Extensão do conjunto de instruções para implementação segura de X25519

Antonio C. Guimarães, Edson Borin, Diego F. Aranha

Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Av. Albert Einstein, 1251 – 13.083-852 – Campinas – SP – Brasil

antonio.junior@students.ic.unicamp.br, {edson, dfaranha}@ic.unicamp.br

Abstract. *This papers presents the results of introducing a new conditional exchange instruction in the x86 architecture and its application in a side-channel resistant implementation of the X25519 protocol based on Diffie-Hellman. Building upon an implementation included in SUPERCOP, we developed 4 different versions of the algorithm: 1 using the new instruction and 3 restricted to the instructions already available. The MARSSx86 simulator was used for prototyping and validation of security and efficiency. Experimental results illustrate the vulnerability in an insecure implementation and the improvements in security and 1.4% in performance after the new instruction was introduced.*

Resumo. *Este artigo descreve os resultados da introdução de uma nova instrução de troca condicional para a arquitetura x86 e sua utilização na implementação resistente a ataques de canal lateral do protocolo X25519 baseado em Diffie-Hellman. Baseando-se na implementação presente no SUPERCOP, foram desenvolvidas 4 versões do algoritmo, sendo uma com a nova instrução e outras 3 utilizando apenas instruções já existentes. Para a prototipação da instrução e validação de segurança e desempenho foi utilizado o simulador MARSSx86. Os resultados experimentais permitiram demonstrar a vulnerabilidade em uma versão insegura, assim como o ganho em segurança e de 1.4% em desempenho com a introdução da nova instrução.*

1. Introdução

A preocupação com ataques de canal lateral em dispositivos embarcados cresce à medida que a Internet das Coisas se populariza. Devido à baixa capacidade computacional, tais dispositivos tiveram sua construção focada especialmente na eficiência, muitas vezes em detrimento da proteção contra vazamento de informações dos dados sensíveis ali processados. Dentre tais dados, os algoritmos criptográficos utilizados nesses dispositivos também estão vulneráveis, sendo, para diversas implementações comuns, como o AES e o RSA, a obtenção de informação secreta por meio de ataques de canal lateral amplamente descrita pela literatura [1-4].

De modo geral, as soluções em *software*, até agora propostas e utilizadas, encontram limitações na falta de suporte do *hardware* para execução segura. Ainda que obtenham sucesso em coibir alguns tipos de vazamento, muitas vezes acabam por ser incompletas, ao permitirem o vazamento por outros meios, ou até mesmo criar novas vulnerabilidades [15].

Neste artigo foi adotada uma abordagem baseada no projeto conjunto de

software e *hardware* com o objetivo de mitigar ataques de canal lateral de tempo. O foco dado foi ao combate de ataques de tempo em algoritmos de curvas elípticas, especificamente a função X25519. Com tal objetivo, foi desenvolvida uma nova instrução para a arquitetura x86, permitindo uma implementação mais segura e eficiente do algoritmo.

Com a escolha desta arquitetura, dado o foco em dispositivos embarcados, a validação de segurança e desempenho da nova instrução foi realizada a partir da simulação de um sistema similar ao Intel Galileo: um processador com núcleo único, com execução de instruções em ordem, com 16KB de *cache* L1 e 256MB de memória RAM. É preciso notar que a escolha da arquitetura impacta diretamente nos resultados da introdução da nova instrução. Assim, é possível que ao aplicar tal abordagem em outras arquiteturas, que não a x86, não se chegue as mesmas conclusões que este artigo.

Apesar de ter sua construção inspirada por um algoritmo criptográfico específico, a nova instrução proposta foi definida de maneira genérica o bastante para ter seu uso facilmente generalizado, podendo ser utilizada em outros algoritmos e aplicações, até mesmo fora do contexto criptográfico ou de segurança.

2. Fundamentação Teórica

2.1. Ataques de Canal Lateral

São considerados ataques de canal lateral aqueles que se baseiam em dados físicos da máquina que executa o criptossistema para obter acesso às informações secretas. Dentre os dados físicos mais utilizados nesse tipo de ataque estão o tempo de execução, o consumo de energia e o espectro eletromagnético. Para a maior parte dos casos, o atacante precisa ter acesso físico ao dispositivo, mas a principal exceção são ataques de tempo, que podem ser executados através de uma conexão remota com o dispositivo por meio, por exemplo, da Internet. Existem ainda ataques que exigem acesso não-privilegiado ao sistema que executa o algoritmo criptográfico, como é o caso do ataque por colisão de *cache* ao AES [3].

Apesar do foco deste artigo estar em dispositivos embarcados, processadores de outras classes, como estações de trabalho e servidores, também estão suscetíveis a ataques de canal lateral. Ainda que suas informações físicas sejam influenciadas por outros fatores, como, por exemplo, outros algoritmos executando simultaneamente e a variação da velocidade de conexão, é possível filtrar tais ruídos monitorando a máquina por tempo suficiente [14].

De modo geral, mesmo que as informações fornecidas a respeito da informação secreta de algoritmos criptográficos pelos canais laterais não sejam suficientes para determinar com exatidão a chave, elas podem fornecer informação suficiente para reduzir o domínio da chave secreta, reduzindo o número de combinações possíveis e, desse modo, possibilitando que um ataque por força bruta seja bem-sucedido.

A abordagem mais adotada no combate aos ataques de canal lateral está em tornar as informações físicas emitidas pela máquina hospedeira do sistema criptográfico independentes das informações secretas do algoritmo.

2.2. Ataques de Tempo

Bastante estudados, a principal característica que coloca os ataques de tempo entre os

mais utilizados é a possibilidade de serem executados de modo completamente remoto [8], utilizando-se apenas das vias tradicionais de acesso ao dispositivo alvo, por exemplo: fazendo requisições que envolvam procedimentos criptográficos e medindo o tempo de resposta.

Em geral, a maior fonte de variação do tempo de execução dos algoritmos em função da entrada é ocasionada por mudanças no fluxo de execução do algoritmo. Felizmente, para algoritmos criptográficos, este tipo de vulnerabilidade já foi amplamente estudada [9] e diversas técnicas estão disponíveis para se evitar esta falha.

Outros fatores também ocasionam tal variação no tempo de execução e, por serem mais complexos e dependentes da arquitetura, não tem soluções bem definidas. Dentre eles, vale destacar as falhas na predição de desvios [10] e as variações no tempo de carregamento e armazenamento dos dados devido à hierarquia de memória das máquinas [3].

2.3. A Função X25519

Apresentada em 2006 por Bernstein [5], a X25519 é uma função para a implementação do algoritmo de Diffie-Hellman utilizando curvas elípticas. Ela foi apresentada por seu autor como um novo recorde de velocidade para implementação segura de tal algoritmo. A Figura 1 demonstra o fluxo de tal implementação.

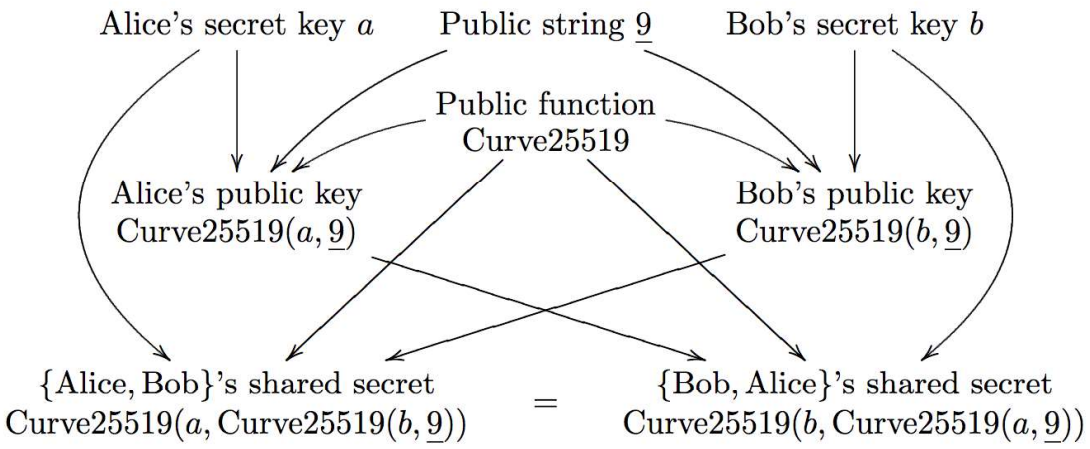

Figura 1. Fluxo de obtenção das chaves secretas compartilhadas no algoritmo de Diffie-Hellman utilizando a X25519. Retirado da referência [5].

A X25519 pode ser basicamente definida pela multiplicação de pontos na curva elíptica $E: y^2 = x^3 + 486662x^2 + x$, definida utilizando o primo $p = 2^{255} - 19$.

Além da velocidade, uma característica bastante atrativa da X25519 é que sua execução se dá em tempo constante, isto é, independente dos dados de entrada, o que implica, a princípio, na sua resistência contra ataques de canal lateral baseados no tempo.

Apesar de tal resistência, este artigo foca em fornecer um método alternativo, utilizando o suporte do *hardware*, para garantir esta proteção e ainda permitir certo ganho de desempenho. A principal razão que dá relevância a alternativa aqui proposta está na questão de que os artifícios utilizados pela implementação original para tornar a execução resistente a ataques de tempo acabam por criar novas vulnerabilidades e fontes

de vazamento de dados, como mostrado por Erick Nascimento, Lukasz Chmielewski, David Oswald e Peter Schwabe [15].

3. Metodologia

O ciclo básico de estudo adotado para este artigo foi:

1. Incluir no simulador de sistema MARSSx86 uma nova instrução, voltada a permitir a execução segura da X25519.
2. Instrumentar o simulador, de modo a permitir a obtenção de novas métricas que facilitem a detecção dos canais laterais e o entendimento dos dados já fornecidos.
3. Partindo da implementação original da X25519, presente no SUPERCOP [6], gerar 3 versões modificadas de tal implementação.
4. Executar as 4 versões da X25519 no simulador, analisar e tratar os dados de cada uma delas.

Cada uma das etapas descritas anteriormente será detalhada nas subseções seguintes.

3.1. Implementações da Função X25519

Foi tomada como base a implementação `ref10` do X25519 presente no SUPERCOP[6]. Nela, o ponto de vulnerabilidade escolhido para estudo foi a função *fe_cswap*, que tem por objetivo fazer ou não a troca de valores entre dois vetores a depender de um *bit* provido pela chave. Tal função, na implementação, é utilizada para substituir o condicional do algoritmo da cadeia de Montgomery [13]. Seu código é mostrado na Figura 2 de maneira simplificada, ocultando trechos não pertinentes à lógica do algoritmo, como declaração de variáveis e iteração sobre vetores.

```
void fe_cswap(fe f, fe g, unsigned int b)
{
  crypto_int32 f0 = f;
  crypto_int32 g0 = g;
  crypto_int32 x0 = f0 ^ g0;
  b = -b;
  x0 &= b;
  f = f0 ^ x0;
  g = g0 ^ x0;
}
```

Figura 2. Código, simplificado, da função fe_cswap do SUPERCOP [6].

As outras 3 versões implementadas a partir da original consistiram basicamente de alterações na maneira como esta função efetua a troca. Na versão original pode-se notar que a execução deve ocorrer em tempo constante, o que será demonstrado neste artigo. Nela, entretanto, é feita a utilização de variáveis auxiliares, o que ocasiona perda de desempenho, além de possível inserção de novas vulnerabilidades [15].

A primeira versão modificada é a versão com uma troca ingênua. Utilizando-se de um condicional simples para efetuar a troca, tal versão é absolutamente insegura e sua vulnerabilidade seria facilmente detectada por uma análise do fluxo de execução. Seu código é mostrado na Figura 3.


```
void ns_select(fe p, fe r, unsigned int b)
{
  crypto_int32 aux;
  if(b == 1){
    aux = p;
    p = r;
    r = aux;
  }
}
```

Figura 3. Código da função de troca insegura.

As versões anteriores foram implementadas utilizando a linguagem C, entretanto, uma vez que a presente pesquisa se dá especificamente para a arquitetura Intel x86, foram também implementadas duas trocas condicionais em *assembly*, de modo a otimizar este trecho de código e abrir espaço a introdução de uma nova instrução.

A segunda versão modificada utiliza-se da instrução *cmov* para efetuar a troca condicional. Seu código, simplificado, é mostrado na Figura 4. O sufixo *e* indica que a condição a ser verificada é a de igualdade.

```
cmpl $1, %edi
.irp i, 0,1,2,3,4,5,6,7,8,9
        mov 4*\i(%edx), %eax
        mov 4*\i(%ecx), %ebx
        mov %eax,-4(%ebp)
        cmove %ebx,%eax
        cmove -4(%ebp),%ebx
        mov %eax, 4*\i(%edx)
        mov %ebx, 4*\i(%ecx)
.endr
```

Figura 4. Código, simplificado, da função de troca utilizando a instrução cmove.

É preciso notar que em todas as implementações anteriores, inclusive na versão em *assembly*, foi necessário uma variável temporária para se efetuar a troca. No caso de uma eventual implementação em uma arquitetura de 64 *bits*, tal variável poderia ser alocada em um registrador. Porém, na arquitetura x86, por falta de registradores, ela acaba por ficar na memória, ocasionando perdas de desempenho.

Finalmente, a última versão implementada consiste na troca condicional através de uma nova instrução de troca condicional, a *cxch.* O funcionamento dessa nova instrução consiste basicamente em trocar os valores dos registradores alvos se a condição definida pelo sufixo for satisfeita pelas *flags* do processador. Tal verificação é similar à realizada pela instrução *cmov*. Este comportamento difere da já existente Compare-And-Swap (CAS) no ponto em que esta realiza a troca com base apenas na igualdade de seus próprios operandos.

O código da implementação com a nova instrução, simplificado, é mostrado na Figura 5. O sufixo *e* indica que a condição a ser verificada é a de igualdade.

Ao contrário das demais implementações, pode-se ver, além de uma redução no número de instruções, que não foi necessário o uso de qualquer variável auxiliar em memória. Desse modo tem-se um provável ganho de desempenho, que será também demonstrado neste artigo.


```
cmpl $1, %edi
.irp i, 0,1,2,3,4,5,6,7,8,9
        mov 4*\i(%edx), %eax
        mov 4*\i(%ecx), %ebx
        cxche %eax, %ebx
        mov %eax, 4*\i(%edx)
        mov %ebx, 4*\i(%ecx)
.endr
```

Figura 5. Código, simplificado, da função de troca utilizando uma nova instrução cxche.

3.2. O Simulador de Sistema MARSSx86

Para se efetuar a prototipação da nova instrução, assim como obter e comparar dados das execuções de cada uma das versões anteriores foi utilizado o simulador de sistema MARSSx86 [7].

Apresentado em 2011, o MARSSx86 é um simulador micro arquitetural e simulador de sistema baseado no QEMU [11] e no PTLSim [12]. Bastante utilizado em pesquisas na área de arquitetura, ele é dotado de alta acurácia em suas simulações e provê um grande número de dados que possibilitam uma análise profunda do comportamento dos algoritmos nele executados.

Além disso, a facilidade de acesso e modificação de seu código aliada à grande precisão que provê garantem a possibilidade de expandir seu conjunto de instruções e de inserir novos componentes de *hardware* com boa precisão em relação a um *hardware* real.

3.3. A prototipação de Instruções

Como citado nas subseções anteriores, a nova instrução inserida no MARSSx86 foi a troca condicional. O pseudocódigo que descreve sua implementação é mostrado na Figura 6. Nele, a operação *Seleciona* retorna o primeiro parâmetro a ela passado, caso o valor do terceiro parâmetro, a variável *Troca,* seja 0. Caso contrário, o segundo parâmetro é retornado. Já a operação *Verifica_Condicao* verifica se a condição definida por seu primeiro parâmetro é satisfeita pelas *flags* passadas por seu segundo parâmetro.

```
Cxch (A, B, Cond, Flags)
  bool Troca = Verifica_Condicao(cond, Flags)
  Temp = A
  A = Seleciona(Temp, B, Troca)
  B = Seleciona(B, Temp, Troca)
```

Figura 6. Pseudocódigo da instrução *cxch*.

Na implementação realizada no simulador, tal pseudocódigo gera 4 micro operações da arquitetura Intel. A primeira, chamada *OP_collcc*, é a responsável por possibilitar a operação *Verifica_Condicao*. Enquanto as outras 3, de mesmo tipo e chamadas *OP_sel*, realizam a operação *Seleciona.* Apesar de ser uma atribuição simples, a atribuição à variável *Temp* também é realizada pela micro operação *OP_sel*.

Foram utilizadas apenas as micro operações já existentes no processador e que

são utilizadas na implementação das demais instruções da arquitetura. Assim, o tempo de execução dessa nova instrução comparado às demais é bastante realista, tendo uma proporção bem próxima a que ocorreria num *hardware* real. As variáveis temporárias *Troca* e *Temp* são registradores temporários internos ao conjunto de instruções.

O simulador é capaz de simular a execução as micro operações em tempo constante, porém, no caso de uma implementação em *hardware* real seria necessário que essa propriedade fosse garantida para que se obtivesse, de fato, uma execução resistente a ataques de canal lateral por tempo.

3.4. O processo de Simulação

Para possibilitar uma melhor análise dos dados foram medidas estatísticas não só para a função criptográfica toda, mas também apenas para os pontos de interesse na execução, no caso, a função que efetua a troca. Apesar das estatísticas se referirem ao código isolado, o sistema operacional é capaz de influir de diversas formas na execução, assim, para se garantir a confiabilidade dos resultados cada simulação foi executada 10 vezes, a fim de se obter um desvio padrão de cada dado.

No total, o processo de simulação fornece cerca de 300 dados sobre a execução realizada. Dentre eles estatísticas sobre o número de ciclos, predições de desvio, acessos à *cache* e à memória, tipos de instruções, tipos de micro operações, dentre outras. Para apresentação e análise dos resultados na seção seguinte, serão apresentados apenas os dados mais relevantes.

4. Resultados Experimentais

O primeiro objetivo do processo de simulação foi o de demonstrar a capacidade da ferramenta de detectar o vazamento de informações por canais laterais. Isso foi feito através da exposição da vulnerabilidade da versão insegura a ataques de tempo. Neste processo também foi demonstrado a segurança das demais versões. Para alcançar tal objetivo foram executados dois testes em ambiente simulado.

A função de troca, descrita na subseção 3.1, recebe como parâmetro uma variável binária *b* e a utiliza para definir se efetuará ou não a troca. Tal variável é definida pela operação de ou exclusivo (*xor*) entre dois *bits* sequenciais da chave. Tais *bits* são percorridos sequencialmente do mais significativo para o menos significativo. Assim, temos que quanto mais transições de *bits* a chave tiver, mais trocas o algoritmo fará.

Com isso, o primeiro teste consistiu em executar as 4 versões para 3 chaves privadas diferentes, com o número de transições de *bits* crescente em cada uma delas. A chave pública permaneceu inalterada. São elas:

- K1: Uma chave composta por 128 *bits* zeros seguidos de 128 *bits* uns.
- KX: Uma chave fixa gerada aleatoriamente. Possui 120 transições aleatoriamente distribuídas.
- K256: Uma chave comporta por 128 pares de *bits* “01”.

O foco está nos ataques de tempo e é possível detectar variação diretamente através do número total de ciclos executados. O gráfico da Figura 7 mostra o número total de ciclos para uma execução completa do algoritmo, enquanto o gráfico da Figura

8 mostra o número de ciclos gastos apenas pelas funções de troca, também durante uma execução completa do algoritmo.

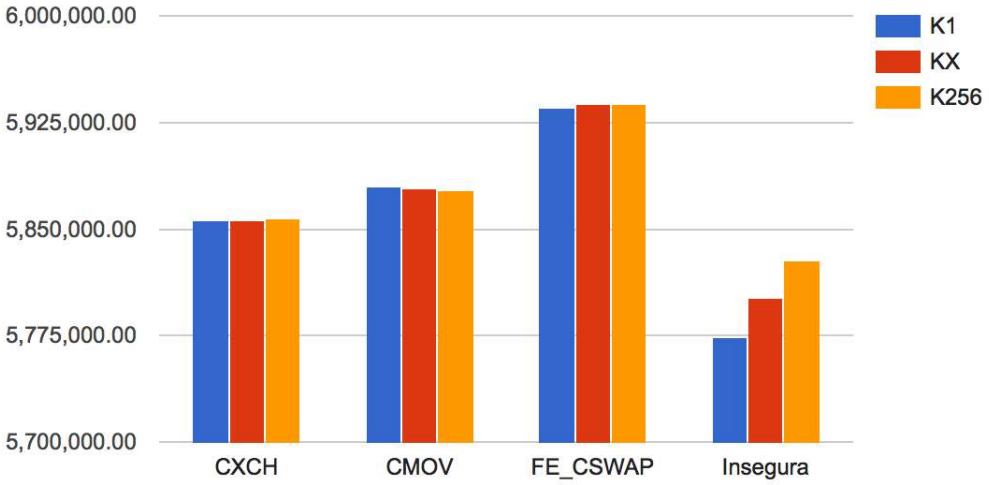

Figura 7. Gráfico do número de ciclos da execução do algoritmo completo.

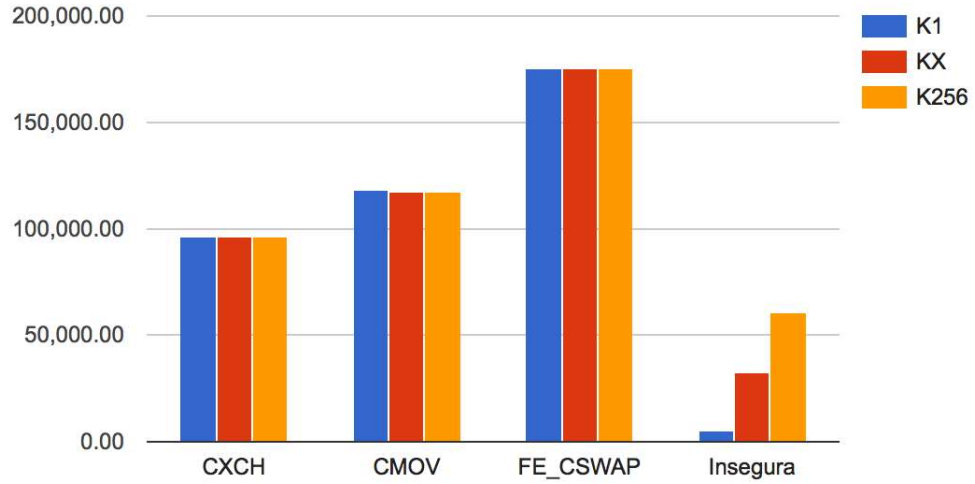

Figura 8. Gráfico do número de ciclos da execução da função de troca.

Analisando os dados do gráfico da Figura 7, é possível notar com clareza a vulnerabilidade da versão insegura. O tempo de execução dela varia diretamente de acordo com o número de transições da chave. Tal fato decorre de suas grandes variações no fluxo de execução da função de troca, o que garante a ela um desempenho bastante superior às versões seguras, mas que permitiria a um atacante adquirir facilmente a informação sobre o número de transições da chave. As demais versões, ao menos em relação a ataques baseados em tempo, neste teste, se mostram resistentes. O gráfico da Figura 8 permite confirmar a fonte da vulnerabilidade da versão insegura.

O segundo teste realizado foi a execução das 4 versões para duas chaves privadas com o mesmo número de transições, porém em uma delas, kB, as transições estão dispostas de maneira padronizada, facilmente predizíveis, enquanto na outra, kA, estão dispostas aleatoriamente. O gráfico da Figura 9 exibe a razão entre o número de ciclos das execuções das funções de troca com cada uma das chaves.

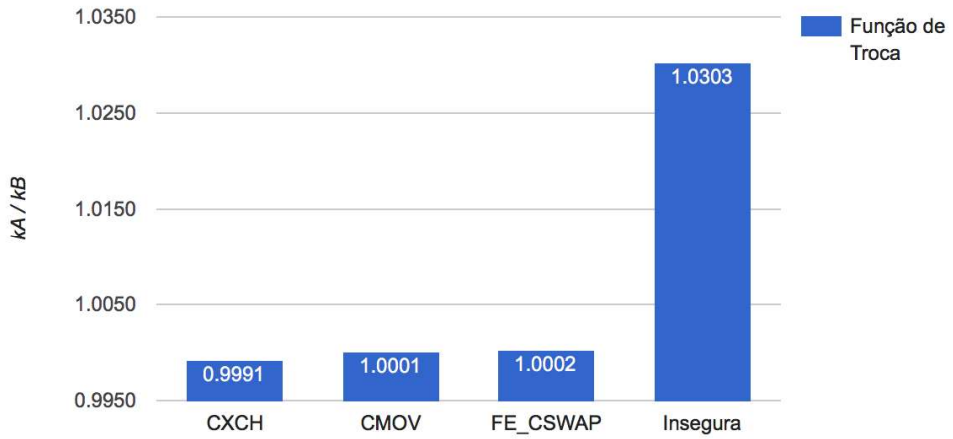

Figura 9. Razão entre o número de ciclos das execuções da funções de troca com as chaves kA e kB.

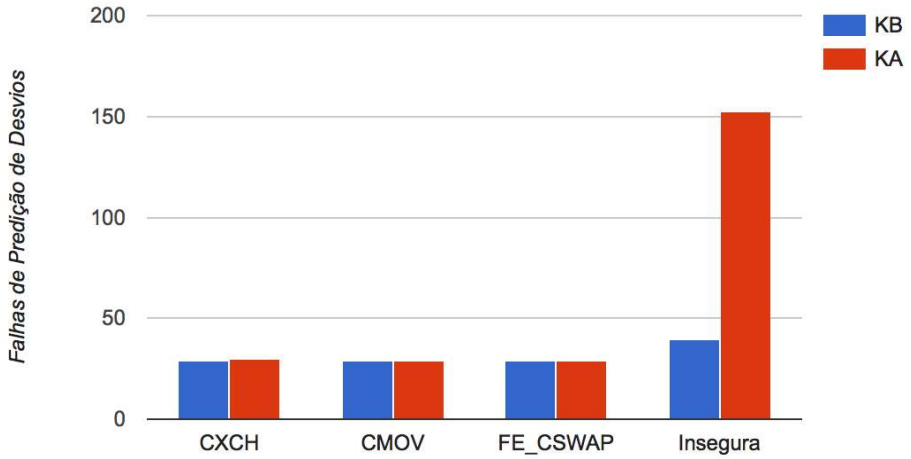

Figura 10. Gráfico do número de falhas de predição de desvios da execução do algoritmo completo.

O gráfico da Figura 9 permite notar que enquanto as versões seguras têm variação na ordem de 0.01% no número de ciclos, decorrente da influência do sistema operacional, a versão insegura tem uma variação de mais de 3% no número de ciclos, mesmo com as duas chaves tendo o mesmo número de transições. Esta variação se dá devido às falhas de predição de desvio no condicional da função de troca. Quando as transições seguem um padrão, o preditor consegue determiná-las corretamente e o tempo de execução cai. Já quando elas são aleatórias, ele passa a errar mais e, consequentemente, o tempo de execução aumenta. O gráfico da Figura 10 mostra o número de falhas de predição de desvio ocorridas durante toda a execução de cada uma das versões, para as duas chaves kA e kB. Seus resultados corroboram a explicação anterior.

Após este último teste, tem-se que a versão insegura revela, sob um ataque de tempo, não só a informação referente ao número de transições da chave, mas também à disposição de tais transições. Assim, demonstra-se a razoável capacidade de detecção de vazamento de informações por canais laterais da ferramenta utilizada, bem como o nível de segurança das versões implementadas.

O segundo objetivo da simulação está em demonstrar o ganho de desempenho da versão com a nova instrução *cxch* em relação às demais versões seguras. Considerando apenas as funções de troca, em relação à *fe_cswap*, presente na implementação original, o ganho de desempenho provido pelo uso da nova instrução chega a 45%, como é mostrado no gráfico da Figura 8. Enquanto para o algoritmo todo, ele fica em torno de 1.4%, também em relação à versão original. O gráfico da Figura 11 ajuda a entender o ganho alcançado. Ele representa dados de acesso à *cache*, número de instruções e micro operações executadas por cada uma das versões das funções de troca. Através do gráfico, é possível notar, por exemplo, que a versão original faz o dobro de acesso à *cache* de dados do que a versão com a nova instrução. Para facilitar a visualização, os dados são normalizados com relação aos dados medidos na versão com a instrução *cxch*.

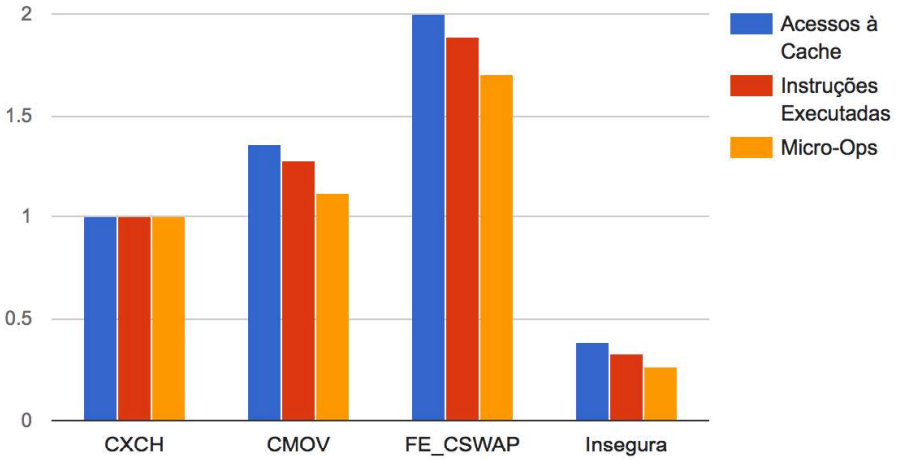

Figura 11. Gráfico de acessos à *cache*, número de instruções e micro operações executadas pelas funções de troca normalizados com relação aos dados medidos na versão com a instrução CXCH.

5. Conclusões

Neste artigo foram apresentados os resultados da prototipação de uma nova instrução de troca condicional para a arquitetura x86 e sua utilização na implementação resistente a ataques de canal lateral do algoritmo de Diffie-Hellman com a X25519. Através das simulações realizadas, foi possível demonstrar a capacidade de detecção de vazamento de informações por canais laterais do método utilizado, assim como demonstrar a segurança e comparar, em termos de eficiência, diferentes implementações da X25519.

A proposta da nova instrução de troca condicional se mostra bastante razoável, uma vez que tal instrução possibilitou um ganho de desempenho de 45% na função de troca da X25519, em relação a implementação original. Além de ajudar a evitar, por fazer todo o procedimento de troca internamente, que se encontrem outras vulnerabilidades naquele ponto. É preciso ressaltar, entretanto, que tal abordagem tem como desvantagens a necessidade de se fazer modificações no *hardware* e o fato de ter sido desenvolvida para uma arquitetura específica.

Pretende-se, como trabalho futuro, a replicação do método de estudo para outros algoritmos criptográficos, resultando na criação de novas instruções e componentes de *hardware*. Pretende-se também validar todo procedimento com a prototipação das instruções e execução dos algoritmos em *hardware* real, através da utilização de

FPGAs. Outros tipos de vazamento de dados por canais laterais, como, por exemplo, o consumo de energia, também serão estudados.

6. Agradecimentos

À Intel Semicondutores do Brasil Ltda e à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processo nº 2014/50704-7, pelo financiamento e apoio à pesquisa.

Referências

[1] Kocher, Paul C. "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems." *Annual International Cryptology Conference*. Springer Berlin Heidelberg, 1996.

[2] Bernstein, Daniel J. "Cache-timing attacks on AES." (2005).

[3] Tromer, Eran, Dag Arne Osvik, and Adi Shamir. "Efficient cache attacks on AES, and countermeasures." *Journal of Cryptology* 23.1 (2010): 37-71.

[4] Brumley, David, and Dan Boneh. "Remote timing attacks are practical." *Computer Networks* 48.5 (2005): 701-716.

[5] Bernstein, Daniel J. "Curve25519: new Diffie-Hellman speed records." *International Workshop on Public Key Cryptography*. Springer Berlin Heidelberg, 2006.

[6] Bernstein, Daniel J. "Supercop: System for unified performance evaluation related to cryptographic operations and primitives." (2009).

[7] Patel, Avadh, Furat Afram, and Kanad Ghose. "Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors." *1st International Qemu Users' Forum*. 2011.

[8] Brumley, David, and Dan Boneh. "Remote timing attacks are practical." *Computer Networks* 48.5 (2005): 701-716.

[9] Molnar, David, et al. "The program counter security model: Automatic detection and removal of control-flow side channel attacks." *International Conference on Information Security and Cryptology*. Springer Berlin Heidelberg, 2005.

[10] Acıiçmez, Onur, Çetin Kaya Koç, and Jean-Pierre Seifert. "Predicting secret keys via branch prediction." *Cryptographers' Track at the RSA Conference*. Springer Berlin Heidelberg, 2007.

[11] Bellard, Fabrice. "QEMU, a fast and portable dynamic translator." *USENIX Annual Technical Conference, FREENIX Track*. 2005.

[12] Yourst, Matt T. "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator." *2007 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 2007.

[13] Montgomery, Peter L. "Speeding the Pollard and elliptic curve methods of factorization." *Mathematics of computation* 48.177 (1987): 243-264.

[14] Le, Thanh-Ha, et al. "Noise reduction in side channel attack using fourth-order cumulant." *IEEE Transactions on Information Forensics and Security* 2.4 (2007): 710-720.

[15] Erick Nascimento, Lukasz Chmielewski, David Oswald and Peter Schwabe. "*Attacking embedded ECC implementations through cmov side channels.*" *23rd Conference on Selected Areas in Cryptography* (SAC 2016)

