stallings 2003] são boas fontes para estudo do SSL, havendo também literatura específica [Chandra et al. 2002, Ristic 2015] sobre a implementação de código aberto mais conhecida deste protocolo, o OpenSSL [OpenSSL.org ].

A terceira versão do SSL (SSLv3) serviu de base para o *Transport Layer Security* (TLS).

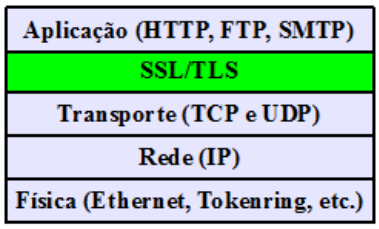


**Figura 2.6. Localização do Protocolo SSL/TLS na Pilha TCP/IP.**

O TLS é um protocolo padronizado pelo *Internet Engineering Task Force* (IETF) com o objetivo de substituir o formato proprietário do SSL. O SSLv3 oferece os seguintes características de segurança disponíveis até hoje no TLS:

- Autenticação do servidor com assinaturas digitais e certificados X.509, por meio dos algoritmos RSA, DSA e ECDSA, ou até por meio de uma chave simétrica pré-configurada;
- Autenticação (opcional) do cliente com assinaturas digitais e certificados X.509;
- Sigilo com encriptação da informação trocada entre cliente e servidor;
- Integridade (com uso de MACs) da informação trocada entre cliente e servidor.

Por razões históricas, o protocolo TLS também é referido como SSL/TLS. A Figura 2.6 posiciona o SSL/TLS em relação a pilha de protocolos TCP/IP. A camada de *socket* seguro fica logo acima da camada de transporte. De fato, as aplicações (na camada de aplicação) podem usar os *sockets* seguros como se fossem um serviço oferecido pela camada de transporte.

Inicialmente, o SSL foi usado para garantir o sigilo, autenticação e integridade dos dados transmitidos via uma conexão HTTP. Quando uma conexão HTTP é protegida com SSL, o protocolo é chamado HTTPS. Na autenticação do servidor SSL/TLS, para iniciar a canal seguro, o servidor HTTPS primeiro envia seu próprio certificado digital. O cliente faz verificações no certificado (estas verificações são descritas em outra seção) e indica que o certificado é válido e confiável. O canal seguro é estabelecido e o servidor envia o documento requisitado para o cliente. Já na autenticação do cliente SSL/TLS, o servidor requisita o certificado digital do cliente, que o envia ao servidor. O servidor faz verificações no certificado do cliente e indica que o certificado é válido e confiável. Em seguida, o servidor envia o conteúdo requisitado para o cliente autenticado. Este tipo de autenticação do cliente pode substituir o uso de senhas em sites web.

O SSL/TLS possui dois sub-protocolos componentes: a saudação (*handshake*), que autentica as partes e negocia os parâmetros de segurança criptográfica da comunicação, e o protocolo de registro, que usa os parâmetros escolhidos na saudação para proteger o tráfego entre as partes comunicantes. A Figura 2.7 ilustra o funcionamento da saudação (*handshake*) do SSL/TLS v1.2. As etapas do diálogo de saudação entre cliente e servidor são detalhadas e enumeradas na Figura 2.7. O objetivo principal deste diálogo é o estabelecimento de uma chave de sessão (uma chave criptográfica secreta, simétrica e temporária). O diálogo de saudação pode ser dividido em quatro partes:

1. Negociação de parâmetros (versão, ID de sessão, algoritmos criptográficos e compressão);
2. Envio do certificado do servidor e solicitação opcional do certificado do cliente;

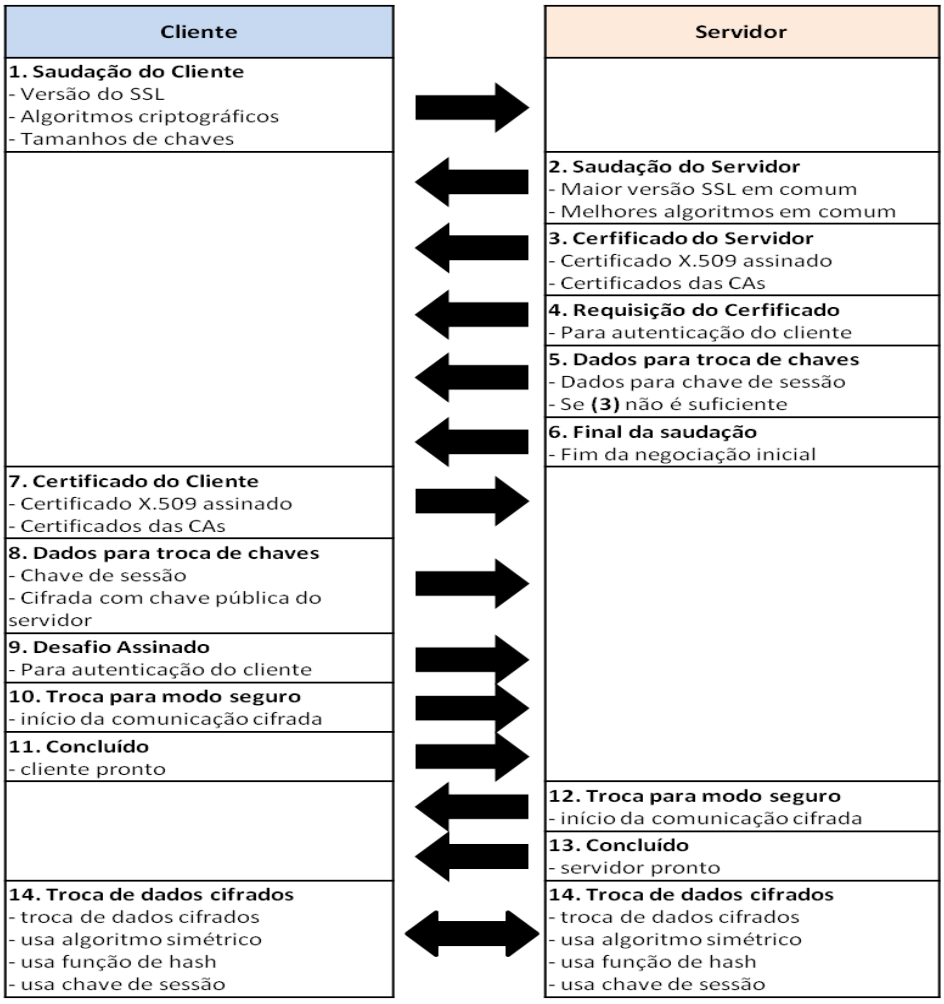


**Figura 2.7. Estabelecimento de sessão e envio de mensagens SSL/TLS v1.2.**

3. Envio do certificado do cliente, se solicitado;
4. Escolha de algoritmos criptográficos e finalização;

Esta implementação do *handshake* em duas rodadas é considerada [Sullivan 2018] uma causa de perda de desempenho na execução do protocolo e foi otimizada na versão 1.3. Após a realização do protocolo de saudação (*handshake*), entra em ação o protocolo de registro que realiza a troca, entre cliente e servidor, de informação encriptada segmentada em blocos chamados de registros. Este texto não trata o protocolo de registro.

O SSL/TLS é independente de aplicação e, como tal, não especifica como ele deve ser adicionado a um protocolo de aplicação. Por isto, decisões de projeto importantes são deixadas a cargo dos implementadores de protocolos de aplicação, tais como em que momento o *handshake* deve ser iniciado ou como interpretar a autenticação dos certificados digitais trocados entre as partes. Esta situação abre espaço para a inclusão de defeitos de implementação que levam a vulnerabilidades exploráveis.

### 2.2.7. Versões do SSL/TLS e suas vulnerabilidades

Além do SSLv3 que deu origem ao TLSv1, vale ainda mencionar outras versões do SSL/TLS que se destacam por possuírem características específicas e vulnerabilidades conhecidas. Em linhas gerais, existem seis versões em uso do SSL/TLS que podem ser encontradas em implantações reais: SSLv2, SSLv3, TLSv1.0, TLSv1.1, TLSv1.2 e TLSv1.3. As características gerais de cada uma delas são descritas a seguir:

- SSLv2 é considerado inseguro e não deve mais ser usado. Esta versão é vulnerável a ataques conhecidos, tais como o BEAST (Browser Exploit against SSL/TLS) e o DROWN (Decrypting RSA with Obsolete and Weakened eNcryption) [Aviram et al. 2016], e outras vulnerabilidades como *Cipher Suite Rollback* e *ChangeCipherSpec Message Drop*.
- SSLv3 é obsoleto e não deve mais ser utilizado. O HTTPS sobre SSLv3 é vulnerável aos ataques POODLE (Padding Oracle on Downgraded Legacy Encryption) [Möller et al. 2014] e *Lucky 13* [Fardan and Paterson 2013] e sofre de vulnerabilidades como o *Version Rollback* e o *Key Exchange Algorithm Confusion*.
- TLSv1.0 (RFC 2246) ainda é usada em sistemas legados. TLSv1.0 é vulnerável aos ataques BEAST e *Lucky 13* [Fardan and Paterson 2013]. Esta versão inicial do TLS também é vulnerável a ataques de negação de serviço e que exploram falhas na renegociação. Padrões de segurança como o PCI-DSS recomendam que esta versão seja abandonada.
- TLSv1.1 (RFC 4346) é uma versão relativamente recente e que não apresenta vulnerabilidades conhecidas sem mitigação, mas ainda oferece algoritmos criptográficos antigos.
- TLSv1.2 (RFC 5246) era a versão atual até Agosto (acabou de ser substituída pela versão 1.3) e já oferece esquemas criptográficos novos com encriptação autenticada.
- TLSv1.3 (RFC 8446) foi lançado na forma final em Agosto de 2018 [Rescorla 2018] e representa o rompimento definitivo com o passado pela eliminação de diversas características inseguras ou obsoletas mantidas até hoje por compatibilidade com legados.

Há testes específicos para implantações do SSL/TLS [Ristic 2015, Eldewahi et al. 2015, OWASP 2015]. Outros ataques contra SSL/TLS bastante conhecidos são o CRIME (*Compression Ratio info-leak Made Easy*)[CRI 2012], o BREACH (*Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext*) [BRE 2013, Gluck et al. 2013], o *Bleichenbacher Attack on PKCS#l* [Bleichenbacher 1998], os ataques ao RC4 [AlFardan et al. 2013], o Heartbleed, que afeta implementações como o OpenSSL, e o LogJam [Adrian et al. 2015, Log 2016], que afeta as implementações dos protocolos de acordo de chaves.

### 2.2.8. A versão 1.3 do SSL/TLS

Uma vez que a versão 1.3 do TLS [Rescorla 2018] foi lançada de fato na forma final em Agosto deste ano (2018), faz sendo incluir neste texto um breve resumo dos avanços apresentados por esta nova versão em relação a sua antecessora. Já há análises detalhadas do TLSv1.3 [Sullivan 2018] disponíveis para o público em geral. Grosso modo, a nova versão 1.3 atende a quatro objetivos de projeto: redução da latência do *handshake*, encriptação de partes do *handshake*, aumento da robustez contra ataques e remoção de funções legadas. A lista a seguir contém as principais diferenças entre o TLSv1.2 e o TLSv1.3 [Rescorla 2018]:

- A lista de opções de algoritmos simétricos foi bastante reduzida e só contém encriptação autenticada (*authenticated encryption with additional data* - AEAD). Cifras de bloco em modo CBC e a cifra RC4 foram removidos desta versão.
- O conceito de *suite* criptográfica foi simplificado e agora separa os mecanismos de autenticação e troca de chaves dos mecanismos de encriptação, *hash* usado na derivação de chaves e MACs.
- O *handshake* simplificado, em uma única rodada, acelera a saudação. Além disso, o *handshake* em zero rodadas (sem negociações, só comunicação de escolhas pré-definidas) foi incluído para atender às aplicações com restrições de tempo ainda maiores.
- RSA e DH com parâmetros estáticos foram removidos, de modo que todas as *suites* criptográficas agora possuem *forward secrecy*. Porém, só o DHE/ECDHE é usado para troca de chaves.
- Todas as mensagens do *handshake* depois do "ServerHello"agora são encriptadas e assinadas pelo servidor. esta medida protege contra os ataques práticos (FREAK, LogJam) e outros mais teóricos, como a troca maliciosa de curvas elípticas (*curveswap*).
- A criptografia de curvas elípticas faz parte da especificação principal protocolo, mas negociação de formatos de representação de pontos da curva foi removida, para incluir apenas um formato.
- Em termos de criptografia de chaves públicas, o RSA-PSS é o único *padding* do RSA permitido. O *padding* inseguro do padrão PKCS#1 v1.5 foi removido, assim como também foram removidos o DSA e a compressão.
- DH com parâmetros efêmeros customizados não é mais permitido. Há agora um número fixo de opções sabidamente seguras. Isto evita a parametrização fraca do DH que leva, por exemplo, ao ataque LogJam [Log 2016].
- A Mensagem "ChangeCipherSpec"foi removida do *handshake*, simplificando o protocolo e aumentando a robustez contra ataques. Já o *handshake* com chave pré-compartilhada *pre-shared key* (PSK) foi simplificado para aumentar a robustez contra ataques.

### 2.2.9. Testando uma conexão TLS com OpenSSL

O software OpenSSL [OpenSSL.org ] vem com uma ferramenta cliente TLS que pode ser usada para estabelecer conexões com um servidor. O comando a seguir mostra como o cliente `s_client` usa a opção `-connect`, o nome do servidor e a porta 443 (padrão do SSL) para estabelecer uma conexão segura (na versão 1.2 `-tls1.2`) com um servidor.

```
$openssl s_client -connect exemplo.com:443 -tls1_2
```

O comportamento padrão do comando é tentar uma conexão na versão mais alta do protocolo disponível na instalação do OpenSSL. Versões específicas podem ser usadas explicitamente com as opções `-ssl2`, `-ssl3`, `-tls1`, `-tls1_1`, `-tls1_2` e `-tls1_3` (somente em distribuições novas). Além disso, versões podem ser excluídas com as opções `-no_ssl2`, `-no_ssl3`, `-no_tls1`, `-no_tls1_1`, `-no_tls1_2`.

O suporte a algoritmos específicos pode ser testado pela opção `-cipher` do comando `s_client` seguido do nome da *suite* criptográfica. Já o comando `-ciphers -s`

lista os algoritmos suportados na instalação local no OpenSSL. Os comandos a seguir listam os algoritmos disponíveis e depois testam uma conexão com `AES128-SHA`, uma configuração fraca. A opção `-cipher kECDHE` valida o uso do ECDH efêmero no acordo de chaves.

```
$openssl -ciphers -s
$openssl s_client -connect exemplo.com:443 -cipher AES128-SHA
$openssl s_client -connect exemplo.com:443 -cipher kECDHE
```

### 2.2.10. Considerações finais sobre SSL/TLS

A História do protocolo SSL se confunde com a história do comércio eletrônico na Internet. Na última década do milênio passado, a empresa Netscape [Wikipedia 2018] dominava tanto o mercado de softwares navegadores web (*browsers*) como o de servidores Web. Em um modelo de negócios inovador para a época, a Netscape distribuía gratuitamente o *browser*, mas ganhava muito dinheiro com a venda do seu servidor web, o único na época a possuir proteções de sigilo contra monitoramento e interceptação da comunicação. A segurança da comunicação era garantida por um protocolo criptográfico na camada de transporte do TCP/IP que ficou conhecido como SSL.

O SSL tornou possíveis as transações sigilosas entre o *browser* na máquina do cliente e o servidor web (do lojista). O uso do SSL é praticamente transparente para o cliente, o que facilitou muito a ampla adoção deste protocolo como padrão de fato para segurança na Internet. Com o SSL, os números de cartões de crédito puderam transitar pelas redes abertas com sigilo, viabilizando o comércio eletrônico como acontece hoje em dia.

Entretanto, o SSL não resolve todos os problemas de segurança. Por exemplo, uma vez que a informação cifrada do cartão de crédito chega ao lojista, ela precisa ser decifrada para que o pagamento seja efetivado junto à instituição financeira (banco ou operadora de cartão de crédito). Neste momento, as informações do cartão ficam expostas ao lojista e aos seus funcionários. O SSL não resolve este problema, pois ele trata apenas da comunicação entre *browser* e servidor web. Outros protocolos de segurança devem ser usados para proteger as informações de pagamento.

Finalmente, hoje em dia, surgem modalidades de ataques que exploram a confiança já estabelecida em sites web que possuem conexões SSL/TLS legítimas. O abuso da confiança depositada por usuários na conexão SSL/TLS entre *browser* e servidor web ocorre quando uma conexão SSL/TLS legítima é usada em ataques. Atualmente, mesmo sites web maliciosos, utilizados por exemplo em ataques de *phishing*, podem ter certificados SSL legítimos em seus servidores. Isto se deve ao fato de autoridades certificadores emitirem, de forma simplificada na Internet, certificados gratuitos e de validade curta. Estes certificados, mesmo com validade reduzida, ainda podem ser usados por tempo suficiente para viabilizar ataques de engenharia social [Calvo 2018].

## 2.3. Exemplos de bons usos da criptografia assimétrica

Esta seção usa programas que utilizam a API criptográfica padrão Java e a biblioteca criptográfica BouncyCastle [BouncyCastle 2018] para mostrar os seguintes conceitos: encriptação assimétrica aleatorizada, assinatura digital probabilística e validação de certificados digitais. Além disso, a seção mostra o uso dos algoritmos RSA-OAEP, RSA-PSS, ECDSA, ECIES e curvas elípticas seguras. Alguns dos programas foram adaptados de [Braga et al. 2017a].

### 2.3.1. Encriptação e decriptação assimétrica

Esta subseção ilustra a encriptação assimétrica e aleatorizada com dois esquemas criptográficos diferentes: RSA-OAEP e ECIES. Nos programas que se sequem, o leitor deve imaginar a interação entre os dois personagens tradicionais de criptossistemas, Ana e Beto, ou Alice e Bob.

#### 2.3.1.1. RSA Aleatorizado: RSA-OAEP

O código na Listagem 2.1 contém o método principal (`main`) de um programa Java para encriptação com RSA aleatorizado RSA-OAEP. A linha 3 seleciona o provedor criptográfico *BouncyCastle*. A linha 4 cria um texto claro na variável `pt`. Nas linhas 6 e 7, cria-se o par de chaves RSA com 2048 bits. A linha 8 define o RSA como a implementação OAEP com *hash* SHA256 e *padding* MGF1 (o único disponível). As linhas de 9 a 12 criam duas máquinas criptográficas. Uma para encriptação (configurada com a chave pública) e outra de decriptação (usando a chave privada). Finalmente, as linhas de 14 a 17 realizam três encriptações sucessivas do mesmo texto claro para mostrar que elas são diferentes de fato. A linha 18 faz a decriptação. As linhas de 19 a 21 mostram todas as exceções tratadas para o RSA-OAEP em Java. Vale a pena mencionar as exceções de chave inválida, de tamanho de bloco inválido e de *padding* ruim ou desconhecido.

**Listagem 2.1. RSA Aleatorizado com RSA-OAEP.**

```
1 public static void main(String args[]) {
2   try {
3     Security.addProvider(new BouncyCastleProvider()); // provedor BC
4     byte[] pt = ("Randomized RSA").getBytes();
5
6     KeyPairGenerator g = KeyPairGenerator.getInstance("RSA", "BC");
7     g.initialize(2048); KeyPair kp = g.generateKeyPair();
8     String RSA_OAEP = "RSA/None/OAEPWithSHA256AndMGF1Padding";
9     Cipher e = Cipher.getInstance(RSA_OAEP, "BC");
10    e.init(Cipher.ENCRYPT_MODE, kp.getPublic());
11    Cipher d = Cipher.getInstance(RSA_OAEP, "BC");
12    d.init(Cipher.DECRYPT_MODE, kp.getPrivate());
13
14    U.println("Plaintext: " + U.b2x(pt));
15    U.println("Ciphertext: " + U.b2x(e.doFinal(pt)));
16    U.println("Ciphertext: " + U.b2x(e.doFinal(pt)));
17    U.println("Ciphertext: " + U.b2x(e.doFinal(pt)));
18    U.println("Plaintext: " + U.b2x(d.doFinal(e.doFinal(pt))));
19  } catch (NoSuchAlgorithmException | NoSuchPaddingException |
20           InvalidKeyException | IllegalBlockSizeException |
21           BadPaddingException | NoSuchProviderException e)
22  { System.out.println(e);}}
```

#### 2.3.1.2. Outra maneira de usar o RSA-OAEP

O trecho de código na Listagem 2.2 mostra uma segunda maneira de utilizar o RSA aleatorizado OAEP. A linha 1 seleciona o provedor de serviços criptográficos *BouncyCastle*. As linhas 2 e 3 criam o par de chaves RSA com 2048 bits utilizado por Beto para receber mensagens encriptadas por Ana com a chave pública de Beto.

As linhas de 5 a 9 mostram as configurações comuns para Ana e Beto, em que os parâmetros do OAEP são definidos manualmente. Primeiro, a função de preenchimento MGF1 é configurada para usar a função de *hash* SHA512. Em seguida, os parâmetros do OAEP são definidos como SHA512, MGF1 (única opção disponível) e a fonte de aleatorização *default*. Finalmente, a linha 9 instancia um RSA-OAEP sem configurações ("RSA/None/OAEPPadding").

As linhas de 12 a 16 mostram como Ana encripta com a chave pública de Beto. Primeiro, nas linhas 12 e 13, a máquina de encriptação é configurada com os parâmetros do OAEP (definidos anteriormente) e a chave pública de Beto. Em seguida, na linha 14, é calculado o tamanho adequado, em bytes, do texto claro de entrada para o encriptador. Na fórmula, 2048 e 512 são os tamanhos, em bits, da chave e do *hash*, respectivamente. A linha 15 extrai um trecho do texto claro como tamanho adequado e a linha 16 realiza a encriptação.

Finalmente, nas linhas de 18 a 20, Beto usa sua chave privada (linha 18) para configurar a máquina criptográfica em modo de decriptação e com os parâmetros OAEP (linha 19), obtendo o texto claro a partir do criptograma (linha 20).

O RSA-OAEP limita o tamanho do texto claro que pode ser encriptado em uma única chamada da função. Este limite está relacionado ao tamanho do corpo finito (e da chave) usado na aritmética modular do algoritmo RSA e pode ser determinado, em bytes, pela fórmula *(ks-2*hs)/8–2*, onde *ks* é o tamanho da chave RSA em bits e *hs* é o tamanho do *hash* em bits usado pelo *padding* OAEP.

**Listagem 2.2. Outro RSA Aleatorizado.**

```
1 Security.addProvider(new BouncyCastleProvider());
2 KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA","BC");
3 kpg.initialize(2048); KeyPair kp = kpg.generateKeyPair();
4
5 // configuracoes comuns para Ana e Bato
6 MGF1ParameterSpec mgf1ps = MGF1ParameterSpec.SHA512;
7 OAEPParameterSpec OAEPps = new OAEPParameterSpec("SHA512","MGF1",
8         mgf1ps,PSource.PSpecified.DEFAULT);
9 Cipher c = Cipher.getInstance("RSA/None/OAEPPadding", "BC");
10
11 // Encriptacao pela Ana com a chabe publica de Beto
12 Key pubk = kp.getPublic();
13 c.init(Cipher.ENCRYPT_MODE, pubk, OAEPps);
14 int maxLenB = (2048-2*512)/8-2;//tamanho max do texto claro (bytes)
15 byte[] pt = U.cancaoDoExilio.substring(0, maxLenB).getBytes();
16 byte[] ct = c.doFinal(pt);
17
18 Key privk = kp.getPrivate();//decriptacao com chave privada do Beto
19 c.init(Cipher.DECRYPT_MODE, privk, OAEPps); //modo de decriptacao
20 byte[] ptBeto = c.doFinal(ct); // Decriptando
```

#### 2.3.1.3. Encriptação assimétrica com curvas elípticas: ECIES

O ECIES é um esquema de encriptação com curvas elípticas [Hankerson et al. 2004]. A encriptação com ECIES é aleatorizada e autenticada, combinado no esquema criptográfico um algoritmo simétrico (como o AES) e um MAC para autenticação do criptograma. O ECIES pode ser uma alternativa ao RSA-OAEP com a vantagem de utilizar chaves menores que aqueles do RSA, mantendo o mesmo nível de segurança. Nenhum dos provedores criptográficos disponíveis na instalação da plataforma Java oferece uma implementação do ECIES, por isso foi utilizada a implementação do ECIES no provedor criptográfico *BouncyCastle* (BC).

O trecho de código da Listagem 2.3 mostra como utilizar o ECIES do provedor BC. A linha 3 adiciona dinamicamente o provedor BC e a linha 4 define o texto claro. Um gerador de par de chaves para o ECIES é instanciado na linha 5 e configurado para uma curva elíptica com chaves de 224 bits de tamanho (linha 6). Então, o par de chaves é criado na linha 6. Instancias do ECIES, para encriptação com a chave pública (linha 8) e decriptação com a chave privada (linha 10), são criadas a partir do identificador "ECIESwithAES–CBC", um ECIES com AES no modo CBC, sem *padding*.

As linhas de 13 a 16 computam duas vezes o criptograma sobre o mesmo texto claro, para mostrar que os criptogramas são diferentes. A linha 16 decripta o criptograma, recuperando o texto claro. Vale ainda observar que o criptograma gerado pelo ECIES é bem mais longo que o texto claro utilizado no exemplo. Este é uma característica da criptografia assimétrica aleatorizada, que carrega um *nonce* no criptograma, entre outras informações específicas de cada esquema criptográfico, como por exemplo a *tag* de autenticação. Este característica tem questões de armazenamento de podem inibir o uso do ECIES para proteger textos claros.

**Listagem 2.3. Encriptação e decriptação com ECIES da BouncyCastle.**

```
1 public static void main(String args[]) {
2  try {
3    Security.addProvider(new BouncyCastleProvider());// provedor BC
4    byte[] ptAna = ("Teste do ECIES").getBytes();
5    KeyPairGenerator g = KeyPairGenerator.getInstance("ECIES","BC");
6    g.initialize(224); KeyPair kp = g.generateKeyPair();
7
8    Cipher e=Cipher.getInstance("ECIESwithAES-CBC/NONE/NOPADDING","BC");
9    e.init(Cipher.ENCRYPT_MODE, kp.getPublic());
10   Cipher d=Cipher.getInstance("ECIESwithAES-CBC/NONE/NOPADDING","BC");
11   d.init(Cipher.DECRYPT_MODE, kp.getPrivate());
12
13   U.println("Plaintext : " + U.b2x(ptAna));
14   U.println("Ciphertext: " + U.b2x(e.doFinal(ptAna)));
15   U.println("Ciphertext: " + U.b2x(e.doFinal(ptAna)));
16   U.println("Plaintext : " + U.b2x(d.doFinal(e.doFinal(ptAna))));
17 } catch (NoSuchAlgorithmException | NoSuchPaddingException    |
18               InvalidKeyException | IllegalBlockSizeException|
19               BadPaddingException | NoSuchProviderException e)
20 { System.out.println(e); }}
```

### 2.3.2. Assinaturas digitais não-determinísticas

Esta subseção ilustra o uso de assinatura digital como mecanismo de autenticação de origem (autoria) de uma informação. Dois esquemas criptográficos de assinaturas digitais não-determinísticas são ilustrados: assinaturas probabilísticas com RSA-PSS e aleatorizadas com o ECDSA.

#### 2.3.2.1. Assinaturas digitais probabilísticas com RSA-PSS

O trecho de código na Listagem 2.4 mostra como utilizar o RSA-PSS em Java. As linhas de 2 até 4 selecionam o provedor criptográfico "BC"e criam o par de chaves RSA com 3072 bits. As linhas de 7 a 10 criam uma instância da máquina de assinatura (o assinador) para o PSS, indicado pela string "SHA256withRSAandMGF1", e constroem um objeto de parâmetros do PSS com *hash* SHA256 e MGF1, de modo análogo ao RSA-OAEP.

As linhas de 12 a 15 realizam o processo de geração da assinatura digital. Primeiro, a linha 12 cria a mensagem (texto claro) que será assinada neste exemplo. Em seguida, a linha 13 inicializam o assinador com a chave privada e um gerador de números pseudoaleatórios seguro, enquanto a linha 14 alimenta o assinador com a mensagem a ser assinada e a linha 15 calcula a assinatura digital, com o método `sign()`.

Por outro lado, as linhas 17 e 20 realizam o processo de verificação da assinatura digital. Primeiro, na linha 17, a máquina de assinatura e inicializada para verificação e parametrizada com a chave pública correspondente àquela chave privada usada na assinatura. Em seguida, o verificador é alimentado com a mensagem em texto claro (Neste caso, para que o *hash* da mensagem seja recalculado). Finalmente, a linha 19 faz a verificação da assinatura digital com o método `verify()`.

**Listagem 2.4. Assinatura digital aleatorizada com RSA-PSS.**

```
1 public static void main(String[] args) throws Exception {
2   Security.addProvider(new BouncyCastleProvider()); // provedor BC
3   KeyPairGenerator kg = KeyPairGenerator.getInstance("RSA", "BC");
4   kg.initialize(3072, new SecureRandom());
5   KeyPair kp = kg.generateKeyPair();
6
7   Signature sig = Signature.getInstance("SHA256withRSAandMGF1", "BC");
8   PSSParameterSpec spec = new PSSParameterSpec("SHA256", "MGF1",
9         MGF1ParameterSpec.SHA256, 20, 1);
10  sig.setParameter(spec);
11
12  byte[] m = "Testing good RSA PSS".getBytes();
13  sig.initSign(kp.getPrivate(), new SecureRandom());
14  sig.update(m);
15  byte[] s = sig.sign();
16
17  sig.initVerify(kp.getPublic());
18  sig.update(m);
19  if (sig.verify(s)) { System.out.println("Verification succeeded.");}
20  else               { System.out.println("Verification failed.");    }
21 }
```

#### 2.3.2.2. Assinaturas digitais aleatorizadas com ECDSA

O trecho de código na Listagem 2.5 é o primeiro exemplo deste texto que utiliza explicitamente as curvas elípticas. Por isto, as linhas de 2 a 5 têm novidades. A linha 2 contém a escolha da curva "secp256r1"ou NIST P-256. A linha 3 instancia um gerador de par de chaves para curvas elípticas, enquanto a linha 4 inicializa o gerador com a curva selecionada. A linha 5 gera o par de chaves. O gerador de par de chaves utiliza o provedor criptográfico "SunEC", que agrupa diversas implementações criptográficas relacionadas às curvas elípticas e está disponível na plataforma Java desde a versão 6 [Oracle b].

As linhas 7 e 8 instanciam um objeto `SecureRandom` com o melhor PRNG disponível para a implantação Java, geram uma semente de 24 bytes e configuram o objeto `SecureRandom` com ela. As linhas de 10 a 13 realizam os passos para geração de uma assinatura digital. A linha 10 instancia um objeto assinador para o esquema criptográfico ECDSA identificado por "SHA256withECDSA". A linha 11 configura o assinador com a chave privada e o PRNG. A linha 12 obtém o documento a ser assinado enquanto a linha 13 alimenta o assinador com o documento e calcula a assinatura digital.

As linhas de 15 a 21 repetem os passos de instanciação e configuração de um assinador e geração de uma assinatura digital para o mesmo documento assinado anteriormente, porém, desta vez armazenados em objetos diferentes. As duas assinaturas digitais são comparadas na linha 23, mostrando que elas são diferentes, apesar de terem sido geradas sobre o mesmo documento. Este exemplo não faz a verificação da assinatura digital. O programa que mostra a verificação de uma assinatura digital pode ser encontrado em [Braga and Dahab 2015b].

**Listagem 2.5. ECDSA com nonce não repetido.**

```
1 public static void main(String[] args) throws Exception {
2   ECGenParameterSpec ecps = new ECGenParameterSpec("secp256r1");
3   KeyPairGenerator kpg = KeyPairGenerator.getInstance("EC","SunEC");
4   kpg.initialize(ecps);
5   KeyPair kpAna = kpg.generateKeyPair();
6
7   SecureRandom sr1 = SecureRandom.getInstanceStrong();
8   byte[] seed = sr1.generateSeed(24); sr1.setSeed(seed);
9
10  Signature signer1 = Signature.getInstance("SHA256withECDSA","SunEC");
11  signer1.initSign(kpAna.getPrivate(), sr1);
12  byte[] doc = U.cancaoDoExilio.getBytes();
13  signer1.update(doc); byte[] sign1 = signer1.sign();
14
15  SecureRandom sr2 =  SecureRandom.getInstanceStrong();
16  sr2.setSeed(seed);
17
18  Signature signer2 = Signature.getInstance("SHA256withECDSA","SunEC");
19  signer2.initSign(kpAna.getPrivate(), sr2);
20  doc = U.cancaoDoExilio.getBytes();
21  signer2.update(doc); byte[] sign2 = signer2.sign();
22
23  boolean ok = Arrays.areEqual(sign1, sign2);
24  if (ok){System.out.println("Nonce repeated! Signatures are equal!");}
25  else   {System.out.println("Signatures are different!");}
26 }
```

### 2.3.3. Curvas elípticas seguras

O provedor criptográfico "SunEC"oferece diversas opções de curvas elípticas. A Tabela 2.2 lista as curvas padronizadas e ainda consideradas seguras pelos padrões internacionais [NIST 2013, SEC 2010] e que estão disponíveis no provedor "SunEC", de acordo com [Mart and Hern 2013]. Na tabela, as curvas seguem a nomenclatura de SEC-2 [SEC 2010], com nomes no formato `sec[p|t]XXX[r|k]v`, onde `[p|t]` indica uma curva sobre corpo primo (`p`) ou binário (`t`). Ainda, o número `XXX` indica o tamanho em bits da chave e a letra `[r|k]` indica que a curva usa os parâmetros de Koblitz (`k`) ou randômicos (`r`), em uma determinada versão (`v`).

Na Tabela 2.2, a coluna da esquerda mostra o nível de segurança (em bits) e as outras duas colunas mostram as curvas elípticas sobre corpos primos $F_p$ e binários $F_{2^m}$. As curvas secp224r1 (NIST P-224), secp256k1, secp256r1 (NIST P-256) e secp384r1 (NIST P384), marcadas com um asterisco, são consideradas inseguras por Daniel Bernstein [Bernstein et al. 2013]. Ele considera que estas curvas são complexas de implementar, facilitando a ocorrência de defeitos que levam a vulnerabilidades, entre outros problemas.

**Tabela 2.2. Curvas elípticas seguras no provedor SunEC (de [Mart and Hern 2013]).**

| Segurança | Curvas sobre $F_p$ | Curvas sobre $F_{2^m}$ |
|---|---|---|
| 80 | – | sect163k1,sect163r1,sect163r2 |
| 96 | secp192k1,secp192r1 | – |
| 112 | secp224k1,*secp224r1** | sect233k1, sect233r1 |
| 115 | – | sect239k1 |
| 128 | *secp256k1*,secp256r1** | sect283k1, sect283r1 |
| 192 | *secp384r1** | sect409k1, sect409r1 |
| 256 | secp521r1 | sect571k1, sect571r1 |

O trecho de código na Listagem 2.6 cria pares de chaves para cada uma das curvas na tabela 2.2 e lista os parâmetros das chaves. A execução do programa mostra que a chave pública é um ponto, em coordenadas $(x,y)$, da curva elíptica, enquanto a chave privada é uma sequência de bits derivada de um número inteiro muito longo. Além disso, os nomes alternativos das curvas (quando existem) também são listados e a chave privada está codificada no formato PKCS#8.

**Listagem 2.6. Selecionando as curvas seguras de SunEC.**

```
1 public static void main(String argv[]) {
2  String[] curves={
3     "secp192k1","secp192r1","secp224k1","secp224r1","secp256k1",
4     "secp256r1","secp384r1","secp521r1","sect163k1","sect163r1",
5     "sect163r2","sect233k1","sect233r1","sect239k1","sect283k1",
6     "sect283r1","sect409k1","sect409r1","sect571k1","sect571r1"};
7  try { for (String curve : curves) {
8      ECGenParameterSpec ecps = new ECGenParameterSpec(curve);
9      KeyPairGenerator kpg = KeyPairGenerator.getInstance("EC","SunEC");
10     kpg.initialize(ecps); KeyPair kp = kpg.generateKeyPair();
11     U.println("EC parameters "+ecps.getName());
12     U.println("Pub.  key: " + kp.getPublic());
13     U.println("Priv. key: " + U.b2x(kp.getPrivate().getEncoded()));
14     U.println("Algorithm: " + kp.getPrivate().getAlgorithm());
15     U.println("Format   : " + kp.getPrivate().getFormat());
16   }}catch(NoSuchAlgorithmException|InvalidAlgorithmParameterException|
17         NoSuchProviderException e){System.err.println("Error:"+e);}}
```

### 2.3.4. Validação completa de certificados digitais

O programa na Listagem 2.7 faz a validação de um certificado digital e sua cadeia de certificação em uma conexão segura SSL/TLS. As linhas de 4 a 6 estabelecem a conexão segura e realizam o (*handshake*) do protocolo. As linhas de 8 a 14 mostram informações da sessão segura, tais como versão do protocolo, *suite* criptográfica e os nomes do servidor e do sujeito no certificado.

Não há validação automática do nome do servidor, que deve ser feita manualmente. A comparação do nome do servidor (obtido da sessão) com o nome do sujeito no certificado é feita nas linhas 17 e 18. A validação do certificado é feita nas linhas de 20 a 24. A linha 20 obtém toda a cadeia de certificação associada ao certificado do servidor, incluindo ele próprio. A linha 22 verifica, no método `checkValididy()`, a data de validade do certificado do servidor, enquanto a linha 23 verifica a assinatura digital do certificado do servidor com o método `verify()`. Se a validação falhar, uma exceção específica é disparada (linha 24).

O restante do programa faz a verificação da cadeia de certificação assim como da revogação do certificado do servidor por meio de uma CRL. O processo possui dois passos preparatórios e um passo de verificação: (*i*) Obtenção do certificado raiz e dos outros certificados da cadeia de certificação, de modo que a cadeia possa ser completamente construída (linhas de 27 a 35). (*ii*) Designação dos certificados raízes como âncoras de confiança da cadeia (linhas 38 e 39). (*iii*) Validação da cadeia de certificação e do *status* de revogação do certificado do servidor (linhas 41 até 65). Por questões de espaço, apenas este passo é detalhado.

Nas linhas de 42 a 50, é criado um validador de cadeia de certificação (instância de `CertPathValidator`) com quatro opções configuradas: falha suavemente se não obtiver a CRL e nem a verificação por OCSP, sem mecanismo de recuperação, verifica apenas os certificados na ponta da cadeia, e prefere a verificação por CRL em vez de OCSP.

Nas linhas de 52 a 57, uma CRL associada ao certificado raiz (ancora de confiança) é recupera de modo confiável (detalhes omitidos) e adicionada aos parâmetros da verificação. Finalmente, as linhas de 60 a 65 realizam a verificação. O método `validate()` da linha 61 verifica a cadeia e a CRL passadas como parâmetros. Se exceções não ocorrem (verificação bem sucedida), as políticas de uso do certificado (linha 62) e a chave pública (linha 63) podem ser obtidas com confiança. No caso do SSL/TLS, o conteúdo da página indicada na URL é recuperado (linha 67). Senão, alguma exceção é disparada (linhas 63 e 65).

**Listagem 2.7. Validação completa de Certificados.**

```
1 public static void main(String[] args) throws Exception {
2  SSLSocket s = null; boolean ok = true;
3  try {
4    SSLSocketFactory f=(SSLSocketFactory)SSLSocketFactory.getDefault();
5    s = (SSLSocket) f.createSocket("www.google.com", 443);
6    s.startHandshake();//all validations happen after handshake
7
8    System.out.println("Session infos");
9    SSLSession ss = s.getSession();
10   System.out.println("Protocol: " + ss.getProtocol());
11   System.out.println("Ciphersuite: " + ss.getCipherSuite());
12   System.out.println("Host name: " + ss.getPeerHost());
13   Principal peer = ss.getPeerPrincipal();
14   System.out.println("SSL Peer Principal: " + peer);
15
```

```
16    // Cert, date, and Host validation
17    if (!peer.getName().contains("CN=" + ss.getPeerHost()))
18    {throw new CertificateException("Host and Principal mismatch");}
19
20    Certificate[] peerCerts = ss.getPeerCertificates();
21    if (peerCerts != null && peerCerts.length >= 2) {
22      ((X509Certificate) peerCerts[0]).checkValidity();
23      peerCerts[0].verify(peerCerts[1].getPublicKey());
24    } else {throw new CertificateException("Unable to verify cert.");}
25
26    // Cert Path validation
27    // Step 1. Obtain CA root certs and the cert path to validate
28    Certificate[] certs = ss.getPeerCertificates();
29    X509Certificate[] x509certs = new X509Certificate[certs.length-1];
30    for (int i = 0; i < certs.length-1; i++)
31      { x509certs[i] = (X509Certificate) certs[i];}
32    X509Certificate anchor = (X509Certificate) certs[certs.length-1];
33    List l = Arrays.asList(x509certs);
34    CertificateFactory cf=CertificateFactory.getInstance("X.509","SUN");
35    CertPath cp = cf.generateCertPath(l);
36
37    // Step 2. Create a PKIXParameters with the trust anchors
38    TrustAnchor ta = new TrustAnchor(anchor, null);
39    PKIXParameters params=new PKIXParameters(Collections.singleton(ta));
40
41    // Step 3. Use a CertPathValidator to validate the certificate path
42    CertPathValidator cpv=CertPathValidator.getInstance("PKIX","SUN");
43    PKIXRevocationChecker rc =
44            (PKIXRevocationChecker)cpv.getRevocationChecker();
45    rc.setOptions(EnumSet.of(PKIXRevocationChecker.Option.SOFT_FAIL));
46    rc.setOptions(EnumSet.of(PKIXRevocationChecker.Option.NO_FALLBACK));
47    rc.setOptions(EnumSet.of(
48                            PKIXRevocationChecker.Option.ONLY_END_ENTITY));
49    rc.setOptions(EnumSet.of(PKIXRevocationChecker.Option.PREFER_CRLS));
50    params.addCertPathChecker(rc);
51
52    // now it gets a valid CRL for Anchor
53    X509CRL crl = CertUtils.getCRL(anchor);// supposed valid CRL
54    List list = new ArrayList();list.add(crl);
55    CertStoreParameters csp=new CollectionCertStoreParameters(list);
56    CertStore store = CertStore.getInstance("Collection", csp);
57    params.addCertStore(store);
58
59    // validate certification path with specified params
60    PKIXCertPathValidatorResult cpvr
61            = (PKIXCertPathValidatorResult) cpv.validate(cp,params);
62    PolicyNode policyTree = cpvr.getPolicyTree();
63    PublicKey subjectPK = cpvr.getPublicKey();
64  } catch (CertificateException|InvalidAlgorithmParameterException|
65           NoSuchAlgorithmException|CertPathValidatorException e)
66  { System.out.println(e); ok = false; }
67  if (ok){System.out.println();CertUtils.handleSocket(s);}
68  else {System.out.println("Something wrong in cert validation.");}}
```

## 2.4. Maus usos criptográficos e vulnerabilidades associadas

Esta seção é organizada em torno dos maus usos de programação de criptografia assimétrica que levam a vulnerabilidades. Os maus usos são exemplificados por casos reais e ilustrados programaticamente por meio de programas em Java. Os maus usos comuns de criptografia assimétrica tratados no texto são os seguintes: encriptação determinística com RSA, problemas da versão 1 do padrão PKCS#1 (assinaturas determinísticas e *padding* inseguro), configurações fracas do RSA-OAEP, assinaturas digitais inseguras (com RSA, DSA e ECDSA), acordo de chaves DH e ECDH sem autenticação ou com chaves fracas e curvas elípticas inseguras.

### 2.4.1. Criptografia determinística com RSA

O trecho de código na listagem 2.8 mostra o uso indiscriminado do algoritmo RSA canônico, isto é, sem aleatorização, conforme descrito anteriormente no texto [Braga and Dahab 2015b] e na seção 2.1. O exemplo usa uma chave pequena de 1024 bits, insegura para os padrões atuais. Em Java, este mau uso pode ser identificado já na instanciação do algoritmo criptográfico, por exemplo, pelo método `getInstance(a)`, da classe `Cipher`, em que `a` é o nome do algoritmo. Há três opções inseguras para resolver o nome do algoritmo `a` para o RSA canônico: *(i)* apenas o nome do algoritmo, por exemplo, "RSA"; *(ii)* nome do algoritmo e modo ECB sem *padding*, por exemplo, "RSA/ECB/NoPadding"; e *(iii)* nome do algoritmo, sem modo e sem *padding*, por exemplo, "RSA/None/NoPadding". Vale observar que a primeira opção leva naturalmente ao erro, uma vez que, na falta de uma escolha explícita, o RSA canônico é a opção padrão implícita.

**Listagem 2.8. Três maneiras de RSA determinístico.**

```
1 public static void main(String args[]) {
2   try {
3     Security.addProvider(new BouncyCastleProvider()); // provedor BC
4     byte[] textoClaroAna = ("Cripto deterministica").getBytes();
5     KeyPairGenerator g = KeyPairGenerator.getInstance("RSA", "BC");
6     g.initialize(1024); KeyPair kp = g.generateKeyPair();
7     String[] rsa = {
8       "RSA","RSA/ECB/NoPadding","RSA/None/NoPadding", //deterministico
9       "RSA/None/PKCS1Padding","RSA/None/OAEPWithSHA1AndMGF1Padding"};
10    for (int a = 0; a < rsa.length; a++) {
11      Cipher enc = Cipher.getInstance(rsa[a], "BC");
12      enc.init(Cipher.ENCRYPT_MODE, kp.getPublic());
13      Cipher dec = Cipher.getInstance(rsa[a], "BC");
14      dec.init(Cipher.DECRYPT_MODE, kp.getPrivate());
15
16      U.println("Encriptado com: " + enc.getAlgorithm());
17      byte[][] criptograma = new byte[2][];
18      for (int i = 0; i < 2; i++) {
19        criptograma[i] = enc.doFinal(textoClaroAna);
20        byte[] textoClaroBeto = dec.doFinal(criptograma[i]);
21        U.println("Criptograma    : " + U.b2x(criptograma[i]));
22    }}
23  } catch (NoSuchAlgorithmException|NoSuchPaddingException|
24        InvalidKeyException|IllegalBlockSizeException|
25        BadPaddingException|NoSuchProviderException e)
26  { System.out.println(e); }}
```

### 2.4.2. Problemas do RSA no padrão PKCS#1

Esta subseção aborda em maior detalhe os problemas da versão 1 do padrão PKCS#1 a partir de três exemplos. Primeiro, a ausência de *padding* que leva à encritação com o RSA canônico. Em seguida, o uso de assinaturas digitais determinísticas com o PKCS#1. Em terceiro, encriptação com o *padding* PKCS#1v1.5, considerado inseguro, apesar de ser aleatorizado.

#### 2.4.2.1. Ausência de *padding* na encriptação com RSA

O trecho de código na listagem 2.9 mostra mais uma vez o uso do RSA canônico para encriptação. A utilização do algoritmo RSA sem *padding* aleatorizado é considerada um mau uso porque pode levar a revelação indevida de informação pela identificação de padrões de texto claro no criptograma. Isto é, quando um mesmo texto claro é encriptado mais de uma vez com mesma chave. A identificação de tais padrões pode, por exemplo, favorecer a análise de tráfego por um atacante capaz de observar a ocorrência de padrões na comunicação.

Há outros ataques sobre o RSA canônico que são mencionados pela literatura especializada [Boneh 1999], dentre eles há dois ataques considerados simples. O ataque de *broadcast* acontece quando a mesma mensagem $m$ é encriptada para $e_i$ destinatários diferentes, onde $e_i$ é o valor de alguma chave privada $i$. Cada destinatário tem seu próprio par de chaves. Nesta caso, é possível decriptar a mensagem sem precisar de qualquer chave privada $d$. A solução para este ataque, além de um *padding* seguro, é usar um sistema criptográfico híbrido com chaves assimétricas para transporte de chaves simétricas.

O ataque da $e$-nézima raiz de $c$ acontece quando a mensagem $m$ é muito pequena e o valor de $e$ é muito baixo (por exemplo, no ataque da raiz cúbica com $e = 3$). Neste caso, o criptograma $c = m^e$ é menor que o módulo $n$ e a decriptação pode ser obtida simplesmente pela computação da $e$-nézima raiz de $c$.

**Listagem 2.9. RSA canônico sem *padding*.**

```
1 public static void main(String args[]) {
2   try {
3     Security.addProvider(new BouncyCastleProvider()); // provedor BC
4     byte[] pt = ("Cripto deterministica").getBytes();
5     KeyPairGenerator g = KeyPairGenerator.getInstance("RSA","BC");
6     g.initialize(2048); KeyPair kp = g.generateKeyPair();
7
8     Cipher e = Cipher.getInstance("RSA/None/NoPadding","BC");
9     e.init(Cipher.ENCRYPT_MODE, kp.getPublic());
10    Cipher d = Cipher.getInstance("RSA/None/NoPadding","BC");
11    d.init(Cipher.DECRYPT_MODE, kp.getPrivate());
12    U.println("Texto claro: " + U.b2x(pt));
13    U.println("Encriptado com: " + e.getAlgorithm());
14    U.println("Criptograma: " + U.b2x(e.doFinal(pt)));
15    U.println("Criptograma: " + U.b2x(e.doFinal(pt)));
16    U.println("Criptograma: " + U.b2x(e.doFinal(pt)));
17    U.println("Texto claro: " + U.b2x(d.doFinal(e.doFinal(pt))));
18  } catch (NoSuchAlgorithmException|NoSuchPaddingException|
19           InvalidKeyException|IllegalBlockSizeException |
20           BadPaddingException|NoSuchProviderException e)
21  { System.out.println(e);}
```

#### 2.4.2.2. Assinatura digital RSA determinística com PKCS#1v1

O trecho de código na listagem 2.10 mostra a assinatura digital determinística com RSA sobre um *hash* (no exemplo, SHA-512). O programa exemplo utiliza dois provedores criptográficos diferentes, o "SunRSASign"para assinatura e o "BC"para verificação. O contrário também funcionaria, porque os provedores são intercambiáveis e interoperáveis. Este foi o formato inicialmente usado no padrão PKCS#1 para assinaturas digitais e tinha a finalidade de evitar o uso do RSA canônico que é susceptível ao ataque de falsificação de assinaturas conhecido como *Blinding* [Boneh 1999].

Na listagem 2.10, as linhas de 2 a 6 faze o seguinte. A linha 2 adiciona dinamicamente o provedor criptográfico *BouncyCastle* (BC). A linha 3 instancia um gerador de par de chaves RSA do provedor BC. A linha 4 configura o gerador para chaves com 2048 bits de tamanho e um PRNG seguro, enquanto a linha 5 cria um par de chaves nas configurações indicadas.

As linhas de 7 a 17 realizam o processo de geração da assinatura digital. A linha 7 cria uma instancia do objeto assinador "SHA512withRSA"do provedor "SunRsaSign", enquanto a linha 8 declara o texto claro usado no exemplo. A linha 10 configura o assinador com a chave pública e uma instância de um PRNG seguro. As linhas de 10 a 16 configuram o objeto assinador 3 vezes com a mesma mensagem e PRNGs diferentes. Mesmo assim, as três assinaturas são idênticas. Vale observar que o formato "SHA512withRSA"não usa o PRNG, apesar da API permitir um objeto deste tipo. Esta é uma inconsistência da API.

As linhas de 19 a 22 realizam o processo de verificação da assinatura. A linha 19 cria um objeto assinatura "SHA1withRSA"do provedor "BC", enquanto a linha 21 configura este objeto para o modo de verificação com a chave pública e alimenta o verificador com a mensagem que foi assinada. Finalmente, a linha 22 faz a verificação da assinatura digital.

**Listagem 2.10. Assinatura digital RSA determinística.**

```
1 public static void main(String[] args) throws Exception {
2   Security.addProvider(new BouncyCastleProvider()); // provedor BC
3   KeyPairGenerator kg = KeyPairGenerator.getInstance("RSA", "BC");
4   kg.initialize(2048, new SecureRandom());
5   KeyPair kp = kg.generateKeyPair();
6
7   Signature sig = Signature.getInstance("SHA512withRSA","SunRsaSign");
8   byte[] m = "Testing RSA Signature PKCS1".getBytes();
9
10  sig.initSign(kp.getPrivate(), SecureRandom.getInstanceStrong());
11  sig.update(m); U.println("Signature: "+ U.b2x(sig.sign()));
12  sig.initSign(kp.getPrivate(), SecureRandom.getInstanceStrong());
13  sig.update(m); U.println("Signature: "+ U.b2x(sig.sign()));
14  sig.initSign(kp.getPrivate(), SecureRandom.getInstanceStrong());
15  sig.update(m); byte[] s = sig.sign();// generate a signature
16  sig.update(m); U.println("Signature: "+ U.b2x(s));
17
18  // verify a signature
19  Signature verifier = Signature.getInstance("SHA512withRSA","BC");
20  System.out.println("Algorithm: "+ sig.getAlgorithm());
21  verifier.initVerify(kp.getPublic()); verifier.update(m);
22  if (verifier.verify(s)) { U.println("Verification succeeded.");}
23  else                    { U.println("Verification failed.");    }}
```

#### 2.4.2.3. Encriptação com RSA PKCS#1 v1.5

O trecho de código na Listagem 2.11 descreve a encriptação RSA com o preenchimento (ou *padding*) conforme o padrão PKCS#1 v1.5. Este padrão de preenchimento é, em certos casos, considerado inseguro [Bleichenbacher 1998], devido a sua susceptibilidade ao ataque de *padding oracle*, mas que ainda é bastante utilizado no SSL/TLS até a versão v1.2. Na listagem 2.11, as linhas 3 a 6 definem o provedor criptográfico e o texto claro usados no exemplo e criam o par de chaves com tamanho de 2048 bits.

O *padding* é feito conforme a ilustração a seguir, onde "02"são 16 bits indicando que o criptograma possui um *padding* PKCS#1, *nonce* é um valor pseudoaleatório de uso único e "00"é um separador de 16 bits entre o texto claro original e o *nonce*.

| 02 | *nonce* | 00 | texto claro |
|---|---|---|---|

Apesar da vulnerabilidade do PKCS#1v1.5 ao ataque de *padding oracle* ter sido publicada já há vinte anos, em 1998 [Bleichenbacher 1998], ainda hoje é possível achar implementações do PKCS#1v1.5 em sistemas criptográficos em produção e padrões em uso, como o SSL/TLS v1.2. Este fato levanta a questão de como é difícil seguir as boas práticas.

Ainda na Listagem 2.11, as linhas de 8 a 11 criam duas máquinas criptográficas para o RSA com *padding* PKCS#1v1.5, indicado pela *string* "RSA/None/PKCS1Padding": Uma para encriptação (configurada com a chave pública) e outra de decriptação (usando a chave privada). Finalmente, as linhas de 14 a 16 realizam três encriptações sucessivas do mesmo texto claro para mostrar que elas são diferentes de fato. A linha 17 faz a decriptação. As linhas de 18 a 21 mostram todas as exceções tratadas para o RSA PKCS#1v1.5 em Java.

**Listagem 2.11. RSA com PKCS#1.**

```
1 public static void main(String args[]) {
2   try {
3     Security.addProvider(new BouncyCastleProvider());// provedor BC
4     byte[] ptAna = ("Randomized RSA").getBytes();
5     KeyPairGenerator g = KeyPairGenerator.getInstance("RSA","BC");
6     g.initialize(2048); KeyPair kp = g.generateKeyPair();
7
8     Cipher e = Cipher.getInstance("RSA/None/PKCS1Padding","BC");
9     e.init(Cipher.ENCRYPT_MODE, kp.getPublic());
10    Cipher d = Cipher.getInstance("RSA/None/PKCS1Padding","BC");
11    d.init(Cipher.DECRYPT_MODE, kp.getPrivate());
12
13    U.println("Plaintext : " + new String(ptAna));
14    U.println("Ciphertext: " + U.b2x(e.doFinal(ptAna)));
15    U.println("Ciphertext: " + U.b2x(e.doFinal(ptAna)));
16    U.println("Ciphertext: " + U.b2x(e.doFinal(ptAna)));
17    U.println("Plaintext : " + U.b2x(d.doFinal(e.doFinal(ptAna))));
18  } catch (NoSuchAlgorithmException | NoSuchPaddingException |
19          InvalidKeyException | IllegalBlockSizeException |
20          BadPaddingException | NoSuchProviderException e)
21  { System.out.println(e); }}
```

### 2.4.3. Configurações fracas do RSA-OAEP

A Figura 2.8 mostra as combinações possíveis para funções de *hash* e tamanho de chaves (módulo) do RSA-OAEP, até o valor de 15360 de módulo (nível de segurança de 256 bits). Cada célula da figura tem a quantidade de bits processada pelo RSA-OAEP. As combinações tachadas têm níveis de segurança menores que 112 bits. As combinações sublinhadas têm nível de segurança de 112 bits. As combinações em negrito com módulos de 3072, 7680 e 15360 têm 128, 192 e 256 bits de segurança, respectivamente.

| Tamanho da chave RSA OAEP | Tamanho do hash | | | | Tamanho máximo do texto claro |
|---|---|---|---|---|---|
| Módulo | 160 | 256 | 384 | 512 | |
| 256 | | | | | |
| 384 | ~~48~~ | | | | |
| 512 | ~~176~~ | | | | |
| 768 | ~~432~~ | ~~240~~ | | | |
| 1024 | ~~688~~ | ~~496~~ | ~~240~~ | | |
| 2048 | ~~1712~~ | *1520* | *1264* | *1008* | |
| **3072** | ~~2736~~ | **2544** | **2288** | **2032** | |
| **4096** | ~~3760~~ | **3568** | **3312** | **3056** | |
| **7680** | ~~7344~~ | *7152* | **6896** | **6640** | |
| **15360** | ~~15024~~ | *14832* | *14576* | **14320** | |
| | Tamanho máximo do texto claro | | | | |

**Figura 2.8. Combinações de tamanho de chave (módulo) e *hash* para o RSA-OAEP.**

O trecho de código na Listagem 2.12 mostra um exemplo de configuração insegura do RSA-OAEP em que as chaves tem tamanho de 1024 bits (linha 5) e o *hash* é de 160 bits, conforme a *string* "RSA/None/OAEPwithSHA1andMGF1Padding"(linha 7). Este código processa apenas 86 bytes de cada vez (linha 10).

**Listagem 2.12. RSA-OAEP com chave de 1024 bits e SHA-1.**

```
1 public static void main(String args[]) {
2  try {
3    Security.addProvider(new BouncyCastleProvider());
4    KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA","BC");
5    kpg.initialize(1024); KeyPair kp = kpg.generateKeyPair();
6
7    Cipher c=Cipher.getInstance("RSA/None/OAEPwithSHA1andMGF1Padding");
8
9    c.init(Cipher.ENCRYPT_MODE, kp.getPublic());
10   byte[] ptAna = U.cancaoDoExilio.substring(0,86).getBytes();
11   byte[] ct = c.doFinal(ptAna);
12
13   c.init(Cipher.DECRYPT_MODE, kp.getPrivate());
14   byte[] ptBeto = c.doFinal(ct); // Decriptando
15 } catch (NoSuchAlgorithmException | NoSuchPaddingException |
16         InvalidKeyException      | BadPaddingException     |
17         IllegalBlockSizeException | NoSuchProviderException e)
18 {System.out.println(e);}}
```

### 2.4.4. Assinaturas digitais inseguras

Esta subseção trata vulnerabilidades associadas às assinaturas digitais fracas. A primeira é a repetição do *nonce (number used once)* na geração de assinaturas ECDSA e a segunda é a utilização de configurações dos esquemas de assinatura ECDSA, DSA e RSA.

#### 2.4.4.1. *Nonce* repetido na assinatura ECDSA

O ECDSA pode ser mal utilizado e produzir assinaturas digitais vulneráveis. No ECDSA, cada assinatura requer um número aleatório único e imprevisível, que deve ser utilizado apenas uma vez (*nonce*). Por isto, o ECDSA é vulnerável à geração de números pseudoaleatórios ruins. Aplicações de assinaturas digitais ECDSA são muito sensíveis aos maus usos do ECDSA, tais como sementes de PRNGs com entropia baixa e *nonces* repetidos (com valores fixos em programas). Assinaturas digitais geradas com o mesmo *nonce*, ou *nonces* parecidos (com alguns bits em comum) podem revelar a chave privada em aplicações reais como *eWallets* de *bitcoins* [Bos et al. 2014, Braga et al. 2017b].

Na Listagem 2.13, as linhas 6 e 7 instanciam um objeto `SecureRandom` do tipo "SHA1PRNG"e o configuram com uma semente de 24 bytes. As linhas de 9 a 12 realizam os passos para geração da assinatura digital. A linha 9 instancia um assinador para o "SHA256withECDSA". A linha 10 configura o assinador com a chave privada e o PRNG. A linha 12 calcula a assinatura digital sobre o documento. As linhas de 14 a 20 repetem os passos de instanciação e configuração de um assinador e geração da assinatura digital para o mesmo documento assinado anteriormente, mas, desta vez, reutilizando a mesma semente para o "SHA1PRNG". Por isso, as duas assinaturas digitais comparadas são iguais (linha 22).

**Listagem 2.13. Nonce repetido com ECDSA.**

```
1 public static void main(String[] args) throws Exception {
2   KeyPairGenerator kpg = KeyPairGenerator.getInstance("EC","SunEC");
3   kpg.initialize(256, SecureRandom.getInstanceStrong());
4   KeyPair kpAna = kpg.generateKeyPair();
5
6   SecureRandom sr1 = SecureRandom.getInstance("SHA1PRNG","SUN");
7   byte[] seed = sr1.generateSeed(24); sr1.setSeed(seed);
8
9   Signature signer1 = Signature.getInstance("SHA256withECDSA","SunEC");
10  signer1.initSign(kpAna.getPrivate(), sr1);
11  byte[] doc = U.cancaoDoExilio.getBytes();
12  signer1.update(doc); byte[] sign1 = signer1.sign();
13
14  SecureRandom sr2 = SecureRandom.getInstance("SHA1PRNG","SUN");
15  sr2.setSeed(seed);
16
17  Signature signer2 = Signature.getInstance("SHA256withECDSA","SunEC");
18  signer2.initSign(kpAna.getPrivate(), sr2);
19  doc = U.cancaoDoExilio.getBytes();
20  signer2.update(doc); byte[] sign2 = signer2.sign();
21
22  boolean ok = Arrays.areEqual(sign1, sign2);
23  if (ok) {U.println("Nonce repeated! Signatures are equal!");}}
```

#### 2.4.4.2. Configurações fracas de assinatura ECDSA

A Listagem 2.14 ilustra como uma curva elíptica insegura pode ser utilizada de modo implícito pelas configurações da assinatura digital ECDSA. No programa exemplo, a linha 3 mostra a criação de um par de chaves de 112 bits para curvas elípticas. Na linha 4, um objeto assinador é criado para "SHA1withECDSA". As curvas de 112 bits combinadas com SHA-1 (160 bits) resultam em segurança de 80 bits apenas. As linhas de 7 a 10 realizam os passos para geração da assinatura digital: configuram o assinador com a chave privada, parametrizam o assinador com o documento a ser assinado e calculam a assinatura digital do documento. Já as linhas de 11 a 13 realizam os passos para a verificação da assinatura digital com a chave pública.

**Listagem 2.14. ECDSA com chave de 112 bits mas segurança de 80 bits.**

```
1 public static void main(String[] args) throws Exception {
2    KeyPairGenerator kpg = KeyPairGenerator.getInstance("EC","SunEC");
3    kpg.initialize(112, SecureRandom.getInstanceStrong());
4    Signature signer = Signature.getInstance("SHA1withECDSA","SunEC");
5    KeyPair kp = kpg.generateKeyPair();
6
7    signer.initSign(kp.getPrivate(), new SecureRandom());
8    byte[] doc = U.cancaoDoExilio.getBytes();
9    signer.update(doc); byte[] sign = signer.sign();
10
11   Signature verifier = Signature.getInstance("SHA1withECDSA","SunEC");
12   verifier.initVerify(kp.getPublic()); verifier.update(doc);
13   if(!verifier.verify(sign)){System.out.println("Signature not OK!");}
```

#### 2.4.4.3. Configurações fracas de assinatura RSA

O trecho de código da Listagem 2.15 mostra na linha 4 três configurações fracas de assinaturas digitais RSA com *hashes*: MD2withRSA, MD5withRSA e SHA1withRSA. As função de *hash* MD2 é obsoleta, o MD5 já foi quebrado e o SHA-1 foi descontinuado. Por isto, assinaturas sobre estes *hashes* são inseguras. Chaves de 1024 bits adicionam outra vulnerabilidade ao código.

**Listagem 2.15. Três configurações inseguras de assinaturas digitais RSA.**

```
1 public static void main(String[] args) throws Exception {
2   KeyPairGenerator
      kpg=KeyPairGenerator.getInstance("RSA","SunRsaSign");
3   kpg.initialize(1024,SecureRandom.getInstanceStrong());
4   String[] weakrsa = {"MD2withRSA", "MD5withRSA", "SHA1withRSA"};
5   byte[] doc = U.cancaoDoExilio.getBytes();
6   for (String rsa : weakrsa) {
7     Signature signer = Signature.getInstance(rsa, "SunRsaSign");
8     KeyPair kp1 = kpg.generateKeyPair();
9     signer.initSign(kp1.getPrivate(), new SecureRandom());
10    signer.update(doc);byte[] sign = signer.sign();
11
12    Signature verifier = Signature.getInstance(rsa, "SunRsaSign");
13    verifier.initVerify(kp1.getPublic()); verifier.update(doc);
14    if (verifier.verify(sign)){U.println("Signature OK!");}}}
```

#### 2.4.4.4. Configurações fracas de assinatura DSA

O trecho de código da Listagem 2.16 mostra duas configurações inseguras de assinaturas digitais DSA: "SHA1withDSA"e "NONEwithDSA". No primeiro caso, a assinatura tem o nível de segurança do elemento mais fraco, o SHA-1, com segurança menor que 80 bits, apesar das chaves de 2048 bits. No segundo caso, o "NONEwithDSA"permite que um *hash* seja calculado externamente ao assinador. Porém, a API só permite que o SHA-1 seja usado para cálculo do *hash* devido a uma restrição ao tamanho dos dados passados como parâmetros para o assinador, que devem ser de exatamente 20 bytes (128 bits), o tamanho da saída do SHA-1.

Ainda na Listagem 2.16, as linhas de 2 a 4 criam um par de chaves DSA de 2048 bits. As linhas 6 e 7 identificam as esquemas de assinatura digital usados no exemplo e o texto claro, o documento, a ser assinado. O laço das linhas 10 até 23 realizam os passos de geração e verificação de assinatura para os dois esquemas do exemplo: "SHA1withDSA"e "NONEwithDSA".

Primeiro, nas linhas de 10 a 15 criam o objeto assinador DSA, que é configurado com a chave privada e um PRNG. Para o caso da assinatura sobre *hash* externo, uma instância do `MessageDigest` SHA-1 é criada e usada para calcular o *hash* do documento. Na linha 16, o assinador recebe o documento a ser assinado ou o *hash* do documento e gera a assinatura digital.

Em seguida, nas linhas de 18 a 20, um verificador de assinaturas para o DSA é criado, configurado com a chave pública e alimentado com o documento (ou o *hash* do documento, no caso do *hash* externo), que foi assinado. Enquanto a linha 21 contém a verificação da assinatura propriamente, com o método `verify()` do verificador.

**Listagem 2.16. Duas configurações inseguras de assinaturas digitais DSA.**

```
1 public static void main(String[] args) throws Exception {
2   KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA", "SUN");
3   kpg.initialize(2048, SecureRandom.getInstanceStrong());
4   KeyPair kp = kpg.generateKeyPair();
5
6   String[] weakdsa = {"SHA1withDSA","NONEwithDSA"};
7   byte[] doc = U.cancaoDoExilio.getBytes();
8
9   for (String dsa : weakdsa) {
10    Signature signer = Signature.getInstance(dsa, "SUN");
11    signer.initSign(kp.getPrivate(), new SecureRandom());
12    if (dsa.equals("NONEwithDSA")){
13      MessageDigest md = MessageDigest.getInstance("SHA1");
14      doc = md.digest(doc);//Data must be exactly 20 bytes
15    }
16    signer.update(doc); byte[] sign = signer.sign();
17
18    Signature verifier = Signature.getInstance(dsa, "SUN");
19    verifier.initVerify(kp.getPublic()); verifier.update(doc);
20
21    if (verifier.verify(sign)){U.println("Signature OK!");}
22    else                       {U.println("Signature not OK!");}
23    }}
```

### 2.4.5. Acordo de chaves sem autenticação

Esta seção contempla os maus usos criptográficos associados aos protocolos de acordo de chaves baseados em Diffie-Hellman (DH) tradicional e sobre curvas elípticas (ECDH). A seção mostra quatro programas exemplo bastante parecidos entre si, mas com variações sutis que indicam o mau uso: DH sem autenticação com parâmetros estáticos ou dinâmicos e ECDH sem autenticação com parâmetros estáticos ou dinâmicos.

Conforme mencionado anteriormente na seção 2.2, tanto o DH quanto o ECDH não possuem autenticação intrínseca das trocas de mensagens entre os participantes do protocolo. A capacidade de autenticação das mensagens é obtida externamente pela combinação do protocolo com um esquema de assinaturas digitais. Os provedores criptográficos comumente usados em Java não possuem implementações do acordo de chaves autenticado, seja para DH ou ECDH. No SSL/TLS, o DH/ECDH sem autenticação é chamado de DH/ECDH anônimo, passando uma falsa sensação de segurança por anonimato para os usuários deste protocolo. Atualmente, o DH/ECDH anônimo do SSL/TLS é considerado inseguro.

#### 2.4.5.1. DH estático não autenticado

Chama-se de DH estático à instância do DH clássico em que os parâmetros do protocolo compartilhados entre os participantes (o gerador $g$ e o primo $p$) são definidos uma vez (possivelmente *a priori*) e reutilizados em várias execuções do protocolo. O trecho de código da Listagem 2.17 mostra uma implementação didática do DH não autenticado e com parâmetros estáticos, em que as duas partes comunicantes (Alice e Bob) são implementadas no mesmo código. A Listagem 2.17 contém o primeiro exemplo a explorar o acordo de chaves em Java e por isto será explicado em maior detalhe que os outros programas semelhantes.

As linhas de 3 a 8 são responsáveis por criar os parâmetros estáticos. Um gerador de parâmetros DH do provedor SunEC é instanciado na linha 4 e inicializado (linha 5) para chaves com 1024 bits de tamanho, enquanto as linhas 6 contém a geração dos parâmetros propriamente. Já as linhas 7 e 8 contém a formação dos parâmetros de acordo com a especificação da API.

As linha de 10 a 15 realizam os passos de criação de uma par de chaves DH para Alice e inicialização do protocolo DH. A linha 11 instancia um gerador e par de chaves para o DH. Na linha 12, o gerador de chaves é configurado com os parâmetros calculados anteriormente e um par de chaves é gerado de acordo. A linha 13 instancia um acordo de chaves DH, que é configurado com a chave privada da Alice (linha 14). A chave pública da Alice é codificada na linha 15, para que seja enviada para Bob.

As linhas de 17 a 24 realizam os passos do protocolo do ponto de vista do Bob. A linha 18 cria uma fábrica de chaves DH que recebe a chave pública da Alice (que foi codificada no passo anterior e transportada até Bob) e a converte (linha 19) em um instancia válida de chave pública da Alice que é mantida por Bob (linha 20). As linhas de 22 a 24 criam um par de chaves DH para Bob (linha 24) com base nos parâmetros compartilhados com a Alice, obtidos da chave pública da Alice (linha 22).

A instância do acordo de chaves, do ponto de vista de Bob, é criada na linha 26 e configurada com a chave privada de Bob na linha 27. Já a linha 28 codifica a chave pública de Bob para que seja enviada para Alice.

As linhas de 30 a 35 voltam para o ponto de vista da Alice. A linha 31 cria uma fábrica de chaves DH que recebe a chave pública da Bob (que foi codificada no passo anterior e transportada até Alice) e a converte (linha 32) em um instancia válida de chave pública da Alice que é mantida por Bob (linha 33). Já a linha 34 realizada o segundo passo do DH sobre a chave pública de Bob e calcula o segredo da Alice.

A linha 37 volta o protocolo para o ponto de vista de Bob, realizando o segundo passo do DH sobre a chave pública de Alice e calculando o segredo de Bob. Neste momento, Alice e Bob concluíram o protocolo e chegaram ao mesmo segredo. Na prática uma mensagem autenticada por um HMAC serve de evidência entre as partes que ambas compartilham um segredo.

**Listagem 2.17. DH não autenticado e com parâmetros estáticos.**

```
1 public static void main(String argv[]) {
2  try {
3    AlgorithmParameterGenerator apg
4        = AlgorithmParameterGenerator.getInstance("DH","SunJCE");
5    apg.init(1024); // here is the size
6    AlgorithmParameters p = apg.generateParameters();
7    DHParameterSpec dhps
8        = (DHParameterSpec) p.getParameterSpec(DHParameterSpec.class);
9
10   //Alice starts DH by creating a key pair
11   KeyPairGenerator aKPG = KeyPairGenerator.getInstance("DH","SunJCE");
12   aKPG.initialize(dhps); KeyPair aKP = aKPG.generateKeyPair();
13   KeyAgreement aKA = KeyAgreement.getInstance("DH","SunJCE");
14   aKA.init(aKP.getPrivate());
15   byte[] aPubKe = aKP.getPublic().getEncoded();
16
17   // Let's turn over to Bob.
18   KeyFactory bKF = KeyFactory.getInstance("DH","SunJCE");
19   X509EncodedKeySpec x509ks = new X509EncodedKeySpec(aPubKe);
20   PublicKey aPubK = bKF.generatePublic(x509ks);
21
22   DHParameterSpec dhps2 = ((DHPublicKey) aPubK).getParams();
23   KeyPairGenerator bKPG = KeyPairGenerator.getInstance("DH","SunJCE");
24   bKPG.initialize(dhps2); KeyPair bKP = bKPG.generateKeyPair();
25
26   KeyAgreement bKA = KeyAgreement.getInstance("DH","SunJCE");
27   bKA.init(bKP.getPrivate());
28   byte[] bPubKe = bKP.getPublic().getEncoded();
29
30   // Alice uses Bob's public key
31   KeyFactory aKF = KeyFactory.getInstance("DH","SunJCE");
32   x509ks = new X509EncodedKeySpec(bPubKe);
33   PublicKey bPubK = aKF.generatePublic(x509ks);
34   aKA.doPhase(bPubK, true); byte[] aSecret = aKA.generateSecret();
35
36   // Bob uses Alice's public key
37   bKA.doPhase(aPubK, true); byte[] bSecret = bKA.generateSecret();
38
39   //Alice and Bob completed DH key agreement protocol.
40   if (!Arrays.equals(aSecret,bSecret)) // not this way in real life
41        {throw new Exception("Secrets differ");}
42 }catch(Exception e){System.err.println("Error:"+e); System.exit(1);}}
```

### 2.4.5.2. DH efêmero não autenticado

Chama-se de DH efêmero à instância do DH clássico em que os parâmetros compartilhados (o gerador *g* e o primo *p*) são redefinidos para cada execução do protocolo e sempre descartados ao final da execução, sem reuso dos parâmetros. O trecho de código da Listagem 2.18 mostra uma implementação didática do DH não autenticado e com parâmetros descartáveis.

Em linhas gerais, a Listagem 2.18 é bastante parecida com aquela mostrada anteriormente, mas com uma diference importante: os parâmetros compartilhados (*g* e *p*) não são gerado explicitamente, mas sim criados implicitamente na geração do par de chaves de Alice. As linha de 3 a 7 realizam os passos de criação de uma par de chaves DH para Alice e a inicialização do protocolo DH. A linha 3 instancia um gerador e par de chaves para o DH. Na linha 4, o gerador de chaves é configurado para chaves com 1024 bits de tamanho (fracas) e um par de chaves é gerado de acordo (linha 5). A linha 6 instancia um acordo de chaves DH, que é configurado com a chave privada da Alice (linha 7). Os passos análogos para Bob estão nas linhas de 15 a 19.

**Listagem 2.18. DH não autenticado e com parâmetros descartáveis ou efêmeros.**

```
1 public static void main(String argv[]) {
2  try {
3    KeyPairGenerator aKPG=KeyPairGenerator.getInstance("DH","SunJCE");
4    aKPG.initialize(1024);
5    KeyPair aKP = aKPG.generateKeyPair();
6    KeyAgreement aKA = KeyAgreement.getInstance("DH", "SunJCE");
7    aKA.init(aKP.getPrivate());
8
9    byte[] aPubK = aKP.getPublic().getEncoded();
10
11   KeyFactory bKF = KeyFactory.getInstance("DH", "SunJCE");
12   X509EncodedKeySpec x509ks = new X509EncodedKeySpec(aPubK);
13   PublicKey apk = bKF.generatePublic(x509ks);
14
15   KeyPairGenerator bKPG=KeyPairGenerator.getInstance("DH","SunJCE");
16   bKPG.initialize(1024);
17   KeyPair bKP = bKPG.generateKeyPair();
18   KeyAgreement bKA = KeyAgreement.getInstance("DH", "SunJCE");
19   bKA.init(bKP.getPrivate());
20
21   byte[] bPubK = bKP.getPublic().getEncoded();
22
23   KeyFactory aKF = KeyFactory.getInstance("DH", "SunJCE");
24   x509ks = new X509EncodedKeySpec(bPubK);
25   PublicKey bobPubKey = aKF.generatePublic(x509ks);
26   aKA.doPhase(bobPubKey, true);
27   byte[] aSecret = aKA.generateSecret();
28
29   bKA.doPhase(apk, true);
30   byte[] bSecret = bKA.generateSecret();
31
32   if(!Arrays.equals(aSecret,bSecret)) // not this way in real life
33        throw new Exception("Secrets differ");
34 }catch(Exception e){System.err.println("Error: "+e);System.exit(1);}}
```

#### 2.4.5.3. ECDH efêmero não autenticado

O trecho de código da Listagem 2.19 é análogo ao anterior e mostra uma implementação didática do ECDH não autenticado e com parâmetros descartáveis. Em Java, a única diferença relevante entre o ECDH e o DH tradicional está na geração do par de chaves. As linha de 3 a 7 realizam os passos de criação de uma par de chaves ECDH para Alice. A linha 3 instancia um gerador e um par de chaves para o ECDH. Na linha 4, o gerador de chaves é configurado para chaves com 224 bits de tamanho (chaves com apenas 112 bits de segurança) e um par de chaves é gerado de acordo (linha 5).

Os passos análogos, do ponto de vista de Bob, acontecem nas linhas de 15 e 16. Vale mencionar uma vantagem da criptografia de curvas elípticas, as chaves ECDH são bem menores que as chaves correspondentes (no mesmo nível de segurança) do DH clássico. Em particular, a chave de 224 bits usada neste exemplo do ECDH correspondem a uma chave de 2048 bits no DH.

**Listagem 2.19. ECDH não autenticado e com parâmetros efêmeros.**

```
1 public static void main(String argv[]) {
2  try {
3    KeyPairGenerator aKPG = KeyPairGenerator.getInstance("EC", "SunEC");
4    aKPG.initialize(224); // only 112 bits of security
5    KeyPair aKP = aKPG.generateKeyPair();
6
7    KeyAgreement aKA = KeyAgreement.getInstance("ECDH","SunEC");
8    aKA.init(aKP.getPrivate());
9    byte[] aPubKe = aKP.getPublic().getEncoded();// this goes to Bob
10
11   KeyFactory bKF = KeyFactory.getInstance("EC", "SunEC");
12   X509EncodedKeySpec x509ks = new X509EncodedKeySpec(aPubKe);
13   PublicKey apk = bKF.generatePublic(x509ks);
14
15   KeyPairGenerator bKPG = KeyPairGenerator.getInstance("EC","SunEC");
16   bKPG.initialize(224); KeyPair bKP = bKPG.generateKeyPair();
17
18   KeyAgreement bKA = KeyAgreement.getInstance("ECDH","SunEC");
19   bKA.init(bKP.getPrivate());
20
21   byte[] bPubKe = bKP.getPublic().getEncoded();//this goes to Alice
22
23   KeyFactory aKF = KeyFactory.getInstance("EC","SunEC");
24   x509ks = new X509EncodedKeySpec(bPubKe);
25   PublicKey bPubK = aKF.generatePublic(x509ks);
26   aKA.doPhase(bPubK, true);
27   byte[] aSecret = aKA.generateSecret();
28
29   bKA.doPhase(apk, true);
30   byte[] bSecret = bKA.generateSecret();
31
32   if(!Arrays.equals(aSecret,bSecret)) // not this way in real life
33       throw new Exception("Secrets differ");
34 }catch(Exception e){System.err.println("Error: "+e);System.exit(1);}}
```

### 2.4.6. Acordo de chaves com chaves assimétricas fracas

A utilização de chaves assimétricas fracas é um mau uso de criptografia comum em sistemas criptográficos de chave pública e, em relação aos protocolos de acordo de chaves, pode ocorrer tanto no DH quanto no ECDH, comprometendo a segurança do segredo compartilhado, que foi gerado por uma rodada do protocolo com chaves assimétricas inseguras.

#### 2.4.6.1. Chaves assimétricas fracas com DH

O trecho de código da Listagem 2.20 é análogo aos exemplos anteriores e ilustra o uso de chaves assimétricas fracas no DH. Na linha 4, um gerador de par de chaves é configurado para gerar chaves DH de 512 bits e um par de chaves é gerado para Alice com este tamanho, na linha 5. O par de chaves de Bob é gerado nas linhas de 16 a 18, também com tamanho de 512 bits. Atualmente, qualquer tamanho de chaves DH menor que 2048 já é considerado inseguro.

**Listagem 2.20. DH não autenticado e chaves fracas.**

```
1 public static void main(String argv[]) {
2  try {
3    KeyPairGenerator aKPG = KeyPairGenerator.getInstance("DH","SunJCE");
4    aKPG.initialize(512);
5    KeyPair aKP = aKPG.generateKeyPair();
6
7    KeyAgreement aKA = KeyAgreement.getInstance("DH", "SunJCE");
8    aKA.init(aKP.getPrivate());
9
10   byte[] aPubKe = aKP.getPublic().getEncoded();
11
12   KeyFactory bKF = KeyFactory.getInstance("DH", "SunJCE");
13   X509EncodedKeySpec x509ks = new X509EncodedKeySpec(aPubKe);
14   PublicKey aPubK = bKF.generatePublic(x509ks);
15
16   KeyPairGenerator bKPG = KeyPairGenerator.getInstance("DH","SunJCE");
17   bKPG.initialize(512);
18   KeyPair bKP = bKPG.generateKeyPair();
19
20   KeyAgreement bKA = KeyAgreement.getInstance("DH","SunJCE");
21   bKA.init(bKP.getPrivate());
22
23   byte[] bPubKe = bKP.getPublic().getEncoded();
24
25   KeyFactory aKF = KeyFactory.getInstance("DH","SunJCE");
26   x509ks = new X509EncodedKeySpec(bPubKe);
27   PublicKey bPubK = aKF.generatePublic(x509ks);
28   aKA.doPhase(bPubK, true);
29   byte[] aSecret = aKA.generateSecret();
30
31   bKA.doPhase(aPubK, true);
32   byte[] bSecret = bKA.generateSecret();
33
34   if(!Arrays.equals(aSecret,bSecret)) // not this way in real life
35       throw new Exception("Secrets differ");
36 }catch(Exception e){System.err.println("Error: "+e);System.exit(1);}}
```

#### 2.4.6.2. Chaves assimétricas fracas com ECDH

O trecho de código da Listagem 2.21 é análogo aos exemplos anteriores e ilustra o uso de chaves assimétricas fracas no ECDH. Na linha 4, um gerador de par de chaves é configurado para gerar chaves ECDH de 160 bits de tamanho (80 bits de segurança) e um par de chaves é gerado para Alice, na linha 5. O par de chaves de Bob é gerado nas linhas de 16 a 18, também com tamanho de 160 bits.

Atualmente, qualquer tamanho de chaves ECDH menor que 224 bits (112 bits de segurança) já é considerado inseguro. Além disso, o leitor não deve se enganar com a diferença de tamanho entre as chaves DH e ECDH, uma chave DH de 512 bits é muito mais fraca que uma chave ECDH de 160 bits e ambas são inseguras. A próxima seção aborda curvas elípticas inseguras e os tamanhos de chaves correspondentes.

**Listagem 2.21. CEDH não autenticado e com chaves fracas.**

```
1 public static void main(String argv[]) {
2  try {
3    KeyPairGenerator aKPG = KeyPairGenerator.getInstance("EC", "SunEC");
4    aKPG.initialize(160); // this has only 80 bits of security
5    KeyPair aKP = aKPG.generateKeyPair();
6
7    KeyAgreement aKA = KeyAgreement.getInstance("ECDH", "SunEC");
8    aKA.init(aKP.getPrivate());
9
10   byte[] aPubKe = aKP.getPublic().getEncoded();
11
12   KeyFactory bKF = KeyFactory.getInstance("EC", "SunEC");
13   X509EncodedKeySpec x509ks = new X509EncodedKeySpec(aPubKe);
14   PublicKey aPubK = bKF.generatePublic(x509ks);
15
16   KeyPairGenerator bKPG = KeyPairGenerator.getInstance("EC", "SunEC");
17   bKPG.initialize(160);
18   KeyPair bKP = bKPG.generateKeyPair();
19   KeyAgreement bKA = KeyAgreement.getInstance("ECDH", "SunEC");
20   bKA.init(bKP.getPrivate());
21
22   byte[] bPubKe = bKP.getPublic().getEncoded();
23
24   KeyFactory aKF = KeyFactory.getInstance("EC", "SunEC");
25   x509ks = new X509EncodedKeySpec(bPubKe);
26   PublicKey bPubK = aKF.generatePublic(x509ks);
27   aKA.doPhase(bPubK, true);
28   byte[] aSecret = aKA.generateSecret();
29
30   bKA.doPhase(aPubK, true);
31   byte[] bSecret = bKA.generateSecret();
32
33   if(!Arrays.equals(aSecret, bSecret)) // not this way in real life
34       throw new Exception("Secrets differ");
35 }catch(Exception e){System.err.println("Error: "+e);System.exit(1);}}
```

### 2.4.7. Curvas elípticas inseguras

Em Java, o programador pode usar a criptografia de curvas elípticas de forma implícita, indicando apenas o tamanho das chaves assimétricas, como foi mostrado nos exemplos até aqui. Outra opção disponível ao programador é a escolha explícita de uma curva elíptica, conforme mostrado no trecho de código da Listagem 2.22. Porém, cuidados devem ser tomados para evitar a seleção de curvas elípticas fracas (inseguras).

A Tabela 2.3 mostra curvas elípticas que fizeram parte da primeira versão do padrão SEC 2 [SEC 2000], mas que foram excluídas da segunda versão deste mesmo padrão [SEC 2010], por serem consideradas inseguras, com níveis de segurança já bastante baixos para as necessidades atuais. Todas estas curvas ainda podem ser usadas em Java, porque ainda estão disponíveis no provedor SunEC, conforme lustrado na Listagem 2.22.

**Tabela 2.3. Curvas elípticas inseguras no SunEC (de [Mart and Hern 2013]).**

| Segurança | Curvas sobre $F_p$ | Curvas sobre $F_{2^m}$ |
|---|---|---|
| 56 | secp112r1,secp112r2 | sect113r1, sect113r2 |
| 64 | secp128r1,secp128r2 | sect131r1, sect131r2 |
| 80 | secp160k1,secp160r1,secp160r2 | – |
| 96 | – | sect193r1, sect193r2 |

O trecho de código da Listagem 2.22 mostra como uma curva elíptica pode ser escolhida *a priori* e servir de insumo para a geração do par de chaves ECDH. As linhas de 2 a 4 listam todas as curvas fracas. O programa funciona em um laço (da linha 6 a linha 17) que cria os pares de chaves para cada uma das curvas inseguras. Nas linhas de 7 a 10, um gerador de par de chaves é configurado com uma curva passada como parâmetro na inicialização. Então, o par de chaves é criado como de costume, na linha 10. As linhas de 11 a 15 exibem diversas informações sobre o par de chaves.

**Listagem 2.22. Curvas inseguras do SunEC.**

```
1 public static void main(String argv[]) {
2  String[] curves={"secp112r1","secp112r2","secp128r1",
3       "secp128r2","secp160k1","secp160r1","secp160r2","sect113r1",
4       "sect113r2","sect131r1","sect131r2","sect193r1","sect193r2"};
5  try {
6    for (String curve : curves) {
7      ECGenParameterSpec ecps = new ECGenParameterSpec(curve);
8      KeyPairGenerator kpg=KeyPairGenerator.getInstance("EC","SunEC");
9      kpg.initialize(ecps);
10     KeyPair kp = kpg.generateKeyPair();
11     U.println("EC parameters " + ecps.getName());
12     U.println("Pub  key: " + kp.getPublic());
13     U.println("Priv key: " + U.b2x(kp.getPrivate().getEncoded()));
14     U.println("Algorithm: " + kp.getPrivate().getAlgorithm());
15     U.println("Format: " + kp.getPrivate().getFormat());
16     System.out.println();
17   }
18 }catch(NoSuchAlgorithmException|InvalidAlgorithmParameterException|
19         NoSuchProviderException e){System.err.println("Error:"+e);}}
```

## 2.5. Verificação insegura de certificados digitais

Este seção é organizada em torno das práticas inseguras de programação relacionadas à verificação de certificados digitais e ao uso programático do protocolo SSL/TLS, no lado cliente (isto é, um software cliente SSL/TLS). Os maus usos já foram relacionados aos ataques conhecidos contra o TLS na seção 2.2. Aqui, os maus usos são ilustrados programaticamente por meio de programas em Java. Os problemas de validação de certificados tratados nesta seção são aqueles relacionados à falta de validação dos elementos dos certificados, tais como: data de expiração, cadeia de certificação, nome do emissor e nome do sujeito, além da chave pública. Também são tratadas as armadilhas de programação que inibem a completa validação de certificados digitais e as configurações criptográficas inseguras do cliente SSL/TLS.

### 2.5.1. Certificado recebido e usado sem qualquer validação

Esta seção inicia uma sequência de exemplos de utilização de SSL/TLS em Java, em que as verificações dos elementos de certificado digital são acrescentadas aos poucos. Neste exemplo em particular, nenhuma validação é feita sobre o certificado recebido e a chave pública contida no certificado é utilizada de boa fé (uma abordagem bastante insegura).

O trecho de código da Listagem 2.23 mostra como a validação manual de um certificado pode ser realizada em Java. Este código ilustra o uso de certificação diretamente pela aplicação, por exemplo, para estabelecer um canal SSL/TLS com outros sistemas que não servidores web com HTTPS. As linhas de 1 a 15 contém o método `validate()`, cuja única função é validar certificados digitais contra as diversas informações passadas como parâmetros para o método, tais como a CA, o emissor, o sujeito/servidor e o período de validade. O método tem um defeito grave, a única verificação (linha 5) está comentada e por isto não é realizada. Caso a linha 5 seja descomentada, então o certificado de cliente terá sua assinatura digital verificada contra a chave pública da autoridade certificadora. Um ponto de atenção é que não há qualquer garantia quanto a autenticidade, ou mesmo integridade, da chave pública da CA.

O método `main()` mostrado nas linhas de 16 a 39 da Listagem 2.23 ilustra o uso do método `validate()` e cria uma cadeia de certificação que será utilizada em todos os exemplos deste seção. Esta cadeia de certificação é composta do certificado raiz (da CA de mais alto nível), certificado intermediário (CA de nível médio) e certificado de usuário final.

**Listagem 2.23. Sem qualquer validação dos certificados recebidos.**

```
1 public static boolean validate(X509Certificate cert,X509Certificate ca,
2   X500Principal issuer,X500Principal subj,Date date){
3   boolean ok = false;
4   try {
5     //cert.verify(ca.getPublicKey());
6     ok = true;
7   } catch (CertificateException ex) {
8     ok = false; System.out.println(ex);
9   } catch (NoSuchAlgorithmException | InvalidKeyException |
10            NoSuchProviderException | SignatureException ex){
11    ok = false; System.out.println(ex);
12  }
13  return ok;
14 }
15
```

```
16 public static void main(String[] args) {
17   Security.addProvider(new BouncyCastleProvider());
18   try {
19     KeyPair rkp = CertUtils.genRSAKeyPair();//create keys and root cert
20     X509Certificate root = CertUtils.buildSelfSignedCert(rkp);
21
22     // generate intermediate (middle) certificate
23     KeyPair mkp = CertUtils.genRSAKeyPair();
24     X509Certificate middle = CertUtils.buildMiddleCert(mkp.getPublic(),
25       "CN=Middle CA Cert", rkp.getPrivate(), root);
26
27     // generate end entity certificate
28     KeyPair ekp = CertUtils.genRSAKeyPair();
29     X509Certificate user = CertUtils.buildEndCert(ekp.getPublic(),
30       "CN=User Cert",mkp.getPrivate(), middle);
31
32     X500Principal issuer = new X500Principal("CN=Root Certificate");
33     X500Principal subj1  = new X500Principal("CN=Middle CA Cert");
34     X500Principal subj2  = new X500Principal("CN=User Cert");
35
36     if(validate(user, middle, subj1, subj2, null))
37     System.out.println("User Cert successfully validated");
38   } catch (Exception ex) { System.out.println(ex); }}
```

O trecho de código da Listagem 2.24 mostra como estabelecer um canal HTTPS com um servidor web de URL conhecida e utilizando a porta 443. Este exemplo ilustra o caso de um *browser* proprietário. No exemplo, o cliente HTTPS não faz qualquer verificação do certificado digital recebido do servidor. Após o *handshake* (linha 6), algumas informações sobre a conexão são obtidas (linhas 10 e 11) e exibidas (linhas de 11 a 17). Finalmente, se nada de errado ocorreu durante o estabelecimento da conexão, o conteúdo da página é recebido e tratado (linha 23). Este exemplo também serve de base para as verificações nos exemplos seguintes.

**Listagem 2.24. Sem validação de certificados SSL/TLS.**

```
1 public static void main(String[] args) throws Exception {
2   SSLSocket s = null; Boolean ok = true;
3   try {
4     SSLSocketFactory f=(SSLSocketFactory)SSLSocketFactory.getDefault();
5     s = (SSLSocket) f.createSocket("www.google.com", 443);
6     s.startHandshake();// all validations happen after handshake
7     SSLSession session = s.getSession();
8     Principal peerPrincipal = session.getPeerPrincipal();
9
10    System.out.println("Protocol: " + session.getProtocol());
11    System.out.println("Ciphersuite: " + session.getCipherSuite());
12    System.out.println("Host name: " + session.getPeerHost());
13    System.out.println("\n"+peerPrincipal);
14    System.out.println("\nPeer certificates");
15    Certificate[] peerCertificates = session.getPeerCertificates();
16    for (Certificate c : peerCertificates) { System.out.println(c);}
17  } catch (Exception e) { System.out.println(e); ok = false; }
18
19  if (ok) {CertUtils.handleSocket(s);}
20  else    {System.out.println("Something wrong in cert validation.");}}
```

### 2.5.2. Acrescentando a verificação do período de validade

O trecho de código da Listagem 2.25 mostra como verificar o período de validade de um certificado digital por meio de duas variações do método `checkValidity()` (linhas 5 e 6), com a data do sistema ou com uma data como parâmetro. Além disso, há duas exceções específicas (linha 8) para os casos de certificado expirado e ainda não válido. Se o certificado estiver dentro do período de validade, a assinatura é verificada (linha 13).

**Listagem 2.25. Validação do período de validade.**

```
1 public static boolean validate(X509Certificate cert,X509Certificate ca,
2   X500Principal issuer, X500Principal subj, Date date) {
3   boolean ok = false;
4   try {
5     if (date != null) { cert.checkValidity(date);}
6     else              { cert.checkValidity();    }
7     ok = true;
8   }catch(CertificateExpiredException|CertificateNotYetValidException e)
9   { ok = false; System.out.println(e);}
10
11  if (ok) {
12    try { ok = false; cert.verify(ca.getPublicKey()); ok =  true;}
13    catch (CertificateException e)                    { ok = false;}
14    catch (NoSuchAlgorithmException|InvalidKeyException |
15           NoSuchProviderException |SignatureException e){ok = false;}
16  }
17 return ok;}
```

O trecho de código da Listagem 2.26 mostra como um cliente pode verificar o período de validade de um certificado do servidor em uma conexão HTTPS. A linha 11 obtém a lista de certificados enviada pelo servidor. A linha 12 testa se a lista não é nula e tem pelo menos 2 certificados. As linhas 13 e 14 ativam os métodos `checkValidity()` e `verify()` sobre o primeiro certificado da lista e usando a chave pública do segundo certificado.

**Listagem 2.26. Validação do período de validade do certificado SSL/TLS.**

```
1 public static void main(String[] args) throws Exception {
2   SSLSocket s = null; boolean ok = true;
3   try {
4     SSLSocketFactory f=(SSLSocketFactory)SSLSocketFactory.getDefault();
5     s = (SSLSocket) f.createSocket("www.google.com", 443);
6     s.startHandshake();//all validations happen after handshake
7
8     SSLSession session = s.getSession();
9     Principal peerPrincipal = session.getPeerPrincipal();
10
11    Certificate[] peerCertificates = session.getPeerCertificates();
12    if (peerCertificates != null && peerCertificates.length >= 2) {
13      ((X509Certificate) peerCertificates[0]).checkValidity();
14      peerCertificates[0].verify(peerCertificates[1].getPublicKey());
15    }
16  }catch(CertificateExpiredException|CertificateNotYetValidException|
17      NoSuchAlgorithmException|InvalidKeyException|
18      NoSuchProviderException |SignatureException e) {ok = false;}}
```

### 2.5.3. Acrescentando a validação da cadeia de certificação

O trecho de código da Listagem 2.27 mostra como o método `validate()`, cuja implementação já foi mostrada anteriormente nesta seção, pode ser utilizado para validar manualmente não apenas os certificados na ponta da cadeia de certificação, como também, os certificados de CAs intermediárias. Nas linhas de 4 a 18, uma cadeia de certificação é criada. Na linha 20, o método `validate()` é usado para verificar o certificado da CA intermediária. Na linha 22, o método `validate()` é usado para verificar o certificado de usuário final.

**Listagem 2.27. Validação manual dos certificados intermediários.**

```
1 public static void main(String[] args) {
2   Security.addProvider(new BouncyCastleProvider());
3   try {
4     KeyPair rkp = CertUtils.genRSAKeyPair();
5     X509Certificate root = CertUtils.buildSelfSignedCert(rkp);
6
7     KeyPair mkp = CertUtils.genRSAKeyPair();
8     X509Certificate middle = CertUtils.buildMiddleCert(mkp.getPublic(),
9             "CN=Middle CA Certificate", rkp.getPrivate(), root);
10
11    // generate end entity certificate
12    KeyPair ekp = CertUtils.genRSAKeyPair();
13    X509Certificate user = CertUtils.buildEndCert( ekp.getPublic(),
14            "CN=User Certificate", mkp.getPrivate(), middle);
15
16    X500Principal issuer=new X500Principal("CN=Root Certificate");
17    X500Principal subj1=new X500Principal("CN=Middle CA Certificate");
18    X500Principal subj2=new X500Principal("CN=User Certificate");
19
20    if (validate(middle,root,issuer,subj1,null))
21      System.out.println("Middle certificate successfully validated");
22    if (validate(user,middle,subj1,subj2,null))
23      System.out.println("User Certificate successfully validated");
24  } catch (Exception ex) {System.out.println(ex);}
25 }
```

O trecho de código da Listagem 2.28 mostra a validação da cadeia de certificação para uma conexão SSL/TLS. As linhas de 4 a 8 estabelecem o canal seguro e obtém as informações da sessão. O restante do programa faz a verificação da cadeia de certificação. O processo possui dois passos preparatórios e um passo de verificação: (*i*) Obtenção do certificado raiz e dos outros certificados da cadeia de certificação, de modo que a cadeia possa ser completamente construída (linhas de 10 a 20); (*ii*) Designação dos certificados raízes como âncoras de confiança da cadeia (linhas 22 e 25); (*iii*) Validação da cadeia de certificação do certificado do servidor (linhas 27 até 36).

Na linha 28, é criado um validador de cadeia de certificação (instância de `CertPathValidator`). Finalmente, as linhas de 30 a 35 realizam a verificação. O método `validate()` da linha 32 verifica a cadeia passada como parâmetro. Se exceções não ocorrerem (verificação bem sucedida), as políticas de uso do certificado (linha 33) e a chave pública (linha 34) podem ser obtidas com confiança. No caso do SSL/TLS, o conteúdo da URL é recuperado (linha 41). Senão, alguma exceção é disparada (linhas 38 e 39).

Vale observar que o exemplo da Listagem 2.28 não verifica se o certificado do servidor, ou algum certificado da cadeia de certificação, foi revogado.

**Listagem 2.28. Validação da cadeia de certificação no SSL/TLS.**

```
1 public static void main(String[] args) throws Exception {
2  SSLSocket s = null; boolean ok = true;
3  try {
4    SSLSocketFactory f=(SSLSocketFactory)SSLSocketFactory.getDefault();
5    s = (SSLSocket) f.createSocket("www.google.com", 443);
6    s.startHandshake();//all validations happen after handshake
7    SSLSession session = s.getSession();
8    Principal peerPrincipal = session.getPeerPrincipal();
9
10   // Step 1. Obtain root certs and cert path to validate
11   Certificate[] certs = session.getPeerCertificates();
12   X509Certificate[] x509certs = new X509Certificate[certs.length-1];
13   for (int i = 0; i < certs.length-1; i++) {
14      x509certs[i] = (X509Certificate) certs[i];
15   }
16   X509Certificate anchor = (X509Certificate) certs[certs.length-1];
17
18   List l = Arrays.asList(x509certs);
19   CertificateFactory cf=CertificateFactory.getInstance("X.509","SUN");
20   CertPath cp = cf.generateCertPath(l);
21
22   // Step 2. Create a PKIXParameters with the trust anchors
23   TrustAnchor ta = new TrustAnchor(anchor, null);
24   PKIXParameters params=new PKIXParameters(Collections.singleton(ta));
25   params.setRevocationEnabled(false);//Revocation status ignored!
26
27   // Step 3. Use a CertPathValidator to validate the certificate path
28   CertPathValidator cpv = CertPathValidator.getInstance("PKIX","SUN");
29
30   // validate certification path with specified params
31   PKIXCertPathValidatorResult cpvr
32        = (PKIXCertPathValidatorResult) cpv.validate(cp, params);
33   PolicyNode policyTree = cpvr.getPolicyTree();
34   PublicKey subjectPK = cpvr.getPublicKey();
35   System.out.println("Certificate Chain successfully validated");
36   System.out.println(subjectPK);
37
38 }catch(CertificateException|InvalidAlgorithmParameterException|
39  NoSuchAlgorithmException|CertPathValidatorException e){ok = false;}
40
41 if(ok){CertUtils.handleSocket(s);}
42 else  {System.out.println("Something wrong in cert validation.");}}
```

### 2.5.4. Acrescentando a validação dos nomes do sujeito e do emissor

No trecho de código da Listagem 2.29, não apenas o período de validade (linhas 5 e 6) e a assinatura do certificado do cliente (linha 12) são verificados, como também os nomes do sujeito (contido no certificado) e do emissor são validado contra os nomes associados à aplicação. Este exemplo supõe que as verificações da cadeia de certificação e da revogação dos certificados já foram realizadas do lado de fora do método.

Na linha 21, o nome do emissor do certificado é comparado ao nome esperado pela aplicação. Enquanto na linha 24, o nome do sujeito do certificado é comparado ao nome esperado pela aplicação. Vale observar que estes nomes são recuperados de um certificado no formato X.509 e, por isto, seguem este padrão de formatação.

**Listagem 2.29. Validação do nome do Sujeito ou Host.**

```
1 public static boolean validate(X509Certificate cert,X509Certificate ca,
2   X500Principal issuer, X500Principal subj, Date date) {
3   boolean ok = false;
4   try {
5     if (date != null) { cert.checkValidity(date);}
6     else              { cert.checkValidity();    }
7     ok = true;
8   }catch(CertificateExpiredException|CertificateNotYetValidException e)
9   { ok = false; System.out.println(e); }
10
11  if (ok) { //DANGER: PubKey may not have been validated !!!!
12    try { ok = false; cert.verify(ca.getPublicKey());ok = true; }
13    catch (CertificateException ex) {
14      ok = false; System.out.println(ex);
15    } catch (NoSuchAlgorithmException | InvalidKeyException |
16             NoSuchProviderException | SignatureException ex)
17    { ok = false; System.out.println(ex); }
18  }
19  if (ok) {
20    ok = false;
21    if (cert.getIssuerX500Principal().equals(issuer)){ ok =  true;}
22    else { System.out.println("Issuer name mismatch"); ok = false;}
23
24    if (ok && cert.getSubjectX500Principal().equals(subj)){ok = true;}
25    else { System.out.println("Subject name mismatch"); ok = false;}
26  }
27  return ok;}
```

### 2.5.5. Validação da lista de certificados revogados

O último exemplo desta seção mostra as verificações da cadeia de certificação e da revogação dos certificados. O trecho de código da Listagem 2.30 faz a verificação da cadeia de certificação assim como do *status* de revogação do certificado do servidor por meio de uma CRL. O processo possui dois passos preparatórios e um passo de verificação: (*i*) Obtenção do certificado raiz e dos outros certificados da cadeia de certificação, de modo que a cadeia possa ser completamente construída (linhas de 4 a 8); (*ii*) Designação dos certificados raízes como âncoras de confiança da cadeia (linhas 10 e 12); (*iii*) Validação da cadeia de certificação e do *status* de revogação do certificado do servidor (linhas 14 até 43). Somente o terceiro passo é detalhado.

Nas linhas de 15 a 22, é criado um validador de cadeia de certificação (instância de `CertPathValidator`) com quatro opções configuradas: (*a*) falha suavemente se não obtiver a CRL e nem a verificação por OCSP, (*b*) sem mecanismo de recuperação, (*c*) verifica apenas os certificados na ponta da cadeia, e (*d*) prefere a verificação por CRL em vez de OCSP.

Nas linhas de 25 a 30, uma CRL associada ao certificado raiz (ancora de confiança) é recuperada de modo confiável (com o método `revokeCerts()`) e adicionada aos parâmetros da verificação. Finalmente, as linhas de 32 a 38 realizam a verificação. O método `validate()` da linha 34 verifica a cadeia e a CRL passadas como parâmetros. As exceções, incluindo a exceção de validação de cadeia de certificação, são capturadas nas linhas 35 e 36.

**Listagem 2.30. Validação da lista de certificados revogados.**

```
1 public static void main(String[] args) {
2  Security.addProvider(new BouncyCastleProvider());
3  try {
4    // Step 1. Obtain root certs and cert path to validate
5    X509Certificate[] certs = getCertificateList();
6    List l = Arrays.asList(certs);
7    CertificateFactory cf=CertificateFactory.getInstance("X.509","SUN");
8    CertPath cp = cf.generateCertPath(l);
9
10   // Step 2. Create a PKIXParameters with the trust anchors
11   TrustAnchor ta = new TrustAnchor(rootCA,null);
12   PKIXParameters params=new PKIXParameters(Collections.singleton(ta));
13
14   // Step 3. Use a CertPathValidator to validate the certificate path
15   CertPathValidator cpv = CertPathValidator.getInstance("PKIX","SUN");
16   PKIXRevocationChecker rc =
17         (PKIXRevocationChecker)cpv.getRevocationChecker();
18   rc.setOptions(EnumSet.of(Option.SOFT_FAIL));
19   rc.setOptions(EnumSet.of(Option.NO_FALLBACK));
20   rc.setOptions(EnumSet.of(Option.ONLY_END_ENTITY));
21   rc.setOptions(EnumSet.of(Option.PREFER_CRLS));
22   params.addCertPathChecker(rc);
23
24   // now it revokes the very same list of certificates
25   X509CRL crl=CertUtils.revokeCerts(rootCA,rootKP.getPrivate(),certs);
26   List list = new ArrayList();list.add(crl);
27   CertStoreParameters csp = new CollectionCertStoreParameters(list);
28   CertStore store = CertStore.getInstance("Collection", csp);
29   params.addCertStore(store);
30   //params.setRevocationEnabled(false);//remove comment to desable CRL
31
32   // validate certification path with specified params
33   PKIXCertPathValidatorResult cpvr =
34         (PKIXCertPathValidatorResult) cpv.validate(cp, params);
35 }catch(InvalidAlgorithmParameterException|CertPathValidatorException|
36       CertificateException                |NoSuchAlgorithmException e)
37                   { System.out.println(e);
38 }catch(Exception e){ System.out.println(e);}}
```

## 2.6. Considerações finais

Três anos após a publicação do primeiro texto voltado à criptografia simétrica, este capítulo dá outro passo na direção da construção de um arcabouço que proporcione ao programador não especialista em criptografia a confiança necessária para a codificação segura de software criptográfico. Alguns dos assuntos deixados para trás naquela ocasião foram recuperados e (esperamos) abordados de modo satisfatório para o leitor programador. Em linhas gerais, este capítulo abordou a criptografia assimétrica em Java com a API padrão da plataforma, apontando diversas maneiras de usar mal a API, causando vulnerabilidades, assim como também mostrando maneiras de usar bem a criptografia de chave pública. Em particular, foram cobertos os maus usos criptográficos associados às configurações inseguras de algoritmos assimétricos para acordo de chaves, encriptação e assinaturas digitais, à utilização programática de curvas elípticas inseguras e à validação incompleta de certificados digitais.

O desenvolvimento de software criptográfico é cheio de decisões de projeto e armadilhas de programação que confundem o programador comum, não acostumado à complexidade da tecnologia criptográfica. Neste texto, tentamos evitar a postura combativa contra os programadores, favorecendo o ponto de vista do desenvolvedor de software no tratamento dos maus usos criptográficos. Deste modo, esperamos contribuir para o fortalecimento da **segurança de software criptográfico** [Braga 2017] como um novo campo de estudo preocupado com o desenvolvimento sistemático de software criptográfico seguro. Esta nova área de pesquisa adota uma bordagem preventiva, buscando não apenas se antecipar aos maus usos criptográficos, mas também compreender as necessidades e dificuldades dos desenvolvedores de software.

A observação sistemática do mundo real, por meio de experimentos e análises empíricas, trouxe, com o tempo, a perspectiva necessária para perceber que há uma grande lacuna entre o que os criptologistas veem como maus usos de criptografia e aquilo que os desenvolvedores percebem como uso inseguro da tecnologia criptográfica. Este texto contribui para preencher essa lacuna, oferecendo para a comunidade de desenvolvedores de software insumos práticos que viabilizam tanto a aplicação no curto prazo, quanto os estudos futuros.

Como era de se esperar, o tema é fértil e o assunto é inesgotável. Por isto, há tópicos que gostaríamos de ter tratado, mas que por falta de espaço, foram deixados para outra oportunidade. Em particular, não foram explorados os maus usos criptográficos em outras linguagens de programação, como por exemplo, JavaScript. Além disso, a criptografia pós-quântica não fez parte deste texto porque ainda não há evidência empírica de como esta tecnologia pode ser mal utilizada na prática em atividades rotineiras de programação de software. Porém, a padronização e popularização da criptografia pós-quântica podem trazer oportunidades para uma possível continuação deste texto, no futuro.

## Agradecimentos

Este trabalho foi realizado no Laboratório de Segurança e Criptografia (LASCA) do Instituto de Computação, Universidade Estadual de Campinas. Os autores agradecem o apoio financeiro do projeto ATMOSPHERE (**A**daptive, **T**rustworthy, **M**anageable, **O**rchestrated, **S**ecure, **P**rivacy-assuring, **H**ybrid, **E**cosystem for **RE**silient Cloud Computing), RNP e MCTIC, no âmbito to acordo de cooperação Número 51119. *ATMOSPHERE is funded by the European Commission under the Cooperation Programme, Horizon 2020 grant agreement No 777154*.

## Referências


[CRI 2012] (2012). The CRIME attack: netifera.com. *URL: http://netifera.com*.

[BRE 2013] (2013). The BREAH attack: SSL GONE IN 30 SECONDS. *URL: http://breachattack.com*.

[Log 2016] (2016). The logjam attack and weak diffie-hellman. *URL: https://weakdh.org*.

[Acar et al. 2017] Acar, Y., Backes, M., Fahl, S., Garfinkel, S., Kim, D., Mazurek, M. L., and Stransky, C. (2017). Comparing the usability of cryptographic apis. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*.

[Adrian et al. 2015] Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J. A., Heninger, N., Springall, D., Thomé, E., Valenta, L., and Others (2015). Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17. ACM.

[Akhawe et al. 2013] Akhawe, D., Amann, B., Vallentin, M., and Sommer, R. (2013). Here's My Cert, So Trust Me, Maybe?: Understanding TLS Errors on the Web. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 59–70. International World Wide Web Conferences Steering Committee.

[Alashwali 2013] Alashwali, E. S. (2013). Cryptographic vulnerabilities in real-life web servers. In *Third International Conference on Communications and Information Technology (ICCIT)*, pages 6–11. Ieee.

[AlFardan et al. 2013] AlFardan, N. J., Bernstein, D. J., Paterson, K. G., Poettering, B., and Schuldt, J. C. N. (2013). On the security of rc4 in tls. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 305–320, Berkeley, CA, USA. USENIX Association.

[Anderson 1993] Anderson, R. (1993). Why cryptosystems fail. *Proceedings of the 1st ACM Conference on Computer . . .* , pages 215–227.

[Aviram et al. 2016] Aviram, N., Schinzel, S., Somorovsky, J., Heninger, N., Dankel, M., Steube, J., Valenta, L., Adrian, D., Halderman, J. A., Dukhovni, V., Käsper, E., Cohney, S., Engels, S., Paar, C., and Shavitt, Y. (2016). DROWN: Breaking TLS using SSLv2. *Proceedings of the 25th USENIX Security Symposium*, (August):1–18.

[Bernstein 2006] Bernstein, D. J. (2006). Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer.

[Bernstein et al. 2013] Bernstein, D. J., Lange, T., et al. (2013). Safecurves: choosing safe curves for elliptic-curve cryptography. *URL: http://safecurves.cr.yp.to*.

[Bleichenbacher 1998] Bleichenbacher, D. (1998). Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In *Annual International Cryptology Conference*, pages 1–12. Springer.

[Bleichenbacher et al. 2017] Bleichenbacher, D., Duong, T., Kasper, E., and Nguyen, Q. (2017). Project Wycheproof - Scaling crypto testing. In *Real World Crypto Symposium*, New York, USA.

[BlueKrypt ] BlueKrypt. Cryptographic Key Length Recommendation. *URL: http://www.keylength.com*.

[Boneh 1999] Boneh, D. (1999). Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS*, pages 1–16.

[Bos et al. 2014] Bos, J. W., Halderman, J. A., Heninger, N., Moore, J., Naehrig, M., and Wustrow, E. (2014). Elliptic curve cryptography in practice. In *Financial Cryptography and Data Security*, pages 157–175. Springer.

[BouncyCastle 2018] BouncyCastle (2018). The Legion of the Bouncy Castle. *URL: http://www.bouncycastle.org/*.

[Braga and Dahab 2015a] Braga, A. and Dahab, R. (2015a). A Survey on Tools and Techniques for the Programming and Verification of Secure Cryptographic Software. In *XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais — SBSeg 2015*, pages 30–43, Florianópolis, SC, Brazil.

[Braga and Dahab 2015b] Braga, A. and Dahab, R. (2015b). Introdução à Criptografia para Programadores: Evitando Maus Usos da Criptografia em Sistemas de Software. In *Caderno de minicursos do XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais — SBSeg 2015*, pages 1–50. Sociedade Brasileira de Computação.

[Braga and Dahab 2016] Braga, A. and Dahab, R. (2016). Mining Cryptography Misuse in Online Forums. In *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 143–150.

[Braga and Dahab 2017] Braga, A. and Dahab, R. (2017). A Longitudinal and Retrospective Study on How Developers Misuse Cryptography in Online Communities. In *XVII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg'17)*, Brasília, DF, Brazil.

[Braga et al. 2017a] Braga, A., Dahab, R., Antunes, N., Laranjeiro, N., and Vieira, M. (2017a). Practical Evaluation of Static Code Analysis Tools for Cryptography: Benchmarking Method and Case Study. In *The 28th IEEE International Symposium on Software Reliability Engineering (ISSRE)*.

[Braga et al. 2017b] Braga, A., Marino, F., and Santos, R. (2017b). Segurança de Aplicações Blockchain Além das Criptomoedas. In SBC, editor, *Livro de minicursos do XVII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg'17)*, chapter 3.

[Braga 2017] Braga, A. M. (2017). Towards the safe development of cryptographic software (Rumo ao desenvolvimento seguro de software criptográfico).

[Calvo 2018] Calvo, R. (2018). The true cost of certificate authority trials: Can you trust them? *URL: https://www.isc2.org/News-and-Events/Infosecurity-Professional-Insights*.

[Chandra et al. 2002] Chandra, P., Messier, M., and Viega, J. (2002). Network security with OpenSSL. *O'Reily, June*.

[Diffie and Hellman 1976] Diffie, W. and Hellman, M. (1976). New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654.

[Egele et al. 2013] Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. (2013). An empirical study of cryptographic misuse in android applications. *ACM SIGSAC conference on Computer & communications security - CCS '13*, pages 73–84.

[Eldewahi et al. 2015] Eldewahi, A. E. W., Sharfi, T. M. H., Mansor, A. A., Mohamed, N. A. F., and Alwahbani, S. M. H. (2015). Ssl/tls attacks: Analysis and evaluation. In *2015 International Conference on Computing, Control, Networking, Electronics and Embedded Systems Engineering (ICCNEEE)*, pages 203–208.

[Fahl et al. 2012] Fahl, S., Harbach, M., and Muders, T. (2012). Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *ACM conference on Computer and communications security*, pages 50–61.

[Fardan and Paterson 2013] Fardan, N. A. and Paterson, K. (2013). Lucky thirteen: Breaking the TLS and DTLS record protocols. *Security and Privacy (SP), 2013 IEEE Symposium on (2013)*.

[Georgiev et al. 2012] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and Shmatikov, V. (2012). The most dangerous code in the world. In *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12*, page 38. ACM Press.

[Gluck et al. 2013] Gluck, Y., Harris, N., and Angel, A. (2013). BREACH: Reviving the CRIME Attack. In *Black Hat Conference*.

[Google ] Google. Google Android Developers. *URL: https://groups.google.com/forum/#!forum/android-developers*.

[Gutmann a] Gutmann, P. Everything you Never Wanted to Know about PKI but were Forced to Find Out. *URL: https://www.cs.auckland.ac.nz/ pgut001/pubs/pkitutorial.pdf*.

[Gutmann b] Gutmann, P. Godzilla crypto tutorial - Part 2, Key Management and Certificates.

[Gutmann 2002] Gutmann, P. (2002). Lessons Learned in Implementing and Deploying Crypto Software. *Usenix Security Symposium*.

[Hankerson et al. 2004] Hankerson, D., Vanstone, S., and Menezes, A. (2004). *Guide to elliptic curve cryptography*.

[IACR 2012] IACR (2012). Real world crypto symposium. *URL: https://rwc.iacr.org/index.html*.

[Jonsson and Burt Kaliski 2003] Jonsson, J. and Burt Kaliski (2003). RSA Laboratories Public-Key Cryptography Standards (PKCS)#1: RSA Cryptography Specifications Version 2.1. *URL: https://tools.ietf.org/html/rfc3447*.

[Koblitz 1987] Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209.

[Marlinspike 2009] Marlinspike, M. (2009). New tricks for defeating SSL in practice. *URL: https://blackhat.com/presentations/bh-europe-09/Marlinspike/blackhat-europe-2009-marlinspike-sslstrip-slides.pdf*.

[Mart and Hern 2013] Mart, V. G. and Hern, L. (2013). Implementing ECC with Java Standard Edition 7. *International Journal of Computer Science and Artificial Intelligence*, 3(4):134–142.

[Menezes et al. 1996] Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A. (1996). *Handbook of applied cryptography*. CRC press.

[Miller 1985] Miller, V. S. (1985). Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques*, pages 417–426. Springer.

[Möller et al. 2014] Möller, B., Duong, T., and Kotowicz, K. (2014). The POODLE attack. *URL: https://www.openssl.org/ bodo/ssl-poodle.pdf*.

[NIST 2012] NIST (2012). Recommendation for Key Management – Part 1: General (Revision 3). *URL: http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf*.

[NIST 2013] NIST (2013). Digital Signature Standard (DSS).

[OpenSSL.org ] OpenSSL.org. OpenSSL Cryptography and SSL/TLS toolkit. *URL: OpenSSL.org*.

[Oracle a] Oracle. Java Cryptography Architecture (JCA) Reference Guide. *URL: docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html*.

[Oracle b] Oracle. Java Cryptography Architecture Oracle Providers Documentation for Java Platform Standard Edition 8. *URL: docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html*.

[OWASP 2015] OWASP (2015). OWASP Testing Project v4. *URL: https://www.owasp.org/index.php/OWASP_Testing_Project*.

[Rescorla 2018] Rescorla, E. (2018). The transport layer security (tls) protocol version 1.3. *URL: https://tools.ietf.org/html/rfc8446*.

[Ristic 2015] Ristic, I. (2015). *OpenSSL cookbook*. Feisty Duck, 2nd. ed. (version 2.1-draft published in june 2018) edition.

[Rivest et al. 1978] Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

[Schneier 1998] Schneier, B. (1998). Cryptographic design vulnerabilities. *Computer*, (September):29–33.

[SEC 2000] SEC, S. (2000). Sec 2: Recommended elliptic curve domain parameters, version 1. *Standards for Efficient Cryptography Group, Certicom Corp (http://www.secg.org/)*.

[SEC 2010] SEC, S. (2010). Sec 2: Recommended elliptic curve domain parameters, version 2. *Standards for Efficient Cryptography Group, Certicom Corp ()http://www.secg.org/*.

[Shuai et al. 2014] Shuai, S., Guowei, D., Tao, G., Tianchang, Y., and Chenjie, S. (2014). Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications. In *IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC)*, pages 75–80.

[Stallings 2003] Stallings, W. (2003). *Cryptography and network security, principles and practices*.

[Sullivan 2018] Sullivan, N. (2018). A detailed look at rfc 8446 (a.k.a. tls 1.3). *URL: https://blog.cloudflare.com/rfc-8446-aka-tls-1-3*.

[Wikipedia 2018] Wikipedia (2018). Netscape. *URL: https://en.wikipedia.org/wiki/Netscape*.