Reimplementando o SMART: um Exemplo Prático de Verificação Remota para Dispositivos Embarcados

Gabriel Capella
Supervisor: Prof. Dr. Alfredo Goldman
Bacharelado em Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo
gabriel@capella.pro

Introducão

Devido à nova indústria de Internet das Coisas (*Internet of Things, IoT*), há uma demanda crescente por novos dispositivos conectados à Internet. Eles são responsáveis principalmente por sensoriar informações e atuar no ambiente.

Apesar de serem dispositivos simples, eles têm sido alvos de diversos ataques [4]. Quando sofrem um ataque, é difícil perceber que o mesmo ocorreu, pois muitas vezes o dispositivo não altera o seu funcionamento. Existem ataques que visam infectar o dispositivo para posteriormente utilizá-lo para cometer atos ilícitos e outros que visam alterar o seu funcionamento ou comportamento.

Identificar esses ataques é uma tarefa importante. Existe uma técnica conhecida como verificação remota (*Remote Attestation*) que provê uma solução para esse problema. Essa técnica verifica o estado interno do dispositivo remotamente. Entende-se por estado a configuração da memória do dispositivo em um dado momento. Ela pode ser implementada de diversas maneiras.

Para computadores existem soluções bem estabelecidas. No entanto, para dispositivos embarcados há pouca pesquisa na área.

Objetivos

O objetivo principal deste trabalho é implementar uma solução de verificação remota em um dispositivo. Além disso, realizar ataques nesse protótipo e ver se eles são corretamente identificados.

A implementação de verificação remota escolhida foi a sugerida no artigo *SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust* [2]. Ela foi escolhida, pois é referência na área e é utilizada como base para diversos artigos posteriores. Além disso, ela tem como foco manter o baixo custo desses dispositivos.

No processo de verificação remota existem dois elementos principais, o contestado e o verificador [3]. O contestado é o dispositivo inseguro sobre o qual queremos realizar a verificação remota. Já o verificador seria, por exemplo, o servidor que enviará desafios para o contestado com o objetivo de validar seu estado.

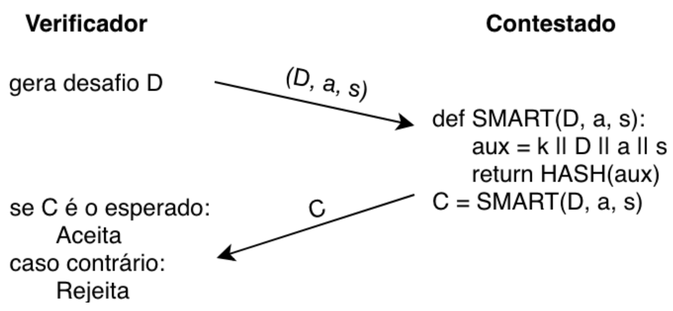

Figura 1: *Esquema do das informações trocadas durante o processo de verificação remota. Para simplificar o diagrama foi utilizado D para simbolizar o desafio, a para a posição inicial da memória, s para o números de bytes da memória para serem considerados a partir de a, k para a chave e C para o resultado da computação.*

Na implementação sugerida pelo SMART, o processo de verificação inicia-se com o verificador criando uma sequência de bytes aleatórias, chamado de desafio. Esse desafio junto com uma descrição de uma espaço de memória do dispositivo é enviado para o contestado. Ele deve pegar esses valores e calcular um *hash* deles concatenado com o conteúdo da região de memória descrita na solicitação. Esse valor calculado é enviado novamente para o verificador. O verificador sabe o estado (conteúdo da memória) que o dispositivo deve possuir, sendo assim é capaz de verificar se o valor calculado pelo contestado é correto.

Durante o artigo algumas suposições são consideradas. A primeira é que o dispositivo (contestado) não sofrerá nenhum ataque físico. A segunda é que devemos considerar que o seu software é totalmente violável. Ou seja, pode conter erros que possibilitem que um atacante mude ele por completo.

Para conseguir prover a verificação remota levando em consideração essas suposições é necessário que o dispositivo contenha uma função de *hash* inviolável - que não possa ser reprogramada e que quando chamada não possa ser desviada e nem interrompida. Além disso, essa função deve conter uma chave, a qual somente ela tem acesso. O principal objetivo dessa função e dessa chave é impedir que uma solicitação realizada pelo verificador seja forjada. Ou seja, o dispositivo não deve ser capaz de computar uma resposta que seja idêntica à esperada se não possuir as informações corretas.

Com esses elementos, o cálculo da função inviolável, chamado de código do SMART, será o *hash* do valor concatenada da chave, desafio, posição inicial da memória e número de bytes da memória a serem considerados. Como a função é inviolável, o atacante somente consegue mudar a entrada dela, o que resultaria em um valor diferente do esperado pelo verificador. Note que nessa disposição o verificador deve possuir a chave da função de *hash* para computar o resultado esperado. A figura 1 exemplifica esse processo.

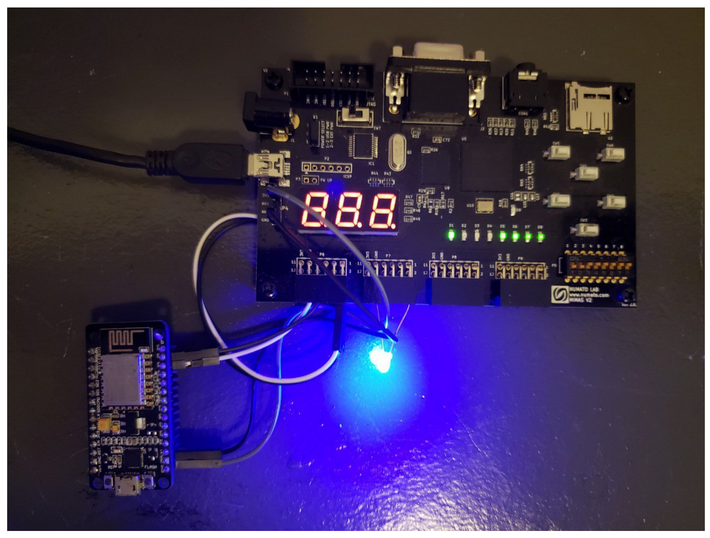

Figura 2: *Disposição final do dispositivo do protótipo. Na parte superior a placa de prototipagem Numato Mimas V2, contendo o FPGA Spartan 6. Na parte inferior o módulo de acesso a internet ESP8266 na placa Amica Node MCU. A comunicação entre eles é serial, utilizando o protocolo de comunicação RS-232. Nessa foto o cabo USB está ligado a um carregador de parede e o dispositivo está com o status do seu LED sendo controlado remotamente.*

Implementação

Para implementar um dispositivo com todas essas características, foi escolhido utilizar como base o microcontrolador openMSP430 [1], o qual foi contruído em um FPGA Xilinx Spartan 6 XC6SLX9. Além disso, o dispositivo foi conectado a um módulo ESP8266, tornando-o capaz de trocar informações com a Internet.

Para criar um código inviolável e uma chave só disponível para ele, foram criados dois módulos de acesso de controle de memória. Esses módulos foram escritos na linguagem de descrição de *hardware* Verilog e foram colocados no barramento de memória entre microcontrolador e sua memória. Ambos são iguais entre si, no entanto são configurados com parâmetros distintos. Na Figura 3 é possível a disposição final deles.

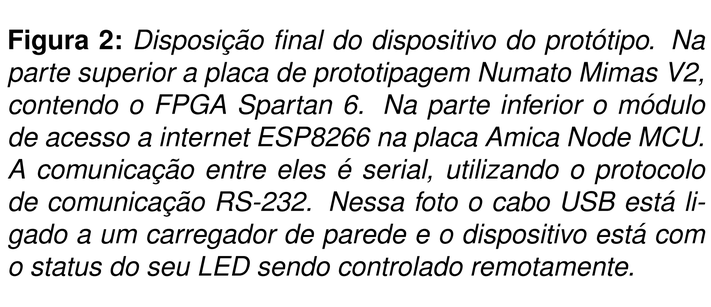

Figura 3: *Interface entre a memória ROM e o microcontrolador openMSP430. No meio, estão os módulos de controle de acesso à memória.*

A ideia básica desse módulo é observar constantemente o registrador contendo o ponteiro de instruções do microcontrolador e, dependendo do seu valor, liberar o acesso à uma região específica da memória. Caso haja uma tentativa de acesso à essa memória sem o ponteiro de instrução passar por uma posição específica, o dispositivo é reiniciado.

O primeiro módulo é responsável por proteger o próprio código do SMART. Ele impede que o código seja chamado parcialmente. O segundo, visa proteger a chave. Ela só se torna acessível se o código do SMART for chamado.

Outra propriedade importante do módulo é que ele possui um estado interno que caracteriza se o acesso à memória está ativo. Esse estado se torna ativo somente se o ponteiro de instrução aponta para a primeira instrução de uma função específica. Ele se torna inativo se o ponteiro sai dessa função.

Para programar o microcontrolador, foi necessário reescrever os arquivos de montagem (*Linker Scripts*) utilizada pelo GCC. Esses arquivos são responsáveis por descrever onde cada função, variável ou informação do programa deve ficar na memória.

Outra diferença em relação ao artigo original foi a utilização da função *hash* SHA256 implementada *hardware*.

Resultados

	Número of Slice Registers	Número of Slice LUTs
SMART sem o *hardware* do SHA256	1025	2492
SMART com o *hardware* do SHA256	2702	4764
Microcontrolador sem os módulos com o *hardware* do SHA256	2694	4710

Tabela 1: *Número de LUTs e registradores do FPGA dependendo da ativação dos módulos de controle de acessa a memória e do hadware do SHA256.*

Foi possível implantar o verificador e o contestado e testá-los com sucesso. Além disso, foram simulados alguns tipos ataques ao dispositivo, os quais foram identificados com êxito. Na Figura 2 é possível ver como ficou o protótipo final.

Na Tabela 1 existe uma análise dos recursos consumidos pela implementação do SMART. Apesar da função de cálculo de *hash* parecer causar um acréscimo significativo no tamanho do dispositivo, ele não o causa, pois quando utilizamos uma implementação de uma função *hash* em *hardware* podemos criar um dispositivo com uma memória menor - não há necessidade de salvar o código que computa a função *hash*.

Conclusão

Implementar e testar programas escritos em linguagem de descrição de *hardware* é uma tarefa relativamente simples. No entanto executar esses programas em um FPGA de verdade pode resultar em diversos erros.

Appesar deles, através dos testes e experimentos realizados pode-se concluir que introduzir a funcionalidade de verificação remota em dispositivos embarcados é uma tarefa simples que não envolve profundas modificações no hardware dos mesmos. Além disso, as modificações causadas pelo o acréscimo da função de hash são significativas para o custo final deles, idependentemente se é implementada em memória ou em *hardware*.

Referências

[1] *openMSP430*, 2018. https://opencores.org/project/openmsp430.

[2] Karim Eldefrawy, Daniele Perito, and Gene Tsudik. Smart: Secure and minimal architecture for (establishing a dynamic) root of trust. 01 2012.

[3] Aurelien Francillon, Quan Nguyen, Kasper B. Rasmussen, and Gene Tsudik. A minimalist approach to remote attestation. pages 1–6, 01 2014.

[4] A.-R Sadeghi, Christian Wachsmann, and Michael Waidner. Security and privacy challenges in industrial internet of things. 2015, 07 2015.