1
Introdução

1.1
Motivação

O advento da Internet mudou a vida de grande parte dos usuários de computadores, já que, atualmente, a maioria dos programas que os mesmos precisam não são comprados em pacotes fechados, e sim, são obtidos através da WEB. Desta forma, os resultados podem ser obtidos com muito mais rapidez: códigos prontos podem ser encontrados para a maioria das tarefas que precisam ser realizadas. Porém, uma questão relevante se apresenta: como ter a certeza de que o código se comporta como o esperado? O que pode evitar que sua ação venha a apagar arquivos, mandar dados secretos para alguém ou usar furtivamente precioso tempo de CPU [19]?

A solução ideal seria verificar a segurança do código, no entanto, esta opção pode consumir muito tempo. Outra solução é fazer com que o próprio código prove que ele é seguro. O conceito de *proof-carrying code* (PCC) é baseado nessa idéia: um programa carrega consigo uma prova de sua aderência a certas políticas de segurança [19].

1.2
Objetivos

Proof-carrying code diz respeito a um código que carrega a prova de que o mesmo é seguro. Para tanto, ele carrega uma prova a respeito de propriedades do próprio código. Portanto, os mesmos métodos formais usados para a verificação formal de programas podem ser utilizados para esta tecnologia. Um desses métodos formais é o cálculo de Hoare, que faz a prova de correção de um programa usando puramente análise sintática do texto do mesmo. Ele basicamente tenta provar que um programa que satisfaz uma dada pré-condição satisfará uma determinada pós-condição se o mesmo parar.

Assim, este trabalho tem por objetivo estudar como o cálculo de Hoare aplicado a códigos-fonte escritos em uma linguagem de programação imperativa,

um método formal para realizar a verificação de programas, pode ser útil à técnica de PCC.

A geração de provas de segurança é muito mais difícil do que a verificação das mesmas. Por isso, neste trabalho dá-se ênfase à geração de provas de correção de programas utilizando o cálculo de Hoare. Desta forma, quer-se tornar possível a geração de provas de segurança para PCC utilizando o cálculo de Hoare. Essa dificuldade de geração de provas resulta do fato de ser um problema indecidível, por se tratar de construção de provas de teoremas. Tem-se, ainda, a (provável) existência de provas de tamanho exponencial, já que não se sabe se P ≠ NP [28, 32, 45].

Logo, deseja-se estudar heurísticas para uma geração eficiente dessas provas de modo a torná-las o mais automática possível, sem que seja necessária a utilização de um provador de teoremas para verificar as sentenças que nelas aparecem e que devem ser válidas para que as provas sejam corretas.

Por fim, para validar as heurísticas pesquisadas e para auxiliar na pesquisa de novas, constrói-se um corretor de programas que as implementa e que é testado com diversos programas para tais tais finalidades.

1.3 Outras Abordagens ao Problema da Segurança

De acordo com [19], a solução mais clara para o problema da segurança é estender a idéia convencional de confiança ao domínio aqui estudado (programas obtidos através de fontes não-confiáveis, como a WEB). Note-se que existem problemas similares, embora em menor escala. Por exemplo, há os problemas de vírus e cavalos de Tróia que se espalham através de disquetes e pela Internet, que têm como solução o uso de apenas programas confiáveis, que venham com a garantia de uma marca. Pode-se fazer o mesmo aqui: programas podem ser assinados digitalmente pelo produtor de código. Esta idéia foi posta em uso por grandes corporações, incluindo a Sun e a Microsoft, que assinam suas classes de *applets* para convencer os usuários de que eles não contém código espúrio ou malicioso. Entretanto, os produtores de programas em questão nem sempre são grandes companhias, e um programador desconhecido em alguma parte remota do planeta garantindo que seu código é seguro pode não significar muito.

Outra abordagem possível é aquela que impede que o código suspeito acesse o sistema de arquivos, faça conexões a *hosts* arbitrários e realize outras operações potencialmente perigosas, algo feito pela Netscape. No entanto, essa solução tem o efeito prejudicial de minar o poder dos programas. Pode-se remover tais restrições sem reduzir a segurança realizando verificações em tempo de execução, mas isso seria, obviamente, muito caro em termos de desempenho.

Uma maneira mais elegante utiliza linguagens de programação com segurança na tipagem, tais como Java e o subconjunto seguro de Modula-3 [41]. Programas cujos tipos são checados possuem a “garantia” de não causar efeitos perniciosos pois operações que não são garantidas como seguras em tempo de execução são detectadas durante a compilação e, desta forma, não são permitidas. No entanto, isto não resolve todos os problemas. Produtores de código podem não querer disponibilizar o código fonte, e sim apenas o binário, que pode ser manipulado para produzir efeitos indesejáveis. Existem vários modos de se lidar com isto, como o compilador Java, que emite uma codificação (chamada de Java Bytecode) que é enviada ao usuário. Ele, então, checa sua consistência antes de executá-lo. Porém, geralmente estas verificações são computacionalmente caras e devem ser realizadas a cada execução do código se for desejado rodar o programa um determinado número de vezes. Possíveis maneiras de se resolver este problema é colocando outro compilador do lado do usuário. Este compilador converte o Bytecode em um executável binário.

A abordagem acima apresenta outra desvantagem: o conjunto de propriedades de segurança que se pode assegurar utilizando-a é limitada pela linguagem de programação. No caso das qualidades demandadas ao programa não serem facilmente expressadas na linguagem escolhida, pode ser necessário utilizar programas inseguros ou implementações ineficientes.

1.4 Trabalhos Relacionados

Em [1, 3], o código não-confiável está em código de máquina para o processador DEC Alpha e as especificações de segurança são escritas em extensões da lógica de primeira ordem. As extensões são predicados denotando requisitos de segurança específicos da aplicação, juntamente com suas regras de derivação. Como um meio de relacionar código de máquina a especificações,

é usado uma forma do gerador de condição de verificação de Floyd [18, 25] que extrai as propriedades de segurança do programa em código de máquina como um predicado de primeira ordem na lógica que está sendo utilizada. Este predicado deve, então, ser provado pelo produtor de código através do uso de axiomas e regras de inferência dadas pelo consumidor de código como parte da política de segurança. Ele possui a propriedade de que sua validade com respeito às regras da lógica predicativa e com às de tipos são uma condição suficiente para assegurar o respeito à política de segurança.

Esta técnica de PCC foi implementada utilizando o Edinburgh Logical Framework (LF) [42], que é essencialmente um λ-calculus tipado para codificar e checar as provas. O princípio básico do LF é que provas são representadas por expressões, enquanto predicados, por tipos. Portanto, para checar a validade de uma prova é necessário somente fazer a verificação de tipos em sua representação. Ou seja, o validador de provas é um verificador de tipos. É importante citar que o verificador de tipos em LF é independente da política de segurança ou da lógica usada. Já para provar os predicados, que são as condições de verificação, foi utilizada a linguagem de programação Elf, onde as provas dos mesmos foram produzidas na representação de LF. Elf é uma linguagem de programação lógica baseada em LF.

Um dos objetivos dessa pesquisa é mostrar como usar *proof-carrying code* para desenvolver extensões de *assembly* a programas escritos em ML de forma segura, ou seja, como permitir a usuários não-confiáveis ligar funções escritas em uma outra linguagem (*assembly*) a um ambiente seguro de execução de uma linguagem (ML). Em [3], o ambiente usado foi uma versão simplificada do compilador TIL para ML padrão.

Os mesmos autores da pesquisa anterior, em [18], estudam o problema sobre como estabelecer garantias a respeito do comportamento intrínseco de programas móveis. Eles desejam realizar interações eficientes entre os componentes do *software* (alguns dos quais podem vir de fontes não confiáveis), de tal maneira que certos invariantes que valem em cada componente não sejam violados pelos outros componentes. Para tanto, utilizam a técnica de *proof-carrying code*.

Em [4, 12], descreve-se a base de um projeto para a construção de um compilador para a certificação de códigos em ML e Java de alta qualidade, robusto e com otimização de código, permitindo interoperabilidade em essência e geração totalmente automática de *proof-carrying code* nessas linguagens reais. Particular atenção é dada à escalabilidade, à interoperabilidade, à

eficiência e ao princípio da interação entre políticas de segurança com mecanismos de contenção.

Também é mostrado como os clientes podem confiar em um conjunto de axiomas muito mais genérico, de forma que os produtores de código possam ter muito mais flexibilidade na escolha de linguagens de programação e compiladores.

Além disso, as políticas de segurança são generalizadas e simplificadas para permitir maiores confiança e flexibilidade nas próprias. Outro objetivo é o desejo de se usar PCC para expressar políticas de segurança bastante sofisticadas.

Nesta pesquisa é usada FLINT [43], uma linguagem intermediária tipada, para compilar códigos em linguagens orientadas a objetos, funcionais e imperativas. Linguagens intermediárias tipadas são uma tecnologia para a construção de compiladores para a certificação.

Os autores desse trabalho têm como meta construir um sistema para código móvel seguro baseado em segurança e lógicas de autenticação (lógicas que permitem a prova de propriedades de sistemas e protocolos que verificam a identidade dos usuários e decidem se certas operações podem ser permitidas ou não), compiladores para a certificação baseados em tipos e *proof-carrying code*. Eles trabalham com λProlog com mais representações de provas explícitas, e pretendem implementar alguns protótipos tanto em LF quanto em λProlog para comparar diretamente as vantagens e desvantagens de cada um, a fim de investigar qual a melhor infra-estrutura a ser utilizada no trabalho com PCC.

Já em [13, 14], é dito que as provas de PCC tradicional são especializadas em tipo, de forma que requerem axiomas de um sistema de tipo específico. Portanto, nestes trabalhos propõe-se o PCC fundamentado (FPCC), no qual as provas definem e provam explicitamente todas as propriedades requeridas desses tipos assumindo apenas uma fundamentação fixa em matemática como lógica de alta ordem. Portanto, ao contrário do que ocorre em *proof-carrying code* tradicional, neste tipo de PCC é evitado qualquer compromisso com um sistema particular de tipos. FPCC é a verificação através do menor conjunto de axiomas possível, usando o verificador mais simples e o menor sistema de execução possíveis [14]. Ainda, em [13], é explicado que para FPCC são necessários modelos semânticos de sistemas de tipos baseados em máquinas de Von Neumann e que *proof-carrying code* fundamentado funciona tanto em λ-calculus como em Pentiums.

Nessa pesquisa, a lógica utilizada é a lógica de alta ordem de Church (lógica predicativa clássica com termos do λ-calculus tipado [20]) com axiomas aritméticos. Essa lógica é representada e as provas são checadas no *framework* lógico LF, implementado no *framework* metalógico Twelf.

Em outro trabalho [19], são propostas técnicas que evitam o envio da prova inteira ao consumidor de código, prova esta que pode ser extremamente longa. São usadas idéias da área de provas verificáveis probabilisticamente da teoria da complexidade. Com isso, pode-se reduzir o custo de comunicação, no entanto, há a perda da certeza da segurança (o usuário tem uma alta probabilidade de ter certeza da segurança). Porém essa é a melhor solução apresentada no estado da arte atual, já que mandar a prova inteira pode vir a ser um processo muito longo graças ao envio de uma prova extensa.

Por fim, os autores de [15] usam o cálculo de Hoare para aplicações de PCC a fim de lidar com segurança de instruções de linguagem de máquina codificadas em um *framework* lógico. Neste artigo, fala-se sobre a importância de se implementar eficientemente as substituições quando as atribuições de um programa estiverem sendo tratadas.

Logo, são comparadas diferentes definições e implementações de substituições em um *framework* lógico, de forma a maximizar a eficiência. Em uma abordagem convencional, a definição de uma substituição como uma fórmula lógica é feita através da especificação de mudanças sintáticas que a substituição realiza nas expressões. Ao invés disso, os autores escolheram uma definição semântica que descreve a relação comportamental entre a expressão original e a correspondente depois da substituição.

Eles ainda propõem um método diferente de implementação, o qual faz melhor uso das primitivas providas pelo framework lógico, e é capaz de reduzir a complexidade a uma quantidade proporcional ao número de variáveis livres.

1.5 Organização do Texto

No capítulo dois é descrita a tecnologia de *proof-carrying code* em detalhes, explicando do que se trata, as vantagens de seu uso, suas aplicações principais e os problemas do crescimento do tamanho das provas, da axiomatização da aritmética de Peano, da automatização da geração de provas e outros mais.

No capítulo três, faz-se a introdução e a explicação sobre o que é a semântica axiomática de uma linguagem de programação, utilizada no cálculo de Hoare. Posteriormente, é abordada a correção de programas utilizando esta mesma lógica. Depois, seus axiomas e regras de inferência são dados e explicados para alguns comandos e estruturas de uma linguagem imperativa de alto nível. Também, discute-se o problema de se encontrar o invariante de um *loop*, devido a sua importância para se provar a correção de um programa. Apresenta-se, ainda, um comentário de como a prova de uma sub-rotina pode ser usada como um lema na prova de um programa. Por fim, discute-se sobre o uso do cálculo de Hoare para linguagens imperativas de alto nível para *proof-carrying code* e é dado um exemplo do uso do cálculo de Hoare para provar segurança para uma das aplicações de PCC.

Já no capítulo quatro, começa-se falando da necessidade de haver heurísticas para a construção eficiente de provas, comparando tal processo com a normalização para a dedução natural da lógica clássica de primeira ordem. Posteriormente, aborda-se cada uma das heurísticas descobertas utilizando uma versão preliminar do corretor de programas descrito no capítulo a seguir. Também descreve-se os problemas encontrados com o tipo de dados vetor e com o crescimento exagerado das sentenças a serem provadas e das asserções.

No capítulo cinco, descreve-se a implementação do corretor de programas, dando ênfase à implementação das regras do cálculo de Hoare e das heurísticas definidas para a geração de provas. Ainda, descreve-se a técnica de substituição de variáveis quando há quantificadores.

Quanto ao capítulo seis, nele são abordadas duas técnicas para verificação formal distintas da que foi utilizada nos capítulos anteriores, que são a álgebra de Kleene com testes e a lógica dinâmica. Ambas também são comparadas com o cálculo de Hoare, analisando-se as vantagens e desvantagens de cada uma.

Por fim, no capítulo sete, tem-se a conclusão deste trabalho, onde é enfatizado o porquê de se utilizar o cálculo de Hoare para a geração das provas de PCC. Ainda, explica-se o motivo de o corretor de programas ter sido implementado na linguagem Prolog e aborda-se os trabalhos futuros que podem vir a ser frutos do presente texto.