Título: A First Step Towards Automatic Application of Power Analysis Countermeasures. **Bibliografia**: A first step towards automatic application of power analysis countermeasures; Bayrak, Ali Galip and Regazzoni, Francesco and Brisk, Philip and Standaert, François-Xavier and Ienne, Paolo; DAC '11 **Autor**: Erick Nogueira do Nascimento. **RA**: 032483

O trabalho propõe uma técnica para aplicação automática de uma contra medidas em software contra ataques de análise de potência em implementações em software de algoritmos criptográficos. A técnica proposta tem aplicação geral, mas os autores tomaram como exemplo uma implementação em software do algoritmo de encriptação simétrico AES executando em um microcontrolador AVR 8-bit e utilizando *peephole transformation* como proteção.

A técnica proposta consiste em 3 etapas:

1. Análise de vazamento de informação. As entradas desta etapa são o código assembly do programa e a plataforma de hardware. A saída consiste no código assembly original com uma anotação, onde cada instrução tem associado um valor de sensibilidade. Esta etapa é executada tomando-se o código assembly, executando-o e coletando os traços de potência (*power traces*) em alta frequência, resumindo um valor único de potência para cada ciclo (adotou-se o maior valor medido dentro de um ciclo). Este passo inicial é repetido para diversos pares (texto claro, chave) escolhidos aleatoriamente, e os valores de potência são gravados. A seguir, utilizando as informações gravadas no passo anterior, é computado para cada ciclo o valor de sua sensibilidade através da métrica de teoria da informação adotada: informação mútua entre a informação vazada (potência) e a chave simétrica empregada pelo algoritmo. Por fim, é associado a cada ciclo a instrução assembly executada naquele ciclo. Obs: os autores não deixam claro se este último passo pode ser feito em um processador pipeline também, já que várias instruções estarão executando simultaneamente em estágios diferentes.
2. Identificação das instruções que serão transformadas. Esta etapa recebe como entrada a saída da etapa anterior. É escolhido um limiar para o valor de sensibilidade, de modo que as instruções acima deste limiar são consideradas sensíveis, e portanto deverão ser protegidas.
3. Transformação do código. O código é transformado segundo a técnica *peephole transformation*, a qual consiste em realizar modificações locais no código, introduzindo assim instruções adicionais no datapath, com o intuito de modificar o padrão dos traços de potência. Segundo os autores, para o microcontrolador AVR 8-bits utilizado, a inserção de instruções com operandos aleatórios antes e depois de cada instrução sensível (*random precharging*) é suficiente (e é a transformação adotada) para a proteção do código, pois o consumo de potência em tal plataforma depende do valor do dado que está fluindo pelos fios, portas ou unidades funcionais do datapath, logo, este consumo também será aleatorizado.

Experimento. As medidas foram realizadas usando um PC, uma placa com o microcontrolador, um osciloscópio e uma sonda diferencial. Todos os componentes foram ajustados para reduzir ao máximo o ruído eletrônico. Todas as medidas foram realizadas 25 vezes e a média aritmética foi utilizada.

Os resultados do experimento mostram, conforme previsto, que o código protegido de fato fez com que todas as instruções (as do código original e as novas) ficassem com o valor de sensibilidade abaixo do limiar definido, diminuindo assim a informação mútua trazida pelo vazamento de potência.

Entretanto, como os autores argumentam, a *peephole transformation* com *random precharging* não garante segurança perfeita, mas eleva o esforço (em número de traços de potência) requerido para um ataque bem sucedido. Para uma avaliação mais precisa do esforço que o atacante teria para conseguir realizar o ataque contra o código protegido, os autores realizaram ataques DPA (*differential power analysis*) baseados em correlação e concluíram que o número de traços de potência requeridos para um ataque bem sucedido é pelo menos 76 vezes maior do que a versão desprotegida.

No requisito desempenho, o código protegido apresentou um acréscimo de 126% no número de ciclos de execução em relação ao código desprotegido. Ainda assim, o código protegido levou apenas 64% do número de ciclos de clock de uma implementação que protege *todas* as instruções originais com o mesmo método de proteção.