Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Programa de Pós-Graduação em Ciência da Computação

Dissertação de Mestrado

Modelo de Segurança Independente de Plataforma para Execução de Software Não Confiável

Tales Ribeiro Morais Gurjão

Campina Grande, Paraíba, Brasil
Setembro de 2016

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Programa de Pós-Graduação em Ciência da Computação

Modelo de Segurança Independente de Plataforma para Execução de Software Não Confiável

Tales Ribeiro Morais Gurjão

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de software

Hyggo Oliveira de Almeida
Angelo Perkusich
(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

G979m Gurjão, Tales Ribeiro Morais.

Modelo de segurança independente da plataforma para execução de software não confiável / Tales Ribeiro Morais Gurjão. – Campina Grande, 2016.

67 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática. 2016.

"Orientação: Prof. Dr. Hyggo Oliveira de Almeida, Prof. Dr. Angelo Perkusich".

Referências.

1. Modelo de Segurança. 2. Códigos Maliciosos. 3. *Malware*s. 4. Isolamento de Ambiente. I. Almeida, Hyggo Oliveira de. II. Perkusich, Angelo. III. Título.

CDU 004.056.57(043)

"MODELO DE SEGURANÇA INDEPENDENTE DE PLATAFORMA PARA EXECUÇÃO DE SOFTWARE NÃO CONFIÁVEL"

TALES RIBEIRO MORAIS GURJÃO

DISSERTAÇÃO APROVADA EM 01/09/2016

HYGGO OLIVEIRA DE ALMEIDA, D.Sc, UFCG
Orientador(a)

ANGELO PERKUSICH, D.Sc, UFCG
Orientador(a)

KYLLER COSTA GORGÔNIO, Dr., UFCG
Examinador(a)

ANA LUISA FERREIRA DE MEDEIROS, Dra., UFCG
Examinador(a)

LENARDO CHAVES E SILVA, Dr., UFERSA
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Ataques a sistemas informatizados sempre foram um problema e evoluíram de simples investidas contra instalações físicas nos anos de 1970 a ataques coordenados usando milhares de computadores espalhados ao redor do mundo. Essas ofensivas têm como principal vetor códigos maliciosos, também conhecidos como *malwares*, que por vezes se passam por benignos mas se instalam no sistema e agem de forma maligna. Técnicas de isolamento de ambiente de execução e detecção de comportamento são empregadas para mitigar o risco ao executar um código desconhecido e potencialmente perigoso. Porém, muitas alternativas são custosas e, por vezes, dependem de ferramentas externas. Neste trabalho, propõe-se um modelo independente de plataforma para prover segurança na execução de códigos não confiáveis, sem efeitos colaterais para o hospedeiro e para terceiros. O modelo de segurança desenvolvido é constituído de dois módulos principais, analisador e executor, os quais (a) extraem metadados referentes ao programa e os utiliza para realizar uma análise prévia do código e (b) realizam checagens em tempo de execução que objetivam a preservação da integridade do sistema e dos recursos associados. A validação da abordagem foi realizada através de estudo de caso de aplicação em computação voluntária.

Palavras-Chave: códigos maliciosos, malware, isolamento de ambiente, modelo de segurança.

Abstract

Attacks on computer systems have always been a problem and have evolved from simple attacks against physical facilities in the 1970s to coordinated attacks using thousands of computers spread around the world. The main vector of these offensives are softwares that sometimes pass by benign programs but when are installed in a system act in a malicious manner. Environment isolation and behavior detection techniques are used to mitigate the risk of running an unknown and potentially dangerous code. However, many alternatives are expensive and sometimes requires external tools. In this paper, we propose a platform-independent model to provide security to execute untrusted code with no side effects to the host and to third parties. The model consists of two main modules, analyzer and executor, which (a) extracts metadata related to the program and uses them to conduct a preliminary analysis of the code and (b) carry out checks at runtime aimed to preserve the integrity of the system and its associated resources. The validation of the approach was performed by a case study on a volunteer computing application.

Agradecimentos

Primeiramente a meus orientadores, Prof. Dr. Hyggo Oliveira de Almeida e Prof. Dr. Angelo Perkusich, pela confiança, paciência e orientação concedida.

Aos meus pais, que me deram a vida e o saber.

À minha família por entender os momentos em que tive que ficar ausente para cursar o mestrado.

A Deus, por ter me dado saúde, paciência e sabedoria.

À CAPES pela concessão da bolsa de estudos, o que viabilizou a realização deste trabalho.

Aos professores e funcionários da COPIN que foram sempre solícitos.

Aos Professores das Disciplinas cursadas neste Mestrado que seguramente contribuíram com a fundamentação de minha pesquisa.

À Universidade Federal de Campina Grande, ao Centro de Engenharia Elétrica e Informática, ao Departamento de Sistemas e Computação e ao Laboratório de Sistemas Embarcados e Computação Pervasiva (Embedded) que apoiaram integralmente a realização do meu trabalho.

Conteúdo

Lista de Símbolos

SMS	*Serviço de mensagens curtas*
MMS	*Serviço de mensagens multimídia*
CPU	*Unidade central de processamento*
DoS	*Negação de serviço*
RBAC	*Controle de acesso baseado em papéis*
VM	*Máquina virtual*
VMM	*Gerenciador de máquina virtual*
API	*Interface de programação de aplicações*
IO	*Entrada e saída*
UID	*Identificador do usuário*
GPS	*Sistema de posicionamento global*
TCP	*Protocolo de controle de transferência*
IP	*Protocolo de internet*
JSON	*Notação de objeto em JavaScript*
URL	*Localizador uniforme de recursos*
AOT	*Ahead of time*
JIT	*Just in time*
IPC	*Intercomunicação de processos*
HTTP	*Protocolo de transferência de hipertexto*
SO	*Sistema operacional*
REST	*Transferência de estado representacional*
MB	*Mega bytes*
RAM	*Memória de acesso aleatório*

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

A segurança de sistemas computacionais data da segunda guerra mundial, na qual os *mainframes* foram utilizados para descriptografar mensagens inimigas. As necessidades iniciais de segurança desses sistemas eram basicamente evitar o roubo de equipamentos, espionagem dos produtos desenvolvidos e até sabotagem. Para tanto, esses computadores eram instalados em locais com um rígido controle de acesso, como instalações militares. O controle de acesso até então era feito de forma muito simples com o uso de distintivos e reconhecimento facial por parte do funcionário encarregado da segurança. Porém, à medida que a tecnologia evoluiu e os computadores passaram a ser usados em larga escala, novos mecanismos de segurança tiveram que ser adotados, principalmente quando os computadores passaram a se comunicar através de redes na década de 60 [33].

Desde então, a segurança de computadores passou a ter uma importância muito grande, visto que não apenas equipamentos deverão ser protegidos, mas também dados que podem colocar em risco milhões de pessoas. Wittman e Mattord [33] definem que uma organização deve ter várias camadas de segurança para proteger suas operações: segurança física para proteger contra acesso não autorizado ou mau uso de equipamentos e itens físicos; segurança pessoal para proteger indivíduos que podem acessar determinados recursos; segurança operacional para proteger os detalhes de uma atividade; segurança de comunicação para proteger meios de comunicação; segurança de rede para proteger conexões e conteúdos trafegados; segurança de informações para proteger a confidencialidade, integridade e disponibilidade de recursos.

Com a evolução dos computadores foi possível criar diversos tipos de aplicações, gerar e

armazenar, cada vez mais, uma grande quantidade de dados. Essas características tornam os sistemas muito atrativos a pessoas mal intencionadas que desejam explorar vulnerabilidades dos sistemas como um todo, mais especialmente falhas de software, para atacar e roubar dados ou realizar outras ações que possam causar danos a terceiros.

Os softwares que são maliciosos, conhecidos como *malwares*, estão atuando cada vez mais em dispositivos pessoais e móveis (smartphones e tablets) devido à sua grande fatia no mercado[1] e a disponibilidade de informações e recursos que podem utilizados. Estima-se que a maioria dos códigos maliciosos para dispositivos móveis tenham a intenção de roubar dados pessoais e enviar mensagens via serviços de mensagens de texto (SMS)[9], o que eventualmente gera prejuízo aos usuários. No relatório publicado pelo grupo Kaspersky[2], afirma-se que apenas no terceiro trimestre de 2015 surgiram mais de 320 mil novos programas maliciosos para dispositvos móveis. Além disso, o relatório afirma que o tipo de aplicação mais atacada são as desenvolvidas para navegadores web, concentrando 58% dos ataques, seguido por aplicativos para a plataforma Android, somando 19% do total.

Aplicativos antivírus foram desenvolvidos para identificar esses códigos maliciosos e, com isso, diminuir o dano em potencial que estes podem causar. O mecanismo de detecção baseia-se principalmente na análise do código em busca de padrões, ou assinaturas, que são previamente conhecidos pelo software antivírus. O ponto negativo desse tipo de proteção é que deve haver uma base de dados muito atualizada, caso contrário, o código malicioso não poderá ser identificado [3]. Em [3] é apresentada uma alternativa a essa análise estritamente estática de código ao mapear o comportamento de um software dentro de um ambiente de execução isolado, o qual protege o restante do sistema.

Técnicas para isolar a execução de aplicativos já são empregadas há vários anos, em especial na área de *grid computing*, no qual a virtualização é utilizada para prover um ambiente seguro enquanto garante a homogeneidade do sistema de execução[22]. Outras técnicas também são utilizadas para criar ambientes seguros de execução, tais como controle de acesso orientado a usuários e controle de acesso orientado a aplicações[31]. De modo geral as técnicas que isolam processos e limitam o acesso a recursos são chamadas de *sandbox*, pois, o programa consegue executar apenas dentro do seu espaço, ou *sandbox*, sem que as ações

[1] http://www.gartner.com/newsroom/id/2944819

[2] https://securelist.com/analysis/quarterly-malware-reports/72493/it-threat-evolution-in-q3-2015/

sejam refletidas fora do ambiente confinado (outros processos ou sistema).

1.1 Problemática

Códigos maliciosos sempre foram um problema no mundo da computação e diversas soluções foram propostas ao longo dos anos para poder identificar quando um código é malicioso. Porém, poucas soluções independentes de plataforma foram propostas para contornar esses problemas e muitas soluções comerciais implementam mecanismos inteligentes com o objetivo de detectar, analisar e reportar comportamentos maliciosos ou suspeitos[3] [4] [5]. Essas ferramentas são utilizadas, principalmente, para perfilar o comportamento de um código ou programa, sem o intuito de permitir que o software seja executado de forma segura independentemente das ações que o mesmo realiza. Há também alguns exemplos de soluções que sequer executam num disposivo físico, mas sim na nuvem[6], porém com o mesmo objetivo: identificar e analisar o comportamento de softwares maliciosos.

A maioria das ferramentas existentes não têm como objetivo executar um código de forma isolada, assim como outros trabalhos majoritariamente teóricos que propõem em sua maioria modelos que interceptem e analisem as ações de softwares. Dessa forma, o principal problema abordado nesse trabalho pode ser definido como um modelo independente de plataforma que permita a execução de códigos arbitrários mesmo que estes não sejam confiáveis.

1.2 Objetivos

Neste trabalho o objetivo principal é a elaboração de um modelo de segurança independente de plataforma que permita a execução de códigos não confiáveis sem que hajam efeitos colaterais indesejados tanto para o usuário quanto para o dispositivo e até para terceiros. Os efeitos indesejados para o usuário estão relacionados ao acesso não autorizado a informações pessoais. Já para o dispositivo incluem-se roubo de recursos, tais como alto uso da CPU

[3] https://cuckoosandbox.org/
[4] http://www.joesecurity.org/
[5] http://www.threatexpert.com/
[6] https://anubis.iseclab.org/

(unidade de processamento central) e rede além de acesso ou danificação de arquivos. Os prejuízos causados para terceiros são caracterizados quando o código usa indevidamente recursos de um dispositivo para realizar ataques a outros dispositivos ou serviços. O roubo de recursos computacionais apesar de não causar um prejuízo diretamente, é algo grave pois em caso extremo pode ocasionar DoS (negação de serviços) local e impossibilitar que outros programas possam ser executados.

Dessa forma, o modelo elaborado deve ser robusto o suficiente para prevenir que tais ações sejam executadas e também deve ser flexível o suficiente para poder executar em dispositivos de diversas plataformas, incluindo dispositivos móveis pessoais, tais como smartphones e tablets. O esquema básico de segurança utiliza técnicas de sandbox, análise estática e mediador de acesso a recursos para criar um ambiente de execução de códigos seguro e isolado do ambiente do usuário e de outros aplicativos.

Os seguintes objetivos específicos podem ser definidos com base no objetivo geral:

1. Definição do modelo de segurança: uma vez que os dispositivos alvo do modelo podem guardar informações pessoais ou até sigilosas, é necessário assegurar que códigos maliciosos não consigam acessar tais informações tanto para garantir a segurança do usuário quanto para garantir a integridade dos dados. Além de proteger os dados pessoais, o modelo deve evitar que os códigos que executem consigam acessar espaço de outras aplicações e que façam uso de recursos de rede, disco, CPU e memória de forma excessiva para não transformar o dispositivo em um *zumbi* na rede ou gerar DoS local. Para tanto são utilizadas técnicas que não impliquem numa perda considerável de desempenho.

2. Definição do mecanismo de autorização de recursos: para garantir que o código não acesse recursos indesejados que, dependendo do sistema, podem ser críticos, o modelo de segurança também deve ser capaz de permitir (ou proibir) que determinados códigos acessem recursos específicos, que podem ser: mecanismos de comunicação em rede (wi-fi, bluetooth, etc), acesso a componentes de hardware (teclado, drives óticos, etc), acesso a arquivos públicos, entre outros.

3. Modelagem geral: após a finalização dos objetivos anteriores o modelo foi elaborado através de artefatos como diagramas de componentes, sequência e definição formal

dos requisitos.

1.3 Relevância

O modelo proposto deverá permitir que vários tipos de aplicações sejam executadas de forma segura em diversos dispostivos, incluindo dispositivos com limitações de hardware como smartphones e tablets. Dentre os tipos de aplicações que poderão usar o modelo podemos destacar os detectores de comportamento de códigos, os quais permitem que a aplicação execute num ambiente controlado e, assim, monitoram as atividades a fim de determinar se o código realiza ações maliciosas e também pode extrair características como consumo de rede, memória e CPU.

Outro tipo de aplicação que pode implementar esse modelo são as que envolvem computação distribuída tendo em vista que o interesse em sistemas de grids móveis está crescendo. Assim, será possível ampliar o espectro de dispositivos que participam desses sistemas ao integrar dispositivos móveis e pessoais e trazer benefícios para as entidades que desejem reduzir custos relativos ao processamento, uma vez que não é necessário ter computadores nem gastos com infraestura (espaço físico, refrigeração, rede, energia elétrica, etc).

1.4 Metodologia

Para elaborar um modelo de segurança eficaz na execução de códigos maliciosos diversos é necessário entender como esse tipo de código atua a fim de identificar como eles se disseminam e quais componentes são mais explorados e vulneráveis. A metodologia desse trabalho consistiu em:

1. Revisão bibloiográfica
    (a) Sobre técnicas de confinamento de código
    (b) Sobre detecção e ação de códigos maliciosos
2. Estudo de ameaças recentes
    (a) Estudo de *malwares* recentes

(b) Definição dos pontos de ataques mais críticos em uma aplicação

3. Elaboração de um modelo conceitual

4. Implementação do modelo

   (a) Definição de tecnologia e arquitetura

   (b) Implementação do modelo

5. Validação

   (a) Criação de códigos que simulem o comportamento de malware

   (b) Execução dos códigos maliciosos sob a implementação do modelo de segurança

1.5 Organização do Documento

Este capítulo foi introdutório com informações gerais acerca do problema, objetivos, relevância e metodologia sobre o trabalho realizado.

A fundamentação teórica sobre sistemas de controle de acesso, códigos maliciosos e *sandbox* é apresentada no Capítulo 2. Alguns trabalhos relacionados à detecção e classificação de *malwares* bem como técnicas utilizadas para isolar ou executar esses códigos maliciosos são apresentadas no Capítulo3.

No Capítulo 4 apresenta-se um estudo sobre os *malwares* apontando suas principais características como formas de atuação, propagação, evasão e a segmentação com relação ao tipo e plataformas mais atingidas. Posteriormente são apresentadas algumas técnicas utilizadas para proteger dispositivos e, finalmente, são apresentados os requisitos do modelo proposto neste trabalho.

O modelo proposto é explicado no Capítulo 5 com uma divisão em duas partes: análise prévia de código, chamada de pré processamento, e o ambiente de execução, chamado de executor. No Capítulo 6 é apresentado um estudo acerca do modelo através da implementação de um software de computação voluntária. Por fim, no Capítulo 7 apresentam-se as conclusões gerais.

Capítulo 2

Fundamentação Teórica

Nessa seção são apresentados os conceitos elementares de segurança e alguns mecanismos básicos usados em outros esquemas de segurança mais robustos, além de uma introdução ao conceito de códigos maliciosos. Posteriormente são apresentados alguns modelos que fazem uso de *sandbox* e suas variações.

2.1 Controle de Acesso

Controles de acesso foram empregados em sistemas comerciais e militares desde os anos 1970 com a simples idéia de que pessoas possuem papéis com diferentes responsabilidades e privilégios dentro uma organização. Essa ideia foi utilizada para criar um mecanismo rudimentar de segurança no qual os usuários tinham diferentes níveis de acesso dentro de um sistema baseado apenas no "papel" que lhe era atribuído, dessa forma os usuários eram segregados dentro dos sistemas computacionais de maneira análoga ao que acontece nas organizações[11]. Tal mecanismo é conhecido como controle de acesso baseado em papéis ou regras (*RBAC*).

Nos sistemas de controle puramente baseados em regras, entidades ativas (também chamadas de *subjects*) possuem todos os privilégios numa máquina local, independentemente do tipo de aplicação sendo executada, uma vez que a restrição de acesso se aplica aos recursos compartilhados da organização e não ao acesso em si e privilégios de execução em uma máquina local[31].

Como observado na Figura 2.1, cada objeto é acessível apenas através de papéis, os

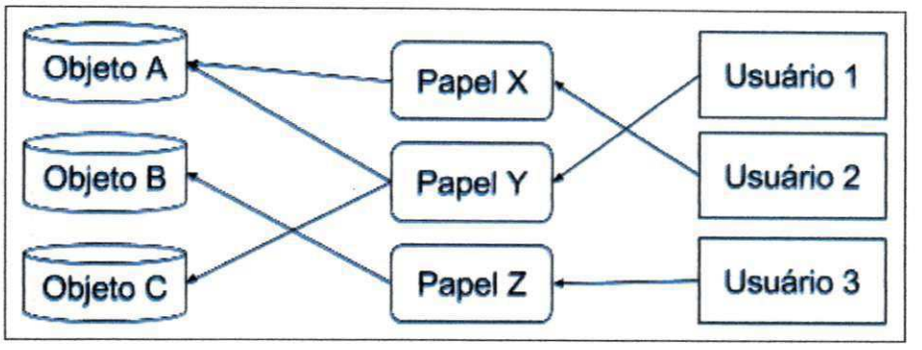

Figura 2.1: Diagrama de controle de acesso baseado em papéis

quais são atribuídos a usuários. Dessa forma, todos os usuários que precisam ter acesso a um determinado objeto devem ter um papel correspondente. Por exemplo, o *Usuário 3* tem acesso apenas ao *Objeto B*, o *Usuário 2* tem acesso apenas ao *Objeto A* e o *Usuário 1* tem acesso a ambos *Objeto A* e *Objeto C*. Com base nesse esquema um usuário pode migrar de papel e, assim, adquirir novos privilégios de acesso. Três regras básicas definem como o esquema *RBAC* deve funcionar:

1. Atribuição de papéis: um *subject* pode executar uma transação somente se o *subject* em questão foi selecionado ou teve o papel atribuido a ele.

2. Autorização de papéis: um papel ativo de um *subject* deve ser autorizado para o *subject*.

3. Autorização de transação: um *subject* pode executar uma transação somente se a transação está autorizada para o papel ativo do *subject*.

Devido ao fato de que a segurança reside apenas em nível de acesso a objetos compartilhados, esses sistemas são vulneráveis a falhas de software ou malwares, uma vez que podem rodar processos locais maliciosos independentemente do nível de acesso dos *subjects* e possibilitar que atacantes ganhem acesso aos recursos protegidos e executem ações maliciosas. O impacto desses ataques pode ser significativo, pois o software malicioso irá executar após roubar e assumir a identidade de algum usuário autenticado e poderá utilizar todos os provilégios atribuídos a ele. Esse tipo de proteção, se usado isoladamente, é útil para proteger recursos do sistema e objetos de usuários de outros usuários, mas não con-

segue proteger de forma eficiente os usuários e recursos de outras aplicações, em especial, aplicações maliciosas[31].

2.2 Códigos Maliciosos

Códigos maliciosos ou *malware* (abreviação do inglês de *malicious software*) são códigos que executam em um computador hospedeiro e podem causar danos ao usuário ou computador no qual são executados a partir da exploração de falhas dos softwares alvos, incluindo os sistemas operacionais[18]. Os malwares são geralmente diferenciados pelo modo o qual se propagam e o pelo modo que executam, tendo uma taxonomia muito diversidicada como *trojan*, *virus*, *worm*, *adware*, *spyware*, *rootkit*, entre outros[1].

Os *malwares* são usados principalmente pela capacidade de automação e infecção, onde podem comprometer uma variedade muito grande de dispositivos. Esses códigos maliciosos evoluíram bastante para não explorar somente falhas de engenharia, mas também explorar os usuários que por falta de conhecimento ou porque foram induzidos abrem emails ou websites infectados[26].

Além dos ataques tradicionais que infectam e comprometem computadores de forma individual, há os *botnets* que funcionam como uma espécie de sistema distribuído de *malwares*, os quais coordenam a propagação e execução de códigos maliciosos em múltiplos computadores e sistemas de forma simultânea[6].

2.3 Execução Seletiva

Tendo em vista que os sistemas tradicionas de controle baseado em regras permitem a execução de programas com todas as permissões do usuário, criou-se uma técnica simples que pode reduzir os riscos e consiste apenas em categorizar os programas e executar os que são marcados como confiáveis. Com essa técnica os programas ainda rodam no espaço do usuário que estiver autenticado, porém reduz o risco de execução de malwares. Essa técnica pode ser baseada em *whitelist* ou *blacklist*[20] na qual há uma lista de aplicações confiáveis (*whitelisted*) ou uma lista de aplicações que são proibidas de executar (*blacklisted*).

[1] http://www.malwaretruth.com/the-list-of-malware-types/

O método mais simples de criar *whitelists* ou *blacklists* é criar um banco de dados com os nomes (ou *paths*) dos arquivos ou programas que são confiáveis. Porém, malwares conseguem facilmente burlar esse mecanismo ao simplesmente alterar o nome dos arquivos para algo que esteja na *whitelist* ou para algo que não esteja na *blacklist*. Para combater esse problema, armazenam-se *fingerprints* dos arquivos ao invés dos nomes, dessa forma cada arquivo contém um identificador único e evita que um *malware* se passe por outro programa [20]. Mesmo assim, alguns sistemas possuem uma interface que permite que o usuário execute programas que não estão na *whitelist*.

Apesar desse mecanismo funcionar para barrar aplicações conhecidamente maliciosas, vulnerabilidades podem ser introduzidas no sistema de forma involuntária, tais como falha no design do projeto ou implementação, resultando num ponto de entrada para ataques.

Outra técnica explicada em [12], utiliza um modelo baseado em reputação no qual há uma base de dados que relaciona arquivos, reputação e níveis de confiança, de forma que esses valores são atualizados constantemente e utilizados para decidir se o arquivo pode ser executado, ao invés de simplesmente usar uma *whitelist* ou escanear o arquivo a procura de códigos maliciosos. Essa forma de proteção traz o benefício de detectar possíveis ameaças automaticamente antes mesmo do arquivo em questão ser adicionado à lista de ameaças confirmadas.

Na Figura 2.2 ilustra-se o fluxo de execução em um sistema baseado em *whitelist*. Quando a aplicação tentar executar o sistema a intercepta e verifica se a mesma está na base de dados *whitelist*, caso esteja, a aplicação é executada normalmente e caso não esteja alguma ação é executada pelo sistema, como exibir um aviso ao usuário.

2.4 Controle de Acesso Orientado à Aplicação

Controles de acesso orientados à aplicação conseguem suprir uma falha dos sistemas tradicionais de controle baseado em regras no sentido de que as permissões são garantidas à aplicação com base no que ela se propõe a fazer, ao invés de dar permissão para todas as aplicações de um usuário. Essa abordagem reduz a possibilidade de uma aplicação acessar objetos fora do seu escopo e executar ações que causem danos[29].

Com esse propósito, o controle é feito através de políticas de acesso as quais podem ser

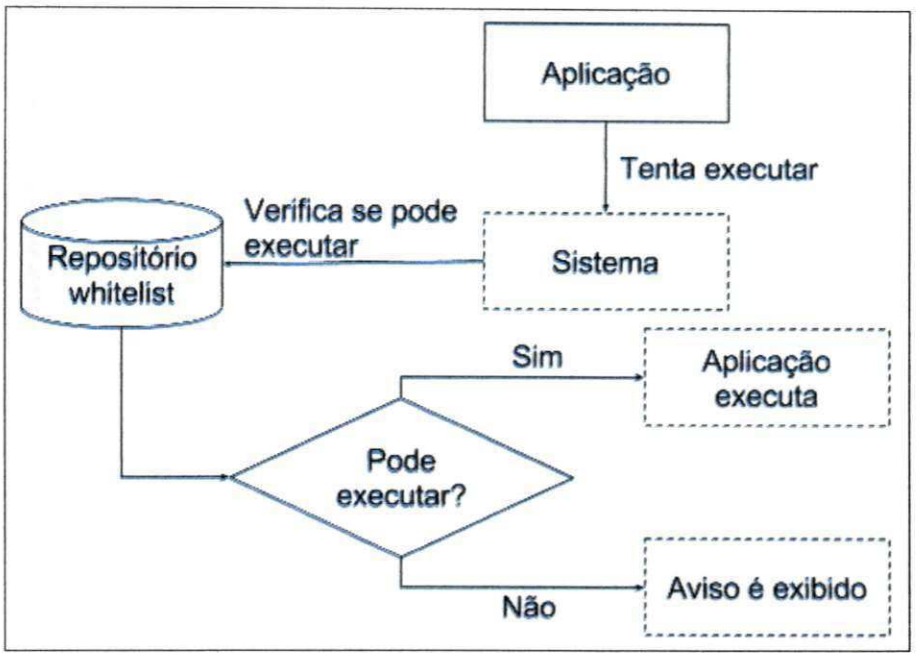

Figura 2.2: Esquema de execução baseado em whitelist

definidas de várias formas, tais como restrições aplicadas à própria plataforma e arquivos de configuração. Restrições também podem ser discricionárias, ou seja, o usuário tem a possibilidade de controlá-las ou podem ser não discricionárias, as quais são impostas ao usuário. Em [30] é proposta uma linguagem para especificação de políticas de controle de acesso orientado à aplicação a qual define três tipos de arquivos de políticas: confinamento, que especifica conjuntos de restrições e usuários que podem alterá-los; políticas de aplicação, que especificam como as aplicações são identificadas e restringidas; funcionalidades, que são módulos utilizados para especificar as políticas de aplicação.

Um exemplo prático é o sistema de segurança da plataforma Android, o qual utiliza como parte do mecanismo geral de segurança uma forma de controle de acesso orientado a aplicações [8] no qual o desenvolvedor inclui todas as permissões que o aplicativo necessita no arquivo *Manifest.xml*. Essas permissões incluem, entre outras, acesso ao sistema de arquivo, comunicação em rede, utilização de recursos de hardware (bluetooth, câmera, sensores), acesso a métodos de pagamento e dependendo do grau de segurança podem ser discricionárias ou obrigatórias.

2.5 Controle de Acesso Orientado a Isolamento de Aplicação

Outra forma utilizada para restringir o acesso de um programa a recursos é o confinamento do código, de forma que o ambiente de execução do programa seja diferente do restante do sistema. Esse confinamento é usado como uma espécie de contêiner no qual o contexto interno fica isolado do contexto que iniciou sua execução. Uma forma de confinamento muito conhecida é *sandbox* e, geralmente, é usada para isolar tipos muito específicos de aplicações, tais como visualizadores de documentos, *applets*, executor de *scripts*, dentre outros. Além disso, são usados para executar de forma proposital *malwares* a fim de estudar seu comportamento[15].

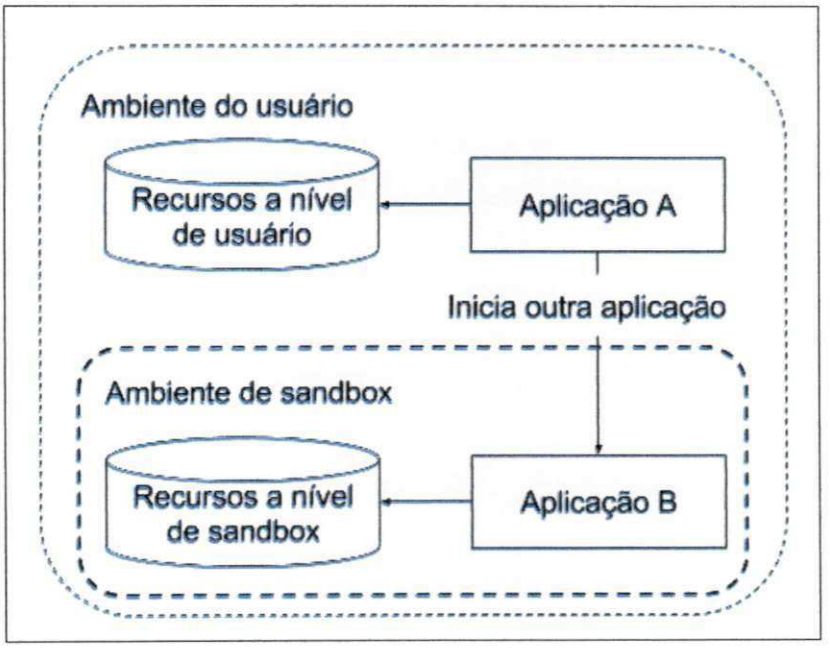

Figura 2.3: Sandbox baseado em isolamento

O funcionamento básico de um *sandbox* baseado em isolamento é ilustrado na Figura 2.3 na qual a *Aplicação A* executa no ambiente do usuário com acesso a todos os recursos da máquina local e inicia a *Aplicação B* que irá executar em um *sandbox*. Uma vez que a *Aplicação B* está confinada ao ambiente do *sandbox*, esta não terá acesso aos mesmos recursos que *Aplicação A*, mas sim ao recursos providos apenas para o *sandbox*. Caso a *Aplicação*

B inicie outros processos, estes herdarão seus privilégios e também ficarão confinados ao ambiente do *sandbox*.

Além de *sandboxes*, outros métodos de virtualização podem ser empregados para isolar aplicações [17]. O princípio básico da virtualização é criar uma cópia virtual de outras coisas, por exemplo, um sistema operacional, e caso essa cópia seja de um sistema computacional com recursos de hardware dá-se o nome de máquina virtual (VM)[31].

Tendo em vista que VMs utilizam diversos recursos de hardware e software, faz-se uso de outra camada para abstrair as diferenças que possam existir entre os sistemas. Essa camada é conhecida como monitor de máquinas virtuais (VMM) e possui 3 características importantes[17]:

1. Fidelidade: o ambiente de hardware criado para executar a VM deve ser em sua essência idêntico ao ambiente original;
2. Isolamento e segurança: o VMM deve ter total controle dos recursos do sistema;
3. Desempenho: deve haver uma diferença mínima de desempenho entre a VM e um dispositivo real.

Devido à evolução e surgimento de novas CPUs, memória e hardware, em geral o gerenciamento de recursos torna-se uma tarefa muito complexa mas também permite a criação de novos tipos de virtualização que levam em conta o tipo e o nível do sistema. Dentre os tipos de virtualização pode-se destacar[17]:

- Virtualização de hardware - virtualização com abstração da lógica e de alguns componentes;
- Virtualização de infraestrutura - abstração de componentes básicos da arquitetura dos computadores, tais como armazenamento e rede e é muito utilizada para construir *clusters* de alta capacidade;
- Virtualização de sistema operacional - um dos principais tipos e permite que o mesmo kernel seja usado por múltiplas instâncias que irão rodar em diferentes espaços de usuário.

2.6 Controle de Acesso Orientado à Aplicação Baseado em Regras

Controles de acesso baseados em regras que visam o confinamento de aplicações são, de modo geral, alternativas que representam um baixo impacto em termos de performance mas, ainda assim, conseguem lidar com questões de segurança de forma eficiente. Nesta seção são abordados dois métodos principais para confinamento baseado em regras: uso de regras em ambientes de sandbox e regras implementadas a nível de sistema.

2.6.1 Sandboxes Baseados em Regras

Alguns modelos de *sandboxes* podem empregar uma técnica similar ao controle de acesso orientado à aplicação, no qual um programa não é completamente isolado dentro do *sandbox*, mas é executado com respeito a políticas e regras. Essa forma de *sandbox* permite que aplicações executem no mesmo espaço de usuário e compartilhem recursos, desde que as respectivas políticas permitam isso.[31]. O framework Android faz uso de ambas as soluções em níveis distintos[2]:

- Nível de aplicação: é o nível mais alto do framework e contém as permissões específicas de aplicação definidas pelos desenvolvedores. O sistema de segurança realiza a checagem dessas permissões através do componente "permission check", e as exibe para que o usuário decida quais podem ser executadas. Esse tipo de permissão geralmente inclui acesso a API's (interface de programação de aplicações) protegidas tais como acesso a informações pessoais e acesso a dispositivos que realizam operações de entrada e saída (IO) a exemplo de câmeras, adaptadores bluetooth e wifi, etc.

- Nível de kernel: é o nível mais baixo e utiliza mecanismos de segurança de sistemas linux tais como SELinux[3] além da técnica *unique user id* (UID) a qual executa cada aplicativo em um espaço de usuário distinto criando, assim, um *sandbox de aplicação*. Nesse nível de segurança, políticas de acesso e permissões para aplicativos e processos podem ser definidas através dos componentes do *SELinux*.

[2]https://source.android.com/security/

[3]http://selinuxproject.org/page/Main_Page

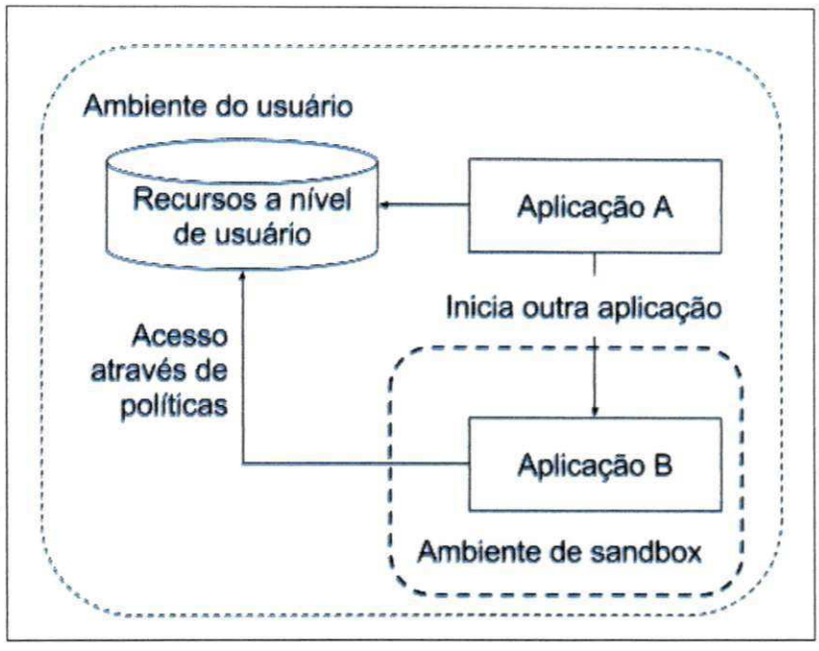

Figura 2.4: Sandbox baseado em regras

Dessa forma é possível ter uma estrutura de *sandbox* com aplicações executando em seu próprio espaço e isoladas umas das outras mas que ainda compartilham recursos entre si. Conforme ilustrado na Figura 2.4, o recurso compartilhado pode ser acessado diretamente pela *Aplicação A* e através de políticas e regras de acesso pela *Aplicação B*, uma vez que este está sendo executado dentro do ambiente de *sandbox*. Também pode-se observar que *Aplicação A* pode iniciar outro processo dentro de um *sandbox* que estará sujeito a políticas de acesso ao recurso[31].

2.6.2 Regras a Nível de Sistema

Como mencionado anteriormente, é possível definir políticas e regras para restringir aplicações a nível de segurança do kernel do sistema operacional. Essa estratégia envolve a execução de aplicações atrelada a uma política. Para isso, os programas devem ser identificados e as regras devem ser baseadas nas suas características de execução. Para identificação do programa podem ser usadas diversas técnicas, por exemplo, caminho do arquivo, *fingerprint* ou outra técnica usada na execução seletiva de *whitelist* ou *blacklist*[31].

Na Figura 2.5 ilustra-se o funcionamento de aplicações que são confinadas com base em

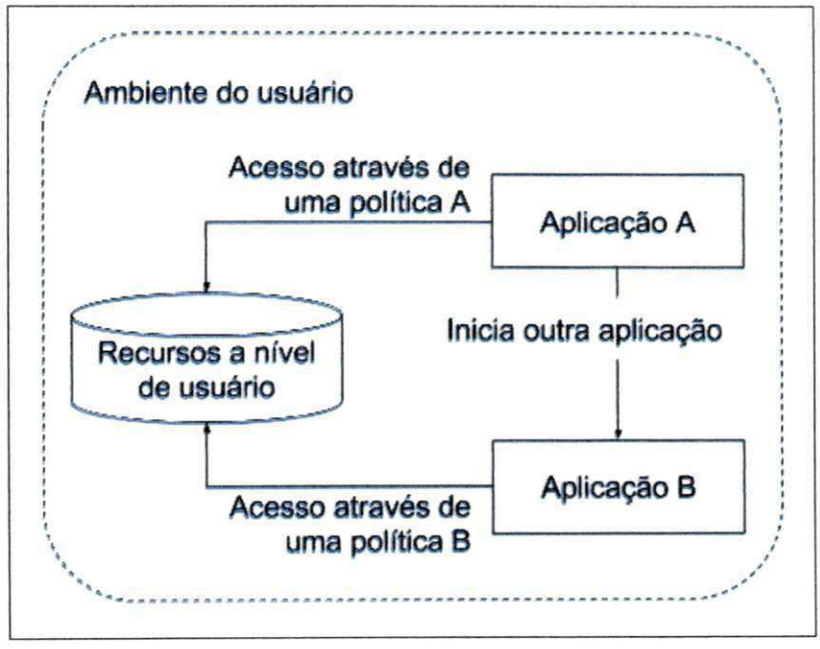

Figura 2.5: Controle com regras a nível de sistema

políticas a nível de sistema. Da mesma forma da Figura 2.4, uma aplicação pode iniciar outra, com a diferença de que todas as aplicações estão sujeitas a políticas de acesso. A política de uma aplicação não é herdada da outra aplicação que a criou, apesar de poderem ser iguais. Outra diferença para o modelo exibido na Figura 2.4 é que a *Aplicação B* não precisa executar num *sandbox*, uma vez que as regras definidas a nível de sistema já impõem restrições de segurança.

Capítulo 3

Trabalhos Relacionados

Este capítulo apresenta o estudo de trabalhos relacionados à execução de softwares não confiáveis. Os trabalhos estudados podem ser classificados em (a) técnicas de deteção de *malwares* e (b) soluções implementadas em sistemas reais.

3.1 Sandboxes e Detecção de Comportamento

Técnicas de sandbox são utilizadas de diversas formas, desde simulações de ambientes até isolamento e perfilamento de códigos. Como exemplo, tem-se o trabalho elaborado por Pirscoveanu et al.[13], que visou a criação de um sistema de *sandbox*, chamado TxBox, que funciona como um monitor de segurança executando no kernel do sistema operacional, no qual as transações são interceptadas antes de serem executadas pelo kernel e são analisadas a fim de determinar seus efeitos no sistema. Se, de acordo com alguma política definida, as transações não tiverem impacto negativo, elas são submetidas para o sistema. Da mesma forma que o modelo proposto nessa dissertação, os autores propõem uma técnica que usa *blacklists* ou *whitelists* para criar regras que definam quais instruções podem ser executadas. Além disso, a técnica proposta faz uso de outros operadores e permite que as regras sejam definidas com expressões regulares.

Outra ferramenta que executa no kernel e intercepta chamadas de sistema é proposta por Pirscoveanu et al.[3] e define uma abordagem em duas fases (análise estática de código e análise dinâmica de system calls) que vão de encontro com a proposta deste trabalho. Porém essa ferramenta, chamada pelos autores de AASandbox, não propõe um método para

identificar, ou até inibir, ações maliciosas, ao invés disso a idéia é salvar todas as chamadas a funções nativas em arquivos para uma análise posterior.

De forma semelhante à ferramenta AASandbox, o método desenvolvido por Pirscoveanu et al. [25] permite que um código malicioso execute num *sandbox*, o qual irá registrar todas as suas ações. Esse método propõe o uso de técnicas de aprendizagem e predição para extrair características, classificar e, posteriormente, realizar análise dinâmica para identificar códigos mal intencionados. O principal foco desse método é classificar os *malwares* e não reduzir os riscos causados por suas ações, mas há um aspecto de similaridade com o modelo proposto nessa dissertação: a preocupação que códigos maliciosos acessem a Internet. Para tanto, os autores optaram por usar um middleware que simule uma conexão com a Internet.

Outro método que utiliza *sandbox* para classificar *malwares* é proposto em [14]. Segundo os autores, a proposta é coletar métricas de comportamento e sumarizar os dados gerados, o que permite aumentar a acurácia na classificação dos *malwares*. Outros métodos que utilizam *sandbox* para detectar programas com comportamento suspeito incluem [15], que além de interceptar *system calls* utiliza a pilha das chamadas para detectar comportamentos abusivos e [19] que propõe um framework independente de plataforma capaz de coletar métricas e dados gerados pelos programas que podem ser utilizados para análises futuras.

3.2 Soluções Existentes

Esta seção aborda alguns trabalhos relevantes que exploram temas relacionados ao confinamento de código e que foram utilizados na implementação de sistemas reais e amplamente difundidos.

3.2.1 BOINC

Cano and Vargas-Lombardo[4] apresentam um estudo sobre as ameaças aos sistemas de computação voluntária usando o BOINC, que é um projeto de código aberto que fornece uma solução de *grid* (desde o servidor que gerencia os projetos até clientes que executam os códigos) para projetos que necessitam de um grande poder computacional. Uma vez que o projeto pode executar códigos não confiáveis, uma série de problemas de segurança ficam

evidentes [1]: falsificação de resultados, distribuição de executáveis maliciosos, ataques DoS aos servidores do projeto, roubo de informações tanto dos servidores quanto dos usuários, dentre outros.

Alguns desses problemas são similares aos observados neste trabalho, principalmente os que dizem respeito à disseminação códigos maliciosos, roubo de informações do usuário e ataques DoS a servidores. A principal diferença relativa às ameaças está no fato que o BOINC, até o momento, não considera o sequestro de recursos (DoS local) como uma ameaça, possivelmente porque que a própria natureza do projeto exige o alto uso desses recursos, inclusive para projetos que não são maliciosos.

Para contornar esses problemas o cliente do BOINC faz uso de técnicas de checagem de assinatura e dois níveis de isolamento do ambiente. O primeiro nível é um *sandbox* baseado em contas do usuário, no qual o cliente roda numa conta de usuário sem privilégios e, desta forma, só tem acesso a seus próprios arquivos não conseguindo acessar outros recursos que necessitem de privilégio. Essa técnica vai ao encontro da proposta desse trabalho para proteger contra o roubo de informações, porém, depende de como o sistema operacional lida com contas de usuário e difere do modelo apresentado no sentido de que não protege contra abuso no uso de recursos, como sistema de arquivos e rede.

Outra forma de defesa é apresentada por McGilvary et al.[22] e faz uso de máquinas virtuais para criar um ambiente completamente isolado e que não depende de implemetação do sistema operacional, porém o princípio de segurança é o mesmo da técnica anterior: evitar que aplicações interfiram no espaço privado de outras. E assim como a técnica de *sandbox* usando contas de usuário, a segurança por virtualização não impede outras ações maliciosas abordadas neste trabalho, tais como ataques DoS (local e remoto) e disseminação do programa malicioso.

3.2.2 *Native Client*

Native client é um ambiente *sandbox* criado pelo Google[2] para executar códigos nativos no navegador web de forma eficiente e que não comprometa a segurança de outros códigos, independentemente se estão rodando no próprio navegador ou fora. Um estudo sobre o sistema

[1] http://boinc.berkeley.edu/trac/wiki/SecurityIssues

[2] https://developer.chrome.com/native-client

é apresentado por Yee et al.[34] e descreve detalhes de implementação e design, tais como o mecanismo de segurança baseado em dois *sandboxes*, chamados de interior e exterior. Similarmente à abordagem proposta neste trabalho, o *sandbox* interior realiza uma análise estática de código para detectar problemas que afetem a segurança, porém são utilizadas técnicas e ferramentas adicionais para garantir que o código nativo possa ser verificado.

A análise estática, realizada pelo *sandbox* interior, tem como principal função garantir que qualquer instrução utilizada pela aplicação pertence a um subconjunto predefinido de instruções permitidas. Já o *sandbox* exterior atua como um mediador de *system calls*, adicionando uma camada extra de segurança, funcionando de forma similar ao executor definido neste trabalho.

De modo geral o sandbox proposto no *native client* tem o mesmo intuito deste trabalho, que é permitir a execução de códigos não confiáveis de modo que ações maliciosas não causem efeito no sistema. O modelo de sandbox também pode ser comparado com os módulos pré executor e executor: (a) o sandbox interior realiza análise de código de forma análoga ao pré executor e (b) o sandbox externo monitora as chamadas de baixo nível, algo semelhante à funcionalidade combinada do pré executor e do executor. A grande diferença entre as propostas é que o *native client* é preparado para lidar com verificação e checagem de códigos nativos, fornecendo um conjunto de técnicas e ferramentas para serem utilizadas pelos desenvolvedores.

Um ponto dissonante entre as abordagens é que o modelo proposto neste trabalho além de realizar a análise estática, realiza o monitoramento de recursos e restringe a comunicação com a rede a fim de evitar que códigos maliciosos se propaguem para outros dispositivos.

3.2.3 Android e iOS

Devido à popularidade de ambas as plataformas, o número de ataques e distribuição de programas maliciosos continua crescendo, fazendo com que tanto a Apple quanto o Google invistam em formas de coibir os ataques. Mohamed and Patel[23] apresentam em seu trabalho uma comparação entre os mecanismos gerais de segurança empregados em ambas as plataformas. A primeira linha de defesa de ambas as plataformas está no controle de distribuição dos aplicativos através das lojas oficiais. No caso da Apple cada aplicativo é rigorosamente testado antes de ser aprovado, já no caso do Google não há um controle tão grande, o que

possibilita que aplicativos maliciosos sejam distribuídos mais facilmente.

Independentemente do controle sobre a publicação, aplicativos maliciosos acabam se espalhando e isso levou à elaboração de uma série de mecanismos de defesas que funcionam de forma semelhante no Android e no iOS, a exemplo das permissões. Aplicativos iOS já são instalados com um conjunto mínimo de permissões para que possam ser executados e, adicionalmente, podem pedir outras permissões à medida que necessitem acessar recursos específicos como GPS (sistema de posicionamento global), notificações, etc. Aplicativos Android funcionam de forma semelhante, porém, podem requisitar todas as permissões durante a instalação. Esse método é muito similar ao proposto neste trabalho no qual, através de um arquivo descritor, um programa lista as permissões que deseja ter acesso. Uma diferença crucial entre as abordagens está no fato que os aplicativos Android e iOS podem ser instalados com um conjunto mínimo e à medida que executam vão agregando novas permissões.

Outros mecanismos de segurança estudados por Ahmad et al.[1] incluem isolamento de aplicativos, randomização de memória, armazenamento e criptografia de dados. Ambas as plataformas utilizam um modelo de *sandbox* baseado em permissões, o que reduz o risco de ataques uma vez que cada aplicativo é isolado dos outros e do próprio sistema operacional. Essa ideia também é similar à proposta deste trabalho, o qual assegura o isolamento de códigos através do monitoramento no uso de recursos e execução de código em processos distintos. A maior diferença entre os modelos de segurança dessas duas plataformas e o modelo proposto neste trabalho é o fato que este lida com duas questões adicionais: monitoramento de recursos a fim de evitar DoS local e restrições de comunicação em rede a fim de evitar a propagação de programas maliciosos.

Capítulo 4

Malwares

Aplicações podem ser atacadas tanto porque contêm registros e dados importantes ou porque podem ser ponto de entrada para códigos maliciosos infectarem outras aplicações, sistema ou até se espalhar para outros dispositivos. Este capítulo apresenta os tipos de *malwares* mais comuns bem como suas principais características e, posteriormente, são apresentados alguns métodos que podem ser utilizados para evitar que esses códigos sejam bem sucedidos em tarefas que possam causar danos.

4.1 Natureza e Funcionamento

Os *malwares* já evoluíram bastante desde que os primeiros ataques foram feitos, uma vez que do ponto de vista de um atacante, um código que seja bem sucedido pode gerar uma forma de recompensa, já do ponto de vista dos produtores de softwares e usuários um ataque bem sucedido fatalmente acarretará em algum tipo de prejuízo. Dessa forma, os grandes produtores de software e hardware sempre buscam formas de detectar e evitar ataques, em contra partida, os atacantes buscam meios de burlar a segurança dos sistemas.

4.1.1 Formas de Evasão

Essa batalha constante entre os dois lados leva a uma evolução natural nos métodos de criação, replicação e execução dos *malwares*. Os mecanismos empregados para burlar os sistemas de detecção vão desde obfuscação do código até métodos de força bruta como DoS[21].

Dentre esses métodos podemos listar os seguintes:

1. Ofuscação: método no qual o código fonte é alterado de forma que fique difícil de entender sua semântica e, com isso, consegue evadir alguns métodos de detecção baseados em identificação textual;

2. Fragmentação de pacotes: trechos do código são transferidos em diferentes pacotes TCP/IP (conjunto de protocolos de comunicação em rede), dessa forma o sistema de proteção não consegue identificar um padrão maligno pois apenas partes distintas do código são entregues;

3. DoS: essa técnica explora falha de projeto do sistema de detecção de intrusão, o qual precisa monitorar todas as requisições de rede e com isso, fazem um grande uso de CPU, memória e disco. Ataques com essa característica buscam sobrecarregar os recursos computacionais para que o processo de detecção não consiga executar de forma satisfatória;

4. Reúso de código: essa técnica explora falhas de componentes do kernel e são muito difíceis de serem detectadas. O objetivo principal é executar códigos arbitrários que conseguem gerar inconsistências de memória levando a falhas de componentes do sistema e pode até permitir que um programa consiga acesso ao fluxo de controle de outra aplicações.

4.1.2 Formas de Propagação

Para os códigos maliciosos se manterem ativos eles precisam ser flexíveis o suficiente para se propagar por seus próprios meios, sem depender da ação de um agente externo. As formas de proliferação evoluíram substancialmente, pois antes a disseminação era feita em sua maioria a partir da inserção de mídias físicas mas atualmente diversos meios de comunicação são utilizados para disseminar tais códigos, a destacar [24]:

1. Serviços de mensagem (SMS, MMS e Push): são usados principalmente em dispositivos celulares, nos quais os atacantes injetam conteúdos maliciosos nas mensagens que são lidas diretamente por componentes do sistema operacional e podem até permitir que o atacante tenha controle remoto sobre o sistema.

2. Redes sem fio: tecnologias sem fio, tais como bluetooth, podem ser um ponto frágil do sistema a medida que outro dispositivo mal intencionado pode conseguir parear com um dispositivo alvo injetando códigos maliciosos ou até enviando comandos remotos. Esse tipo de ataque tem limitações inerentes à tecnologia, tal como alcance e segurança no pareamento, mas também podem usar diversos artifícios e falhas de protocolo para atacar o dispostivo. Exemplos de toolkit que exploram falhas no bluetooh incluem: (1) *Blue Bump* que explora falhas de como a tecnologia lida com chaves e permite que o atacante use as redes móveis do dispositivo; (2) *Bluejacking* que envia mensagem com um código de acesso falso e permite que o atacante tenha controle do dispositivo.

3. Internet: a maior parte dos dispositivos hoje em dia estão conectados em redes sem fio com acesso a internet (wi-fi, 3G, 4G, etc.) e isso permite que todos os tipos de aplicativos e até o sistema operacional envie e receba dados, criando assim, um ambiente no qual os *malwares* consigam meios mais fáceis de se proliferar. A disseminação pode ocorrer de diversas formas: compartilhamento de arquivos, emails infectados, compartilhamento de links através de redes sociais, navegação em sites maliciosos, etc.

4.1.3 Classificação

As formas de atuação com relação às ameças dependem de um estudo sobre os códigos maliciosos a fim de entender como eles agem. Uma forma de simplificar esse processo é agrupá-los e categorizá-los de acordo o tipo de dano que causam, forma de disseminação e componentes atacados. Apesar das diversas categorias e nomenclaturas utilizadas, podemos classificá-los da seguinte forma [7]:

1. *Virus* é um trecho de código que necessita de um programa hospedeiro para ser executado e pode se propagar de diversas formas;

2. Worms são parecidos com virus porém pode ser executado independentemente de um software hospedeiro e não se propaga localmente, ao invés disso cria uma cópia completa de si mesmo e geralmente se propaga para diversos sistemas através da rede;

3. *Trojans* são códigos maliciosos que se disfarçam de programas benignos de forma que

o usuário seja convencido a executá-lo e também podem dar a capacidade do atacante controlar o sistema;

4. *Spywares* geralmente se instalam no sistema quando o usuário executa outro tipo software e passam a monitorar o sistema de forma análoga a um espião, dessa forma os dados contidos no dispositivo, incluindo dados pessoais, ficam vulneráveis;

5. *Adwares* são programas desenvolvidos para exibir anúncios e podem extrair informações sensíveis (como dados de pagamentos do usuário) ou facilitar a instalação de *trojans* e seu principal método de ação é interceptar e observar requisições de rede;

6. *RiskTools* são programas com diversas funções tais como ocultar arquivos do sistema, finalizar processos e esconder aplicações em execução e são geralmente usados para facilitar a ação de outro *malware*.

4.2 Segmentação de Malwares

De acordo com o 3º relatório trimestral de 2015 publicado pelo grupo KasperskyLab [1] e mostrado na Figura 4.1, os *malwares* do tipo *adware* concentram o maior número de ataques registrados a dispositivos móveis, seguidos de perto pelos *risktool* e dos *trojans* e suas várias derivações. Como pode-se observar, também houve um aumento substancial dos ataques *adware* em comparação ao 2º trimestre do mesmo ano, concentrando 52.2% dos ataques ante a 19% no período anterior.

Ainda, de acordo com o relatório, houve um aumento de aproxidamente 50% no número de instalação de aplicativos maliciosos e surgiram mais de 300 mil novos aplicativos desse tipo em comparação com o período anterior. Também pode-se observar através Figura 4.2 que aplicativos feitos para a plataforma Android representam a segunda maior fonte de ataques, atrás apenas de aplicativos feitos para navegadores web.

Segundo a Tabela 4.1 que lista o número total de vulnerabilidades por produto, podemos observar que apesar da plataforma Android conter um alto índice de ataques (de acordo com o relatório publicado pelo KasperskyLab), ela ainda tem um número relativamente baixo de

[1] https://securelist.com/analysis/quarterly-malware-reports/72493/it-threat-evolution-in-q3-2015/

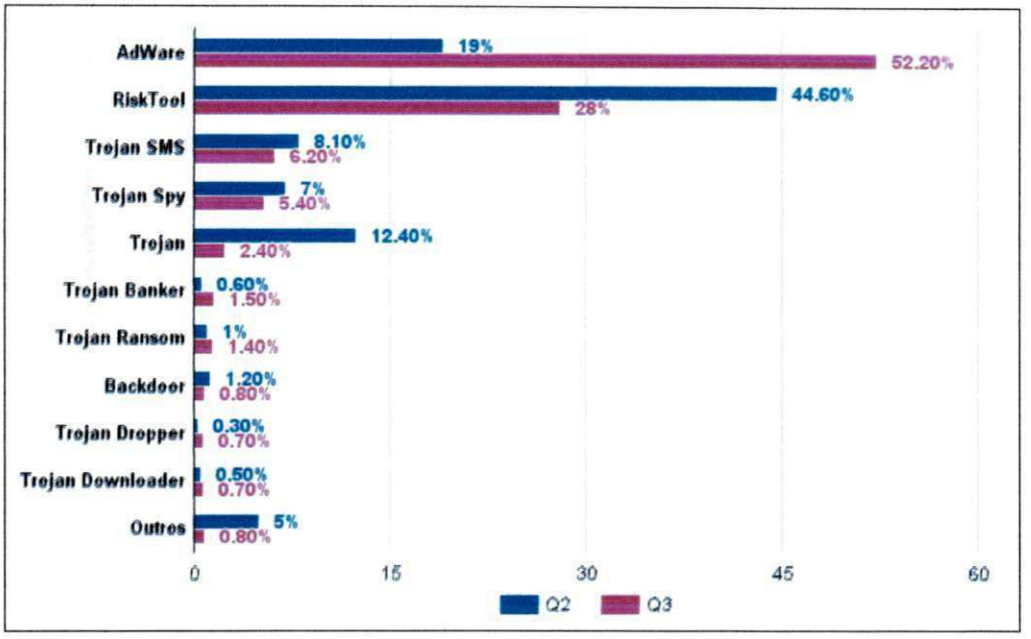

Figura 4.1: Distribuição de malwares em sistemas móveis no ano 2015

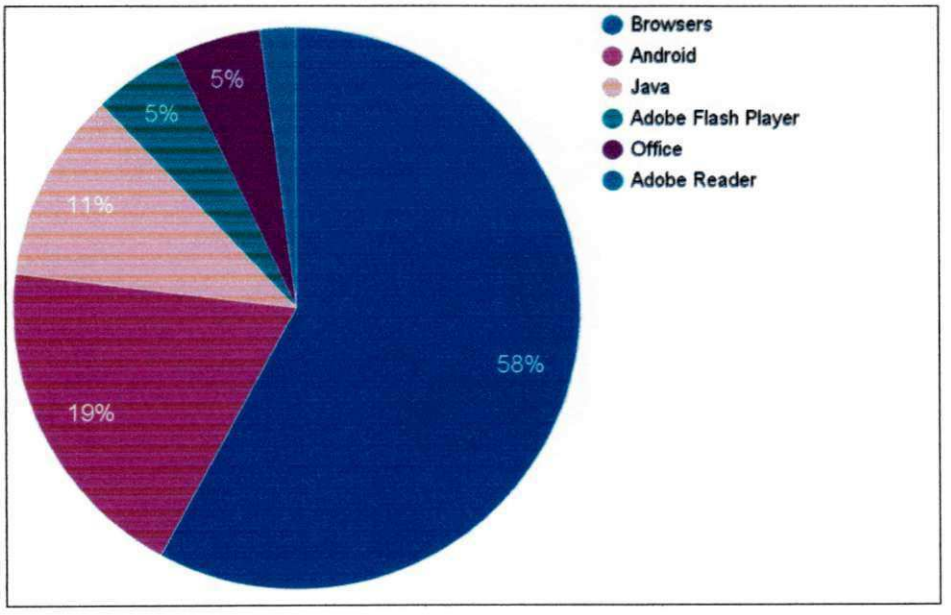

Figura 4.2: Distribuição de tentativas de ataque por produto no ano 2015

vulnerabilidades, já os três princiais browsers da lista se somados contém aproximadamente 3500 pontos considerados vulneráveis.

Posição	Produto	Núm. de vulnerabilidades
1	Mac Os X	1600
2	Linux Kernel	1461
3	Firefox	1391
4	Chrome	1314
5	Iphone Os	919
6	Flash Player	892
7	Internet Explorer	776
8	Windows Xp	726
9	Windows Server 2008	705
29	Android	431

Tabela 4.1: Número total de vulnerabilidades por produto[2].

4.3 Mecanismos de Proteção

Diversas técnicas são utilizadas para reduzir os danos de executar códigos maliciosos e são utilizadas tanto para analisar o comportamento de programas quanto para permitir que o usuário os execute de forma segura. Essas técnicas focam em dois aspectos importantes: impedir que um *malware* se instale no dispositivo e assegurar que qualquer código execute sem prejudicar outros softwares ou componentes do sistema. Para tanto são utilizadas técnicas como *sandbox*, acesso aleatório de memória e integridade de fluxo de execução.

A plataforma Android usa algumas dessas técnicas e também emprega variantes de controle de acesso, encriptação de dados, dentre outras para garantir a integridade de dados e evitar que aplicações interfiram no espaço privado de outras. As principais técnicas utilizadas pelo framework Android são [5]:

[2]http://www.cvedetails.com/top-50-products.php?year=0

1. Sandbox: aplicações são executadas em modo sandbox e são proibidas de acessar dados em disco e memória ou códigos de outras aplicações.

2. Controle de acesso baseado em permissões: permissões para acessar recursos do sistema ou dados do usuário são garantidas de forma individual a cada aplicação durante a instalação ou durante a execução, desde que o usuário as conceda explicitamente. Cada permissão é declarada em um arquivo de manifesto e dizem respeito a recursos específicos do dispositivo, tais como: acesso a câmera, acesso a dados públicos, acesso a configurações do sistema, etc.

3. Intercomunicação de processos (IPC): aplicações podem interagir com outras, porém não podem fazer isso de forma direta e tem que usar IPC através de interfaces definidas pela própria aplicação que será acessada.

4. Gerenciamento de memória e processos: cada aplicação roda em um processo separado com uma própria instância de uma VM, o que dificulta ataques que exploram falhas relacionadas a gerenciamento de memória e controle de fluxo de processos.

Esses mecanismos de segurança utilizados na plataforma Android se mostram bastante eficazes, pois apesar do grande número de ataques sofridos (ilustrados na figura 4.2) o sistema ainda pode ser considerado seguro em relação ao número de vulnerabilidades quando comparado a outros sistemas operacionais (listados na tabela 4.1).

4.4 Requisitos Para um Modelo de Proteção

Apesar das medidas adotadas em outros sistemas serem relativamente eficazes, elas não cobrem outros casos mais específicos sem uso de ferramentas externas, tais como, evitar que um código malicioso seja baixado ou replicado via rede, restrição de comunicação em rede, restrição de funções ou padrões de códigos. Dessa forma, pode-se elaborar os seguintes requisitos para desenvolvimento do novo modelo:

1. O ambiente de execução deve ser capaz de executar códigos independentes de forma que fiquem isolados uns dos outros.

2. O ambiente de execução deve ter controle de acesso a recursos e os programas devem descrever explicitamente quais recursos irão utilizar através de um arquivo descritor.

3. O ambiente de execução deve prover mecanismos para restringir o acesso de programas a internet, de forma a evitar ataques DoS, formação de botnets ou até replicação de código.

4. O ambiente de execução deve prover mecanismos de monitoramento do tráfego, incluindo formas de fragmentação de pacotes, a fim de evitar que conteúdos maliciosos sejam enviados ou baixados.

5. O ambiente de execução deve ser capaz de realizar análise prévia do código a fim de determinar alguns padrões maliciosos ou uso de funções não permitidas.

6. O ambiente de execução deve ser capaz de restringir o acesso a arquivos e diretórios públicos a fim de evitar roubo de informações ou danificação de arquivos sensíveis.

7. O ambiente de execução deve ser capaz de restringir e monitorar o acesso a recursos físicos a fim de evitar DoS.

Capítulo 5

Modelo Proposto

O modelo proposto consiste em dois módulos principais: pré-processador e executor de código. O pré-processador é a primeira parte que será executada a fim de realizar análise prévia do código e extrair metadados usados posteriormente na fase de execução pelo módulo executor.

Na Figura 5.1 representa-se o fluxo de execução do ponto de vista dos dois módulos principais no qual o ambiente de execução (Java, Node.js, Lua, etc.) carrega o programa (código e arquivos descritores) e o repassa para o pré processador. Caso alguma violação seja detectada pelo pré processador a execução é terminada imediatamente, caso contrário o código segue para o executor que analogamente termina a execução de imediato se outra violação for encontrada. Caso nenhuma violação seja detectada por ambos os módulos o programa segue o fluxo natural de execução do respectivo ambiente.

As seções seguintes deste capítulo descrevem em maiores detalhes os módulos do pré processador e executor, bem como seus submódulos e demais componentes.

5.1 Pré-processador

O pré-processador é a primeira etapa do mecanismo de segurança, sendo responsável por interpretar meta dados relativos bem como realizar uma análise prévia do código que será executado. O diagrama da Figura 5.2 representa o fluxo de execução dos dois submódulos principais do pré-processador (analisadores do arquivo descritor e código). O ponto de entrada é o analisador de descritor, o qual irá interpretar um arquivo que contém alguns me-

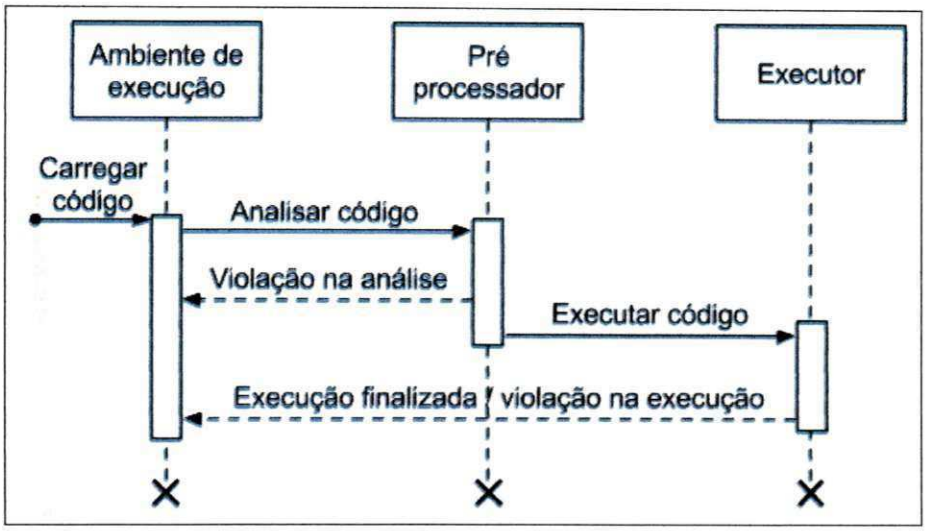

Figura 5.1: Diagrama de sequência dos módulos principais

tadados e regras e, após essa etapa, o fluxo segue para o analisador de código, o qual irá realizar uma checagem estática no código a fim encontrar alguns tipos específicos de violações. Após ambas as etapas o código passa para o módulo executor, onde irá ser executado de fato.

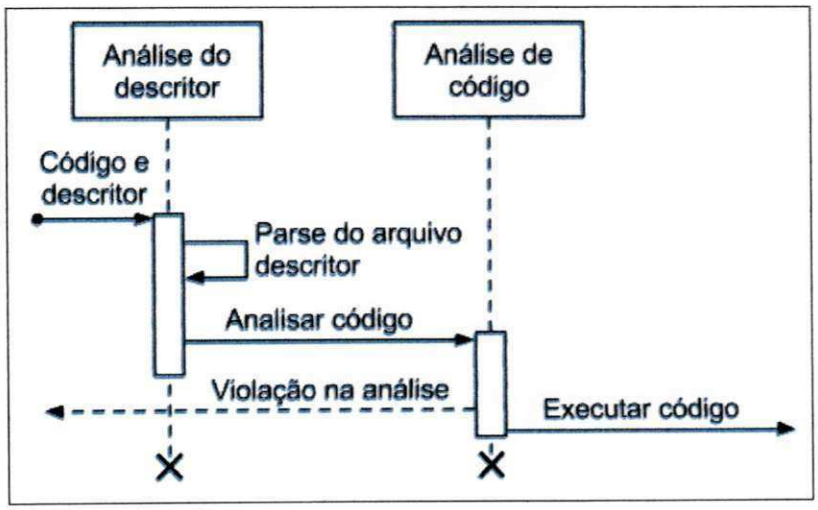

Figura 5.2: Diagrama de sequência do pré processador

5.1.1 Arquivo Descritor

Arquivos descritores são responsáveis por definir o comportamento da aplicação de uma maneira formal e baseada em regras, o qual deverá ser entendido pelo módulo de execução a fim de detectar possíveis violações no uso de recursos não declarados. Tal módulo faz-se necessário para evitar que aplicações consigam acesso indiscriminado a todos os tipos de recursos disponíveis no ambiente de execução.

Modelos de segurança podem implementar esse módulo de diversas formas, por exemplo: uma plataforma de execução com interface gráfica pode exibir para o usuário a lista de recursos definidos no descritor para que o usuário em questão tenha ciência do que o código terá acesso e, portanto, escolher se permite ou não a execução. Outra aplicação baseada em computação em grid pode usar essa técnica para permitir ou bloquear acesso a determinados recursos baseados na natureza do *job* que será executado, tal como, pode criar uma lista de exceções para que os *jobs* só consigam se conectar a determinados endereços da internet, a fim de evitar que códigos não confiáveis realizem ataques a servidores.

As regras, sintaxe e semântica nas quais esse arquivo poderá ser elaborado dependem da plataforma e tecnologia utilizadas mas devem ser definidas de forma não ambígua tendo em vista que esse pode ser um ponto de ataque no qual códigos não confiáveis fatalmente irão explorar.

Permissões

Modelos de segurança baseados em permissões já são utilizados em sistemas desde a década de 70 [11] [31], porém essas técnicas garantiam o acesso a um conjunto pré determinado de recursos para cada usuário do sistema através de papéis e grupos. Esse mecanismo tem a evidente desvantagem de garantir privilégios ao usuário e não a aplicações. Outro ponto negativo é que a gerência das permissões são mantidas centralizadas e organizadas através de grupos ou papéis. Caso um usuário necessite de outra permissão, este tem que ser adicionado em outro grupo ou migrar para um que contenha a nova permissão (ou um conjunto de permissões). Outra alternativa é modificar o grupo ao qual ele faz parte para conter a nova permissão. Nessa última opção, junto com o usuário em questão todos os demais usuários também herdarão a nova permissão.

Apesar desse modelo de permissões baseado em papéis ser útil para diversos tipos de sistemas, ele não é útil para uma plataforma que não possua diferentes usuários. Porém é possível fazer uma adaptação dessa idéia ao mapear os conceitos, entidades e relacionamentos envolvidos.

Definição 1 *Definição formal do modelo de acesso baseado em regras*

1 *seja USUÁRIOS o conjunto de usuários do sistema*

2 *seja PAPÉIS o conjunto de papéis*

3 *seja OBJETOS o conjunto objetos sujeitos a restrição de acesso*

4 *seja OPERAÇÕES o conjunto de operações permitidas*

5 $UA \subseteq$ *USUÁRIOS* $\times$ *PAPÉIS*

6 *usuários_atribuídos(r)* $= \{u \in$ *USUÁRIOS* $\mid (u, r) \in UA\}$

7 *PERMISSÕES* $= 2^{(OBJETOS \times OPERAÇÕES)}$

8 $PA \subseteq$ *PERMISSÕES* $\times$ *PAPÉIS*

9 *permissões_atribuídas(r)* $= \{p \in$ *PERMISSÕES* $\mid (p, r) \in PA\}$

A definição acima explicita o comportamento de um sistema baseado em papéis, no qual as entidades são usuários, papéis, objetos e operações. A partir da definição acima pode-se inferir os seguintes relacionamentos: usuários podem ter vários papéis; um papel pode possuir várias permissões; permissões são tuplas que definem o tipo de acesso (*OPERAÇÕES*) a um determinado recurso (*OBJETOS*).

A diferença essencial do modelo acima descrito para o modelo de permissões proposto neste trabalho está no fato que permissões são atribuidas a aplicativos (ou códigos) e não a usuários, dessa forma, podemos fazer as seguintes alterações (também ilustradas na figura 5.3):

1. a entidade *USUÁRIOS* será substituída por *PROGRAMA*

2. a entidade *PAPÉIS* e seus relacionamentos serão removidos, pois os programas não são categorizados por papeis uma vez que cada código ou programa tem um conjunto diferente de permissões.

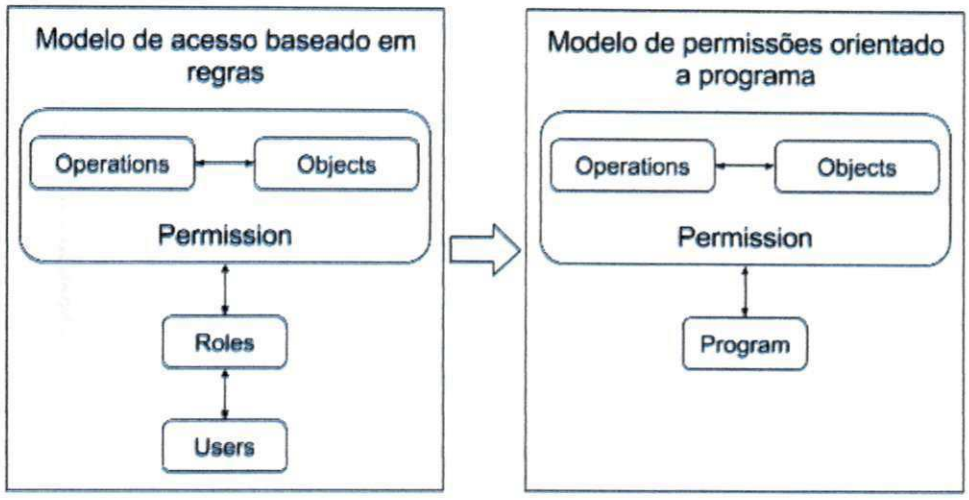

Figura 5.3: Mapeamento entre modelos de acesso e permissão

3. o relacionamento *permissões_atribuídas* será modificado para representar as permissões de um programa ao invés de representar as permissões de um papel.

Definição 2 *Definição formal do modelo de permissões*

1 *seja PROGRAMA um programa que executa no espaço do usuário*

2 *seja OBJETOS o conjunto de objetos sujeitos a restrição de acesso*

3 *seja OPERAÇÕES o conjunto de operações permitidas*

4 $PERMISSÕES = 2^{(OBJETOS \times OPERAÇÕES)}$

5 $PA \subseteq PERMISSÕES \times PROGRAMA$

6 $permissões\_atribuídas(pr)) = \{p \in PERMISSÕES \mid (p, pr) \in PA\}$

A partir do modelo formalmente definido em 2 é possível construir um método de verificação de permissões utilizando as tecnologias que forem convenientes, bem como a definição e granularidade das permissões que ficarão disponíveis a códigos de terceiros. A definição da granularidade é um aspecto que deve ser analisado com bastante cuidado quando for implementado, pois se determinado recurso possuir uma granularidade maior pode tornar o modelo muito permissivo, já se for definido com granularidade mais fina pode tornar-se muito restritivo[10].

Como exemplo pode-se citar o acesso a arquivos: se um sistema define uma política de acesso a diretórios com base numa granularidade maior, pode acabar permitindo que um

programa acesse arquivos de locais sensíveis (como arquivos do sistema), em contrapartida uma granularidade menor pode exigir que o programa tenha que declarar o acesso a cada arquivo individualmente no arquivo descritor de permissões.

O arquivo descritor deve ter uma seção com a especificação de todas as permissões que o programa deseja ter acesso. O trecho de Código 5.1 define, através de uma sintaxe JSON (notação de objetos na linguagem *javascript*), as permissões com base uma lista de objetos e suas possíveis operações definidos na Tabela 5.1. Essa política definida no conjunto de permissões diz que o programa so poderá acessar a câmera, recuperar a versão do sistema operacional (SO) e localização do sistema além de só poder fazer requisições HTTP do tipo GET, POST e PUT. Quaisquer outras ações que não estejam de acordo com essas permissões devem ser bloqueadas.

Objetos	Operações
Arquivos de sistema	leitura e escrita
Processo	spawn, fork e kill
Informações do sistema	localização, versão do SO e identidade do usuário
Uso de hardware	Câmera, microfone e bluetooth
Requisições HTTP	GET, POST, PUT e DELETE

Tabela 5.1: Exemplo de objetos e operações

```
{
  "permissoes": {
    "hardware": [
      "camera"
    ],
    "requisicoes_http": [
      "GET",
      "POST",
      "PUT"
    ],
    "informacoes_sistema": [
      "versao_so",
      "localizacao"
    ]
```

```
    }
}
```

Código Fonte 5.1: Descritor de permissões

White/Black List de URL

Atualmente a Internet é a maior distribuidora de malwares e, portanto, a principal causa de infecção de computadores com códigos maliciosos. Devido a esse grande perigo, administradores de serviços online, redes locais e até os próprios usuários finais buscam maneiras de evitar a infecção dos dispositivos a partir de sites não confiáveis e uma das maneiras mais simples, porém eficientes, é o uso de blacklists. Essas blacklists consistem simplesmente de uma lista de identificadores de objetos maliciosos e podem conter endereços de IP, domínios ou URLs[2].

Essa técnica é bastante eficaz principalmente a ataques realizados contra navegadores web, uma vez que ao abrir uma página web o navegador passa a executar seu código e este pode explorar vulnerabilidades para baixar outros arquivos que infectarão o sistema, roubar informações ou realizar outros tipos de ataques[32]. Contudo, uma aplicação que não execute um código que esteja embutido numa página também pode sofrer ataques, Por exemplo, um aplicativo pode simplesmente se conectar a um servidor para consumir dados de uma API REST, porém se esse servidor estiver infectado ou a comunição for interceptada, um atacante pode enviar quaisquer tipos de dados e com isso, realizar ataques.

Nesse cenário, as aplicações também podem lançar mão de técnicas de blacklist ou whitelist como uma camada adicional de segurança para evitar conexão com partes duvidosas. Outra vantagem do uso dessa técnica é a possibilidade de restringir conexões, mesmo com fontes seguras, a fim de criar uma política de tráfego da dados, por exemplo, o sistema pode querer diminuir o tráfego de dados ao detectar que o dispositivo está em uma rede com conectivade limitada, para tanto pode desabilitar a conexão a determinados serviços como anúncios. Assim, pode-se definir o modelo de black/white list da seguinte forma:

Definição 3 *Definição Formal do Modelo de Black/White List*

1 *seja PROGRAMA um programa que executa no espaço do usuário*

2 *seja URL um endereço de rede*

3 *seja REQUISIÇÃO uma requisição de rede*

4 $PR = 2^{(PROGRAMA \times REQUISIÇÃO)}$

5 $BLACKLISTED \subseteq PR \times URL$

6 *requisições_permitidas(u)* $= \{pr \in PR \mid (pr, u) \notin BLACKLISTED\}$

Na Definição 3, foi criada a regra que permite apenas que conexões sejam estabelecidas com uma URL que não esteja na blacklist, porém pode ser conveniente criar uma regra oposta que permita conexão apenas com URLs que estejam numa whitelist. Esta abordagem não lida com o fato de identificar de forma dinâmica quais URLs são confiáveis ou não, tendo em vista que endereços na web são dinâmicos e adotam padrões e assinaturas muito distintas entre si. Tais problemas são discutidos em [2] e [32].

O código 5.2 define, usando o conceito de blacklist, um conjunto de urls que não podem ser acessadas. Qualquer tentativa de acessar uma url não permitida deverá ser identificada e a chamada deve ser bloqueada.

```
{
  "url_black_list": [
    "http://webserver1.com",
    "https://main-domain.net"
  ]
}
```

Código Fonte 5.2: Descritor de black list de url

5.1.2 Analisador de Código

A segunda parte do modelo de segurança trata de um analisador de código, responsável por processar o código que será executado a fim de encontrar algumas violações que não estão conforme as restrições impostas no arquivo descritor ou não correspondem a padrões considerados seguros. Nessa etapa a análise realizada é estática de forma a garantir uma

verificação rápida de alguns pontos chave para a segurança sem ter impacto no desempenho durante a execução do código.

A análise de código proposta nesse trabalho pode ser dividida em duas partes: verificador de dependência e pré-checagem de código. A primeira trata de buscar dependências não permitidas e a segunda busca por padrões potencialmente perigosos. Ambas as etapas fazem uso de análise estática de código, a qual pode ser definida como um processo de analisar a estrutura de um programa através de seu código fonte ou seu código intermediário[27].

Métodos de análise estática já são usados em muitos projetos com o intuito de detectar erros de sintaxe, semântica, violações de boas práticas ou até condições que possam levar a possíveis bugs, tudo sem a necessidade de executar o código. Apesar do intuito original dessa técnica, ela também pode ser utilizada para prover uma camada adicional de segurança.

Verificador de Dependências

A maioria das linguagens de programação têm como parte de sua gramática o uso de cláusulas para importar e reusar outros módulos. A depender da linguagem essas cláusulas podem ser usadas para importar arquivos, pacotes, classes ou funções. As Tabelas 5.2 e 5.3 ilustram as variações da sintaxe em diferentes linguagens[1].

Para permitir que certas restrições com relação às dependências importadas sejam analisadas de forma correta, o arquivo descritor pode ser novamente modificado para permitir que tais regras sejam aplicadas.

Definição 4 *Definição formal do modelo de reuso de dependências*

1 *seja PROGRAMA o programa que executa no espaço do usuário*

2 *seja ARQUIVOS o conjunto de arquivos*

3 *seja IMPORTS o conjunto de cláusulas de uso de arquivos*

4 $\textit{INSTRUÇÕES} = 2^{(ARQUIVOS \times IMPORTS)}$

5 $PS \subseteq \textit{INSTRUÇÕES} \times PROGRAMA$

6 $imports\_permitidos(pr) = \{s \in \textit{INSTRUÇÕES} \mid (s, pr) \in PS\}$

[1] https://en.wikipedia.org/wiki/Comparison_of_programming_languages_(syntax)

A Definição 4 leva em consideração que dependências são importadas com base em arquivos e que cada programa tem um conjunto permitido de arquivos que podem ser importados (definidos através do arquivo descritor). Note que esse modelo formal define uma whitelist para os arquivos que poderão ser importados, mas é possível usar a mesma estratégia com blacklist e, dessa forma, definir as dependências que não poderão ser usadas.

Linguagem	Declaração
ASP	#include file="filename"
AutoIt, C, C++	#include "filename", #include <filename>
COBOL	COPY filename.
Falcon	load "filename"
Fortran	include 'filename'
Lua	require("filename")
Mathematica and Wolfram Language	Import["filename"]
MATLAB	addpath(directory)[8]
Objective-C	#import "filename", #import <filename>
Perl	require "filename";
PHP	include "filename";, require "filename";
R	source("filename")
Rust	include!("filename");

Tabela 5.2: Comparação entre sintaxe de linguagens de programação para importar arquivos

Pré Checagem de Código

Diversas técnicas são utilizadas para analisar o comportamento e classificar um código em execução a fim de determinar se este código é malicioso ou não [3] [13]. Tais técnicas são geralmente empregadas em sistemas antivirus e também são usadas para classificar o comportamento de códigos maliciosos. Porém, geralmente há um impacto negativo na performance do sistema como um todo, uma vez que tais sistemas de detecção rodam em paralelo a outros códigos para detectar possíveis problemas em tempo de execução.

Técnicas de análise estática de código também podem ser utilizadas e não causam im-

Linguagem	Declaração
Falcon	import class
Java, MATLAB	import package.class
Python	from module import class
Scala	import package.class, import package. class1 => alternativeName, 'class2 , import package._

Tabela 5.3: Comparação entre sintaxe de linguagens de programação para importar classes

pacto negativo durante a execução do programa. Nesse caso o código fonte é analisado a fim de detectar padrões que impliquem em possíveis ações maliciosas[3]. Apesar da vantagem da rapidez para analisar, essa técnica nem sempre é eficaz pois códigos maliciosos usam muitas técnicas para evadir esse tipo de checagem.

Além de procurar por padrões que indiquem códigos maliciosos, essa técnica pode ser usada para procurar por uso de instruções não permitidas (exposta na Definição 5), por exemplo: uma aplicação de *online judge* executa códigos arbitrários que podem representar um perigo de segurança, para tanto, a aplicação pode restringir o uso de determinadas funções específicas de uma plataforma, tais como funções de leitura/escrita de arquivos, métodos para executar comandos nativos, etc.

Definição 5 *Definição formal do modelo de permissão de uso de instruções*

1 *seja* $CALLS$ *chamadas a funções*

2 *seja* $PROHIBITED$ *conjunto de chamadas não permitidas*

3 $allowed\_calls = \{c \in CALLS \mid c \notin PROHIBITED\}$

5.2 Executor

O executor é a camada do modelo que irá rodar o código própriamente dito, dessa forma, validações que devem ser feitas em tempo de execução são pertinentes a essa camada. Devido à natureza das linguagens de programação e frameworks, um executor de código pode ter variações na forma como se comporta desde a fase de compilação até a fase de execução. As formas de execução de códigos mais comumente utilizadas são:

1. Interpretador: códigos interpretados geralmente são simples arquivos textuais, ou alguma variação com um código intermediário. Esses códigos não podem ser executados diretamente pois não possuem instruções nativas. Portanto, dependem de um software intermediário conhecido como interpretador, o qual lê um código em alguma sintaxe definida e o interpretará em tempo execução e converterá em instruções nativas de modo que este possam ser executadas diretamente na CPU[16].

2. Compilador: códigos são convertidos em outras instruções que possam ser executadas diretamente pela CPU ou por outro ambiente de execução. Compiladores têm a característica de otimizarem o código gerado e podem adotar diferentes estratégias como AOT (ahead of time)[2] e JIT (just in time)[3][28].

Independente da técnica utilizada para gerar e executar códigos, a plataforma ou linguagem escolhida na qual o modelo proposto nesse trabalho será implementada deve prover mecanismos para monitorar em tempo de execução alguns aspectos comportamentais do programa. Esses aspectos são definidos em 3 categorias (rede, arquivos e recursos físicos) e dizem respeito a questões importantes de segurança que podem colocar em risco um dispositivo.

5.2.1 Mediador de Comunicação em Rede

O monitoramento de comunicações em rede tem por finalidade evitar que o aplicativo faça requisições a *hosts* não permitidos e baixe arquivos (ou outros tipos de dados) que possam ser prejudiciais. Essa camada consiste na execução sequencial de 3 passos conforme ilustrado na Figura 5.4: verificador de endereços, transferência e autenticidade. Caso alguma violação seja detectada a comunicação com o *host* é interrompida de imediato, caso contrário, ao final da última etapa os dados são retornados à aplicação que iniciou a comunicação.

Verificador de Endereços

Logo após uma requisição de rede ser criada ela é interceptada e analisada para verificar se o *host* destino é confiável ou não. Nessa etapa o verificador usa as informações extraídas

[2]https://en.wikipedia.org/wiki/Ahead-of-time_compilation

[3]https://en.wikipedia.org/wiki/Just-in-time_compilation

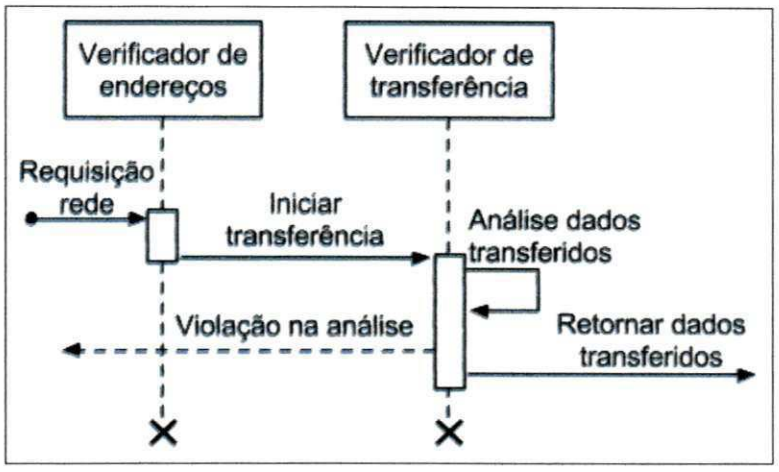

Figura 5.4: Diagrama de sequência do mediador de comunicação em rede

do arquivo descritor referentes a listagem de urls, quer seja usando blacklist ou whitelist. O Código 5.3 explicita que as regras de acesso a *hosts* podem ser bastante flexíveis e funcionar tanto com IP versão 4 e versão 6 além de incluir nomes de domínio, endereços remotos, endereços locais, máscaras e regex. Além de realizar a checagem usando endereços contidos no arquivo descritor uma implementação pode optar por usar uma base de dados ou serviço online que mantém o histórico de confiabilidade de sites e URLs, tais como Cisco SenderBase [4], AVG Threat labs [5], McAfee WebAdvisor [6], entre outros.

```
{
  "url_black_list": [
    "^127\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})",
    "^192\.168\.([0-9]{1,3})\.([0-9]{1,3})",
    "^255\.255\.255\.255",
    "*\.dangerous.com",
    "dangerous.com/*",
    "^::1"
  ]
}
```

Código Fonte 5.3: Blacklist de endereços com suporte a expressões regulares

[4] http://www.senderbase.org/

[5] http://www.avgthreatlabs.com/ww-en/website-safety-reports/

[6] https://home.mcafee.com/root/landingpage.aspx?lpname=get-it-now

Verificador de Transferência

Dependendo do tipo de aplicação pode ser necessário restringir o tipo dos dados que serão transferidos, por exemplo: uma aplicação de online judge pode permitir que o programa baixe arquivos de texto, enquanto um aplicativo pode consumir apenas conteúdos json. Para implementação da verificação à medida que dados são transferidos o executor pode interceptá-los e realizar uma checagem para garantir que são de um tipo permitido.

Outras restrições relativas à transferência de dados podem ser implementadas para aumentar o nível de segurança da plataforma: limitar a quantidade de dados trafegados, limitar o número de conexões simultâneas, limitar a quantidade de conexões abertas em um determinado período.

5.2.2 Mediador de Arquivos

O monitoramento de arquivos é o módulo que servirá de interface de comunicação entre o código e ambiente de execução e sua principal função é impor restrições de leitura e escrita a fim de evitar que códigos arbitrários consigam acessar qualquer arquivo do sistema. Na Figura 5.5 ilustra-se o fluxo de execução quando o código solicita alguma operação de IO no qual o programa solicita alguma operação IO e o monitorador de arquivos intercepta a requisição e realiza todas as checagens e validações necessárias. Caso a operação não viole as regras estabelecidas, o executor submete a transação para o sistema de arquivos.

Além de restringir o acesso a arquivos do sistema, esse mediador pode implementar técnicas para limitar o tamanho dos arquivos criados e limitar o número de operações IO ao disco, dessa forma, reduzindo o risco de algum código realizar DoS do disco.

5.2.3 Mediador de Recursos

Analogamente ao mediador de acesso a recursos de disco deve haver um mecanismo que monitore a utilização dos demais recursos computacionais tais como CPU e memória. Esses recursos são muito importantes para a manutenção das funções do sistema e têm uma capacidade que pode ser atingida muito rapidamente. Diferentemente dos mediadores de rede e arquivos, a implementação de uma interface para acessar alguns recursos específicos (a exemplo da CPU) pode não ser trivial ou até mesmo possível dependendo da plataforma

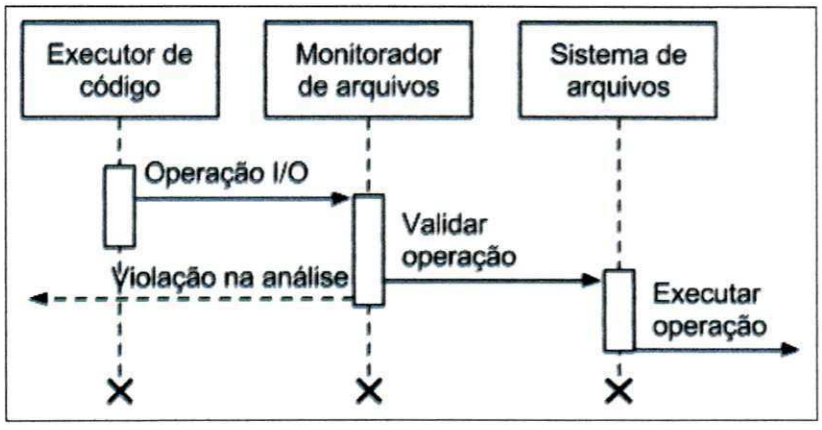

Figura 5.5: Diagrama de sequência do mediador de arquivos

escolhida.

Em tais casos o ideal é criar um sistema que monitore apenas a quantidade de recursos que está sendo utilizada por cada código. No caso de CPU a métrica pode ser tempo de CPU ou ciclos e no caso de memória a métrica pode ser MB e idealmente cada aplicação deve definir um limite máximo para cada recurso que um programa pode utilizar ou, ao invés de usar um tamanho fixo, pode-se utilizar técnicas de alocação.

Capítulo 6

Validação

Neste capítulo é apresentada a validação da abordagem em um ambiente projetado com base no modelo proposto. A aplicação foi desenvolvida com base no ambiente de execução Node.js e simula um cliente de computação voluntária, o qual irá executar códigos não confiáveis em um dispositivo pessoal.

A escolha de simular um cliente de computação voluntária foi baseada no fato que tais sistemas podem executar códigos não confiáveis em qualquer tipo de dispositivo, incluindo computadores pessoais. Dessa forma códigos maliciosos podem tentar causar diversos tipos de danos tais como roubar dados pessoais, danificar arquivos ou até criar *botnets*.

A seguir neste capítulo, serão explicados o ambiente de execução e escolha das tecnologias, detalhes de implementação e métodos utilizados para testar o modelo.

6.1 Plataforma e Tecnologia

O ambiente de execução escolhido para a implementação do modelo foi Node.js o qual é um ambiente desenvolvido com base na *engine* de javascript V8 [1]. Node.js implementa um modelo de arquitetura baseado em eventos IO não bloqueantes o que torna a execução muito eficiente, principalmente para aplicações que possuem muitas operações de leitura e escrita.

Apesar desses benefícios a plataforma foi desenvolvida para ser leve e ter um baixo consumo de memória podendo ser executada nos mais variados dispositivos, desde servidores até dispositivos móveis (como smartphones e tablets). Aliada à sua natureza open-source

[1] https://developers.google.com/v8/

a comunidade de desenvolvimento também é um fator positivo tornando o gerenciador de pacotes do Node.js (NPM)[2] a maior plataforma de bibliotecas open source do mundo.

6.2 Implementação do Modelo

Nessa implementação o ambiente de execução (implementado usando Node.js) servirá para (a) ser o gerenciador de execução e (b) ser o próprio ambiente de execução dos códigos. Do ponto de vista do cliente de computação voluntária, o código que será executado é chamado de *job* e pode ser um código não confiável com diversas violações. Desta forma o gerenciador de execução é o responsável pelo controle do job atuando para interpretar o arquivo descritor, procurar violações de código e monitorar violações em tempo de execução.

Com base em conhecimento prévio de projetos de computação voluntária, as seguintes restrições devem ser observadas com relação à natureza dos jobs:

1. Jobs poderão se conectar somente a um único servidor para enviar ou requisitar dados e só pode haver uma conexão simultânea.
2. Jobs somente poderão baixar arquivos de texto.
3. Jobs só poderão acessar arquivos de um diretório local privado e único a cada job.
4. Jobs não poderão iniciar novos processos.
5. Jobs não poderão mudar o diretório de execução.
6. Jobs só poderão fazer uso da biblioteca *https*.
7. Jobs não podem acessar informações do sistema operacional.
8. Jobs não podem fazer uso de bibliotecas ou códigos nativos.

6.2.1 Arquivo Descritor

Como a estratégia do pré processador consiste em validar o código e permissões, o arquivo descritor deve ser especificado usando uma sintaxe concisa e uniforme para todos os jobs.

[2]https://www.npmjs.com/

```
{
    job_category: "category_id",
    job_id: "job_id",
    job_description: "This job aims to find prime numbers",
    job_url: "https://endpoint.com/my_job.js",
    endpoint : "https://endpoint.com",
    permissions : ["local_storage", "read_platform"]
}
```

Código Fonte 6.1: Arquivo descritor de um job

No Código 6.1 é definida a sintaxe do arquivo descritor de job, o qual contém informações adicionais (*job_category*, *job_description* e *job_id*) que podem ser usadas para atividades extras, por exemplo: *job_description* pode ser usado para exibir para o usuário uma descrição sobre o job que está sendo executado, enquanto *job_id* pode ser usado para salvar métricas de desempenho relacionadas ao job em questão. Já o valor *job_url* é o endereço no qual o arquivo do job em si será baixado.

O valor definido na chave *endpoint* é o único ponto de comunicação entre o job e o seu respectivo gerenciador de projeto, o qual é um servidor responsável por disponibilizar os arquivos necessários para a execução do projeto. Também é papel do gerenciador de projeto receber dados relativos à execução de cada job.

Na Figura 6.1 ilustra-se o fluxo de download e pré processamento de job no qual o ambiente de execução realiza o download do arquivo descritor e usando o valor definido em *job_url* baixa o arquivo do job em si. Em seguida, ambos arquivo descritor e arquivo do job, são analisados para determinar se há violações estáticas.

6.2.2 Analisador

O analisador estático consiste em duas partes: analisar o arquivo descritor e analisar o código do job. Para a aplicação utilizada na validação definiu-se as seguintes regras:

1. Jobs somente poderão usar as seguintes permissões: *local_storage*, *read_platform* e *read_arch*.

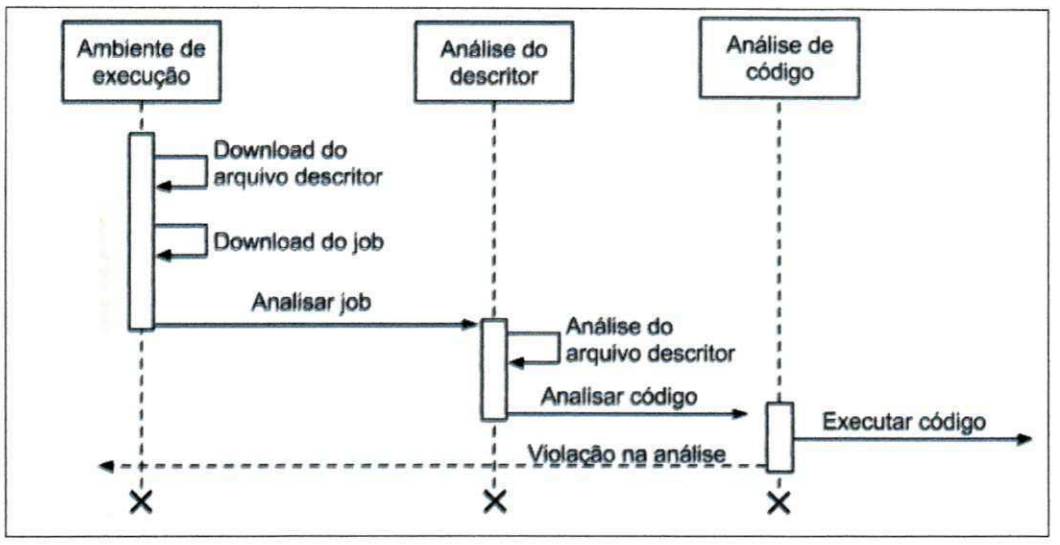

Figura 6.1: Diagrama de sequência de download e pré processamento do job

2. Jobs não poderão acessar endereços diferentes do que foram definidos no arquivo descritor, incluindo expressões regulares.

Durante a análise do descritor basta realizar (1) uma verificação a fim de garantir que todos os atributos estão definidos, (2) verificar se as urls definidas em *job_url* e *endpoint* são válidas e não possuem algum tipo expressão regular e (3) se as permissões definidas em *permissions* pertencem ao conjunto {*local_storage*, *read_platform*, *read_arch*}. Essas etapas estão exemplificadas no Código A.1;

Posteriormente à análise do descritor, o analisador deverá buscar por padrões ou uso de funções e bibliotecas não permitidas. Na aplicação usada nessa validação, foi definido que um job não poderá acessar informações do sistema operacional, dessa forma qualquer leitura de atributos do objeto *process* será considerado uma violação.

Para essa segunda etapa da análise estática todo o código fonte do job é lido à procura desses padrões e, para a situação definida acima, isso pode ser feito usando expressões regulares, conforme ilustrado no trecho de Código A.3. À medida que alguma expressão regular tenha resultado positivo, a execução é abortada imediatadamente, assim como ocorre em caso de violação do arquivo descritor.

6.2.3 Executor

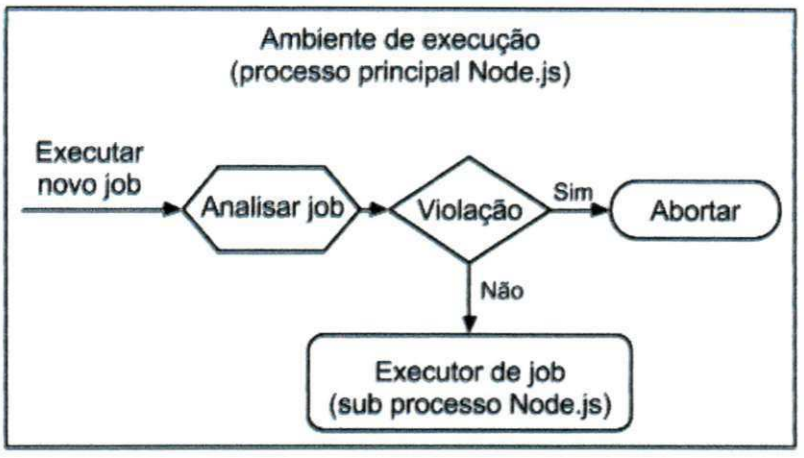

Figura 6.2: Fluxo dos processos de execução do job

Após a análise estática ser bem sucedida o código começa a ser executado em um processo separado, como observado na Figura 6.2 que ilustra o fluxo e relacionamento entre o processo principal e processo filho que irá executar o job. A função desse processo filho é criar um ambiente mais restrito no qual um código mal intencionado não possa causar danos. Essa técnica pode ser entendida como um *sandbox*, a qual deve implementar outras medidas de segurança para garantir que as ações do job fiquem contidas.

Baseado nas restrições do modelo e do job, a fim de evitar que determinadas funções e bibliotecas sejam utilizadas, o mecanismo de segurança do executor pode ser implementado ao sobrescrever a função *Module._load* do Node.js. Essa estratégia, definida no trecho de Código A.7, é bastante eficiente e não adiciona tempo significante para o início da execução, bastando apenas adicionar todas as funções sobrescritas no início do código do job.

Outra etapa crucial na implementação é realizar o mapeamento correto entre as possíveis permissões e suas funções/bibliotecas correspondentes. A estratégia adotada foi sobrescrever todos os métodos relativos a IO bem como os métodos que conseguem ler informações do sistema e fornecer outros métodos proxy que tenham o mesmo efeito. O Código A.8 demonstra como um conjunto de funções pode ser bloqueado para que estas não sejam utilizadas pelo job. A tradução de cada uma das permissões em código deu-se da seguinte forma:

- *local_storage*: o uso da biblioteca *fs* torna-se proibido no escopo do job e para ler ou escrever arquivos foram criadas as funções *writeFile()* e *readFile()* (Código A.5). Dessa forma o job só terá acesso ao sistema de arquivos através dessas duas funções.

- *read_platform*: o uso da biblioteca *os* e leitura do objeto *process.platform* tornam-se proibidos no escopo do job e para acessar informações sobre a plataforma foi criada a função *getPlatformInfo()*, como pode ser observado no Código A.4.

- *read_arch*: da mesma forma que o método anterior, porém foi criada a função *getCpuInfo()*.

Além da criação de interfaces para operações I/O em disco também foi criada outra interface para comunicação em rede, a qual monitora todas as transferências e só permite que sejam transmitidos conteúdos de texto bem como só permite uma conexão ativa por vez. Esse processo de verificação pode ser observado no Código A.2. Por fim, a última parte do executor diz respeito ao monitoramento de recursos físicos e nessa implementação, como observado no Código A.6, apenas o uso de memória RAM é monitorado de modo que, quando o processo filho excede o máximo definido, o processo pai termina imediatamente a execução a fim de evitar sequestro de recursos e possíveis ataques de DoS locais.

6.3 Experimento

A implementação e execução do experimento foi realizada com Node.js versão v4.4.7 LTS e para realizar o experimento e testar a implementação do modelo foi necessário criar scripts que simulem códigos mal intencionados ou que não estejam conforme alguma restrição previamente definida no modelo. A elaboração dos scripts foi realizada com base nos requisitos definidos na Seção 4.4, que são os requisitos base do modelo, e nas restrições definidas na Seção 6.2, que listam as regras referentes ao aplicativo que simula um cliente de computação voluntária.

As primeiras simulações são referentes ao ponto de entrada das aplicações, ilustradas nos trechos de Código 6.2 e 6.3 que fornecem, respectivamente, arquivos descritores com valores não permitidos para as chaves *endpoint* e *permissions*. O intuito deste teste é simplesmente validar que o arquivo descritor está bem formado e que contém apenas valores permitidos.

```
{
  job_category: "category_id",
  job_id: "job_id",
  job_description: "This job aims to find prime numbers",
  job_url: "https://endpoint.com/my_job.js",
  endpoint : "https://*.endpoint.com/*",
  permissions : ["local_storage", "read_platform"]
}
```

Código Fonte 6.2: Arquivo descritor com expressão regular no campo *endpoint*

```
{
  job_category: "category_id",
  job_id: "job_id",
  job_description: "This job aims to find prime numbers",
  job_url: "https://endpoint.com/file.js",
  endpoint : "https://endpoint.com",
  permissions : ["write_system"]
}
```

Código Fonte 6.3: Arquivo com uma permissão inválida no campo *permissions*

A próxima etapa nos testes foi verificar que o componente de comunicação com a internet atende às restrições impostas no arquivo descritor e à restrição específica do cliente que diz respeito a conexões simultâneas a partir de um mesmo job. Dessa forma o código ilustrado em 6.4 tenta estabelecer conexão com um host diferente do que foi definido no seu arquivo descritor e o Código 6.5 tenta estabelecer múltiplas conexões com o mesmo host.

```
const https = require('https');

//allowed endpoint: https://endpoint.com
https.get('https://another.com/', (res) => {
  res.on('data', (d) => {
    processData(data);
  });
});
```

Código Fonte 6.4: Trecho de código que tenta estabeler conexão com uma url diferente da

que foi especificada no descritor

```
const https = require('https');

https.get('https://endpoint.com/', (res) => {
  res.on('data', (d) => {
    processData(data);
  });
});

//fails because simultaneous connections are not allowed
https.get('https://endpoint.com/', (res) => {
  res.on('data', (d) => {
    processData(data);
  });
});
```

Código Fonte 6.5: Trecho de código que tenta estabelecer múltiplas conexões com o mesmo *host*

Outra validação no componente de mediação de internet é garantir que jobs só possam transferir dados do tipo texto e o Código 6.6 tenta baixar um arquivo cujo conteúdo é do tipo *xml*.

```
const https = require('https');

//allowed content is: text/plain
https.get('https://endpoint.com/xml_file.xml', (res) => {
  // res.headers['content-type'] returns application/xml
  res.on('data', (d) => {
    processData(data);
  });
});
```

Código Fonte 6.6: Trecho de código que tenta baixar um tipo de arquivo não permitido

Os componentes de checagem estática e até o próprio executor devem ser capaz de bloquear o uso de determinadas funções e bibliotecas, sendo assim, os Códigos 6.7 e 6.8 tentam utilizar, respectivamente, bibliotecas e funções não permitidas enquanto o Código 6.9 tenta

usar uma biblioteca nativa.

```
const path = require('path');

//path library is not allowed
console.log(path.isAbsolute('.'));
```

Código Fonte 6.7: Trecho de código que tenta utilizar uma biblioteca não permitida

```
//process.platform is not allowed
console.log(process.platform);
```

Código Fonte 6.8: Trecho de código que tenta acessar uma função/propriedade não permitida

```
//the usage of native libs through process.binding is not allowed
const binding = process.binding('os');
console.log(binding.getOSType());
```

Código Fonte 6.9: Trecho de código que tenta utilizar uma biblioteca nativa

A implementação do mediador de arquivos foi testada através do Código 6.10, o qual tenta realizar uma operação não permitida de leitura e escrita de um arquivo fora do seu espaço privado.

```
//cannot write files outside the current directory
writeFile('../file.txt', function (data){
})

//cannot read files outside the current directory
readFile('/system_file.txt', function (data){
})
```

Código Fonte 6.10: Trecho de código que tenta acessar arquivos fora do espaço privado

Já o mediador de recursos foi testado através do trecho de Código 6.11 o qual realiza uma série de operações que alocam e retém uma grande quantidade de memória.

```
var dataArr = [];
for(var i =0 i < 999999; i++){
  for(var j =0 j < 999999; j++){
    readFile('large_file.txt', function (data){
    //this allocates to much memory
```

```
    dataArr.push(data);
    });
  }
}
```

Código Fonte 6.11: Trecho de código que tenta realizar operações que irão alocar muita memória

A integridade de outros programas e processos do sistema depende que o código esteja isolado e que não possa criar ou interagir diretamente com qualquer outro processo. O trecho de Código 6.12 foi utilizado para testar esse requisito, o qual tenta iniciar um novo processo arbitrário.

```
var childProcess = require('child_process'),
process = childProcess.spawn();
```

Código Fonte 6.12: Trecho de código que tenta iniciar um novo processo

A partir da execução dos scripts definidos acima, observou-se que a implementação obteve sucesso ao identificar as violações e bloquear as chamadas proibidas, assim como identificar e bloquear as violações referentes ao uso de recursos. De modo geral pode-se concluir através do experimento que o modelo e a implementação proposta nessa seção foram bem sucedidos pois (a) códigos inicializados dentro do ambiente de execução restrito não conseguem evadir seu espaço privado e interferir na execução de outros programas ou acessar outros arquivos, (b) a implementação de restrições através de arquivos descritores é uma forma eficiente para definir o "perfil" de programas e bloquear comportamentos que fujam a esse perfil, (c) a intermediação de rede é um método eficaz para identificar uso anormal de internet e que possa causar ataques DoS remoto, (d) a pré checagem de código é útil não só para procurar por padrões maliciosos, mas também para verificar uso de funções não permitidas e (e) o monitoramento do uso de outros recursos físicos é uma forma eficiente de evitar que códigos realizem DoS local.

Capítulo 7

Considerações Finais

Este capítulo apresenta as considerações finais desta dissertação. Nas seções seguintes são apresentadas as principais contribuições do trabalho e alguns trabalhos futuros que podem ser agregados para contribuir com a solução proposta.

7.1 Conclusão e Contribuições

Este trabalho visou a elaboração de um conjunto de medidas que possam ser utilizadas para criar um ambiente no qual seja possível executar códigos não confiáveis sem que haja comprometimento na segurança do sistema ou do usuário. Como resultado, foi apresentado um conjunto de requisitos e definição de um modelo que faz uso de técnicas para avaliar e restringir as ações de programa.

Os requisitos para o modelo foram elaborados após uma análise dos principais tipos de ameaças, a destacar suas formas de atuação, propagação e evasão de sistemas detectores. Nessa etapa foram definidos 7 requisitos que visam guiar a elaboração de modelos de segurança os quais buscam proteger o ambiente de execução de forma geral, ou seja, tanto processos e arquivos locais quanto outros dispositivos não podem ser afetados.

A partir desses requisitos, elaborou-se o modelo de proteção que foi definido em duas partes: um pré processador de código e um executor. Do ponto de vista do fluxo, a primeira parte do modelo é o pré processador, o qual deverá realizar uma leitura do código fonte e procurar por uso de funções não permitidas e, adicionalmente, poderá procurar por padrões de código que sejam considerados maliciosos. Como parte do pré processador também foi

elaborado um modelo de arquivo descritor que serve como ponto de entrada para a aplicação e adiciona regras que restringem seu comportamento.

A segunda parte do modelo proposto define as atribuições do executor, que deve ser o responsável por executar os códigos e realizar uma série de monitoramentos, como tráfego de rede, acesso a arquivos e uso de recursos para evitar potenciais ações perigosas. Nesse ponto o modelo também irá prevenir contra abuso no uso de recursos, pois uma aplicação maliciosa pode sequestrar tais recursos e causar DoS local.

A validação do modelo foi realizada através de um experimento que simula um cliente de computação voluntária que implementa todos os módulos definidos no modelo e ainda adiciona mais algumas restrições específicas da aplicação. A implementação foi feita em Node.js para todas as camadas (analisador e executor) e scripts maliciosos foram criados para validar a implementação.

Além da aplicação utilizada no experimento, outras ferramentas que podem fazer uso do modelo apresentado incluem: (a) perfiladores de código: executam códigos arbitrários, podendo ser maliciosos ou não, para extrair métricas de uso e entender seu comportamento, podendo inclusive ser usado para encontrar gargalos ou erros de implementação de algum software; (b) online judge que são aplicativos que rodam em geral na nuvem e executam códigos de diversas plataformas que podem ser submetidos por qualquer usuário; (c) gerenciadores de plugins: são ferramentas que fazem parte de outro aplicativo e tem como intuito adicionar funcionalidades que podem ser criadas por diversos usuários.

De forma geral, o trabalho apresentado contribui para a elaboração de sistemas ou implementação de plataformas que necessitem executar códigos desconhecidos, tais como o experimento realizado neste trabalho ou os exemplos citados acima. Plataformas mais complexas também podem integrar as técnicas apresentadas em seus mecanismos de segurança ou até utilizá-las para criar monitores de alocação e uso de recursos.

7.2 Trabalhos Futuros

Essa seção discute alguns aspectos que não foram abordados no modelo ou que podem ser melhorados de forma que a solução proposta seja mais robusta.

Uma das limitações do modelo é a possibilidade de executar códigos nativos ou criar

outros processos. Por código nativo entende-se que são arquivos binários compilados para executar instruções de máquina ou realizar system calls diretamente. A partir da execução desses códigos nativos torna-se complicado realizar uma análise semântica e interceptar chamadas a fim de monitorar uso de recursos que devem ser protegidos, tais como sistema de arquivos ou uso de rede.

Outra limitação do modelo é que há a necessidade de especificar através de *whitelist* ou *blacklist* uma lista de endereços url aos quais a aplicação pode, ou não, estabelecer conexões. A depender da aplicação, a lista de endereços utilizadas é muito grande, desse modo, uma alternativa seria a criação de uma lista pré-definida de *hosts* confiáveis que são comumente usados em diversas aplicações, tais como google.com, facebook.com e twitter.com. Além disso pode-se expandir o modelo para que exceções sejam adicionadas para permitir o tráfego de dados de fontes também tidas como confiáveis sem as restrições impostas no modelo.

No entanto, há outros aspectos que merecem estudos aprofundados para que possam se tornar mais eficientes e possam ser integrados no futuro, a exemplo da granularidade das permissões. Uma plataforma deve realizar a análise de código com base nas permissões que foram definidas no arquivo, porém alguma implementação desse modelo pode definir um tipo de permissão que seja muito abrangente permitindo, assim, que o código execute funções além do planejado, criando um ponto vulnerável. Em contra partida, um modelo que seja muito restrito pode exigir que um programa declare várias permissões para realizar ações mais simples. Finalmente, fica como sugestão para outro trabalho futuro a forma como as permissões são tratadas pelo ambiente de execução e a interação com o usuário, uma vez que o usuário ou administrador do sistema deve ter noção do que cada permissão significa e também deve ter a opção de escolher quais quer liberar ou bloquear.

Bibliografia

[1] M. S. Ahmad, N. E. Musa, R. Nadarajah, R. Hassan, and N. E. Othman. Comparison between android and ios operating system in terms of security. In *Information Technology in Asia (CITA), 2013 8th International Conference on*, pages 1–4, July 2013. doi: 10.1109/CITA.2013.6637558.

[2] M. Akiyama, T. Yagi, and M. Itoh. Searching structural neighborhood of malicious urls to improve blacklisting. In *Applications and the Internet (SAINT), 2011 IEEE/IPSJ 11th International Symposium on*, pages 1–10, July 2011. doi: 10.1109/SAINT.2011.11.

[3] T. Bläsing, L. Batyuk, A. D. Schmidt, S. A. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 55–62, Oct 2010. doi: 10.1109/MALWARE.2010.5665792.

[4] Paulo Picota Cano and Miguel Vargas-Lombardo. Security threats in volunteer computing environments using the berkeley open infrastructure for network computing (boinc). *International Journal of Computer Technology and Applications*, 3(3), 2012.

[5] V. N. Cooper, H. Shahriar, and H. M. Haddad. A survey of android malware characterisitics and mitigation techniques. In *Information Technology: New Generations (ITNG), 2014 11th International Conference on*, pages 327–332, April 2014. doi: 10.1109/ITNG.2014.71.

[6] David Dagon, Guofei Gu, Cliff Zou, Julian Grizzard, Sanjeev Dwivedi, Wenke Lee, and Richard Lipton. A taxonomy of botnets. In *IN: PROCEEDINGS OF CAIDA DNS-OARC WORKSHOP*, 2005.

[7] Mohsen Damshenas, Ali Dehghantanha, and Ramlan Mahmoud. A survey on malware propagation, analysis, and detection. *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, 2(4):10–29, 2013.

[8] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE Security Privacy*, 7(1):50–57, Jan 2009. ISSN 1540-7993. doi: 10.1109/MSP.2009.26.

[9] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011.

[10] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 7–7, 2011.

[11] DF Ferraiolo and R Kuhn. Role-based access controls. In *Proc. of 15th NIST-NSA National Computer Security Conference*, 1992.

[12] R. Ford and W. H. Allen. Malware shall greatly increase ... *IEEE Security Privacy*, 7 (6):69–71, Nov 2009. ISSN 1540-7993. doi: 10.1109/MSP.2009.181.

[13] S. Jana, D. E. Porter, and V. Shmatikov. Txbox: Building secure, efficient sandboxes with system transactions. In *2011 IEEE Symposium on Security and Privacy*, pages 329–344, May 2011. doi: 10.1109/SP.2011.33.

[14] H. J. Li, C. W. Tien, C. W. Tien, C. H. Lin, H. M. Lee, and A. B. Jeng. Aos: An optimized sandbox method used in behavior-based malware detection. In *Machine Learning and Cybernetics (ICMLC), 2011 International Conference on*, volume 1, pages 404–409, July 2011. doi: 10.1109/ICMLC.2011.6016683.

[15] Z. Li, H. Cai, J. Tian, and W. Chen. Application sandbox model based on system call context. In *Communications and Mobile Computing (CMC), 2010 International Conference on*, volume 1, pages 102–106, April 2010. doi: 10.1109/CMC.2010.77.

[16] Q. Liu and Y. Mao. A hooking interpreter based method for script program protection. In *Control Engineering and Communication Technology (ICCECT), 2012 International Conference on*, pages 673–676, Dec 2012. doi: 10.1109/ICCECT.2012.222.

[17] Shukun Liu and Weijia Jia. A survey: Main virtualization methods and key virtualization technologies of cpu and memory. *The Open Cybernetics & Systemics Journal*, 10 (9):350–358, May 2015. ISSN 1874-110X. doi: 10.2174/1874110X01509010350.

[18] Wanping Liu, Chao Liu, Xiaoyang Liu, Shaoguo Cui, and Xianying Huang. Modeling the spread of malware with the influence of heterogeneous immunization. *Applied Mathematical Modelling*, 40(4):3141–3152, 2016.

[19] U. Lösche, M. Morgenstern, and H. Pilz. Platform independent malware analysis framework. In *IT Security Incident Management IT Forensics (IMF), 2015 Ninth International Conference on*, pages 109–113, May 2015. doi: 10.1109/IMF.2015.21.

[20] Steve Mansfield-Devine. The promise of whitelisting. *Network Security*, 2009(7):4–6, 2009.

[21] J. A. P. Marpaung, M. Sain, and Hoon-Jae Lee. Survey on malware evasion techniques: State of the art and challenges. In *Advanced Communication Technology (ICACT), 2012 14th International Conference on*, pages 744–749, Feb 2012.

[22] G. A. McGilvary, A. Barker, A. Lloyd, and M. Atkinson. V-boinc: The virtualization of boinc. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 285–293, May 2013. doi: 10.1109/CCGrid.2013.14.

[23] I. Mohamed and D. Patel. Android vs ios security: A comparative study. In *Information Technology - New Generations (ITNG), 2015 12th International Conference on*, pages 725–730, April 2015. doi: 10.1109/ITNG.2015.123.

[24] S. Peng, S. Yu, and A. Yang. Smartphone malware and its propagation modeling: A survey. *IEEE Communications Surveys Tutorials*, 16(2):925–941, Second 2014. ISSN 1553-877X. doi: 10.1109/SURV.2013.070813.00214.

[25] R. S. Pirscoveanu, S. S. Hansen, T. M. T. Larsen, M. Stevanovic, J. M. Pedersen, and A. Czech. Analysis of malware behavior: Type classification using machine learning. In *Cyber Situational Awareness, Data Analytics and Assessment (CyberSA), 2015 International Conference on*, pages 1–7, June 2015. doi: 10.1109/CyberSA.2015.7166115.

[26] J. Powers, R. Smith, Z. Korkmaz, and H. Ahmed. Whitelist malware defense for embedded control system devices. In *2015 Saudi Arabia Smart Grid (SASG)*, pages 1–6, Dec 2015. doi: 10.1109/SASG.2015.7449271.

[27] H. Prähofer, F. Angerer, R. Ramler, H. Lacheiner, and F. Grillenberger. Opportunities and challenges of static code analysis of iec 61131-3 programs. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012)*, pages 1–8, Sept 2012. doi: 10.1109/ETFA.2012.6489535.

[28] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java runtime systems: characterization and architectural implications. *IEEE Transactions on Computers*, 50(2):131–146, Feb 2001. ISSN 0018-9340. doi: 10.1109/12.908989.

[29] Z. C. Schreuders, C. Payne, and T. McGill. Techniques for automating policy specification for application-oriented access controls. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 266–271, Aug 2011. doi: 10.1109/ARES.2011.47.

[30] Z. C. Schreuders, C. Payne, and T. McGill. A policy language for abstraction and automation in application-oriented access controls: The functionality-based application confinement policy language. In *Policies for Distributed Systems and Networks (POLICY), 2011 IEEE International Symposium on*, pages 113–116, June 2011. doi: 10.1109/POLICY.2011.11.

[31] Z. Cliffe Schreuders, Tanya McGill, and Christian Payne. The state of the art of application restrictions and sandboxes: A survey of application-oriented access controls and their shortfalls, 2013.

[32] B. Sun, M. Akiyama, T. Yagi, M. Hatada, and T. Mori. Autoblg: Automatic url blacklist generator using search space expansion and filters. In *2015 IEEE Symposium on Computers and Communication (ISCC)*, pages 625–631, July 2015. doi: 10.1109/ISCC.2015.7405584.

[33] Michael E. Whitman and Herbert J. Mattord. Principles of information security, 3ª edição, thompson course technology, 2007.

[34] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, May 2009. doi: 10.1109/SP.2009.25.

Apêndice A

Código fonte do experimento

```
//checks for malformed descriptor
if (typeof descriptor.job_url === 'undefined' ||
           descriptor.endpoint === 'undefined' ||
       descriptor.permissions === 'undefined' ||) {
  system.exit(1);
}

//checks for valid url
var urlPattern = /(http|https):\/\/[\w-]+(\.[\w-]+)+([\w.,@?^=%&amp
    ;:\/~+#-]*[\w@?^=%&amp;\/~+#-])?/
if (!descriptor.job_url.match(urlPattern) ||
    !descriptor.endpoint.match(urlPattern) || ){
  system.exit(1);
}

//checks for valid permissions
var allowedPermissions = ['local_storage', 'read_platform', 'read_arch'];
for (var i =0; i < descriptor.permissions.length; i++){
  if (allowedPermissions.indexOf(descriptor.permissions[i]) < 0){
    system.exit(1);
  }
}
```

Código Fonte A.1: Validação do arquivo descritor

```
//checks for content and multiple connections
var terminated = true;
https.get = function (url, callback){
  if (url !==  descriptor.endpoint){
    system.exit(1);
    return;
  }
  if (!terminated){
    system.exit(1);
    return;
  }
  realHttps.get(url, (res) => {
    terminated = false;
    if (res.headers['content-type'] != 'text/plain'){
      request.abort();
    }
    var data = '';
    res.on('data', (d) => {
      data += d;
    });
  res.on('end', (d) => {
      terminated = true;
      callback(data);
    });
  });
}
```

Código Fonte A.2: Monitor de comunicação via https

```
//checks for prohibited patterns
const requireRegex = /process\./g;
if (code.match(requireRegex)){
  system.exit(1);
}
```

Código Fonte A.3: Verificação de padrões não permitidos

```
//proxy methods
getPlatformInfo = function(){
  return process.platform;
}

getCpuInfo() = function(){
  return os.cpuInfo();
}
```

Código Fonte A.4: Funções proxy

```
//checks for files read/write
fileIsLocal = function(fileName){
  return !path.isAbsolute(fileName) &&
          path.dirname == __dirname;
}

readFile = function(fileName, callback){
  if(!fileIsLocal(fileName)){
    system.exit(1);
    return;
  }
  fs.readFile(fileName, function (err, data) {
    callback(err, data);
  });
}

writeFile = function(fileName, content, callback){
  if(!fileIsLocal(fileName)){
    system.exit(1);
```

```
    return;
  }
  fs.writeFile(fileName, content, function (err, data) {
    callback(err, data);
  });
}
```

Código Fonte A.5: Interface para leitura e escrita de arquivos

```
//monitors memory usage
setInterval(function(){
  child.send('mem_usage');
}, 1000);

child.on('message', function(payload){
  var mem = payload.memUsage.heapTotal / 1024 / 1024;
  //kills the process if memory is more than 25 MB
  if (mem > 25){
    child.kill(1);
  }
});
```

Código Fonte A.6: Monitor de consumo de memória

```
//block requires
mModule._load = function(request, parent, isMain) {
  var currModule = originalModuleLoader(request, parent, isMain);
  var moduleFunctions = blockedFcuntions[request];
  if (moduleFunctions){
    for (var i =0; i < moduleFunctions.length; i++){
      var currentFunc = moduleFunctions[i];
      if (currModule[currentFunc]){
        currModule[currentFunc] = function harden(){
          system.exit(1);
```

```
        };
      }
    }
  }
  return currModule;
}
```

Código Fonte A.7: Verificador de uso de bibliotecas não permitidas

```
var keys = Object.keys(blockedLocal);
for (var i =0; i < keys.length; i++){
  var currentParent = keys[i];
  var methods = blockedLocal[currentParent];
  for (var j =0; j < methods.length; j++){
    var funcName = currentParent + '.' + methods[j];
    global[currentParent][methods[j]] = function (){
      system.exit(1);
    };
  }
}
```

Código Fonte A.8: Verificador de uso de funções não permitidas