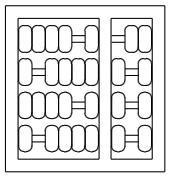

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Prototipação de Instruções para Implementação Segura de Algoritmos Criptográficos

Antonio C. Guimarães Jr. *Diego F. Aranha*

Relatório Técnico - IC-PFG-16-04 - Projeto Final de Graduação

December - 2016 - Dezembro

Prototipação de Instruções para Implementação Segura de Algoritmos Criptográficos

Antonio C. Guimarães Jr. Diego F. Aranha

Resumo

A recente popularização da Internet da Coisas fez surgir uma grande preocupação com vulnerabilidades nos mecanismos criptográficos e ataques de canal lateral em suas implementações. Neste trabalho, utilizou-se da simulação, através do simulador de sistema MARSSx86, e da prototipagem, através da tecnologia FPGA, de uma plataforma similar a tais dispositivos. Nelas, foram implementadas instruções que permitem a implementação segura, isto é, resistente a ataques de canal lateral, de alguns dos principais algoritmos criptográficos em uso. Tais ferramentas foram também utilizadas na validação, em desempenho e segurança, das diversas implementações desses algoritmos. Foram realizados experimentos com o protocolo baseado em curvas X25519 e com a cifra de bloco AES. Nos experimentos, a partir das estatísticas obtidas no simulador e na FPGA, foi possível detectar as vulnerabilidades existentes nas implementações avaliadas; e também avaliar o impacto, em segurança e desempenho, da introdução das novas instruções propostas.

1 Introdução

Com a popularização da Internet das Coisas, ocorrida nos últimos anos, novos desafios surgiram nas diversas áreas da computação. Na área de segurança e criptografia computacional, em especial, surgiu a preocupação com a vulnerabilidade a ataques de canal de lateral nos dispositivos ligados a essa tecnologia. Taís dispositivos foram, assim como os algoritmos para eles projetados, historicamente construídos com foco na eficiência e desempenho computacional, uma vez que dispõem de recursos altamente limitados. Assim, diversos aspectos referentes à segurança, como, por exemplo, a proteção contra vazamento de dados por canal lateral, acabaram por serem deixados em segundo plano.

Este trabalho focou-se em efetuar o desenvolvimento conjunto de *hardware* e *software* no intuito de desenvolver contra medidas para o combate a ataques de canal lateral. Mas especificamente, a partir da identificação dos principais pontos de vulnerabilidade dos algoritmos criptográficos mais comuns, foram elaboradas novas instruções para a arquitetura x86, visando permitir a implementação segura e eficiente de tais algoritmos.

A prototipação e validação das instruções foi realizada em ambiente virtual, através da utilização de Simuladores de Sistema, bem como em *hardware* real, através da utilização da tecnologia FPGA. Foi validado não só a segurança das implementações utilizando as novas

instruções, mas também o impacto da inserção delas no desempenho geral do algoritmo. Avaliando, assim, a viabilidade de uma possível implementação real de tais instruções.

2 Ataques de Canal Lateral

Define-se como ataque de canal lateral todo aquele que se baseia em dados físicos da máquina que executa o criptossistema para obter acesso às informações secretas dele. Para diversas implementações criptográficas comuns, como o AES [7] e o RSA [13], a obtenção de informação secreta por meio de ataques de canal lateral é amplamente descrita pela literatura.

Diversos são os dados físicos utilizados para se efetuar um ataque de canal lateral. Dentre os mais comuns estão o tempo de execução do algoritmo, o padrão de consumo de energia, o campo eletromagnético emitido, os ruídos produzidos pelo sistema, entre outros. Dados internos à máquina, mas de fácil acesso, como, por exemplo, o padrão de acesso a *cache*, também são comumente utilizados na realização deste tipo de ataque. Neste projeto, tomou-se como foco os ataque de canal lateral de tempo. Tais ataque tem como particularidade, além da facilidade de execução, a capacidade de serem executados remotamente.

O método básico de combate a este tipo de ataque consiste em tornar as informações físicas emitidas pela máquina hospedeira independentes das informações secretas do criptossistema executado. De modo geral, as soluções em *software*, até agora propostas e utilizadas, encontram limitações na falta de suporte do hardware para execução segura. Assim, apesar de terem razoável efetividade, muitas das soluções acabam por ser incompletas, ao permitirem o vazamento por outros meios, ou até mesmo criar novas vulnerabilidades.

Neste projeto, novas instruções foram inseridas no conjunto de instruções da arquitetura alvo de modo que os trechos de maior vulnerabilidade dos algoritmos criptográficos pudessem ser reescritos de maneira segura. Assim, as garantias associadas a proteção contra ataques de canal lateral foram levadas para o *hardware* e, consequentemente, um projeto seguro do *hardware* de tais instruções torna-se suficiente para a mitigação do vazamento de dados por canal lateral em tais algoritmos.

3 Ambientes de Prototipação e Validação das Instruções

Apesar de haver certa abordagem teórica na análise dos algoritmos criptográficos e identificação de seus pontos de vulnerabilidade, este projeto manteve o foco na prototipação, implementação, teste e validação, em segurança e desempenho, das novas instruções propostas. Foi foco também, a coleta e análise de estatísticas das execuções dos algoritmos criptográficos visando identificar as possíveis vulnerabilidades a ataques de canal lateral. Considerando tais objetivos, dois ambientes foram utilizados.

Em um primeiro momento, o simulador de sistema MARSSx86 [12] foi utilizado como ferramenta para obtenção de estatísticas e implementação das instruções. Nele, foi imple-

mentada uma instrução de troca condicional e uma instrução para delimitar as áreas de interesse na obtenção de estatísticas. Além disso, apesar de já fornecer uma grande variedade de estatísticas, também foi necessário estendê-lo para a aquisição de novos dados que possibilitassem um melhor entendimento das estatísticas já adquiridas, a fim de se detectar os vazamentos de dados por canal lateral.

Dentre as novas métricas implementadas no simulador estão o traço de micro-operações, traço de erros de predição de desvio e traço de *cache misses*. Também foram implementados um conjunto de *scripts* e ferramentas para o processamento dos dados fornecidos pelo simulador, assim como para integrá-lo a outras ferramentas.

Posteriormente a esta primeira fase, o procedimento realizado com o simulador foi parcialmente reproduzido em um sistema computacional descrito para uma placa FPGA. Neste sistema foi utilizado o processador Altera NIOS Gen II [1], juntamente com outros componentes genéricos fornecidos pela Altera. Sendo um *hardware* real, foi possível obter um maior realismo em relação ao que seria a implementação real das instruções prototipadas. Entretanto, este ambiente forneceu um número reduzido de dados a respeito da execução, limitando as análises realizadas. Além das instruções implementadas no simulador, foram também introduzidas no processador descrito para a FPGA as instruções de movimentação condicional, cálculo de paridade de *bits* e intercalação de *bits*.

As seguintes seções descreverão detalhadamente cada uma das ferramentas e estruturas utilizadas, bem como as dificuldades encontradas na utilização de cada uma delas.

4 O Simulador MARSSx86

Desenvolvido por Avadh Patel, Furat Afram e Kanad Ghose, o MARSSx86 é um simulador micro arquitetural e de sistema para a arquitetura x86 baseado no QEMU [3] e no PTLSim [15]. Ele se apresenta como uma evolução deste último, inserindo diversas melhorias de desempenho e acurácia, além de substituir, como ferramenta de virtualização, o Xen [2] pelo QEMU.

Bastante utilizado em pesquisas na área de arquitetura, o simulador tem como principal característica a capacidade de fazer simulações com alta acurácia nos resultados ao mesmo tempo em que se mantém rápido e eficiente em sua execução. Assim como seu antecessor, o MARSS fornece uma ampla variedade de estatísticas sobre os algoritmos nele executados e permite, por ser código aberto, a fácil adição novas funcionalidades, bem como a extensão de seu conjunto de instruções.

No projeto, o simulador foi utilizado na detecção de vazamentos de dados por canais laterais em implementações de algoritmos criptográficos e na validação de contra medidas para evitá-los. Foram testadas não apenas contra medidas em *software*, mas também no *hardware* simulado, através da adição de novas instruções. A partir das estatísticas forne-

cidas pelo simulador foi possível medir o impacto em desempenho das alterações realizadas em cada implementação.

4.1 Estrutura de Simulação

A estrutura interna do MARSS pode ser basicamente dividida em dois módulos principais: O de emulação e o de simulação. O primeiro, construído sobre o QEMU, é o que permite a execução eficiente do sistema operacional e dos algoritmos nele executados. Neste módulo, não há preocupação em refletir o funcionamento ou estado interno da maquina real que está sendo emulada, mas apenas a de garantir eficiência e corretude comportamental.

O segundo módulo, construído sobre o PTLSim, é o responsável por obter as estatísticas da execução e garantir a acurácia dos dados. Ao contrário do primeiro, ele busca ser fiel ao funcionamento e estado interno da maquina real simulada, a fim de obter medidas e estatísticas mais realistas e acuradas. Aqui, a preocupação com eficiência fica em segundo plano. O código, menos otimizado, é mais claro e bem documentado, permitindo que novas funcionalidades sejam facilmente implementadas.

A Figura 1 mostra a comunicação entre esses módulos e as demais partes do sistema.

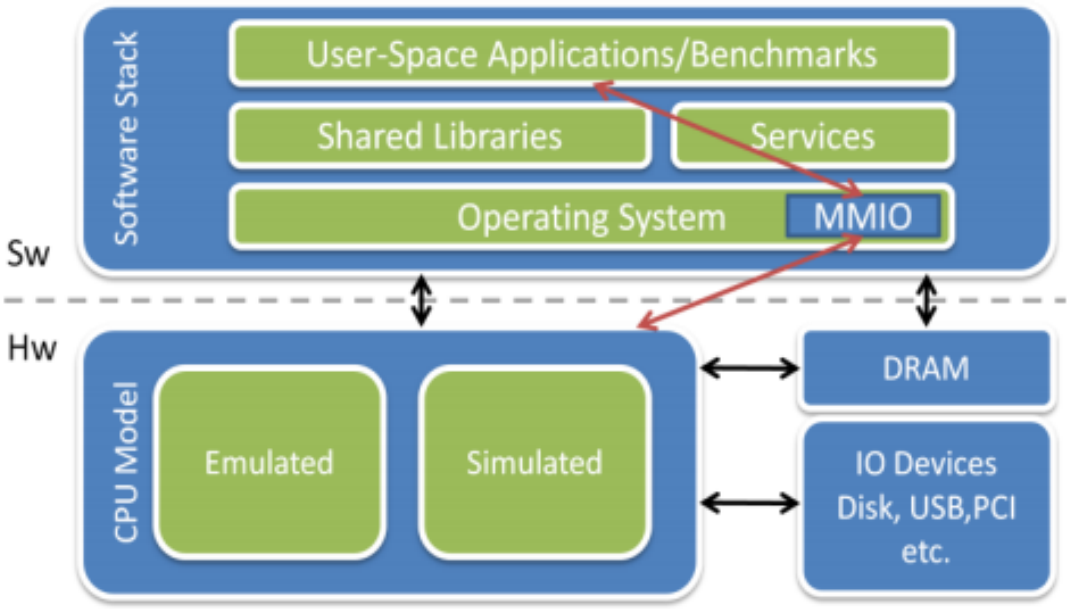

Figura 1: Diagrama de comunicação entre os módulos do MARSS. Retirado da referência [12].

Durante o uso do simulador, os dois módulos trabalham de maneira alternada. A

execução sempre começa no modo de emulação, como emulador inicializando o sistema operacional. A transição entre os modos é feita livremente pelo usuário a qualquer momento por meio de chamadas a uma biblioteca que se comunica através de MMIO, como mostrado na Figura 1. Tal transição ocorre de maneira suave, sendo percebida apenas pela mudança no desempenho da máquina. As estatísticas, no entanto, só são geradas durante o modo de simulação.

A hierarquia de memória é de responsabilidade do módulo que estiver ativo no momento e, durante o modo de simulação, as estatísticas são geradas para todos os seus níveis. Porém, apenas as *caches* são simuladas pelo módulo de simulação, enquanto a memória principal permanece sendo emulada pelo QEMU. Assim, uma alternativa para se obter um simulação mais acurada e detalhada da memória está no uso do DRAMSim2 [14], um simulador de memória que se integra ao MARSS.

O QEMU também é o responsável pelo gerenciamento de dispositivos de entrada e saída, permanecendo ativo mesmo durante o modo simulação, assim como seu terminal de comandos. Através deste também é possível enviar comandos para o modo de simulação ou para alternar ente os modos, constituindo uma alternativa às chamadas por MMIO.

4.2 Funcionamento

De modo geral, o processo de execução do MARSS é bastante simples. Um conjunto de scripts acompanha o simulador, permitindo que o usuário automatize todo o procedimento de execução e tratamento de dados através de arquivos de configuração. Não é exigido quaisquer privilégios do usuário durante sua compilação ou execução. As configurações dos scripts, assim como as do simulador, são em formato YAML [4], o que as faz bastante legíveis e claras.

4.2.1 Processo de simulação e obtenção de estatísticas

A Figura 2 representa o fluxo básico de execução do *software* e obtenção das estatísticas utilizando simulador. As seguintes subseções descrevem de maneira detalhada cada um dos passos envolvidos no procedimento, assim como suas entradas.

4.2.1.1 Compilação

O gerenciamento do processo compilação é realizado pelo SCons [9], uma ferramenta de construção de *software* que atua de maneira similar aos GNU Make e Autoconf, porém utilizando código Python. Assim como o simulador, o SCons também não exige privilégios para sua instalação ou execução.

A documentação do MARSS indica dois parâmetros a serem opcionalmente passados ao SCons durante a compilação: O parâmetro *debug* e o parâmetro *c*. O primeiro tem valor

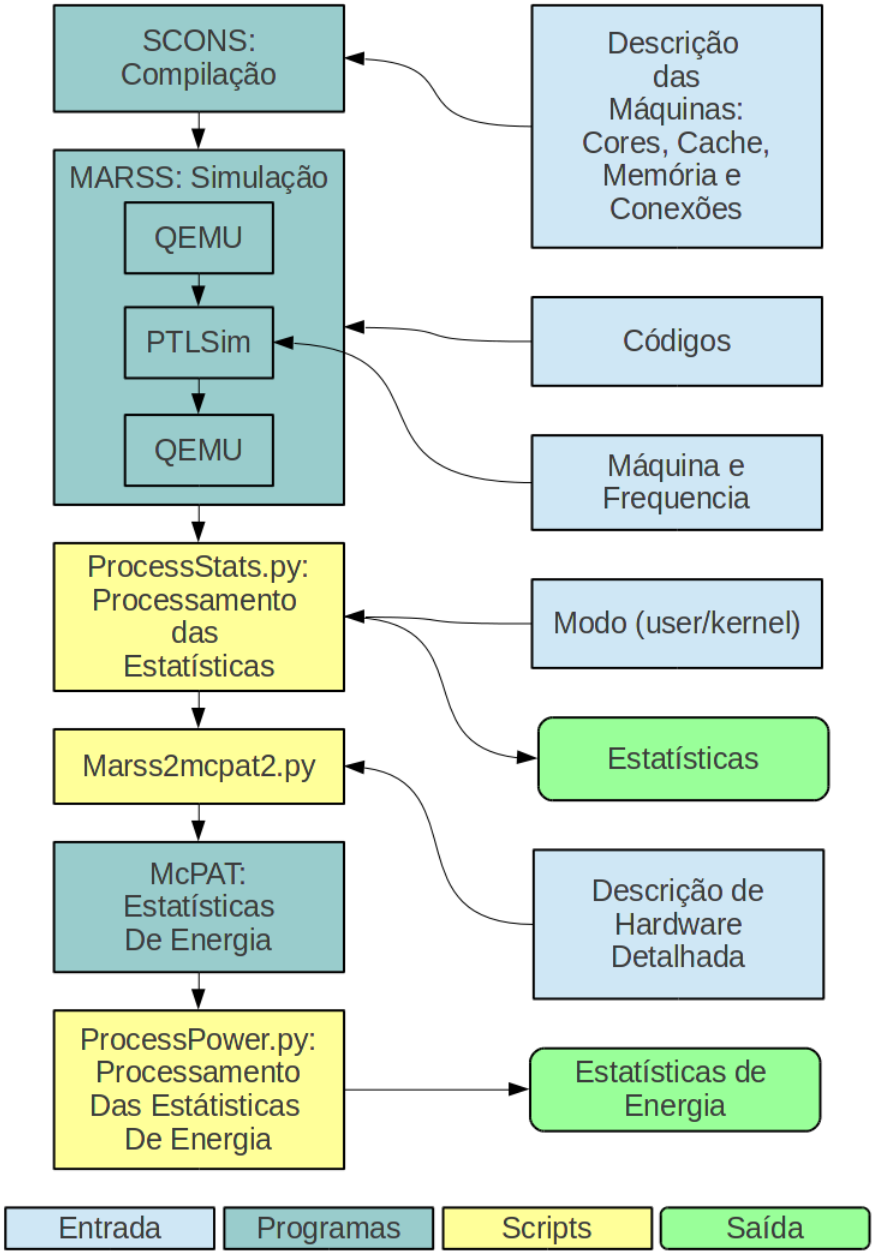

Figura 2: Diagrama do fluxo de execução do simulador e obtenção das estatísticas.

entre 0 e 2 e determina o nível de suporte a depuração que o simulador deverá fornecer, sendo que em 0 não há qualquer suporte. O segundo determina o número de núcleos que

o simulador deverá simular. Além desses parâmetros específicos, comandos genéricos do SCons também podem ser passados.

Antes de iniciar esta fase de compilação, é necessário definir as configurações de *hardware* possíveis para o simulador. Elas são lidas pelo SCons durante a compilação e impactam diretamente no código gerado do simulador. Tais configurações, assim como as demais necessárias ao simulador, são descritas na subseção 4.2.2.

4.2.1.2 Simulação

Este é principal passo do procedimento e é o único a ser de fato executado pelo MARSS. Nele, o código do algoritmo alvo, juntamente com o sistema operacional, é executado e as estatísticas de sua execução são obtidas. Durante esse passo, o usuário alterna entre os modos de emulação e simulação da seguinte forma:

1. No inicio, por questões de desempenho e compatibilidade, o simulador é iniciado no modo de emulação, com o QEMU fazendo o *boot* do sistema operacional.
2. Quando o programa alvo, ou trecho dele que seja de interesse, começa a ser executado, o simulador transita para o modo de simulação, para que as estatísticas desejadas possam ser obtidas.
3. Ao final da execução do programa, ou área do código de interesse, o simulador é colocado de volta no modo de emulação.
4. Os passos 2 e 3 podem ser repetidos livremente para múltiplos trechos ou programas. Sempre que volta ao modo de emulação, o simulador gera um arquivo de estatísticas com todos os dados capturados pela simulação até o momento.
5. Ao final dos testes o simulador é colocado no modo de emulação e desligado. O desligamento durante o modo de simulação só gera estatísticas se for feito pela interface do simulador.

Apesar deste ser o procedimento mais simples, e que foi adotado no projeto, também é possível, para execuções em um único núcleo, definir *Simpoints* baseados em *Checkpoints*, como pode ser visto na documentação do simulador, tornando o procedimento mais rápido.

As transições entre os modos são feitas através de chamadas à biblioteca do PTLSim, que se comunica com o simulador através de MMIO. Ou então através do terminal de comandos do QEMU. Em ambos os casos, os seguintes comandos de ações estão disponíveis:

- run: Inicia o modo de simulação
- stop: Termina o modo de simulação e volta ao de emulação.
- kill: Termina a execução do simulador.

- flush: Limpa os comandos enviados e encerra o modo de simulação.

Além desses, dezenas de comandos de configurações também podem ser passados, assim como escritos em um dos arquivos de configuração do simulador, o que será também detalhado na subseção 4.2.2.

4.2.1.3 Obtenção de Estatísticas

O MARSS gera uma sequência de estatísticas, em formato YAML, para cada vez em que ele saí de seu modo de simulação. Cada sequência de estatísticas contabiliza tudo o que ocorreu durante o modo desde o começo de sua execução e é dividida apenas entre estatísticas do *kernel* e do modo de usuário. Assim, quando se deseja, por exemplo, obter estatísticas de trechos diferentes de código dentro de uma mesma execução, é necessário tratar matematicamente os dados impressos.

O simulador disponibiliza um *script* que auxilia em tal tarefa, porém, para usos mais sofisticados é recomendável que se utilize um *script* próprio que trate os dados de maneira personalizada. Apesar de poder ser facilmente interpretada como texto, existem diversas bibliotecas capazes de trabalhar bem com o formato YAML.

Para o projeto foi desenvolvido um *script* em linguagem Python que interpreta o arquivo de saída, utilizando o *script* fornecido pelo simulador, soma os dados vetoriais e calcula a média e o desvio padrão de múltiplas execuções. Tais cálculos foram necessários para que se pudesse determinar a influência do sistema operacional na execução e garantir resultados mais confiáveis. Na Figura 2, este *script* é denominado *ProcessStats.py*.

Também foi necessário, para o projeto, que as estatísticas do código alvo fossem isoladas do restante do sistema. Isto não pôde ser alcançado através do tratamento dos dados e foi, então, realizado por meio de modificações do simulador. Tais modificações serão detalhadas na Seção 4.4.

4.2.1.4 Estatísticas de Energia

Para obtenção das estatísticas de energia foi utilizado o McPAT [10], uma estrutura para modelação de tempo, área e energia de componentes da arquitetura de um computador. Ele recebe como entrada um arquivo contendo uma descrição detalhada dos componentes da máquina simulada, juntamente com as estatísticas da simulação, geradas pelo MARSS. A união entre o resultado da simulação e a descrição da máquina é feita por um *script* denominado *marss2mcpat.py*, que é fornecido pelo simulador.

Assim como no caso das estatísticas padrões do simulador, também foi aqui necessário que se criasse um *script* para a realização do cálculo de média e desvio dos dados. O *script* original *marss2mcpat.py* também não pôde ser usado, uma vez que era limitado apenas a núcleos com execução fora de ordem, enquanto o projeto tinha por foco núcleos Atom. Assim, foi utilizada uma versão derivada.

4.2.2 Configurações do Simulador

Duas configurações principais determinam o comportamento do simulador: Os arquivos que descrevem o *hardware* a ser simulado e o arquivo que define os parâmetros da simulação.

4.2.2.1 Descrição do *Hardware*

A descrição do *hardware* é feita por um conjunto de arquivos, cada um deles descrevendo um ou mais componentes da arquitetura. Cada componente básico é definido pelos campos *base*, que indica qual será o código do simulador a ser utilizado, e *params*, que lista os parâmetros e valores de cada configuração. Já na configuração da maquina, os campos utilizados são *type*, que indica qual componente básico a ser utilizado; *insts*, que determina o número de instâncias do componente; e *options*, que permite a configuração de parâmetros adicionais.

Tais arquivos são lidos pelos *scripts* do SCons durante a compilação e impactam no código gerado para o simulador. Serão descritas a seguir cada uma das possíveis configurações de *hardware*, tomando como exemplo a configuração utilizada no projeto.

4.2.2.1.1 Núcleo

```
core:
  atom:
    base: atom
    params:
      DISPATCH_Q_SIZE: 16
      ISSUE_PER_CYCLE: 1
```

Código 1: Exemplo de configuração de núcleo *atom* no MARSS.

O núcleo possui duas bases disponíveis: A base *atom*, que constitui um núcleo com execução em ordem, e a base *ooo*, que constitui um núcleo com execução fora de ordem. Para a base *atom*, os parâmetros disponíveis são o *dispatch_q_size* e o *issue_per_cycle*, que determina quantas instruções são expedidas por ciclo. Já a base *ooo* possui os parâmetros *commit_width* e *issue_width*. O Código 1 mostra um exemplo desta configuração.

4.2.2.1.2 *Cache*

Para a *cache* L1, as bases disponíveis são: *wb_cache*, indicando o uso de *cache* do tipo *write-back*; *wt_cache*, indicando o uso de *cache write-through*; e *mesi*, indicando o uso do

```
l1_16K:
  base: wb_cache
  params:
    SIZE: 16K
    LINE_SIZE: 16 # bytes
    ASSOC: 4
    LATENCY: 2 # ns
    READ_PORTS: 2
    WRITE_PORTS: 1
```

Código 2: Exemplo de configuração de *Cache* L1 no MARSS.

protocolo de coerência de *cache* MESI. Os parâmetros disponíveis são apenas os indicados no Código 2. A configuração da *cache* L2 também é obrigatória no MARSS e ocorre de maneira similar a da *cache* L1, como é mostrado no Código 3.

```
l2_512k:
  base: wb_cache
  params:
    SIZE: 512k
    LINE_SIZE: 16 # bytes
    ASSOC: 4
    LATENCY: 1
    READ_PORTS: 2
    WRITE_PORTS: 2
```

Código 3: Exemplo de configuração de *Cache* L2 no MARSS.

4.2.2.1.3 Máquina

Uma vez descritos os módulos de *cache* e núcleo, pode-se descrever a máquina, como é mostrado pelo Código 4. Nele, para facilitar a visualização, foram ocultadas as conexões entre os componentes. Além dos parâmetros *type*, *insts* e *option*, descritos no começo desta seção, tem-se para cada máquina os parâmetros *min* e *max contexts*, que definem, respectivamente, o número mínimo e máximo de núcleos para o qual aquela máquina poderá ser utilizada. Tal número é definido em tempo de compilação, como descrito pela Subseção 4.2.1.1. O nome da configuração da máquina, que no caso é *galileo*, também é importante, pois é utilizado para identificar a máquina durante sua escolha no processo de simulação.

```
galileo:
  description: Galileo configuration
  min_contexts: 1
  max_contexts: 1
  cores:
    - type: atom
      name_prefix: atom_
      option:
          threads: 1
  caches:
    - type: l1_16K
      name_prefix: L1_D_
      insts: $NUMCORES # Per core L1 cache
    - type: l1_16K
      name_prefix: L1_I_
      insts: $NUMCORES # Per core L1 cache
    - type: l2_512k
      name_prefix: L2_
      insts: 1 # Shared L2 config
  memory:
    - type: dram_cont
      name_prefix: MEM_
      insts: 1 # Single DRAM controller
      option:
          latency: 50 # In nano seconds - 1.6Ghz
  interconnects:
    - type: p2p
      connections:
              [...]
```

Código 4: Exemplo de configuração de *Cache* L2 no MARSS.

Múltiplas máquinas e componentes podem ser definidos no mesmo arquivo, porém, após a compilação, apenas estarão disponíveis aquelas que tenham satisfeito o parâmetro *contexts*.

4.2.2.2 Parâmetros da Simulação

```
# Sample Marss simconfig file
-machine galileo

# Logging options
```

```
-logfile test.log
-loglevel 8
-startlog 0

# Stats file
-yamlstats test.stats

-bbdump dumpcode.bbdump
```

Código 5: Parâmetros da simulação utilizados no MARSS.

Com o simulador já em operação, no momento de transitar do modo de emulação para o de simulação, é preciso fornecer as configurações mostradas pelo Código 5. Ela define qual maquina será utilizada e quais os arquivos de saída para as estatísticas, *logs* e *dumps*. Outras opções, que não as mostradas no código, também podem ser fornecidas. A lista completa de comandos pode ser encontrada na documentação do simulador.

4.3 Introdução de Novas Instruções

A extensão do conjunto de instruções do MARSS, com a inserção de uma nova instrução, foi o meio escolhido no projeto para se implementar, em *hardware*, contra medidas para ataques de canal lateral. No caso, a nova instrução inserida foi uma troca condicional. A instrução foi inserida apenas no modo de simulação do MARSS, uma vez que a execução no modo de emulação não é necessária para a obtenção dos dados e estatísticas fornecidos pelo simulador.

4.3.1 Organização do Código

Neste ponto, 5 códigos pertencentes ao módulo de simulação do MARSS foram importantes para a introdução da nova instrução. O primeiro deles é o *decode-core.cpp*, responsável por iniciar a decodificação e efetuar a definição do *opcode* interno de cada instrução. É necessário que seu fluxo seja bem entendido, a fim de que seja possível implementar a nova instrução no local correto e com o devido *opcode* interno.

Os outros 4 códigos utilizados foram: *decode-x87.cpp*, *decode-fast.cpp*, *decode-complex.cpp* e *decode-sse.cpp*. Cada um deles é responsável por efetuar a decodificação de um conjunto de instruções.

Para determinar qual o arquivo de decodificação e qual o opcode interno de uma instrução, foi criado o digrama da Figura 4. Nele, foi considerado uma instrução com o *Opcode* no formato apresentado pela Figura 3. Onde *PF* representa o prefixo opcional da instrução e *0F* indica a presença do *byte* 0x0F como primeiro, ou segundo, no caso de haver prefixo,

byte da instrução. *PO* é o *opcode* primário da instrução e único campo obrigatório no formato, enquanto *SO* representa o *opcode* secundário.

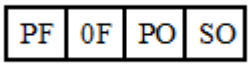

Figura 3: Modelo de *opcode* considerado

No diagrama, a variável *op* representa o *opcode* interno da instrução e a variável *modrm.reg* tem como valor os bits 5 a 3 de *SO*, da direita para esquerda. A matriz *twobyte_uses _SSE_prefix* pertence ao código *decode-core.cpp* e pode ser editada.

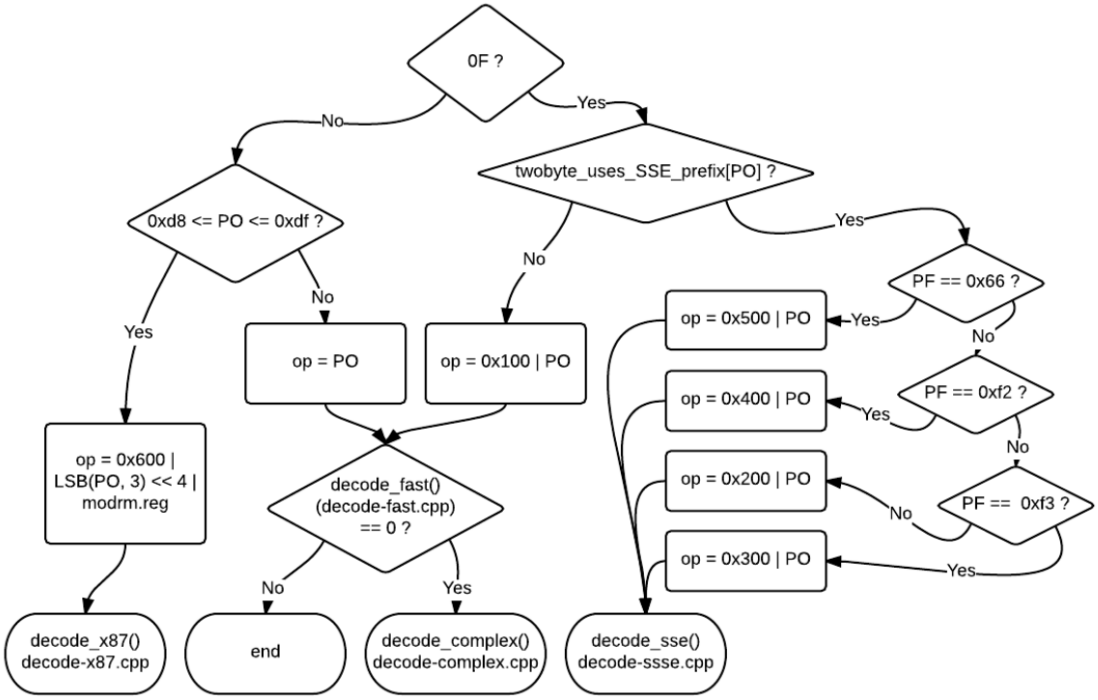

Figura 4: Diagrama para determinar o arquivo de decodificação e opcode interno de uma instrução no MARSS.

4.3.2 Implementação da Instrução

Uma vez definido o arquivo de decodificação e o *opcode* interno da nova instrução, através no diagrama da Figura 4, a nova instrução pode ser implementada. A implementação de uma instrução no MARSS consiste basicamente em decodificar os operandos e, em seguida, criar uma sequência de micro operações que serão executadas pelo sistema.

O Código 6 mostra o código da implementação da instrução de troca condicional, *cxch*, implementada durante o projeto. Seus principais pontos serão explicados nas próximas subseções. Operações simples e comuns a todas as instruções, como, por exemplo, a tradução de registradores, não serão aqui detalhadas.

```
case 0x104: {
    DECODE(gform, rd, v_mode);
    DECODE(eform, ra, v_mode);
    EndOfDecode();
    int opCond = fetch1();
    int srcreg;
    int destreg = arch_pseudo_reg_to_arch_reg[rd.reg.reg];
    int sizeshift = reginfo[rd.reg.reg].sizeshift;

    if (ra.type == OPTYPE_REG) {
      srcreg = arch_pseudo_reg_to_arch_reg[ra.reg.reg];
    } else {
      assert(ra.type == OPTYPE_MEM);
      prefixes &= ~PFX_LOCK;
      operand_load(REG_temp7, ra);
      srcreg = REG_temp7;
    }

    int condcode = bits(opCond, 0, 4);
    const CondCodeToFlagRegs& cctfr = cond_code_to_flag_regs[condcode];

    int condreg;
    if (cctfr.req2) {
      this << TransOp(OP_collcc, REG_temp0, REG_zf, REG_cf, REG_of, 3, 0, 0,
      FLAGS_DEFAULT_ALU);
      condreg = REG_temp0;
    } else {
      condreg = (cctfr.ra != REG_zero) ? cctfr.ra : cctfr.rb;
    }
    assert(condreg != REG_zero);

    TransOp transop(OP_sel, REG_temp0, REG_temp0, srcreg, condreg, sizeshift);
    transop.cond = condcode;
    this << transop;
    TransOp transop2(OP_sel, srcreg, srcreg, destreg, condreg, sizeshift);
    transop2.cond = condcode;
    this << transop2;
    TransOp transop3(OP_sel, destreg, destreg, REG_temp0, condreg, sizeshift);
```

```
39      transop3.cond = condcode;
40      this << transop3;
41      if (ra.type == OPTYPE_MEM){
42          result_store(REG_temp7, REG_zero, ra);
43      }
44      break;
45    }
```

Código 6: Instrução de troca condicional implementada no MARSS.

4.3.2.1 A decodificação dos operandos

A decodificação é feita através do macro Decode, que recebe 3 parâmetros. O primeiro indica a forma da codificação e pode conter os seguintes valores:

- gform: É um registrador
- eform: É um registrador ou um endereço de memória
- iform: É um valor imediato

O segundo parâmetro é o registrador que receberá o valor decodificado, podem ser utilizados os registradores *rd*, *ra*, *rb* e *rc*. O terceiro parâmetro diz respeito ao modo de codificação, que varia de acordo com o tamanho do dado, as seguintes opções podem ser utilizadas:

- b_mode: Byte
- v_mode: O tamanho do operando depende do prefixo.
- w_mode: Word
- d_mode: Double Word
- q_mode: Quad Word
- x_mode: Operando de 16 bytes XMM
- dq_mode: O tamanho do operando depende do prefixo REX.

No caso da *cxch* foram utilizados dois *Decodes*, o primeiro com *gform* e o segundo com *eform*, ambos com o tamanho definido pelo prefixo(*v_mode*). Desse modo a instrução deve receber como primeiro operando um registrador e como segundo operando um registrador ou endereço de memória.

4.3.2.2 Obtenção de Parâmetros Adicionais

Como o MARSS não leva o *Opcode* secundário em consideração durante o processo de decodificação da instrução, é necessário, para poder utilizá-lo, obtê-lo dentro da instrução. No caso, isto é feito através da chamada da função *fetch1()*, que obtêm um *byte* da instrução. Na *cxch*, este *byte* é utilizado para determinar a condição que definirá a ocorrência ou não da troca.

Alternativamente, caso se deseje obter uma maior aproximação com a realidade em termos de comportamento, pode-se alterar o processo de decodificação da instrução de modo a obter o *Opcode* secundário ainda nesta fase. Assim, não seria necessário fazer operações de *fetch* já dentro da instrução.

4.3.2.3 Operações de acesso à memória

Na *cxch*, logo após obtidos os operandos, é feita a verificação de tipo do operando *ra*. Isto é necessário, uma vez que, caso *ra* seja um endereço de memória, será necessário carregá-lo para um registrador antes de executar a instrução e devolvê-lo à memória após sua execução. A verificação é feita comparando o valor do parâmetro *type* de *ra* com as constantes *OPTYPE_REG* e *OPTYPE_MEM*.

O carregamento e armazenamento de dados da memória são feitos através das seguintes funções:

- operand_load(R0, ra): Carrega o valor armazenado no endereço ra para o registrador temporário R0.
- result_store(R0, R1, ra): Armazena o valor do registrador R0 no endereço ra + R1.

No caso da *cxch*, *R1* é *REG_zero*, que é um registrador com valor 0.

Sempre que uma operação de memória é executada em uma instrução é necessário adicionar um *lock* aos prefixos, indicando o acesso a memória. Isto é feito através da seguinte operação:

```
prefixes &= ~ PFX_LOCK;
```

4.3.2.4 Decodificação e Tratamentos Adicionais

Quaisquer operações adicionais necessárias à decodificação da instrução podem ser inseridas na implementação. No caso da *cxch*, a única operação adicional feita é a extração dos 4 *bits* da condição, através da função *bits*. Exemplos mais complexos de operações adicionais podem ser obtidos através da leitura de outras instruções já implementadas no simulador.

4.3.2.5 Micro Operações

O último e principal passo para a implementação da instrução é a determinação de suas micro operações. 130 micro operações estão disponíveis no simulador, elas podem ser consultadas nos arquivos *ptlhwdef.cpp*, *ptlhwdef.h* e *uopimpl.cpp*.

As micro operações seguem um padrão de sintaxe com os seguintes argumentos:

- Micro Operação: OP_sel, OP_mov...
- Registrador destino.
- Registrador fonte 1.
- Registrador fonte 2.
- Registrador fonte 3.
- Registrador com as flags.
- Tamanho dos operandos.

As micro operações possuem o tipo *TransOp* e devem ser inseridas no *TraceDecoder* (*this*), como no exemplo do Código 6. Elas também podem possuir parâmetros adicionais, como o parâmetro *cond*, no exemplo.

No caso da *cxch*, 4 micro operações são determinadas: uma para concatenar as *flags* que serão utilizadas para avaliar a condição(*OP_collcc*) e outras 3 para selecionar um valor dentre os registradores fontes com base na condição e nas *flags*(*OP_sel*). Dessa forma, quando as micro operações forem executadas, a troca será ou não feita com base nas *flags* salvas no registrador temporário *condreg*. É interessante notar o uso o de registradores temporários como intermediários para a troca, tais registradores são internos ao conjunto de instruções e não são visíveis para o usuário.

4.3.3 Introdução de Nova Micro Operação

Além da introdução de nova instrução, também é possível inserir novas micro operações no simulador. No projeto, foi inserida apenas uma micro operação simples, chamada *OP_aaa*, para auxiliar na obtenção das estatísticas, como será detalhado na Subseção 4.4.1 . Os códigos mostrados como exemplo nessa subseção são os implementados no projeto.

```
-- ptlhwdef.cpp --

enum {
```

```
  [...]
  OP_vpack_ss,
  OP_ast,
  OP_aaa,
  OP_MAX_OPCODE,
};

 -- ptlhwdef.cpp --

const OpcodeInfo opinfo[OP_MAX_OPCODE] = {
  [...]
  {"vpack.ss", OPCLASS_VEC_ALU, opAB },
  {"aaa", OPCLASS_LOGIC, opNOSIZE},
  {"ast", OPCLASS_SPECIAL, opABC|ccABC },
};

 -- atomcore.h --

const FunctionalUnitInfo fuinfo[OP_MAX_OPCODE] = {
  [...]
  {OP_vpack_ss, 2, P1, 1, ANYFPU},
  {OP_aaa, A, AP, 1, ANYFU},
  {OP_ast, 4, P0, 0, ANYINT},
};
```

Código 7: Inserção da nova micro operação OP_aaa nas estruturas de dados do MARSS.

O processo de inserção de nova micro operação é mais simples que o de nova instrução e pode ser resumido nos seguintes passos:

1. Inserção da micro operação nas estruturas de dados que as registram, exemplificado pelo Código 7:

   - No arquivo *ptlhwdef.h*, a micro operação é inserida na enumeração de micro operações na mesma posição em que será inserida nas demais estruturas.
   - No arquivo *ptlhwdef.cpp*, a micro operação é inserida na estrutura de informações de *Opcodes*, *opinfo*, juntamente com sua classe e tamanho.
   - No arquivo que representa o cabeçalho do núcleo do processador utilizado, no caso do projeto, *atomcore.h*, a micro operação é inserida na estrutura de informações de unidades funcionais, *fuinfo*. São inseridos, junto a ela, a sua latência, máscara interna e componente a ser utilizado para seu processamento.

2. Implementação da micro operação no arquivo *uopimpl.cpp*, como exemplificado pelo Código 8:
    - A micro operação deve ser inserida no *switch* da função *get_synthcode_for_uop*, onde ela deve atribuir a função que a implementa à variável *func*.
    - A função de implementação deve sempre seguir a assinatura mostrada no exemplo.

```
  case OP_vpack_ss:
    func = implmap_vpack_ss[size]; break;
  case OP_aaa:
    func = uop_impl_aaa; break;
  default:
    ptl_logfile << "Unknown uop opcode ", op, flush, " (", nameof(op), ")",
    endl, flush;
    assert(false);
}

[...]

void uop_impl_aaa(IssueState& state, W64 ra, W64 rb, W64 rc, W16 raflags,
                  W16 rbflags, W16 rcflags) {
    Context ctx = *ptl_contexts[0];
    statsMemoryInterval_begin = ctx.eip - 0xf000;
    statsMemoryInterval_end = ctx.eip + 0xf000;
    statsEnabled = statsEnabled ? false : true ;
    ptl_logfile << "Simulation Mark - uop: ", statsEnabled, endl;
}
```

Código 8: Implementação da micro operação OP_aaa no MARSS.

4.4 Modificações para Obtenção de Novas Estatísticas

4.4.1 A Instrução AAA

Originalmente desenvolvida para conversão de números do formato BCD para decimal, a instrução AAA caiu em desuso há anos e não é mais emitida pelos compiladores modernos. Ainda assim, para suportar código legado, os montadores continuam traduzindo tal instrução e as máquinas atuais continuam a implementando em seu conjunto de instruções. Desse modo, a instrução se torna útil para as simulações, uma vez que seu comportamento no simulador pode ser alterado para ter ações personalizadas, sem que o programa perca a compatibilidade com um processador real.

No projeto, a instrução foi utilizada como um marcador para iniciar ou parar a coleta de estatísticas no modo de simulação. No MARSS, o mecanismo original para fazê-lo é a alternância entre os modos de simulação e emulação, realizada por um biblioteca fornecida pelo simulador. Esta biblioteca, entretanto, não está disponível para compilação em 32 bits, como era necessário se fazer.

Além disso, também era necessário isolar as estatísticas do programa alvo das dos demais códigos de usuário sendo executados pelo sistema operacional, enquanto o MARSS apenas separava os códigos do *Kernel* dos demais. Assim, além de iniciar ou parar a coleta de estatísticas, a instrução AAA também foi configurada para utilizar seu endereço de memória para definir o intervalo de memória em que as estatísticas devem ser coletadas.

O código da implementação da instrução AAA é mostrado pelo Código 9. A implementação da micro operação *OP_aaa*, por ela emitida, é mostrada pelo Código 8. Sua função prática é a de definir o valor das variáveis *statsMemoryInterval_begin*, *statsMemoryInterval_end* e *statsEnabled*. As duas primeiras determinam qual o intervalo de memória em que as estatísticas devem ser coletadas e têm sua atribuição determinada pelo endereço da instrução AAA. Já a terceira variável define se a coleta de estatísticas deve ou não ser habilitada naquele momento.

```
case 0x37: {
    /* AAA */
    EndOfDecode();
    this << TransOp(OP_aaa, REG_zero, REG_zero, REG_zero, REG_zero, 3);
    break;
}
```

Código 9: Implementação da instrução AAA no MARSS.

O controle do processo de coleta de estatísticas é mostrado pelo Código 10. Ele efetua o isolamento das estatísticas do código alvo, direcionando todas as outras estatísticas para o modo *kernel*. Nele, o método *get_cs_eip* retorna o endereço sendo executado atualmente pelo simulador, o método *set_default_stats* define em que objeto as estatísticas devem ser acumuladas e o atributo *ctx.kernel_mode* sinaliza se a execução está sendo realizada em algum trecho do *kernel*. As demais variáveis são as mesmas do Código 8.

4.4.2 Impressão de estatísticas por instrução

Em alguns casos, o conhecimento das estatísticas totais de um código não foi suficiente para entender seu comportamento ou estatísticas produzidas. Assim, foi necessário implementar um mecanismo para se obter qual instrução do código gerou ou contribuiu para cada dado medido pelo simulador. Tal procedimento foi realizado para o número de *branch predictions* e de *cache misses*.

```
if(running_thread->ctx.kernel_mode) {
    running_thread->set_default_stats(kernel_stats);
}else if(statsEnabled && ptl_contexts[0]->get_cs_eip()
           > statsMemoryInterval_begin &&
         ptl_contexts[0]->get_cs_eip() <
           statsMemoryInterval_end){
    running_thread->set_default_stats(user_stats);
}else{
    running_thread->set_default_stats(kernel_stats);
}
```

Código 10: Controle da coleta de estatísticas implementado no MARSS.

```
if(!hit) {
    if(statsEnabled && ptl_contexts[0]->get_cs_eip() >
         statsMemoryInterval_begin &&
         ptl_contexts[0]->get_cs_eip() <
         statsMemoryInterval_end){
        fprintf(dcacheMissFile, "%llx\n", addr);
    }
    st_dcache.misses++;
}
```

Código 11: Código para obtenção de estatísticas por instrução no MARSS.

O Código 11 mostra a implementação deste método de obtenção de dados. Nele, pode-se notar que um traço dos endereços onde a estatística é incrementada é impresso para um arquivo. Para tornar os dados impressos mais compreensíveis foi desenvolvido um *script* para efetuar a contagem das ocorrências de endereços e combinar tais dados com os fornecidos pelo *objdump* do código alvo.

4.5 Estatísticas Fornecidas pelo Simulador

O simulador fornece um total de 296 estatísticas sobre o código executado. Outras 13 são fornecidas pelo McPAT. Esta seção discutirá os principais grupos de estatísticas fornecidos e as observações feitas a respeito deles durante o desenvolvimento do projeto. Além delas, o simulador também fornece o número total de ciclos da execução.

4.5.1 Módulos de *Cache*

São fornecidas 15 estatísticas sobre cada um dos módulos de *cache*, mostradas no diagrama da Figura 5. Para o projeto, foram configurados 3 módulos: L2, L1_I e L1_D. Nenhum deles

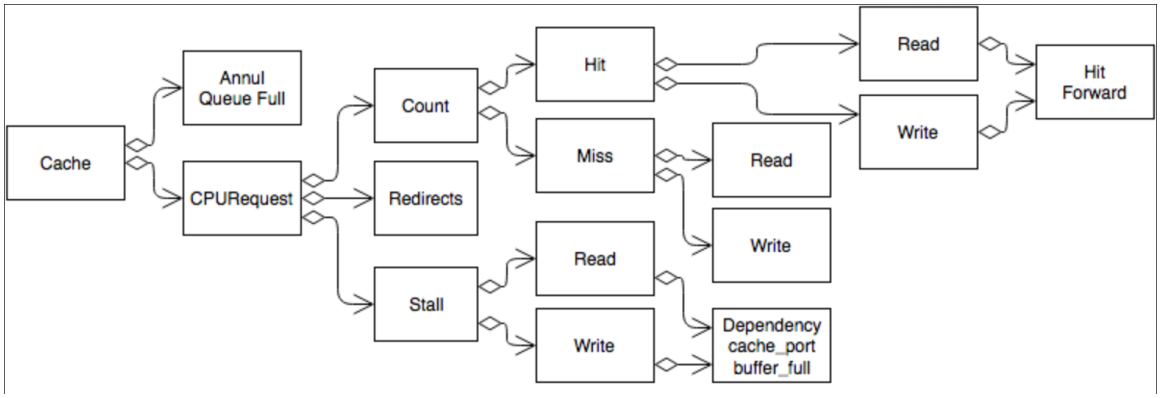

Figura 5: Diagrama das estatísticas da *cache*.

foi foco de análise durante o projeto. Além de tais estatísticas, o MARSS também fornece dados sobre a *cache* medidos para cada *thread* de processamento. Eles serão descritos na próxima subseção.

4.5.2 Estatísticas da *thread*: iCache, dCache, ITLB e DTLB

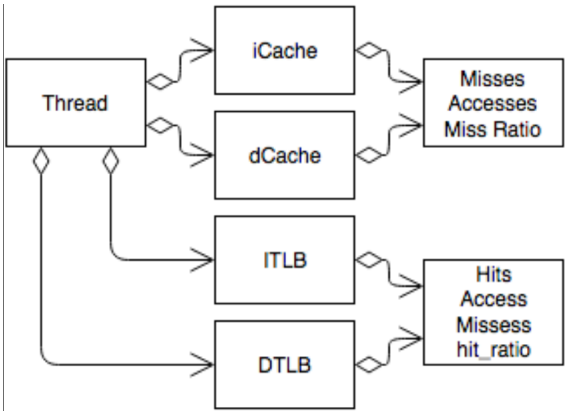

Figura 6: Digrama de estatística da *thread* para icache, dcache, ITLB e DTLB.

A Figura 6 apresenta 14 estatísticas associadas à *thread*. Dentre tais estatísticas, os dados relacionados à *cache* de dados, na Figura, dCache, foram bastante utilizados na análise do experimento do AES, que será descrito na Subseção 6.2. Os experimentos executados durante o projeto foram realizados utilizando uma única *thread*, como mostra as configurações da máquina na Subseção 4.2.2.1.3.

4.5.3 Predições de Desvio

Para a predição de desvios, 3 estatísticas são fornecidas: O número de predições, o número de falhas de predições e o número de atualizações do preditor. Tais dados, especialmente os

dois primeiros, tiverem fundamental importância na análise do experimento realizado com o protocolo X25519, que será descrito na Subseção 6.1.

Foi observado que o simulador executa a predição para todas as instruções, mesmo para as que não são desvios. Somente depois de executar ao menos uma predição em uma dada instrução é que ele passa a não predizer as que não são desvios.

4.5.4 Assistências

Assistência é a nomenclatura utilizada pelo MARSS para se referir a funções que auxiliam na execução eficiente de instruções mais complexas. Algumas delas traduzem instruções do MARSS diretamente para instruções nativas da máquina hospedeira, como é o caso da divisão.

São fornecidas 106 estatísticas a respeito das assistências. Durante os experimentos nenhuma delas apresentou resultados relevantes ou que auxiliassem quaisquer análises.

4.5.5 *Fetch*, *Decode*, *Issue* e *Commit* de Instruções e Micro Operações

O simulador fornece 48 estatística sobre *fetch*, 28 sobre *issue*, 6 sobre *commit* e 30 sobre *decode*. Elas são mostradas na Figura 7, exceto as que compõem a estatística *opclass* de *fetch*. Durante os experimentos, tais estatísticas foram importantes para, por exemplo, determinar a quantidade de cada tipo de *branch* executado.

As contagens de instruções fornecidas por esse grupo de estatísticas também foram bastante úteis à análise do comportamento dos algoritmos estudados.

4.5.6 Memória

O simulador fornece 4 vetores com estatísticas a respeito da memória. Cada um deles contém o número de leituras, de escritas, de acessos e de atualizações. Nos experimentos, como não havia interesse direto em tais dados, foi feita a soma dos dados de cada vetor e apenas o número final foi analisado. Não foi estudado a forma como ocorre a divisão de tais dados em vetores.

4.5.7 Energia

Como descrito pela Subseção 4.2.1.4 , é possível obter estatísticas sobre consumo de energia no MARSS através da utilização do McPAT. A Figura 8 mostra as estatísticas disponíveis para a configuração utilizada.

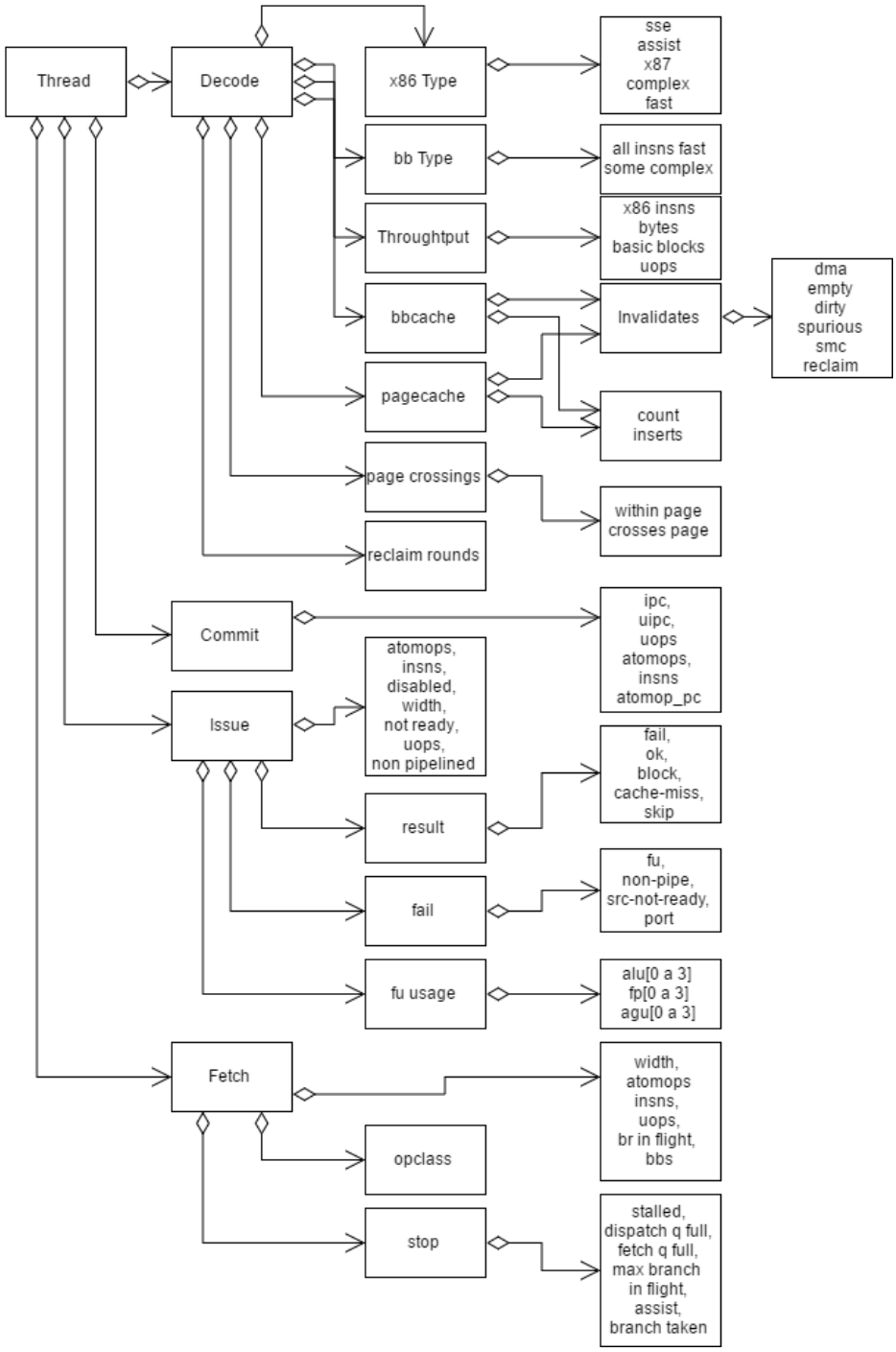

Figura 7: Diagrama das estatísticas de *Decode*, *Commit*, *Issue* e *Fetch* por *thread* no MARSS.

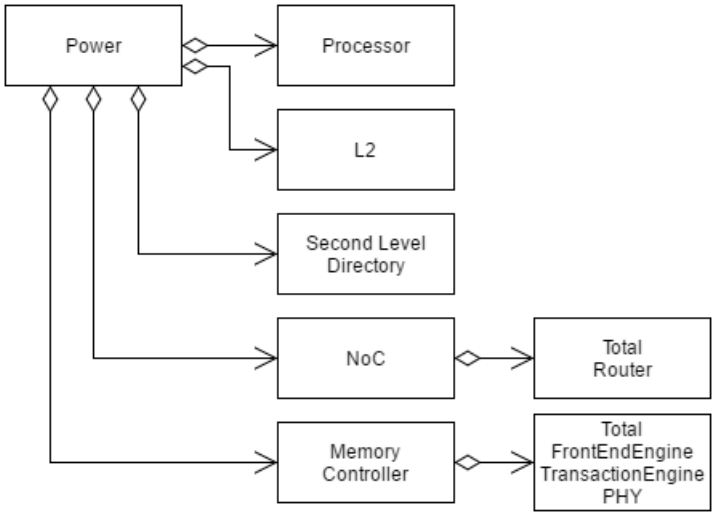

Figura 8: Diagrama das estatísticas de energia disponibilizadas pelo McPAT.

5 Plataforma FPGA

A realização dos experimentos com o simulador tiveram grande importância na obtenção e compreensão de estatísticas sobre o funcionamento dos algoritmos testados e obtiveram bons resultados na detecção de vulnerabilidades a ataques de canal de lateral. As instruções nele implementadas também puderam ser testadas e o impacto de suas inserções, em termos de segurança e desempenho, puderam ser profundamente analisados a partir do vasto conjunto de estatísticas fornecidas pelo simulador. Entretanto, todos esses resultados obtidos estavam apoiados na possibilidade, que apenas um simulador fornece, de se implementar instruções de forma ideal. No simulador, pode-se facilmente configurar uma instrução, em sua implementação micro arquitetural, para executar em tempo constante e sem vazar qualquer tipo de dado. Já numa implementação real, atingir tais garantias de segurança pode ser uma tarefa mais desafiadora.

Desse modo, considera-se prudente, para garantir a viabilidade de uma possível implementação real, que as instruções prototipadas também sejam testadas em *hardwares* reais, ainda que através da utilização de FPGAs.

Neste projeto, para protipação e teste das instruções em *hardware* real, foi utilizado a estrutura de descrição de *hardware* e programação de placas FPGAs da Altera e o *chip* FPGA Cyclone V. Nesta infraestrutura, foi descrito um sistema computacional utilizando o processador Altera NIOS. Dispondo de uma interface simplificada para extensão de seu conjunto de instruções, tal processador foi especificamente desenvolvido para tais placas, sendo utilizado tanto em aplicações de pesquisa, quanto comerciais.

A obtenção de estatísticas, agora sob a imposição de limitações físicas associadas ao

hardware real, teve de ser parcialmente realizada através das ferramentas de simulação da Altera. Diversos dados importantes, no entanto, como, por exemplo, o tempo de execução dos algoritmos, puderam ser obtidos diretamente da placa FPGA, permitindo a validação dos experimentos anteriormente realizados com o simulador.

5.1 Ambiente Utilizado

Do conjunto de ferramentas fornecidas pela Altera, 3 *softwares* foram principalmente utilizados. As subseções seguintes apresentam uma descrição de cada um deles e seus usos no projeto.

5.1.1 Quartus Prime

Ambiente de desenvolvimento integrado (IDE, na sigla em inglês) da Altera para descrição de *Hardware*, o Quartus Prime atua como porta de acesso para as demais ferramentas, além de prover diretamente a edição e compilação dos códigos de descrição do sistema. Nele, foram descritas cada uma das novas instruções implementadas, bem como o código de alto nível do projeto, responsável por fazer a instanciação dos componentes e efetuar sua conexão com os pinos da placa.

A ferramenta também foi utilizada na análise de tempo e frequências de operação do sistema e das instruções implementadas, servindo como interface para o *TimeQuest Timing Analyzer*, *software* que de fato executa tal função.

5.1.2 Qsys - System Integration Tool

Também parte do conjunto de ferramentas do Quartus, o Qsys é um *software* voltado a facilitar a integração de componentes pré configurados de um sistema. Nele foram instanciados cada um dos componentes do sistema computacional utilizado, como, por exemplo, o processador, memórias, conexões de dados, instruções personalizadas, entre outros. O *software* efetua de maneira semi-automática a conexão entre tais componentes e gera um novo componente como resultado da combinação dos componentes instanciados. Dessa forma, torna-se possível e prática a criação de um projeto hierárquico de componentes que auxilia e acelera a instanciação de grandes sistemas.

Os componentes instanciados na criação do sistema computacional utilizado no projeto serão descritos na Subseção 5.3

5.1.3 NIOS II Software Build Tools for Eclipse

Construído sobre o Eclipse, o *software* dispõe de ferramentas que permitem a programação e depuração de códigos para o sistema instanciado no Quartus.

Atuando como interface para o NIOS II Command Shell, ele efetua a criação automática do *software* de suporte (BSP, sigla em inglês para *Board Support Package*), a compilação cruzada do código, a transmissão do *elf* gerado para a placa e o controle da execução de tal código. Permite ainda a integração com o perfilador GNU gprof [8], além de fornecer interface para a depuração direta do código. O sistema também gera automaticamente um conjunto de *macros* para facilitar o controle sobre os componentes de *hardware* instanciados no Qsys, inclusive para as instruções customizadas definidas pelo próprio usuário.

Apesar das diversas facilidades oferecidas por este ambiente, no decorrer do projeto, foi necessária a criação de *scripts* que efetuassem tais tarefas diretamente pela NIOS II Command Shell, para que fosse possível, desse modo, automatizar um grande número de execuções.

5.2 Funcionamento

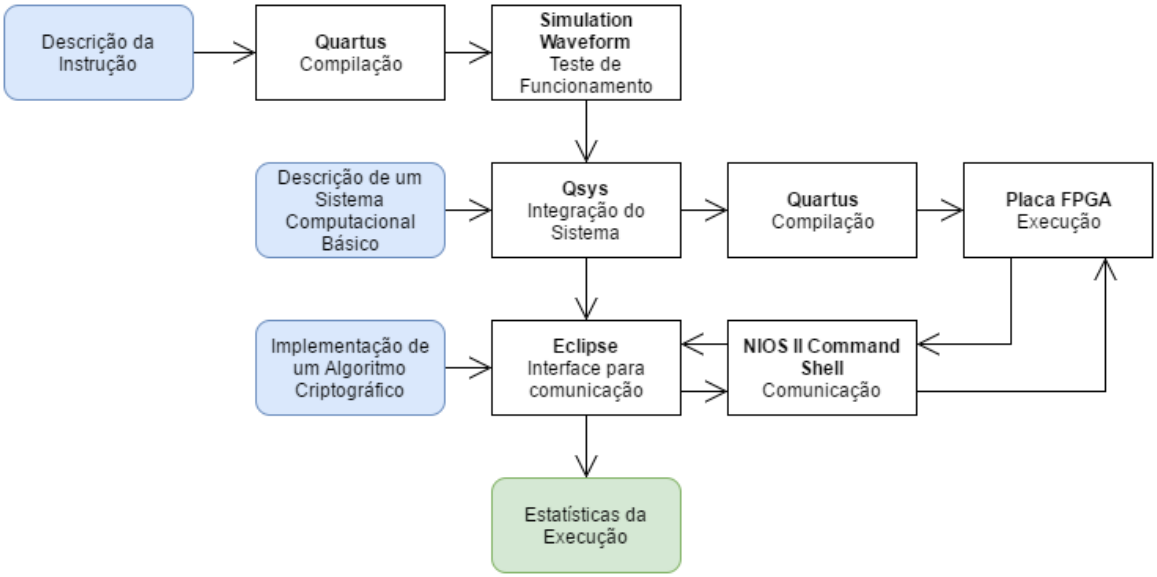

Figura 9: Diagrama do processo de utilização do ambiente da Altera

A Figura 9 descreve o o processo efetuado para obtenção de estatísticas a partir da utilização do conjunto de ferramentas da Altera e da placa FPGA. Nela, os quadrados arredondados azuis representam as entradas, enquanto os verdes representam a saída. O processo consiste, basicamente, dos seguintes passos:

1. A Instrução é descrita em linguagem de descrição de *hardware*. No projeto, foi utilizado a linguagem Verilog.
2. A descrição é compilada pelo Quartus e a instrução tem seu funcionamento testado no *Simulation Waveform*

3. Utilizando o Qsys, um sistema computacional básico, tendo o NIOS como processador, é instanciado e a nova instrução descrita é inserida nele.
4. O Quartus é, mais uma vez, utilizado na compilação de todo o sistema.
5. O Sistema, já compilado, é instalado na FPGA.
6. O Algoritmo criptográfico, implementado em linguagem C, é compilado através do Eclipse.
7. O Eclipse, através da NIOS II Command Shell, se comunica com aplaca FPGA, executando o código e obtendo as estatísticas de execução.
8. As estatísticas são forncidas para o usuário.

5.3 Sistema Computacional Descrito

No projeto, foi descrito um sistema composto por 8 componentes básico e 3 componentes contendo instruções customizadas. O componentes básicos utilizados foram:

- Clock: Clock básico do sistema. Foi definido em 50MHz. Entretanto, segundo o *TimeQuest Timing Analyzer*, tal frequência poderia ser de até 94Mhz.
- NIOS Gen II: Processador. Será descrito na próxima subseção.
- JTAG_UART: Componente para possibilitar comunicação da NIOS II Command Shell com o sistema.
- Onchip Memory: Memória RAM do sistema. Utilizada na execução dos códigos e armazenamento dos dados durante a operação. Foi configurada para armazenar 204kb.
- SYS ID: ID do sistema. Utilizado na identificação do sistema. Este ID é comparado pelo Eclipse ao ID do BSP utilizado nos códigos a serem executados.
- LED: Componente para controlar os LEDs da placa.
- Sys Clock Timer: Contador de tempo para o sistema, capaz de gerar uma interrupção de tempo para o processador.
- Performance Counter: Periférico para medição de tempo de trechos dos códigos executados.

5.3.1 Processador NIOS

O processador NIOS é um processador embarcado especificamente desenvolvido para executar em placas FPGAs da Altera. Ele possuí arquitetura RISC e é facilmente integrável com periféricos e instruções customizadas desenvolvidas pelo usuário. Ele está disponível em duas versão: A versão E, que otimiza o uso de recursos da placa, e a versão F, que otimiza o processamento. A Figura 10 mostra uma comparação básica entre as versões,

Caracteristica		Versão	
		E	F
Área		<700 LEs; <350 ALMs	Sem MMU ou MPU:, <1800 LEs;, <900 ALMs Com MMU:, <3000 LEs;, <1500 ALMs Com MPU:, <2400 LEs;, <1200 ALMs
Pipeline		1 Estágio	6 Estágios
Espaço de Endereçamento Externo		2 GB	4 GB
Instruções	Cache	-	512 bytes a 64 KB
	Predição de Desvios	-	Dinâmico ou Estático
Dados	Cache	-	512 bytes a 64 KB
	Cache Bypass	-	Instruções de I/O Bit-31 cache bypass MMU Opcional
ALU	Multiplicação em Hardware	-	1 ciclo
	Divisão em Hardware	-	Opcional (35 ciclos)
	Shift	1 ciclo por bit	1 ciclo
Suporte a Modo de Usuário		Não. Sempre Supervisor	Sim. Quando habilitado MMU e MPU
MMU E MPU		-	Opcional

Tabela 1: Tabela comprativa entre as versões do NIOS. Traduzida do Manual [1]

enquanto a Tabela 1 mostra uma comparação mais detalhada.

No projeto, foi escolhida a versão F do processador. A *cache* de instruções foi configurada para 4Kb, enquanto a de dados foi configurada para 2 Kb. Ambas possuem como limite 64Kb. Foi também configurado um preditor de desvios dinâmico com 256 entradas e a multiplicação estendida foi ativada. Os demais parâmetros permaneceram com os valores padrões.

	Nios II/e	Nios II/f
Summary	**Resource-optimized 32-bit RISC**	**Performance-optimized 32-bit RISC**
Features	**JTAG Debug** **ECC RAM Protection**	**JTAG Debug** **Hardware Multiply/Divide** **Instruction/Data Caches** **Tightly-Coupled Masters** **ECC RAM Protection** **External Interrupt Controller** **Shadow Register Sets** **MPU** **MMU**
RAM Usage	2 + Options	2 + Options

Figura 10: Comparação básica entre as versões do NIOS. Retirada da ferramenta.

5.4 Inserção de Instruções

O NIOS fornece uma interface bastante simplificada para a introdução de novas instruções. O Código 12 mostra um modelo básico para a implementação.

```
module cmov(
         clk,          // CPU system clock (required for multicycle or extended multicycle)
         reset,          // CPU master asynchronous active high reset
         n,                  // N-field selector (required for extended)
         done,
         start,
         clk_en,
         reset_req,
         dataa,          // Operand A (always required)
         datab,          // Operand B (optional)
         a,                  // Internal operand A index register
         b,                  // Internal operand B index register
         c,                  // Internal result index register
         readra, // Read operand A from CPU (otherwise use internal operand A)
```

```
         readrb, // Read operand B from CPU (otherwise use internal operand B)
         writerc,// Write result to CPU (otherwise write to internal result)
        result         // Result (always required)
);


//INPUTS
input reset;
input reset_req;
input [1:0] n; // modify width to match the number of unique operations
input clk;
input clk_en;
input start;

 // in the instruction
input [4:0] a;
input [4:0] b;
input [4:0] c;
input readra;
input readrb;
input writerc;
input [31:0] dataa;
input [31:0] datab;

//OUTPUTS
output [31:0] result;
output reg done;

endmodule
```

Código 12: Modelo básico para implementação de instruções no NIOS

Dentre os sinais apresentados nele, os sinais *clk*, *reset*, *done*, *start*, *clk_en* e *reset_req* são exigidos apenas em caso de instruções multi-ciclo. Para as instruções combinacionais, que executam em 1 ciclo, os sinais comumente utilizados são *dataa* e *datab*, que representam os operadores, e *n*, que é uma constante seletora, utilizada para indicar diferentes operações internas.

5.4.1 Movimentação e Troca Condicionais

As instruções de movimentação e troca condicionais foram implementadas para que se pudesse reproduzir um experimento similar ao realizado no simulador. Como o modelo do NIOS só é capaz de receber dois operandos e retornar um valor por vez, foi necessário que a

instrução fosse implementada como multi-ciclo e que armazenasse estado entre as chamadas.

O Código 13 mostra a implementação de tais instruções.

```
reg [31:0] temp;
reg cmp;
reg [31:0] res;
always @ (negedge clk)
begin
        if(clk_en) begin
                if(n == 1) begin
                        cmp = (dataa == datab) ? 1 : 0;
                        res = cmp;
                end else if (n == 2) begin
                        res = (cmp == 1) ? dataa : datab;
                        temp = (cmp == 1) ? datab : dataa;
                end else if (n == 3) begin
                        res = temp;
                end
                done = 1;
        end
end

assign result = res;
```

Código 13: Implementação das instruções de movimentação e troca condicional.

Assim, quando n é igual a 1, a instrução compara os dados de entrada e armazena no registrador *cmp* o valor 1 se forem iguais, ou 0, caso contrário. Quando n é igual a 2, é retornado o operando *dataa* ou *datab*, a depender do valor anteriormente atribuído para *cmp*. Assim, concretiza-se a instrução de movimentação condicional. Para o caso da troca condicional, há ainda uma terceira chamada com n igual a 3. Esta retorna o valor que não foi retornado na chamada em que n era igual a 2.

5.4.2 Paridade de *Bits*

Prosseguindo com a análise de outros algoritmos, foi implementada uma instrução para o cálculo da paridade de *bits*. Tal instrução é capaz de calcular, simultaneamente, uma das seguintes opções:

- Paridade de 1 dado de 64bits.
- Paridade de 1 dado de 32bits.
- Paridade de 2 dados de 16bits.

- Paridade de 4 dados de 8bits.
- Paridade de 8 dados de 4bits.
- Paridade de 16 dados de 2bits.

Seu código é mostrado no Código 14.

```
assign step0 = dataa ^ datab;

generate for(i = 0; i < 31; i = i + 2) begin: redu1
assign step1[31 - i : 30 - i] = {1'b0, step0[31 - i] ^ step0[30 - i]};
end endgenerate //0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_

generate for(i = 0; i < 31; i = i + 4) begin: redu2
assign step2[31 - i : 28 - i] = {3'b0, step1[30 - i] ^ step1[28 - i]};
end endgenerate //000_000_000_000_000_000_000_000_

generate for(i = 0; i < 31; i = i + 8) begin: redu3
assign step3[31 - i : 24 - i] = {7'b0, step2[28 - i] ^ step2[24 - i]};
end endgenerate //0000000_0000000_0000000_0000000_

assign step4 = {15'b0, step3[24] ^ step3[16], 15'b0, step3[8] ^ step3[0]};
//000000000000000_000000000000000_

assign step5 = step4[16] ^ step4[0];

assign result = (n == 0) ?
        step1 :
        (n == 1) ?
                step2 :
                (n == 2) ?
                        step3 :
                                (n == 3) ?
                                step4 :
                                        step5;
```

Código 14: Implementação das instrução de paridade.

5.4.3 Intercalação de *Bits*

A instrução de intercalação de *bits* recebe dois inteiros e intercala seus bits dois a dois, quatro a quatro, oito a oito ou dezesseis a dezesseis. Seu código é mostrado no Código 15. Não foram implementados algoritmos para testar tal instrução.

```
wire [31:0] result1, result2, result4, result8;
genvar i;

// n = 1
generate for(i = 0; i < 16; i = i + 1) begin: intercalacaoN1
assign result1[31 - 2*i] = dataa[31 - i];
assign result1[31 - 2*i - 1] = datab[31 - i];
end endgenerate

//n = 2
generate for(i = 0; i < 16; i = i + 2) begin: intercalacaoN2
assign result2[31 - 2*i : 31 - 2*i - 1] = dataa[31 - i : 31 - i - 1];
assign result2[31 - 2*i - 2 : 31 - 2*i - 3] = datab[31 - i : 31 - i - 1];
end endgenerate

// n = 4
generate for(i = 0; i < 16; i = i + 4) begin: intercalacaoN4
assign result4[31 - 2*i : 31 - 2*i - 3] = dataa[31 - i : 31 - i - 3];
assign result4[31 - 2*i - 4 : 31 - 2*i - 7] = datab[31 - i : 31 - i - 3];
end endgenerate

// n = 8
generate for(i = 0; i < 16; i = i + 8) begin: intercalacaoN8
assign result8[31 - 2*i : 31 - 2*i - 7] = dataa[31 - i : 31 - i - 7];
assign result8[31 - 2*i - 8 : 31 - 2*i - 15] = datab[31 - i : 31 - i - 7];
end endgenerate

assign result = (n == 0) ?
        result1 :
        (n == 1) ?
                result2 :
                (n == 2) ?
                        result4 :
                                result8;
```

Código 15: Implementação das instrução de intercalação.

6 Experimentos Realizados

Utilizando os ambientes descritos nas seções anteriores, experimentos com dois algoritmos foram realizados. Suas descrições, objetivos e resultados são apresentados nas próximas subseções. Como o foco deste trabalho esteve na prototipação e validação das instruções,

não serão detalhadas as fundamentações teóricas dos algoritmos e dos testes executados.

6.1 O Protocolo X25519

6.1.1 O Algoritmo

Desenvolvida por Daniel Bernstein, a X25519 [5] é uma função de curva elíptica para a implementação do algoritmo de Diffie-Hellman. Seu funcionamento é ilustrado pela Figura 11.

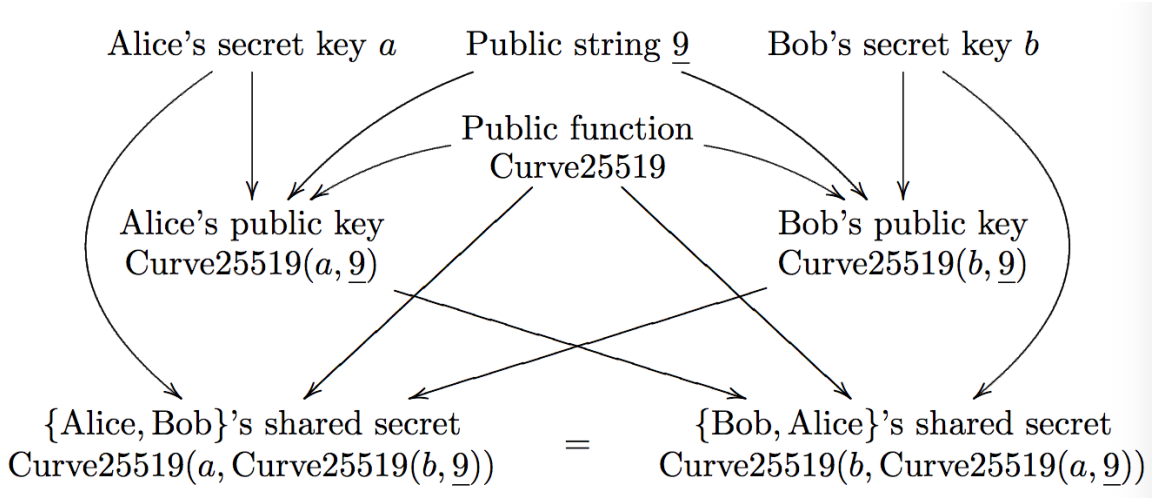

Figura 11: Protocolo de funcionamento da função X25519. Retirado da referência [5]

Para o experimento foi tomado como base a implementação `ref10`, presente no SUPERCOP [6]. Nela, a fim de se substituir o condicional presente na cadeia de Montgomery [11], foi utilizada uma função que efetua troca de valores entre dois vetores. Tal função atua com base diretamente nos bits da chave secreta, percorrendo-os sequencialmente e executando a troca somente se houver uma transição entre eles. Dessa forma, é necessário que o algoritmo garanta que a execução dessa função, executando a troca ou não, ocorra em tempo constante, do contrário cria-se margem a um ataque de canal lateral por tempo.

6.1.2 O Experimento

O experimento consistiu em testar em 4 versões da X25519 com o objetivo de se detectar possíveis fontes de vazamento de dados por canais laterais. Elas foram geradas a partir de modificações na função de troca da implementação presente no SUPERCOP. 3 delas são versões consideradas seguras, onde se espera que as estatísticas fornecidas pelo simulador não variem de acordo com a chave entrada, enquanto a quarta é uma versão insegura, utilizada para testar a capacidade do simulador de demonstrar sua vulnerabilidade.

O código das 4 versões da função de troca é mostrado a seguir, simplificado de modo a ocultar trechos não pertinente à lógica da função. O primeiro deles, *fe_cswap*, mostrado no

Código 16, é a função de troca original presente no SUPERCOP.

```
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;
}
```

Código 16: Código da fe_cswap, presente no SUPERCOP.

O segundo, denominado *ns_select* e mostrado pelo Código 17, é uma função de troca insegura, baseado em um condicional simples e que terá sua vulnerabilidade demonstrada pelo experimento.

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

Código 17: Código da função de troca insegura.

O terceiro, implementado em linguagem de montagem, efetua a troca utilizando a instrução de movimentação condicional presente na arquitetura x86. Ele é mostrado no Código 18. O quarto, mostrado pelo Código 19, utiliza uma nova instrução de troca condicional especialmente construída para este algoritmo, a *cxch*. Para o teste com o processador NIOS, essas duas implementações foram refeitas com as instruções correspondentes desta arquitetura.

Cada uma das versões foi testada para dois conjuntos de chaves diferentes. O primeiro conjunto apresenta chaves com número de transições de *bits* crescente, enquanto o segundo apresenta chaves com mesmo número de transições, porém com disposição diferente. Para possibilitar uma melhor análise dos dados, foram medidos tanto o algoritmo completo, quanto apenas a função de troca.

```
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
```

Código 18: Código da função de troca utilizando a instrução CMOV.

```
.endr
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
```

Código 19: Código da função de troca utilizando a nova instrução CXCH.

6.1.3 Resultados do Simulador

Para este experimento, as principais estatísticas analisadas foram o número de ciclos, pois é através dele que se detecta a vulnerabilidade a um ataque de canal lateral por tempo, e o número de predições de desvio incorretas, que são a causa da variação no número de ciclos nos casos em que o número de transições da chave permanece constante.

Para o primeiro conjunto de chaves, com número de transições crescente, foram obtidos os números de ciclos mostrados nos gráficos das Figura 12 e Figura 13. As chaves K1, KX e K256 são chaves com 1, 120 e 256 transições de *bits*, respectivamente. Os resultados são exatamente o esperado para tal experimento: Enquanto as versões seguras permanecem com dados constantes, a menos da variação do próprio sistema operacional; a versão Insegura tem uma variação clara no número de ciclos, o que permitiria a um atacante determinar o número de transições na chave.

Já para o segundo conjunto de chaves, com mesmo número de transições, porém distribuição diferente, foram utilizadas as seguintes chaves:

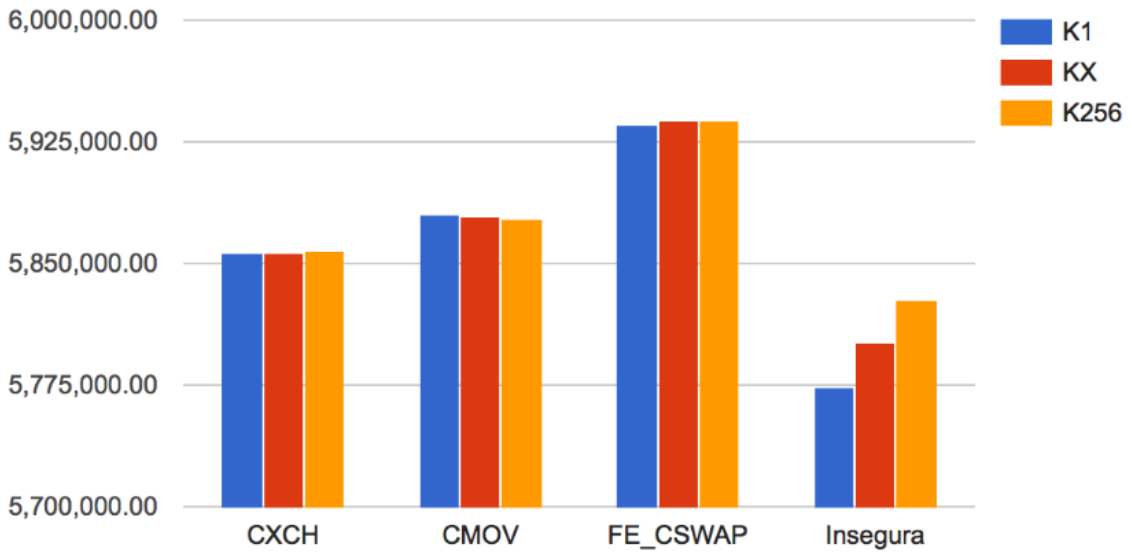

Figura 12: Número de ciclos de execução do algoritmo completo.

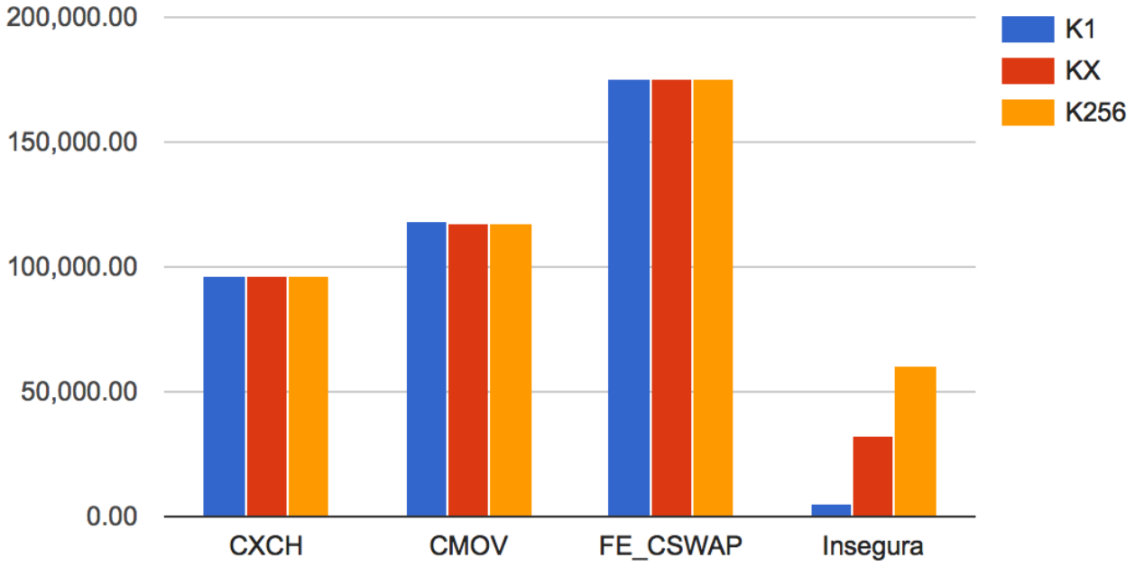

Figura 13: Número de ciclos de execução da função de troca.

- kA: Chave com 120 transições distribuídas de forma aleatória.
- KB: Chave com 120 transições bem comportadas, facilmente prediziveis.

Os resultados obtidos são mostrados no gráfico da Figura 14. Nele, é apresentada a

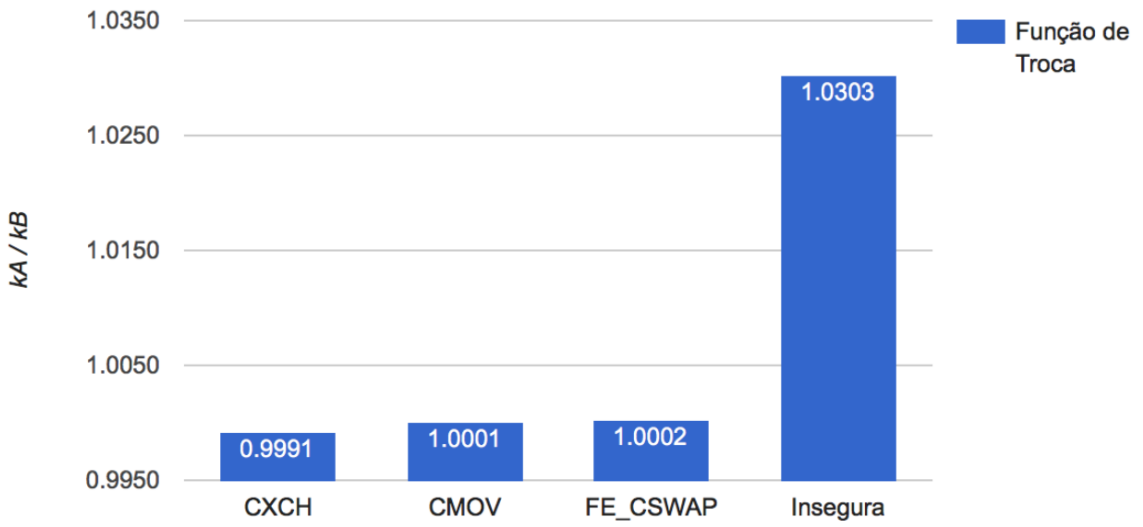

Figura 14: Razão entre o número de ciclos de execução da função de troca com a chave kA e com a chave kB.

razão entre o número de ciclos da execução com chave kA e o da execução com a chave kB. Nota-se que, enquanto a variação das versões seguras permanecem na ordem de 0.01%, a versão insegura tem variação maior que 3%.

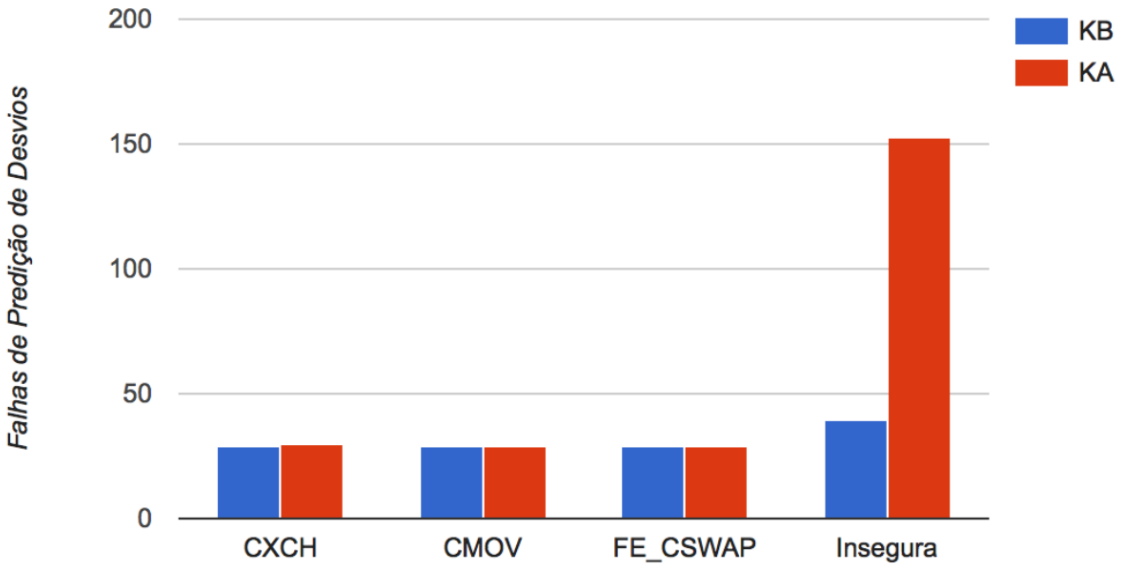

Figura 15: Número de falhas na predição de desvio do algoritmo completo.

O aumento do número de ciclos na versão insegura pode ser explicado pelo gráfico da Figura 15. Nele, é possível notar um grande aumento no número de falhas de predições de desvios da versão insegura, que ocorre devido a incapacidade, do processador, de predizer as transições de bits na chave em que elas estão dispostas de forma aleatória. Com o aumento de tais falhas, o tempo de execução do algoritmo também aumenta.

Assim, conclui-se que, em relação a implementação insegura, um atacante conseguiria não só obter informação sobre o número de transições de *bits*, mas também sobre a disposição de tais transições na chave.

Um segundo objetivo desse experimento esteve em demonstrar o ganho em desempenho da execução com a nova instrução. O gráfico da Figura 13 mostra um ganho de desempenho de 45% da função de troca com a *cxch*, em relação a versão original, *fe_cswap*. Para o algoritmo todo, o ganho de desempenho ficou em 1.4%, também em relação a versão original.

6.1.4 Resultados FPGA

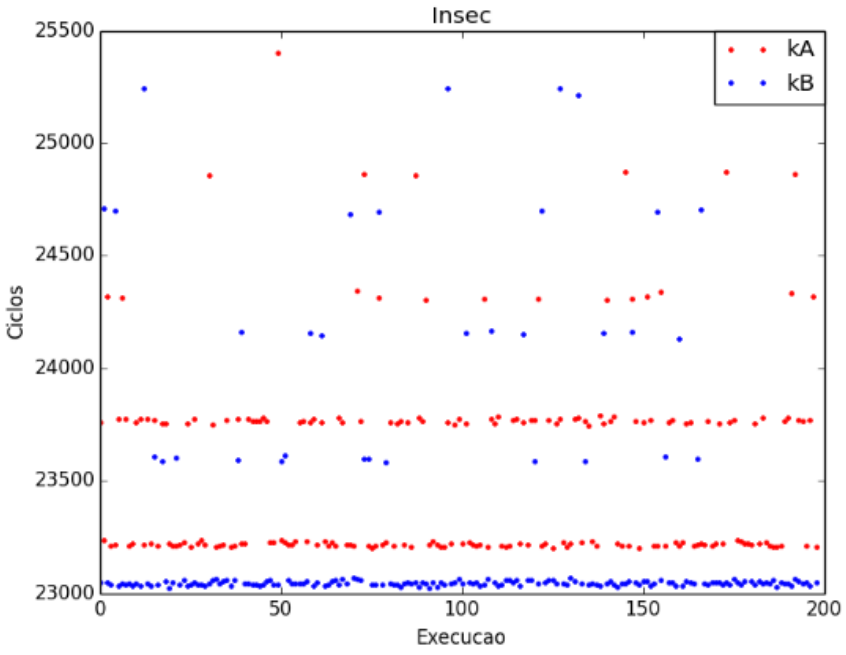

Figura 16: Tempo de execução do trecho de troca na FPGA para 200 execuções da versão insegura.

A FPGA apresentou maiores variações nos resultados, que não permitiram detectar o vazamento diretamente pela média dos dados. Entretanto, observando os padrões dos tem-

pos de execuções nota-se claramente a vulnerabilidade insegura, enquanto as versões seguras permanecem com tempo constante. Foi executado nela somente o segundo teste descrito anteriormente. O gráfico da Figura 16 mostra os tempos de execução da versão insegura, enquanto os da Figura 17 mostra os da versão segura.

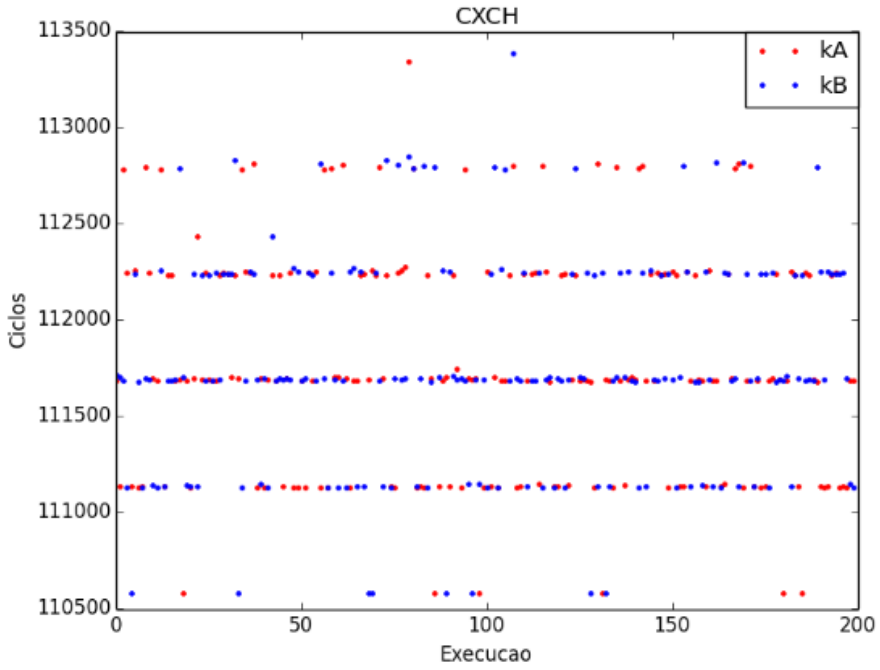

Figura 17: Tempo de execução do trecho de troca na FPGA para 200 execuções da versão segura.

A partir dos resultados dos gráficos, fica nítido que a chave aleatória kA consume, em geral, mais tempo que a kB, devido aos erros de predição de desvio. Assim, confirma-se a vulnerabilidade da versão insegura.

Durante a execução com a FPGA foi também identificado um vazamento de dados nas operações de multiplicação. Como mostra a Figura 18, na versão segura, havia uma diferença de aproximadamente dois mil ciclos entre as diferentes chaves. O razão de tal vazamento estava na forma como o compilador implementava a multiplicação de 64 *bits*. Como o NIOS, em sua configuração padrão, não é capaz de fazer multiplicação estendida, ela era implementada em *software*. Assim, o compilador, visando obter melhor desempenho, inseriu um condicional na implementação de multiplicação para, caso possível, a operação fosse encerrada assim que o ultimo *bit* significativo fosse processado. Desse modo, o tempo passou a variar de acordo com os *bits* dos operandos. A solução para tal problema foi a habilitação, nas configurações do processador, da multiplicação estendida em *hardware*.

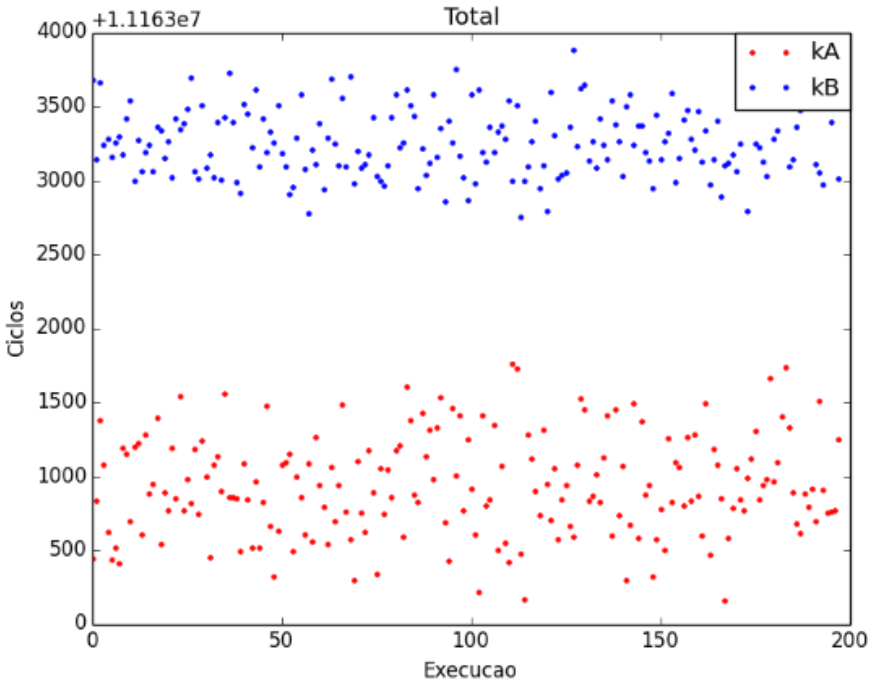

Figura 18: Tempo de execução do trecho de multiplicação na FPGA para 200 execuções da versão segura.

6.2 AES

Como não foi possível obter estatísticas sobre o número de *cache misses* na FPGA, este experimento foi executado apenas no simulador.

6.2.1 O Algoritmo

Originalmente conhecido como Rijndael [7], o AES é o atual padrão para criptografia simétrica escolhido pelo NIST em 2001. Ele é composto por 10, 12 ou 14 turnos de execução, a depender do nível de segurança escolhido. O padrão de propagação de atividade de um turno é mostrado na Figura 19.

A etapa *ByteSub* consiste em substituir os *bytes* do estado do algoritmo utilizando uma tabela de 256 *bytes*, denominada *sbox*. A substituição depende diretamente dos *bits* do estado e é um ponto de vulnerabilidade conhecido, uma vez que um ataque de canal lateral por colisão de *cache* é capaz de detectar quais posições da *sbox* estão sendo acessadas e, consequentemente, qual o estado do algoritmo.

Como alternativa ao uso da tabela *sbox*, os valores da substituição *ByteSub* podem ser matematicamente calculados durante a execução, o que elimina a vulnerabilidade, mas ocasiona um grande aumento no tempo de processamento.

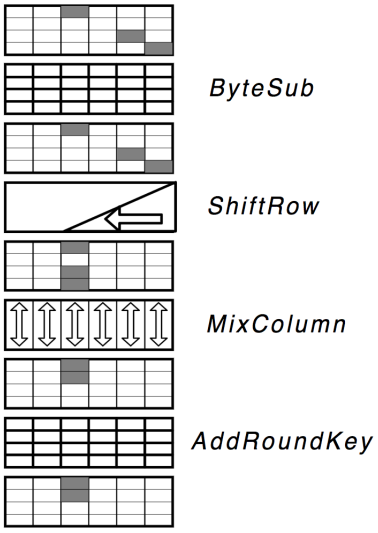

Figura 19: Padrão de propagação de atividade do AES durante 1 round. Retirado da referência [7].

6.2.2 O Experimento

O experimento consistiu em testar duas implementações do AES, uma utilizando a tabela *sbox* e outra fazendo os cálculos durante a execução do algoritmo. Foram medidos o número de ciclos de cada versão, a fim de medir o impacto em desempenho da retirada da tabela, e o número de *cache misses*, com o objetivo demonstrar qual o ponto de vulnerabilidade na versão com a tabela.

6.2.3 Resultados

Como mostrado na Figura 20, o número de ciclos da versão sem tabela foi quase 40 vezes maior do que na versão com tabela. Uma queda de desempenho bastante significativa e que, atualmente, justifica o uso da versão com tabela, ainda que seja vulnerável. O gráfico também apresenta o número de ciclos apenas para a etapa *ByteSub*, permitindo notar que, enquanto esta etapa representa apenas 23% do processamento na versão com tabela, ela representa 98% na versão sem tabela.

O número de acessos à *cache* e de *cache misses* também foi medido. A princípio, esperava-se obter números maiores na versão com tabela, devido aos acessos a esta. Porém, como pode ser observado na Figura 21, os números da versão sem tabela foram maiores.

Como detalhado na Subseção 4.4.2, foi implementado no simulador a impressão de estatísticas por endereço. Desse modo, pode-se detectar o padrão dos acessos à *cache* e o alto número de acessos e *cache misses* na versão sem tabela pôde ser explicado. As principais fontes de *cache miss* da versão com tabela estavam igualmente distribuídas pela

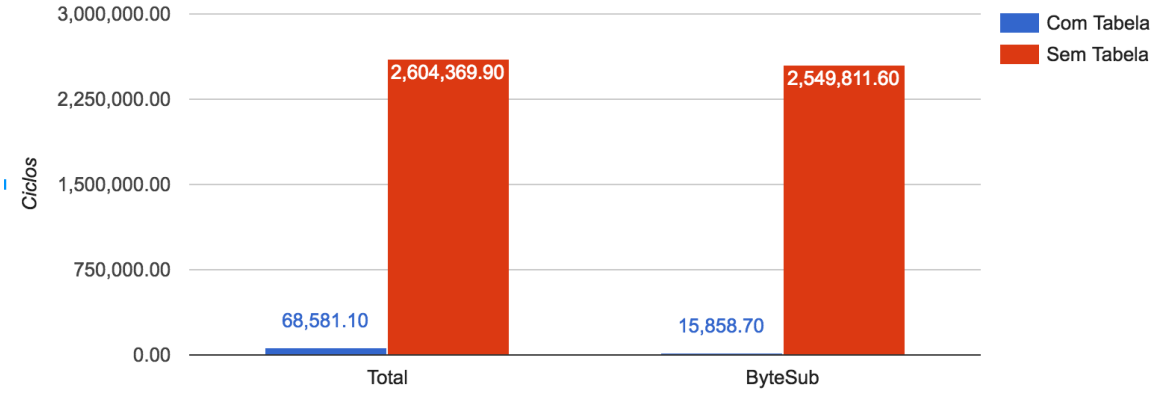

Figura 20: Número de ciclos das execuções com e sem tabela para o o Algoritmo todo e apenas para a substituição de *bytes*.

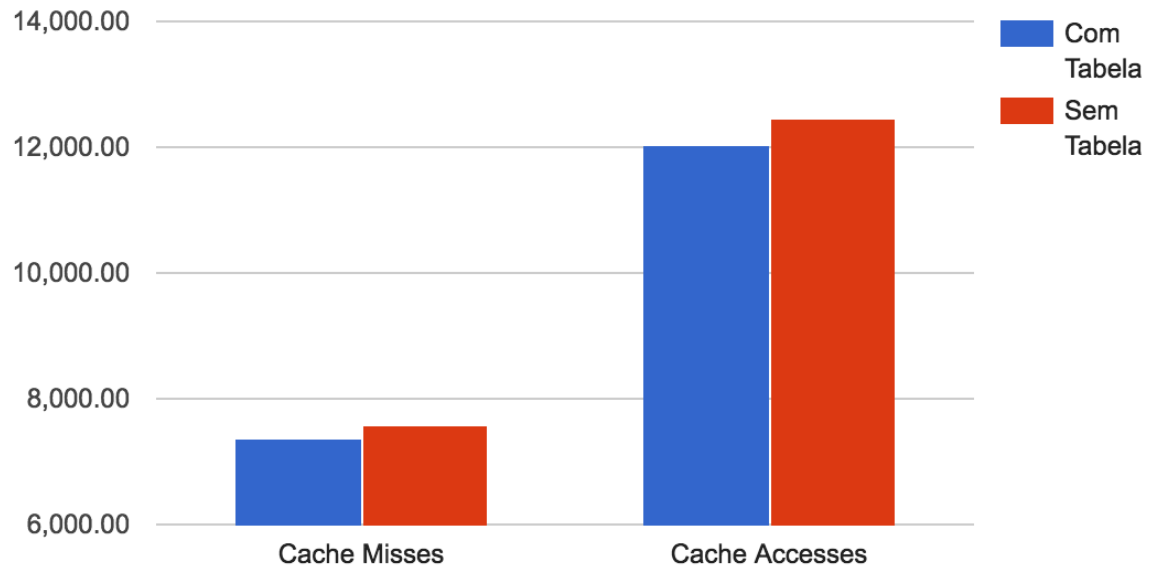

Figura 21: Número de acessos à *cache* e *cache misses* nas versões com e sem tabela.

sbox. Enquanto, na versão sem tabela, os *cache misses* se concentravam em poucas variáveis, o que é provavelmente causado por passagens de parâmetros ou *spill* de registradores.

7 Conclusão

Neste relatório foram apresentados os resultados da prototipação de instruções para implementação resistente a ataques de canal lateral de algoritmos criptográficos. Apresentou-se também a validação, em segurança e desempenho, de tais implementações. Para realização dos experimentos, dois ambientes foram apresentados: Um virtual, construído com um simulador de sistema, que é menos realista, mas que fornece maior número de estatísticas; e um real, construído a partir da utilização da tecnologia FPGA.

Nos experimentos, o simulador atuou como esperado, fornecendo medidas acuradas e com razoável eficiência. Os dados por ele fornecidos possibilitaram conclusões interessantes sobre os algoritmos executados, assim como sobre o próprio simulador. No caso do experimento com o protocolo X25519, por exemplo, o simulador se mostrou uma ferramenta bastante precisa na detecção de vazamento de dados por canais laterais. Já o experimento com o AES permitiu demonstrar a importância da possibilidade de alteração do código do simulador. Nele, dados que a princípio contrariavam as expectativas, foram facilmente explicados a partir da implementação da coleta de novas estatísticas.

Os experimentos com a FPGA, apesar de apresentarem maiores dificuldades, também cumpriram com o esperado. Os resultados do simulador foram devidamente confirmados e detecções de vulnerabilidades inesperadas, porém corretas, como foi o caso da implementação da multiplicação em *software*, ocorreram neste ambiente.

De modo geral, em ambos os ambientes foi possível detectar os vazamentos por canal lateral e validar corretamente a segurança das implementações criptográficas testadas, bem como medir o impacto, em desempenho, da inserção das novas instruções.

Referências

[1] Nios II Altera. Processor reference, 2014.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 164–177. ACM, 2003.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[4] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml^{TM}) version 1.1. *yaml. org, Tech. Rep*, 2005.

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

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

[7] Joan Daemen and Vincent Rijmen. The block cipher rijndael. In *International Conference on Smart Card Research and Advanced Applications*, pages 277–284. Springer, 1998.

[8] Jay Fenlason and Richard Stallman. Gnu gprof: the gnu profiler. *Manual, Free Software Foundation Inc*, 1997.

[9] Steven Knight, Chad Austin, Charles Crain, Steve Leblanc, and Anthony Roach. Scons software construction tool, 2011.

[10] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480. ACM, 2009.

[11] Peter L Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[12] Avadh Patel, Furat Afram, and Kanad Ghose. Marss-x86: A qemu-based microarchitectural and systems simulator for x86 multicore processors. In *1st International Qemu Users' Forum*, pages 29–30, 2011.

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

[14] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[15] Matt T Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 23–34. IEEE, 2007.