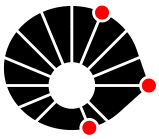

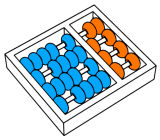

Estudo e Implementação do Ataque de Canal Lateral no ECDSA do OpenSSL

F. R. Novaes *D. F. Aranha* *Y. Yarom*

Relatório Técnico - IC-PFG-17-24
Projeto Final de Graduação
2017 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Estudo e Implementação do Ataque de Canal Lateral no ECDSA do OpenSSL

Felipe R. Novaes* Diego F. Aranha† Yuval Yarom‡

Resumo

O algoritmo de assinatura digital de curvas elípticas ECDSA (*Elliptic Curve Digital Signature Algorithm*) possui uma implementação eficiente e segura no software *OpenSSL*. Entretanto, esse trabalho mostra que essa implementação para curvas sobre $\mathbb{F}_{2^m}$ na versão corrente é suscetível a ataques de canal lateral de temporização para conseguir recuperar o bit mais significativo do escalar *nonce* durante a fase de multiplicação.

Para demonstrar o vazamento, esse trabalho implementa um ataque usando a técnica de *FLUSH+RELOAD* conjuntamente com degradação de desempenho em assinaturas feitas com a chave privada gerada pela curva de koblitz sect163r1. O teste em questão gerou 10.000 assinaturas aplicando o ataque para recuperar o bit mais significativo, demorando em torno de 8 horas. O resultado obtido foi uma precisão de $97,3\%$ e $89,6\%$ de revocação para casos com o bit 1 e $90,3\%$ e $90,3\%$ para o caso do bit 0.

Trabalhos futuros podem usar esse bit para quebrar o ECDSA encontrando a chave privada através do ataque modificado de Bleichenbacher. Além disso, trabalho futuros podem investigar métodos de acelerar o ataque de recuperar o bit diminuindo o efeito da degradação de desempenho, retirando-a do restante da assinatura, o que tornaria o ataque mais prático e eficiente.

1 Introdução

No mundo moderno, comunicações digitais tornam-se cada vez mais presentes, as quais necessitam de uma implementação segura, provendo confidencialidade, integridade, disponibilidade e autenticidade. Para este fim, a biblioteca de software *OpenSSL* se apresenta através de uma implementação segura e eficiente de código aberto dos algoritmos criptográficos que provém aquelas necessidades, como também os protocolos *Transport Layer Security* (TLS) e seu predecessor, *Secure Sockets Layer* (SSL).

O algoritmo de assinatura digital é uma das funções implementadas pelo *OpenSSL*. Esse algoritmo pode garantir autenticidade aos dados, replicando para o mundo virtual a ideia de assinatura escrita a mão. Umas das implementações desse algoritmo, disponível

*Graduando, Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP
†Professor Associado, Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP
‡Pesquisador Associado, School of Computer Science, Univeristy of Adelaide, Adelaide, SA

no software, é a versão sobre a criptografia de curvas elípticas, chamada de *Elliptic Curve Digital Signature Algorithm* (ECDSA).

O sistema de criptografia de curvas elípticas foi proposto inicialmente por Koblitz [Koblitz 1987] e Miller [Miller 1986] e surge como uma versão mais eficiente da criptografia assimétrica do RSA. Ela é baseada na intratabilidade de calcular eficientemente o valor de um escalar k que originou o ponto $Q = kP$, dados Q e P. A operação sobre os pontos depende dos corpos finitos sobre qual a curva está definida, normalmente, esses corpos são binários $\mathbb{F}_{2^m}$ ou primos $\mathbb{F}_p$. Dependendo do corpo finito empregado, as operações subjacentes para realizar os algoritmos criptográficos irão ter diferentes características.

O projeto descrito nesse documento é uma análise da implementação do algoritmo ECDSA sobre curvas binárias na versão corrente do *OpenSSL*, buscando encontrar vazamento do bit mais significativo (MSB) do escalar chamado de *nonce* (gerado aleatoriamente para cada assinatura) durante a multiplicação por escalar. Esse ataque terá uma continuação que consistirá em implementar o método descrito no trabalho de [Aranha et al. 2014] para conseguir recuperar a chave privada usada para assinar o ECDSA, apenas com um bit do *nonce*, quebrando dessa forma o algoritmo criptográfico.

Este documento está organizado da seguinte maneira: a Seção 2 apresenta os conceitos essenciais para o ataque implementado nesse trabalho, incluindo a teoria necessária de criptografia de curvas elípticas (2.1), os algoritmos de multiplicação por escalar (2.1.3) que são o alvo desse ataque e o algoritmo de assinatura digital sobre curvas elípticas (2.2). A Seção 3 faz uma breve revisão dos ataques de canal lateral implementados em criptografia de curvas elípticas, enquanto que a Seção 4 descreve o ataque de canal lateral implementado nesse trabalho. O resultados dos experimento são abordados na Seção 5. Por fim, na Seção 6 são discutidas as contramedidas e conclusões sobre esse trabalho.

2 Preliminares

A criptografia tem como objetivo garantir uma comunicação segura na presença de indivíduo malicioso. De modo geral, essa segurança pode ser garantida através de 4 pilares:

- Confidencialidade: garantir que somente o indivíduo autorizado tem capacidade de ler o dado no canal de comunicação;
- Integridade de Dados: garantir que o dado não foi alterado;
- Autenticação: garantir que a identidade do remetente seja legítima;
- Não-repudiação: garantir que um remetente não possa negar a autoria do dado;

Essas garantias podem ser satisfeitas usando duas abordagens da criptografia. A primeira é chamada de *criptografia simétrica*, a qual ambas as partes devem possuir um segredo em comum conhecido apenas por elas. Dessa forma, esse segredo é usado em algoritmos para cifrar/decifrar a informação trocada. Apesar desse método possuir bom desempenho computacional, ele apresenta dificuldades na distribuição e gerenciamento do segredo simétrico (precisa-se de um canal seguro).

Para solucionar o problema da primeira abordagem, foi proposta uma segunda abordagem chamada de *criptografia assimétrica* [Diffie and Hellman 1976]. Nesse modelo cada indivíduo possui um par de chaves, sendo uma chave pública (qualquer indivíduo tem acesso) e uma chave privada (somente conhecida pelo seu proprietário). Uma mensagem assinada com a chave privada pode ser somente verificada usando o par público dela, assim como, uma mensagem cifrada com o par público pode somente ser decifrada com o par privado.

De maneira prática, se nesse modelo assimétrico Alice, com chave pública A_{pub} e privada A_{priv}, deseja alcançar confidencialidade e autenticidade ao enviar uma mensagem M para Bob, com chave pública B_{pub} e par privado B_{priv}. Alice deve, primeiramente, assinar a mensagem com o par privado dela: $C_{alice} = A_{priv}(M)$. Posteriormente, aplica-se a chave pública de Bob: $C_{bob} = B_{pub}(C_{Alice})$. Então, envia-se C_{bob}, essa cifra pode somente ser recuperada por Bob, pois somente ele possui o par privado da chave B_{pub} (garante confidencialidade): $P_{bob} = B_{priv}(C_{bob})$. Mas P_{bob} ainda precisa ser verificada através da chave pública de Alice: $M = A_{pub}(P_{bob})$ (garantindo que a mensagem foi enviada por B). Note que nesse exemplo a troca não exige um canal seguro, pois as chaves usadas para cifrar/decifrar entre as partes são públicas. Entretanto, ainda precisa garantir a distribuição da chave pública, o que é resolvido por uma terceira parte confiável.

A propriedade essencial da não necessidade de um canal seguro entre as partes é alcançada através das funções matemáticas de alçapão. Para essas funções, encontrar a função inversa é um problema intratável (não possui algoritmo eficiente conhecido para esse fim). Logo, mesmo que o atacante tenha acesso a $f(x_1)$ e a própria função $f(x)$, não se sabe como encontrar x_1 em tempo polinomial. Entretanto, essas funções possuem a propriedade que dada uma informação adicional chamada alçapão ("trapdoor"), computar x_1 é possível em tempo polinomial. Dessa forma, os parâmetros que definem $f(x)$ podem ser públicos, enquanto que a informação de alçapão é privada.

O problema de fatoração de número inteiros é usado para construir sistemas criptográficos assimétricos através do algoritmo de RSA. Todavia, são conhecidos algoritmos sub-exponenciais para solução desse problema, o que obriga a utilizar chaves maiores para conseguir níveis de segurança satisfatórios. A criptografia de curvas elípticas soluciona essa questão, pois é construída em cima de um problema matemático considerado mais difícil (logaritmo discreto em curvas elípticas), podendo usar chaves mais curtas [López and Dahab 1999]. Essa propriedade das curvas elípticas é muito interessante para ambientes com baixo poder computacional, como explicitado no trabalho de criptografia em redes de sensores sem fio [Aranha et al. 2010].

2.1 Criptografia de Curvas Elípticas

O entendimento da criptografia de curvas elípticas depende de compreender dois aspectos matemáticos: corpos finitos e teoria de curvas elípticas. A definição do primeiro é introduzida na Seção 2.1.1, enquanto que o segundo na Seção 2.1.2. Por fim, na Seção 2.1.3 são descritos alguns dos algoritmos de multiplicação por escalar, operação essencial para criptografia de curvas elípticas e objetivo de ataque desse trabalho. A teoria descrita nessas seções são uma revisão do material exposto no livro [Hankerson et al. 2004].

2.1.1 Corpos Finitos

Define-se grupo albeliano $(G, *)$ como um conjunto G com a operação binária $*$: $G \times G \rightarrow G$ satisfazendo as propriedades de associatividade, existência de elemento de identidade, existência de inverso e comutatividade. A ordem de n de G é o número de elementos existentes no conjunto (caso seja finito), nesse caso, para um grupo multiplicativo $(G, \cdot)$ o ponto $g \in G$ é gerador de G quando a ordem do conjunto $\langle g \rangle = \{g^i : 0 \leq i \leq (t-1)\}$ é igual a ordem G, onde t é o menor inteiro tal que $g^t = 1$ (o elemento identidade desse grupo). Analogamente, para o grupo aditivo $(G, +)$ é verdade quando o conjunto $\langle g \rangle = \{ig : 0 \leq i \leq (t-1)\}$ é da mesma ordem e $gt = 0$ (o elemento identidade desse grupo), onde t é o menor divisor de n. Em ambos os casos, dizemos que o grupo G e o subgrupo $\langle g \rangle$ são cíclicos.

Os corpos são abstrações da família de sistema de números tais como $\mathbb{R}$ e $\mathbb{C}$. Eles são conjuntos denotados por $\mathbb{F}$ (quando conjunto é finito é chamado de corpo finito) possuindo duas operações: adição $(+)$ e multiplicação $(\cdot)$, satisfazendo:

1. $(\mathbb{F}, +)$ é um grupo albeliano de adição com identidade 0 e inverso de a sendo $-a$
2. $(\mathbb{F}\backslash\{0\}, \cdot)$ é um grupo albeliano de multiplicação com identidade 1 e inverso de a sendo a^{-1}
3. Distributiva: $(a + b) \cdot c = a \cdot c + b \cdot c$ para todos $a, b, c \in \mathbb{F}$

Um corpo finito existe se, e somente se, a sua ordem q (número de elementos) é uma potência de um número primo: $q = p^m$, onde p é um número primo (chamado de característica do corpo finito). Assim, denotamos um corpo finito da seguinte maneira $\mathbb{F}_{p^m}$. De forma prática, duas classes de corpos finitos são largamente usados: corpos primos $(\mathbb{F}_p)$ e corpos binários $(\mathbb{F}_{2^m})$.

Os corpos primos $\mathbb{F}_p$ possuem os elementos $\{0, 1, 2, .., (p-1)\}$ e todas operações aritméticas são realizadas módulo p.

Entretanto, para esse trabalho em específico o foco é o corpo finito binário $\mathbb{F}_{2^m}$. Esse corpo finito pode ser representado pelo polinômio de grau $m - 1$:

$$\mathbb{F}_{2^m} = \{a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + ... + a_1x + a_0 : a_i \in \{0, 1\}\} \quad (1)$$

Para esse corpo finito é necessário definir um polinômio $f(x)$ irredutível de ordem m (não pode ser fatorado em um produto de polinômios). Para operações de multiplicação entre polinômios do corpo finito, o resultado é reduzido módulo $f(x)$. Para operação de adição é realizada módulo 2 para cada elemento.

Exemplo: O corpo finito $\mathbb{F}_{2^4}$ apresenta 16 (2^4) polinômios mostrados a seguir:

0	x^2	x^3	$x^3 + x^2$
1	$x^2 + 1$	$x^3 + 1$	$x^3 + x^2 + 1$
x	$x^2 + x$	$x^3 + x$	$x^3 + x^2 + x$
x+1	$x^2 + x + 1$	$x^3 + x + 1$	$x^3 + x^2 + x + 1$

Exemplos de operações nesse corpo finito de exemplo dado o polinômio de redução $f(x) = x^4 + x + 1$:

1. Adição: $(x^3 + x^2 + 1) + (x^2 + x + 1) = x^3 + x$

2. Subtração: $(x^3 + x^2 + 1) - (x^2 + x + 1) = x^3 + x$

3. Multiplicação: $(x^3 + x^2 + 1) \cdot (x^2 + x + 1) = x^2 + 1$, dado que:
   $(x^3 + x^2 + 1) \cdot (x^2 + x + 1) = x^5 + x + 1$
   $(x^5 + x + 1) mod(f(x)) = x^2 + 1$

4. Inversão: $(x^3 + x^2 + 1)^{-1} = x^2$ dado que:
   $((x^3 + x^2 + 1) \cdot (x^2)) mod(f(x)) = 1$

Nesse trabalho não entramos em detalhes nos algoritmos para computar eficientemente as quatro operações no corpo finito binário. Não obstante, é importante notar que operações de adição e subtração são computacionalmente mais simples do que multiplicação (requer aplicar redução módulo $f(x)$) e essa, por sua vez, menos custosa que a operação de inversão. Assim, deve-se evitar operações de inversão para melhor eficiência dos algoritmos sobre esses corpos binários.

2.1.2 Curvas Elípticas

Define-se matematicamente uma curva elíptica E sobre um corpo K (E/K) pela seguinte equação (também chamada de *Equação de Weierstrass*) :

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2}$$

Onde a_1, a_2, a_3, a_4, $a_6 \in K$ e $\Delta \neq 0$, sendo Δ o discriminante de E, definido da seguinte forma:

$$\left.\begin{array}{c} \Delta = -d_2^2 d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \\ d_2 = a_1^2 + 4a_2 \\ d_4 = 2a_4 + a_1a_3 \\ d_6 = a_3^2 + 4a_6 \\ d_8 = a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2 \end{array}\right\} \tag{3}$$

A condição $\Delta \neq 0$ é importante para garantir a propriedade que nenhum ponto na curva possua mais de duas linhas tangentes.

Se L é qualquer extensão do campo K, então, o conjunto de pontos $L - racionais$ na curva E é dado por:

$$E(L) = \left\{(x, y) \in L \times L : y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6 = 0\right\} \cup \{\infty\} \quad (4)$$

Esse conjunto pode ser visto como todos os pontos (x, y) que satisfazem a curva E e possuem as coordenadas x e y pertencentes ao conjunto L (por definição o ponto usado como identidade no infinito ∞ também pertence ao conjunto).

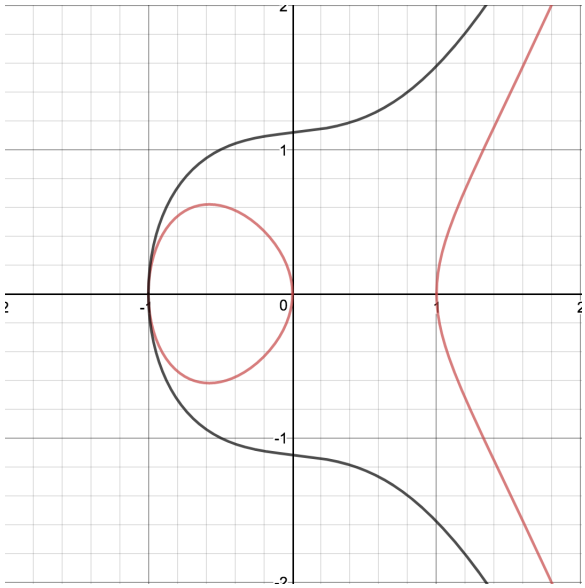

Figura 1: Duas curvas elípticas sobre o conjunto $\mathbb{R}$. A vermelha representa a equação $E_1 : y^2 = x^3 - x$, enquanto que o gráfico preto é a curva $E_2 : y^2 = x^3 + \frac{x}{4} + \frac{5}{4}$

Visualmente uma curva elíptica sobre o corpo real $\mathbb{R}$ pode ser vista conforme a Figura 1, na qual a o gráfico vermelho representa a curva $E_1 : y^2 = x^3 - x$, enquanto que o gráfico preto é a curva $E_2 : y^2 = x^3 + \frac{x}{4} + \frac{5}{4}$. Importante notar a simetria característica vista em relação ao eixo x.

A equação 2 pode ser simplificada para o corpo finito $\mathbb{F}_{2^m}$ que é enfoque desse trabalho, através de uma mudança admissível de variável. A primeira transformação de variável ocorre quando o coeficiente $a_1 \neq 0$, dada por:

$$(x, y) \rightarrow \left(a_1^2x + \frac{a_3}{a_1}, a_1^3y + \frac{a_1^2a_4 + a_3^2}{a_1^3}\right) \quad (5)$$

Com ela chega-se a seguinte curva E_{ns}, chamada de *ordinária*:

$$E_{ns} : y^2 + xy = x^3 + ax^2 + b \quad (6)$$

Onde a, $b \in \mathbb{F}_{2^m}$ e tem discriminante $\Delta = b$.

A segunda transformação é quando $a_1 = 0$, com a seguinte mudança admissível de variável:

$$(x, y) \to (x + a_2, y) \tag{7}$$

Transformando para a seguinte curva E_s, chamada de *supersingular*:

$$E_n : y^2 + cy = x^3 + ax + b \tag{8}$$

Onde a, b, $c \in \mathbb{F}_{2^m}$ e tem discriminante $\Delta = c^4$.

O sistema de criptografia de curvas elípticas é construído em cima do grupo albeliano aditivo $(E(K), +)$, onde K é o corpo finito e E a curva elíptica (note que o ponto ∞ pertencente ao conjunto $E(K)$ é usado como identidade). A regra de adição para esse conjunto é definida geometricamente, dado P e Q, $P+Q$ é o ponto simétrico horizontalmente ao ponto que intersecta a curva na reta formada por P e Q. Caso $P = Q$, essa reta seria a tangente ao ponto em questão.

Para uma curva *ordinária* E_{ns} sobre $\mathbb{F}_{2^m}$, a regra de adição do grupo albeliano dada por:

- Identidade: $P + \infty = \infty + P = P$ para todo $P \in E(\mathbb{F}_{2^m})$

- Negativo: Dado $P = (x, y) \in E(\mathbb{F}_{2^m})$, então $(x, y)+(x, x+y) = \infty$. O ponto $(x, x+y)$ é chamado de $-P$ (note que $-\infty = \infty$).

- Adição de ponto: Dados $P = (x_p, x_p) \in E(\mathbb{F}_{2^m})$ e $Q = (x_q, y_q) \in E(\mathbb{F}_{2^m})$ e $P \neq Q$. Então, a soma $P + Q = (x_{p+q}, y_{p+q})$ é dada por:

  $x_{p+q} = \lambda^2 + \lambda + x_p + x_q + a$

  $y_{p+q} = \lambda(x_p + x_{p+q}) + x_{p+q} + y_p$

  $\lambda = \frac{y_p+y_q}{x_p+x_q}$

- Dobro: Dado $P = (x_p, x_p) \in E(\mathbb{F}_{2^m})$ onde $P \neq -P$, então $2P = (x_{2p}, y_{2p})$ é:

  $x_{2p} = \lambda^2 + \lambda + a$

  $y_{2p} = x_p^2 + \lambda x_{2p} + x_{2p}$

  $\lambda = \frac{x_p+y_p}{x_p}$

As operações citadas acima são implementadas de acordo com os algoritmos do corpo finito subjacente. No caso do $\mathbb{F}_{2^m}$ conforme explicado na seção anterior, por exemplo:

Exemplo: Suponha uma curva *ordinária* E_{ns} sobre $\mathbb{F}_{2^4}$ dada por:

$$E_{ns} : y^2 + xy = x^3 + z^3x^2 + (z^3 + 1) \tag{9}$$

Ou seja, $a = z^3$ e $b = z^3 + 1$. O elemento $a_3z^3 + a_1z^2 + a_1z^1 + a_0 \in \mathbb{F}_{2^4}$ é representado pelo vetor $(a_3a_2a_1a_0)$. Para esse mesmo exemplo, define-se o polinômio de redução como sendo $f(z) = z^4 + z + z$. Assim, os pontos do grupo albeliano $(E(\mathbb{F}_{2^4}), +)$ são:

∞	$(0011, 1100)$	$(1000, 0001)$	$(1100, 0000)$
$(0000, 1011)$	$(0011, 1111)$	$(1000, 1001)$	$(1100, 1100)$
$(0001, 0000)$	$(0101, 0000)$	$(1001, 0110)$	$(1111, 0100)$
$(0001, 0001)$	$(0101, 0101)$	$(1001, 1111)$	$(1111, 1011)$
$(0010, 1101)$	$(0111, 1011)$	$(1011, 0010)$	
$(0010, 1111)$	$(0111, 1100)$	$(1011, 1001)$	

Com adição explicitada anteriormente, temos que $(0010, 1111)+(1100, 1100) = (0001, 0001)$ e $2(0010, 1111) = (1011, 0010)$.

Dado todos os conceitos acima, pode-se definir o problema intratável que garante a viabilidade prática do sistema de criptografia de curvas elípticas, chamado de problema de logaritmo discreto em curvas elípticas (ECDLP, do inglês *Elliptic Curve Discrete Logarithm Problem*):

Definição Dado uma curva elíptica E sobre um corpo finito $\mathbb{F}_q$, um ponto P gerador $E(\mathbb{F}_q)$ de ordem n e um ponto $Q \in \langle P \rangle$, então, encontrar $k \in [0, n-1]$ tal que $Q = kP$ é chamado de problema de logaritmo discreto em curvas elípticas.

Como calcular Q dado k é factível ser realizado eficientemente, conforme será mostrado na próxima Seção 2.1.3, enquanto que para calcular k dado Q (operação inversa) não é conhecido nenhum algoritmo eficiente (sequer algoritmos sub-exponencial, como no caso RSA), pode-se criar um sistema de criptografia assimétrica com k sendo privado, enquanto que Q é tornado público (curva, corpo finito, P e outros parâmetros também são públicos).

2.1.3 Multiplicação por Escalar

A multiplicação de um ponto P por um escalar k é, normalmente, a principal operação do esquema de criptografia sobre curvas elípticas, pois ela domina o tempo de execução dos algoritmos criptográficos. Além disso, essa operação envolve o segredo privado do sistema (escalar k), exigindo além de eficiência também segurança na implementação do algoritmo para não ocorrer vazamento dos bits de k. Para que esse vazamento não ocorra é necessário que o fluxo de execução seja regular, não variando o tempo de execução conforme os bits de k.

O Algoritmo 1 abaixo é um exemplo não regular, conhecido como método *left-to-right, double-and-add* [Rivain 2011]. Esse problema pode ser inferido analisando a linha 6 que computa a soma entre dois pontos, pois essa linha pode ou não ser executada dependendo do valor do bit k_i. Dessa forma, esse algoritmo viabiliza ataques de recuperação do escalar k através do tempo de execução do laço.

Algoritmo 1 Left-to-right double-and-add

Entrada: $P \in E(\mathbb{F}_p), k = (k_{t-1}, \ldots, k_1, k_0)_2 \in \mathbb{N}$
Saída: $kP \in E(\mathbb{F}_p)$

```
1: N ← P
2: Q ← 0
3: for i from t − 1 downto 0 do
4:    N ← 2N
5:    if k_i = 1 then
6:       Q ← Q + N
7:    end if
8: end for
9: return Q
```

É interessante que o algoritmo de multiplicação implementado tenha a propriedade de regularidade, mantendo a eficiência. Nesse sentido a variante do algoritmo *Montgomery ladder* proposta por [López and Dahab 1999] é muito apropriada, visto que consegue alcançar essas duas propriedades. Esse é o algoritmo implementado na versão corrente do *OpenSSL* alvo do ataque desse trabalho.

O algoritmo *Montgomery ladder* de [López and Dahab 1999] é uma implementação eficiente do algoritmo proposto em [Agnew et al. 1993], dado que esse último não apresenta uma eficiente implementação em termos da multiplicação no corpo binário. O algoritmo de [Agnew et al. 1993], por sua vez, é uma variação do algoritmo de Montgomery [Montgomery 1987] para curvas elípticas *ordinária* sobre $\mathbb{F}_{2^m}$.

O algoritmo de Montgomery modifica a abordagem do método binário clássico de multiplicação, usando da observação que a coordenada x da soma de dois pontos, quando se sabe a diferença, pode ser computada usando as coordenadas x dos pontos envolvidos. O algoritmo é mostrado abaixo:

Algoritmo 2 Método Binário

Entrada: $P \in E(\mathbb{F}_p), k = (k_{t-1}, \ldots, k_1, k_0)_2 \in \mathbb{N}$
Saída: $kP \in E(\mathbb{F}_p)$

```
1: P_1 ← P
2: P_2 ← 2P
3: for i from t − 1 downto 0 do
4:    if k_i = 1 then
5:       P_1 ← P_1 + P_2
6:       P_2 ← 2P_2
7:    else
8:       P_2 ← P_2 + P_1
9:       P_1 ← 2P_1
10:   end if
11: end for
12: return Q = P_1
```

Para diminuir o número de inversões (execução lenta) no campo finito, o algoritmo proposto por [López and Dahab 1999] utilizada da representação dos pontos em coordenadas projetivas. Assim, o algoritmo de *Montgomery ladder* de [López and Dahab 1999] é mostrado abaixo:

Algoritmo 3 Multiplicação por escalar Montgomery

Entrada: $P = (x, y) \in E(\mathbb{F}_p), k = (k_{t-1}, \ldots, k_1, k_0)_2 \in \mathbb{N}$
Saída: $Q = kP \in E(\mathbb{F}_p)$

1: **if** $k = 0$ ou $x = 0$ **then**
2: $Q = (0, 0)$
3: **return** Q
4: **end if**
5: Set $X_1 \leftarrow x$, $Z_1 \leftarrow 1$
6: Set $X_2 \leftarrow x^4 + b$, $Z_2 \leftarrow x^2$
7: **for** i from $t - 1$ **downto** 0 **do**
8: **if** $k_i = 1$ **then**
9: $Madd(X_1, Z_1, X_2, Z_2)$
10: $Mdouble(X_2, Z_2)$
11: **else**
12: $Madd(X_2, Z_2, X_1, Z_1)$
13: $Mdouble(X_1, Z_1)$
14: **end if**
15: **end for**
16: **return** $Q = Mxy(X_1, Z_1, X_2, Z_2)$

Importante visualizar que o Algoritmo 3 é regular, ou seja, independente do valor do bit k_i ele faz uma operação $Madd$ e uma $Mdouble$. Essas duas operações são importantes para o trabalho descrito aqui, a operação de $Mdouble$ requer uma multiplicação de dois pontos, uma multiplicação por constante, quatro operações de quadrado, uma variável temporária e nenhuma inversão é demandada, o Algoritmo 4 implementa essa operação:

Algoritmo 4 $Mdouble$: Calcula $2P$

Entrada: Corpo finito $E(\mathbb{F}_p)$; os elementos a e $c = b^{2^{m-1}}$ que definem a curva E; coordenada x projetiva X/Z do ponto P;
Saída: coordenada x X/Z do ponto $2P$

1: $T_1 \leftarrow c$
2: $X \leftarrow X^2$
3: $Z \leftarrow Z^2$
4: $T_1 \leftarrow Z \times T_1$
5: $Z \leftarrow Z \times X$
6: $T_1 \leftarrow T_1^2$
7: $X \leftarrow X^2$
8: $X \leftarrow X + T_1$

Já a operação de $Madd$ requer três multiplicações de dois pontos, uma multiplicação por x, uma operação de quadrado, duas variáveis temporárias e nenhuma inversão é demandada, o Algoritmo 5 implementa essa operação:

Algoritmo 5 $Madd$: Calcula $P_1 + P_2$

Entrada: Corpo finito $E(\mathbb{F}_p)$; os elementos a e b que definem a curva E; coordenada x projetiva X_1/Z_1 do ponto P_1; coordenada x projetiva X_2/Z_2 do ponto P_2

Saída: coordenada x X_1/Z_1 do ponto $P_1 + P_2$

1: $T_1 \leftarrow x$
2: $X_1 \leftarrow X_1 \times Z_2$
3: $Z_1 \leftarrow Z_1 \times X_2$
4: $T_2 \leftarrow X_1 \times Z_1$
5: $Z_1 \leftarrow Z_1 + X_1$
6: $Z_1 \leftarrow Z_1^2$
7: $X_1 \leftarrow Z_1 \times T_1$
8: $X_1 \leftarrow X_1 + T_2$

É demonstrado em [López and Dahab 1999] que o Algoritmo 3 executa exatamente o seguinte número de operações sobre o corpo binário para calcular kP: 1 inversão, $3\lfloor log_2 k \rfloor + 7$ adições, $6\lfloor log_2 k \rfloor + 10$ multiplicações e $5\lfloor log_2 k \rfloor + 3$ quadrados. A única operação de inversão é realizada para converter a coordenada projetiva após calcular a multiplicação na operação Mxy.

2.2 Algoritmo de Assinatura Digital de Curvas Elípticas

Conhecido como ECDSA (do inglês *Elliptic Curve Digital Signature Algorithm*), é uma variação do algoritmo DSA (*digital signature algorithm*), mas construído em cima da base teórica vista anteriormente de curvas elípticas. Ele foi proposto inicialmente por [Vanstone 1992] como resposta ao questionamento da entidade *NIST* (*National Institute of Standards*) para propor esquemas de assinatura digital. Aceito somente em 1998 como padrão *ISO*, em 1999 como padrão *ANSI* e em 2000 *IEEE*.

O esquema de assinatura digital é projetado para prover um método digital que replica a ideia de assinatura feita a mão. Dessa forma, a assinatura digital pode fornecer alguma das garantias de segurança como integridade de dados, autenticação de dados e não repudiação. Idealmente, a assinatura digital não pode ser forjada, quando um atacante recebe uma mensagem com assinatura de A, ele não deve ser capaz de reutilizar a mesma em outra mensagem [Johnson et al. 2001].

Para implementação desse mecanismo é necessário duas principais primitivas: *Assinatura* e *Verificação*. Essas primitivas são desenvolvidas sobre um domínio de curvas elípticas descrito na Seção 2.1, onde a e b são os parâmetros da curva, n é a ordem do grupo cíclico gerado por $P \in E(K)$. As primitivas são descritas para o ECDSA conforme os algoritmos abaixo:

Algoritmo 6 Assinatura: ECDSA

Entrada: Domínio ECC (a, b, P, n); chave privada d, mensagem m; Algoritmo de hash H
Saída: Assinatura (r, s), com r e s valores inteiros

1: Seleciona $k \in [1, n-1]$
2: $(x_1, y_1) \leftarrow kP$ e converte x_1 para um inteiro $\bar{x_1}$
3: $r \leftarrow \bar{x_1} \bmod n$.
4: **if** $r = 0$ **then**
5: Retorna para passo 1
6: **end if**
7: $e \leftarrow H(m)$
8: $s \leftarrow k^{-1}(e + dr) \bmod n$
9: **if** $s = 0$ **then**
10: Retorna para passo 1
11: **end if**
12: **return** (r, s)

Algoritmo 7 Verificação: ECDSA

Entrada: Domínio ECC (a, b, P, n); chave pública Q, mensagem m; assinatura (r, s); Algoritmo de hash H
Saída: Assinatura aceita ou negada

1: **if** $r \notin [1, n-1]$ ou $s \notin [1, n-1]$ **then**
2: **return** "Assinatura negada"
3: **end if**
4: $e \leftarrow H(m)$
5: $w \leftarrow s^{-1} \bmod n$
6: $u_1 \leftarrow ew \bmod n$
7: $u_2 \leftarrow rw \bmod n$
8: $(x_1, y_1) \leftarrow u_1 P + u_2 Q$
9: **if** $(x_1, y_1) = \infty$ **then**
10: **return** "Assinatura negada"
11: **end if**
12: Converte x_1 para inteiro $\bar{x_1}$
13: $v \leftarrow \bar{x_1} \bmod n$
14: **if** $v = r$ **then**
15: **return** "Assinatura aceita"
16: **else**
17: **return** "Assinatura negada"
18: **end if**

O passo 1 do Algoritmo 6 é de vital importância, pois reutilizar o inteiro aleatório k (chamado de *nonce*) levou a quebra da implementação do ECDSA da empresa Sony no Playstation 3. De fato, conhecimento do *nonce* implica em descobrir a chave privada d do

assinante, dado que s, r, e e n são informações acessíveis por um atacante e, matematicamente, tornam d computável em função de k [Yarom and Benger 2014]:

$$d = r^{-1}(ks - e) \mod n \tag{10}$$

Não somente com o conhecimento integral do *nonce* k é possível recuperar a chave privada d, mas também com o conhecimento parcial dos bits em conjunto com as informações acessíveis da assinatura. Entretanto, diferente da equação 10 que depende somente de uma assinatura, a abordagem com conhecimento parcial exige um número bem maior de assinaturas. Essa abordagem será explorada no ataque descrito nesse trabalho e, portanto, estará em maior detalhe apresentada na Seção 4.

3 Ataques de canal lateral sobre curvas

Para avaliar a segurança de um protocolo criptográfico é assumido que o atacante tem completo conhecimento sobre protocolo e os dados trocados publicamente entre as partes (chave pública, dado cifrado, parâmetros e etc). Com essas informações, o adverśario deve atacar o sistema por ser capaz de resolver o problema matemático intratável ou explorando alguma falha de fluxo do protocolo.

Os ataques tradicionais exploram falhas matemáticas na especificação do protocolo. Entretanto, recententemente ataques que exploram especificações de implementação e ambientes de execução vem sendo estudados. Essa última classe de ataques é conhecida como ataque de canal lateral, a qual utiliza do vazamento de informações durante a execução de algum algoritmo criptográfico e, através dessa informação, conseguir recuperar informação privada.

Como o ataque é na implementação e não no algoritmo em si, a informação obtida nem sempre afeta todos os ambientes, por exemplo, recuperar a informação privada através do consumo de potência pode ser plausível em um smart card, entretanto, em uma estação de trabalho segura essa técnica não é factível. Então, analisar a praticabilidade do ataque é um importante fator do ataque de canal lateral.

Existem diversos vazamentos de informações que podem ser exploradas para realizar ataques, tais como electromagnetismo, consumo de potência, tempo de execução entre outras. Por exemplo, a Figura 2 mostra o consumo de potência durante a execução da multiplicação por escalar, a qual poderia ser usado para encontrar o escalar em uma execução do ECDSA e, assim, revelar a chave privada.

O ataque de temporização utiliza do fato que o tempo para executar uma aritmética pode variar de acordo com os operandos. Com isso em mente, o atacante pode medir o tempo de execução com precisão e, analisando esses dados, deduzir qual era a informação secreta usada. Espera-se que os ataques de temporização sejam especialmente difíceis de montar em esquemas de assinatura de curva elíptica, como ECDSA, dado que para cada assinatura um novo escalar é invocado, revelando menos informação que RSA e DES.

Apesar dessa dificuldade, o ataque realizado em [Yarom and Benger 2014] conseguiu com sucesso recuperar a chave privada do ECDSA utilizando da técnica de temporização no código implementado pelo *OpenSSL*. Pois, conforme o trabalho mostra, os bits do escalar

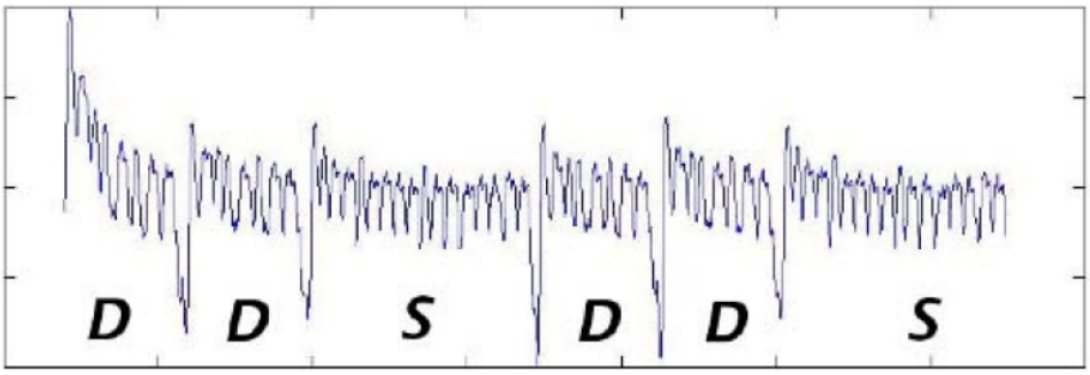

Figura 2: Consumo de potência durante o cálculo da operação de multiplicação por escalar das curvas elípticas [Hankerson et al. 2004]

podem ser recuperados com técnicas de diferenciação entre tempo de cache e memória e, dado o valor do *nonce* a chave privada é revelada. Essa mesma técnica é também implementada no trabalho descrito nesse documento e será explorada na Seção 4.

4 Descrição do método de ataque

O ataque desse trabalho tem como alvo a implementação da versão corrente do *OpenSSL 1.1.0g* [The OpenSSL Project 2003] do algortimo ECDSA, descrito na Seção 2.2. O objetivo é conseguir recuperar a chave secreta d usada para assinar, através do monitoramento do passo 2 do Algoritmo 6, que implementa a multiplicação do *nonce*, escolhido aleatoriamente para cada assinatura, pelo ponto gerador P.

Para cada assinatura, através de uma abordagem de ataque de canal lateral, é recuperado o bit mais significativo (MSB) do *nonce*. Computada essa operação para muitas assinaturas, usa-se a abordagem de Bleichenbacher descrita em [Aranha et al. 2014] para recuperar a chave secreta d. Portanto, o ataque pode ser dividido em duas partes: (i). Recuperar o MSB do *nonce*; (ii). Aplicar Bleichenbacher. O trabalho descrito nesse documento refere-se somente à parte (i) do ataque.

Apesar do Algoritmo 3 para multiplicação por escalar descrito na Seção 2.1.3 ter fluxo de execução regular, é possível obter o MSB do *nonce* através de um ataque de temporização. Esse vazamento acontece pelo fato do bit MSB controlar a necessidade ou não de redução modular em algumas das linhas das primitivas $Madd$ e $Mdouble$. Esse controle acontece através de três situações: primeiro, no Algoritmo 3 os parâmetros Z_1 e Z_2 são trocados de acordo com o bit do *nonce*; segundo, para o MSB o parâmetro Z_1 é fixado como constante 1, enquanto que os outros parâmetros (incluindo Z_2) são fixados de acordo com a coordenada x do ponto P (polinômio que, provavelmente, não é uma constante); terceiro, polinômio constante não exigirá redução modular em operações de multiplicação e quadrado, já o polinômio não constante poderá exigir. Portanto, de acordo com o bit do MSB algumas linhas das primitivas $Madd$ e $Mdouble$ irão ter diferença de tempo, o que pode ser capturado

por um ataque de canal lateral de temporização.

Existem algumas linhas de $Mdouble$ e $Madd$ em que podem ser verificadas essa diferença de tempo. Por exemplo, quando $k_{t-1} = 1$ a primitiva $Madd$ fica como $Madd_1(x, 1, x^4, x^2)$, enquanto que $k_{t-1} = 0$ fica $Madd_0(x, x^2, x^4, 1)$. A linha 2 do Algoritmo 5 para o caso $Madd_1$ fica $x \times x^2$ (provavelmente irá precisar de redução modular), já para o caso $Madd_0$ a linha será $x \times 1 \equiv x$ (não necessita de redução modular). Analogamente, as linhas 3, 4 e 5 do $Mdouble$ também possuem essa propriedade.

Para colocar em prática a análise anterior utilizamos a abordagem de canal lateral *FLUSH+RELOAD* [Yarom and Falkner 2014] para determinar o número de ciclos requerido para executar uma determinada linha de código. Essa técnica permite atacar o programa alvo mesmo não estando no mesmo núcleo, visto que ela depende apenas do acesso à última nível de cache (compartilhada entre os núcleos). Assim, *F+R* é bem conveniente para ambientes com virtualização de máquinas.

A técnica de *FLUSH+RELOAD* pode ser definida como a tentativa de dizer se uma dada instrução de código foi executada ou não em um espaço de tempo. Ela é baseada na diferença de tempo entre uma instrução buscada na memória (*cache miss*) com uma instrução buscada na cache (*cache hit*). Para forçar essa diferença na instrução monitorada três fases são usadas: *Flush*, *Wait* e *Reload*, realizadas nessa ordem: 1. elimina dados da linha de cache que tem a instrução de código monitorado (de toda hierarquia de cache); 2. apenas aguarda por um período (chamado de *slot time*); 3. então, acessa a instrução de código monitorado e calcula o tempo t_r ciclos para esse acesso. Como existe uma diferença de tempo entre acessar da cache (t_c) e da memória, então, se $t_r < t_c$ implica que o dado originou da cache (usado no período de *Wait*), caso contrário o dado não foi acessado. O parâmetro t_c é chamado de *threshold* (limiar) do algoritmo de *FLUSH+RELOAD* é dependente da arquitetura da máquina, portanto, saber esse valor para a máquina da vítima é um passo essencial para implementação em cenários reais de ataque.

Para calcular o número de ciclos executados entre duas linha de código I_1 e I_2, coloca-se uma sonda em cada linha. O resultado do *F+R* é a afirmação se no período de *slot time* aquela instrução foi utilizada. Após o slot que teve I_1 acessado, conta-se o número de slots passados até encontrar um slot em que I_2 foi acessado. Então, o número de ciclos para computar de I_1 para I_2 é dado pelo número de ciclos do *Wait* multiplicado pelo número de slot.

Embora pela colocação anterior pode-se concluir que quanto menor for o *slot time* mais acurado será o cálculo, a técnica de *F+R* não funciona com valores pequenos de *slot time*. A razão para isso pode ser verificada na Figura 3, se o *slot time* for muito pequeno implica em muitos casos de sobreposição, pois o tempo de *Reload* e *Flush* não pode ser considerado desprezível. Entretanto, se o *slot time* for muito longo pode acontecer de diversos acessos durante a fase de *Wait* (caso E da Figura 3). Consequentemente, escolher o número de ciclos da fase de *Wait* é um *tradeoff* entre a resolução do ataque e a probabilidade de perder acessos [Yarom and Benger 2014].

A implementação do *FLUSH+RELOAD* é mostrada na Figura 4. A implementação é feita diretamente em código de máquina para a arquitetura x86. A operação essencial é feita pelo comando $cflush$ na linha 14, o qual garante esvaziar a linha de um cache em todos os níveis e não precisa de privilégio. Outra operação importante nesse código é a contagem

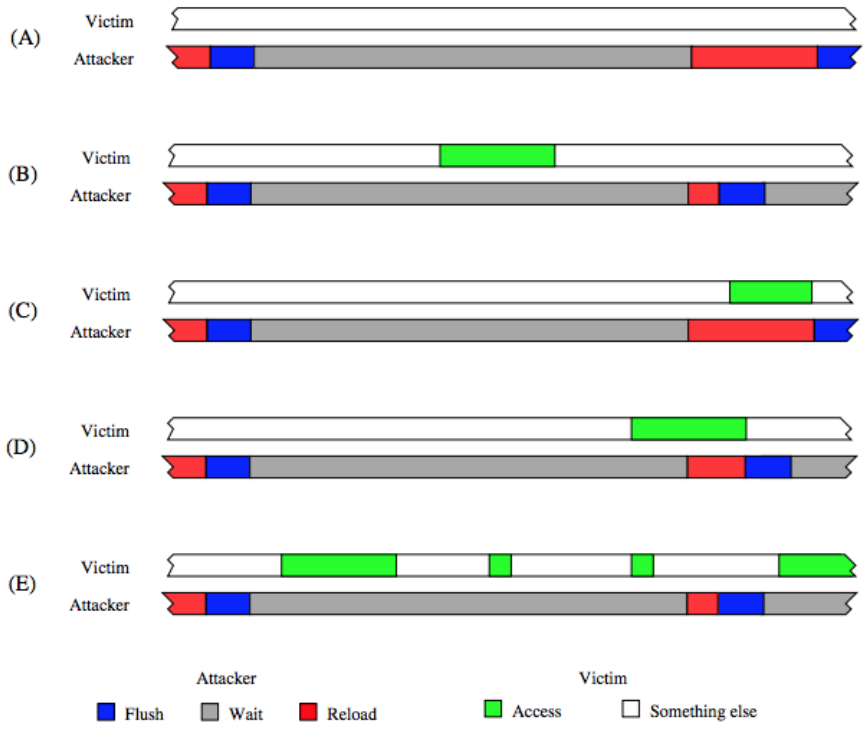

Figura 3: Ilustração dos casos possíveis do *F+R* [Yarom and Falkner 2014]. (A) Sem acesso da vítima; (B) Acesso da vítima durante slot; (C) Acesso com sobreposição total; (D) Acesso sobreposição parcial; (E) Diversos acessos durante o slot

de ciclos que é feita pela diferença do registrador *rdtsc* em dois momentos, o qual salva o número de *timestamp* desde o reset.

Conforme [Walter 2004] mostra, ataques de canal lateral têm resolução diretamente proporcional ao tamanho da chave atacada, pois chaves mais longas implicam em um tempo maior entre os extremos. No caso do *FLUSH+RELOAD* a resolução possível está limitada pelo *slot time*, que se comporta melhor em curvas mais longas. Contudo, o ataque discutido aqui depende de duas partes e, na parte 2, chaves maiores implicam em necessidade de um número maior de assinaturas. Para endereçar esse problema o ataque se concentra em chaves mais curtas, mas para tentar manter uma boa resolução emprega a abordagem de degradação de desempenho [Allan et al. 2016].

A degradação aumenta o tempo consumido pelo algoritmo, usando para isso do fato de que uma parcela pequena do código é executada com uma frequência muito maior que o restante dos algoritmos criptográficos (chamado de *hot code*). Essa parte menor do código fica quase todo tempo na cache, diminuindo significativamente o tempo de execução do algoritmo. Portanto, o retardamento da execução pode ser alcançado eliminando o *hot code*

```
1  int probe(char *adrs) {
2    volatile unsigned long time;
3
4    asm __volatile__ (
5      "  mfence             \n"
6      "  lfence             \n"
7      "  rdtsc              \n"
8      "  lfence             \n"
9      "  movl %%eax, %%esi  \n"
10     "  movl (%1), %%eax   \n"
11     "  lfence             \n"
12     "  rdtsc              \n"
13     "  subl %%esi, %%eax  \n"
14     "  clflush 0(%1)      \n"
15     : "=a" (time)
16     : "c" (adrs)
17     :  "%esi", "%edx");
18   return time < threshold;
19 }
```

Figura 4: Código base para implementar a técnica de *F+R*

da cache continuamente, realizado pela degradação de desempenho através da execução da instrução de *cflush* diversas vezes.

Suponha que uma primitiva demora 2.000 ciclos em média para quando a entrada é 1 e demora 500 ciclos em média para entrada 0. Com essa diferença de apenas 1.500 ciclos não é factível usar a técnica *F+R* para determinar qual foi a entrada, visto que o *slot time* precisaria ser muito curto, o que geraria demasiadas sobreposições. Mas se essa primitiva possuir *hot code*, então, aplicando degradação de desempenho pode-se alcançar taxas de retardamento de até 20 vezes (depende do número de atacante e do código), conforme mostra o trabalho [Allan et al. 2016]. O tempo, nesse exemplo, passaria para 40.000 para 1 e 10.000 para 0, ou seja, uma diferença amplificou para 30.000 ciclos viabilizando um ataque de *FLUSH+RELOAD*.

Em resumo, o ataque tem como finalidade encontrar a chave privada d usada no ECDSA sobre $\mathbb{F}_{2^m}$ através do vazamento do MSB bit do *nonce* de muitas assinaturas. As etapas desse ataque podem ser descritas sucintamente como:

1. Alterar diretamente o código para encontrar verificar a diferença de tempo no Algoritmo 3 implementado pelo *OpenSSL*
2. Analisar a viabilidade do ataque sobre diferentes curvas
3. Encontrar pontos de *hot code* para aplicar degradação de desempenho
4. Aplicar *FLUSH+RELOAD* em conjunto com degradação de desempenho para decidir se o MSB é 0 ou 1

5. Aplicar a abordagem de Bleichenbacher descrita em [Aranha et al. 2014] em cima das assinaturas conseguidas e o bit extraído da etapa anterior (não é feito nesse trabalho)

5 Resultados experimentais

Conforme foi analisado de maneira teórica, o bit mais significativo do *nonce* k pode ser obtido durante a computação da multiplicação do ponto kP na implementação do *OpenSSL* do Algoritmo 3. Esse vazamento ocorre em algumas das computações das primitivas $Madd$ e $Mdouble$.

Para validar essa hipótese foram realizados experimentos no código do *OpenSSL* que implementa a multiplicação (*bn2_mult.c*). Foi adicionado uma função *inline* para capturar o número de ciclos através do registrador *rdtsc* (*time stamp counter*), a qual é mostrada na Figura 5. Assim, o número de ciclos entre dois pontos do código pode ser calculado como $rdtsc(t_{i+1}) - rdtsc(t_i)$, onde $t_{i+1} > t_i$.

```
1  static __inline__ unsigned long long rdtsc(void)
2  {
3      unsigned hi, lo;
4      __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
5      return ( (unsigned long long)lo)|( ((unsigned long long)hi)<<32 );
6  }
```

Figura 5: Função implementada para obter o número de ciclo desde de o último reset (registrador *rdtsc*). Dessa forma, pode se calcular em número de ciclos passados entre t_i e t_{i+1}, como: $rdtsc(t_{i+1}) - rdtsc(t_i)$

Usando essa função, podem ser extraídas as análises estatísticas temporais das operações em relação ao MSB para as primitivas $Madd$ e $Mdouble$. É importante notar que o Algoritmo 5 ($Madd$) é implementado exatamente dessa forma no *OpenSSL*, enquanto que o Algoritmo 4 ($Mdouble$) é implementado com uma versão ligeiramente diferente na execução das operação sobre as coordenadas. A versão do $Mdouble$ do *OpenSSL* é dada pelo Algoritmo abaixo:

Algoritmo 8 OpenSSL $Mdouble$: Calcula $2P$

Entrada: Corpo finito $E(\mathbb{F}_p)$; os elementos a e b que definem a curva E; x-coordenada projetável X/Z do ponto P;
Saída: x-coordenada X/Z do ponto $2P$

1: $X \leftarrow X^2$
2: $T_1 \leftarrow Z^2$
3: $Z \leftarrow X \times T_1$
4: $X \leftarrow X^2$
5: $T_1 \leftarrow T_1^2$
6: $T_1 \leftarrow b * T_1$
7: $X \leftarrow X + T_1$

Quando Z_1 é fixado para 1 (caso do MSB), no Algoritmo 8 todas as linhas vão mudar de acordo com o MSB, exceto as linhas 1 e 4. Já para o Algoritmo 5, quando Z_1 é fixado para 1, as linhas 2 e 3, separadamente, podem informar o MSB. Logo, pode-se analisar 4 casos:

1. Algoritmo 8: linhas 2, 3, 4, 5, 6 e 7 juntas
2. Algoritmo 8: linhas 5, 6 e 7 juntas
3. Algoritmo 5: linha 2
4. Algoritmo 5: linha 3

Nos experimentos foi utilizado a curva de Koblitz sobre corpo binário chamada de $sect163r1$. Essa curva equivale a segurança de uma chave de 1024 bits RSA. Essa curva possui a seguinte especificação:

- Curva E_{ns} sobre $\mathbb{F}_{2^{163}}$ definida por:
  a = 0x07B6882CAAEFA84F9554FF8428BD88E246D2782AE2
  b = 0x0713612DCDDCB40AAB946BDA29CA91F73AF958AFD9

- Polinômio de redução $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$

- Ponto base G é dado por (forma serial não compactada):
  0x040369979697AB43897789566789567F787A7876A65400435EDB42EFAF
  B2989D51FEFCE3C80988F41FF883

- Ordem n sendo:
  0x03FFFFFFFFFFFFFFFFFFF48AAB689C29CA710279B

Foi criada uma chave privada utilizando os parâmetros dessa curva através do seguinte comando do *OpenSSL*:

```
$  openssl ecparam -out ec_key.pem -name sect163r1 -genkey
```

Com essa chave foram realizados os testes das 4 hipóteses levantadas anteriormente. Cada teste utilizou 100.000 assinaturas ECDSA computando o tempo em ciclos da hipótese usando a função de *rdtsc*.

Conforme pode ser visto, as hipóteses foram confirmadas e, de fato, aquela que apresenta maior diferença de tempo é a $Mdouble$, tendo quase a primitiva inteira variando no valor fixo de Z.

Outro ponto interessante de notar é que única situação em que o *nonce* com MSB 0 precisa de mais ciclos do que os *nonces* com MSB 1, é na hipótese 4 9. Esse fato era esperado, visto que na linha 3 do Algoritmo 5, quando $MSB = 0$ tem-se $Z_1 = x^2$ e $Z_2 = 1$

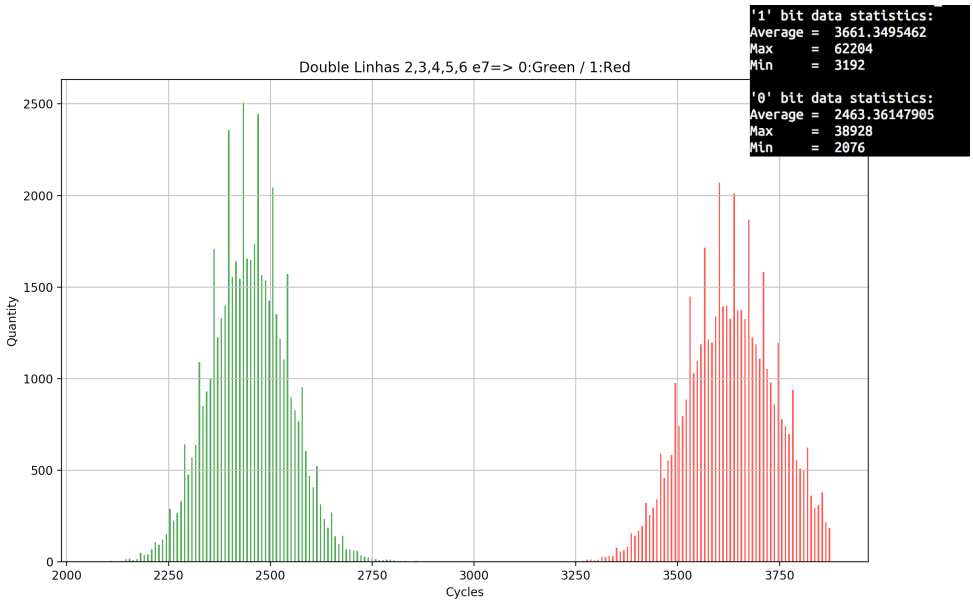

Figura 6: Hipótese 1: Algoritmo 8 contando o tempo desde a linha 2 até a linha 7. Testado com 100.000 assinaturas na curva $sect163r1$ do ECDSA.

(linha 3 pode precisar de redução modular), contrariamente, para $MSB = 1$ tem-se $Z_1 = 1$ e $Z_2 = X^2$ (não tem redução modular, ocorrendo em menos ciclos). Logo, essa linha deve ser atacada para casos em que é requerido conseguir saber os *nonces* com MSB 0. Todos os outros casos a resolução para o MSB 1 é maior.

5.1 Recuperando MSB

Por questões práticas para implementação da técnica de *FLUSH+RELOAD*, foi optado por estudar mais a fundo a hipótese 3, a qual define o bit MSB através do tempo de execução da linha 2 do Algoritmo 5 $Madd$. Ainda para o caso testado na seção anterior, é possível plotar o gráfico de precisão e revocação mostrado na Figura 10.

Precisão P é a medida estatística que estima a probabilidade de uma classificação feita pelo método ser correta, para uma dada classe. Assim, para o caso aqui estudado está interessado no bit MSB 1, consequentemente, a precisão diz respeito somente aos *nonces* classificados como MSB 1.

$$P = \frac{TP}{FP + TP} \tag{11}$$

Onde TP é o número elementos positivo verdadeiro (bits classificados como 1 que são 1) e FP número de elementos falso positivo (bit 0 classificado como 1). Ou seja, se estamos focado em classificar o MSB como bit 1, a precisão diz quanto o método faz afirmações positivas. Ou seja, se estamos focado em classificar o MSB como bit 1, a precisão diz quanto o método faz afirmações positivas.

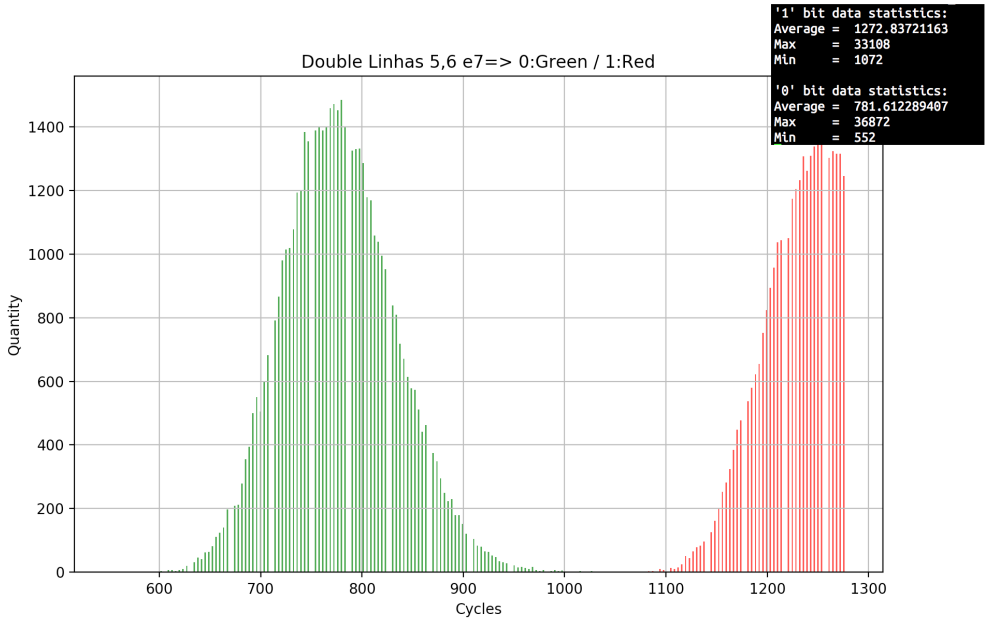

Figura 7: Hipótese 2: Algoritmo 8 contando o tempo desde a linha 5 até a linha 7. Testado com 100.000 assinaturas na curva $sect163r1$ do ECDSA.

Revocação R por sua vez é a medida estatística para medir o quanto de uma classe o método consegue recuperar. A fórmula é dada por:

$$R = \frac{TP}{FN + TP} \tag{12}$$

Onde TP é o mesmo informado anteriormente e FN é o número de elementos falsos negativos (bit 1 que foi classificado como 0). Note que a revocação e precisão são medidas que se auto complementam. Contudo, nesse trabalho o interesse é bem maior em precisão, mesmo que se perda alguns *nonces*, pois a segunda parte do ataque requer bastante certeza na afirmação do valor de MSB do *nonce*.

A Figura 10 mostra a análise de precisão e revocação para classificação do MSB como 1, quando usa-se de um limiar (*threshold*) para separar bits 1 e 0 dos dados gerados pela hipótese 3 da seção anterior. A utilização de um limiar é simples, mas pode funcionar bem para casos em que estão bem separado os dois conjuntos, como nesse caso mostrado.

Apesar de os dados estarem claramente separados, a quantidade de ciclos não é suficiente para aplicar o *FLUSH+RELOAD* com uma resolução boa. Assim, utiliza da degradação de desempenho para amplificar o número de ciclos entre o MSB 0 e 1.

Encontrar o *hot code* é relativamente trivial para essa aplicação. Conforme foi explicado na Seção 4, a diferença de ciclos é devido a utilização ou não da operação de redução modular do polinômio de acordo com o MSB do *nonce*. Assim, para amplificar o ataque dois processos atacantes são utilizados para fazer *flush* continuamente da operação $t^{p[k]}$ que reduz um polinômio (parte da função *BN_GF2m_mod_arr* com laço mais interno). Esse

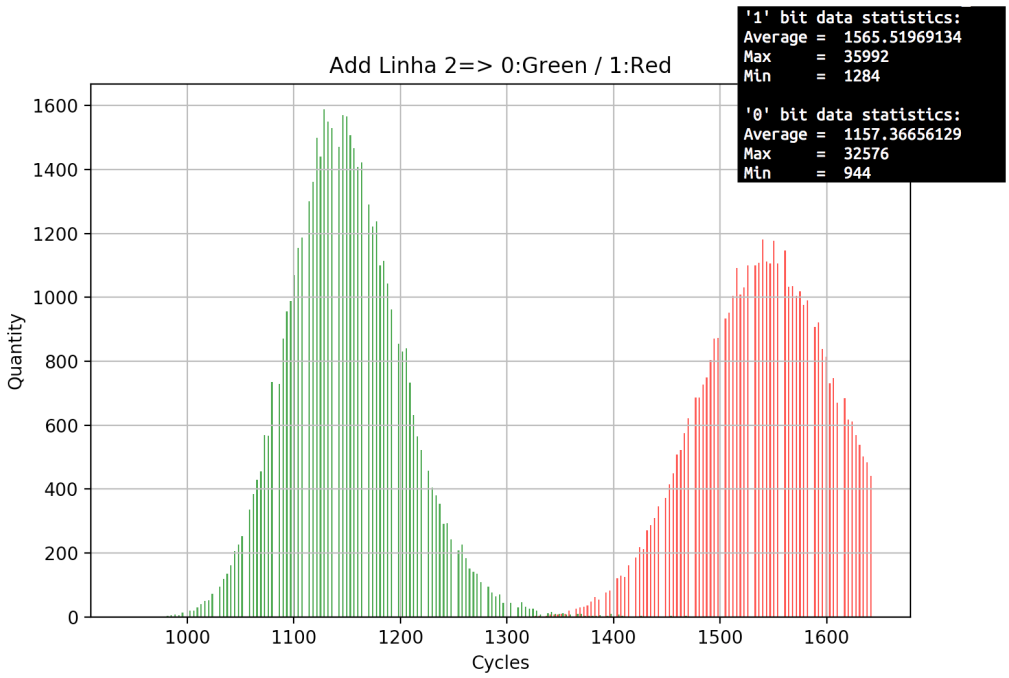

Figura 8: Hipótese 3: Algoritmo 5 contando o tempo da linha 2. Testado com 100.000 assinaturas na curva $sect163r1$ do ECDSA.

método pode conseguir em média 10 vezes de retardamento, aumentando a diferença de 1.000 ciclos para 10.000 ciclos, tornando agora esse vazamento susceptível ao ataque de canal lateral .

Com 10.000 ciclos de separação entre 0 e 1, pode-se usar o *slot time* como 2.500 ciclos. Com esse cenário, colocam-se duas sondas para rastrear a execução do ECDSA do *OpenSSL*. A primeira na chamada da linha 2 do Algoritmo 5 e outra na linha 3 do mesmo algoritmo. O número de slots entre esses dois acessos é usado para determinar qual é o MSB do *nonce*.

Essa abordagem foi testada em uma máquina com configuração descrita na Figura 11, usando a implementação tanto do *FLUSH+RELOAD* quanto da degradação de desempenho fornecida pelo ToolKit de ataque de canal lateral *Mastik* [Yarom 2016]. Foram coletados 10.000 rastros de assinaturas ECDSA fornecida como resposta do *F+R*. Esse processo atacante foi executado da seguinte maneira:

```
$  ./FR-trace -r 10000 -F results/out.%05d.dat -l 10000
  -s 2500 -c 10000 -l 500 -p 2 -H -f ./openssl-debug
  -m ec2_mult.c:143 -m ec2_mult.c:145
  -t bn_gf2m.c:410 -t bn_gf2m.c:410+64
```

O *OpenSSL* foi ligeiramente alterado para, após execução da multiplicação por escalar, imprimir no *stderr* o MSB daquele escalar. Essa mudança não afeta o ataque, sendo utilizada apenas para validar o método usado. A vítima executou uma versão de *debug* como:

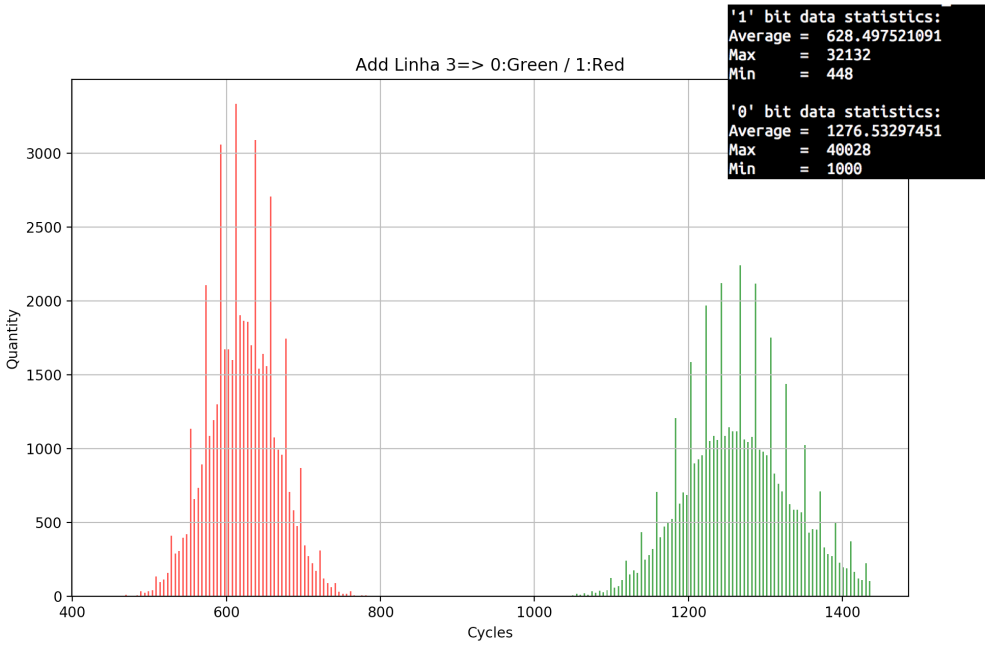

Figura 9: Hipótese 4: Algoritmo 5 contando o tempo da linha 3. Testado com 100.000 assinaturas na curva $sect163r1$ do ECDSA.

```
$  for i in {1..100}; do echo $i; date >> log;
./openssl-debug dgst -ecdsa-with-SHA1 -sign ec_key.pem f
> /dev/null 2>> log; sleep 3;done
```

A Figura 12 mostra o exemplo de dois arquivos obtidos do rastro feito pelo atacante. Do lado esquerdo da imagem tem-se o rastro de uma assinatura com bit MSB 0 do *nonce*, enquanto que o da direita par ao bit 1. No arquivo do 0 pode se ver que o tempo levado entre as duas sondas serem acessadas foi de 1 *slot time* (2.500 ciclos), pois a primeira ocorre na linha 40 e a segunda na 41. Enquanto que para o arquivo do bit 1 (direita), os acessos ocorreram na linha 40 e 45, levando 5 *slot times* ou 12.500 ciclos (5×2.500). Além disso, é importante notar que no início do arquivo do rastro o valor do *timestamp* está definido com precisão de segundos.

O ataque das 10.000 assinaturas levou, aproximadamente, 8 horas para ser concluído. Para analisar os dados foi usado o Algoritmo 9 abaixo, o qual foi encontrado após uma análise visual do comportamento dos dados coletados e em conjunto com a análise temporal feito na seção anterior, foi proposto o seguinte algoritmo para determinar o MSB do *nonce* baseado no rastro do *FLUSH+RELOAD*:

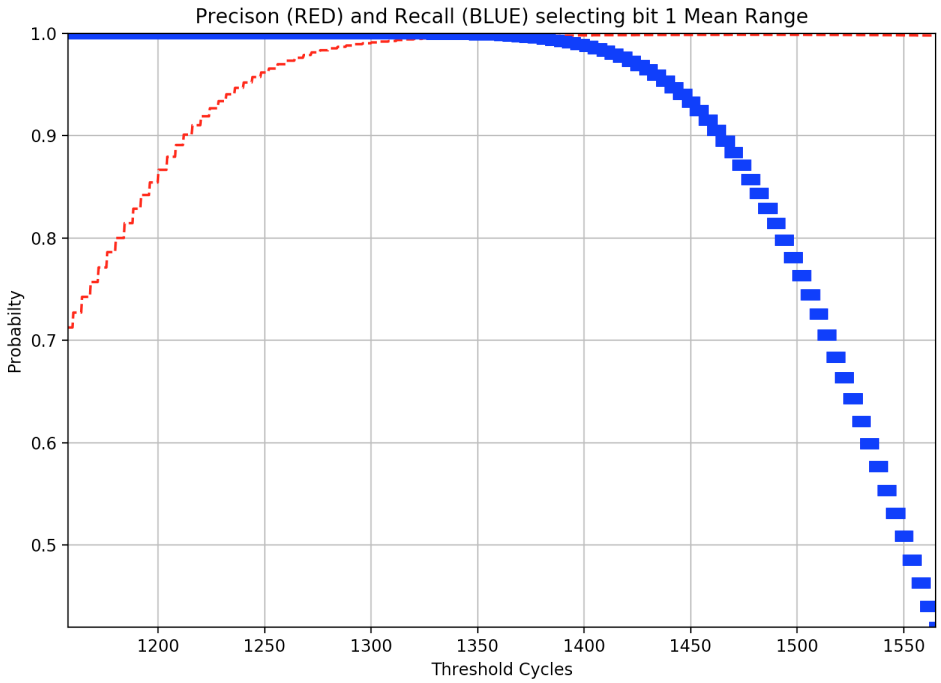

Figura 10: Gráfico de precisão e revocação dado a escolha de um limiar (*threshold*) para separar os dados gerados no teste da hipótese 3 da Seção 5

Algoritmo 9 Determinação do MSB do *Nonce*

Entrada: Matriz $\mathbb{M}_{nx2}$ representando o monitoramento da assinatura digital. $\mathbb{M}[i][j]$ indica o tempo em ciclos da i-éssima fase de *Reload* para a sonda j; T_c valor do limiar para *FLUSH+RELOAD* na máquina do rastro $\mathbb{M}[n][2]$ para o segundo

Saída: Bit do *nonce* 0 ou 1 ou X (sem informação suficiente)

1: $start \leftarrow 0$
2: **for** i from 2 **to** n **do**
3: **if** $\mathbb{M}[i][1] < T_c$ e $start = 0$ **then**
4: **if** $i < 10$ ou $i > 17$ **then**
5: **return** X
6: **end if**
7: $start \leftarrow i$
8: **end if**
9: **if** $\mathbb{M}[i][2] < T_c$ e $start > 0$ **then**
10: $s \leftarrow i - start$
11: **if** $s = 2$ ou $s > 6$ **then**
12: **return** X
13: **else if** $s \geq 3$ **then**
14: **return** 1
15: **else**
16: **return** 0
17: **end if**
18: **end if**
19: **end for**
20: **return** X

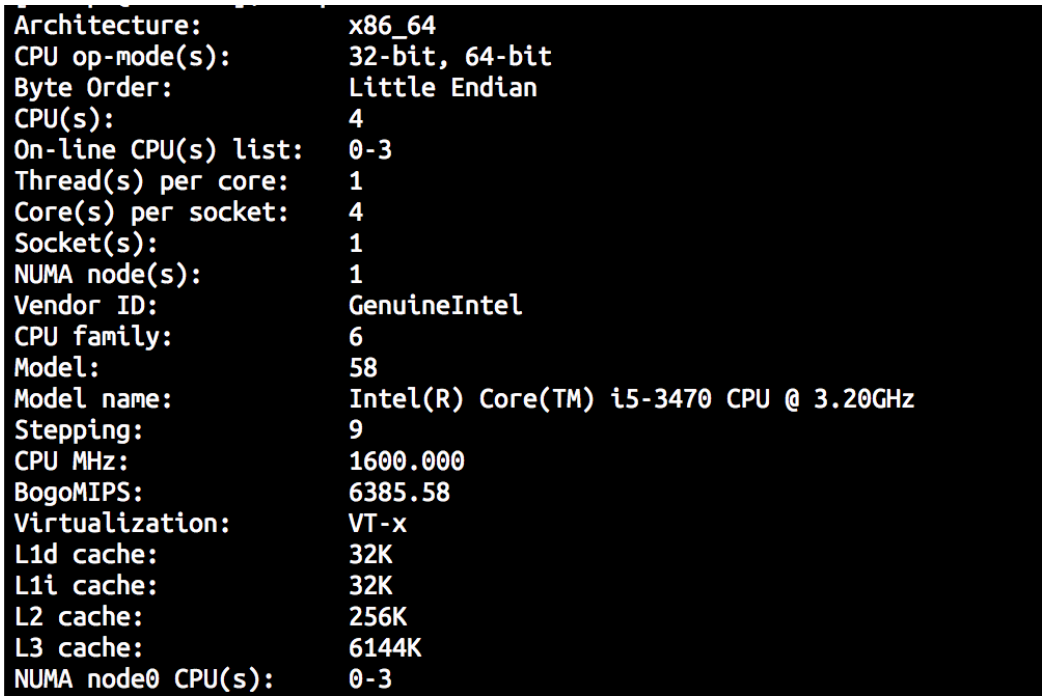

Figura 11: Descrição técnica da máquina usada para todos os testes descrito nesse trabalho.

Foi aplicado o Algoritmo 9 usando $T_c = 100$ (valor obtido através do conhecimento de ataques feitos anteriormente na máquina de teste) nos arquivos de rastro gerado pelo comando do atacante (*out.00000.dat*, *out.00001.dat*, ..., *out.09999.dat*). Esses arquivos contêm o instante de tempo (*timestamp*) do rastro e a matriz $\mathbb{M}_{nx2}$ das fases de *Reload* do rastro, assim, foi gerado como saída o vetor $V_{estimativa}$ referente à predição feita pelo atacante para as assinaturas. Esse vetor contém 10.000 tuplas contendo cada uma delas a informação do *timestamp* do rastro e o bit de saída da do Algoritmo 9 (0, 1 ou X).

Com o vetor $V_{estimativa}$ foi estimado o erro da predição do atacante usando o Algoritmo 9. Para tanto, foi comparados os valores desse vetor com o arquivo *log* gerado pelo comando da vítima. Esse arquivo contém, para cada assinatura gerada, o bit MSB do *nonce* (código

Tabela 1: Resultados do teste do ataque em 10.000 assinaturas ECDSA sobre a curva *sect163r1* que levou 8 horas para ser realizado. O atacante emprega *FLUSH+RELOAD*, degradação de desempenho e o Algoritmo 9 para determinar o bit do nonce. O Erro pode ter três razões: X (algoritmo não sabe determinar o bit), Bit (bit correto é outro) e *timestamp* (sem *log* com *timestamp* no rastro).

	Acertou	**Errou**			**Total**
		X	**Bit**	***timestamp***	
Nonce bit 1	4421	66	511	46	5.044
Nonce bit 0	4738	57	123	38	4.956

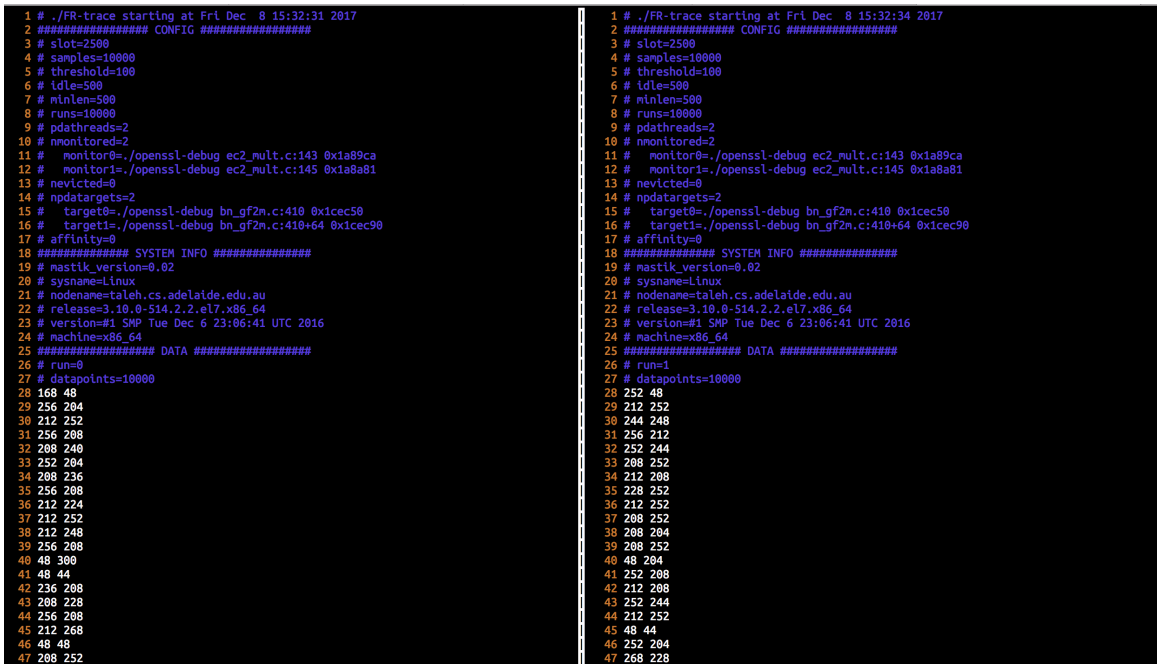

Figura 12: Exemplo de parte de dois rastros do teste de 10.000 assinaturas gerados pela execução do *Mastik*. O arquivo da esquerda é um exemplo de uma assinatura com bit do *nonce* MSB 0 e o da direita do bit 1.

modificado do *OpenSSL*) e o instante de tempo de início da assinatura. Comparando os valores dos bits de $V_{estimativa}$ com os valores de *log* para as entradas que contêm o mesmo *timestamp*, foi gerada a Tabela 1. Essa tabela divide o erro de um bit em três possíveis situações: X (o algoritmo 9 não conseguiu estimar o bit), Bit (o bit correto é o oposto do estimado) e *timestamp* (o bit da assinatura não possui nenhum arquivo de rastro com mesmo *timestamp*). Como pode ser visto na Figura 11, o *timestamp* do rastro tem precisão de 1 segundo e, como o comando da vítima é gerado somente a cada 3 segundos, o erro para esse campo acaba sendo minimizado.

Pela análise da Tabela 1, pode-se estimar que a precisão P_1 e revocação R_1 da abordagem implementada para determinar o bit 1:

$$\begin{aligned} P_1 &= \frac{TP}{TP+FP} \\ P_1 &= \frac{4421}{4421+123} \\ P_1 &= 0,972931338 \end{aligned} \tag{13}$$

$$\begin{aligned} R_1 &= \frac{TP}{TP+FP} \\ R_1 &= \frac{4421}{4421+511} \\ R_1 &= 0,8963909165 \end{aligned} \tag{14}$$

A mesma análise pode ser feita para a determinação do P_0 e R_0 do bit 0:

$$P_0 = \frac{TP}{TP + FP}$$
$$P_0 = \frac{4738}{4738 + 511} \quad (15)$$
$$P_0 = 0,9026481235$$

$$R_0 = \frac{TP}{TP + FP}$$
$$R_0 = \frac{4738}{4738 + 123} \quad (16)$$
$$R_0 = 0,9746965645$$

Portanto, a precisão para a técnica usada para encontrar o MSB do *nonce* da assinatura é, aproximadamente, $97,3\%$ para determinar um bit como 1 e $90,3\%$ para o bit 0. Como era esperado o valor de P_1 é maior do que P_0, pois o ataque amplifica o tempo da redução modular feita somente para quando o MSB é 1 (alvo foi a linha 2 do Algoritmo 5). A revocação para os dois casos ficaram próximos de 90%, conseguindo recuperar boa parte das assinaturas atacadas.

Para implementar a segunda fase do ataque é mais interessante ter uma boa precisão do que revocação, além também de o bit MSB ser 1. Dessa forma, para aumentar a precisão para o bit 1 deve-se diminuir o número de MSB com bit 0 ser classificado como 1, o que pela Tabela 1, é atualmente de 127 ou $2,5\%$ dos *nonces* com bit 0. Esses bits são classificados erradamente porque o rastro gerado pela técnica implementada apresenta algumas limitações. Consequentemente, para melhorar os resultados seria necessário uma mudança de abordagem da parte do código atacada, o que não foi encontrado nada melhor durante esse trabalho.

6 Contramedidas e conclusões

Nesse trabalho foi estudado e implementado um ataque real para revelar o bit mais significativo do *nonce* quando executado o algoritmo ECDSA sobre curvas $\mathbb{F}_{2^m}$ na implementação corrente do *OpenSSL*. O vazamento do bit foi mostrado como ocorrendo em diversos pontos do código que executa a multiplicação por escalar.

O ataque usou da abordagem de *FLUSH+RELOAD* em conjunto com degradação de desempenho para recuperar o bit através da implementação dessas técnicas no ToolKit do *Mastik*. Para curva de Koblitz *sect163r1* foram geradas 10.000 assinaturas, enquanto um atacante operava a técnica, levando 8 horas para finalizar todas as assinaturas. Os bits preditos pelo ataque foi comparado com o esperado, obtendo $97,3\%$ de precisão para quando o bit era 1 e $90,3\%$ para o bit 0.

O trabalho mostrou que existe o vazamento na implementação e que é possível implementar uma técnica real de ataque usando o mesmo. Além disso, foi demonstrado o Algoritmo 9 com boa taxa de precisão e revocação para determinar o bit usando um rastro encontrado pela técnica *FLUSH+RELOAD*.

Contramedidas contra o ataque devem ser implementadas, visto que o ataque demonstrou ser prático. Uma possível contramedida é inviabilizar a abordagem de *FLUSH+RELOAD* a qual pode ser feita modificando a arquitetura x86 para restringir o acesso à operação *cflush*. Entretanto, essa técnica não é muito prática, visto que a arquitetura deveria ser revista como um todo. Outra técnica seria não prover compartilhamento de páginas na

memória, mas também parece impraticável. Diminuir a precisão na medição do *clock* seria uma técnica de mitigação que restringiria muito a colocação do ataque descrito na prática, sem solucionar completamente. Uma solução mais robusta e real seria eliminar o vazamento do algoritmo modificando a implementação para o primeiro bit.

A continuação desse trabalho deverá utilizar a abordagem modificada de Bleichenbacher [Aranha et al. 2014] para conseguir recuperar a chave privada da vítima. Como esse método depende de um número grande de assinaturas, experimentos deverão ser realizados para investigar métodos que aumentem a eficiência do ataque que recupera o bit do *nonce* através da diminuição do tempo de execução do mesmo. Esse tempo longo deve-se, principalmente, a degradação de desempenho utilizada para amplificar a qualidade da técnica de *FLUSH+RELOAD*, o que restringe a praticidade do ataque e inviabiliza o ataque de Bleichenbacher para curvas de tamanhos reais. Uma possível abordagem seria utilizar a degradação apenas para os bits iniciais, visto que o alvo é o primeiro bit do *nonce*. Nessa direção experimentos deverão ser conduzidos para verificar o impacto nos resultados.

Referências

[Agnew et al. 1993] Agnew, G., Mullin, R., and Vanstone, S. (1993). An implementation of elliptic curve cryptosystems over f2155. *IEEE journal on selected areas in communications*, 11(5).

[Allan et al. 2016] Allan, T., Brumley, B. B., Falkner, K., van de Pol, J., and Yarom, Y. (2016). Amplifying side channels through performance degradation. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 422–435, New York, NY, USA. ACM.

[Aranha et al. 2010] Aranha, D., Dahab, R., López, J., and Oliveira, L. (2010). Efficient implementation of elliptic curve cryptography in wireless sensors. *Advances in Mathematics of Communications*, 4(2):169–187.

[Aranha et al. 2014] Aranha, D., Fouque, P., Gérard, B., Kammerer, J., Tibouchi, M., and Zapalowicz, J. (2014). Glv/gls decomposition, power analysis, and attacks on ecdsa signatures with single-bit nonce bias. *Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014*, 8873:262–281.

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

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

[Johnson et al. 2001] Johnson, D., Menezes, A., and Vanstone, S. (2001). The elliptic curve digital signature algorithm (ecdsa). *International Journal of Information Security*, 1(1):36–63.

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

[López and Dahab 1999] López, J. and Dahab, R. (1999). *Fast Multiplication on Elliptic Curves Over GF(2m) without precomputation*, pages 316–327. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Miller 1986] Miller, V. S. (1986). *Use of Elliptic Curves in Cryptography*, pages 417–426. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Montgomery 1987] Montgomery, P. (1987). Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264.

[Rivain 2011] Rivain, M. (2011). Fast and regular algorithms for scalar multiplication over elliptic curves. *IACR Cryptology ePrint Archive*, 2011:338.

[The OpenSSL Project 2003] The OpenSSL Project (2003). OpenSSL: The open source toolkit for SSL/TLS. `www.openssl.org`.

[Vanstone 1992] Vanstone, S. (1992). Responses to nist's proposal. *Commun ACM*, 35.

[Walter 2004] Walter, C. D. (2004). *Longer Keys May Facilitate Side Channel Attacks*, pages 42–57. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Yarom 2016] Yarom, Y. (2016). Mastik: A Micro-Architectural Side-Channel Toolkit. `https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf`.

[Yarom and Benger 2014] Yarom, Y. and Benger, N. (2014). Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack. *Cryptology ePrint Archive*, 140.

[Yarom and Falkner 2014] Yarom, Y. and Falkner, K. (2014). FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In Fu, K. and Jung, J., editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 719–732. USENIX Association.