2
PCC

2.1
O que é PCC?

Proof-Carrying code (PCC) é uma técnica com a qual um consumidor de código (por exemplo, um *host*) pode verificar se um código (possivelmente na forma binária [3]) provido por um produtor de código não-confiável está em conformidade com um conjunto predefinido de regras de segurança. Estas regras, também referenciadas como política de segurança, são escolhidas pelo consumidor de código de tal maneira que elas sejam garantias suficientes para o comportamento seguro dos programas [1].

Essa é uma tecnologia relativamente nova, desenvolvida por Peter Lee e George Necula, e que foi proposta para a verificação de segurança no uso de memória, segurança na tipagem, respeito às variáveis privadas e consumo limitado de recursos [4, 12].

Portanto, PCC pode dar aos usuários finais proteção para várias falhas em programas, as quais podem ser erros de tipagem, erros de gerenciamento de memória, violações no limite de recursos e no controle de acesso, além de outras políticas de segurança [24].

Proof-carrying code também livra o projetista de sistemas de contar apenas com a verificação durante a execução para assegurar segurança. Por exemplo, com relação à proteção de memória, isto pode ser conseguido através de mecanismos relativamente caros durante a execução, tais como a proteção de memória garantida através de *hardware* e o largo uso de verificação de dados em tempo de execução. Por estar limitado apenas a estes tipos de mecanismos, o projetista deve impor restrições substanciais na estrutura e na implementação de todo o sistema. Já se PCC for usado, é possível prover grande flexibilidade aos projetistas no desenvolvimento de sistemas, além de permitir a existência de políticas de segurança mais abstratas e mais refinadas [22].

Em uma instância típica de PCC, quem vai executar o código estabelece um conjunto de regras de segurança que garantem o comportamento seguro de programas, e o produtor do código cria uma prova formal de segurança que

PUC-Rio - Certificação Digital Nº 0124811/CA

garante a aderência às regras de segurança ao código não-confiável. Então, o primeiro é capaz de usar um validador de provas simples e rápido para verificar, com exatidão, que a prova é válida e, conseqüentemente, que o código não-confiável é seguro para ser executado [1, 3].

Com o uso dessa técnica, o consumidor de código se certifica sobre o código fornecido por um produtor de código não-confiável, sabendo que este código tem um determinado conjunto de propriedades previamente combinadas [3].

De acordo com [1], qualquer implementação de *proof-carrying code* deve conter, no mínimo, quatro elementos:

- Uma linguagem de especificação formal usada para expressar a política de segurança. No caso desta pesquisa, a linguagem é a lógica de primeira ordem;
- Uma semântica formal da linguagem usada pelo código não-confiável, geralmente na forma de uma lógica relacionando programas a especificações. Nesta pesquisa, a semântica é o Cálculo de Hoare;
- Uma linguagem usada para expressar provas. A apresentada neste trabalho é a sintaxe utilizada no cálculo de Hoare; e
- Um algoritmo para a validação das provas. No caso desta pesquisa, é um algoritmo que determina se a prova de correção de um programa está correta (sintaticamente de acordo com o cálculo de Hoare, sem contradições nas pós-condições e com todas as sentenças demonstráveis).

PCC se baseia nos mesmos métodos formais dos de verificação de programas. Todavia, possui a significante vantagem de que as propriedades de segurança são muito mais fáceis de se provar do que a correção de programas. Ou seja, *proof-carrying code* aplica técnicas tradicionais de verificação de programas ao domínio de verificação de propriedades de segurança. Usualmente, a prova formal do produtor não prova que o código produz um resultado correto e significativo, e sim que sua execução não produzirá nenhum perigo [4]. A não ser, claro, que a segurança desejada seja a de que o programa tenha um determinado comportamento, o que pode ser muito importante em determinados casos. Portanto, *proof-carrying code* não pode substituir outros métodos para a confiança em programas [4].

Na figura a seguir é mostrado como PCC é usado. Inicialmente, é necessário que o produtor e o consumidor decidam sobre a política de segurança e sobre o formato de codificação da prova antes da transferência. Essa é a chamada fase de negociação [19]. A política de segurança é codificada

PUC-Rio - Certificação Digital Nº 0124811/CA

em um gerador de condição de verificação (GCV), que mapeia cada programa (possívelmente binário) a um teorema de segurança. Nesta pesquisa, esse teorema é o programa a ser transmitido pela rede juntamente com suas pré e pós-condições de segurança. Isto é, o GCV gera as pré e pós-condições que especificam a política de segurança a qual o programa deve seguir. O produtor e o consumidor executam o GCV utilizando o programa como entrada para a obtenção do teorema. O produtor, então, consegue a prova do teorema. Neste trabalho, a prova é a prova de correção do programa. Essa fase da geração da prova é chamada de certificação. O produtor pode armazenar o programa PCC para uso futuro ou enviá-lo ao consumidor de código. Se a segunda opção for a escolhida, após a certificação, vem a fase de validação, onde o consumidor confere se a prova do teorema de segurança, que também é transmitida pela rede, é uma prova para o teorema de segurança do programa que está sendo validado. Se assim for, o programa é seguro de ser executado [3, 4].

A política de segurança consiste de dois componentes principais: as regras de segurança e a interface. As primeiras descrevem todas as operações autorizadas e as pré-condições de segurança associadas às mesmas. Já a segunda descreve as convenções de chamadas entre o consumidor de código e o programa, isto é, os invariantes que valem quando o consumidor chama a execução do programa e os invariantes que o programa deve estabelecer antes de chamar funções providas pelo consumidor ou antes de retornar a ele [3]. Além disso, é na política de segurança que é definido o conjunto de axiomas usados para a validação do predicado de segurança [21].

A verificação de provas é um processo simples e mecânico, relativamente tão simples quanto realizar a verificação de tipos em uma linguagem de programação [4]. Portanto, a validação é rápida e realizada por um algoritmo direto. É apenas na implementação desse algoritmo que o consumidor deve confiar além da corretude de sua política de segurança [3].

Isso significa que a base computacional confiável (BCC), a parte do sistema na qual um erro pode levar a uma violação de segurança, de todo o sistema é consideravelmente pequena e eficiente, consistindo apenas dos componentes GCV e Verificador. Totalmente fora da BCC estão a metodologia de engenharia de *sofware* do produtor, o código-fonte do programa, o compilador que realiza otimização de código, o *hardware* e o *software* da rede de transmissão e até o provador de teoremas [4].

Por outro lado, achar provas para teoremas é, em geral, não-tratável por ser indecidível e ter alto custo computacional. Para o processo de prova ser

PUC-Rio - Certificação Digital Nº 0124811/CA

automático para o produtor, deve ser explorada a estrutura do problema, que é o programa para o qual se deseja obter a prova de segurança e o teorema de segurança a ser provado [4]. Este é o foco principal deste trabalho. Propõe-se, então, algumas heurísticas para a prova automática de teoremas no cálculo de Hoare (capítulo 4) e implementa-se as mesmas em Prolog para que possam ser validadas (capítulo 5).

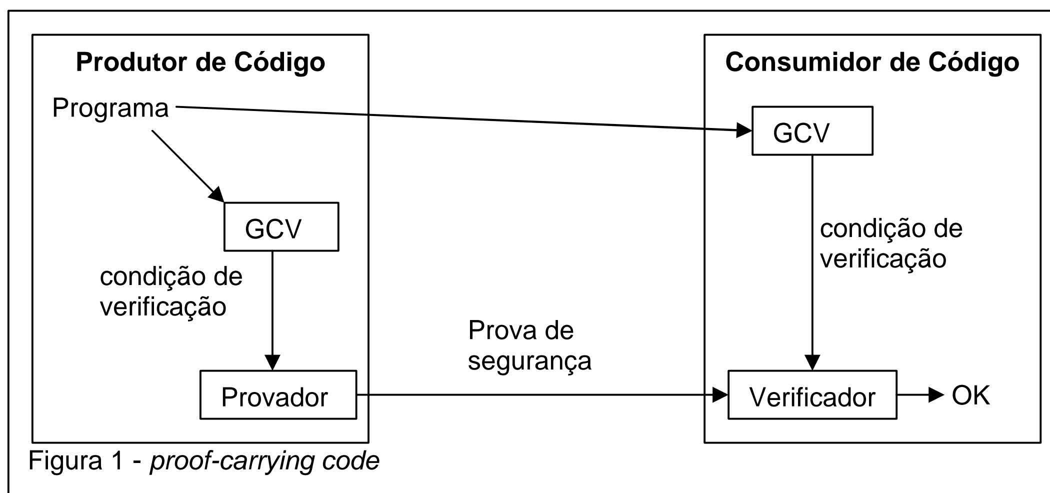

Figura 1 - *proof-carrying code*

PUC-Rio - Certificação Digital Nº 0124811/CA

2.2 Vantagens do Uso de PCC

PCC é, em especial, uma técnica interessante para certificação e validação de aplicações com código móvel, possuindo vantagens para esta e outras aplicações, entre elas [1, 3, 4, 22]:

- ❖ Apesar da possibilidade de haver um grande esforço em estabelecer e provar formalmente a segurança do código móvel, quase todo o trabalho fica a cargo do produtor do código. O consumidor, por outro lado, só precisa fazer um processo de verificação de prova rápido, simples e facilmente confiável. O fato de o verificador de provas ser digno de confiança é uma importante vantagem sobre abordagens que involvem o uso de compiladores complexos, interpretadores, análisadores de programas ou editores de código no lado do consumidor do código;

- O consumidor não se importa com a forma como as provas são construídas. Elas podem ser feitas utilizando um provador de teoremas, contudo, em geral, não há razão (exceto o possível gigantismo do esforço requerido) para que elas não possam ser geradas manualmente. Ou seja, qualquer que seja a maneira pela qual as provas são geradas, o consumidor de código não precisa confiar no processo de geração de provas (afinal, ele confia no verificador de provas);
- Os programas em *proof-carrying code* são a prova de alteração no seguinte sentido: qualquer modificação (seja acidental ou intencional) resultará em uma destas três conseqüências:
  1- A prova não será mais válida e, então, o programa será rejeitado;
  2- A prova será válida, mas não mais será uma prova de segurança para o programa, fazendo com que novamente o programa não seja aceito; e
  3- A prova continuará sendo válida e continuará sendo uma prova de segurança para o programa, apesar das modificações. Neste caso, mesmo que o comportamento do programa possa ter sido modificado, a garantia de segurança continua valendo;
- Não é necessário usar criptografia nem consultar agentes externos confiáveis, já que PCC verifica as propriedades intrínsecas do código e não a sua origem. Nesse sentido, os programas em *proof-carrying code* são “auto-certificáveis”. Todavia, PCC é completamente compatível com outras abordagens de segurança para códigos móveis. Em termos de engenharia, a combinação de abordagens gera ganhos, porém também traz desvantagens (como por exemplo, menor esforço requirido na geração de provas com o custo de menor eficiência em termos de tempo de execução). E essas combinações podem, ainda, levar a uma maior flexibilidade no projeto do sistema;
- Como o código não-confiável pode ser verificado sintaticamente antes de ser executado, o tempo de execução é salvo e a verificação só precisa ser feita uma vez para um dado programa, independentemente do número de vezes que ele será executado se não for alterado (afinal, já se sabe que o código respeita a política de segurança). Isto possui importantes vantagens em termos de engenharia, especialmente nos casos onde a verificação é difícil, consome muito tempo ou necessita de interação com o usuário. Além disso, também são detectadas anteriormente operações potencialmente perigosas, evitando situações onde o consumidor do código deve matar o processo não-

PUC-Rio - Certificação Digital Nº 0124811/CA

confiável depois desse último já ter adquirido recursos ou modificado o ambiente; e

- Não é necessário o uso de um compilador particular ou até mesmo o uso de algum compilador. Desde que o produtor de código possa prover a prova, o consumidor pode se assegurar da segurança. Isso aumenta significantemente a flexibilidade disponível para os desenvolvedores de sistemas.

Esses pontos citados acima são, essencialmente, afirmações sobre a vantagem de verificação estática sobre verificação dinâmica [1]. Além desses, uma força significativa de *proof-carrying code* é a flexibilidade que ela permite na especificação de políticas de segurança. Em contraste com a proteção de memória-virtual, por exemplo, a qual só permite privilégios de leitura e leitura / escrita em unidades de tamanho de página, pode-se especificar e implementar propriedades como as seguintes e combinações destas (que são para código binário) [4, 23]:

- É permitido ler ou escrever em qualquer endereço no intervalo $a_{inicial}$ e a_{final};
- Todas as posições de memória depois da localização e_i podem ser lidas;
- Todas as posições de memória depois da localização e_j podem ser escritas;
- Sair através de um salto para o endereço b_x é seguro;
- O *program counter* deve ser inicializado para a primeira instrução de código;
- É permitido acessar o objeto no endereço p_1 de acordo com o formato dos campos públicos da classe X;
- É permitido percorrer uma estrutura de dados que é uma árvore de acordo com o tipo recursivo t_2 no endereço p_2;
- Não é permitido ocorrer *overflow* nas instruções de adição;
- Não é permitido executar mais do que 150 instruções consecutivas antes de averiguar o endereço a_y;
- É permitido armazenar um 0 nos endereços s_1 e s_2, porém não em ambos; e
- Etc.

Para finalizar, é possível argumentar sobre o uso de criptografia ao invés de PCC para se assegurar que o código foi produzido por uma pessoa ou compilador confiável. Contudo, esse esquema é fraco graças a sua dependência de uma fonte externa (mesmo pessoas confiáveis ou compiladores escritos por elas podem cometer erros ocasionalmente ou, até mesmo, agir maliciosamente) [3].

PUC-Rio - Certificação Digital Nº 0124811/CA

2.3 Aplicações de PCC

Há muitas aplicações potenciais para PCC. Esta técnica possui muitos usos em sistemas cuja base de computação segura é dinâmica, seja por causa de código móvel, ou por conserto regular de *bugs* ou atualizações no código. Nas próximas subseções, serão descritos alguns exemplos de aplicações. Além desses, outros exemplos incluem nós de redes ativas e controladores embutidos onde a segurança é crítica [1].

2.3.1 Código Móvel

Para código móvel, o consumidor de código seria um *host* de Internet (ou seja, um navegador) e o produtor de código, um servidor que envia *applets* [1]. Nesse caso, o *host* precisa se assegurar de que o código não vai danificá-lo, por exemplo, corrompendo suas estruturas de dados internas. Também, o mesmo precisa se assegurar de que o código móvel não vai usar muitos recursos (tais como CPU, memória e outros) ou usá-los por um período de tempo muito longo. Ainda, o *host* precisa fazer essas certificações sem esforço excessivo e efeitos deletérios ao desempenho geral do sistema [18].

2.3.2 Sistemas Operacionais Extensíveis

Em sistemas operacionais extensíveis, pode-se ter um núcleo agindo como *host*, com aplicações não-confiáveis agindo como produtores de código que baixam e executam código no espaço de endereçamento do núcleo [1]. Em muitas ocasiões, isso pode ser lucrativo. Nesse caso, o problema é como o núcleo pode saber se uma aplicação inerentemente não-confiável respeita os invariantes internos do mesmo [3].

2.3.3 Extensões a Linguagens de Programação de Alto-Nível

Uma outra aplicação é para o uso de extensões a códigos de linguagens de programação de alto nível seguras. Linguagens de programação de alto nível

PUC-Rio - Certificação Digital Nº 0124811/CA

são projetadas e implementadas com a hipótese de um mundo fechado. Ou seja, o programador deve normalmente assumir que todos os componentes do programa estão escritos na linguagem de alto nível sendo usada [3].

Na prática, porém, os programas possuem, geralmente, componentes escritos em linguagens de alto nível e em outras, que podem estar até em *assembly*. Em tais situações, perde-se as garantias providas pela linguagem de alto nível a não ser que mecanismos extremamente caros (tais como processos e *sockets*) sejam empregados [3].

Esse problema é agravado no campo de computação distribuída e na WEB, particularmente quando código móvel é permitido. Nesse tipo de situação, um agente A em uma parte da rede pode escrever um componente de um *software* em uma linguagem de alto nível, compilá-lo para código de máquina, e então transmiti-lo a um agente B em outro nó para execução. Como o agente A vai convencer o agente B de que o código enviado é uma extensão segura [3]?

Se o programador for convencido de que o componente escrito em outra linguagem é seguro através de PCC, não é preciso utilizar os mecanismos caros citados acima para proteger a execução do sistema.

PUC-Rio - Certificação Digital Nº 0124811/CA

2.4
Problemas Potenciais de PCC

Embora o uso de *proof-carrying code* possua muitas vantagens para as aplicações já descritas, esta técnica também possui alguns problemas difíceis de serem tratados, como os explicados abaixo:

2.4.1
O Problema do Tamanho das Provas

Sabe-se que alguns teoremas podem ter provas extremamente grandes. Isso é conseqüência do fato de o problema TAUT (TAUT é um problema de decisão onde se deseja saber se uma fórmula proposicional é uma tautologia ou não) ser coNP-completo (um problema de decisão é coNP-completo se seu complemento pertence à classe NP e se todos os outros problemas de coNP se transformam polinomialmente nele). Infelizmente, somente se P = NP haveria sistemas de prova onde todas as provas de teoremas seriam de tamanho polinomial [28, 32, 45]. Mas, para um sistema de PCC na prática, não é viável a

transmissão e a verificação de provas muito extensas, ou seja, provas de tamanho exponencial em relação ao tamanho do programa verificado [4].

Por exemplo, se um programa contém um DAG (*directed acyclic graph*, em inglês) de fluxo de dados, onde um valor computado é usado para mais de um propósito, uma prova usando um método ingênuo provavelmente irá provar várias vezes resultados sobre este valor para cada lugar em que ele é usado. Isso é uma conseqüência natural da maneira como as substituições são feitas no cálculo de Hoare. Utilizando eliminação de sub-expressões comuns - uma técnica de compiladores [44] - é possível fatorar as provas de programas para evitar essa duplicação [4].

Um outro exemplo de algo que pode levar uma prova a crescer muito, se ela estiver sendo aplicada a códigos binários, é um DAG de fluxo de controle. Se um endereço de programa L puder ser atingido via uma instrução de salto de vários pontos J_i, então a condição de verificação para L se torna parte das condições de verificação para cada uma das asserções J_i. Agora há várias cópias dessa fórmula, nas quais diferentes substituições serão realizadas durante o processo de prova. Um provador de teoremas convencional não vai perceber a similaridade das fórmulas, provando várias vezes resultados similares. Utilizando uma outra técnica da tecnologia de compiladores, chamada de *static single-assignment form* [17] (onde cada variável recebe exatamente uma atribuição em toda sua vida, isto é, seu escopo), é possível fatorar provas para evitar esse tipo de explosão [4].

Uma outra maneira de eliminar partes redundantes em provas é fatorar os teoremas em sub-teoremas, chamados de lemas. Desta forma, é possível substituir várias sub-provas similares por uma única. Com isso, será necessário enviar os lemas e suas provas junto com a prova principal. Este tipo de "engenharia de *software* e modularização" de lógica é muitas vezes feita informalmente. Mesmo assim, as provas ainda tendem a conter muitos componentes redundantes, tais como cópias de fórmulas e mais cópias de suas sub-fórmulas. Permitindo que lemas contenham computações limitadas, que regeneram os sub-termos durante a execução, é possível se obter um equilíbrio entre o tamanho da prova e o tempo da verificação da mesma [4].

Ainda assim, alguns programas podem ter provas de tamanho exponencial em relação ao tamanho do programa. Por exemplo, essa explosão tende a ocorrer em programas binários contendo uma grande seqüência de condicionais sem *loops* dentro deles [22].

PUC-Rio - Certificação Digital Nº 0124811/CA

2.4.2 Outros Problemas

Um resultado fundamental da matemática moderna é o teorema da incompletude de Gödel (que afirma que, na aritmética de Peano, sempre vai haver algumas proposições que não podem ser provadas nem verdadeiras e nem falsas através do uso de apenas regras e axiomas desse mesmo ramo). Portanto, um outro problema fundamental é a axiomatização da aritmética de Peano. Afinal, dado qualquer conjunto consistente e recursivo de axiomas desta teoria, há sentenças matemáticas verdadeiras que não podem ser derivadas do mesmo. Mesmo que os axiomas da aritmética sejam aumentados em número por um indefinido número de outros que também são verdadeiros, sempre haverá verdades matemáticas que não são formalmente deriváveis do conjunto aumentado de axiomas [4, 18, 21].

Na implementação de uma técnica de PCC, para se tentar contornar esse problema, pode-se simplesmente adicionar axiomas ao provador de teoremas e ao verificador de provas a medida em que são necessários. Entretanto, obviamente, essa não é uma abordagem escalável, e ainda há o problema filosófico no qual se pergunta se tais extensões à lógica devem ser permitidas ou não [18].

Também, devido à semelhança de *proof-carrying code* com a verificação de programas, visto que *proof-carrying code* é um caso especial, este problema não pode ser resolvido de forma totalmente automática. Na verdade, esse caso é ligeiramente mais fácil porque há a opção de se inserir verificações extras em tempo de execução (tal como é feito em isolamento de falha de *software*), a qual tem como efeito a simplificação do processo de prova ao custo da redução do desempenho [22].

Ainda assim, assegurar que a política de segurança é de fato obedecida pode ser um procedimento muito complicado: geralmente há várias maneiras de se permitir qualquer operação de ser realizada em qualquer sistema real, e a maioria deles possui certas brechas das quais um usuário mal-intencionado pode se aproveitar. Sendo que se ter a certeza de que todos os buracos foram fechados é algo muito difícil [18]. O que se faz, então, é simplesmente assumir que a especificação de segurança está correta.

PUC-Rio - Certificação Digital Nº 0124811/CA

2.5 Implementando a Validação de Provas

A validação de provas seria muito difícil se não se conhecesse métodos eficientes de codificação e verificação de provas. Este problema, no entanto, já foi bem estudado e solucionado através da teoria dos *frameworks* lógicos. Estes são meta-linguagens nas quais se pode especificar e usar sistemas dedutivos, e, portanto, são também aplicáveis ao estudo das linguagens de programação, especialmente sistemas de tipos e semântica operacional [19].

Como *frameworks* lógicos são uma forma de especificar sistemas dedutivos, pode-se usá-los para programas PCC. Para especificar um sistema dedutivo, é necessário especificar sua sintaxe abstrata e suas regras de inferência, e a linguagem deve prover meios concisos de representá-los. Além disso, eles devem ser capazes de checar a validade de deduções e detectar expressões sem significado estaticamente. Mais ainda, todo o processo deve ser feito de uma maneira computacionalmente eficiente [19].

Por fim, dois *frameworks* lógicos muito usados para *proof-carrying code* são o LF e o λProlog, algo que fica claro nos trabalhos relacionados. Outros dois também muito conhecidos são o HOL e o Isabelle [15, 20].

PUC-Rio - Certificação Digital Nº 0124811/CA

2.6 PCC no Mundo Real

Em [26], são abordados alguns tópicos sobre o uso da técnica de PCC em aplicações que não estejam restritas ao mundo acadêmico. Nele, estão explicadas as dificuldades para que *proof-carrying code* seja de fato utilizado. Alguns dos motivos que geram estas dificuldades são os seguintes:

- Poucos clientes potenciais são qualificados para apreciar PCC;
- *Proof-carrying code* não proporciona novas aplicações, apenas acelera a execução de código móvel a velocidades próximas a de código nativo confiável, mas não mais rápido do que isso;
- PCC deve competir com tecnologias similares tais como interpretadores com tipagem segura de Javascript, VBScript, etc. São necessárias ferramentas de produção de qualidade para essa tecnologia; e
- Muitas das falhas de segurança encontradas em sistemas são *bugs* nas especificações de políticas de segurança. Isso acontece porque é dificílimo

escrever o que se entende por segurança. Porém, *proof-carrying code* não lida com esse aspecto.

PUC-Rio - Certificação Digital Nº 0124811/CA