Verificação de integridade de software embarcado através de análise de tempo de resposta

L. F. R. da C. Carmo[1], **R. C. S. Machado**[1]

[1]Diretoria de Metrologia Científica e Industrial
Instituto Nacional de Metrologia, Normalização e Qualidade Industrial (Inmetro)

{lfrust,rcmachado}@inmetro.gov.br

Abstract. *Software integrity verification is a main concern in several applications. It is a challenging problem to verify whether the software being executed in a given device is non-violated without having direct access to the memory area where it is stored the executable code. In the present work we propose a software integrity verification approach that addresses this problem. Our approach is based on the concept of "reflection", in which a software is requested to answer some questions about itself. We show that, under very reasonable assumptions, it is possible to build a protocol in which only an integer software will be able to answer correctly and in the expected time to the questions made to it.*

Resumo. *Verificação de integridade de software embarcado é uma preocupação importante em diversas aplicações. É desafiador o problema de verificar a integridade do software em execução em um dispositivo sem que seja necessário ler o conteúdo da memória de código executável do dispositivo. No presente trabalho, propomos uma abordagem de verificação de integridade de software que dispensa o acesso externo às instruções do programa em execução. Nossa abordagem usa o conceito de "reflexão", no qual o software deve responder a questões sobre ele mesmo. Demonstramos que, sob certas hipóteses bastante plausíveis, é possível construir um protocolo em que apenas o software íntegro será capaz de responder corretamente, e no tempo esperado, às perguntas a ele feitas.*

1. Introdução

O número de dispositivos baseados em software embarcado vem crescendo enormemente. Um risco crítico associado à presença cada vez maior de tais sistemas é o interesse de determinados indivíduos em comprometer a integridade de tais dispositivos com vistas a alcançar interesses individuais. Por exemplo, o proprietário de uma balança ou outro instrumento de medição envolvido em transação comercial pode procurar alterar o software de tal instrumento de tal forma que as medições passem a lhe beneficiar. Softwares embarcados em automóveis podem ser alterados com o objetivo de se ter acesso gratuito a um sistema de GPS ou ainda de se obter maior potência do motor. Assim, é fundamental que se tenha capacidade de verificar se o software embarcado em um dispositivo é realmente uma versão de software que fora previamente analisada e aprovada por autoridade competente. O objetivo do presente trabalho é a verificação da integridade de um software cujo código em execução não está disponível para leitura

externa. Em outras palavras, deseja-se saber se o software em execução corresponde exatamente a uma versão previamente aprovada, sem que seja necessário, no entanto, ler o conteúdo deste software (ou seja, o código executável armazenado no dispositivo). Embora a metodologia de verificação desenvolvida neste trabalho seja potencialmente aplicável a diversos ambientes computacionais, as hipóteses se aplicam mais naturalmente a dispositivos baseados em software embarcado de propósito específico e arquitetura simples, ou seja, dispositivos sem sistema operacional ou recursos como memória virtual ou offset de memória.

Desenvolvemos uma metodologia baseada no conceito de reflexão, no qual o software verificado é solicitado a responder uma série de perguntas a respeito de si mesmo. Baseados na corretude das respostas e no tempo utilizado para o cálculo destas respostas, podemos determinar se o software em execução é, de fato, o software que ele alega ser. A hipótese básica para o uso da reflexão é que o software seja capaz de ler o seu próprio conteúdo, ou seja, acessar a memória de código executável onde estão armazenadas suas próprias instruções. Satisfeita essa premissa, somos capazes de desenvolver um protocolo no qual o software a ser verificado é solicitado a retornar uma série de resumos criptográficos de trechos de seu próprio código. O protocolo é construído de tal forma que, sob certas hipóteses, nenhum programa malicioso P'' é capaz de se comportar como o programa P' previamente aprovado. Possivelmente, P'' será capaz de retornar as mesmas respostas que aquelas esperadas de P', entretanto, P'' nunca será capaz retornar as respostas corretas **no tempo esperado**, ou seja, o programa P' **apresenta comportamento temporal conhecido e único** quando restrito às entradas referentes aos comandos de verificação de integridade.

A Seção 2 apresenta abordagens encontradas na literatura com a finalidade de verificação de integridade de software. Na Seção 3 são definidos conceitos sobre os quais será construída nossa metodologia de verificação de integridade. A Seção 4 apresenta um primeiro exemplo de verificador de integridade, juntamente com uma demonstração de seu funcionamento, enquanto a Seção5 mostra possibilidades de aperfeiçoamento desse verificador. A Seção 6 descreve nossos primeiros resultados experimentais e, finalmente, a Seção 7 contém nossas considerações finais.

2. Trabalhos relacionados

O problema de verificação de integridade de software é um tema freqüentemente abordado. O coprocessador criptográfico IBM4758 [Smith, Palmer, and Weingart 1998] [Smith, Perez, Weingart, and Austel 1999] [Smith and Weingart 1999] implementa uma espécie de "boot seguro" [Arbaugh, Farber, and Smith 1997] [Arbaugh, Keromytis, Farber, and Smith 1998], que parte de um estágio inicial confiável e somente executa o conteúdo de um nível posterior após verificar sua assinatura. Sistemas como TPM [Trusted Computing Group] e NGSCB [Next-Generation Secure Computing Base] também implementam uma forma de boot seguro, mas com mecanismo diferente: resumos criptográficos de diversos componentes de software são armazenados em um coprocessador de segurança. Uma abordagem inteiramente baseada em software é descrita em [Kennell and Jamieson 2003], mas tal abordagem não se aplica a dispositivos baseados em software embarcado. Dentre os diversos métodos de verificação de integridade de software embarcado, o mais simples envolve o acesso ao código em execução no dispositivo – ou seja, a leitura de toda a

área destinada a código executável. Tal abordagem, embora bastante direta, apresenta a desvantagem de que todo o código executável fica disponível para leitura por qualquer um que tenha acesso ao dispositivo. Tal fato, além de colocar em xeque a segurança da propriedade intelectual daquele que desenvolveu o software, pode representar redução de segurança [1]. Além disso, a abordagem de leitura do código executável apresenta o inconveniente de que, em geral, deve-se ter acesso direto ao chip no qual se localiza o software embarcado, o que nem sempre é prático ou possível. Uma abordagem mais sofisticada envolve a utilização de circuitos integrados capazes de retornar resumos criptográficos do código executável nele armazenado. Tal funcionalidade apresenta uma enorme evolução no que diz respeito à proteção da propriedade intelectual. No entanto, ainda existe o problema de ser necessário acesso direto ao circuito integrado. Além do mais, são relativamente poucos – além de caros – os dispositivos eletrônicos integrados que apresentam tal funcionalidade.

Recentemente, uma abordagem alternativa começou a ser avaliada. Tal abordagem, baseada no conceito de reflexão [Smith 1982], consiste em enviar ao software uma série de comandos de verificação e checar se as respostas são recebidas conforme esperado. Tal abordagem possui evidentes vantagens: além de proteger o conteúdo do código executável, permite que a verificação seja feita através dos canais de comunicação usuais do dispositivo, sem que seja necessário remover do dispositivo o chip no qual se localiza o software (ver Figura 1).

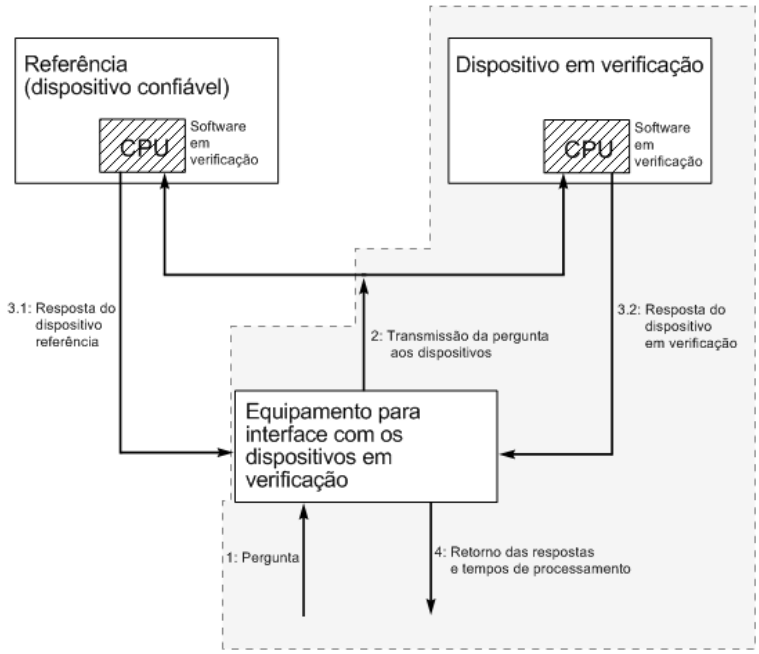

Figura 1. Esquema para verificação de integridade.

Observe que, na prática, caso os resultados esperados (previamente testados junto a um dispositivo confiável) sejam conhecidos, as questões para a verificação de

[1]Não estamos defendendo a segurança por obscuridade. O que acontece é que, em certas arquiteturas, pode ser mais interessante, por exemplo, armazenar chaves criptográficas na mesma memória onde se localiza o código executável.

integridade somente precisarão ser enviadas ao dispositivo em verificação (ou seja, o esquema reduz-se ao diagrama no interior da linha tracejada). O equipamento de interface, que já possuirá as informações sobre as respostas esperadas e o tempo de processamento, poderá informar simplesmente se o dispositivo em verificação respondeu da maneira esperada.

Uma abordagem ingênua para a verificação do software de um dispositivo consiste em solicitar que o dispositivo retorne um resumo criptográfico de seu código executável. Tal abordagem apresenta bons resultados apenas quando se assume que existe pouca memória livre e que o código executável possui baixa compressibilidade (essa segunda hipótese, em geral, não é válida [Douglas 1993]). Uma vez que tais hipóteses não são válidas – ou seja, assumindo a existência de memória livre virtualmente ilimitada – então é perfeitamente viável que um software malicioso mantenha, em memória, uma cópia do software original, e que, toda vez que solicitado a retornar um resumo criptográfico, efetue os cálculos sobre a cópia do software original mantida em memória, e não sobre o código que efetivamente está em execução (Figura 2).

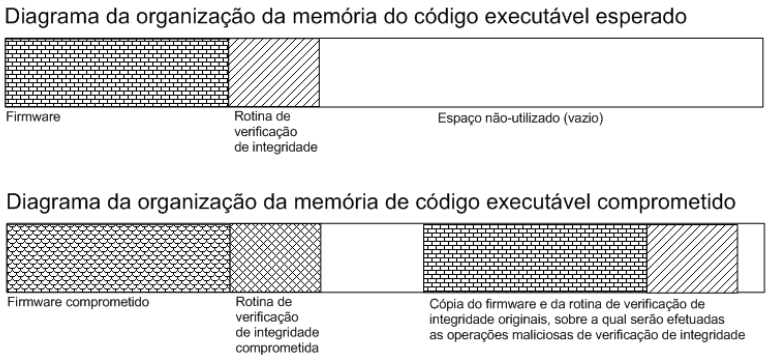

Figura 2. Diagrama da organização da memória em um software comprometido.

Recentes trabalhos apontam para a possibilidade de se verificar a integridade do software em um dispositivo não apenas avaliando as respostas aos comandos de verificação de integridade, mas também o comportamento do dispositivo no cálculo de tais respostas, em particular, o comportamento temporal [Spinnelis 2000] [Seshadri, Perrig, and van Doorn 2004]. Argumenta-se que as operações extras necessárias para que um software malicioso acesse a área de memória onde se localiza a cópia do software original implicam variações no tempo de processamento e uso da memória. Um exemplo de uso desta abordagem é o SWATT [Seshadri, Perrig, and van Doorn 2004]. No protocolo SWATT, o software que está sendo verificado recebe uma semente que será usada para gerar uma seqüência pseudo-aleatória de endereços de memória. As instruções localizadas nestes endereços são acessadas e nelas é baseada a resposta dada pelo software em verificação. A idéia principal na abordagem é que o um software malicioso não pode prever a seqüência de endereços acessados, de tal forma que, para comportar-se como o software original, deverá checar se o endereço acessado corresponde a uma instrução original ou alterada. Mesmo que uma única instrução tenha sido alterada, a execução irá durar um tempo maior, devido à presença de uma instrução condicional no loop principal da rotina de verificação. Os desenvolvedores do SWATT argumentam que a diferença entre os tempos

de resposta do software original e de qualquer software malicioso irá crescer linearmente com o tamanho da seqüência de acessos à memória de código.

Apesar da abordagem bastante original, o SWATT apresenta algumas fraquezas. A primeira fraqueza origina-se do fato de que geradores de números pseudo-aleatórios eventualmente geram ciclos (exatamente a partir do momento em que se repete um estado), o que compromete avaliações de segurança baseadas em comportamento assintótico. Ainda assim, mesmo que fosse razoável efetuar uma análise de comportamento assintótico, o tempo de resposta do software da rotina de verificação não seria linear no tamanho m da entrada m, pois existem, no loop principal da rotina, operações cujo tempo de execução é logarítmico em m. Uma segunda fraqueza decorre do fato de que, embora o acesso à memória se faça de forma bastante "imprevisível", a operação executada sobre a instrução recuperada é bastante simples. Assim, torna-se viável recuperar o código executável a partir de respostas aos comandos de verificação de integridade.

O presente trabalho busca fundamentos teóricos para a idéia de utilizar o comportamento temporal de um software com o objetivo de verificação de integridade. Mais precisamente, mostramos como, a partir de um programa P, construir um programa P' que apresenta exatamente o mesmo comportamento que P quando recebe entradas que são válidas para P, mas que aceita comandos adicionais de verificação de integridade e que responde a esses comandos em um tempo ótimo, de tal forma que qualquer outro programa P'' que responda corretamente a esses comandos de verificação de integridade o fará em tempo perceptivelmente maior que P'.

3. Conceitos e notação

Problema. Um *problema* é uma função $\Pi : X \to Y$ que associa, a cada entrada $x \in X$, uma saída $y \in Y$. Um objetivo geral da Ciência da Computação é, dado um problema, escrever um algoritmo (um programa) que "resolva" o problema, ou seja, que compute a função, da maneira mais "eficiente" possível. De uma forma geral, o termo "eficiente" refere-se a reduzir o uso de memória e processamento.

Máquina. Uma *máquina* é um dispositivo capaz de executar um conjunto de *instruções*, das quais são construídos os programas. Uma máquina utiliza um determinado intervalo de tempo para a execução de cada instrução. O conceito de máquina é essencial para que se possa determinar o tempo exato de execução de um algoritmo/programa, idéia na qual se baseia nossa metodologia de verificação de integridade. Formalmente, uma máquina M é um par ordenado $M = (C, T)$ onde C é o conjunto das instruções que tal máquina é capaz de executar e $T : C \to \mathbb{N}$ é o tempo utilizado na execução de cada instrução. Três instruções especiais, presentes nas máquinas com que lidamos, são as instruções GET, que lêem uma informação de entrada de tamanho fixo, a instrução PUT, que escreve uma informação de tamanho fixo para a saída, e a informação HALT, que determina o fim da execução do programa.

Programa. Um programa é uma seqüência $(c_1, c_2, ..., c_n)$ de instruções de uma máquina $M = (C, T)$ cuja memória de programa possui tamanho n, algumas das quais são a instrução HALT. Uma execução de um programa é uma seqüência $E = (c_{i_1}, c_{i_2}, ..., c_{i_k})$, onde $i_j \in \{1, 2, ..., n\}$ para $j = 1, ..., k$. Durante uma execução, um programa efetua uma série de operações de leitura de entrada, escrita para a saída, e

outras computações, até que encontre uma instrução HALT, momento em que o programa termina. O *tempo de execução* de E é $\sum_{j=1}^{k} T(c_{i_j})$. Para cada entrada (ou conjunto de entradas), a execução de um programa – e, portanto, seu tempo de execução – é unicamente determinada.

Entradas. Para simplificar nossas análises, supomos que a entrada de um programa é feita através de uma única instrução GET sem parâmetros, a qual lê um conjunto de dados de tamanho fixo previamente armazenado em um buffer de entrada. Assim, a leitura do conjunto de dados da entrada de uma rotina é, na verdade, uma seqüência de operações GET, que podem ou não ser executadas em seqüência. Pode-se pensar na entrada como uma fila que vai sendo esvaziada a cada instrução GET.

Comportamento temporal. O *comportamento temporal* de um algoritmo/programa para o conjunto de entradas X é a função $T : X \rightarrow \mathbb{N}$ que informa, para cada possível entrada, o tempo de execução deste algoritmo/programa.

Resumos criptográficos. Um *hash* é uma função $H : M \rightarrow D$ que, para cada *mensagem* $m \in M$, retorna um *resumo* $d \in D$. Em sistemas de segurança, hashes são construídos de tal forma que uma pequena modificação na mensagem ocasiona grande mudança no resumo retornado. Além disso, deve ser extremamente difícil (computacionalmente inviável) encontrar um par de mensagens m_1, m_2 tal que $H(m_1) = H(m_2)$. Observe que um hash se encaixa na definição de problema, pois se trata de uma função. Porém, assim, como muitas primitivas criptográficas, o MAC possui uma característica peculiar: na prática, a maioria dos padrões de hash é definida por um algoritmo que o calcula. Por exemplo, o MD5 é simplesmente a função que, dada uma mensagem, retorna o resumo obtido após a aplicação do algoritmo descrito em [Rivest 1992]. Tal característica levanta uma questão interessante, que é a de se tais algoritmos são algoritmos ótimos. Conforme veremos neste artigo, tal hipótese é bastante razoável, e é baseado nela que desenvolvemos nossa metodologia de verificação de integridade.

Um problema $\Pi' : X' \rightarrow Y'$ é um *problema verificador* de um problema $\Pi : X \rightarrow Y$ se existe um comportamento temporal $T : X' \setminus X \rightarrow \mathbb{N}$ tal que:

1. $X' \supset X$, ou seja, X' contém propriamente X;
2. $\Pi'|_X = \Pi$, ou seja, Π e Π' são o mesmo problema quando restritos a entradas em X; e
3. todo programa P' comportamento temporal T para entradas em $X' \setminus X$ resolve Π.

Neste caso, dizemos que o programa P' que satisfaz à condição temporal descrita no terceiro item *verifica* Π *através de* Π', ou ainda que P' é um *programa verificador de* Π *por* Π'. Na prática, a condição temporal mencionada se refere a alguma forma de "otimalidade".

Nossa estratégia para verificação de integridade é baseada na construção de problemas e programas verificadores. Dados um problema $\Pi : X \rightarrow Y$ e um programa P que resolve Π, descrevemos um método para construir um problema $\Pi' : X' \rightarrow Y'$ verificador de Π e um programa P' que verifica Π através de Π'. A integridade de P' será verificada através de seu comportamento temporal para entradas em $X' \setminus X$. Na prática, nossa abordagem envolve, ainda, um segundo problema: qual o menor subconjunto de $X' \setminus X$ que deverá ser testado para se conhecer o comportamento temporal restrito a este

conjunto de entradas.

4. Um primeiro verificador: retorno de instrução

Descrevemos um primeiro método para a construção de verificadores. Embora este primeiro método descrito apresente fraquezas, ele cumpre o papel de apresentar um exemplo simples de verificador. Além disso, dada a sua simplicidade, será um primeiro exemplo para o qual poderemos apresentar uma prova formal do comportamento temporal de um verificador. Nas Seções 5.1 e 5.2, veremos como aprimorar este verificador, corrigindo suas fraquezas.

Dado um programa P que resolve um problema $\Pi : X \rightarrow Y$, construímos um programa P' que aceita comandos adicionais de verificação, os quais formam um conjunto V. Cada comando de verificação $C = (VERIFIC, addr)$ contém, além de sua identificação $VERIFIC$ como comando de verificação, um endereço $addr$. O programa P' é construído da seguinte forma:

1. Após cada operação de leitura de entrada, insere-se uma instrução que verifica se a entrada é um "comando de verificação".
2. Caso a entrada seja um comando de verificação – e somente neste caso – serão executadas três instruções. A primeira instrução lerá a próxima informação do comando, que é o endereço $addr$. A segunda instrução buscará a instrução localizada no endereço $addr$. A terceira instrução retornará o conteúdo de $addr$.

Seja $\Pi' : X \cup V \rightarrow Y$ o problema resolvido por P'. Argumentamos que Π' é um verificador de Π, e o comportamento esperado de P' para entradas em V é a otimalidade para todas as entradas:

1. $\Pi'|_X = \Pi$. As modificações introduzidas em P' não alteram o comportamento do programa com relação às entradas já tratadas por P.
2. P' resolve otimamente $\Pi'|_V$. Uma vez lido o comando de verificação, apenas duas operações são executadas: a leitura da instrução em $addr$ e a escrita dessa instrução para a saída. Observe que não se poderia deixar de ler $addr$ e a instrução localizada em $addr$, pois a saída **depende** da instrução localizada em $addr$ (supondo que o programa não é composto de uma única instrução) e não se conhece $addr$ *a priori*.
3. Mostraremos que P' é o único programa que resolve otimamente $\Pi'|_V$. Seja P'' um programa que possui pelo menos uma instrução, digamos no endereço a, diferente da instrução de P' no mesmo endereço. Observe que a rotina de tratamento de comandos de verificação de integridade precisará conter ao menos uma instrução adicional para verificar se o endereço $addr$ é igual a a.

Observe que o fato de P' ser o único programa que resolve otimamente $\Pi'|_V$ não significa que não existam outros programa que, **para algumas entradas específicas**, respondam corretamente a um comando de verificação de integridade. Tais programas, no entanto, **não são capazes** de resolver $\Pi'|_V$ **para** todas as entradas possíveis em V, de forma que é possível detectar um programa malicioso, como no exemplo a seguir. Um exemplo de programa $\tilde{P}$ mais rápido que P' seria um programa que respondesse, a cada comando de verificação de integridade com endereço $addr$, uma saída que não dependesse de $addr$ ou da instrução aí localizada. Tal programa poderia ser implementado de duas possíveis maneiras. Uma primeira maneira seria retornar uma informação fixa

para a saída. Esse primeiro programa malicioso é facilmente identificável, bastando que se enviem dois comandos de verificação de integridade que esperam repostas diferentes. Uma segunda maneira seria retornar para a saída uma informação aleatória. Embora seja bastante improvável que tal saída aleatória possa ser implementada de forma eficiente, podemos assumir que, em algumas máquinas, isso seja possível (por exemplo, retornando um registro de horário). Neste caso, para cada comando de verificação de integridade, existe uma probabilidade positiva de que tal retorno aleatório seja correto. No entanto, programa apresentará baixa probabilidade de acerto (no máximo o inverso do número de instruções da máquina), e esta probabilidade poderá se tornar tão baixa quanto se queira, bastando, para isso, efetuar o teste de verificação de integridade várias vezes. Considere o exemplo de um programa malicioso P'' que retorne aleatoriamente uma instrução entre vinte possíveis quando recebe um comando de verificação de integridade. A probabilidade de P'' retornar uma resposta certa é de 0,05 – já bastante pequena. Além disso, a probabilidade de P'' retornar dez respostas certas é uma em $9,7 \times 10^{14}$, certamente uma probabilidade ínfima quando consideramos o pequeno número de testes.

Um problema crítico do verificador construído anteriormente é que o comando de verificação expõe o conteúdo do programa em execução, justamente aquilo que buscamos proteger. Uma possibilidade para contornar essa fraqueza seria aplicar uma operação XOR bit-a-bit entre a instrução recuperada e uma seqüência de bits conhecida apenas pelo software em verificação e pelo usuário que conduz a verificação. Dessa forma, a informação retornada a um usuário malicioso não fará nenhum sentido, pois foi "mascarada" por uma seqüência de bits desconhecida [2]. Ainda assim, resta um segundo inconveniente: embora o programa P' resolva otimamente $\Pi'|_V$, um programa alterado P'' pode responder adequadamente aos comandos de verificação simplesmente mantendo uma cópia de P' em um trecho de memória de código não-utilizado, digamos, a partir do endereço x. O comportamento da rotina de verificação, ao receber o endereço $addr$, seria retornar a instrução localizada no endereço $x + addr$, conforme o ataque ilustrado na Figura 2. Embora o programa P'' não resolva otimamente $\Pi'|_V$, ele gasta apenas uma instrução adicional – uma soma – o que, na prática, representa uma diferença de tempo muito difícil de ser detectada. Nas próximas seções veremos como construir verificadores que contornam os problemas descritos.

5. Verificadores cujo tempo de resposta depende do tamanho da entrada

Na presente seção descrevemos métodos para a construção de verificadores cuja entrada apresenta tamanho variável. Devido a tal característica, pode-se forçar o tempo de resposta a ser tão grande quanto se queira. Em particular, pode-se forçar o tempo de resposta de um programa malicioso a ser tão maior que o tempo de resposta do programa original quanto se queira.

À medida que abandonamos um verificador muito simples como aquele descrito na Seção 4 e passamos a um verificador mais complexo, com uma quantidade infinita de entradas, passamos a precisar de mais hipóteses, tal como a hipótese da otimalidade

[2]Na verdade, como nem toda seqüência de bits é uma informação ou programa válido – ou seja, a entropia de programas é baixa – caso se permita que um indivíduo obtenha todas as instruções de um programa mascaradas por uma mesma seqüência de bits, este indivíduo será capaz de recuperar as instruções originais do programa. Ou seja, a técnica de mascaramento com XOR funciona apenas sob a hipótese de alta entropia. Caso esta hipótese não seja válida, a saída é usar operações criptográficas.

da implementação da operação efetuada no loop principal da rotina de verificação de integridade. Trabalhos futuros certamente deverão abordar a razoabilidade de tais hipóteses e contemplar possíveis demonstrações formais de sua validade.

5.1. Verificadores baseados em XOR

Uma maneira de contornar a uma das fraquezas descrita na Seção 4 – a pequena diferença entre o tempo de reposta de um programa P' ótimo para $\Pi'|_V$ e um programa alterado P'' – é forçar a rotina de verificação a fazer uma grande quantidade de leituras do código executável. Um programa malicioso que busque simular o comportamento do programa original via o ataque descrito na Figura 2 deverá executar, antes de cada operação de leitura de código, uma instrução condicional ou adição, de forma que o número de **instruções adicionais** executadas por qualquer software malicioso será proporcional ao número de acessos ao código executável. A idéia é que o programa verificador receba uma seqüência de endereços e retorne um "resumo" dependente das instruções localizadas naqueles endereços.

Dado um programa P que resolve um problema $\Pi : X \rightarrow Y$, construímos um programa P' que aceitará alguns comandos adicionais de verificação, os quais formam um conjunto V. Cada comando de verificação $C = (VERIFIC, addr_1, ..., addr_k, NOINPUT)$ conterá, além de uma identificação $VERIFIC$ como comando de verificação, uma seqüência de endereços $addr_1, ..., addr_k$ e uma informação $NOINPUT$ indicando o fim da seqüência. O programa P' é construído da seguinte forma:

1. Após cada operação de leitura de entrada, inserir uma instrução que verifique se a entrada é um "comando de verificação".
2. Inicialize uma variável ret com valor nulo. Leia a próxima entrada do comando. Enquanto a entrada for um endereço (ou seja, diferente de $NOINPUT$), leia a instrução $instr$ localizada neste endereço e atualize ret aplicando um XOR bit-a-bit com a instrução com $instr$ (ou seja, $ret \leftarrow ret \oplus instr$).
3. Retorne ret.

Uma análise semelhante à da Seção 4 mostra que $\Pi'|_X = \Pi$. Mostramos que o problema $\Pi' : X \cup V \rightarrow Y$ resolvido por P' é um verificador de Π. Para isso, supomos que saída esperada depende de todas as instruções localizadas em $addr_1, ..., addr_k$ (o que poderia ser verdade, por exemplo, se todas as instruções fossem iguais, situação que parece bastante improvável). Dessa forma, qualquer programa malicioso P'' que resolva $\Pi'|_X$ deverá efetuar todas as iterações do loop principal. Baseado na otimalidade da implementação da operação efetuada no loop principal – neste caso, um simples XOR – uma análise semelhante à da Seção 4 permitirá concluir que todo programa malicioso que resolva Π' irá efetuar ao menos uma operação adicional dentro deste loop principal e que, portanto, não poderá ser ótimo para todas as entradas.

Considere o caso no qual a entrada seja uma seqüência de n endereços, e observe que o tempo utilizado pela rotina de verificação de integridade será $nk + C$, onde k é a soma dos tempos de leitura de entrada, leitura de instrução e operação XOR, e C é uma constante (determinada, por exemplo, por fatores como a configuração do ambiente necessário ao tratamento da entrada da função de verificação). Qualquer programa $P'' \neq P'$ que retorne os mesmos resumos que P' deverá executar instruções adicionais

(por exemplo, instruções de adição ou condicionais) dentro do loop principal da rotina de verificação de integridade. Portanto, o tempo de resposta de P'' será $n(k + x) + C'$, para algum x positivo. Assim, a diferença entre o tempo de execução de P'' e P' será $nx + (C' - C)$, assintoticamente proporcional ao tamanho da entrada – ou seja, para entradas suficientemente grandes, qualquer programa diferente de P' que retorne os resumos corretos o fará com um atraso tão grande quanto se queira forçar (proporcional ao tamanho da entrada) e, portanto, perceptível.

O verificador construído nesta seção apresenta evidente vantagem comparado com aquele construído na Seção 4, no que diz respeito ao comportamento temporal. No entanto, devido à simplicidade da operação XOR, tal verificador ainda expõe o conteúdo do código em execução no dispositivo. Tal código pode ser recuperado de maneira muito simples, ainda que se imponham limites inferiores para o tamanho da seqüência que formará o resumo retornado. Caso se deseje saber a instrução localizada no endereço a, basta enviar o comando de verificação de integridade duas vezes, a primeira vez da forma $C = (VERIFIC, addr_1, ..., addr_k, NOINPUT)$ e a segunda vez, da forma $C = (VERIFIC, addr_1, ..., addr_k, a, NOINPUT)$, onde $addr_1, ..., addr_k$ são endereços quaisquer. Um simples XOR entre os dois resultados revelará a instrução localizada no endereço a. Para contornar esse problema, na Seção 5.2, contornaremos esse problema construindo verificadores baseados em resumos criptográficos.

5.2. Verificadores baseados em operações criptográficas

O verificador criado na presente seção é bastante semelhante àquele descrito na Seção 5.1. A diferença é que, a cada iteração do loop principal da rotina de verificação, em vez de aplicar um simples XOR entre ret e a instrução atual $instr$, efetuamos uma operação criptográfica que depende de ret e $instr$ e atualizamos ret com o resultado. Nossa premissa básica para que a função construída seja, de fato, ótima quando restrita às entradas de verificação, é a de que a operação criptográfica é implementada de maneira ótima. Embora esse fato não seja explicitamente mencionado em nenhuma bibliografia, consideramos bastante razoável que a maioria das primitivas criptográficas padronizadas – as quais são definidas por algoritmos – tenham estes algoritmos como algoritmos ótimos.

Dado um programa P que resolve um problema $\Pi : X \rightarrow Y$, construímos um programa P' que aceitará alguns comandos adicionais de verificação, os quais formam um conjunto V. Cada comando de verificação $C = (VERIFIC, addr_1, ..., addr_k, NOINPUT)$ conterá, além de uma identificação $VERIFIC$ como comando de verificação, uma seqüência de endereços $addr_1, ..., addr_k$ e uma informação $NOINPUT$ indicando o fim da seqüência. O programa P' é construído da seguinte forma:

1. Após cada operação de leitura de entrada, inserir uma instrução que verifique se a entrada é um "comando de verificação".
2. Inicialize uma variável ret com valor nulo. Leia a próxima entrada do comando. Enquanto a entrada for um endereço (ou seja, diferente de $NOINPUT$), leia a instrução $instr$ localizada neste endereço e atualize ret aplicando um "hash" à string resultante da concatenação $ret||instr$.
3. Retorne ret.

Assim como na Seção 5.1, uma hipótese fundamental para que o problema $\Pi' : X \cup V \rightarrow Y$ resolvido por P' seja um verificador de Π é que a implementação a operação

criptográfica executada no segundo passo do item 3 seja ótima. Como as operações criptográficas de chave simétrica, em geral, possuem definições muito simples, em termos de operações de soma, substituição e deslocamento, a implementação de programas ótimos parece uma hipótese bastante razoável. Satisfeita esta hipótese de otimalidade da implementação das operações criptográficas, a análise de P' como programa verificador é semelhante àquela da Seção 5.1.

6. Avaliação experimental

Foi desenvolvida implementação de um verificador baseado em operações criptográficas que permitiu simular o comportamento temporal de programas verificadores maliciosos. Os primeiros testes executados permitiram detectar que, para entradas suficientemente grandes e que forçam um tempo de resposta da ordem de alguns minutos, softwares maliciosos já apresentam atrasos de resposta perceptíveis (da ordem de segundos).

Uma outra atividade em andamento foi o desenvolvimento de uma plataforma de testes de micro-controladores. Tal plataforma permite a interface entre dois micro-controladores (Figura 1), dos quais um é confiável e outro está em verificação. A plataforma projetada possui soquetes para a inserção dos micro-controladores e é responsável pelo fornecimento do sinal de clock a eles. Dessa forma, será possível saber exatamente quantos ciclos foram necessários para responder ao comando de verificação. Esta abordagem permite que se use programas verificadores cuja entrada possui tamanho fixo, tal como o descrito na Seção 4, que são mais simples e rápidos – portanto, cuja otimalidade é mais facilmente comprovável.

Métodos de verificação de integridade baseada em resumos criptográficos estão sendo aplicados a medidores de energia elétrica baseados no micro-controlador ATMEL AT89S8253 e que serão comercializados brevemente. O verificador implementado foi um verificador baseado em resumos criptográficos e com tempo de resposta dependente do tamanho da entrada, assim como descrito na Seção 5.2. Observou-se que, mesmo para entradas relativamente pequenas (com tempo de resposta da ordem de segundos), a inclusão de um único comando malicioso no loop principal já ocasiona uma diferença perceptível no tempo de resposta de um software malicioso (da ordem de milissegundos).

7. Conclusão

O trabalho descrito neste artigo representa contribuição às técnicas de verificação de integridade de software.

Em primeiro lugar, descrevemos formalmente as hipóteses necessárias para a aplicação bem-sucedida de protocolos baseados em reflexão [Smith 1982] ao problema de verificação de integridade de software. Nossos argumentos são fortemente baseados na teoria da complexidade de algoritmos.

Em segundo lugar, identificamos fraquezas nas abordagens SWATT [Seshadri, Perrig, and van Doorn 2004] para verificação de integridade, no sentido de ela permitir que um usuário mal-intencionado recupere o código do software em execução em um dispositivo com uma seqüência especial de comandos de verificação de integridade.

Finalmente, apresentamos métodos para a construção de programas verificadores que protegem o conteúdo do software em verificação e podem forçar um tempo de

resposta proporcional ao tamanho da entrada, o que permite detectar softwares maliciosos pelo tempo de resposta aos comandos de verificação.

Nossa metodologia de verificação de integridade baseia-se na premissa de que um programa P' que resolve o problema $\Pi' : X \cup V \rightarrow Y$ é o **único** que apresenta determinado comportamento temporal quando restrito a V. No entanto, é desejável que não seja necessário testar todas as entradas em V para caracterizar o comportamento temporal de P'. Por exemplo, na Seção 4, mostramos que duas verificações já seriam suficientes para caracterizar o comportamento de P'. Uma linha importante para futuras pesquisas é a de identificar, dada uma família de programas que resolvem $\Pi|_V$, qual o menor subconjunto de V que precisa ser testado de forma a garantir que o programa apresenta o comportamento temporal esperado.

Agradecimento. Os autores agradecem aos revisores anônimos que, com seus comentários e sugestões, possibilitaram uma enorme melhoria do presente artigo, tanto em sua forma quanto em seu conteúdo.

Dedicatória. O segundo autor dedica o presente trabalho a Edson Lopes Rodrigues (in memoriam).

Referências

Seshadri, A., Perrig, A., and van Doorn, L. (2004). Using software based attestation for verifying embedded systems in cars. In *Technical Report*. Carnegie Mellon University.

Kennell, R. and Jamieson, L. H. (2003). Establishing the genuinity of remote computer systems. In Proceedings of the 11th USENIX Security Symposium.

Smith, S. W. and Weingart, S. H. (1999). Building a high-performance, programmable secure coprocessor. Computer Networks (Special Issue on Computer Network Security), 31, pages 831–960.

Smith, S. W., Perez, R., Weingart, S. H., and Austel, V. (1999). Validating a high-performance, programmable secure coprocessor. In 22nd National Information Systems Security Conference.

Smith, S. W., Palmer, E., and Weingart, S. H. (1998). Using a high-performance, programmable secure coprocessor. In 2nd International Conference on Financial Cryptography.

Arbaugh, W. A., Keromytis, A. D., Farber, D. J., and Smith, J. M. (1998). Automated recovery in a secure bootstrap process. In Proceedings of the Symposium on Network and Distributed Systems Security, pages 155–167.

Arbaugh, W. A., Farber, D. J., and Smith, J. M. (1997). A reliable bootstrap architecture. In Proceedings of the IEEE Symposium on Research in Security and Privacy, pages 65–71.

Douglas, F. (1993). The compression cache: using on-line compression to extend physical memory. In *Proceedings of the Third USENIX Conference*, pages 519–529. USENIX Assoc.

Rivest, R. (1992). The md5 message-digest algorithm. In *RFC 1321*. Internet Engineering Task Force.

Smith, B. C. (1982). Procedural reflection in programming languages. In *Ph.D. Thesis*. MIT Laboratory for Computer Science.

Spinnelis, D. (2000). Reflection as a mechanism for software integrity verification. In *ACM Transactions on Information and System Security vol.3 n.1*, pages 51–62.

Next-Generation Secure Computing Base.
http://www.microsoft.com/resources/ngscb/default.mspx.

Trusted Computing Group.
https://www.trustedcomputinggroup.org.