Mecanismo de Verificação de Integridade de Software Baseado em BIOS UEFI

Marciel de Liz Santos[1], Cesar A. Zeferino[1], Michelle S. Wangham[1],

[1]Universidade do Vale do Itajaí (UNIVALI) – Itajaí – SC – Brasil

marciel.dls@gmail.com, {zeferino,wangham}@univali.br

Abstract. *This paper aims to introduce a verification mechanism that takes advantage of UEFI BIOS resources to attest the integrity of embedded systems used in the Internet of Things. The proposed solution is composed of an application, called AVIS UEFI, which is executed in the Pre-Boot Applications phase and uses digital signature and keys stored in a cryptographic device to verify the software integrity. According to the result of the verification, the system is initialized or shut down. As a proof of concept, a prototype was developed and evaluated considering a real case study. The obtained results demonstrate the technical feasibility of the proposed mechanism.*

Resumo. *Este artigo descreve um mecanismo de verificação que aproveita os recursos do BIOS UEFI para atestar a integridade de software de sistemas embarcados utilizados na Internet das Coisas. Este mecanismo tem como componente principal uma aplicação, chamada AVIS UEFI, executada na fase de Aplicativos de Pré-Boot da inicialização que utiliza assinatura digital e chaves armazenadas em um dispositivo criptográfico para verificar se o software foi adulterado. De acordo com o resultado da verificação de integridade, o sistema é inicializado ou desligado. Como prova de conceito, um protótipo foi desenvolvido e avaliado considerando um estudo de caso real. Os resultados obtidos demonstram a viabilidade técnica do mecanismo.*

1. Introdução

Com os avanços tecnológicos dos sistemas embarcados, diversos objetos da vida real estão sendo conectados à Internet, dando origem a Internet das Coisas (*Internet of Things* – IoT), um paradigma cuja ideia central é a presença generalizada de uma variedade de objetos capazes de interagir e cooperar uns com os outros para alcançar objetivos comuns. As aplicações de IoT estão presentes em diversos domínios da vida humana, desde os de ordem pessoal e residencial até os referentes a ambientes empresariais e serviços de utilidade pública [Ara; Shah; Prabhakar 2016].

Como consequência da adoção da IoT, há uma previsão de grande crescimento da Internet nos próximos anos. Segundo pesquisas citadas por Fink (2015), a empresa de consultoria Gartner prevê 25 bilhões de dispositivos conectados à Internet em 2020 e a Cisco prevê 50 bilhões. Esse crescimento na quantidade e diversidade de dispositivos com sistemas embarcados conectados amplia a preocupação com a segurança. Esses dispositivos são mais vulneráveis e mais propensos a ataques, pois muitos deles se encontram expostos e sem segurança física, se comunicam por meio de redes sem fio e possuem recursos computacionais restritos que dificultam a implantação de mecanismos de segurança complexos [Abomhara; Koien 2015].

Com o avanço da eletrônica digital e a diminuição dos custos pela produção em escala, os sistemas embarcados vêm aumentando seu poder de processamento e suas funcionalidades ao incorporar novos componentes em sua estrutura interna. Assim, estão disponíveis microcontroladores de baixo custo e facilmente programáveis como Raspberry Pi e Arduino, que são amplamente utilizados para prototipação, e Galileo, que é usado para o desenvolvimento de aplicações para Internet das Coisas (IoT), e possuem recursos como BIOS (*Basic Input/Output System* – Sistema Básico de Entrada e Saída), sistema operacional, comunicação em rede e sistemas de armazenamento. Esses novos recursos conferem flexibilidade aos sistemas e permitem criar aplicações mais poderosas e para diversas finalidades. No entanto, geram novas vulnerabilidades e preocupações com a segurança, principalmente em situações em que o software não deve ser adulterado pelo usuário [Miritz 2016].

Um dos riscos associados à presença cada vez maior de sistemas embarcados e dispositivos conectados à Internet é o interesse de determinados indivíduos em comprometer a integridade dos dispositivos para obter benefícios de forma fraudulenta. Por exemplo, o proprietário de uma balança ou outro instrumento de medição envolvido em uma transação comercial pode desejar alterar seu software para fraudar as medições em seu benefício. Da mesma forma, softwares embarcados em automóveis podem ser alterados com o objetivo de obter acesso ilegal a funcionalidades não contratadas, como navegação por sistema GPS ou maior potência do motor. Assim, surge a necessidade de verificar se o software embarcado em um dispositivo está íntegro, ou seja, se é a versão do software que foi instalada pelo fabricante ou previamente analisada e aprovada por uma autoridade competente [Carmo e Machado 2009].

De acordo com Liu et al. (2015), a verificação de integridade de um software é fundamental para o correto funcionamento de um sistema, já que seus objetivos vão desde proteger os dados do próprio software e o sistema como um todo, até garantir ao usuário que o software em execução é legítimo. Entidades autorizadas devem ser capazes de verificar se softwares que são executados em dispositivos remotos não-confiáveis foram alterados por usuários maliciosos [Basile et al. 2012].

Sistemas embarcados mais complexos, com recursos como BIOS UEFI (*Unified Extensible Firmware Interface*), são mais flexíveis e permitem criar aplicações de IoT mais variadas e poderosas. Existem diversas abordagens para garantir a integridade de software em situações em que entidades homologadoras ou fabricantes não confiam no usuário. Essas abordagens, no entanto, apresentam limitações e não são apropriadas para sistemas embarcados complexos, conforme apresentado na Seção 2 [Liu 2015, Miritz, 2016].

A solução proposta neste artigo consiste em um mecanismo de verificação que utiliza o BIOS UEFI para atestar a integridade de softwares de sistemas embarcados utilizados em IoT. Esse mecanismo utiliza assinatura digital e deve ser configurado pelo fabricante do software ou por uma terceira parte confiável, que é a entidade responsável pela homologação do mesmo, pelo armazenamento das chaves em dispositivos criptográficos, como *USB Key*, e que protege o BIOS através de uma senha privada.

A principal contribuição deste trabalho é aumentar a segurança de sistemas embarcados utilizados em IoT mediante a verificação de integridade de software, por meio de um mecanismo simples e eficaz que é executado em cada inicialização do equipamento e impede seu uso em caso de adulterações.

O restante deste artigo é organizado em cinco seções. A Seção 2 discute trabalhos relacionados e posiciona o presente em relação ao estado da arte. A Seção 3 apresenta a arquitetura do mecanismo proposto para verificação de integridade de software. A Seção 4 descreve a modelagem do protótipo de avaliação do mecanismo, enquanto a Seção 5 apresenta o estudo de caso utilizado para avaliá-lo. Concluindo, a Seção 6 apresenta as considerações finais.

2. Trabalhos Relacionados

Carmo e Machado (2009) propõem um modelo para verificação de integridade de software embarcado baseado no conceito de reflexão, que utiliza análise de tempo de resposta como critério. A abordagem proposta não requer acesso externo às instruções do programa de execução e preserva a confidencialidade do código-fonte. Os autores aplicam o modelo de protocolos baseado em reflexão definido por Smith (1982) e, desta forma, verificam se o software é capaz de responder perguntas referentes a si mesmo dentro do tempo esperado. Essa abordagem permite detectar softwares adulterados por meio da observação do tempo de resposta.

A proposta de MeiHong e JiQiang (2009) consiste em uma abordagem para proteção de software baseada em *USB Key*. A solução utiliza verificação de integridade de software, por meio da autenticação inicial do usuário e de pontos de proteção, e autenticação mútua entre o software e o *USB Key* (armazenamento selado). Nesta abordagem, a comunicação entre o software e o *USB Key* é criptografada. O primeiro passo da solução é a verificação de integridade baseada na autenticação do usuário. O usuário deve inserir um PIN para que o *USB Key* verifique a integridade da informação dos pontos de proteção por meio do uso de um número de série seguro. Dessa forma, apenas usuários que conhecem o PIN e possuem o *USB Key* podem acessar o software. No segundo passo, o software gera um número aleatório que é usado para a autenticação mútua entre ele e o *USB Key*. Por fim, no terceiro passo, chamado autorização, o software inicia sua execução e, cada vez que um ponto de proteção é alcançado, solicita ao *USB Key* a chave K_i necessária para descriptografar o ponto de proteção i.

Carmo e Machado (2010) apresentam um modelo de verificação de integridade baseado em resumo criptográfico e tempo de resposta com duas possibilidades de execução de testes para conferir maior confiabilidade aos resultados. Na primeira alternativa, é realizada a avaliação sem a necessidade de interromper o funcionamento do equipamento em que a verificação é executada. Na segunda, é utilizada uma plataforma de verificação e são necessárias a interrupção e a remoção dos módulos que armazenam o software.

Zheng e Liu (2010) propõem uma abordagem para verificação de integridade de software baseada em um dispositivo criptográfico externo – *USB Key*. Essa solução utiliza o princípio de que mecanismos que armazenam e executam o software a ser verificado, o algoritmo de verificação e as chaves no mesmo dispositivo apresentam um ponto de falha único e podem ser facilmente atacados. Nesta solução, o software é dividido em diversos blocos e as chaves e os valores de *hash* de cada bloco são armazenados no *USB Key*. Para evitar a existência de um único ponto de falha, a comparação de tais valores também é realizada no *USB Key*.

A proposta de Castro et al. (2013) consiste em uma ferramenta para verificação de integridade de software em sistemas embarcados que é independente das tecnologias

e dos protocolos utilizados pelos mesmos. Para realizar a verificação, a ferramenta utiliza duas abordagens baseadas em desafio-resposta e no conceito de introspecção, segundo o qual o software é responsável pela leitura de seu próprio código. A primeira abordagem é a semente aleatória e consiste no envio de um dado (semente) que serve como chave criptográfica para a execução de um algoritmo de MAC (*Message Authentication Code*) sobre o próprio código do software em verificação. A segunda abordagem é a de intervalos aleatórios e é utilizada quando a primeira abordagem não é viável. Isso ocorre, por exemplo, em situações em que as plataformas nas quais os softwares são executados permitem apenas o cálculo de extrato de intervalos de memória com comprimento máximo definido.

Seshadri et al. (2016) propõem um método de verificação de integridade de software de sistemas embarcados para carros baseado na abordagem desenvolvida anteriormente pelos próprios autores, chamada SWATT (SoftWare-based ATTestation). Nesta abordagem, que utiliza um protocolo de desafio-resposta, um verificador externo atesta a integridade do software por meio da análise do código, dos dados estáticos e da configuração de hardware. Assim, o dispositivo verificado computa a resposta do desafio usando um algoritmo que pode estar previamente programado no dispositivo ou pode ser obtido via *download* antes da verificação.

A Tabela 1 compara o presente trabalho com os trabalhos relacionados, considerando se a solução: (*i*) dispensa inclusão de código no software a ser verificado; (*ii*) preserva confidencialidade do código; (*iii*) dispensa participação humana em cada verificação; (*iv*) garantia verificação periódica; (*v*) utiliza assinatura digital; e (*vi*) utiliza *USB Key*.

Todos os trabalhos analisados dependem da inclusão de código de verificação no software a ser verificado e, portanto, requerem confiança no fabricante do dispositivo. Esta característica não afeta a eficácia das abordagens quando o fabricante é o interessado em manter a integridade do software, porém as invalida nos casos em que existe uma terceira parte responsável por validar ou homologar o software. Basile et al. (2012) resolvem o problema da falta de confiança no fabricante mediante a verificação remota, o que torna o mecanismo mais complexo e caro.

Os três primeiros trabalhos não requerem que o verificador conheça o código do software a ser verificado e, portanto, preservam a confidencialidade e a propriedade intelectual e incrementam a segurança. Já as demais propostas exigem que o agente verificador conheça o código-fonte.

As abordagens de Carmo e Machado (2009), Carmo e Machado (2010) e Castro et al. (2013) não automatizam o processo e, portanto, requerem intervenção humana em cada verificação, o que incrementa os custos. Como consequência, essas abordagens não garantem a execução de verificações periodicamente. Elas são feitas apenas na implantação do sistema e quando há auditorias programadas ou motivadas por denúncia. Os outros trabalhos comparados propõem mecanismos automatizados em que a verificação é realizada em cada execução do software.

Todas as abordagens analisadas utilizam resumo criptográfico em combinação com chave simétrica para a verificação de integridade, porém não aproveitam as vantagens da criptografia assimétrica e da assinatura digital. Já as propostas de Zheng e Liu (2010) e MeiHong e JiQiang (2009) utilizam dispositivos criptográficos seguros para armazenar as chaves e proteger os mecanismos de verificação. O presente trabalho

destaca-se em relação aos demais por ser o único a cobrir todas as características analisadas. O mecanismo proposto, por sua vez, não requer da inclusão de código de verificação no software e preserva a confidencialidade do software a ser verificado. Requer intervenção humana apenas para configuração inicial do sistema e a verificação é feita de forma automática cada vez que o sistema é inicializado impedindo seu funcionamento caso o software não seja íntegro. Utiliza criptografia assimétrica e assinatura digital com apoio de dispositivos externo seguros para armazenamento de chaves.

Tabela 1. Análise comparativa dos trabalhos relacionados

Trabalhos	**Dispensa inclusão de código no software a ser verificado**	**Preserva confiden-cialidade do código**	**Dispensa participação humana em cada verificação**	**Garantia de verificação periódica**	**Utiliza assinatura digital**	**Utiliza *USB Key***
Carmo e Machado (2009)		X				
Meihong e JiQiang (2009)			X	X		X
Carmo e Machado (2010)		X				
Zheng e Liu (2010)			X	X		X
Basile et al. (2012)			X	X		
Castro et al. (2013)		X				
Seshadri et al. (2016)			X			
Este trabalho	X	X	X	X	X	X

3. Verificação de Integridade de Software Baseado em BIOS UEFI

O mecanismo proposto para verificação de integridade de softwares embarcados é baseado em uma aplicação incorporada à área de Aplicativos de Pré-Boot do BIOS UEFI. Ele realiza a verificação em tempo de inicialização do sistema por meio de assinatura digital e de chaves armazenadas em dispositivos criptográficos externos. Caso a verificação seja exitosa, o sistema é inicializado. Caso contrário, o equipamento é desligado.

Para um correto entendimento da solução proposta, considera-se neste trabalho que: (*i*) o fabricante, que projeta e fabrica o dispositivo, é quem instala o software a ser homologado; (*ii*) o homologador é a entidade que deve atestar a integridade do software instalado; e (*iii*) o usuário é a entidade que utiliza o dispositivo e pode ter interesse de modificar o software para obter benefícios ilícitos. Em algumas situações, o próprio fabricante pode ser o interessado em manter a integridade do software instalado e, portanto, nesses casos, fabricante e homologador são a mesma entidade.

A Figura 1 ilustra as entidades apresentadas e os componentes do mecanismo e como estes interagem durante o processo de verificação de integridade. Esses componentes são: (*i*) o software a ser verificado, que se encontra armazenado em memória secundária; (*ii*) a aplicação de verificação de integridade, chamada AVIS (Aplicação de Verificação de Integridade de Software) UEFI, que é executada na fase de aplicativos de Pré-Boot do BIOS e sob controle do mesmo; e (*iii*) um dispositivo criptográfico externo e seguro para o armazenamento da assinatura e das chaves necessárias para a verificação.

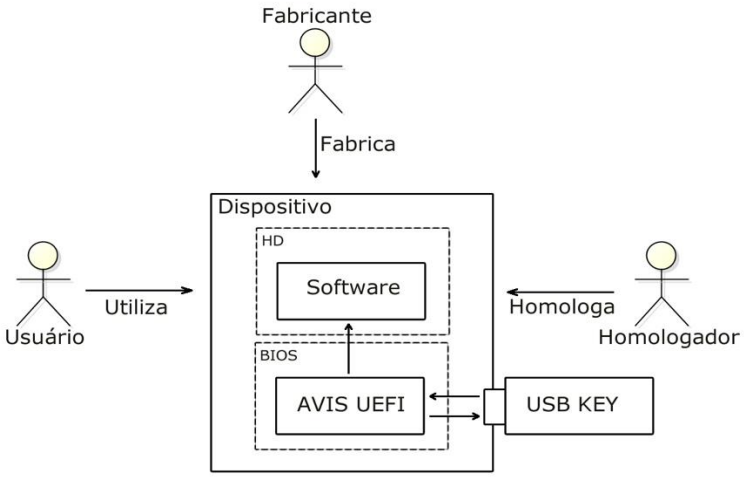

Figura 1. Componentes e entidades do mecanismo de verificação de integridade

A sequência de funcionamento e a interação entre os componentes do mecanismo são descritas nos passos a seguir:

1. O sistema embarcado é ligado e começa o processo de inicialização;
2. O módulo de Aplicativos de Pré-Boot do BIOS UEFI executa a AVIS UEFI;
3. A aplicação acessa o arquivo binário do software a ser verificado;
4. A aplicação calcula o *hash* do arquivo acessado;
5. A aplicação autentica o dispositivo criptográfico *USB Key*;
6. A aplicação lê a assinatura e a chave armazenadas no dispositivo *USB Key*;
7. A aplicação verifica a integridade do software. Para isso, descriptografa a assinatura com a chave pública para extrair o *hash* e o compara com o *hash* calculado no Passo 4;

Se a verificação é bem sucedida, o equipamento é liberado para uso e o sistema operacional é carregado. Caso contrário, o sistema é bloqueado e impedido de uso.

3.1. Fases de Operação

A operação do mecanismo compreende quatro fases. A primeira é executada pelo fabricante e inclui o projeto do sistema e a fabricação do dispositivo. A segunda consiste na configuração que é executada pelo homologador para implantar o mecanismo de verificação. A terceira fase é responsável pela verificação de integridade do software que é executada de forma automática pelo BIOS UEFI quando o usuário inicializa o sistema. Por fim, a última fase é a resposta da aplicação de verificação e

corresponde à ação executada para liberar ou bloquear o sistema conforme o resultado da verificação. Essas fases são nomeadas e descritas a seguir:

1. Fabricação: Nesta fase, o fabricante projeta o sistema e o submete ao processo de homologação. Depois de homologado o projeto, o fabricante produz cada instância do dispositivo e instala o software cujo código e funcionalidade já foram homologados. Com o dispositivo finalizado, solicita ao homologador a configuração para que o mesmo possa entrar em operação;
2. Configuração: Nesta fase, o homologador verifica se o software instalado no dispositivo corresponde ao que foi homologado durante o projeto na Fase 1. A seguir, o homologador assina o software instalado e configura o dispositivo criptográfico USB com a assinatura e os mecanismos de autenticação. Finalmente, instala a AVIS UEFI no BIOS UEFI, configura o *boot* seguro e protege o sistema com senha para impedir alterações indevidas;
3. Verificação: Nesta fase, o usuário inicia o sistema e a AVIS UEFI executa os Passos 3 a 7 descritos anteriormente; e
4. Resposta: Nesta fase, a AVIS UEFI executa o Passo 8.

3.2. Aplicação de Verificação de Integridade de Software

A AVIS UEFI, aplicação de verificação de integridade, deve ser incorporada à área de aplicativos de Pré-Boot do BIOS UEFI e protegida por *boot* seguro de forma a garantir sua integridade e execução em cada inicialização, além de uma senha privada. Conforme ilustra a Figura 2, esta aplicação será executada na fase de Carregamento de Sistemas de Transição, depois do disparo de *boot* e antes de carregar e passar o controle para o sistema operacional. Caso a verificação não seja bem sucedida, a inicialização será interrompida e o comando *shutdown* será enviado para impedir o uso do dispositivo.

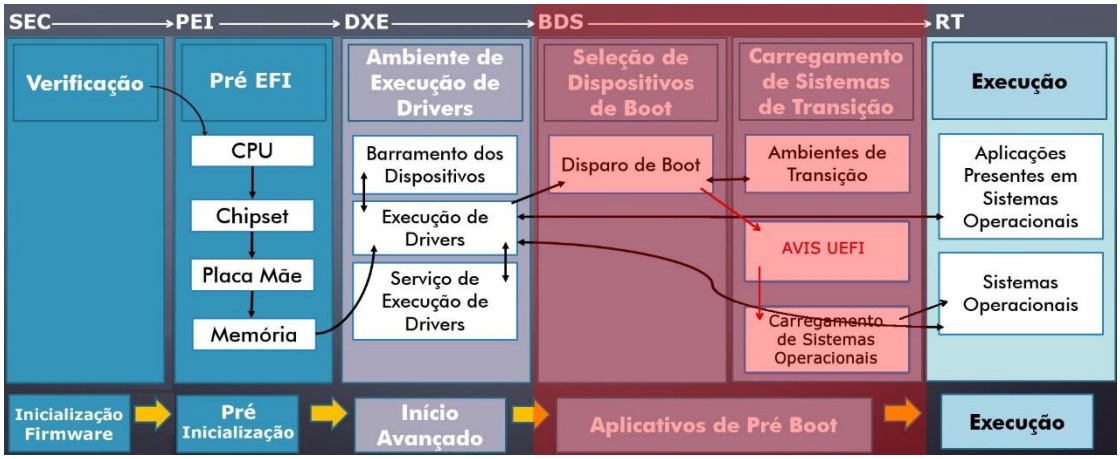

Figura 2. Processo de inicialização do BIOS UEFI com a inclusão da AVIS UEFI

A assinatura utilizada pela aplicação para verificar o software é gerada na fase de configuração por meio do cálculo do resumo criptográfico do mesmo com o

algoritmo SHA-256. Este, por sua vez, é criptografado com o algoritmo assimétrico RSA 2048 e a chave privada da entidade homologadora. No momento da verificação de integridade, a aplicação deve utilizar a chave pública do homologador para extrair o *hash* e compará-lo com o *hash* original.

Os algoritmos de criptografia foram escolhidos por serem os padrões considerados seguros pela comunidade e, como são executados apenas uma vez na inicialização do sistema, não impactam de forma significativa no seu desempenho.

A assinatura e a chave pública são armazenadas em um *USB Key* protegido por um mecanismo de autenticação baseado em PIN e que destrói seu conteúdo caso sejam feitas cinco tentativas incorretas. Assim, a aplicação deve conhecer o PIN do dispositivo e implementar os mecanismos necessários para estabelecer uma comunicação segura com o *USB Key*.

4. Protótipo

Como prova de conceito da solução proposta, um protótipo foi desenvolvido na linguagem de programação C, no ambiente de programação Eclipse 3.8 e com o auxílio do framework UDK versão 2017. Este framework permite implementar aplicações UEFI e possui em sua estrutura um módulo para cada tipo de serviço ou aplicação. O protótipo utiliza os módulos básicos MdePkg e MdeModulePkg, além dos módulos adicionais para criptografia e segurança CryptoPkg e SecurityPkg. Esses módulos permitem acessar os protocolos de serviço especificados pela interface UEFI que são agrupados por *Globally Unique Identifiers* (GUIDs) e estão armazenados em uma tabela chamada *handle database*. Nesta tabela, existe um protocolo específico para cada tipo de operação que se deseja executar, como consultar arquivos em disco.

O diagrama de atividades, mostrado na Figura 3, apresenta o comportamento dinâmico e as interações entre as entidades da solução proposta. A primeira função chamada pela aplicação é responsável pela abertura do sistema de arquivos NTFS e a segunda realiza a leitura do arquivo a ser verificado e o armazena em uma variável *array*. Este arquivo se encontra armazenado em memória secundária. A seguir, a aplicação chama a função responsável por gerar o *hash* do arquivo lido e armazená-lo em uma variável. Após a geração do *hash*, a aplicação chama a função responsável por ler a assinatura e a chave pública armazenadas no *USB Key*. A quinta função chamada pela aplicação é a função que extrai o *hash* da assinatura lida. Para isso, a aplicação descriptografa a assinatura com a chave pública. Finalmente, a aplicação chama a função que compara o *hash* gerado e o *hash* extraído da assinatura e fornece a resposta da aplicação. Se os *hashes* são iguais, a aplicação finaliza sua execução e permite a inicialização do sistema. Caso contrário, o dispositivo é desligado.

Os testes de integração para verificar o correto funcionamento do protótipo foram executados em um ambiente virtual baseado no emulador QEMU fornecido pelo *framework* UDK que disponibiliza uma máquina virtual com BIOS UEFI para a realização de testes.

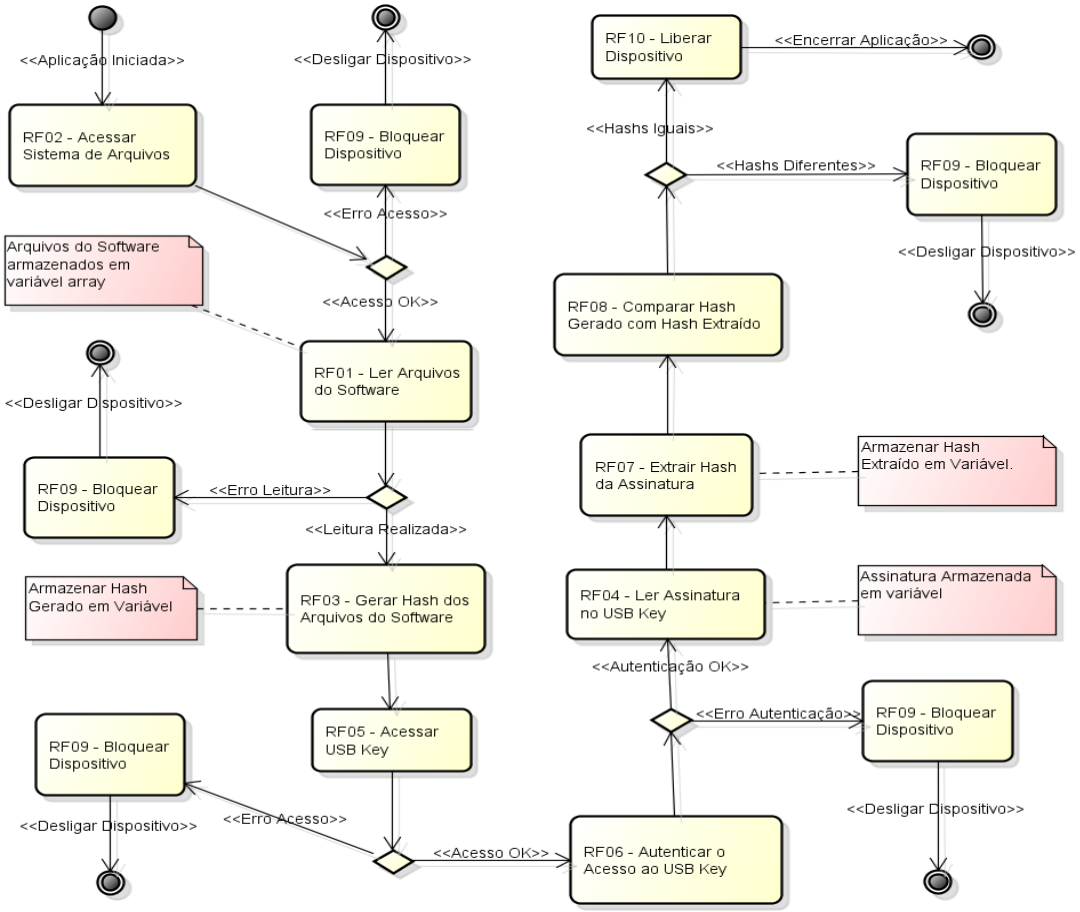

Figura 3. Diagrama de Atividades da AVIS UEFI

5. Estudo de Caso

O estudo de caso consiste na aplicação do protótipo nos simuladores de pista fabricados pela empresa Moss do Brasil e usados no projeto Cronotacógrafo do INMETRO (Instituto Nacional de Metrologia, Qualidade e Tecnologia[1]) com a finalidade de evitar fraudes e reduzir custos mediante a automatização do processo de verificação do software homologado.

O cronotacógrafo é o instrumento destinado a indicar e registrar, de forma simultânea, inalterável e instantânea, a velocidade e a distância percorrida pelo veículo no qual está instalado, em função do tempo decorrido, assim como os parâmetros relacionados com o condutor do veículo, tais como: o tempo de trabalho e os tempos de parada e de direção. Este instrumento é obrigatório em veículos que pesam mais de 4 toneladas ou que transportam mais de 10 passageiros ou carga perigosa, e deve ser aferido a cada dois anos em um ensaio metrológico que é realizado por empresas

[1] O INMETRO é o órgão do Sistema Nacional de Metrologia, Normalização e Qualidade Industrial (Sinmetro) que cria padrões e estabelece normas para diversos tipos de aplicações e projetos. Desta forma, publica editais para que fabricantes possam desenvolver equipamentos em conformidade e é responsável por homologar tais equipamentos e os softwares instalados nos mesmos.

autorizadas pelo INMETRO com o uso de um equipamento chamado simulador de pista [Inmetro 2016].

Os simuladores de pista são equipamentos utilizados para determinar se um cronotacógrafo funciona corretamente por meio da simulação de um percurso. Esses equipamentos dependem de um sistema embarcado com software que recebe dados dos sensores de movimento e compara os resultados com os registrados no cronotacógrafo. O projeto de um simulador de pista deve ser homologado pelo INMETRO e cada equipamento deve ser verificado antes de entrar em produção. Uma parte essencial dessa auditoria é a verificação de integridade do software instalado no equipamento.

O simulador de pista fabricado pela Moss do Brasil é composto de um banco de rolos principais e um banco de rolos auxiliares utilizados para captar o movimento dos pneus do veículo, além de um painel de comando, uma câmera de captura e um Tablet-PC. No banco de rolos principais, encontram-se os sensores, que estão ligados a um Controlador Lógico Programável (CLP) no painel de comando. O CLP transmite os sinais dos sensores a um Mini-PC, ao qual está conectado via USB. Este Mini-PC, também localizado no painel de comando, é o principal componente do simulador e é responsável por receber os sinais dos sensores e efetuar os cálculos das medições, gerando as informações de distância, velocidade e tempo. O Mini-PC é conectado via cabo de rede a um roteador WIFI e, através do sinal WIFI, se comunica com o Tablet-PC, que funciona como tela para visualização das operações e controle remoto para comandar o simulador.

A Figura 4 ilustra o processo da realização do ensaio metrológico de um simulador de pista. A Moss do Brasil fabrica o simulador de pista e o vende a empresas autorizadas pelo INMETRO a realizar ensaios metrológicos, chamadas postos de ensaio. O metrologista, que é um auditor do INMETRO, realiza a auditoria inicial para autorizar o funcionamento do equipamento. Finalmente, os postos de ensaio utilizam os simuladores de pista para validar o cronotacógrafo, que é o equipamento que registra velocidade, distância e tempo em determinados veículos.

5.1. Problema

Os auditores do INMETRO verificam os postos de ensaio nas seguintes situações: (*i*) quando esse órgão homologa um simulador de pista recém adquirido; (*ii*) em verificações periódicas programadas a cada dois anos; e (*iii*) quando há uma denúncia ou é detectada uma irregularidade. A principal limitação desse procedimento é que não impede que, após a avaliação inicial, o software seja modificado e que o estabelecimento utilize uma versão diferente da homologada. Essa situação somente seria detectada pelo INMETRO em caso de denúncia. Outra limitação dessas auditorias do INMETRO é que o processo de verificação de integridade de software nas inspeções periódicas é trabalhoso, uma vez que há a necessidade de romper lacres físicos do equipamento para conectar dispositivos periféricos para a realização de testes.

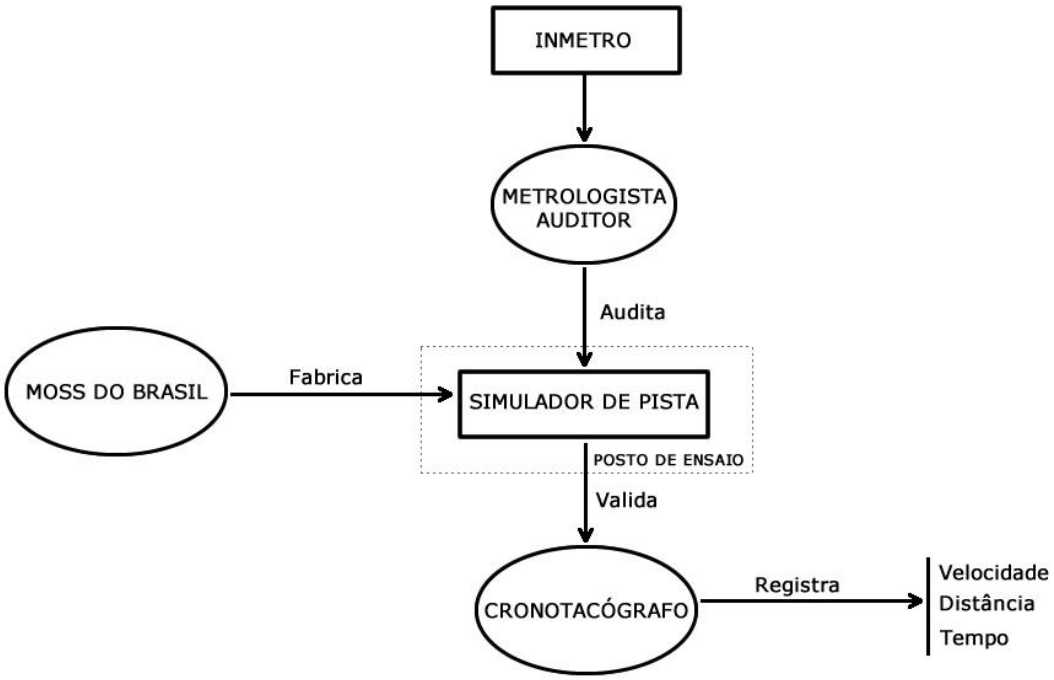

Figura 4. Estudo de caso para aplicação do protótipo

5.2. Aplicação do Mecanismo

A fim de resolver os problemas elencados anteriormente, propõe-se implantar o mecanismo desenvolvido no simulador de pista da Moss do Brasil. Dessa forma, no processo de homologação inicial para colocar o equipamento em funcionamento, o auditor do INMETRO deverá verificar se o software instalado no equipamento é o que foi homologado no projeto, assinar o software, configurar o dispositivo *USB Key*, instalar a aplicação de verificação AVIS UEFI, configurar o BIOS para executá-la e, finalmente, proteger o sistema com uma senha privada.

Com o mecanismo implantado, não haverá possibilidade de utilizar o equipamento com software adulterado, visto que a verificação é feita de forma automática a cada inicialização do equipamento. Caso o software seja modificado, o seu uso será impedido e deverá ser solicitada ao INMETRO uma nova homologação para colocar o equipamento novamente em funcionamento. Além disso não haverá necessidade de verificar a integridade do software nas auditorias periódicas, já que há garantia de que o software está íntegro, se o equipamento inicializa. Isso diminui o tempo gasto na auditoria e aumenta a eficiência do processo.

5. Avaliação de Resultados

Para verificar o correto funcionamento do protótipo, foram definidos oito casos de teste que consistem em inicializar o equipamento em situações diferentes. O ambiente para a realização dos testes foi configurado com um Mini-PC semelhante ao utilizado no painel de comando do simulador de pista da Moss do Brasil. Este possui processador Intel Core i3 1.6 GHz, 4 GB RAM, HD SSD com sistema de arquivos NTFS, BIOS UEFI Intel Visual BIOS versão 1.2.7 e sistema operacional Windows 8.1 Pro. Os casos de teste executados são descritos na Tabela 2.

Tabela 2. Casos de testes definidos para verificação do mecanismo

	Descrição do caso de teste	**Resultado esperado**
1	Inicialização com o equipamento devidamente configurado e o software íntegro	O sistema inicializa corretamente e o equipamento pode ser utilizado
2	Inicialização com o equipamento devidamente configurado, mas com software adulterado	A AVIS UEFI detecta a adulteração e bloqueia o equipamento
3	Inicialização com o equipamento devidamente configurado, mas com o software ausente	A AVIS UEFI detecta a falta do software e bloqueia o equipamento
4	Inicialização com software íntegro, mas com dispositivo *USB Key* ausente	A AVIS UEFI detecta a falta do dispositivo e bloqueia o equipamento
5	Inicialização com software íntegro, mas com dispositivo *USB Key* com PIN incorreto	A AVIS UEFI detecta o PIN incorreto do dispositivo e bloqueie o equipamento
6	Inicialização com software íntegro e com dispositivo *USB Key* presente e com PIN correto, mas com dados apagados.	Espera-se que a aplicação de verificação de integridade detecte a falta do conteúdo no *USB Key* e bloqueie o equipamento.
7	Inicialização com o equipamento devidamente configurado, o software íntegro e a aplicação de verificação de integridade modificada para registrar os horários de início e término de sua execução	A AVIS UEFI registra os instantes em que inicia e conclui sua execução
8	Comparação do tempo de inicialização do equipamento com a aplicação de verificação no BIOS UEFI e sem a aplicação	A AVIS UEFI retarda o início da operação do equipamento em um tempo aceitável

Foram executados os oito casos de teste previstos e todos os resultados esperados foram atingidos. O uso do equipamento foi permitido quando a sua configuração estava correta e impedido em todas as situações em que havia alguma irregularidade, como mostrado na Figura 5. O tempo médio de execução da aplicação AVIS UEFI com software íntegro foi de 600 ms e, nos demais casos, o tempo máximo de execução foi de 2 s. No Caso de Teste 8, o tempo médio de retardo da inicialização foi de 2,14 s. Estes valores estão dentro da margem de tolerância do fabricante do equipamento. Com isso, pode-se afirmar que não há interferência no desempenho do mesmo, visto que a aplicação somente é executada em tempo de inicialização e o tempo de execução da mesma é pequeno. Em termos de código, a AVIS UEFI ocupa 2,7 MB de memória, o que representa um custo baixo de memória para executar o mecanismo proposto. Com os resultados obtidos, pode-se afirmar também que a implantação do mecanismo não tem impacto significativo no tempo de inicialização do equipamento.

6. Conclusão e Trabalhos Futuros

Neste artigo, foi apresentado um mecanismo de verificação que aproveita os recursos do BIOS UEFI para atestar a integridade de software de sistemas embarcados utilizados em IoT. Este mecanismo tem como componente principal uma aplicação de verificação de integridade chamada AVIS UEFI. Essa aplicação utiliza assinatura digital

e chaves armazenadas em dispositivos criptográficos para verificar se o software foi adulterado. De acordo com o resultado, o sistema é inicializado ou desligado.

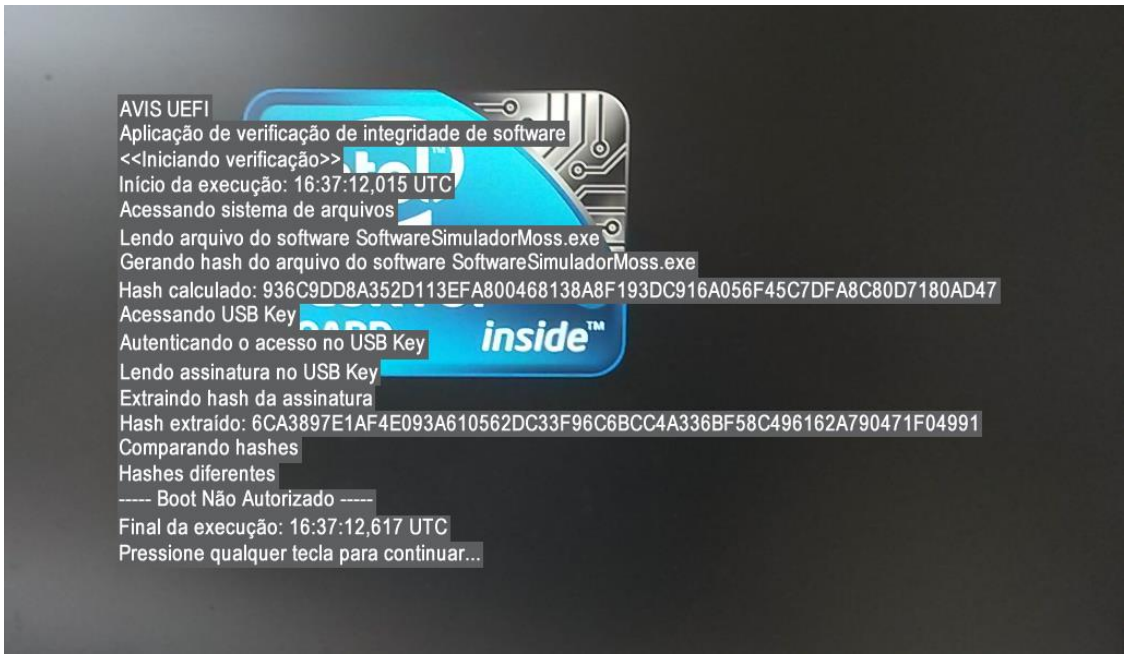

Figura 5. Inicialização interrompida devido a uma adulteração no software

Os trabalhos existentes que visam garantir a integridade de softwares em situações em que entidades homologadoras ou fabricantes não confiam no usuário apresentam limitações e não são apropriados para sistemas embarcados complexos. O mecanismo proposto neste artigo, por sua vez, não requer inclusão de código de verificação no software e, portanto, preserva a confidencialidade e a propriedade intelectual do mesmo. A intervenção humana é necessária apenas para a configuração inicial do sistema e a verificação é feita de forma automática cada vez que o sistema é inicializado impedindo seu funcionamento caso o software não seja íntegro. Para maior segurança, a aplicação de verificação utiliza criptografia assimétrica e assinatura digital com apoio de dispositivos externo seguros.

A principal contribuição deste trabalho é aumentar a segurança de sistemas embarcados utilizados em IoT mediante a verificação de integridade de software, por meio de um mecanismo simples e eficaz que é executado em cada inicialização do equipamento e impede seu uso em caso de adulterações.

O escopo desta pesquisa limitou-se à definição de um mecanismo para verificação de integridade de software para sistemas embarcados que possuem BIOS UEFI. Assim, o mecanismo não será aplicável a sistemas embarcados que não possuem BIOS UEFI.

Como trabalhos futuros, propõe-se realizar análises de segurança para identificar novas vulnerabilidades. Propõe-se desenvolver uma nova solução na qual a aplicação de verificação AVIS UEFI está no *USB Key* e comparar com a solução descrita neste trabalho. Por fim, pretende-se estender o mecanismo de forma a possibilitar a verificação contínua de integridade, ou seja, também em tempo de execução.

Referências

Abomhara, Mohamed; Koien, Geir M. Cyber "Security and the Internet of Things: Vulnerabilities, Threats, Intruders and Attacks". Journal of Cyber Security, River Publishers, v. 4, p. 65-88, 2015.

Ara, Tabassum; Gajkumar Shah, Pritam; Prabhakar, M. Internet of Things Architecture and Applications: A Survey. Indian Journal of Science and Technology, [S.l.], dez. 2016. ISSN 0974 -5645.

Basile, Cataldo; Di Carlo, Stefano; Scionti, Alberto. FPGA based remote code integrity verification of programs in distributed embedded systems Proceedings of. IEEE Transactions on Systems, Man, And Cybernetics - Part C 1, 2012.

Carmo, Luiz Fernando Rust da Costa; Machado, Raphael Carlos Santos. Verificação de integridade de software embarcado através de análise de tempo de resposta. Anais do IX Simpósio Brasileiro em Segurança da Informação. Campinas, v.1, 2009.

Carmo, Luiz Fernando Rust da Costa; Machado, Raphael Carlos Santos. Metrologia Temporal na Verificação de Integridade de Software em Instrumentos de Medição. Produto & Produção, vol. 11, n. 1, p. 80 - 88, fev. 2010 - Edição Metrologia.

Castro, Cristiano G. de; Moraes, Flávio P; Boccardo, Davidson R; Machado, Raphael C. S; Brandão, Paulo C; Carmo, Luiz F. R. C. FVIS: Uma Ferramenta de Verificação de Integridade de Software. Conference: X International Congress on Electrical Metrology, Buenos Aires, Argentina, v. 1, setembro, 2013.

Fink, Glenn A.; Zarzhitsky, Dimitri V; Carrol, Thomas E; Farquhar, Ethan D. Security and privacy grand challenges for the Internet of Things. National Security Directorate. Pacific Northwest National Laboratory: Washington, USA, 2015.

Liu, Hong; Li, Hongmin; Vasserman, Eugene Y. "Practicality of Using Side-Channel Analysis for Software Integrity Checking of Embedded Systems". In: 11th EAI International Conference on Security and Privacy in Communication Networks, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015.

MeiHong, Li; JiQiang, Liu. USB Key-Based Approach for Software Protection. Proceedings of International Conference on Industrial Mechatronics and Automation (ICIMA), 2009, IEEE.

Miritz, Luiz Alfredo Dittgen. Programação de Sistemas Embarcados usando Microcontroladores: um estudo de caso. Anais do EATI - Encontro Anual de Tecnologia da Informação e STIN – Simpósio de Tecnologia da Informação da Região Noroeste do RS. Ano 6 n. 1 p. 85-92 Nov/2016.

Seshadri, Arvind; Perrig, Adrian; Doorn, Leendert van; Khosla, Pradeep. Using Software-based Attestation for Verifying Embedded Systems in Cars. Proceedings of IEEE Symposium on Security and Privacy, 2016.

Zheng, Shi-yuan; Liu, Jun. An USB-Key-Based Approach for Software Tamper Resistance. Proceedings of 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE), 2010, IEEE.