Implementação de PUF - SRAM

Autores

Lucas Finger Roman (lfrfinger@gmail.com) e Gustavo José Carpeggiani (gustavocarpeggiani@hotmail.com)

Motivação:

Métodos convencionais de geração e armazenamento de chaves para a comunicação entre dispositivos não são completamente seguros, visto que tradicionalmente a chave fica armazenada em memória não volátil, o que a deixa suscetível a ataques. PUFs (Physical Unclonable Functions) tentam contornar este problema usando como chave combinações de desafios e respostas aos desafios, sendo a resposta aos desafios uma saída a uma função gerada a partir de características físicas da fabricação do chip que não podem ser clonadas.

Objetivos:

Implementar usando o EPOSMote, um algoritmo capaz de extrair PUF's utilizando as características físicas da SRAM do circuito integrado.

Implementar um modelo onde seja possível um EPOSMote pedir um desafio a outro, e que seja possível verificar sua identidade a partir da resposta recebida e de um banco de dados.

Validar as implementações acima usando pelo menos dois EPOSMote, onde deve ser possível identificar cada um a partir das respostas recebidas, e o formato das respostas emitidas por cada EPOSMote devem ser sempre iguais.

Metodologia:

Primeiro será realizada uma pesquisa para obtenção de dados relevantes ao assunto, procurar algoritmos e soluções testadas e decidir a mais viável e possível de ser implementada usando o EPOSMote.

Após a definição de um algoritmo, será realizada a etapa de implementação do algoritmo e de verificação do padrão da SRAM do circuito integrado, na qual será definida uma solução para o problema de ruído existente e realização de correções de identificação.

Por fim será realizada a etapa de validação, onde a verificação das respostas originadas pelo mesmo circuito será realizada em conjunto com a comparação entre as respostas obtidas por circuitos diferentes.

Tarefas:

1. Pesquisar informações relevantes ao tema, soluções.
1.1 Pesquisa de material para referência
1.2 Pesquisa por problemas semelhantes
1.3 Pesquisa por soluções existentes

2. Implementar algoritmo.
2.1 Implementar a função de leitura da SRAM
2.2 Realizar coleta de dados, amostragem.
2.3 Fazer a normalização dos dados.
2.4 Implementar o sistema de desafios e respostas
2.5 Realizar autenticação entre dois EPOSMote

3. Ajustes no algoritmo.
3.1 Correção de erros
3.2 Otimização de código
3.3 Realização de testes de verificação

4. Apresentação
4.1 Demonstração dos resultados
4.2 Apresentação do funcionamento

Recursos:

- 4 EPOSMote III
- 1 Computador com Linux
- 1 JTAG

Entregáveis:

Cronograma:

E0 - Plano de Trabalho Inicial - (16/04)
E1 - Projeto Detalhado - (30/04)
E2 - Implementação I - (14/05) - Em progresso
E3 - Implementação II - (28/05) - Em progresso
E4 - Integração, otimização e validação - (11/06)
E5 - Apresentação dos resultados - (25/06)

Detalhamento dos entregáveis:

E0 - Rascunho do E1, contém as ideias básicas do projeto.
E1 - Pesquisa completa realizada, definições concretas sobre o que será implementado e realização de prova de conceito.
E2 - Primeira etapa de implementação onde será apresentado código em C da leitura do conteúdo da SRAM não inicializada e demonstração de unicidade, com normalização de dados para os desafios posteriores.
E3 - Segunda etapa de implementação onde será apresentado código em C do sistema de desafio e resposta da PUF e a autenticação entre os EPOSMote.
E4 - Realização de correções necessárias: solução para erros imprevistos, mudanças de modelo (caso necessárias) e validação de funcionamento após os ajustes.
E5 - Apresentação do trabalho, onde será entregue documento PDF do tema, com demonstração do que foi implementado e como funciona.

Planejamento:

Definições básicas:

Physical Unclonable Function:

Uma Physical Unclonable Function (PUF) é uma função que:

- É baseada em um sistema físico.
- É fácil de ser calculada (usando o sistema físico).
- É imprevisível mesmo para um atacante que possua acesso físico.
- Sua saída aparenta ser aleatória.
- Dado um sistema físico, a saída da função sempre será a mesma.

Devido as variações que ocorrem durante o processo de fabricação de chips, não existem circuitos digitais fisicamente idênticos, assim podemos explorar este fato para criar "assinaturas" únicas de cada chip individual.

Dito isto, existem duas maneiras principais para explorar aleatoriedade com PUF's:

- Usando aleatoriedade explícita : O desenvolvedor que insere no circuito algum método de gerar aleatoriedade. Este método possui uma maior capacidade de distinguir um circuito digital de outro e tem variações de ambiente mínimas em comparação com aleatoriedade intrínseca, devido a possibilidade dos parâmetros serem diretamente controláveis e otimizados pelo projetista.

- Usando aleatoriedade intrínseca: Ao contrário da alternativa anterior, este método é mais instável, mas é mais atrativo ao desenvolvedor, devido ao fato de que pode ser incluído em qualquer design sem qualquer modificação no processo de fabricação do chip. E pode ser aplicado em chips já fabricados sem uso de PUF's previsto.

Existem diversas partes de um circuito digital que podem ser exploradas para geração de dados intrínsecos do circuito, nossa escolha para realização deste projeto foi a SRAM (Static Random Access Memory). Decidimos usar a SRAM do EPOSMote III para realização de PUF's usando a aleatoriedade intrínseca do circuito digital.

As duas partes que compõe a PUF implementada são:

- A parte física: o sistema físico que é muito difícil de clonar devido as incontroláveis variações que ocorrem durante o processo de fabricação, neste caso o EPOSMote III.
- A parte operacional: um conjunto de desafios D_i (estímulo) tem que ser disponibilizado no qual o sistema responde com um conjunto suficientemente diferente de respostas R_i.

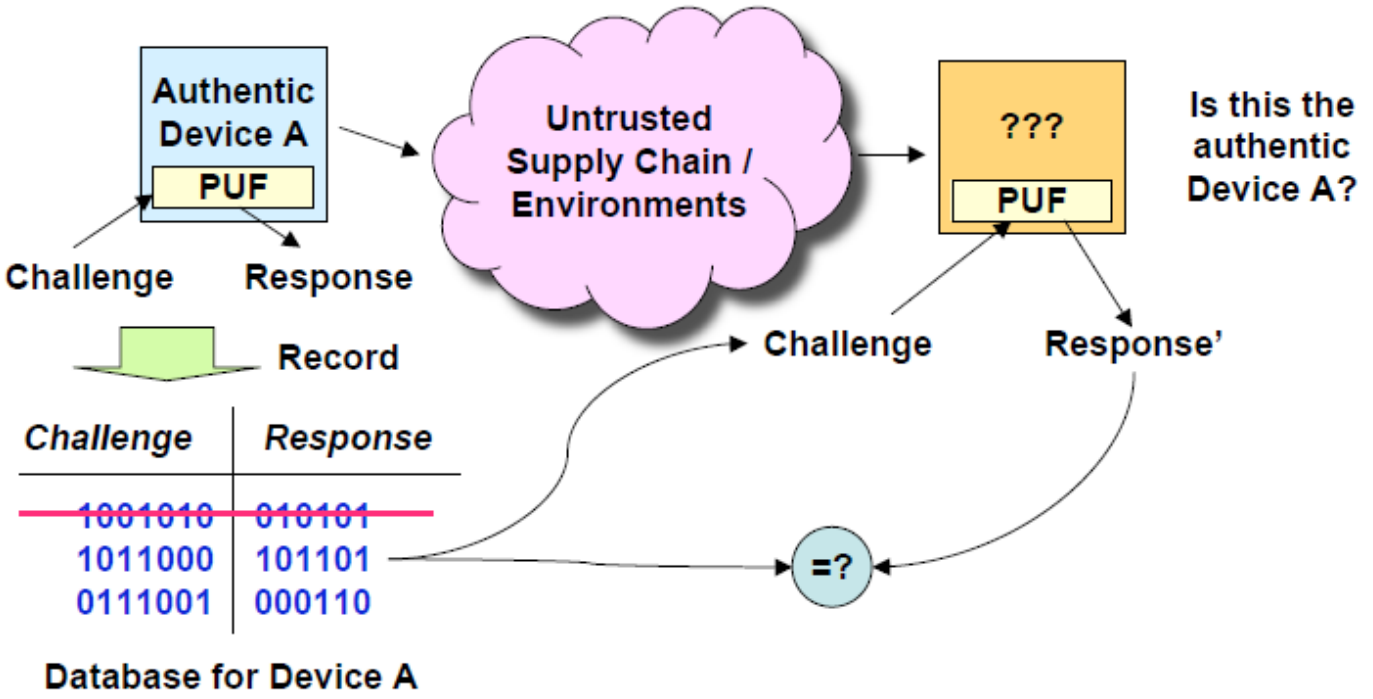
Realização de amostragem para a criação de chave única

Com o sistema físico em mãos, será realizada uma coleta de dados da SRAM do EPOS, antes dela ser escrita pelo circuito. Isso pode ser realizado reservando uma faixa da SRAM para extração da chave. Para tanto, será utilizado os Traits presentes no EPOS de forma que uma faixa de memória seja resguardada após a inicialização do sistema com seu valor original para que seja possível extrair o valor "lixo" presente ali. Será realizada uma amostragem deste "lixo" coletado da SRAM e será aplicada uma análise exploratória de dados usando estatística para detectar anomalias e determinar o desvio padrão de bits (se existente), isto é necessário antes de implementar o sistema de desafio, no qual um EPOSMote irá mandar uma requisição ao outro que apenas aquele pode responder, pois apenas ele possui o circuito digital capaz de gerar a resposta para o desafio.

Autenticação por desafio

Após a obtenção de um padrão de dados estável será criada uma função de desafios, onde um EPOSMote será capaz de desafiar o outro com um conjunto de perguntas e respostas pré-definido. Onde o desafiante sabe que apenas o circuito original tem a capacidade de responder o desafio usando sua PUF.

Modelo de Desafio:



A figura ilustra a autenticação baseada em PUF. Aqui nós exploramos que a PUF pode ter um número exponencial de pares de desafio/resposta, onde em cada um desses pares a resposta é única para um dado circuito digital e respectivo desafio. Também assumimos que é difícil duplicar o chip do circuito digital devido as variações incontroláveis do processo de fabricação.

Uma parte segura, quando possui um circuito integrado autêntico, aplica desafios aparentemente aleatórios para obter respostas imprevisíveis. A parte confiável armazena estes pares de desafio/resposta em uma base de dados para futuras operações de autenticação. Para verificar a autenticidade de um circuito integrado no futuro, o sistema seguro seleciona um desafio previamente salvo na base de dados, e obtém a resposta PUF do circuito integrado. Se a resposta coincide, o circuito sendo desafiado é autenticado, pois apenas aquele circuito seria capaz de responder tal desafio que só pode ser respondido com a sua assinatura elétrica intrínseca.

Prova de viabilidade

Foram realizadas medições nos valores da SRAM em EPOSMote diferentes usando o JTAG no LISHA. Estas medições confirmam que cada EPOSMote possui uma assinatura de bits diferente na inicialização da SRAM, tendo em mente que esta faixa de bits de cada EPOSMote possui variações em sua composição a cada vez que o circuito é reinicializado (verificado usando distância de Hamming).

A distância de Hamming entre duas strings de mesmo comprimento é o número de posições nas quais elas diferem entre si.

Segue o resultado das comparações:

Foram lidos 31KB da SRAM de cada um dos EPOSMote testados.
O primeiro número é o número do EPOS concatenado com o número da leitura realizada.
Ex: outfile11 = EPOSMote #1 no teste de leitura #1.

□□□□□□□□

Bit hamming distance between outfile11 and outfile12 ### 14185 ### File outfile11 length ### 253952 ### File outfile12 length ### 253952 ### Bit hamming distance between outfile11

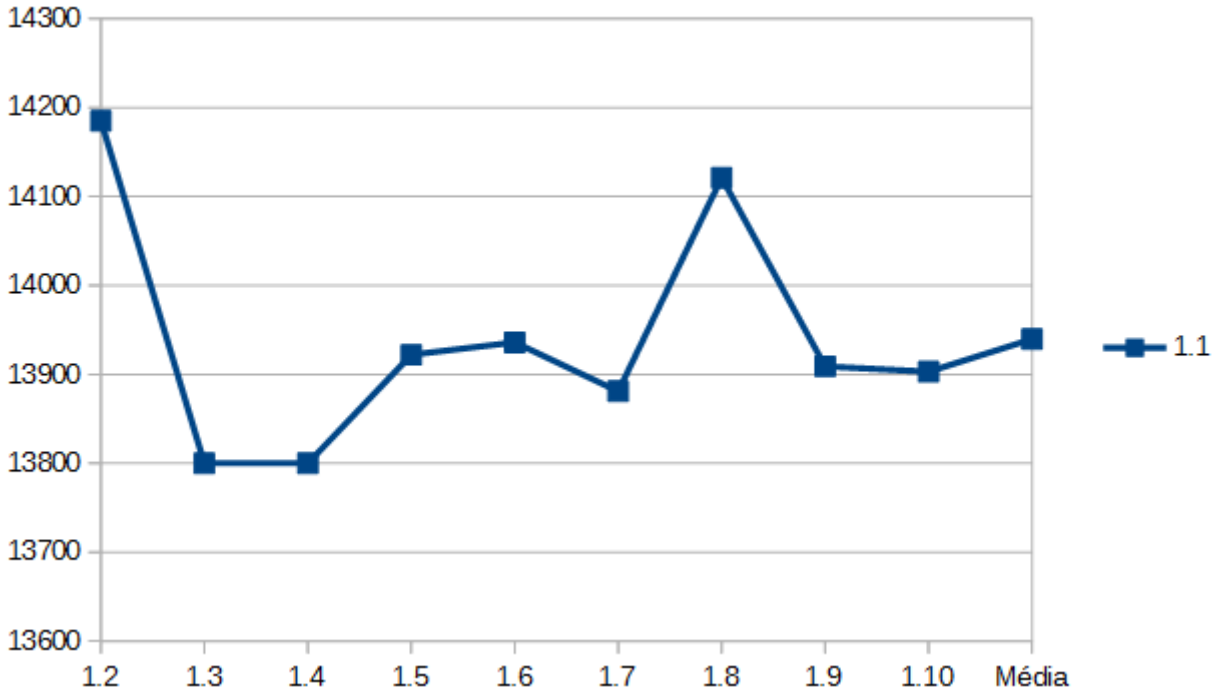
and outfile13 ### 13800 ### File outfile11 length ### 253952 ### File outfile13 length ### 253952 ### Bit hamming distance between outfile11 and outfile14 ### 13800 ### File outfile11 length ### 253952 ### File outfile14 length ### 253952 ### Bit hamming distance between outfile11 and outfile15 ### 13922 ### File outfile11 length ### 253952 ### File outfile15 length ### 253952 ### Bit hamming distance between outfile11 and outfile16 ### 13936 ### File outfile11 length ### 253952 ### File outfile16 length ### 253952 ### Bit hamming distance between outfile11 and outfile17 ### 13881 ### File outfile11 length ### 253952 ### File outfile17 length ### 253952 ### Bit hamming distance between outfile11 and outfile18 ### 14121 ### File outfile11 length ### 253952 ### File outfile18 length ### 253952 ### Bit hamming distance between outfile11 and outfile19 ### 13909 ### File outfile11 length ### 253952 ### File outfile19 length ### 253952 ### Bit hamming distance between outfile11 and outfile110 ### 13903 ### File outfile11 length ### 253952 ### File outfile110 length ### 253952 ### Bit hamming distance between outfile11 and outfile21 ### 106342 ### File outfile11 length ### 253952 ### File outfile21 length ### 253952 ### Bit hamming distance between outfile11 and outfile22 ### 106390 ### File outfile11 length ### 253952 ### File outfile22 length ### 253952 ### Bit hamming distance between outfile11 and outfile23 ### 106488 ### File outfile11 length ### 253952 ### File outfile23 length ### 253952 ### Bit hamming distance between outfile11 and outfile24 ### 106329 ### File outfile11 length ### 253952 ### File outfile24 length ### 253952 ### Bit hamming distance between outfile11 and outfile25 ### 106234 ### File outfile11 length ### 253952 ### File outfile25 length ### 253952 ### Bit hamming distance between outfile11 and outfile26 ### 106198 ### File outfile11 length ### 253952 ### File outfile26 length ### 253952 ### Bit hamming distance between outfile11 and outfile27 ### 106166 ### File outfile11 length ### 253952 ### File outfile27 length ### 253952 ### Bit hamming distance between outfile11 and outfile28 ### 106305 ### File outfile11 length ### 253952 ### File outfile28 length ### 253952 ### Bit hamming distance between outfile11 and outfile29 ### 106159 ### File outfile11 length ### 253952 ### File outfile29 length ### 253952 ### Bit hamming distance between outfile11 and outfile210 ### 106042 ### File outfile11 length ### 253952 ### File outfile210 length ### 253952 ### Bit hamming distance between outfile21 and outfile22 ### 13782 ### File outfile21 length ### 253952 ### File outfile22 length ### 253952 ### Bit hamming distance between outfile21 and outfile23 ### 13810 ### File outfile21 length ### 253952 ### File outfile23 length ### 253952 ### Bit hamming distance between outfile21 and outfile24 ### 13699 ### File outfile21 length ### 253952 ### File outfile24 length ### 253952 ### Bit hamming distance between outfile21 and outfile25 ### 13536 ### File outfile21 length ### 253952 ### File outfile25 length ### 253952 ### Bit hamming distance between outfile21 and outfile26 ### 13774 ### File outfile21 length ### 253952 ### File outfile26 length ### 253952 ### Bit hamming distance between outfile21 and outfile27 ### 13698 ### File outfile21 length ### 253952 ### File outfile27 length ### 253952 ### Bit hamming distance between outfile21 and outfile28 ### 13691 ### File outfile21 length ### 253952 ### File outfile28 length ### 253952 ### Bit hamming distance between outfile21 and outfile29 ### 13731 ### File outfile21 length ### 253952 ### File outfile29 length ### 253952 ### Bit hamming distance between outfile21 and outfile210 ### 13566 ### File outfile21 length ### 253952 ### File outfile210 length ### 253952

Como pode ser observado, diferentes medições da SRAM em um mesmo EPOS possui uma distância de Hamming menor que entre EPOS diferentes. Enquanto as medições entre o mesmo EPOS variavam entre 13000 e 14000 (entre 5 e 10%), medições entre EPOS diferentes chegaram perto de diferença de 50% entre os bits.

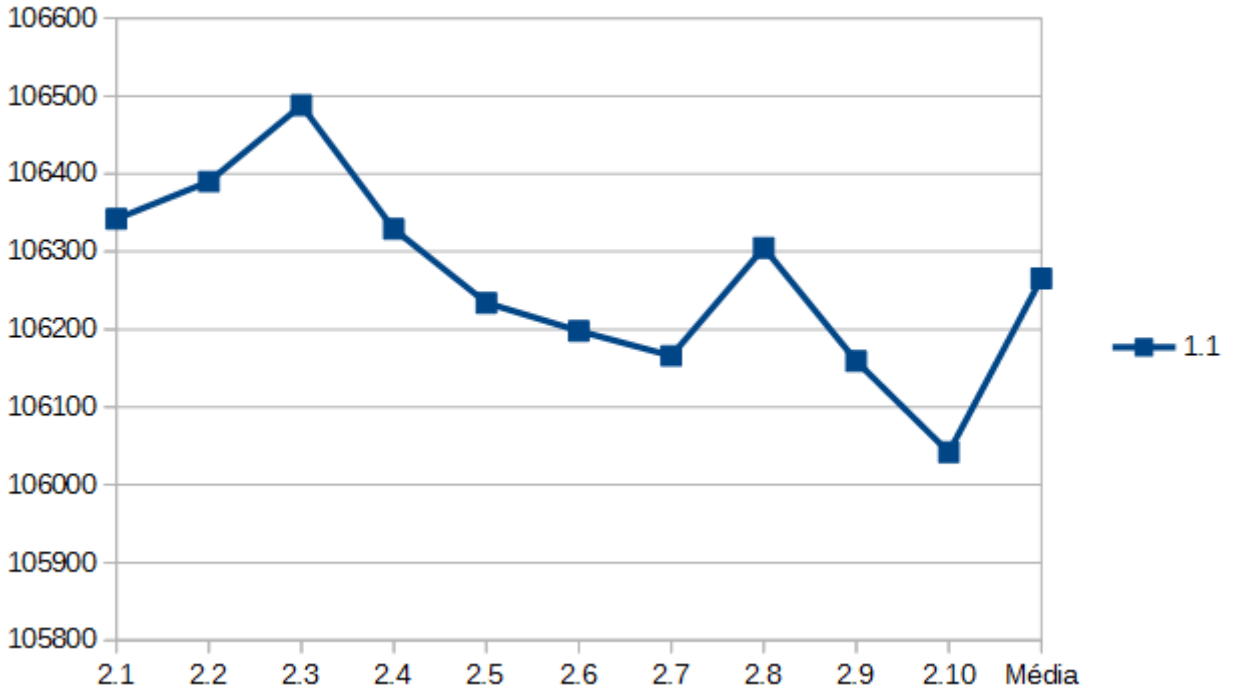
Seguem os gráficos demonstrativos:

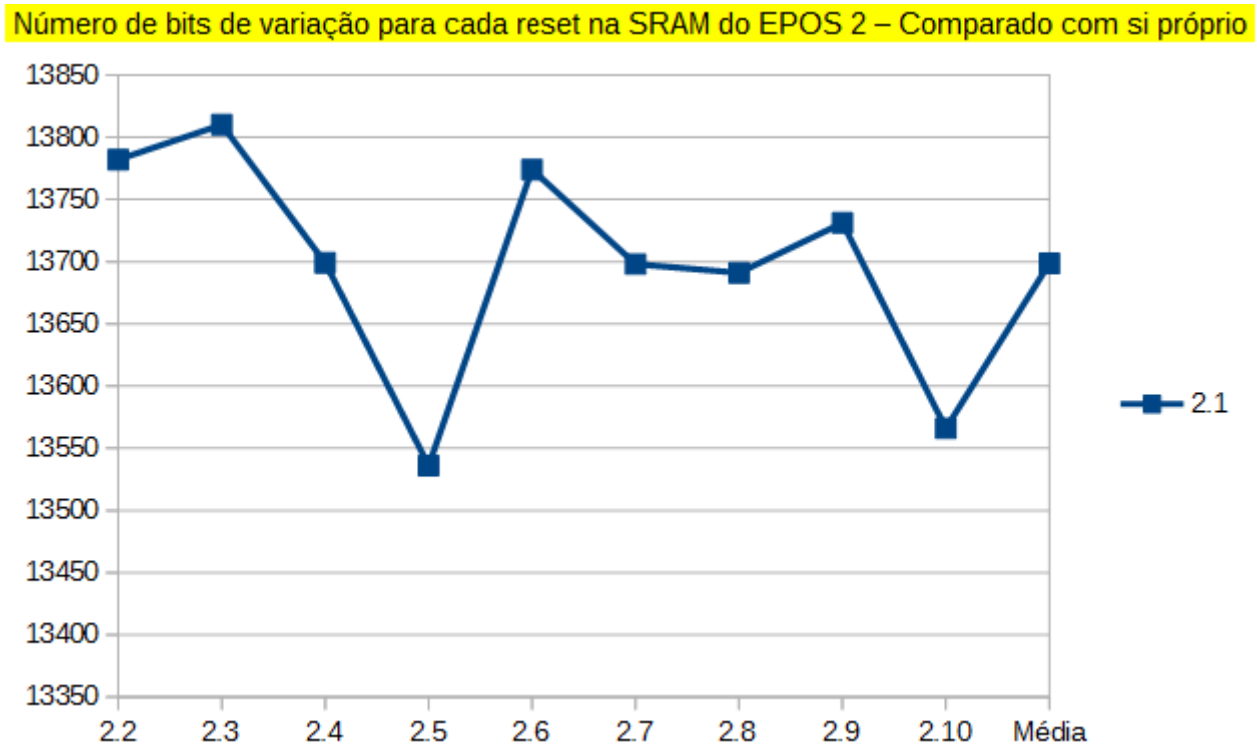
Número de bits:

Número de bits de variação para cada reset na SRAM do EPOS 1 – Comparado com si próprio

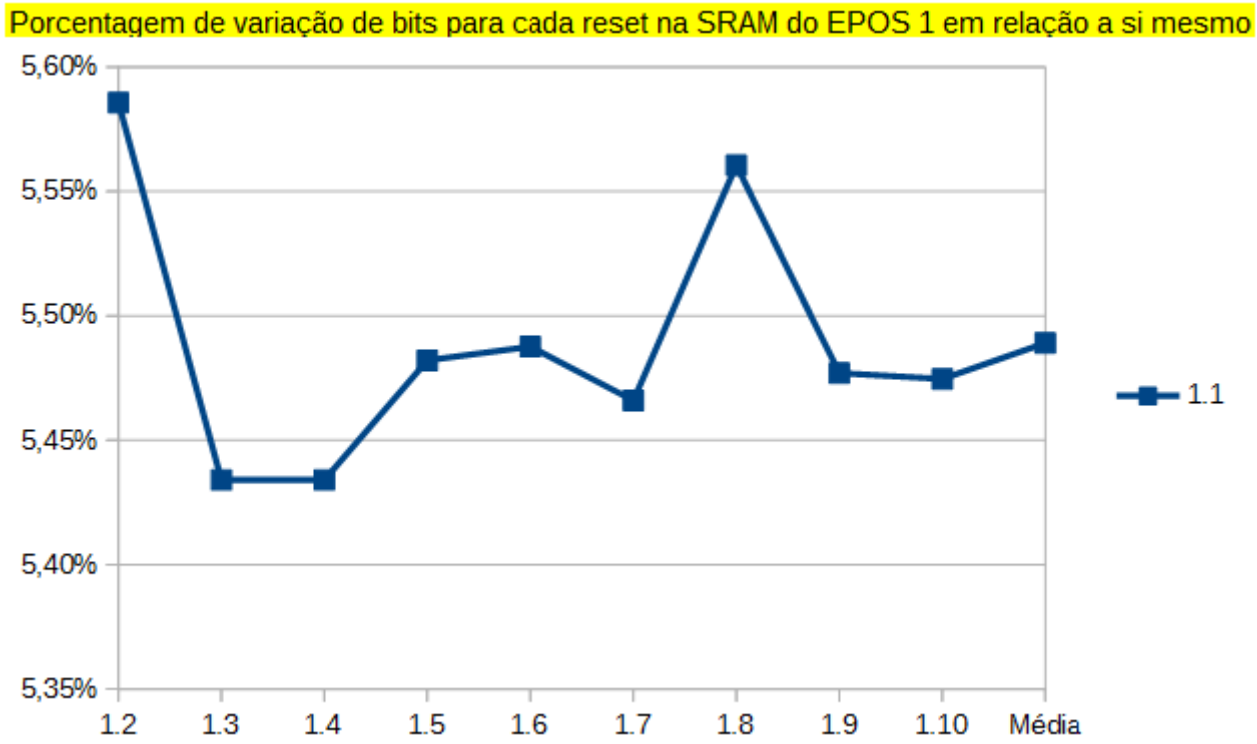


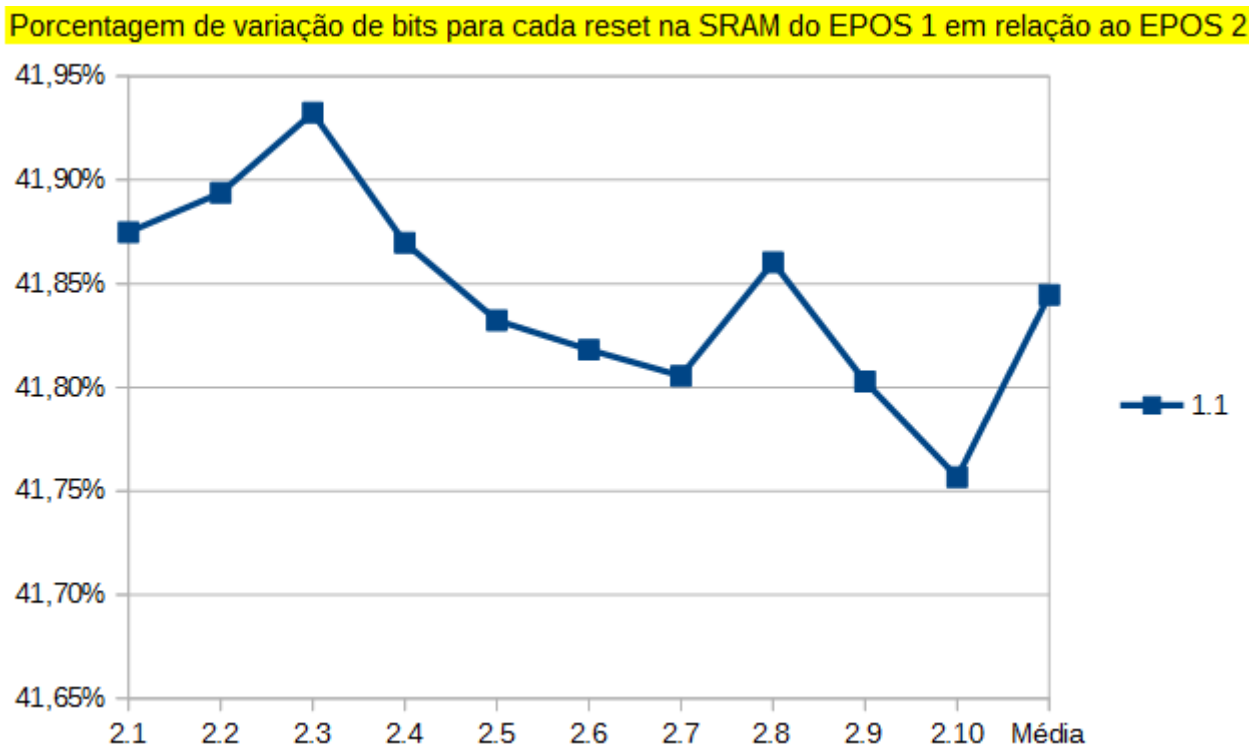
Número de bits de variação para cada reset na SRAM do EPOS 1 – Comparado com EPOS 2

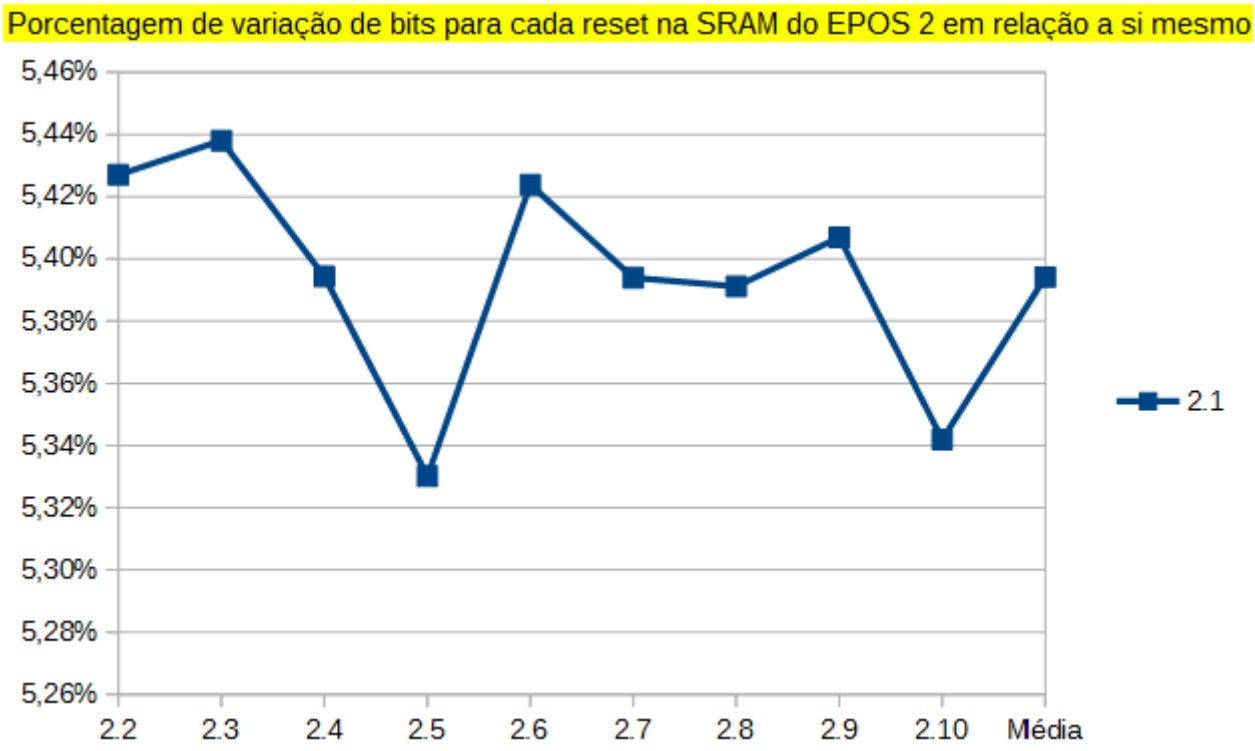




Porcentagem:







Implementação do projeto:

Desafio proposto:

O principal problema a ser resolvido, é gerar uma chave estável usando uma fonte de bits relativamente instável. Para a realização desta tarefa, é necessária uma maneira de estabilizar a chave para que esta possa ser usada por algum protocolo de autenticação. Neste caso é necessário um método em que, dada uma chave extraída da SRAM de um EPOSMote, possamos confirmar que esta chave é de fato pertencente à aquele EPOSMote específico. A solução para este problema foi encontrada na forma de um extrator fuzzy.

Extrator Fuzzy:

Um extrator fuzzy funciona da seguinte maneira: primeiro ele criará uma média dos dados com realização de várias leituras da SRAM, para que depois possa ser montada uma única chave sem ruído. Esta leitura é então usada para criar dois elementos de informação: um valor hash e um secure sketch. O valor hash é gerado primeiro, quando a leitura sem ruído é enviada para uma família de funções de hash. O resultado é então passada em uma segunda função criptográfica para produzir o valor hash final. O secure sketch é gerado usando uma construção code-offset, que adiciona a leitura em uma palavra chave aleatória de um conjunto de códigos de correção de erro usando XOR bit a bit. Depois o secure sketch e o valor hash podem ser usados eficientemente para autenticar leituras com ruído da SRAM: primeiro fazendo uma subtração do secure sketch para produzir uma palavra chave com ruído incluso. Depois um esquema de correção de erro é usado para decodificar a palavra chave, que subsequentemente foi subtraída do secure sketch para produzir a leitura sem ruído original. Finalmente a leitura sem ruído passa pela função hash e é comparada ao valor hash original. O resultado pode ser usado para verificar a autenticidade de um dispositivo.

Além de providenciar uma maneira eficiente de correção de erros, o extrator fuzzy também permite que a privacidade seja mantida tendo apenas o hash e o esboço seguro armazenados no sistema de autenticação.

A família de funções de hash providencia o que é conhecido como "amplificação de privacidade", produzindo um conjunto uniformemente distribuído de valores que não são uniformemente distribuídos. Resumidamente: para cada produção realizada por esta função universal de hash, sempre existe um número igual de possíveis valores de entrada dos quais poderia ter sido derivada.

Algoritmo:

- Primeiro é feita uma série de amostragens da SRAM no campo reservado para a chave, cada amostragem é seguida de um reset completo do EPOSMote para que sua SRAM perca sua eletricidade.
- Com as amostras em mão, é realizada a normalização dos dados para obter a chave referência.
- Agora um número aleatório do mesmo tamanho da chave é gerado, e com ele é feito o código de correção de erro BCH. Resultando no número aleatório concatenado com o resultado BCH.
- Agora é realizado um XOR bit a bit da chave referência com o aleatório que foi processado pelo BCH. O resultado desta operação é o secure sketch.
- O secure sketch é enviado de forma aberta para o outro EPOSMote.
- Agora o outro EPOSMote, realiza um XOR bit a bit da leitura que ele vai retirar da SRAM dele com o secure sketch.
- O EPOSMote agora pega o resultado da operação anterior e aplica o código de correção de erro BCH. Considerando que o secure sketch é formado pelo xor entre os dados não normalizados e o número aleatório gerado, cada bit que diferir entre a leitura da SRAM e as leituras normalizadas irá também diferir entre o resultado desta operação e o número aleatório gerado inicialmente. Com a aplicação do BCH, caso o código seja capaz de corrigir t erros, isto é: a distância de hamming entre a leitura da SRAM e os dados normalizados for menor que um valor t, esta etapa conseguirá recuperar o valor aleatório gerado anteriormente.
- É aplicado um xor bit a bit da saída da etapa anterior com o secure sketch. Como este é formado pelo xor entre o número aleatório e a saída da SRAM normalizada, isto garante a recuperação dos dados normalizados.
- Por fim, é aplicado os algoritmos de Hash neste valor para geração da chave criptográfica.

Extração da chave da SRAM:

Foi reservada uma faixa na memória no final do espaço da SRAM, para que não seja inicializada pelo

bootloader do EPOSMote, e é ali que são extraídas as amostragens da memória para construção da PUF. Foi setado no arquivo de traits do EPOSMote que o máximo limite da memória fosse reduzido em 2048 bits (tamanho da chave).

```
... class PUF; template<> struct Traits<PUF> { static const unsigned int KEY_SIZE = 2048; static const unsigned int KEY_SIZE_BYTES = static_cast<unsigned int>((static_cast<signed int>(KEY_SIZE) - 1) + 8) / 8; }; template<> struct Traits<Machine>: public Traits<Machine_Common> { static const unsigned int CPUS = Traits<Build>::CPUS; static const unsigned int MEM_BEG = 0x20000004; static const unsigned int MEM_END = 0x20007ff7; // 32 KB (MAX for 32-bit is 0x70000000 / 1792 MB) // Physical Memory static const unsigned int MEM_BASE = MEM_BEG; static const unsigned int MEM_TOP = 0x20007ef7; static const unsigned int PUF_BASE = MEM_TOP + 1; //One past last byte static const unsigned int PUF_END = PUF_BASE + (static_cast<unsigned int>((static_cast<signed int>(Traits<PUF>::KEY_SIZE_BYTES) - 1) + 4) / 4) * 4; static const unsigned int FLASH_BASE = 0x00200000; static const unsigned int FLASH_TOP = 0x0027ffff; // 512 KB // Logical Memory Map static const unsigned int APP_LOW = 0x20000004; static const unsigned int APP_CODE = 0x00204000; static const unsigned int APP_DATA = 0x20000004; static const unsigned int APP_HIGH = 0x20007ff7; static const unsigned int PHY_MEM = 0x20000004; static const unsigned int IO_BASE = 0x40000000; static const unsigned int IO_TOP = 0x440067ff; static const unsigned int SYS = 0x00204000; static const unsigned int SYS_CODE = 0x00204000; // Library mode only => APP + SYS static const unsigned int SYS_DATA = 0x20000004; // Library mode only => APP + SYS static const unsigned int FLASH_STORAGE_BASE = 0x00250000; static const unsigned int FLASH_STORAGE_TOP = 0x0027f7ff; // Default Sizes and Quantities static const unsigned int STACK_SIZE = 3 * 1024; static const unsigned int HEAP_SIZE = 3 * 1024; static const unsigned int MAX_THREADS = 7; }; ...
```

Normalização dos dados:

Quando é realizada uma leitura no espaço de memória que foi destinado para a chave, esta leitura é salva para uso na criação da chave referência. Depois de uma leitura o EPOSMote é reiniciado, para que sua SRAM perca os valores e seja reiniciada para uma nova leitura. Este processo continua até uma quantia satisfatória de leituras sejam realizadas e salvas. Com todos os dados de amostragem em mãos basta aplicar a normalização para gerar a chave referência. Este algoritmo foi implementado no Linux devido a necessidade de memória para armazenar as diversas medições em memória para calcular a média.

```
... vector<uint8_t> calculate_array(const vector<vector<uint8_t>>& buffers) { if(!buffers.size()) { return vector<uint8_t>(); } vector<uint8_t> ret(buffers[0].size(), 0); auto sz = ret.size(); for(int i = 1; i < buffers.size(); ++i) { auto& buff = buffers[i]; if(buff.size() != sz) { return vector<uint8_t>(); } } for(int i = 0; i < ret.size(); ++i) { uint16_t count[sizeof(decltype(ret[0])) * 8]; for(int j = 0; j < sizeof(decltype(ret[0])) * 8; ++j) { count[j] = 0; } for(int j = 0; j < buffers.size(); ++j) { uint8_t value = buffers[j][i]; for(int k = 0; k < sizeof(decltype(ret[0])) * 8; ++k) { count[k] += ((value & (1 << k)) >> k); } } for(int j = 0; j < sizeof(decltype(ret[0])) * 8; ++j) { if(count[j] > (buffers.size() / 2)) { ret[i] |= (1 << j); } } } return ret; } ...
```

Resultados da normalização:

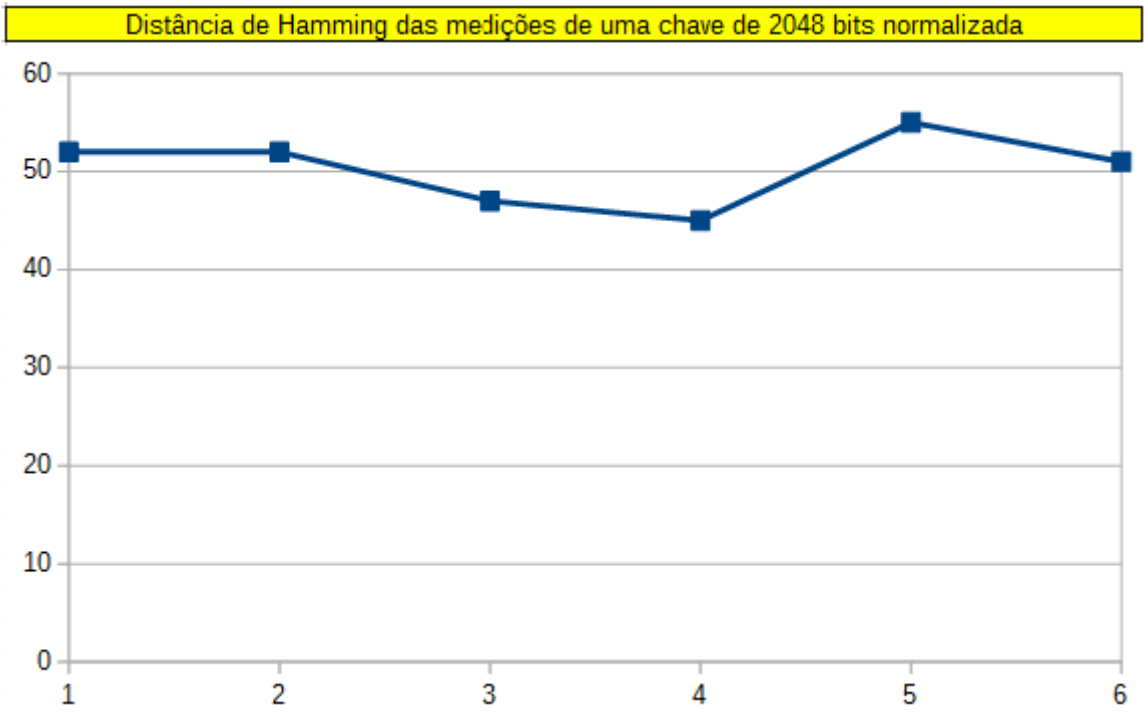
Para testar o algoritmo de normalização de dados, foram realizados alguns testes com uma chave de 2048 bits extraída da SRAM. Uma observação a ser levada em conta é de que quando se realizam amostragens da memória, é necessário esperar um minuto para o EPOSMote perder a energia que está no seu circuito se mais de uma amostra estiver sendo feita, para evitar que o valor amostrado seja igual ao anterior.

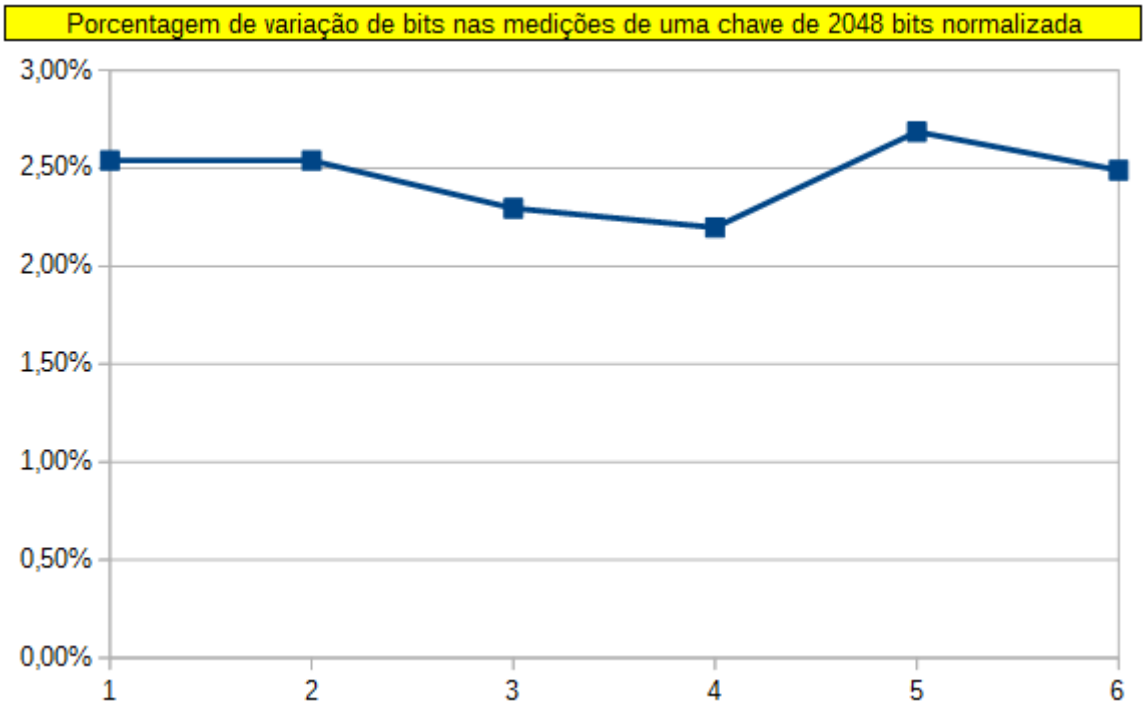
A seguir os resultados:

> Medições de diferença entre uma leitura sem normalização e leituras com normalização entre um mesmo EPOSMote: ### Bit hamming distance between epos_out_1 and standardized ### 52 ### Bit hamming distance between epos_out_2 and standardized ### 52 ### Bit hamming distance between epos_out_3 and standardized ### 47 ### Bit hamming distance between epos_out_4 and standardized ### 45 ### Bit hamming distance between epos_out_5 and standardized ### 55 ### Bit hamming distance between epos_out_6 and standardized ### 51 Medindo o segundo EPOSMote com o primeiro normalizado: ### Bit hamming distance between epos_out_1 and ../measure_epos1/standardized ### 635

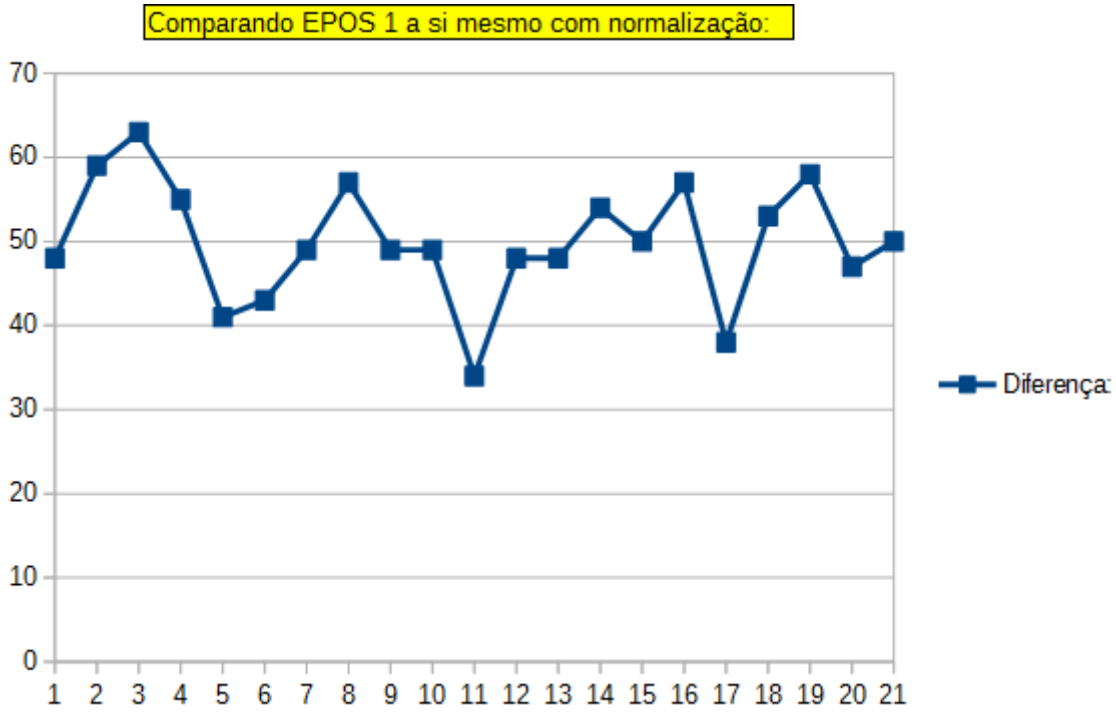
Podemos concluir que a normalização consegue reduzir bastante a faixa de variação dos bits das amostragens, na faixa de ~2,53% para o EPOSMote 1. Enquanto isso ainda possuímos uma variação de ~31% entre dois EPOSMote diferentes.

A seguir os gráficos demonstrativos:

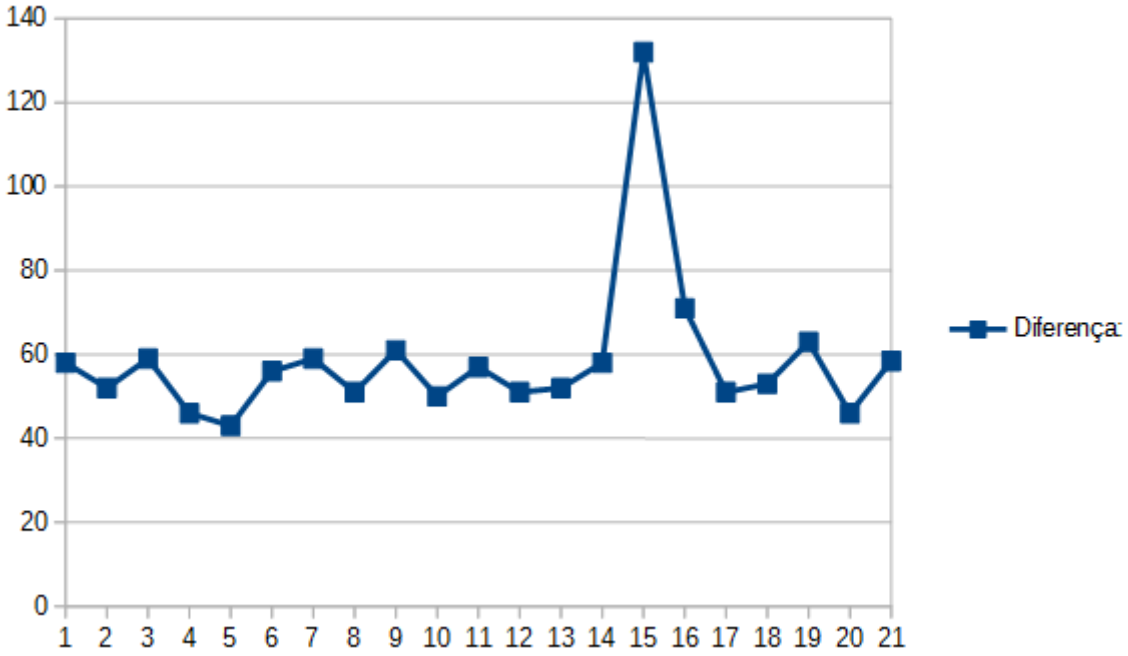




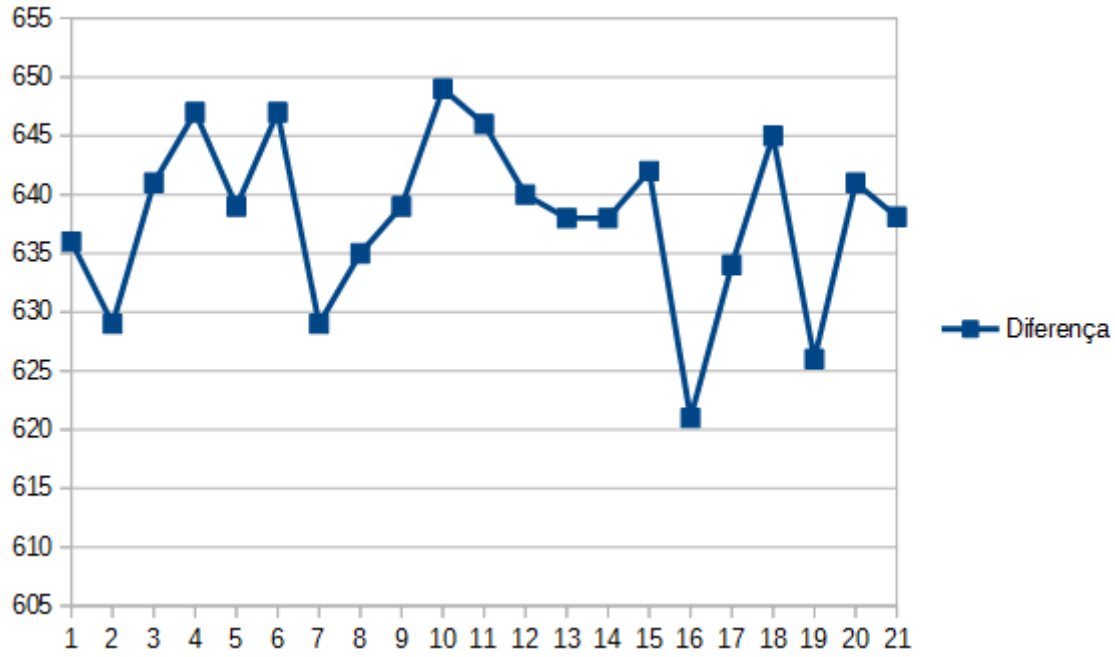
Foi realizada uma segunda rodada de 20 medições em dois EPOSMote, com chave de 2048 bits normalizada em cada EPOSMote. Os valores na posição 21 representam a média aritmética de todas as amostragens.

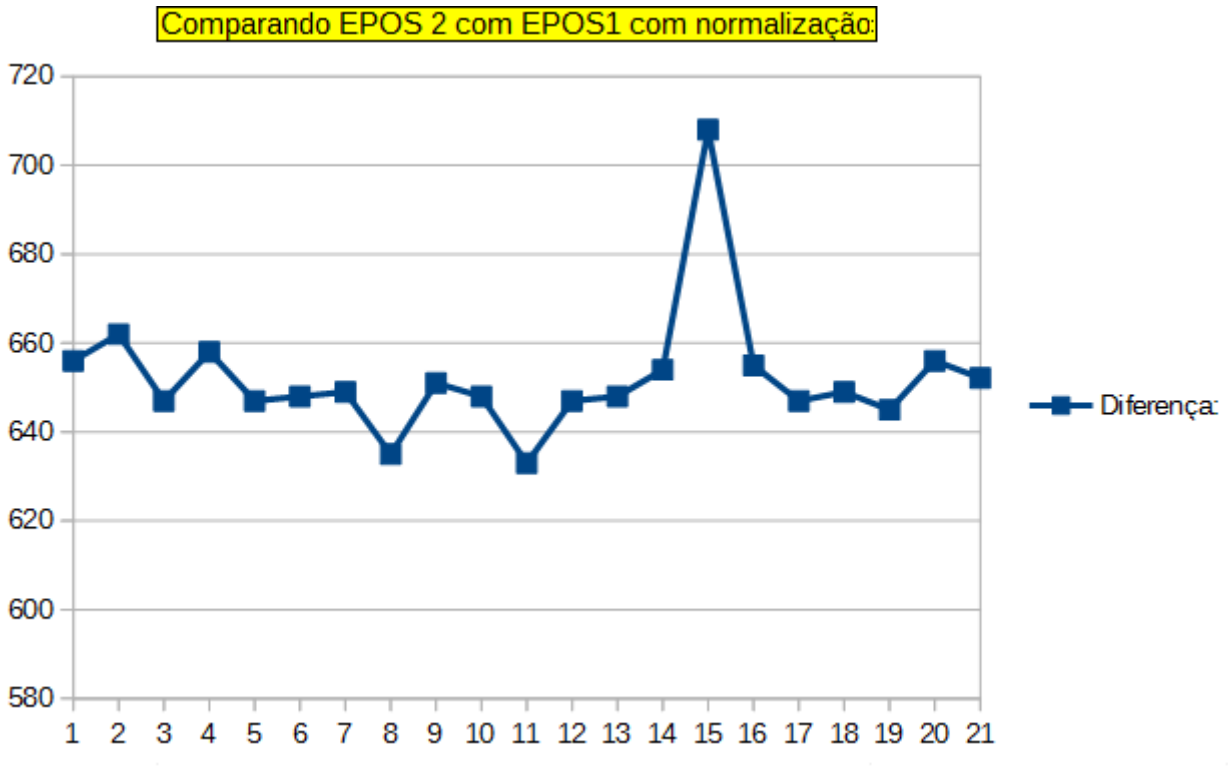


Comparando EPOS 2 a si mesmo com normalização:

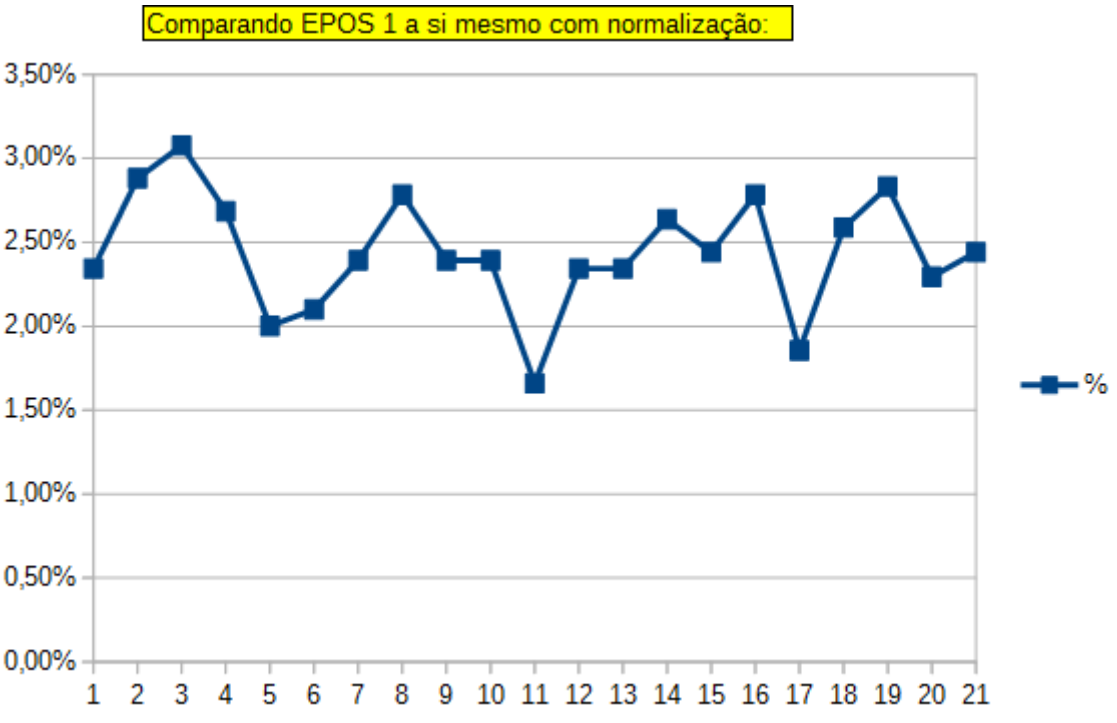


Comparando EPOS 1 com EPOS2 com normalização:

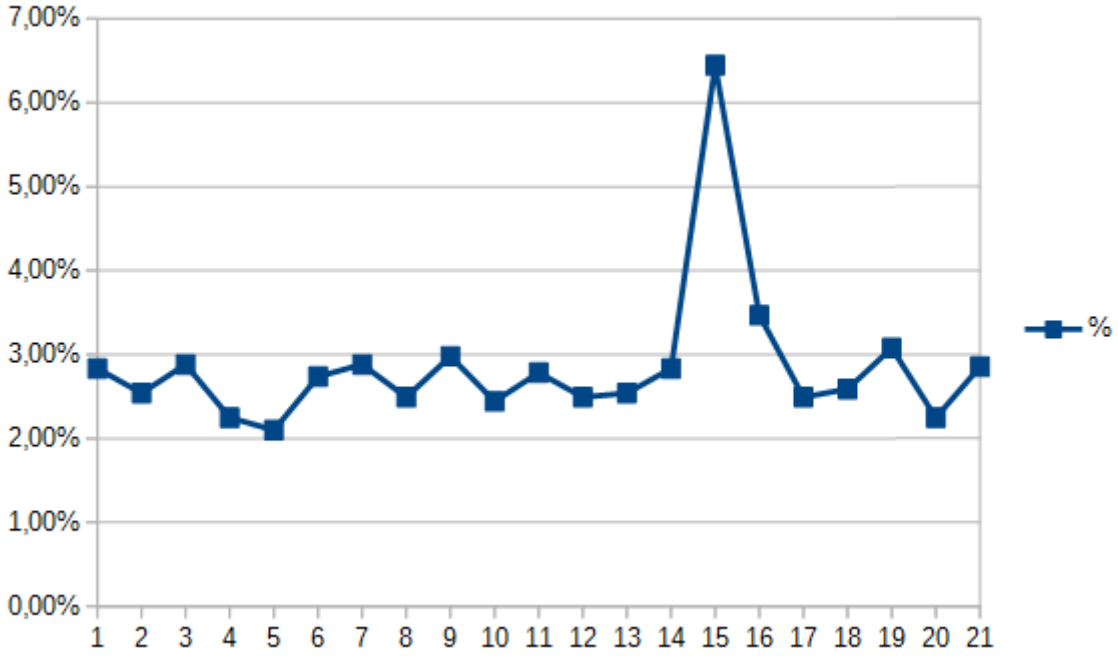




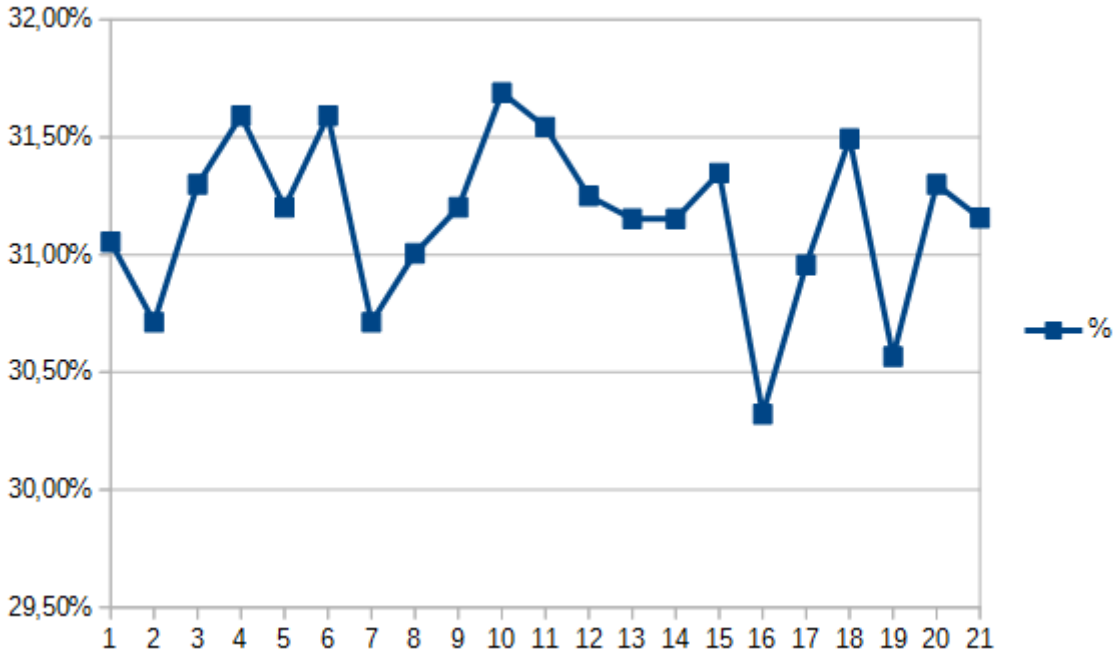
E a seguir os valores em porcentagem:

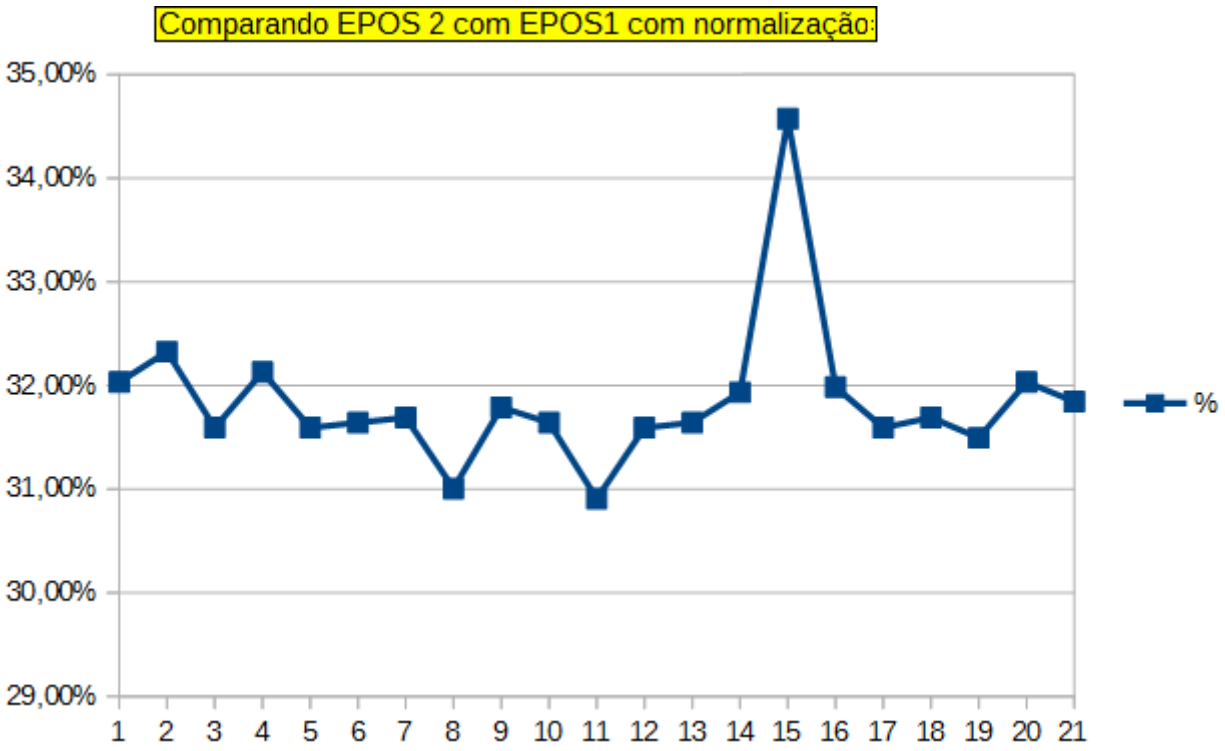


Comparando EPOS 2 a si mesmo com normalização:



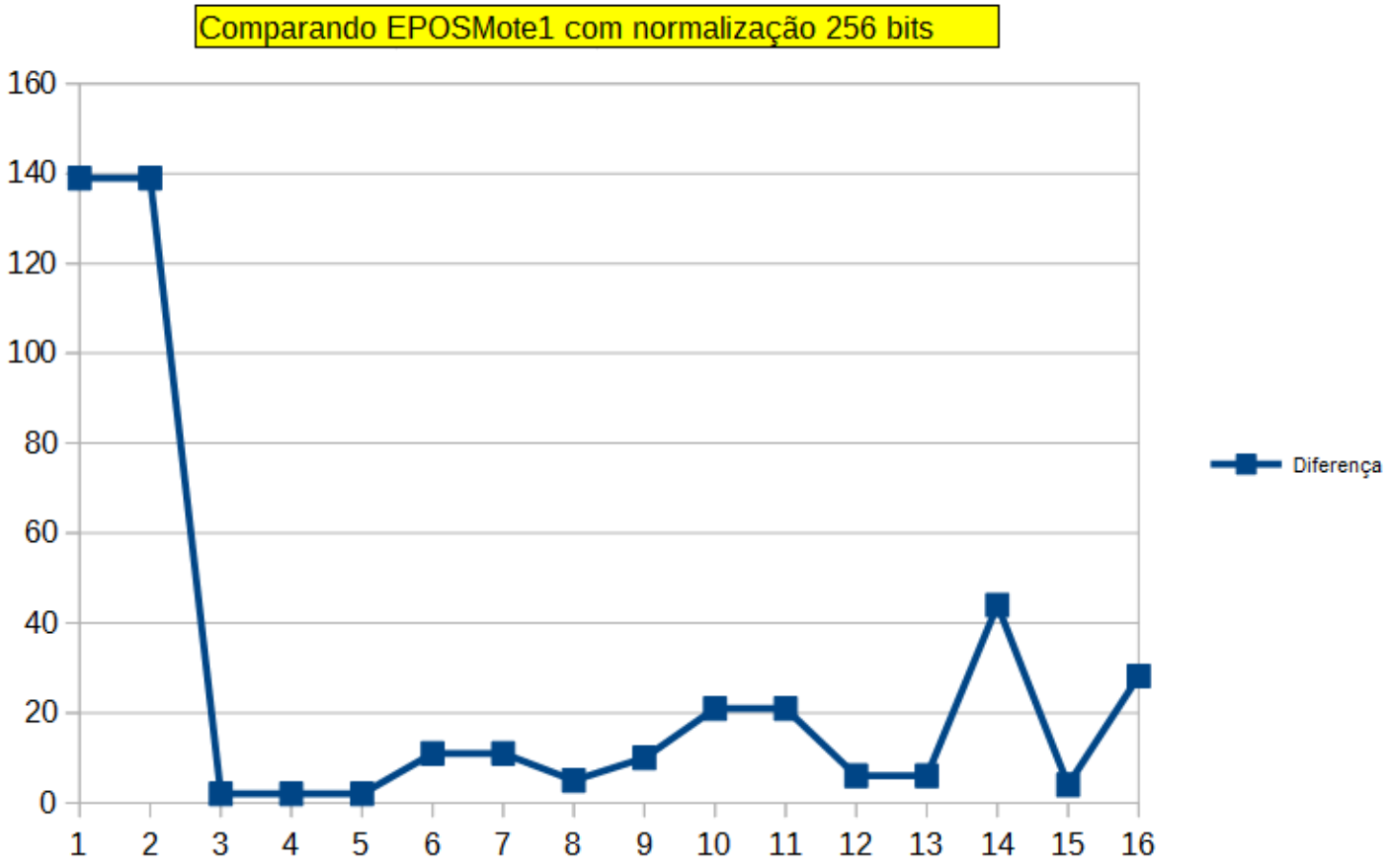
Comparando EPOS 1 com EPOS2 com normalização:

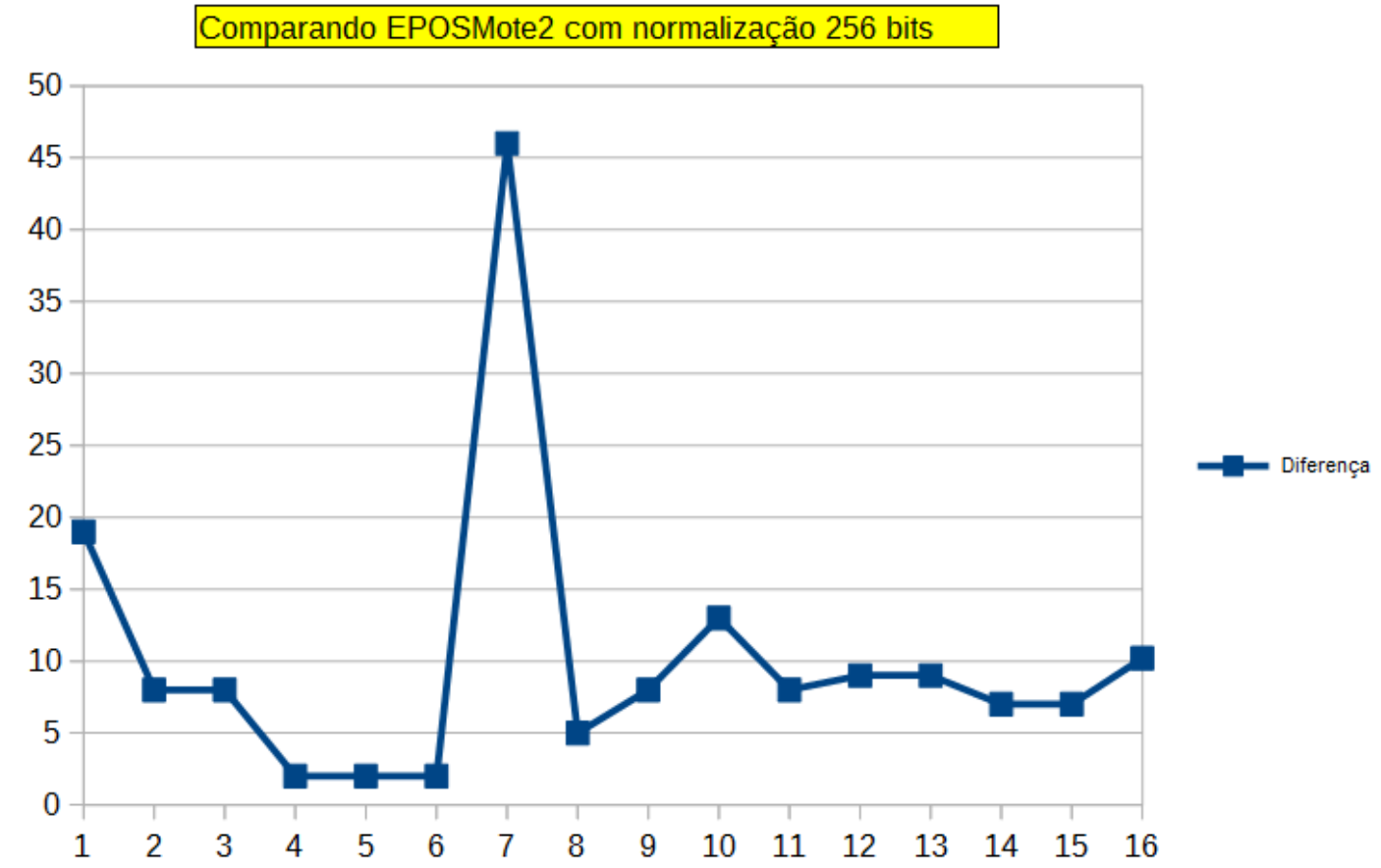




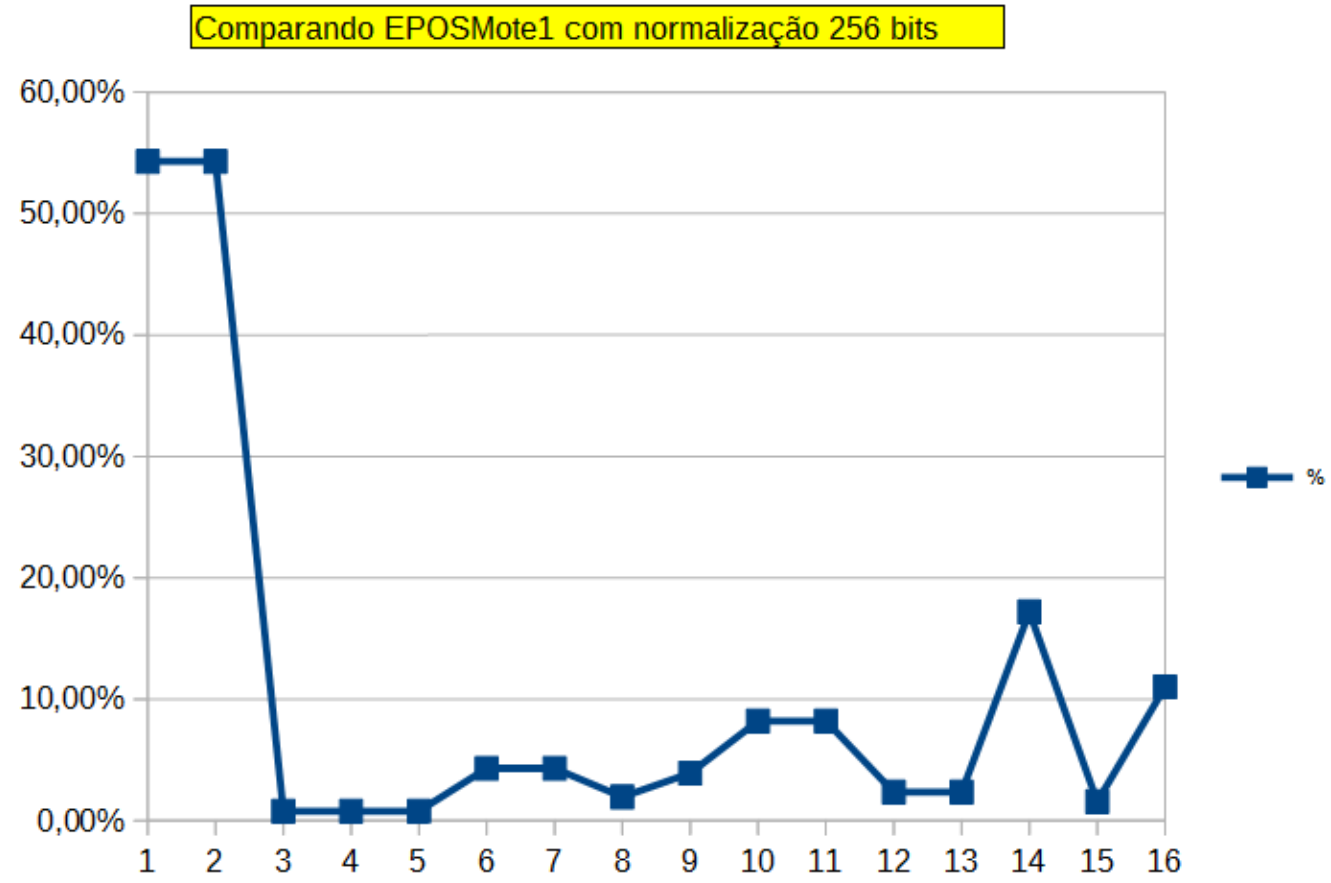
Como podemos observar nos gráficos:
O EPOSMote1 comparado com si mesmo possui uma variação de cerca de 50 bits em relação a chave de 2048 bits, ou 2,44% de variação.
O EPOSMote2 comparado com si mesmo possui uma variação de cerca de 58,45 bits em relação a chave de 2048 bits, ou 2,85% de variação.
Quando comparado o EPOSMote1 em relação ao EPOSMote2 temos uma diferença de 638,1 bits em relação a chave de 2048 bits, ou 31,16%.
Quando comparado o EPOSMote2 em relação ao EPOSMote1 temos uma diferença de 652,15 bits em relação a chave de 2048 bits, ou 31,84%.

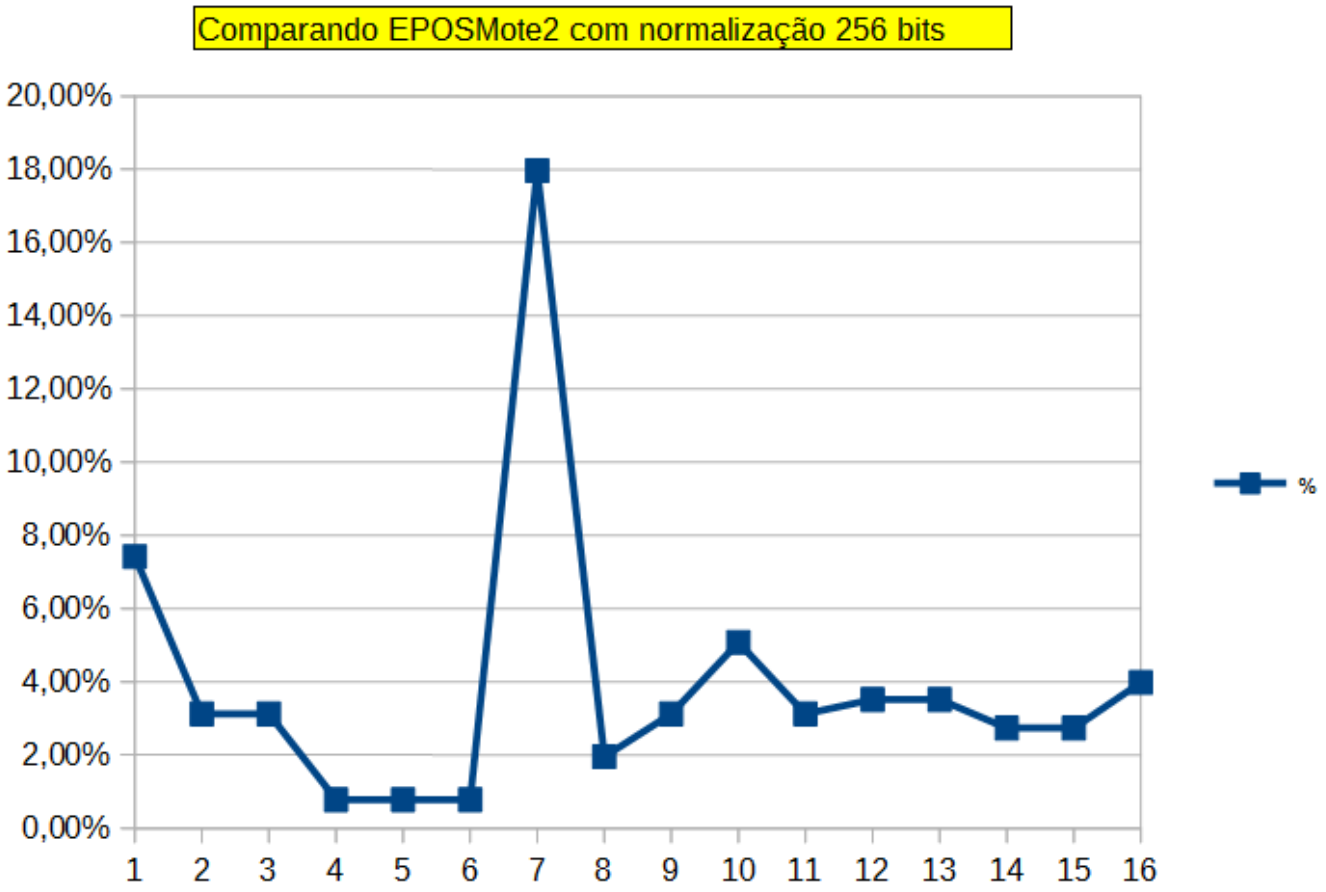
Para averiguar se estes valores estão relativamente consistentes, foi testada uma normalização de leituras com 256 bits de tamanho. A seguir os resultados de 15 medições em cada EPOSMote:





Em porcentagem:





É notória a diferença quando reduzimos o tamanho de chave, as instabilidades aparentes na faixa de bits ocorreram devido a medições sucessivas muito rapidamente. Valores repetidos na tabela também acontecem pela mesma razão. Mas nada impede de gerar uma chave útil para realização de comunicação, tudo o que ocorrerá é que a chance de repetir a autenticação seja um pouco mais alta, mas isto não causará perda de eficiência.

EPOSMote1 comparado com normalização, média de 28,2 bits de diferença em uma chave de 256 bits, ou

11,02% de diferença.
EPOSMote2 comparado com normalização, média de 10,2 bits de diferença em uma chave de 256 bits, ou 3,98% de diferença.

Correção de erro:

Corpo de Galois:

Antes de explicar o algoritmo é necessário um resumo sobre corpo de Galois. Um corpo finito ou corpo Galois é um corpo que possui um número finito de elementos. Tal como acontece com qualquer corpo, um corpo finito é um conjunto em que as operações de adição, subtração, multiplicação e divisão são definidas para satisfazer certas regras básicas.

Exemplo: O corpo finito GF(2) consiste dos elementos 0 e 1 que satisfazem as tabelas de adição e multiplicação a seguir:

+	0	1	*	0	1
0	0	1	0	0	0
1	1	0	1	0	1

Em campos de Galois mais complexos, a multiplicação envolve reduções por um polinômio irredutível definido sobre o campo. Para a implementação do algoritmo BCH, usamos um GF(256).

BCH:

O código de correção de erro implementado foi o BCH (Bose–Chaudhuri–Hocquenghem), que é um código de correção de erro cíclico que é construído usando polinômios sobre um campo finito (campo de Galois). Sua principal característica é que durante a operação, existe um controle preciso sobre a quantia de de erros que podem ser corrigidos pelo código. Outra vantagem também é a facilidade de decodificação, que é feita pelo método algébrico de decodificação de síndrome. Este design o torna propício para ser inserido em dispositivos eletrônicos de baixa potência.

Referências:

1. Physical Unclonable Functions for Device Authentication and Secret Key Generation

2. Open hardware-based and patent-free Physical Unclonable Function (PUF)

3. Physical Unclonable Functions and Applications

4. Modeling SRAM start-up behavior for physical unclonable functions

5. Wikipedia: Physical unclonable function

6. Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers

7. Wikipedia - Hamming distance

8. SRAM PUF Analysis and Fuzzy Extractor

9. Efficient Fuzzy Extraction of PUF-Induced Secrets: Theory and Applications

10. Wikipedia Hamming Code

11. Wikipedia BCH Code