CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DO PARANÁ
Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

---

DISSERTAÇÃO
apresentada ao CEFET-PR
para obtenção do título de

MESTRE EM CIÊNCIAS

por

GUILHERME HERRMANN DESTEFANI

VERIFICAÇÃO OPORTUNISTA DE ASSINATURAS DIGITAIS PARA PROGRAMAS E BIBLIOTECAS EM SISTEMAS OPERACIONAIS PAGINADOS

Banca Examinadora:

Presidente e Orientador:
Prof. Dr. Flavio Neves Júnior — CEFET-PR

Examinadores:
Prof. Dr. Carlos Alberto Maziero — PUC-PR
Prof. Dr. Bruno Müller Júnior — UFPR
Prof. Dr. Douglas Paulo Bertrand Renaux — CEFET-PR

Curitiba - PR, 11 de março de 2005.

Guilherme Herrmann Destefani

VERIFICAÇÃO OPORTUNISTA DE ASSINATURAS DIGITAIS PARA PROGRAMAS E BIBLIOTECAS EM SISTEMAS OPERACIONAIS PAGINADOS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial do Centro Federal de Educação Tecnológica do Paraná, na área de concentração de Informática Industrial, como requisito parcial à obtenção do título de Mestre em Ciências.

Orientador: Prof. Dr. Flávio Neves Júnior

Curitiba
2005

Agradecimentos

A Deus, por tornar tudo possível.

À Maria Herrmann Destefani e Douglas Destefani, pela ajuda que possibilitou que eu chegasse até aqui, além das correções ao texto.

Ao Prof. Dr. Flávio Neves Júnior, pela orientação acadêmica.

Ao Marcus Andreotti, pelo incentivo a continuar pesquisando e pelas sugestões e correções ao trabalho.

À Helix/Perkons, por enxergar mais longe e perceber que a pesquisa é o maior investimento que uma empresa pode fazer.

Sumário

Agradecimentos **iii**

Lista de Figuras **ix**

Lista de Tabelas **xi**

Glossário, Abreviaturas e Siglas **xiii**

Resumo **xvii**

Abstract **xix**

1 Introdução **1**
1.1 Funcionamento do sistema proposto 1
1.2 Combinação com outros mecanismos de segurança 3
1.3 Impacto sobre o desempenho do sistema 4
1.4 Estrutura e objetivos 4

2 Estado da Arte **7**
2.1 Identificação dos problemas que originaram o trabalho 7
2.2 Descrição dos trabalhos existentes 10
2.3 Mecanismos de segurança do microprocessador 10
2.3.1 Controle de níveis de privilégios 11
2.3.2 Controle da memória 14
2.3.3 Proteção utilizando segmentação 15
2.3.4 Proteção utilizando paginação 17
2.4 Assinaturas Digitais 20
2.4.1 Pacotes Java™(arquivos *.jar*) 20
2.4.2 A tecnologia *authenticode*™(Microsoft) 21
2.4.3 Assinatura de pacotes *RPM* 23
2.4.4 Características comuns ao uso de assinaturas digitais 23
2.5 Verificação de integridade via *hash* 25
2.5.1 Verificações periódicas de integridade 25
2.5.2 Verificação de integridade em tempo de execução 26
2.6 Provas de segurança 30

2.6.1 Sistema operacional com segurança completamente baseada em análise de código . . . 31
2.7 Considerações finais . . . 32

3 Transformadas Criptográficas 35
3.1 Uso de *hash* . . . 35
3.1.1 Funções de *hash* adequadas para a aplicação . . . 35
3.1.2 Aplicação de uma tabela de *hash* . . . 37
3.2 Uso de estrutura de chaves públicas . . . 38
3.2.1 Uso de assinaturas digitais . . . 38
3.2.2 Forma de distribuição . . . 39
3.2.3 Arquitetura de assinaturas . . . 39
3.2.4 Uso de um conjunto mínimo de *software* para validar o resto do sistema . . . 40
3.2.5 Funcionamento do sistema proposto . . . 41
3.2.6 Considerações relativas à segurança do modelo . . . 43
3.3 Uso da criptografia no sistema . . . 44

4 Análise Oportunista Durante a Execução 45
4.1 Mecanismo de verificação . . . 45
4.2 Segurança da abordagem . . . 46
4.3 Implementação e validação . . . 47
4.4 Formas de aumentar o desempenho da abordagem . . . 49
4.4.1 Uso do *cache* de disco para reduzir o número de verificações . . . 50
4.4.2 Uso de *hash* por página . . . 51
4.4.3 Verificação preditiva de páginas mapeadas em memória . . . 53
4.5 Relação entre a verificação e o sistema completo . . . 56

5 Assinatura Digital Integrada 57
5.1 Formas de uso do *ELF* . . . 58
5.2 Uso de um arquivo *ELF* para *linking* . . . 58
5.2.1 Cabeçalhos de sessão . . . 58
5.3 Uso de um arquivo *ELF* para execução . . . 59
5.3.1 Cabeçalhos de programas . . . 59
5.4 Relocação de código e uso de um interpretador como *dynamic linker* . . . 61
5.4.1 Cabeçalhos de mapeamento de memória . . . 63
5.5 A proposta de extensão do padrão *ELF* . . . 65
5.5.1 Objetivos da extensão . . . 65
5.5.2 Mudanças e adições ao padrão *ELF* . . . 66
5.6 Uso da assinatura digital integrada pelo sistema de verificação . . . 69

6 Impacto no Desempenho do Sistema Operacional 71
6.1 Objetivo dos testes . . . 71
6.2 Hardware . . . 71
6.3 Sistema Operacional . . . 72

6.4 Verificação de assinaturas 72
6.4.1 O mecanismo de carga preditiva do *Kernel* 74
6.5 Forma de medida 75
6.5.1 Inicialização do sistema 76
6.5.2 Programas avaliados 77
6.5.3 Tempo total de execução dos testes 85

7 Conclusões **87**

8 Trabalhos Futuros **89**
8.1 Usos alternativos para o sistema 89
8.2 Uso de outras formas matemáticas de verificação 89
8.3 Estudo do modelo de verificação preditiva 90
8.4 Estudo do problema de inversão de prioridades 90
8.5 Estudo de um padrão expansível de controle de chaves e assinatura digital 90

Referências Bibliográficas **91**

Lista de Figuras

2.1 Verificação do nível de segurança 12
2.2 Desvio do fluxo de execução 13
2.3 Estado após a exceção 13
2.4 Distribuição da memória do sistema 15
2.5 Exemplo de processador em modo protegido, utilizando segmentação . . . 16
2.6 Exemplo de memória paginada 18
2.7 Exemplo de dois endereços virtuais que utilizam um mesmo endereço físico 19
2.8 Assinatura embutida no binário 27
2.9 Construção de um arquivo contendo o binário e a assinatura 28
2.10 Uso de uma base de assinaturas 29

3.1 Tabela contendo o *hash* de todos os arquivos do sistema operacional 37
3.2 Exemplo de encadeamento de chaves 39
3.3 Exemplo de sistema utilizando uma árvore de certificações 40
3.4 Configuração de um sistema com *bootstrap* para verificação de assinaturas . 41
3.5 Relações entre certificadores e verificação para sistema baseado em chaves públicas 42

4.1 Mapa dos pontos em que é possível inserir verificações de integridade . . . 46
4.2 Forma como o sistema busca por interpretadores para um dado programa ou biblioteca 48
4.3 Funcionamento de agrupamento de serviços em um serviço pai 50
4.4 Uso de *hash* por página mapeada em memória 51
4.5 Processo de mapeamento de memória e carga de páginas 52

5.1 Formato *ELF*, destacando as estruturas de *linking* 59
5.2 Formato *ELF*, destacando as áreas de mapeamento em memória 60
5.3 Correspondência entre arquivo mapeado em memória e a imagem do processo 64
5.4 Exemplo de um arquivo *ELF* com assinatura digital embutida 67
5.5 Formato da sessão com as estruturas de controle da assinatura digital . . . 68

6.1 Gráfico de impacto sobre a velocidade de *boot* do sistema 76
6.2 Gráfico de impacto sobre a velocidade do programa *md5sum* 81
6.3 Gráfico de impacto sobre a velocidade do programa *tar* 81
6.4 Gráfico de impacto sobre a velocidade do editor de imagens *gimp* 84
6.5 Gráfico de impacto sobre a velocidade do programa Open Office 85

6.6 Gráfico de impacto sobre o tempo de todo o ciclo de testes 86

Lista de Tabelas

6.1 Resultado dos testes para o tempo de *boot* 76
6.2 Resultado dos testes para o comando md5sum 78
6.3 Resultado dos testes para o comando tar 82
6.4 Resultado dos testes para o comando gimp 83
6.5 Resultado dos testes para o comando oowriter 84
6.6 Resultado do desempenho para o conjunto de testes 86

Glossário, Abreviaturas e Siglas

ActiveX™	Sistema de Componentes *Microsoft*
applet	Mini aplicativo em linguagem *Java*™
Authenticode™	Sistema de assinatura de programas *Microsoft*
backward compatible	Compatível com versões anteriores
boot	Processo efetuado para ligar o sistema operacional
bootstraping	Ato de alterar o *boot* normal do sistema para efetuar uma ação extra
buffer	Espaço temporário utilizado para computações
buffer overflow	Ataque que usa uma falha no controle do tamanho do buffer para sobre-escrever áreas de memória do processo
bug	Falha em um programa
cache	Sistema de otimização de memória que visa manter em uma memória veloz as informações mais utilizadas
checksum	Número utilizado para verificar a integridade de arquivos
COM™	*Component Object Model*, ou modelo de objetos por componentes
copy on write	Processo de efetuar uma duplicação de uma página de memória compartilhada apenas quando é tentado efetuar uma escrita na página
CRC	Implementação de um *checksum* através do resto da divisão dos dados por um polinômio fixo
CRHF	*collision resistant hash function*, função de *hash* resistente a colisão
daemon	Processo dito sobrenatural no sentido de sempre estar executando, ou seja, não tem um ciclo de vida como os processos comuns
denial of service	Ataque que visa derrubar um serviço de rede através de uma série de requisições iguais às legítimas
desktop	Sistema computacional para usuários finais
DNS	*Domain Name Service*, servidor de nomes de domínio da *Internet*
device drivers	Componentes de software que comunicam com um *hardware* específico em benefício do sistema operacional
Drivers	Idem
dynamic linker	Programa que efetua a ligação do processo com as bibliotecas dinâmicas em tempo de execução
ELF	*Executable and Linkable Format*, ou Formato Executável e Ligável
EXT3	Terceiro sistema de arquivos *Linux* extendido
Firewall	Filtro de pacotes de rede
Flags	Espaço reservado de informação utilizado para sinalizar algum estado

GB	1024 MB
GNU	*Gnu's not Unix*, ou *GNU* não é *Unix*
hardware	Componente eletrônico de um sistema computacional
Harvard	Universidade que deu o nome a uma arquitetura de acesso a memória e barramento utilizada em processadores digitais de sinais
hash	Função criptográfica unidirecional, do verbo picar, fazer em pedaços
honey pot	Denota um sistema computacional que é utilizado como isca para detecção de invasões
HTTP	Protocolo de transferência de *Hypertexto*, ou servidor *WEB*
i386	Arquitetura de microprocessadores desenvolvida pela Intel
inode	Estrutura de dados que contém informações relativas a arquivos
I/O	Input/Output, ou entrada e saída, denota dispositivos de comunicação com hardwares periféricos
java™	Linguagem de programação desenvolvida pela empresa Sun
kB	1024 bytes
kernel	Núcleo do sistema operacional
lazy binding	Processo de ligação preguiçosa de programas e bibliotecas, consiste em ligar somente quando preciso
ld	Nome comum do linker em sistemas *Unix*
linker	Programa que efetua o processo de ligação
linking	Processo de efetuar conexão de símbolos
link	Processo de ligação
Linux™	*Linux* é um sistema operacional aberto para pesquisa e desenvolvimento, criado por Linus Tovalds
LOG	Registro do sistema, para fins de gravação de eventos
LSM	*Linux Security Modules*, padrão de desenvolvimento de módulos para implementações de sistemas de segurança no *Kernel Linux*
MAC	*Message Autentication Code*, ou código de autenticação de mensagem
Man-in-the-middle	Homem no meio, ataque que consiste em servir de ponte para uma comunicação , possibilitando a escuta e adulteração de dados
MB	1024 kB
MD5	*Message Digest 5*, tipo de algoritmo de hash
MDC	*Modification Detection Code*, ou Código para Detecção de Modificações
offset	Deslocamento

overhead	Carga adicionada por um sistema
OWHF	*one way hash function*, função de hash unidirecional
page fault	Falha de página, interrupção gerada pelo microprocessador quando um processo tenta acessar um endereço virtual sem uma página correspondente em memória
PC	*Personal Computer*, ou computador pessoal
PF	Abreviatura de *Page Fault*, exceção emitida quando um processo acessa uma área de memória que não tem uma página correspondente em *RAM*
OpenPGP	*Open Pretty Good Privacy*, padrão de criptografia de mensagens com suporte a hierarquia de chaves públicas
POSIX	*Portable Operating System Interface*, padrão IEEE para sistemas operacionais portáveis
RAID	Matriz de discos redundantes, utilizada com objetivo de prover tolerância às falhas de discos
RAM	Memória de Acesso Randomico
root kit	Conjunto de adulterações do sistema operacional que possibilita a um invasor obter acesso de super-usuário a um sistema
RPM	*RPM Package Manager*, ou gerenciador de pacotes *RPM*
RSA™	Transformada de criptografia assimétrica criada por R. Rivest, A. Shamir e L. Adleman
sandbox	Mecanismo utilizado na tecnologia *Java*™para prover isolamento entre aplicativos e a máquina que os executa
scripts	Programa em arquivo texto, interpretado durante a execução
SIGBUS	Sinal *POSIX* que é enviado a um programa quando ocorre um erro de acesso a memória
swap	Espaço de armazenamento temporário de páginas de memória em disco
swapping	Mecanismo de transferir páginas de memória para um meio externo de armazenamento, para possibilitar ao sistema utilizar mais memória que o tamanho da *RAM*
system call	Chamada de sistema, utilizada pelos programas para pedirem serviços ao sistema operacional
thread	Fluxo de execução de um programa no sistema operacional
timer	Temporizador
trojans	Cavalo de Tróia, programa que se disfarça de *software* inofensivo para auxiliar ataques ao sistema
Unix™	Padrão de sistema operacional, marca registrada do *The Open Group*
upgrade	Ato de atualizar o sistema
Windows™	Sistema Operacional desenvolvido pela Microsoft™

Resumo

Este trabalho apresenta a combinação de uma série de mecanismos de segurança baseados em criptografia com conceitos de sistemas operacionais. Esta combinação cria uma arquitetura inovadora, que aproveita características do funcionamento do sistema operacional para protegê-lo de vulnerabilidades relacionadas com a adulteração de programas e bibliotecas.

Esta arquitetura consiste em um mecanismo de geração de assinaturas digitais integradas a programas e bibliotecas, para garantir a autenticidade e a integridade dos mesmos e de cada parte do sistema operacional em execução. O sistema funciona de forma transparente para programadores, e possibilita que o *software* com assinaturas digitais seja compatível com sistemas que não tenham suporte a verificação de assinatura, simplificando o processo de distribuição.

A verificação de assinaturas é integrada com o mecanismo de paginação do microprocessador, de forma a efetuar uma verificação sob demanda somente da parte dos programas e bibliotecas que é efetivamente utilizada. Esta integração diminui o impacto sobre o desempenho e proporciona transparência para os usuários finais e administradores de sistemas.

A validação do modelo proposto foi realizada como uma extensão à *interface* binária de aplicação *Unix ELF*, implementada na plataforma *GNU/Linux*. O impacto da assinatura digital para desenvolvedores e distribuidores de sistemas operacionais sob a ótica de complexidade de uso, efeitos do sistema para usuários finais e a sobrecarga no desempenho do sistema foram verificados.

Abstract

This work discusses a series of security mechanisms based in cryptography, combining them with concepts of operating systems, in order to create an innovative architecture that uses some functionalities characteristics of operating system to protect it against vulnerabilities related with the adulteration of computer programs and libraries.

This architecture consists in a mechanism for generation of digital signatures integrated into the computer programs and libraries, in order to guarantee the software and operating system authenticity and integrity while in execution by the target machine. The system is transparent to programmers, and it is also possible that software with digital signatures can be executed on systems without support to verification of signature, what simplifies the software distribution process.

The verification of signatures is integrated with the paging mechanism of the microprocessor, making on-demand verification only in the part of the programs and libraries that is going to be effectively used, diminishing the impact on the performance and providing transparency for final users and system administrators.

The validation of the model was carried through as an extension to the application binary interface *Unix ELF*, implemented in the *GNU/Linux* platform. The impact of the digital signature for developers and distributors of operating systems under the point of view of complexity, effect for final users and overload in performance of the system was verified.

Capítulo 1

Introdução

A segurança dos Sistemas Operacionais consiste de uma combinação entre mecanismos que garantem a privacidade das informações, mecanismos que garantem a disponibilidade dos serviços e recursos gerenciados, e mecanismos que garantem a integridade do Sistema Operacional. Este trabalho tem como foco os mecanismos para garantir a integridade do Sistema Operacional.

Com a aplicação do conceito de reutilização de código e o desenvolvimento da programação voltada a objetos e componentes, os sistemas operacionais estão sendo construídos através de uma série de programas e bibliotecas, ligados em tempo de execução.

Devido à complexidade e interdependência entre esta série de *softwares*, existe uma grande dificuldade em garantir a integridade de um dado código binário presente no sistema quanto à presença de ameaças à segurança, como vírus e *trojans* [Romans and Ratliff, 2001, Thimbleby et al., 1998].

1.1 Funcionamento do sistema proposto

O sistema constitui-se em uma estrutura de confiança baseada em criptografia assimétrica, para possibilitar a detecção de modificações, envolvendo a inserção de assinaturas criptográficas nos próprios executáveis a serem protegidos, e de uma modificação no sistema operacional para verificar todo o *software* utilizado.

O sistema de verificação empregado baseia-se em assinaturas digitais aplicadas sobre os binários de programas e bibliotecas armazenados no sistema de arquivos, as quais

são inseridas dentro do padrão de binários do sistema operacional (*ELF* [TIS, 1993]), de forma a impossibilitar a separação do *software* e sua respectiva assinatura. O processo completo vai desde o momento em que é utilizada uma chave para assinar um *software* pela empresa que distribui o *software*, uma agencia certificadora, desenvolvedor credenciado, ou o próprio administrador, até o ponto em que o sistema operacional verifica o *software* antes de utiliza-lo. Este método simplifica o uso do sistema de segurança pelos usuários e administradores do sistema operacional, criando um sistema seguro e de funcionamento transparente. A inserção é proposta em uma etapa final à montagem do binário, após a compilação e durante a ligação (*linking*), entre os vários arquivos objeto que o compõem, em que é utilizada a chave privada de assinatura do desenvolvedor para assinar os *softwares* que fazem parte da imagem de um processo (código e dados).

Este método simplifica a tarefa de instalação do sistema, além de tornar transparente posteriores atualizações de versões de *software*.

A assinatura é efetuada através de criptografia assimétrica, em que a chave privada, utilizada para assinar um dado código binário, é guardada em segredo pelo desenvolvedor ou autoridade certificadora, e a chave pública é distribuída para ser empregada durante o uso do sistema operacional.

A verificação das assinaturas é feita desde a montagem da imagem em memória do processo, e durante a sua execução. A chave pública é empregada quando o núcleo (*kernel* [Kamel et al., 1998]) do sistema operacional precisa verificar a origem de um *software*. O ponto em que a verificação ocorre foi escolhido de forma a detectar adulterações no sistema mesmo que um inimigo tenha acesso direto ao sistema de arquivos ou ao *hardware* empregado para o seu armazenamento.

O uso de uma estrutura de chaves públicas possibilita a construção de uma arquitetura de certificação cruzada, em que uma chave pública pode ser assinada por outra chave e assim sucessivamente, de forma a possibilitar a criação de uma estrutura em árvore. Este tipo de estrutura pode ser utilizada para distribuir *software* escrito por um grande conjunto de indivíduos ou organizações diferentes. Por exemplo, o sistema operacional pode se basear em uma chave de uma agência certificadora, que atesta sua confiança em uma dada empresa, a qual por sua vez confia em cada um dos seus desenvolvedores.

O processo envolve a empresa fabricante ou distribuidora do sistema operacional, que deve gerenciar as assinaturas de todos os pacotes, criando uma estrutura de chaves públicas [ITU, 2000, Lee and Kim, 1999]. Esta estrutura de chaves públicas pode de-

pender de outras entidades, como uma agência certificadora, ou depender dos próprios desenvolvedores, que podem ter a autoridade delegada de assinar pacotes.

As chaves necessárias podem ser obtidas dinamicamente através de uma rede, e armazenadas localmente para usos futuros, facilitando a gerência e distribuição de chaves.

O sistema é instalado no mínimo com a chave do fabricante ou distribuidor do sistema operacional, mas o administrador pode incluir outras chaves públicas (a sua própria, de desenvolvedores de confiança, etc). O administrador ou o distribuidor podem, por sua vez, confiar no julgamento de terceiros, possivelmente desenvolvedores ou agências certificadoras para assinar pacotes para o sistema, simplificando a tarefa de administrar a distribuição de um grande número de pacotes.

Caso seja encontrada alguma característica indesejável em um *software*, é possível através da verificação da assinatura digital correspondente a uma dada pessoa/organização provar que aquele *software* foi assinado pela chave privada do par, que pertence ao fornecedor/certificador. Partindo do pressuposto de que a segurança da chave privada em questão não foi comprometida, é possível provar que a característica indesejável é de responsabilidade do fornecedor/certificador do *software*.

1.2 Combinação com outros mecanismos de segurança

Pode ser feita uma analogia interessante entre a segurança de um sistema operacional e a segurança de uma edificação. Assim como em uma edificação, a segurança de um sistema operacional depende da segurança individual de uma série de partes. É necessário em uma edificação protege-la de invasões por uma série de formas.

Uma forma de invasão de uma edificação é simplesmente entrar pela porta. Uma porta é protegida através de uma fechadura robusta, permitindo que apenas pessoas autorizadas acessem o sistema. Outra forma de invadir um edifício é utilizar uma janela, o que pode ser mais fácil que usar uma porta. Supondo que um edifício seja construído com uma porta segura, como a de um cofre de banco, e uma janela muito robusta, então talvez seja mais simples invadir o edifício quebrando uma parede.

Esta analogia simplificada mostra que a segurança de um sistema operacional, assim como a segurança de um edifício, é igual a segurança de seu elemento mais fraco. Em uma segunda analogia, uma corrente é tão forte quanto seu elo mais fraco. Mesmo com um sistema muito robusto, pode-se aplicar técnicas de engenharia social e convencer um

usuário a auxiliar o invasor. Deste forma, o invasor ganha a mesma capacidade de agir no sistema quanto a capacidade da pessoa enganada.

Estas analogias evidenciam que a segurança do sistema como um todo depende de uma série de fatores, e a proposta deste trabalho fortalece somente um dos fatores. Portanto a aplicação deste sistema não pode ser entendida como uma solução definitiva de segurança. A arquitetura proposta deve ser empregada em conjunto com outras soluções, como o uso de um sistema operacional robusto e um *hardware* com recursos de segurança confiáveis, para aumentar a dificuldade de realizar uma invasão. Aumentando esta dificuldade, é possível reduzir muito a probabilidade de uma invasão efetivamente acontecer.

1.3 Impacto sobre o desempenho do sistema

Apesar destas vantagens, a presença de um esforço de verificação criptográfico acarreta em uma perda de desempenho, que pode ser minimizada (conforme capítulo 4), mas não eliminada completamente. Outro ponto negativo da abordagem é a possibilidade de confiar em um grande número de entidades, como agências certificadoras, empresas e desenvolvedores, o que aumenta a possibilidade de um *software* falso ser assinado e utilizado para invadir o sistema.

1.4 Estrutura e objetivos

Este trabalho está organizado em 8 capítulos. O capítulo 2 consiste de uma descrição do problema e do estado da arte das pesquisas, trabalhos e mecanismos de segurança atualmente empregados e passíveis de serem utilizados para fortalecer a segurança dos sistemas operacionais, assim como uma comparação entre os vários métodos citados. No capítulo 3, é feito um levantamento das características dos algoritmos e as transformadas criptográficas adequadas para a aplicação em questão. O capítulo 4 apresenta a metodologia empregada e o ponto mais adequado em que é possível alterar um sistema operacional para o mesmo verificar a segurança dos programas e bibliotecas. O capítulo 5 trata da inserção dos dados gerados pela criptografia descrita no capítulo 3, que serão empregados na verificação do programas e bibliotecas pelo método descrito no capítulo 4.

No capítulo 6 é mostrada a metodologia proposta nos capítulos 3, 4 e 5 e o desempenho aplicado no sistema operacional (*GNU/Linux*). As conclusões são apresentadas no

capítulo 7. O capítulo 8 apresenta possíveis aplicações para esta abordagem de segurança, assim como fornece diretrizes para trabalhos futuros.

Capítulo 2

Estado da Arte

2.1 Identificação dos problemas que originaram o trabalho

Os sistemas operacionais modernos são compostos por uma série de *drivers*, programas e bibliotecas dinâmicas, que interagem para oferecer suporte para os aplicativos. A complexidade dos sistemas operacionais tende a aumentar naturalmente à medida em que são acrescentadas funcionalidades.

Várias abordagens para desenvolvimento de *software* são aplicadas para aumentar a qualidade e eficiência destes sistemas, como o uso extensivo de bibliotecas dinâmicas e componentes. Além da complexidade causada pelo sistema ser desenvolvido utilizando bibliotecas compartilhadas, os aplicativos utilizam-se de um conjunto de bibliotecas, o qual freqüentemente se confunde com o do próprio sistema operacional.

O uso de bibliotecas dinâmicas apresenta uma série de vantagens:

- Possibilita o reaproveitamento de trabalho;
- Diminui o tempo de desenvolvimento;
- Melhora a eficiência, na medida em que aumenta a velocidade de carga de aplicativos;
- Reduz o uso de memória para armazenamento e execução.

As vantagens relativas ao desempenho decorrem do fato de que cada cópia da biblioteca

dinâmica na memória tem sua própria memória de dados, mas apenas uma única cópia da parte executável compartilhada para todas as instâncias.

Em contrapartida, o uso de bibliotecas dinâmicas apresenta algumas desvantagens relativas ao gerenciamento de versões diferentes e a segurança. Para oferecer alguma funcionalidade, a biblioteca executa com os mesmos direitos que o programa que a utiliza no sistema operacional.

Ao ser executado, um aplicativo que depende de bibliotecas dinâmicas é dinamicamente ligado (*linked*) a um conjunto de bibliotecas, e este conjunto de bibliotecas faz parte da imagem em memória do processo. Portanto ao utilizar-se de uma ou mais bibliotecas dinâmicas para o desenvolvimento de um programa, está sendo estabelecido um vínculo de confiança entre os desenvolvedores do programa e os desenvolvedores das bibliotecas.

Como um sistema é composto de um conjunto de programas, é comum a situação em que uma série de programas depende de uma mesma biblioteca, especialmente se a mesma faz parte do sistema operacional. Caso um programa necessite de uma atualização de uma biblioteca do sistema, todos os outros programas que dependiam da versão antiga da biblioteca passarão a depender da nova versão, e o vínculo de confiança destes outros programas com a biblioteca anterior é quebrado.

Devido a esta situação, este tipo de sistema operacional tende a ter alguns problemas conceituais, mesmo em um ambiente seguro em que todos os envolvidos são confiáveis [Devanbu, 1999].

A situação fica mais crítica para o caso de um sistema em que existe um inimigo explorando vulnerabilidades. Existe uma série de pontos de ataque nos quais é possível inserir códigos (programas) que posteriormente serão executados com os privilégios do programa hospedeiro, ou biblioteca.

Uma forma possível de ataque é injetar um programa dentro de uma biblioteca do sistema, e convencer o usuário a fazer o *upgrade* da mesma para a versão infectada. Desta forma, uma atualização rotineira de bibliotecas pode ser utilizada para inserir falhas de segurança no sistema operacional.

Além de ataque em bibliotecas e programas, existem outros fatores que tornam um sistema vulnerável, entre estes:

- falhas em restrições para usuários;

- problemas de segurança no sistema de rede;
- falta de robustez quanto a estouros de *buffer*;
- pouca segurança do sistema de arquivos.

Uma forma de quebrar a segurança de um sistema é corromper o próprio sistema operacional, inserindo código malicioso dentro de alguma parte do mesmo, que age como hospedeiro.

No caso de vírus, *trojans* e outras ferramentas de invasão de sistemas, existem vários outros pontos além das bibliotecas que podem ser utilizados, desde *device drivers*, o próprio *kernel*, ou um programa.

Este trabalho concentra esforços em garantir que todos os componentes executáveis do sistema operacional possam ser verificados antes que os mesmos tenham acesso ao sistema. O uso de criptografia para outras formas de proteção [Ghosh and Voas, 1999, Wilhelm, 1997], assim como outras vulnerabilidades inerentes a construção do sistema estão fora do escopo do trabalho, apesar de serem citadas como alternativas complementares de segurança do sistema como um todo.

Caso exista a possibilidade de um invasor interagir diretamente com o *hardware* de armazenamento de programas, existe ainda a possibilidade de injetar código executável diretamente no sistema de arquivos, em qualquer ponto do sistema, sem que seja possível detectar posteriormente a falha de segurança, comprometendo a segurança do equipamento e possivelmente utilizar este equipamento como ponte para corromper outros.

O sistema apresentado propõe uma abordagem para garantir a confiabilidade dos componentes do sistema operacional (inclusive suas bibliotecas e *device drivers*), assim como dos programas e bibliotecas comuns. A abordagem apresenta como característica desejada a garantia de que todos os *softwares* do sistema só tenham acesso aos recursos de execução da máquina se o *software* não foi adulterado. Observe que existem formas de injetar dinamicamente código para execução, como por exemplo explorando *overflows* de *buffers* ou alterando o segmento de memória com a pilha de chamadas de função. Este tipo de problema constitui uma falha de segurança do próprio sistema operacional, às vezes colocada intencionalmente por algum desenvolvedor mal intencionado. Neste caso o sistema será considerado íntegro, devido à sua autenticidade, mas não será necessariamente seguro.

2.2 Descrição dos trabalhos existentes

Os sistemas operacionais utilizam uma série de mecanismos de segurança, em diversos níveis, para proteger o sistema e os programas de possíveis problemas, tanto causados intencionalmente quanto acidentalmente. A segurança de um sistema computacional é comparável com a segurança de um edifício, no ponto de vista que todos os pontos de ataque devem ser protegidos. Um sistema de segurança de um edifício não pode ser considerado eficiente mesmo se proteger as portas com a segurança de um cofre de banco e deixar uma única janela aberta. Algum inimigo, explorando vulnerabilidades, pode utilizar esta "janela", conseguir um primeiro acesso ao sistema, e continuar explorando o sistema até conseguir o controle total do equipamento, o que poderia ser comparado com uma chave mestra (no caso computacional, um *root kit* [Romans and Ratliff, 2001]). Desta forma, um sistema deve ser protegido em todos os pontos de entrada, e mesmo dentro do sistema os recursos do sistema (na comparação com o edifício, as salas) devem ser protegidos um do outro, de forma a proteger abusos por usuários mal intencionados.

Estes mecanismos de proteção envolvem vários recursos de um sistema informatizado, desde os mecanismos providos pelo *hardware*, passando pelos mecanismos do sistema operacional até o controle efetuado pela linguagem de programação/ambiente de execução, chegando aos mecanismos providos pelos próprios programas ou bibliotecas.

2.3 Mecanismos de segurança do microprocessador

O microprocessador disponibiliza uma série de funcionalidades que podem ser utilizadas para a construção de um sistema operacional seguro. Devido ao fato do microprocessador ser o componente responsável pela maioria das ações dentro de um microcomputador, o controle da segurança da memória, assim como de outros dispositivos periféricos são implementados através dos mecanismos de segurança do microprocessador. Os mecanismos de segurança envolvem um conjunto de conceitos, que quando aplicados em conjunto garantem que o *software* do sistema operacional possa controlar, gerenciar e proteger todos os outros *softwares* do sistema.

2.3.1 Controle de níveis de privilégios

Os níveis de privilégios constituem um mecanismo utilizado pelo microprocessador para criar um ambiente controlado para a execução de programas não confiáveis. Os níveis de privilégio atuam com a função mais básica dentre todos os mecanismos de segurança, que é a de proteger o sistema do próprio microprocessador quando estiver executando programas não confiáveis. A idéia de proteger o microprocessador dele mesmo é implementada através de uma definição de níveis de privilégio, em que existem no mínimo dois níveis [Intel, 2003a, cap. 6 pag. 8], sendo que o de menor privilégio (usado para a execução de programas potencialmente perigosos) será chamado de modo usuário, e o mais privilegiado, que será chamado de modo supervisor (que permite a realização de algumas tarefas potencialmente perigosas mas necessárias para o controle do Sistema Operacional). Alguns microprocessadores implementam mais níveis de segurança, permitindo que níveis intermediários entre usuário e supervisor possam ter acesso a algumas funções que podem comprometer a segurança do sistema como um todo.

O modo supervisor é o modo no qual o microprocessador permite que um programa em execução possa executar instruções que tem o potencial de congelar o sistema, configurar, habilitar e desabilitar interrupções (mecanismo utilizado para desviar o fluxo de execução do microprocessador devido a um evento externo, como *hardware*, ou devido a uma situação gerada pelo próprio *software* que precisa de tratamento, conforme será descrito posteriormente neste trabalho), alterar os outros mecanismos de proteção, programar as estruturas de dados utilizadas para controlar a memória assim como as outras tarefas de controle do sistema operacional.

O modo usuário é ativado no exato momento em que o sistema operacional está na iminência de executar um código não seguro. A ativação é feita através de uma instrução que chaveia o microprocessador, usualmente alterando um registro de controle do mesmo, o que constitui em uma mudança do nível de privilégio. Após a mudança no nível, o próprio microprocessador faz uma série de checagens de segurança a cada instrução executada.

O sistema operacional deve registrar uma rotina para que o microprocessador desvie o fluxo de execução para a mesma, caso o microprocessador detecte que algum código está tentando executar uma instrução incompatível com o nível de privilégio atual. Imediatamente antes da execução de cada instrução, o microprocessador verifica se a execução é compatível com o nível de segurança atual (figura 2.1).

Se a instrução for autorizada, ela é executada normalmente. Caso a instrução viole o

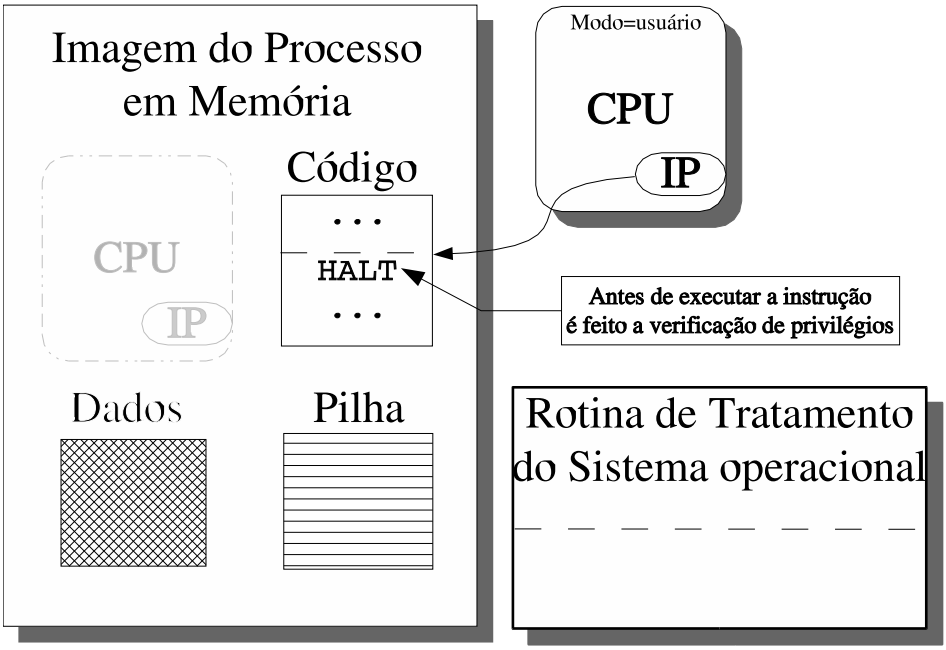

Figura 2.1: Verificação do nível de segurança

nível de privilégio atual, o microprocessador dispara um mecanismo chamado de exceção. O disparo de uma exceção [Intel, 2003b, cap. 1 pag. 5][Intel, 2003c, cap. 5 pag. 6] consiste na interrupção do fluxo de execução do microprocessador, e o desvio do fluxo de execução para a rotina previamente registrada pelo sistema operacional. O desvio do fluxo de execução devido a uma exceção automaticamente salva o estado do processador no momento da exceção (figura 2.2), por *hardware* ou pela rotina de atendimento de exceções.

Como o microprocessador não chega a executar a instrução, é possível inclusive que o sistema operacional continue a execução do programa como se a exceção não tivesse acontecido [Intel, 2003c, cap. 5 pag. 6] (figura 2.3) Este mecanismo é utilizado principalmente no controle da memória.

Este desvio do fluxo de execução pode e é utilizado para a criação de um sistema preemptivo, bastando para isto utilizar um *timer* por *hardware* que desvia o fluxo para a rotina apropriada, executando um escalonador de tarefas (o escalonador pode ser disparado por outras interrupções, ou até por uma exceção que não seja disparada por motivos de segurança, como requisição de um acesso de I/O exclusivo do modo supervisor).

Através de níveis de prioridade e do desvio do fluxo de execução (via rotinas de tra-

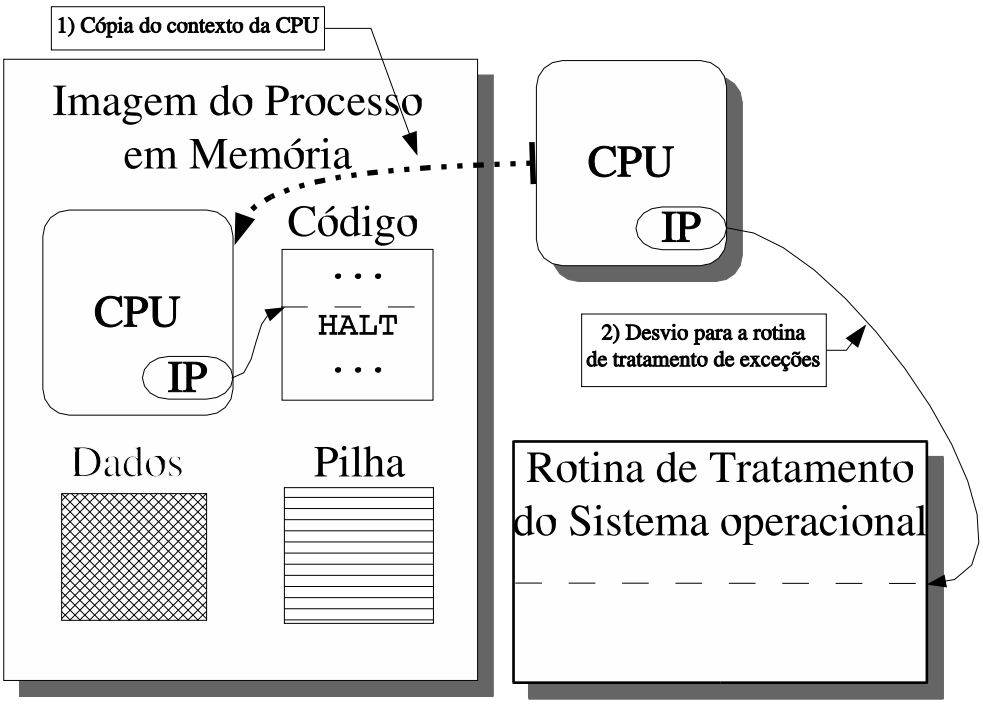

Figura 2.2: Desvio do fluxo de execução

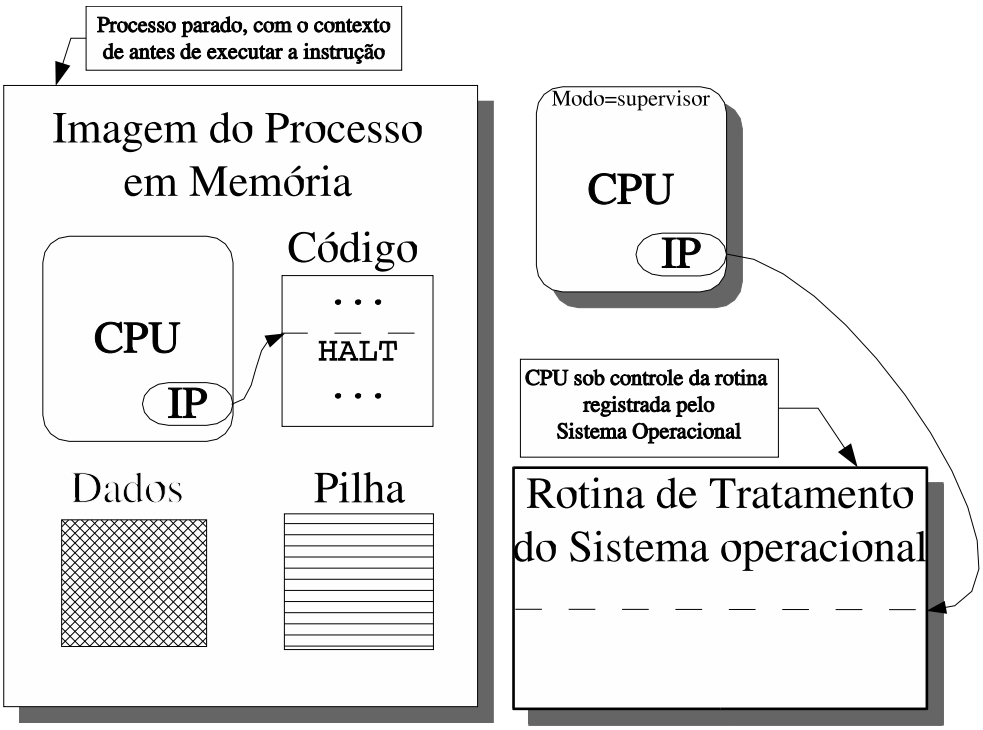

Figura 2.3: Estado após a exceção

tamento de exceções), é possível que o microprocessador execute seguramente código não confiável, e o bloqueio imposto pelo microprocessador protegendo a si próprio garante a conformidade com o padrão de segurança.

O mecanismo de proteção pode ser inclusive utilizado junto com o mecanismo de pré-decodificação de instruções. Esta lógica adicional permite que a segurança não cause nenhum impacto em termos de desempenho. A verificação de segurança das próximas instruções pode ser executada em paralelo com a execução da instrução atual.

2.3.2 Controle da memória

O controle da memória é crítico para o desenvolvimento de sistemas operacionais seguros. O sistema operacional armazena em memória, além dos dados a serem processados, as seguintes informações (figura 2.4):

- estruturas de controle de memória;
- estruturas de controle de interrupções;
- privilégios de *I/O*;
- estruturas de pilhas.

Nas estruturas de pilhas são armazenadas as chamadas de funções, endereços de retorno e parâmetros de chamadas. Possibilitando o acesso à pilha, um programa pode alterar a mesma para corromper o sistema, por exemplo através de uma escalada de privilégios. Observe que na arquitetura *Harvard*, utilizada em processadores digitais de sinais, o processador tem mais de um barramento, podendo acessar os dados e código em memórias distintas, e portanto possivelmente fisicamente isoladas.

Devido ao fato de que o código que executa em modo supervisor guarda as suas informações de controle na memória física do sistema, o mesmo utiliza a memória para organizar e controlar a segurança, e confia que estes estarão protegidos após a execução de uma tarefa em modo usuário. O mecanismo de controle de memória serve justamente para garantir uma proteção por *hardware* e isolamento de regiões de memória para cada um dos contextos de execução do processador (*threads*, processos, sistema operacional, etc).

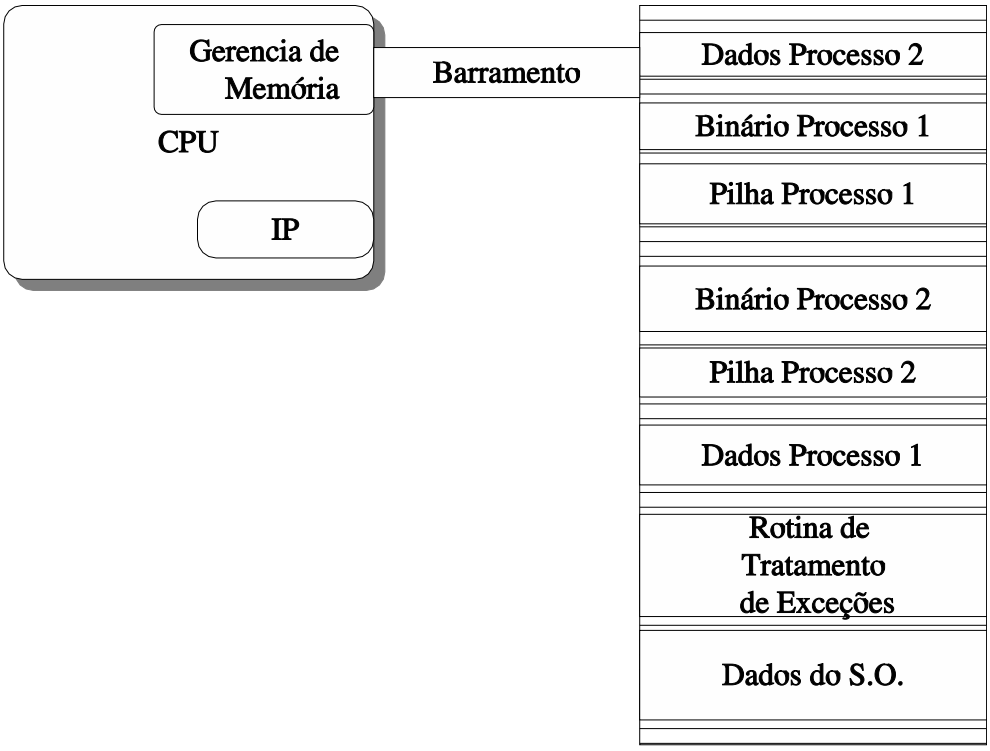

Figura 2.4: Distribuição da memória do sistema

Existem dois tipos principais de proteção de memória, que podem ser utilizados pelo sistema operacional para realizar o isolamento. O sistema operacional pode utilizar os dois simultaneamente, inclusive, e o resultado é a conjunção da proteção dos dois mecanismos.

2.3.3 Proteção utilizando segmentação

A segmentação de memória é um mecanismo desenvolvido especialmente para a proteção, tanto do ponto de vista da segurança quanto para a checagem de tipo (impossibilitando, por exemplo, a execução de um segmento de dados).

A segmentação permite definir o espaço de endereços que um segmento utiliza, seu tamanho, além da classificação das operações às quais este segmento pode ser utilizado. Esta classificação pode envolver:

- O tipo do segmento, podendo ser código (instruções) ou dados (incluindo a pilha);
- Em qual nível de privilégio o segmento se enquadra, podendo pertencer ao modo supervisor, ao modo usuário ou algum modo intermediário;

- Se o mesmo pode ser submetido às operações de leitura, escrita, ou ainda somente execução (somente pode ser executado, não pode ser nem lido pelo programa, no caso particular do segmento de programa [Intel, 2003c, cap. 4 pag. 13]).

Esta classificação permite que o microprocessador faça uma série de checagens de segurança, além dos limites de acesso. A utilização de segmentos possibilita controlar o uso incorreto da memória pelos programas, protegendo do ataque em que um programa acessa as estruturas de dados do sistema operacional para adulterar as mesmas e comprometer a segurança (figura 2.5).

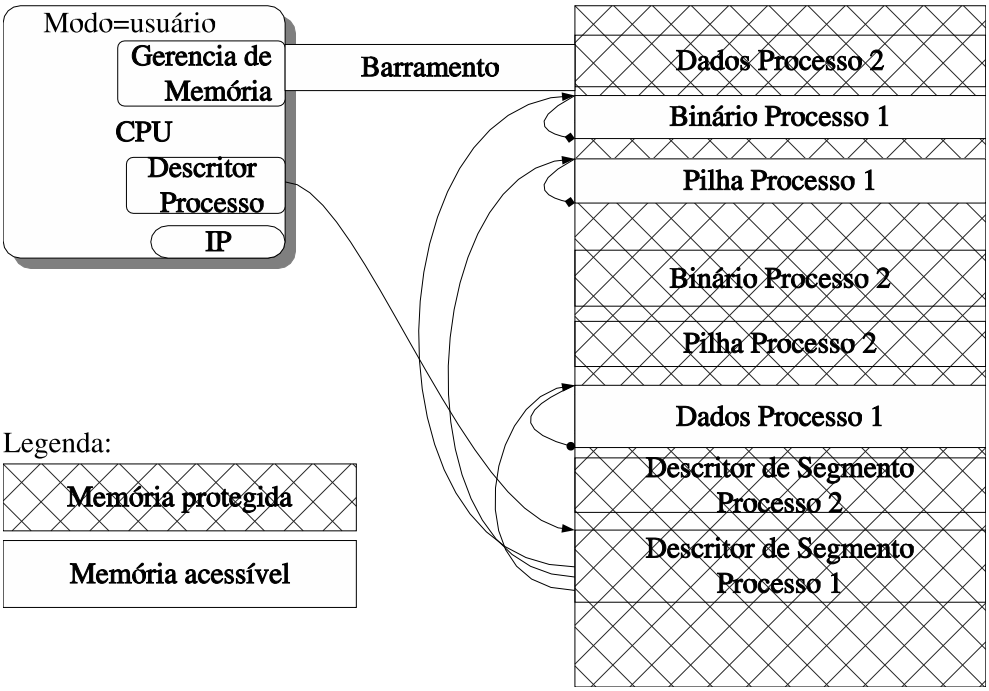

Figura 2.5: Exemplo de processador em modo protegido, utilizando segmentação

A segmentação é tipicamente feita por uma parte do microprocessador chamada de unidade de gerência de memória. Este circuito verifica os limites definidos para o segmento e qualquer tentativa de acesso não autorizado interrompe o fluxo de execução antes do acesso, desviando o fluxo para uma rotina de tratamento que deve ser registrada pelo sistema operacional. Como o fluxo é desviado antes da execução, é possível que o microprocessador salve o estado interno dos registradores (tanto via uma instrução ou a rotina de tratamento de interrupção do sistema operacional efetua o salvamento do contexto). Com o estado do registrador salvo, é possível que o sistema operacional continue a execução do programa, desde que a condição que causou a interrupção seja resolvida.

Com a garantia do desvio do fluxo de execução para uma rotina específica do sistema operacional, é possível dividir a memória entre os processos, garantindo que nenhum processo exceto o próprio núcleo do sistema operacional possa acessar uma área de memória de outro processo.

Como a segmentação é dependente do projeto do microprocessador, tipicamente sistemas desenvolvidos para serem portáveis utilizam a segurança do próprio mecanismo de paginação (descrito a seguir), para simplificar o processo de adaptação para diferentes arquiteturas.

2.3.4 Proteção utilizando paginação

A paginação de memória foi projetada para otimizar a utilização da memória física, através de um processo conhecido como *swapping*, que consiste em colocar partes da memória em uma unidade de armazenamento externa (conseqüentemente mais lenta), liberando espaço da memória principal. A paginação funciona quebrando a memória em blocos de tamanho fixo (tipicamente 4Kbytes), e utilizando um algoritmo de decisão para transferir as páginas de memória menos utilizadas para memórias auxiliares, provavelmente mais lentas. O algoritmo pode considerar a último data de acesso da página, quantidade de memória livre no sistema além da prioridade do processo (para evitar que processos que precisam responder rapidamente dependam de uma memória lenta) para transferir as páginas menos utilizadas para fora da memória principal.

A paginação permite aumentar o tamanho virtual de memória que um processo enxerga, porque o espaço de endereços do processo não precisa necessariamente residir na memória principal(figura 2.6).

Caso o processo precise de uma página que não está em memória, o processador realiza os seguintes passos:

- desvia o fluxo de execução, da mesma forma que quando o processo executa uma violação ao seu nível de privilégio corrente, ou espaço de memória, para uma rotina específica do sistema operacional;
- o desvio automaticamente salva o contexto no ponto exato em que o processo foi interrompido porque a página não existia em memória;
- a rotina de tratamento de *page faults* procura a página de memória em outra memória, possivelmente em disco;

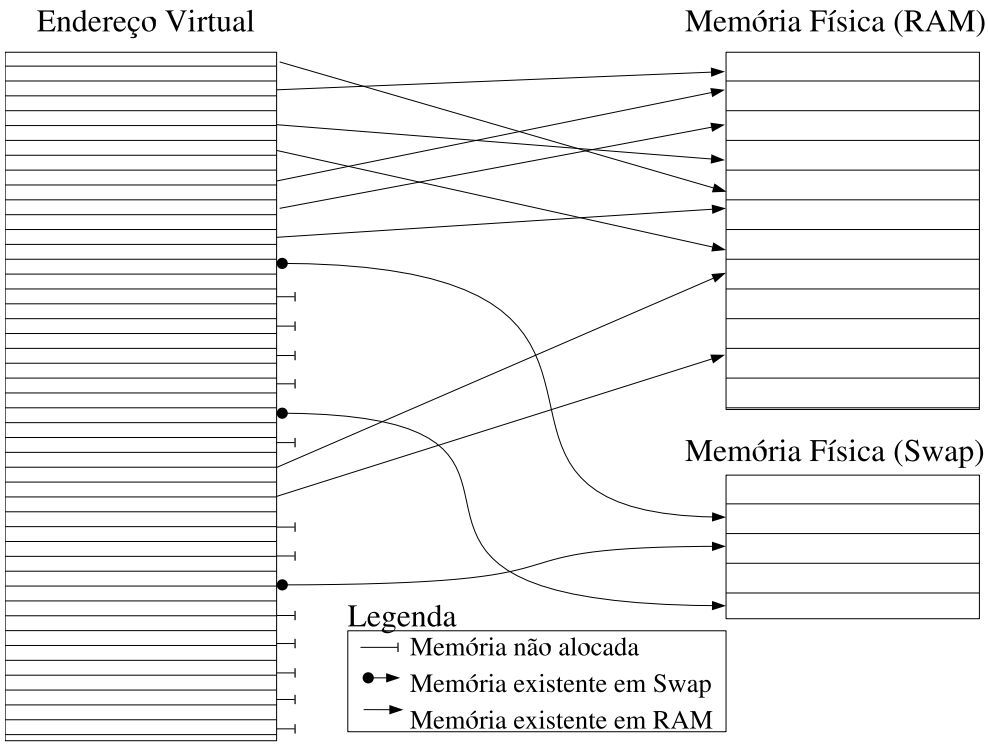

Figura 2.6: Exemplo de memória paginada

- o microprocessador carrega a página de memória para a memória principal;
- o microprocessador carrega o estado do programa (contexto) e o programa continua a execução do ponto exato em que o processo parou.

Se a página de memória não existia (por exemplo, havia sido alocada pelo programa, mas o sistema operacional tinha mapeado um endereço virtual sem que exista uma página de memória real associada), o sistema operacional pode alocar a página no exato momento em que ela é utilizada, de forma a otimizar o uso de memória. A página de memória física só é alocada quando o processo efetivamente a usa.

O mecanismo de paginação possibilita ainda uma otimização do uso da memória [Intel, 2003c, cap. 3], se combinado com o mecanismo de *cache* e *buffers* de arquivo. Desta forma, o sistema operacional reserva uma parte da memória para preservar as páginas de memória mais utilizadas, mesmo que não exista nenhum processo utilizando a mesma no momento. Os acessos posteriores aos mesmos dados não necessitam da transferência de dados de um meio de armazenamento mais lento, melhorando o desempenho do sistema. Isto é especialmente interessante no caso de bibliotecas do sistema operacional que são utilizadas por um grande número de programas. Estas páginas podem ser compartilhadas

entre vários programas [Intel, 2003c, cap. 3 pag. 23] (figura 2.7), reduzindo o gasto de memória porque vários programas podem utilizar a mesma página.

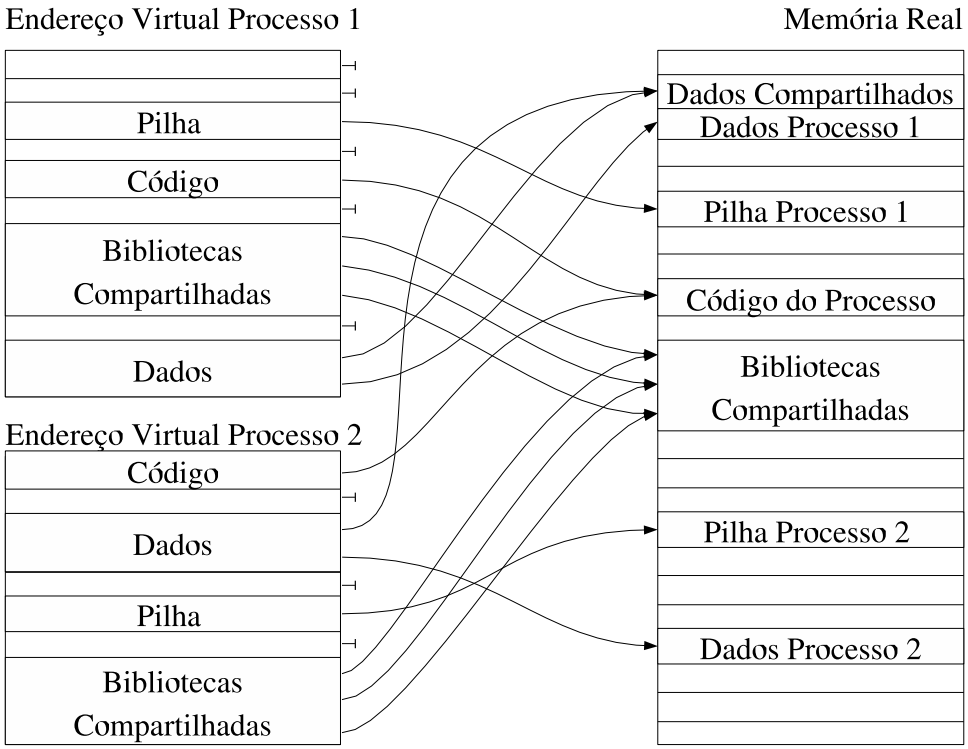

Figura 2.7: Exemplo de dois endereços virtuais que utilizam um mesmo endereço físico

Este mecanismo é aplicado no caso de programas que utilizam várias *threads* (vários fluxos de execução em um mesmo programa), e também na chamada de sistema *fork* do padrão *POSIX* (*Unix)*.

Caso um programa utilize esta rotina, logicamente acontece uma cópia de toda a imagem do processo, que é uma operação dispendiosa. Com o uso inteligente da paginação é possível copiar logicamente a imagem inteira do processo, sem copiar as páginas. Desta forma, existem vários processos enxergando espaços lógicos diferentes, que apontam para a mesma página em memória. Esta cópia envolve as páginas de dados, incluindo a pilha. As páginas compartilhadas podem ser bloqueadas para escrita, de forma a desviar o fluxo de execução do processo se o mesmo tenta escrever nas páginas compartilhadas. Após o desvio, o sistema operacional pode duplicar a página no exato momento em que as mesmas deveriam se tornar logicamente páginas com conteúdo diferente, duplicando a página compartilhada, e retornado o fluxo de execução para o processo que então continua a execução alterando a página. A operação de cópia somente ocorre se a mesma é realmente

necessária, e todas as outras páginas que contém os mesmos dados ficam compartilhadas, economizando memória. Este mecanismo chama-se *copy-on-write* [Intel, 2003c, cap. 4 pag. 31 a 34].

A sobreposição de páginas lógicas em uma mesma página física é utilizada para compartilhar memória entre vários processos distintos.

2.4 Assinaturas Digitais

O uso de assinaturas digitais é uma forma de garantir a autenticidade de um programa pelo uso de criptografia. O uso de assinaturas é aplicado em uma série de sistemas, tipicamente naqueles que distribuem código via redes não seguras como a Internet.

2.4.1 Pacotes Java™(arquivos *.jar*)

A tecnologia Java™utiliza, combinado com uma série de outros mecanismos, este tipo de abordagem para garantir a integridade de pacotes *jar* [Fritzinger and Mueller, 1996, Shin et al., 2001]. O objetivo desta abordagem é resolver o clássico problema do *Man-in-the-middle* (homem-no-meio), que está relacionado com uma comunicação de dados genérica em que algum elemento se coloca entre as duas partes em comunicação, e age como uma ponte para os dois envolvidos nesta comunicação, tendo acesso a todos os dados trafegados, tanto para ler como adulterar. No caso específico deste ataque considerando o *download* de um programa, um inimigo pode alterar os dados a medida que o cliente transfere o *download* do fornecedor legítimo, ou ainda se disfarçar para parecer como o servidor legítimo, via *DNS spoofing* (corromper o sistema de resolução de nomes para que o domínio do fornecedor aponte para um servidor controlado pelo atacante) ou invadindo o *site* do fornecedor legítimo (ou algum servidor espelho) e corrompendo os dados direto no servidor do fornecedor do pacote de *software*.

Os pacotes *java*™são comprimidos e assinados, e todas as partes do aplicativo ou *applet* são assinados. Desta forma, além do programa ser confiável, dificulta-se o ataque em que o invasor altera arquivos não executáveis (como arquivos de configuração) para explorar falhas de codificação do programa.

Esta confiabilidade adicional da tecnologia Java™para programas e *applets* pode ser utilizada ainda para desabilitar outros mecanismos de proteção (*sandbox*, por exemplo)

desenvolvidos especificamente para aplicativos completamente não confiáveis. Desta forma pode-se garantir direitos de acesso ao sistema operacional da máquina caso exista uma relação de confiança verificável entre o usuário e o fornecedor do *software*.

2.4.2 A tecnologia *authenticode*™(Microsoft)

A abordagem de desenvolvimento de controles *ActiveX*™(tipo especial de componente COM™- *Component Object Model*) utiliza-se de código nativo em linguagem de máquina específica para a arquitetura do cliente. Estes códigos interagem diretamente com o sistema operacional, com todos os direitos do usuário que executou o componente [Bellovin et al., 2002], e quase todos os mecanismos de segurança disponíveis baseiam-se no navegador *Internet Explorer*™. Caso o componente seja executado a partir de outro aplicativo, o mesmo é responsável por prover os mecanismos de segurança.

Para garantir a confiabilidade, integridade e autenticidade dos componentes, foi desenvolvida a tecnologia *Authenticode*™[Microsoft, 2003] para programas e pacotes, que considerando apenas o aspecto de assinatura digital, e não outros mecanismos adicionais de segurança como *sandboxing* [Bellovin et al., 2002, Fritzinger and Mueller, 1996, Malkhi and Reiter, 2000], se assemelha com a assinatura de pacotes Java™, protegendo dos mesmos tipos de ameaças de forma semelhante.

A tecnologia *ActiveX*™utiliza a assinatura de pacotes no sistema *Authenticode*™, garantindo desta forma a autenticidade e integridade dos componentes, verificando a assinatura a partir de um certificado digital, e instalando o componente localmente. Não é realizada uma verificação de assinatura para componentes já instalados.

Existem duas áreas para a análise da segurança de controles *ActiveX*™, os riscos inerentes à instalação de um componente e os riscos relativos à execução de componentes, tanto por possíveis falhas nos mesmos, quanto pela exploração do *Windows*™*Scripting Host*, utilizando-se o componente apenas como uma ponte para acessar a interface do mecanismo de *scripts*.

Os problemas relativos à instalação do componente envolvem o fato das assinaturas dos componentes persistirem, e portanto o vínculo de confiança na cadeia de assinaturas continua presente. Desta forma, se um usuário do sistema confia em uma organização e o invasor tem um componente assinado que pode ser utilizado como ferramenta para comprometer a segurança do sistema, o invasor pode instalar o componente na máquina alvo devido à confiança na organização como um todo, estabelecida anteriormente. Este

tipo de vulnerabilidade pode ser combatida através de certificados de revogação em uma estrutura de chaves públicas [ITU, 2000, Lee and Kim, 1999], em que ao ser descoberto um problema em um componente, o mesmo teria sua assinatura revogada, e o certificado de revogação propagaria através da cadeia de chaves da raiz até o componente. Atente-se que, se a verificação de assinatura de um componente falha, o mesmo perde a validade no sistema, e os outros programas que dependiam do componente podem ser invalidados, gerando outro problema, que é escolher a ação corretiva ao detectar-se uma vulnerabilidade.

A forma mais utilizada de impedir a execução de componentes, caso seja descoberto uma falha de segurança, é o *Kill Bit*, que é uma marca no registro do *Windows*™que impede que o componente seja executado e que tem prioridade sobre todas as definições de zonas de segurança do *Internet Explorer*™.

Uma característica do sistema que pode ser explorada consiste no fato de que por padrão um componente precisa de autorização de apenas um usuário registrado na máquina para ser instalado, e o componente fica disponível para todos os usuários. Observe que o *upgrade* de um componente básico do sistema pode ser feito por qualquer usuário, e afeta todos os usuários, que inconscientemente dependem do mesmo. Ou seja, um usuário pode quebrar o vínculo de confiança dos outros usuários.

Pode-se utilizar qualquer usuário para proceder a escalada de privilégios, possivelmente conquistando o privilégio máximo do sistema. Escalada de privilégios é a tarefa que um invasor tem de, ao atacar uma máquina e conseguir os direitos de um usuário, conseguir direitos de outros usuários mais poderosos. Ao chegar ao superusuário, tipicamente o invasor instala um *root kit* [Zovi, 2001], que é um sistema camuflado que permite ao invasor controlar totalmente a máquina, possivelmente de forma remota, podendo posteriormente utilizar a máquina como ponte para invadir outras máquinas na rede.

Existe a possibilidade de limitar para o administrador o poder de aprovar o uso dos componentes para um dado sistema, através da configuração "*Administrator Approved*".

O modelo de desenvolvimento adotado para o *ActiveX*™baseia-se completamente no julgamento humano para a proteção do sistema, e proporciona uma série de funcionalidades para uso do desenvolvedor, que podem ser utilizadas para comprometer a segurança do sistema.

2.4.3 Assinatura de pacotes *RPM*

O sistema de gerenciamento de pacotes *RPM* [Bailey, 1997, Foster-Johnson, 2003], desenvolvido pela *RedHat* para a distribuição de pacotes do sistema operacional *Linux* utiliza assinaturas *OpenPGP* para a verificação de integridade (o sistema *RPM* prevê a extensibilidade para outros tipos de assinaturas). Como o sistema *OpenPGP* [Copeland et al., 1999, Garfinkel, 1994] é um padrão genérico para assinatura e criptografia de mensagens e arquivos, esta abordagem utiliza a estrutura do *OpenPGP* para escolha de uma série de padrões de criptografia simétrica e assimétrica, gerenciamento de chaves e infraestrutura de chaves públicas, flexibilizando o sistema.

A instalação e *upgrade* de pacotes é atividade exclusiva do superusuário, o qual baseia-se em julgamento humano para determinar a segurança do código a ser incorporado no sistema.

A conferência de assinaturas é executada somente durante a instalação do pacote, portanto a segurança do sistema garante a integridade até o momento da instalação, mas não oferece nenhum mecanismo para verificar a integridade do sistema após a instalação do pacote.

2.4.4 Características comuns ao uso de assinaturas digitais

O uso de assinaturas digitais apresenta uma série de vantagens comuns às abordagens citadas anteriormente, com algumas particularidades de acordo com a solução específica, conforme descrito a seguir:

- Existe a possibilidade de provar a integridade dos dados, possibilitando a detecção de adulterações intencionais ou acidentais (que podem ser igualmente perigosas) nos dados.

- A prova de autenticidade dos dados, que possibilita que o usuário final possa comprovar que o pacote ou programa instalado foi realmente produzido pelo fornecedor correto, e possibilita que, caso seja encontrado um problema de segurança, pode-se provar que este problema foi originário do fabricante.

- Existe a possibilidade do administrador realizar uma auditoria no sistema e saber que foi feita uma tentativa de ataque, além de evitar a mesma [Williams, 2002].

Pode-se utilizar este mecanismo para criar máquinas isca, conhecidas como *Honey Pots*, para detectar e estudar invasões e vírus.

- A facilidade e transparência durante o uso inibe que os próprios usuários tentem burlar o sistema [Rosu and Segerlind, 1999, Fritzinger and Mueller, 1996].

Em contrapartida, existe uma série de problemas e falhas inerentes ao modelo, como:

- A segurança do modelo de assinaturas não garante a benevolência do código: esta característica baseia-se completamente no julgamento humano da entidade que assina os programas, ou em avaliações externas ao sistema quanto à segurança dos códigos assinados [Weber, 2000] [Devanbu and Stubblebine, 1997] [Ghosh and Voas, 1999] [Bellovin et al., 2002].

- Para o caso de hierarquias de assinaturas, é violado o princípio de confiança mínima [Rosu and Segerlind, 1999] (ao confiar em uma agência certificadora, o sistema estará automaticamente confiando em uma série de outros terceiros, aumentando a probabilidade de que um do grupo tenha sido comprometido).

- A eficiência da checagem de assinatura deve ser suficiente para não causar uma perda de desempenho que motive o uso de códigos não assinados [Williams, 2002, Fritzinger and Mueller, 1996, Rosu and Segerlind, 1999].

- *Bugs* nos códigos assinados (como *buffer overflows* [MacDonald, 1998] e abuso de pilha de chamadas) podem ser explorados para comprometer a segurança do sistema.

- Programas que executam *scripts* podem ser utilizados para quebrar a segurança do sistema [Weber, 2000] (uma possível solução é assinar os próprios *scripts* [Williams, 2002]).

- Após o código passar pela verificação da assinatura, o código pode ser adulterado livremente. O ponto de adulteração depende da abordagem utilizada, desde alteração de arquivos, caso a verificação seja feita apenas na instalação dos programas, ou mais internamente ao sistema operacional [MacDonald, 1998], alterando a memória [Zovi, 2001, Romans and Ratliff, 2001] ou ainda durante a paginação para disco ou rede, por exemplo.

- Os sistemas operacionais podem não apresentar um grau de segurança suficiente para oferecer suporte confiável ao sistema [Weber, 2000].

2.5 Verificação de integridade via *hash*

Uma forma utilizada para garantir integridade de dados é o uso de funções unidirecionais, ou *hash functions*. Uma função *hash* [Menezes et al., 1996] é uma transformada matemática que a partir de uma entrada de tamanho variável produz como saída um número de tamanho fixo, de forma similar a um *checksum* ou um CRC. A vantagem de uma função *hash* consiste no fato de que é computacionalmente impossível (ou seja, o tempo de processamento gasto na tarefa inviabiliza a empreitada) descobrir uma entrada com o objetivo de conseguir um dado número na saída da função. Desta forma, a função unidirecional protege contra corrupções acidentais dos dados, de forma análoga ao *checksum* (ou o CRC, que oferece um nível maior de proteção), e ainda fornece proteção contra uma adulteração dos dados.

As aplicações e sistemas baseados em *hash functions* tipicamente baseiam-se na confiança que o sistema está seguro em um dado momento, e aplicam o *hash* para todos os *softwares* a serem monitorados, criando um banco de dados relacionando cada *software* com um número de *hash* único, para posterior verificação.

A forma de verificação deste *hash* pode ocorrer de várias maneiras, dependendo do propósito do sistema e da abordagem utilizada.

2.5.1 Verificações periódicas de integridade

Em sistemas baseados em verificação periódica, todo o sistema é verificado em intervalos periódicos, e tem seu *hash* comparado com a base de dados gerada anteriormente.

Nesta abordagem, aplicada na ferramenta *tripwire* [Kim, 2002, Tripwire, 2002], todos os arquivos são verificados, independente da taxa de uso do *software*. Sistemas deste tipo permitem que seja feita a verificação de arquivos não executáveis, como arquivos de configuração, assim como a verificação de *scripts*. Devido ao fato de que a verificação não tem relação alguma com a utilização de um *software* do sistema, a partir do momento em que o sistema é considerado inválido é impossível precisar se o possível código malicioso teve acesso aos recursos do sistema e foi realizada uma invasão, e qual foi a extensão dos danos, se existiu algum. As únicas informações disponíveis são que o sistema estava intacto até a data da última verificação, e qual parte do sistema foi violada.

Um agravante deste tipo de abordagem é a possibilidade de que o código malicioso adultere o sistema para ocultar a sua própria existência, de forma a mascarar uma invasão

e possivelmente permanecer não detectado. Uma forma de mascarar a invasão pode ser implementada pelo uso de *root kits* [Zovi, 2001], em que é inserido dentro do núcleo do sistema operacional um código que executa no modo supervisor, que é o modo utilizado pelo sistema operacional para executar todas as tarefas críticas para a segurança do sistema. Neste caso, o invasor pode mascarar as *system calls* do sistema operacional, que são a interface entre os programas e o sistema, e esconder a presença do código malicioso, inclusive mascarando uma leitura de arquivo para que o programa que tente verificar o sistema leia o programa não adulterado, e a invasão permaneça não detectada. Este tipo de ameaça inviabiliza o uso de uma verificação periódica, porque no intervalo entre duas verificações é possível que o *root kit* seja instalado e esconda a sua presença do próprio sistema. O sistema infectado pode não ser capaz de verificar a infecção, mas um outro sistema poderia verificar todo o sistema vítima em uma análise forense.

2.5.2 Verificação de integridade em tempo de execução

A idéia central da verificação em tempo de execução é a checagem de integridade imediatamente antes do uso de qualquer parte do sistema operacional, biblioteca ou aplicativo.

Este tipo de abordagem implica em alterações no próprio sistema operacional, inserindo a verificação do arquivo pouco antes do carregamento do *software* protegido.

A vantagem deste tipo de abordagem sobre a anterior consiste no fato de que é garantido que um *software* não foi adulterado no intervalo de tempo entre a verificação e a execução do programa.

Existem duas formas diferentes de executar esta verificação:

1. Efetuar uma verificação por *hash* de *softwares*;

2. Utilizar *hash* e criptografia assimétrica, compondo uma assinatura digital.

A primeira forma de efetuar a verificação [Williams, 2002], consiste em efetuar um *hash* de cada arquivo do sistema operacional, e armazenar o conjunto em uma base. Toda execução posterior é acompanhada de uma transformada *hash* dos dados e de uma comparação do resultado com a base de dados de *hash* do sistema operacional, e o mesmo pode decidir se permite ou não a execução do *software* com base no resultado da comparação, ou se envia algum tipo de alerta.

Drivers do próprio sistema operacional e *scripts* podem ser protegidos, e deve-se confiar apenas em um dispositivo para carregar o *kernel*, e posteriormente todos os módulos podem ser conferidos via *hash*. Existe a necessidade de que o banco de dados de assinaturas seja protegido contra adulterações.

A segunda abordagem consistem em utilizar assinaturas digitais aplicadas sobre os arquivos, evitando o uso da base de *hash*, simplificando a distribuição e manutenção do sistema.

Estudos diferentes propõem o uso de assinaturas digitais de duas formas:

1. Inserir a assinatura digital dentro do próprio arquivo;
2. Alterar o arquivo criando um pacote assinado;
3. Guardar a assinatura em uma base separada do *software*.

A primeira abordagem [Arbaugh et al., 2003, Apvrille et al., 2004] consiste em criar uma assinatura, embutida dentro do formato do arquivo que contém o *software*, que é utilizada antes de toda execução (figura 2.8). Observe que o estudo propõe a cooperação

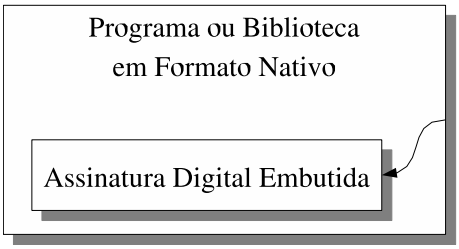

Figura 2.8: Assinatura embutida no binário

entre o *linker* dinâmico e o sistema operacional para possibilitar a verificação de bibliotecas. Como conseqüência, *linker* dinâmico passa a ser um ponto para exploração de vulnerabilidades do sistema de segurança.

A proposta foi implementada, provando a possibilidade de integração com o sistema operacional, e foram realizados testes de desempenho. Nos testes de desempenho a sobrecarga chegou a 96% de acréscimo no tempo de execução. O estudo propõe o uso de um mecanismo de *cache* para evitar verificações repetidas, reduzindo a perda de velocidade para execuções seguidas de um mesmo *software*.

Todos os testes realizados no estudo são baseados em executáveis de pequeno a médio porte, portanto seria interessante comparar o peso em processamento da verificação de

um executável grande, baseado em um grande número de bibliotecas, como é o caso dos programas mais utilizados em um sistema *desktop* padrão.

A segunda abordagem para inserção de assinaturas consiste no empacotamento dos *softwares* em um arquivo assinado [Catuogno and Visconti, 2002] (figura 2.9).

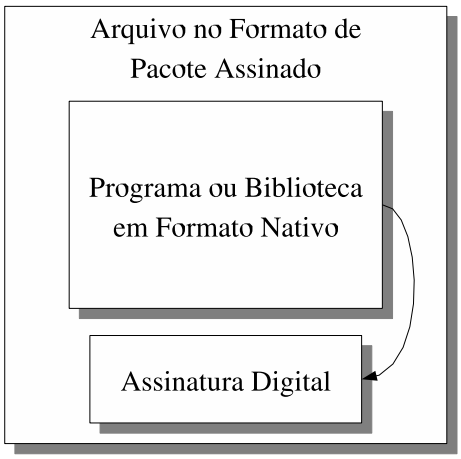

Figura 2.9: Construção de um arquivo contendo o binário e a assinatura

O funcionamento do sistema é similar ao primeiro sistema, com a vantagem de simplificar a aplicação para vários padrões diferentes de binários.

Esta vantagem é contra-balanceada por uma série de desvantagens, como a perda de compatibilidade com versões anteriores do sistema operacional, e a impossibilidade de usar *softwares* assinados em um sistema operacional sem assinaturas.

A terceira abordagem consiste em uma assinatura separada do *software*, armazenada em uma base de assinaturas (figura 2.10) [Borchardt et al., 2003, Borchardt and Maziero, 2001a, Borchardt and Maziero, 2001b, Patil et al., 2004]. Esta arquitetura tem como vantagem o fato de permitir a verificação de assinatura de qualquer tipo de arquivo, como programas, bibliotecas, *scripts* ou arquivos de configuração, independente do formato. Em contrapartida, a gerência da base de assinaturas recai sobre o administrador e/ou distribuidor do Sistema Operacional.

Observe que estas soluções não protegem da exploração de *bugs* existentes nos programas assinados, e não protegem do acesso direto à memória do sistema para adulterar o *kernel*.

O acesso direto à memória do *kernel* pode ser realizado através de um grampo no *hardware*, ou através do acesso a memória pela interface */proc* (no caso específico do

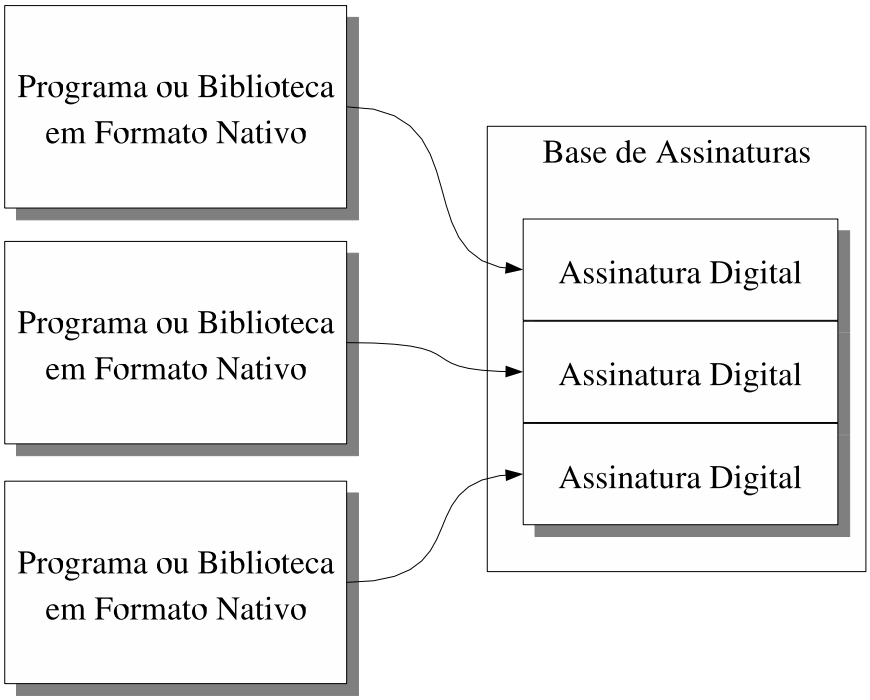

Figura 2.10: Uso de uma base de assinaturas

Linux), utilizando-se do *device /dev/kmem* (toda a memória do sistema, incluindo o *kernel* e seus dados), se o dispositivo permitir a escrita. Através deste acesso é possível reprogramar o *kernel* em tempo de execução, inserindo uma vulnerabilidade, como por exemplo, remapeando as *system calls* para rotinas desenvolvidas pelo atacante.

Em todas as abordagens apresentadas para verificação de assinatura, o desempenho do sistema é prejudicado porque é acrescentada uma etapa pesada antes do carregamento de outros processos. Esta etapa sempre acontece em um momento crítico, quando o sistema operacional está usando recursos para carregar a imagem do processo, e é obrigatório que esta etapa seja executada antes da execução do processo. Uma alternativa para contornar a perda de desempenho é executar a verificação somente quando dados são lidos do disco, e caso os dados estejam em *buffers* de memória a verificação é suprimida, considerando que a memória do sistema é um dispositivo seguro. Atente ao fato de que se o *buffer* de memória que contém o processo é invalidado no sistema operacional, é preciso fazer uma nova verificação. Paginação de memória (para *swap*) em meios inseguros, como rede, por exemplo, compromete a segurança do sistema como um todo.

2.6 Provas de segurança

O uso de provas de segurança [Rosu and Segerlind, 1999, Necula and Lee, 1997] é uma técnica que utiliza meta-dados como forma de garantir a segurança de um programa. Estes meta-dados consistem de uma prova matemática formal de que um dado código binário respeita uma especificação, fornecida pelo cliente do *software* ao desenvolvedor. O desenvolvedor do *software* é encarregado de fornecer a prova de que o *software* realmente respeita a especificação, e o sistema operacional aplica a prova sobre o binário em linguagem *assembly* e verifica se o mesmo realmente respeita a especificação.

Devido ao fato de ser aplicada uma verificação com matemática formal sobre o *software*, após a verificação é possível provar que o *software* respeita completamente a mesma. Portanto, a única forma de um *software* conter algum elemento nocivo é explorar uma forma de corromper o sistema que não tenha sido contemplada na especificação, ou seja, o sistema é tão forte quanto for abrangente a especificação.

O desenvolvimento do método de provas ainda depende de pesquisas a respeito da melhor lógica a ser usada para especificar a política de segurança, além de uma geração automatizada das mesmas. Outro ponto importante que ainda precisa de mais estudos é

o tamanho que as mesmas podem ocupar, que nos protótipos apresentados pelos estudos pesquisados [Rosu and Segerlind, 1999] chegam a ocupar um espaço semelhante ao do próprio código binário. A grande força deste método é a prova formal e matemática da segurança, independente do uso de criptografia e de confiança em um agente certificador externo.

2.6.1 Sistema operacional com segurança completamente baseada em análise de código

A análise de código, tanto em uma forma simplificada (como simplesmente procurar alguma instrução privilegiada dentro de um binário) quanto como formas mais sofisticadas (como o uso de provas de segurança [Rosu and Segerlind, 1999, Necula and Lee, 1997]) podem ser utilizadas, junto com controle de paginação ou segmentação, para garantir a segurança de um sistema operacional.

A proposta é levantada para a construção de um sistema inteiramente baseado em análise de código, em que o uso de *system calls* como pontos de acesso para a transição de um modo usuário para o modo supervisor do microprocessador é apresentado como um gargalo tanto do ponto de vista de desempenho quanto principalmente um gargalo entre diferentes interfaces de acessos a métodos de *software*.

Além de não utilizar *system calls*, é proposto a substituição de outros métodos de segurança atualmente utilizados, como o uso de modo privilegiado, exceções lançadas pelo microprocessador ao sistema tentar executar algum *opcode* inválido, *page faults* e inclusive o uso de interrupções para executar a preempção de um processo.

A abordagem em uso é ignorar o uso de modos de operações protegidos do microprocessador, ou seja, todo código executa em modo privilegiado, baseando a segurança completamente em análise de código, antes do mesmo ser instalado no sistema, e no controle da memória que um *software* pode acessar. Todo o acesso ao *hardware* é relegado a *softwares* auxiliares do sistema operacional, como o acesso a *hardware* para o desenvolvimento de *device drivers*, inclusive gerenciamento de interrupções.

Para a proteção da memória entre processos é proposto provar matematicamente que um código somente acessa a sua área de memória privada, além de provar que o código não vai tentar se alterar (impedindo que o processo mapeie sua página de código como página de dados e se modifique, burlando a verificação). Este tipo de abordagem é interessante

principalmente em maquinas que não contam com mecanismos adicionais de segurança já implementados por *hardware*.

O problema levantado no estudo [Rosu and Segerlind, 1999, Necula and Lee, 1997] é o fato de que a análise simples do código não garante a preempção do programa, porque não é possível, através de um algoritmo, garantir que um código sempre libera o processador.

Outro ponto importante é que o sistema de arquivos deve ser completamente seguro, para que um processo não possa criar outro arquivo em disco ou alterar a si próprio, criando uma nova versão que poderia corromper o sistema.

O fato de que a verificação de segurança só ocorre durante a instalação de um *software* potencializa os perigos que um ataque ao sistema de arquivos pode causar, além de simplificar a proliferação de vírus.

O objetivo principal do trabalho é o desenvolvimento de um sistema voltado à verificação de código, e portanto a forma de segurança aplicada não é o enfoque principal, e sim o aumento de desempenho e flexibilidade do sistema.

2.7 Considerações finais

Este capítulo apresenta um conjunto de idéias e mecanismos que são utilizados para prover integridade a sistemas operacionais, assim como pesquisas e tecnologias relacionadas com a proteção contra adulterações de programas e bibliotecas.

Os mecanismos essenciais para o desenvolvimento de sistemas operacionais seguros estão presentes no *hardware*, e são apresentados na seção 2.3. Existem dois mecanismos complementares, a proteção via segmentação (parágrafo 2.3.3) e a proteção via paginação (parágrafo 2.3.4). Para o desenvolvimento de um sistema operacional seguro é necessário o uso de no mínimo um dos dois mecanismos.

As tecnologias relacionadas com este estudo podem ser divididos em dois grupos:

- Sistemas baseados em criptografia;
- Sistemas baseados em provas de segurança.

O primeiro grupo engloba a maioria das soluções, e pode utilizar a criptografia com transformadas *hash* (seção 2.5) ou através da composição de assinaturas digitais (seção 2.4) usando transformadas *hash* e criptografia assimétrica.

O segundo grupo (seção 2.6) é uma forma inovadora de trabalhar com a segurança (utiliza provas matemáticas de segurança). As provas de segurança, apesar de serem uma forma promissora de fortalecer a segurança de sistemas operacionais, somente poderão ser utilizadas em sistemas operacionais completos após pesquisas futuras que viabilizem a sua aplicação.

O método apresentado neste trabalho baseia-se em criptografia.

Dentre os sistemas baseados em criptografia, existem soluções baseadas em transformadas de *hash* e soluções baseadas em assinaturas digitais. As soluções baseadas em *hash* são utilizadas para a detecção de modificações no *software* que compõe o sistema operacional. As soluções baseadas em assinaturas digitais são utilizadas tanto para a detecção de modificações quanto para provar a autenticidade do *software*.

O estudo desenvolvido neste trabalho utiliza assinaturas digitais, sendo que as transformadas criptográficas estão descritas no capítulo 3 e o formato da Assinatura Digital integrado com o formato do arquivo utilizado para a validação do sistema é apresentado no capítulo 5.

Ambos as formas de criptografia pode ser aplicadas em pontos diferentes do sistema. Os pontos em que a verificação ocorrem podem ser:

- Na instalação do *software* (parágrafos 2.4.1, 2.4.2, 2.4.3);
- Periodicamente (parágrafo 2.5.1);
- Sempre que utilizado (parágrafo 2.5.2).

Este trabalho foi desenvolvido para que a verificação ocorra sempre que o *software* é utilizado pelo Sistema Operacional. Além da verificação ser feita sempre que necessário, a verificação é feita de forma oportunista, sendo que somente as partes do *software* que são efetivamente utilizadas são verificadas.

Capítulo 3

Transformadas Criptográficas

Este capítulo apresenta considerações relativas ao uso de assinaturas digitais para a verificação de *software*. Existem formas diferentes de verificação, conforme apresentado no capítulo 2. Serão consideradas para este desenvolvimento as formas de verificação aplicáveis durante a carga do executável, baseadas em criptografia, que serão descritas nos capítulos 4 e 5.

3.1 Uso de *hash*

3.1.1 Funções de *hash* adequadas para a aplicação

O termo função de *hash* é empregado para uma grande família de funções, que obedece às seguintes propriedades:

- Efetuam o mapeamento de uma seqüência de bits x de tamanho variável para uma seqüência de tamanho fixo $h(x)$;
- É computacionalmente fácil calcular uma saída $y = h(x)$ para uma determinada seqüência de entrada x.

As funções de *hash* são divididas em duas grandes famílias, as funções chamadas *MDC* (código para detecção de modificações) e as funções chamadas *MAC* (código para autenticação de mensagens).

As funções do tipo *MAC* utilizam um algoritmo e uma chave, e a saída da função depende da chave aplicada no processo. As funções do tipo *MAC* apresentam como ca-

racterística a possibilidade de uso para fins de autenticação, devido ao uso de uma chave para o processo de geração do *hash* (é computacionalmente impossível para um atacante gerar um *hash* $y = h_k(x)$ sem o conhecimento da chave k).

A família *MDC* aplica somente o algoritmo, e gera uma saída, portanto é fácil para um atacante gerar um *hash* deste tipo para qualquer mensagem, considerando que não existe uma chave em segredo sendo utilizada. Para o problema específico de verificação de integridade de um *software*, a família de funções mais apropriada é a *MDC*. O objetivo de aplicar uma função *hash* sobre o *software* é detetar adulterações, sem a necessidade de provar a fonte da mensagem. Dentro da família de funções *MDC*, existem três características importantes que devem ser analisadas:

1. Resistência a encontrar uma pré-imagem x (entrada que gera um certo *hash*), ou seja, para um dado valor de *hash* y, é computacionalmente impossível encontrar uma pré-imagem x tal que $h(x) = y$;

2. Resistência a encontrar a segunda pré-imagem, ou seja, para uma entrada x fixa, é difícil para um atacante encontrar outra imagem x' com o mesmo *hash* ($h(x) = h(x')$);

3. Resistência a colisão, que é a dificuldade em, mesmo tendo escolha livre de duas entradas, encontrar um par que apresente o mesmo *hash*, ou seja, $h(x) = h(x')$ sendo que o atacante pode escolher x e x' arbitrariamente.

As funções de *hash* que apresentam a primeira e a segunda propriedade são chamadas de *OWHF* (*one way hash function*), e as funções que apresentam as três propriedades são chamadas de *CRHF* (*collision resistant hash function*).

Dependendo do tipo de aplicação a ser empregado o *hash*, é necessário um algoritmo de *hash* que obedeça uma ou mais destas características.

Para o uso na verificação de dados em que o atacante tem total controle sobre uma das entradas do algoritmo, a função de *hash* deve apresentar as características 1 e 2 (tanto uma função *OWHF* quanto *CRHF* satisfazem a essa restrição [Menezes et al., 1996]). Caso o caso do invasor adultere parte do sistema operacional, o atacante tem acesso tanto ao valor do *hash* y como a entrada x, que é parte do próprio arquivo.

Doravante, neste trabalho as funções de *hash* que devem ser utilizadas neste tipo de aplicação serão chamadas pelo termo completamente genérico de função de *hash*.

3.1.2 Aplicação de uma tabela de *hash*

Algumas soluções [Williams, 2002] são baseadas na verificação de um sistema posteriormente à sua instalação, e o uso de *hash* de todos os arquivos do sistema para criar uma 'fotografia' do estado do sistema naquele instante. Posteriormente, é comparado este instantâneo com o sistema durante o funcionamento buscando por adulterações. Esta abordagem gera uma tabela com um *hash* para cada elemento assinado (figura 3.1), que deve ser armazenada em alguma área de memória protegida pelo sistema operacional.

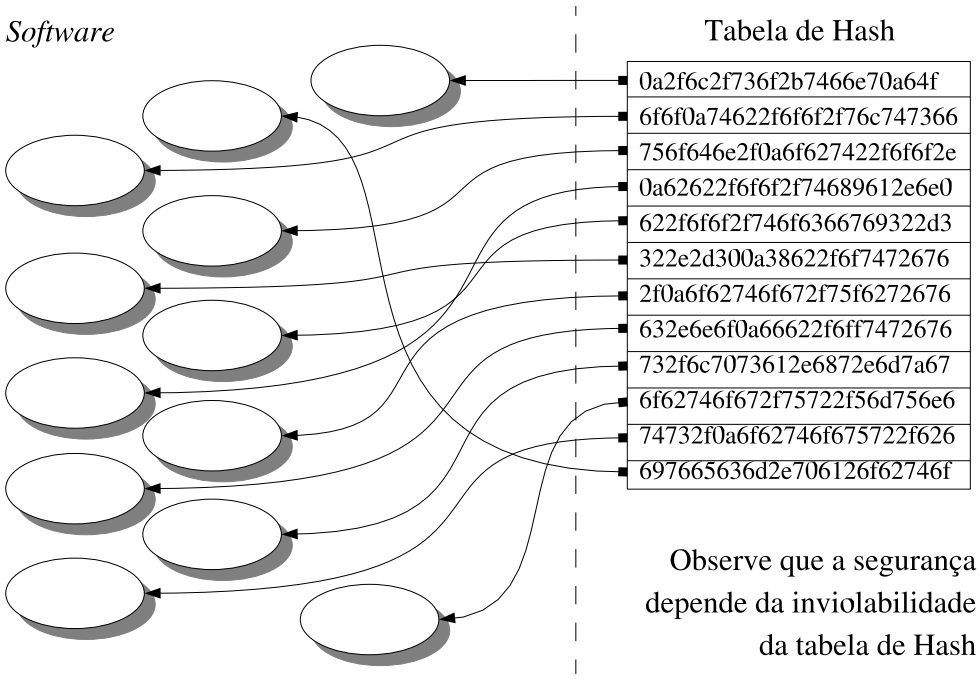

Figura 3.1: Tabela contendo o *hash* de todos os arquivos do sistema operacional

A segurança deste sistema depende da inviolabilidade desta memória, e da proteção que o sistema operacional possibilita para a proteção desta tabela. É possível utilizar um *hardware* para proteger estes dados, aumentando a segurança da solução, ou ainda alterar o sistema operacional para proteger esta área de memória.

As desvantagens desta abordagem envolvem:

- A dificuldade de fazer uma atualização ou instalação de um novo *software* no sistema, porque após cada mudança é necessário atualizar a tabela de controle;
- A responsabilidade de validar todos os *software* do sistema, que recai sobre os administradores.

3.2 Uso de estrutura de chaves públicas

Uma forma de fazer a verificação dos *softwares* consiste em utilizar a criptografia de chave pública para dividir o trabalho de verificar e autenticar os *softwares*. É possível simplificar a administração de um grande número de pacotes, e ainda diminuir a quantidade de dados que deve ser protegida de adulteração, conforme explicação a seguir.

3.2.1 Uso de assinaturas digitais

O uso de uma estrutura de chaves públicas possibilita a utilização de um sistema de segurança mais completo que a aplicação simples de funções de *hash*.

A idéia principal da assinatura digital é utilizar algoritmos de criptografia assimétrica para construir uma forma eficiente e segura de verificar a autenticidade, identificação e não-repudiação (impedir que quem gerou a assinatura possa negar que tenha assinado) da mensagem. Uma forma simples de gerar uma assinatura digital seria utilizar criptografia assimétrica, mas esta abordagem tem a desvantagem de que a criptografia assimétrica é muito mais lenta que a criptografia simétrica e as funções de *hash* [Menezes et al., 1996]. Uma forma de aumentar a velocidade da assinatura digital é combinar algoritmos de funções de *hash* e criptografia assimétrica [Copeland et al., 1999]. Desta forma, é efetuado um *hash* de um *software* x, de tamanho finito arbitrário gerando um *hash* de tamanho fixo $y = h(x)$, sendo que a função $h()$ é do tipo *CRHF*. A seguir aplica-se a criptografia somente ao *hash* y, e como o *hash* tem tamanho fixo muito menor que o arquivo, a criptografia assimétrica do *hash* é muito mais rápida que a criptografia da mensagem (arquivo) inteira. Portanto, o processo de aplicar o *hash* e criptografar o mesmo gerando a assinatura é mais rápido que aplicar a criptografia assimétrica a toda a mensagem.

Para o caso da assinatura digital de um *hash* do *software* a proteger, é especialmente importante que a função de *hash* seja do tipo *CRHF*. Com uma função que apresenta resistência a colisão, é muito difícil que um atacante consiga encontrar dois *softwares* com o mesmo valor de *hash* (uma colisão por definição). Caso um atacante conseguisse obter um outro *software* com o mesmo valor de *hash*, o atacante poderia simplesmente trocar os *softwares* um pelo outro, e desta forma a criptografia assimétrica verificaria o valor do *hash*, a assinatura do mesmo seria positiva, e portanto o atacante poderia trocar *software* no sistema sem ser detectado, comprometendo a segurança.

3.2.2 Forma de distribuição

O primeiro ponto a ser estudado é a mudança dos meta-dados para verificação do arquivo, de forma a armazenar os mesmos junto com o *software* a ser verificado (capítulo 5). Desta forma, os meta-dados enfrentam o mesmo risco de adulteração que o *software*. O risco de um ataque do tipo *denial of service* via adulteração intencional dos dados usados para verificar o *software* existe da mesma forma que o risco de corromper o próprio *software*. A segurança do método consiste em não permitir que um *software* corrompido execute, e não em impedir que o *software* seja corrompido. Caso o invasor tenha acesso direto ao sistema de arquivos ou ao *hardware*, existem formas muito mais simples de danificar o sistema para provocar uma pane (*denial-of-service*).

3.2.3 Arquitetura de assinaturas

Através do uso de assinaturas digitais, é possível encadear a autenticação do *software*, possibilitando a criação de uma estrutura de chaves (figura 3.2).

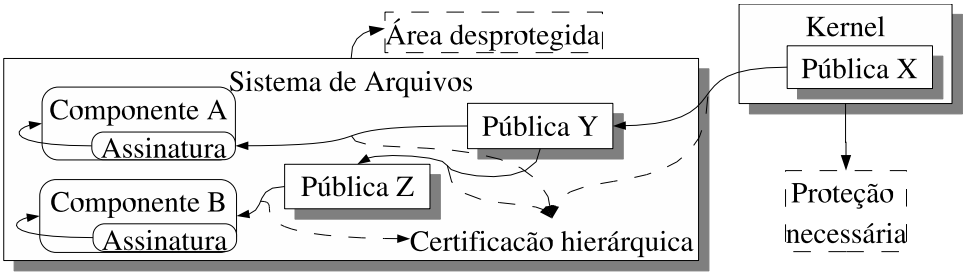

Figura 3.2: Exemplo de encadeamento de chaves

A vantagem principal do encadeamento é permitir que um conjunto mínimo de chaves verifique em cascata um conjunto maior de chaves até chegar ao *software*. Isto simplifica a construção de um sistema seguro de verificação, porque é necessário proteger apenas um conjunto mínimo de dados, e utilizar a criptografia de chaves públicas para estender a segurança de forma encadeada até chegar ao *software* alvo a verificar.

Nas seções seguintes será utilizado o seguinte exemplo de árvore de certificação (figura 3.3).

O esquema proposto para o exemplo baseia-se em um sistema operacional distribuído por uma entidade, que utiliza um par de chaves X para assinar os pacotes que compõe o

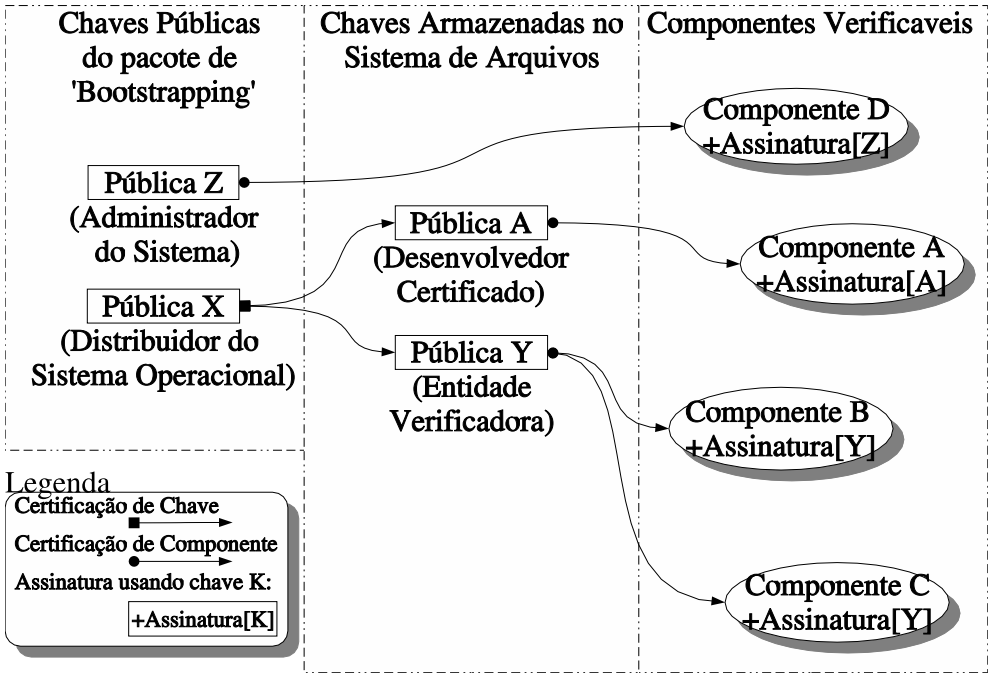

Figura 3.3: Exemplo de sistema utilizando uma árvore de certificações

sistema e ainda confiar a assinatura a outra entidade verificadora (através da assinatura da chave Y), que no exemplo certifica a segurança dos pacotes B e C. Observe ainda que o administrador do sistema pode adicionar uma chave ao pacote de *boot* de forma a poder inserir binários de sua autoria no sistema, e possivelmente certificar outros desenvolvedores para criarem e testarem *softwares*.

3.2.4 Uso de um conjunto mínimo de *software* para validar o resto do sistema

Devido à redução da necessidade de informações que são sensíveis e devem ser protegidas para que a solução seja segura, é possível isolar uma parte mínima do sistema operacional de forma a efetivar a proteção deste pacote através do armazenamento em mídia segura. Neste tipo de sistema, é proposto utilizar um conjunto mínimo do sistema operacional (tipicamente o *kernel* com o sistema de verificação de assinaturas e os *drivers* necessários para realizar a operação de *boot* da máquina), junto com um conjunto mínimo de chaves pai (por exemplo, a chave pública do administrador da máquina e da empresa que realiza a distribuição do sistema operacional), e proteger este pacote utilizando um sistema forte de proteção via *hardware* (uso de um meio de armazenamento somente para leitura, ou

um meio removível que é utilizado somente para efetuar o *boot* da máquina, devendo ser removido logo após o *boot* (figura 3.4).

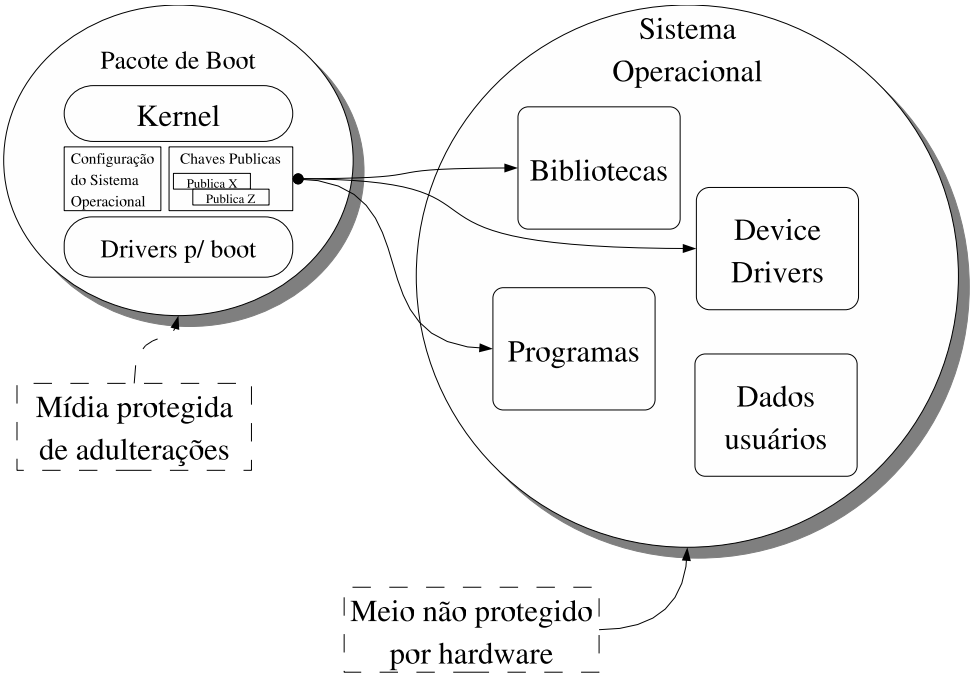

Figura 3.4: Configuração de um sistema com *bootstrap* para verificação de assinaturas

3.2.5 Funcionamento do sistema proposto

Através da infra-estrutura de chaves públicas [ITU, 2000, Lee and Kim, 1999], é possível construir um sistema capaz de verificar a autenticidade de todos os executáveis, conforme descrito no diagrama 3.5.

No exemplo apresentado, observe que o sistema operacional foi iniciado com apenas duas chaves públicas, e que o sistema pode validar as outras chaves envolvidas no processo de verificação de segurança. Na situação apresentada, a chave X representa a chave fornecida pela empresa que distribui o sistema operacional, e a chave Z foi inserida pelo administrador de sistema. Devido à necessidade de verificar os *softwares* A, B, C e D o sistema operacional necessita estabelecer um caminho da assinatura feita no *software* até uma das duas chaves pai. Este caminho necessita da chave A, que é a chave pública de um desenvolvedor no qual a empresa confia. O sistema deve buscar a chave, através de um agente de *software*, por exemplo, tanto de um banco de chaves presente em disco, ou através de um servidor de chaves em alguma rede a qual o sistema tenha acesso. Além

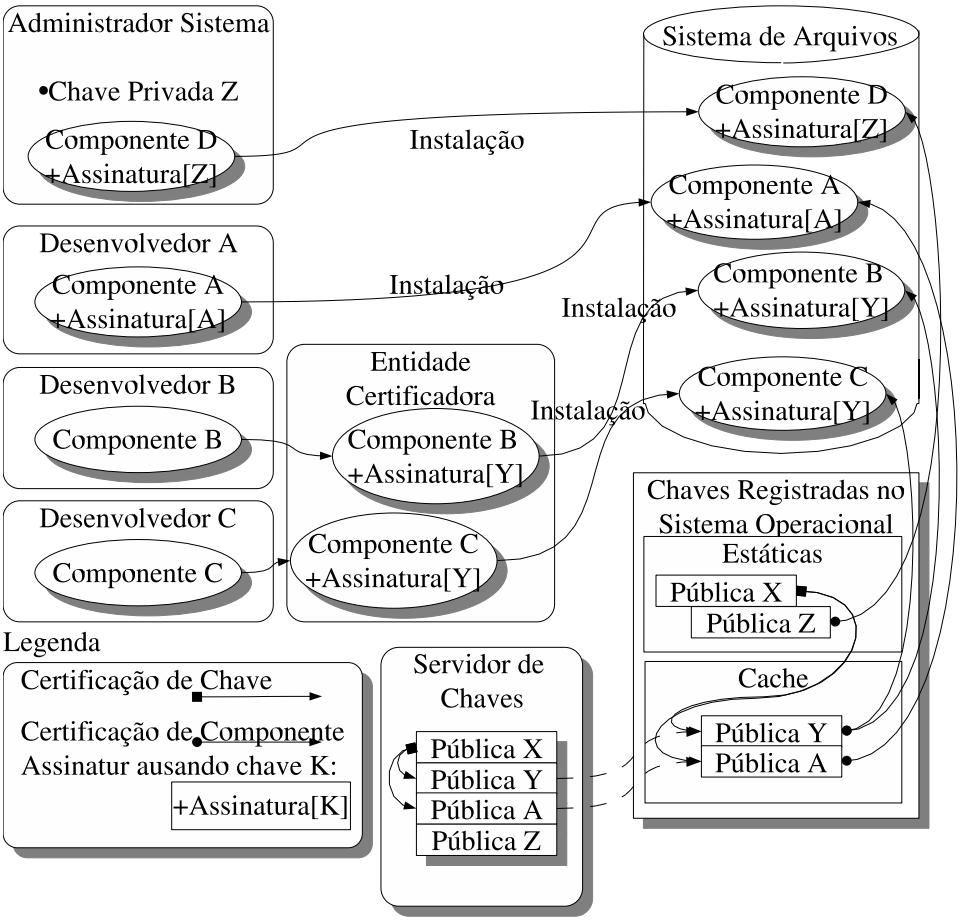

Figura 3.5: Relações entre certificadores e verificação para sistema baseado em chaves públicas

desta chave, o sistema precisa da chave Y, que é a chave de uma agência de verificação de *software* em quem a empresa que distribui o sistema operacional confia, para validar os *softwares* B e C. A verificação do *software* D é feita diretamente através da chave pública Z. Através desta estrutura, o sistema pode validar de forma encadeada vários executáveis, que podem ser assinados separadamente e distribuídos de várias fontes distintas, simplificando o processo de atualização de *softwares* individuais.

Observe que o Sistema Operacional deve proteger adequadamente as chaves em *cache*, e sempre que uma chave é carregada de um meio não confiável (*cache* em disco, por exemplo), toda a verificação de autenticidade deve ser executada novamente até chegar a uma das chaves raiz.

3.2.6 Considerações relativas à segurança do modelo

A solução apresentada tem como vantagem a verificação *software* a *software*, ou seja, assinar cada *software* individualmente, em momentos distintos. Esta análise *software* a *software* possibilita uma verificação mais completa, considerando o que o administrador de sistemas faria em um sistema instalado com um grande número de pacotes de *software*. Em contrapartida, é confiado em um grande número de entidades, o que enfraquece a segurança.

Outro problema deste método é que a segurança é a confiança na autoridade que verificou e assinou os *softwares*, portanto se esta agência ou autoridade certificou a versão do programa com o *bug*, este *software* e o respectivo *bug* será autorizado pelo sistema de verificação. Neste tipo de modelo, como a segurança é constantemente verificada, existe a importância de chaves de revogação para cancelar o elo de confiança previamente estabelecido [ITU, 2000]. No caso deste cancelamento, outro problema surge, que é permitir ou não a execução do *software*. Caso seja permitido, o sistema é vulnerável, caso seja impossibilitado, o sistema possivelmente não funcionará. A parada de sistemas freqüentemente é um problema crítico. O ideal é o sistema sinalizar que precisa de uma manutenção urgente, porque foi detectado que o sistema operacional encontra-se em estado inseguro.

A ação tomada ao encontrar uma adulteração pode depender da função do computador na rede. Se o mesmo for um servidor, é importante que o mesmo mantenha-se funcionando, mas neste caso o comprometimento da segurança pode resultar em danos muito maiores do que a parada do serviço. Para um servidor com nível de segurança muito elevado, o mesmo pode efetivamente parar para evitar que algum invasor utilize esta vulnerabilidade para

realizar algum ataque, e a necessidade de manutenção torna-se urgente. Se o computador for uma máquina com a função de servir de isca [Williams, 2002] para algum invasor, função conhecida através do nome de *'honey pot'*, provavelmente a máquina deve continuar funcionando como se o sistema de segurança não existisse, e de forma silenciosa a máquina enviaria um *log* de segurança para o administrador da rede.

3.3 Uso da criptografia no sistema

As transformadas criptográficas apresentados neste capítulo são utilizados para construir a assinatura digital que garante a integridade e autenticidade do *software* a ser protegido.

Esta assinatura digital foi desenvolvida para ser integrada dentro do formato do arquivo que contém o programa ou biblioteca, conforme descrito no capítulo 5. Esta integração proporciona transparência para os usuários finais e administradores, e simplifica a distribuição de *software* feita por empresas ou desenvolvedores individuais.

Capítulo 4

Análise Oportunista Durante a Execução

4.1 Mecanismo de verificação

O mecanismo utilizado neste trabalho para garantir a integridade do sistema é a verificação do binário a ser executado durante a carga do mesmo, para evitar que uma adulteração feita depois da instalação do binário no sistema de arquivos não seja detectada. A verificação de segurança foi posicionada internamente ao sistema operacional, possibilitando a detecção de adulterações que aconteçam até imediatamente antes da parte do *software* ser necessária para o processo, e portanto, carregada para formar a imagem do mesmo.

A figura 4.1 ilustra a relação entre o ponto no qual pode ser feita a verificação e o efeito que a mesma impacta na segurança do sistema. Observe que quanto mais a direita na figura 4.1, mais tarde é feita a verificação, e portanto mais seguro o sistema se torna. Quanto mais tarde for feita a verificação, mais próximo do código ir para execução pelo microprocessador, portanto menor a possibilidade da adulteração ser feita depois da verificação. Em contrapartida, aumenta o número de vezes em que a verificação é feita. Quanto maior o número de vezes que o algoritmo é aplicado, maior é o atraso causado pela verificação.

Neste trabalho é proposto a verificação toda vez que algum processo precise de um parte (página) do *software*, e a mesma seja carregada pelo microprocessador para a memória. Desta forma, são utilizados todos os mecanismos de otimização do sistema operacional e microprocessador, que buscam evitar uma repetição de carga da mesma

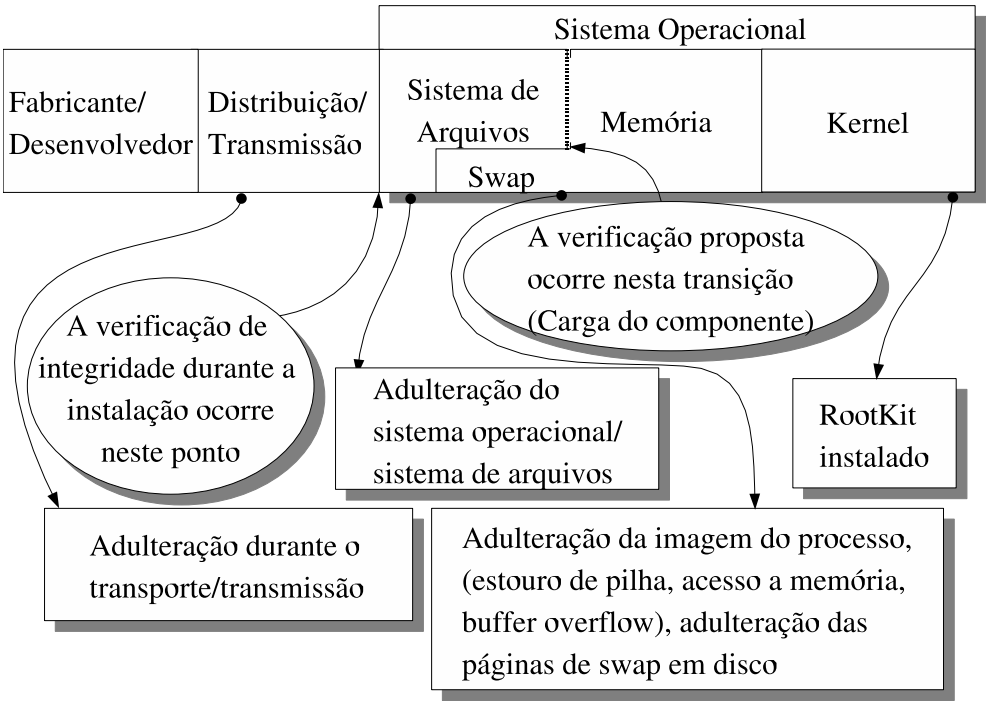

Figura 4.1: Mapa dos pontos em que é possível inserir verificações de integridade

informação para a memória várias vezes, considerando que cada carga constitui uma operação custosa para o sistema. Fazendo-se uma verificação a cada carga, o mesmo mecanismo que otimiza a gerência de memória do sistema otimiza a verificação criptográfica, evitando verificar várias vezes a mesma informação (página), verificação que é uma operação computacionalmente custosa para o sistema.

4.2 Segurança da abordagem

Observe que este sistema não é capaz de detectar uma adulteração na imagem do processo depois da carga do *software*, devido ao momento em que a verificação é proposta, portanto a segurança do sistema está limitada à segurança do programa em execução, e do sistema operacional empregado (a segurança do sistema de arquivos ou a do controle da memória, entre outros fatores).

No caso da segurança do programa, caso seja possível corromper um programa através da utilização de um *bug* no mesmo (estouro de pilha, estouro de *buffer*, uso de uma condição não prevista pelo desenvolvedor, etc), este pode se comportar de forma a comprometer a segurança, e a verificação feita durante a carga não tem efeito nenhum sobre

o mesmo, considerando que a adulteração foi feita depois da verificação (salvo no caso de provas de segurança, conforme descrito no capítulo 3).

Para o caso da segurança do sistema operacional, o mesmo é responsável por controlar todo o sistema e prover áreas lógicas independentes para os programas. Qualquer falha na gestão destas áreas, ou ainda qualquer *bug* que possa ser utilizado para adulterar o sistema operacional pode resultar numa falha de segurança fora do alcance da solução proposta.

Caso o sistema operacional seja suficientemente seguro a ponto de impedir adulterações do programa e dados utilizados pelo mesmo, o ponto fraco da verificação é a dificuldade com que um invasor pode burlar o método utilizado para validar o *software*, ou seja, a segurança do sistema cai ao nível da segurança do método de verificação (baseada em criptografia), que é apresentado no capítulo 3.

Finalmente, caso o *hardware* em uso não tenha mecanismos de segurança suficientes para o sistema operacional utilizar, ou exista uma falha nos mesmos, todo o sistema que depende do hardware está vulnerável a ataques independente do *software* que está sendo utilizado.

A segurança do sistema é uma conjunção de uma série de mecanismos, e é tão forte quanto o mais fraco dos mecanismos (assim como os elos de uma corrente).

4.3 Implementação e validação

A implementação do trabalho foi feita sobre o sistema operacional *Linux*, com o objetivo de validar a proposta. A implementação da chamada de verificação de dispositivos depende de detalhes intrínsecos ao sistema operacional utilizado para a validação, portanto os dados e métodos levantados para a validação deste trabalho tem uma dependência com a arquitetura utilizada.

A implementação foi baseada interceptando algumas chamadas do sistema operacional, onde é oportuno efetuar algumas checagens de segurança. Para tanto foi utilizado o sistema de módulos de segurança do *Linux* [Morris et al., 2002b, Morris et al., 2002a, Loscocco and Smelley, 2001].

A validação depende do formato utilizado pelo sistema para o binário [TIS, 1993], porque o mecanismo de validação depende de meta-dados relativos ao binário que serão

embutidos dentro do arquivo do programa (conforme descrito no capítulo 5) para facilitar a distribuição do *software*.

O mecanismo de carga de binários e bibliotecas utilizados no sistema operacional *Linux* possibilita o uso de vários padrões de arquivos, podendo carregar binários feitos para versões antigas do sistema, para outros sistemas operacionais e inclusive existe a possibilidade de utilizar o mecanismo de carga de executáveis do *kernel* para executar formatos que dependam de um interpretador, como por exemplo *shell scripts* ou arquivos *bytecode* da linguagem *java*.

O *Linux* utiliza uma lista encadeada de formatos, e para cada binário o sistema operacional busca na lista um carregador apropriado (figura 4.2).

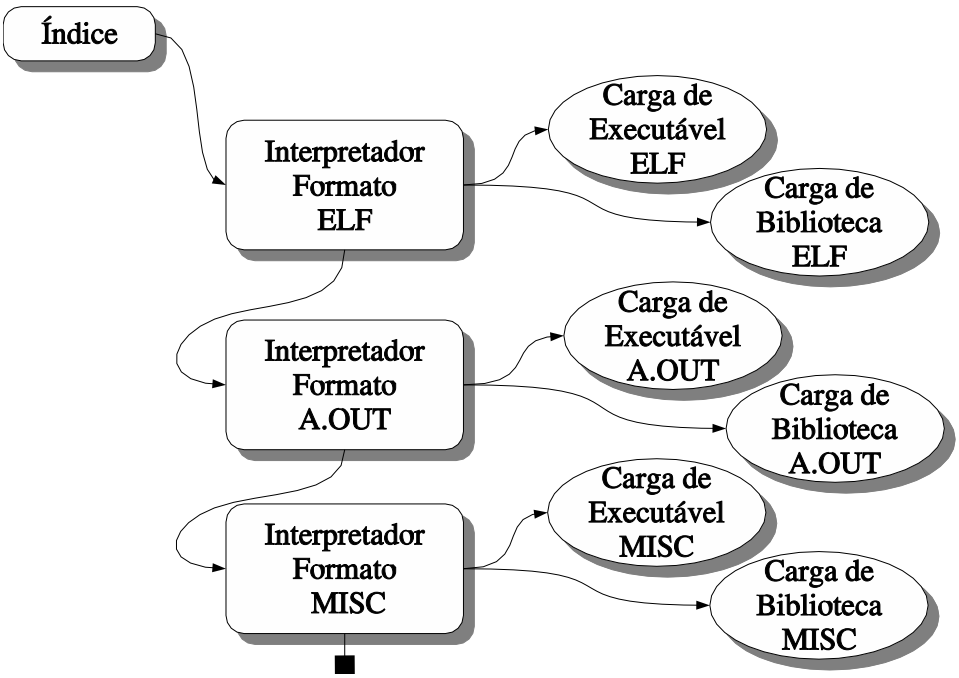

Figura 4.2: Forma como o sistema busca por interpretadores para um dado programa ou biblioteca

Portanto, para que o sistema de segurança implemente a verificação dos binários, é necessário a implementação da verificação para cada um dos formatos que serão utilizados no sistema operacional destino. O ponto exato onde será inserida a verificação é dentro das rotinas individuais para carga de programas e bibliotecas, específicas para o formato em questão.

4.4 Formas de aumentar o desempenho da abordagem

A segurança desta forma de abordagem está ligada com um impacto no desempenho do sistema, devido ao custo computacional dos algoritmos criptográficos aplicados aos programas. A verificação impacta na velocidade do sistema principalmente devido ao fato de que quando o sistema está carregando a imagem do processo, existe um grande esforço de transferência de dados para a criação de um novo processo. Realizando a verificação criptográfica incremental à medida que o mecanismo de paginação transfere dados para a memória, é possível utilizar o tempo que o Sistema Operacional está esperando a transferência de uma página para verificar a página anterior, de forma a aumentar a eficiência do sistema.

Em um sistema orientado a eventos (por exemplo, um computador *desktop*), os programas ficam esperando por eventos do usuário e do sistema operacional, para efetuar algum processamento e bloquear esperando novos eventos. O momento em que este tipo de sistema carrega aplicativos é um dos pontos em que existe a maior preocupação com a velocidade, devido à espera do usuário por uma resposta do sistema. A verificação adicional ocasiona um aumento no tempo de carga de programas, e causa uma espera maior pelo usuário.

Para um sistema como um servidor, por exemplo, existe a possibilidade de agrupar um grande número de serviços em um único serviço pai, que se comporta para os clientes como cada um dos serviços diferentes gerenciados, aguardando requisições externas da mesma forma que se cada um dos servidores estivesse individualmente executando. Após a chegada de uma requisição externa, este servidor dispara o servidor real que atende a requisição (figura 4.3).

Para um sistema com um grande número de execuções seguidas de processo, como no exemplo citado acima, existe um grande número de cargas de uma mesma imagem de processo, e conseqüentemente o impacto de carregar seguidamente o processo e verificar a integridade é mais sensível no desempenho do sistema como um todo.

Apesar deste grande número de execuções, os mecanismos de gerência de memória dos sistemas operacionais tendem a manter em memória as informações mais utilizadas, para diminuir a transferência de dados, que é uma operação custosa. Combinar a verificação com o mecanismo de gerência de memória resulta neste caso em uma otimização do próprio

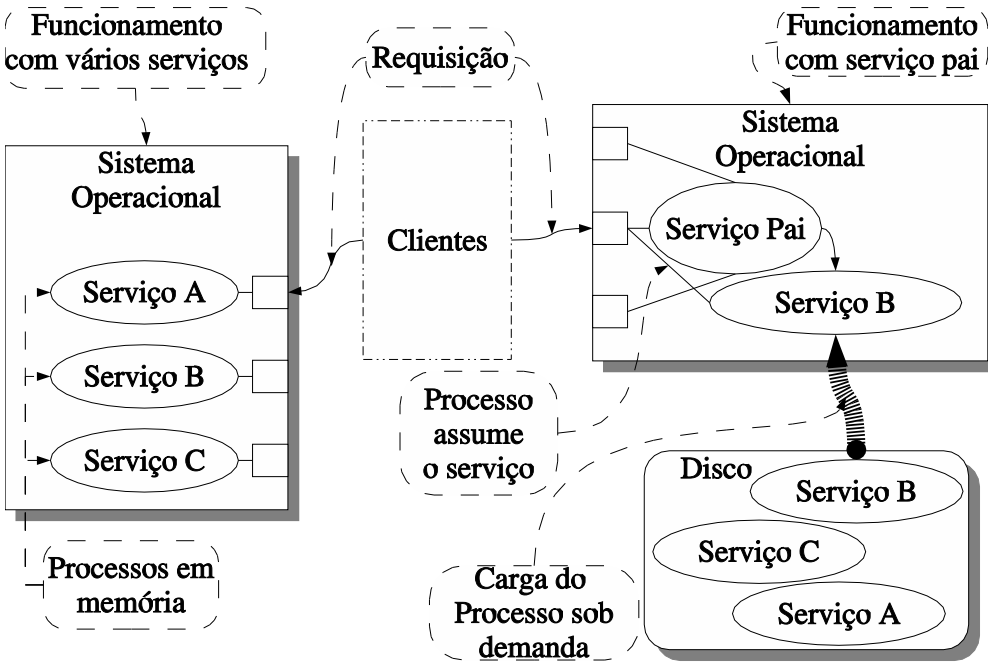

Figura 4.3: Funcionamento de agrupamento de serviços em um serviço pai

mecanismo de verificação.

4.4.1 Uso do *cache* de disco para reduzir o número de verificações

Os sistemas operacionais tipicamente utilizam um mecanismo chamado *cache*, para aumentar a velocidade dos acessos de disco. O mecanismo de *cache* consiste em manter uma cópia em memória dos últimos dados mais acessados, mesmo que não exista mais nenhum processo utilizando os mesmos, de forma a prevenir um acesso a um meio lento (sistema de arquivos) através da sua cópia na memória do sistema. Para uma carga sucessiva a um mesmo *software*, por exemplo, cada carga gera um acesso de leitura para as páginas correspondentes ao *software* em disco. Se o mesmo *software* foi executado há algum tempo atrás, e as páginas correspondentes continuam em memória, o sistema operacional não precisa buscar as páginas através de um acesso lento. Portanto, a velocidade do sistema aumenta.

Durante um acesso a um programa ou biblioteca do sistema, tipicamente é feito um mapeamento de páginas de memória para disco e o arquivo é bloqueado pelo sistema para modificações. Caso o arquivo deixe de ser utilizado e permaneça em *cache*, uma alteração

subseqüente no arquivo em disco invalida o *cache*, forçando uma nova leitura do disco para uma nova carga. Como o sistema impede automaticamente que o arquivo seja alterado se o mesmo estiver sendo utilizado, e invalida o *cache* caso seja feito uma alteração no seu espelho em disco, é possível verificar a assinatura do programa ou biblioteca quando o mesmo é carregado do disco, mas utilizar o mecanismo de *cache* para evitar que sejam feitas novas verificações sobre os *softwares* que ainda estiverem presentes em *cache*.

4.4.2 Uso de *hash* por página

Devido ao fato do *software* ser mapeado do sistema de arquivos para a memória (utilizando o sistema de paginação descrito no capítulo 2), é possível quebrar o *hash* aplicado ao arquivo em um *hash* para cada página que será mapeada para a memória (figura 4.4).

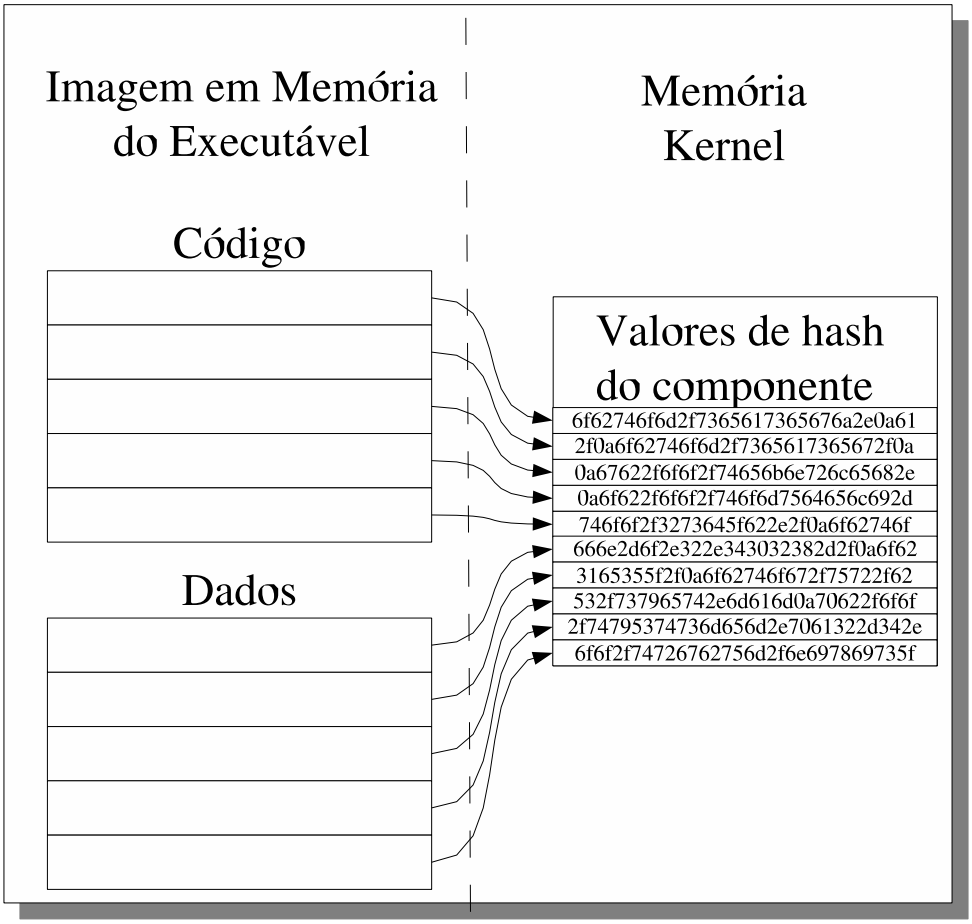

Figura 4.4: Uso de *hash* por página mapeada em memória

Ao utilizar um *hash* para cada página, é possível proceder a verificação individualmente para cada uma das páginas do arquivo. Ao realizar a verificação não para todo o arquivo e sim para cada página, o impacto da verificação é transferido para depois da montagem da imagem do processo, tornando-se uma verificação por página acessada (figura 4.5, que consiste em um detalhamento da figura 4.1).

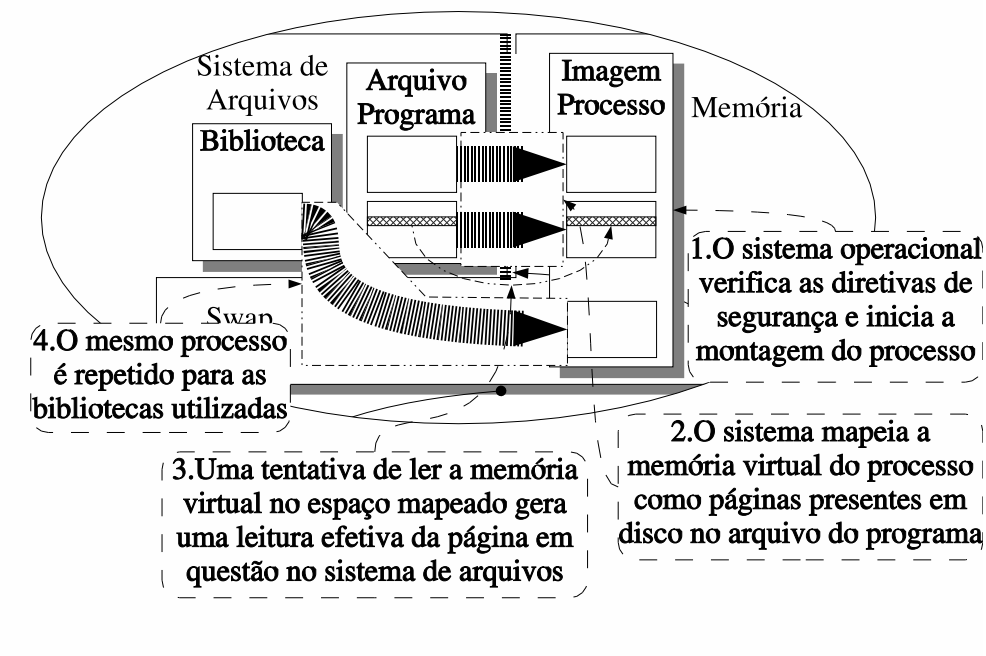

Figura 4.5: Processo de mapeamento de memória e carga de páginas

Para que um sistema efetue a verificação de cada página carregada do sistema de arquivos, é efetuado a verificação somente das páginas que são efetivamente utilizadas, o que é especialmente importante no caso das bibliotecas. As bibliotecas são compostas de uma coleção de rotinas, e os programas tipicamente utilizam apenas uma fração da biblioteca. Devido ao fato de que somente uma parte do arquivo deve ser verificada, evita-se uma grande transferência de dados do sistema de arquivos e um grande dispêndio de processamento para verificar dados que não são realmente utilizados. A verificação é feita de forma oportunista, apenas as partes dos programas e bibliotecas que são utilizados pelo processo são verificadas.

A divisão do programa em páginas para efetuar o *hash* necessita de duas considerações relativas a segurança:

1. Aumenta a necessidade de utilizar uma função de *hash* segura, considerando que efetuar o *hash* em pequenas páginas aumenta a quantidade de valores de *hash* presentes

em um sistema operacional. O aumento do número de valores de *hash* aumenta o universo de amostra no qual um adversário pode procurar por uma segunda pré-imagem (capítulo 3), facilitando portanto a tarefa de encontrar duas páginas com o mesmo valor de *hash*.

2. Torna-se necessário que, ao aplicar o mecanismo de assinatura digital aos valores de *hash*, seja empregada uma única assinatura para toda a tabela, de forma a proteger não somente o valor do *hash* das páginas individuais, mas proteger a ordem na qual as páginas aparecem no arquivo.

A ordem das páginas é uma informação importante de ser verificada, para evitar que um invasor troque a ordem das páginas assinadas para corromper o programa, e possivelmente utilize partes de um programa para gerar outro. Por exemplo, uma forma de corromper um programa é trocar as páginas para inserir um *bug* que abra uma brecha para um ataque de *buffer overflow*.

Apesar do enfraquecimento da segurança contra *colisões* (capítulo 3, duas entradas diferentes com o mesmo valor de *hash*) geradas pela primeira consideração acima, ao aplicar a segunda consideração o sistema torna-se mais seguro ao risco de um adversário encontrar dois *softwares* com a mesma assinatura digital. Para utilizar uma colisão em um sistema de *hash* por páginas e assinatura digital da tabela de páginas, não basta encontrar uma colisão em uma página de um *software*. É preciso encontrar dois *softwares*, com o mesmo número de páginas, e com os mesmos valores de *hash* para todas as páginas dos arquivos, na mesma ordem para ambos os arquivos.

4.4.3 Verificação preditiva de páginas mapeadas em memória

A verificação individual das páginas do programa ou biblioteca permite que o sistema operacional utilize processamento ocioso para fazer a verificação de páginas que ainda não foram requisitadas pelo programa. Este mecanismo antecipa o gasto de processamento, evitando interromper o programa durante a carga da próxima página. O processo de verificação preditiva de páginas pode ser disparado no exato momento após o mapeamento de memória de um processo, na forma de uma *thread* de baixa prioridade do próprio *kernel*.

Desta forma, evita-se atrasos causados pela paginação de um meio de armazenamento mais lento somado ao atraso da verificação criptográfica de cada página.

O uso de paginação preditiva para escolher as páginas com maior probabilidade de uso

futuro é objeto de estudos relativos ao desempenho de sistemas operacionais baseados em paginação [Albers, 1993].

A verificação preditiva deve considerar o tipo da página (código ou dados), e se a mesma foi mapeada na imagem do processo do arquivo em disco ou de uma biblioteca (os processos tipicamente utilizam apenas uma pequena fração das bibliotecas), de forma a verificar primeiro as páginas com maior probabilidade de uso. Observe que para vários processos recomenda-se utilizar um único *daemon* (processo) do *kernel* para verificar todas as páginas do sistema, de forma a direcionar esforços para as páginas mais prováveis tendo como universo de análise todo o sistema. Portanto, verifica-se as páginas com maior probabilidade de uso dos programas ou bibliotecas com maior chance de entrar em execução.

A verificação preditiva como um processo do *kernel*, considerando o desempenho, pode trabalhar de duas formas:

1. Verificar as páginas em disco, e marcar as mesmas como verificadas, para que no próximo *page fault* (necessidade de transferir uma página do sistema de arquivos para a memória) o sistema tenha apenas o *overhead* de transmitir a página do sistema de arquivos, evitando apenas o gasto adicional com a verificação criptográfica (feito anteriormente).

2. Causar artificialmente *page faults* de forma aproveitar o custo com transferência para colocar as possíveis páginas prontas para uso no sistema.

A primeira abordagem apresenta como vantagem o fato de não interferir com o processo atual de controle de memória do *kernel*, evitando que o algoritmo de predição interfira com os algoritmos de controle de paginação, *buffer* de acesso a disco e *cache*.

Em contrapartida, esta abordagem gera um esforço dobrado de I/O(entrada e saída) para o sistema de arquivos e respectivo *hardware*, considerando que é feito uma chamada de I/O para trazer a página para verificação criptográfica e uma segunda chamada de I/O para os mesmo dados em disco durante a carga conseqüente do uso da página (*page fault*). Observe que o mecanismo de *cache*, se presente no sistema operacional, pode eliminar este problema se a página (ou parte do arquivo correspondente no sistema de arquivos) ainda estiver em *cache*.

Na segunda abordagem, o *daemon* do sistema operacional gera um *page fault* efetivo, e a página é verificada e transmitida para a memória principal. A grande desvantagem deste método é que a página, ao ser transferida para a memória, interfere com o gerenciamento

de memória do *kernel* como se o sistema estivesse efetivamente precisando de páginas para executar programas, e esta interferência pode degradar artificialmente o mecanismo de controle de memória.

Esta degradação ocorre porque, quando o sistema coloca em *swap* páginas que esteja realmente necessitando, é reduzido o *buffer* dos sistemas de arquivos e o *cache* para o acesso a arquivos abertos. Isto diminui o desempenho de todo o sistema operacional para acessos a I/O, e gera trabalho para escrever no sistema de arquivos caso as páginas tenham sido modificadas. Em seguida o sistema operacional vai trazer as mesmas páginas para memória. Em contrapartida, é feito apenas uma transferência e verificação criptográfica e a página está pronta para uso, o que é, considerando apenas o acesso a páginas, um ganho de desempenho.

A escolha entre uma das duas abordagens depende de um estudo relativo ao sistema operacional e os mecanismos de aumento de desempenho empregados pelo mesmo em conjunto ao estudo do algoritmo de predição utilizado. Dependendo de cada combinação utilizada, um dos dois mecanismos pode ser mais apropriado para a tarefa.

No caso específico da implementação sobre o *Kernel Linux*, a segunda abordagem é utilizada pelo próprio sistema operacional, através de uma janela de tamanho variável de páginas para leitura preditiva. Devido à combinação do mecanismo de gerência de memória e da verificação criptográfica, efetivamente a segunda abordagem é empregada para a leitura preditiva de páginas e verificação de segurança das páginas carregadas.

Um ponto importante de estudo consiste na possibilidade da verificação ocorrer assim que termina a transferência de dados. A verificação pode ocorrer dentro da rotina de atendimento a interrupções do sistema operacional, com a chance de ocorrer uma inversão de prioridades entre processos concorrentes. A inversão ocorre quando um processo de alta prioridade esperará para utilizar o microprocessador devido ao sistema operacional estar em uma rotina de atendimento a interrupções verificando páginas para processos de prioridade menor.

Este problema pode ser resolvido com uma alteração da rotina de tratamento de interrupções para a mesma marcar a página como transferida, mas não liberada para o processo.

O processo continua esperando a página ser liberada até que a mesma seja verificada por um processo *Daemon* do sistema operacional, com a prioridade herdada da maior prioridade dentre os processos que estão bloqueados por tentar utilizar aquela página.

Este processo compete com todos os processos do escalonador do sistema operacional e verifica a mesma em benefício do processo cliente, liberando o mesmo para acessar a página caso a verificação seja bem sucedida. Como resultado, a verificação da página para um processo de baixa prioridade não compete com processos de prioridade superior, e portanto não existe inversão de prioridade.

4.5 Relação entre a verificação e o sistema completo

Os conceitos apresentados neste capítulo referem-se à verificação incremental do *software* à medida em que os processos são executados no Sistema Operacional. Esta verificação depende de um sistema de criptografia seguro, apresentado no capítulo 3, e de uma integração com o formato do arquivo que armazena os programas e bibliotecas, apresentada no capítulo 5.

Capítulo 5

Assinatura Digital Integrada

Este capítulo apresenta uma proposta de extensão do padrão de arquivos executáveis e bibliotecas *ELF* [TIS, 1993] (*Executable and linkable format*), para oferecer suporte a assinatura digital.

Apesar da implementação deste trabalho ser inteiramente feita como uma extensão para este formato, os conceitos propostos podem ser extrapolados para outros padrões.

A idéia central do padrão *ELF* consiste em criar arquivos que possam servir tanto para armazenar o código objeto, gerado durante as etapas de compilação e posteriormente utilizado para montar os binários, quanto para criar o arquivo final do *software*, seja o arquivo um programa ou biblioteca.

O padrão *ELF* abrange:

- O controle do alinhamento das seções, possibilitando o uso do mecanismo de paginação para a carga gradual do *software* à medida que o processo necessite;
- Um mecanismo de relocação, que permite que um dado código possa ser carregado em uma posição arbitrária na memória;
- Um mecanismo de ligação dinâmica, que possibilita o uso de bibliotecas compartilhadas.

Este capítulo introduz os aspectos mais relevantes do padrão *ELF*, considerando a aplicação de assinatura digital, e posteriormente propõe uma forma de extensão que aplique assinatura digital para proteger o *software*.

5.1 Formas de uso do *ELF*

Dependendo da etapa de compilação executada, o padrão *ELF* pode ser utilizado para dois propósitos:

1. Armazenar o código objeto gerado pela compilação, que pode ser utilizado para a etapa de ligação;
2. Criar um *software* que pode ser executado pelo sistema operacional, tanto como programa, quanto como biblioteca.

Para cada uma destas funções existe um conjunto de estruturas de dados a ser gravada dentro do arquivo.

Existe ainda a possibilidade de gravar ambas as estruturas, criando um arquivo que pode ser utilizado tanto para a leitura de sessões (para uso como biblioteca estática), quanto para a montagem de um executável em memória ou carga de uma biblioteca .

5.2 Uso de um arquivo *ELF* para *linking*

5.2.1 Cabeçalhos de sessão

Todos os dados armazenados dentro de um arquivo *ELF* são organizados em sessões. O termo sessão corresponde a um conjunto de informações dentro do arquivo que tem um registro correspondente na tabela de sessões. Se um arquivo *ELF* contém os cabeçalhos de sessão (figura 5.1), os mesmos podem ser utilizados pelo *linker* como índice para encontrar e manipular os dados armazenados, além de descrever o tipo de dado. Os cabeçalhos de sessão contém uma série de informações, sendo as mais relevantes para este estudo:

- Nome da sessão;
- Tamanho;
- Alinhamento.

Desta forma, é possível para o *linker* recuperar e combinar sessões preservando as suas características, especialmente o alinhamento que cada sessão requer. As informações contidas auxiliam em uma série de tarefas úteis, como o controle de índices utilizando uma

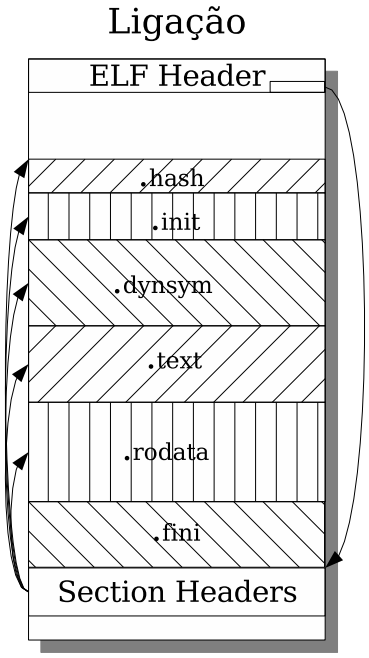

Figura 5.1: Formato *ELF*, destacando as estruturas de *linking*

função de *hash* (função definida pela própria especificação com o objetivo de auxiliar a procura de índices, e não para prover proteção criptográfica), informação sobre ligação de binários, dentre outras. As estruturas utilizadas para o controle das sessões tipicamente ficam no final do arquivo (figura 5.1), em um conjunto chamado tabela de descritores de sessões.

5.3 Uso de um arquivo *ELF* para execução

5.3.1 Cabeçalhos de programas

Durante a carga do *software*, o sistema operacional utiliza o cabeçalho principal do *ELF* para descobrir onde estão as estruturas que descrevem como o *software* deve ser executado (os cabeçalhos de programa), assim como o número de estruturas presentes no arquivo. Estas estruturas ficam no começo do arquivo (figura 5.2), de forma a otimizar a carga, agrupando-as na primeira página do arquivo.

Os cabeçalhos de programas podem conter vários tipos de entradas, com informações a respeito de:

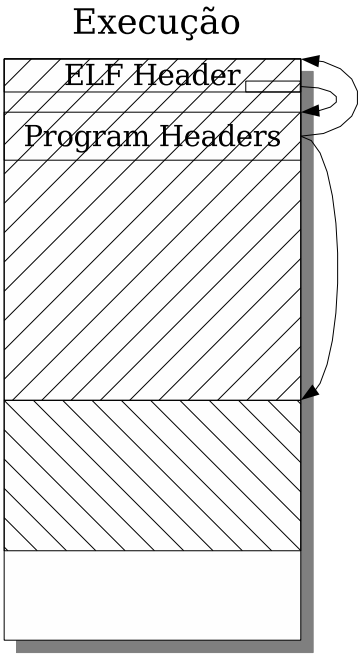

Figura 5.2: Formato *ELF*, destacando as áreas de mapeamento em memória

- Tipo do cabeçalho de programa;
- Tamanho em arquivo;
- Tamanho em memória;
- Se definir uma área em memória, o endereço virtual que a área definida ocupa no espaço de memória do processo;
- Se definir uma área em memória, o alinhamento da área definida;
- *Flags*, que indicam as permissões uso da área mapeada em memória.

O tamanho em memória pode ser maior que o tamanho em arquivo. Isto é necessário para possibilitar a reserva de um espaço na imagem em memória do processo sem o uso de espaço em arquivo. Nestes casos, o padrão *ELF* define que a área reservada e não presente em disco seja iniciada com o valor zero.

Observe que, no caso específico da sessão *bss*, a especificação exige que o espaço truncado na última página contenha zero. Devido ao sistema de arquivos carregar um valor não definido para este espaço, o *hash* desta página deve ser feito de forma diferente. Deve-se calcular o *hash* apenas do espaço utilizado, com uma entrada para o algoritmo de *hash*

menor que uma página. Ignora-se o alinhamento da página inicializado como zero, com conteúdo diferente da página em disco.

O tipo do cabeçalho da sessão identifica a função, sendo que os tipos mais importantes para este estudo são:

- PT_LOAD, que indica que este cabeçalho representa uma área do arquivo que deve ser mapeada em memória;
- PT_DYNAMIC, indicando que este cabeçalho é usado para obter informações a respeito de como ligar dinamicamente este *software*;
- PT_INTERP, que contém o nome do programa ou biblioteca que será responsável por fazer a ligação (*link*) deste *software* com bibliotecas imediatamente antes e durante a execução.

5.4 Relocação de código e uso de um interpretador como *dynamic linker*

Caso o *software* seja um programa que deve ser ligado (*linked*) em tempo de execução, o cabeçalho tipo PT_INTERP contém o nome do programa ou biblioteca que efetua a ligação (*link*) em benefício do programa. As informações utilizadas neste processo estão contidas em uma sessão especial, que tem um cabeçalho de programa do tipo PT_DYNAMIC, indicando a posição da mesma no arquivo.

O mecanismo de ligação (*link*) dinâmica consiste em mapear em memória, além da imagem do processo a ser executado, a imagem de um *software* auxiliar (*dynamic linker*, que pode ser um arquivo de programa ou biblioteca no formato *ELF*), que efetua o mapeamento dos símbolos indefinidos no programa com os símbolos das bibliotecas. Este mapeamento é feito baseado em uma tabela de símbolos, que contém o local onde o programa pode encontrar cada símbolo dentro da área do processo mapeada correspondente às bibliotecas.

Convém ressaltar que para cada processo que usa uma biblioteca, o sistema operacional mantém uma única área de memória mapeada para o arquivo da biblioteca, seja uma área de dados ou de código.

É necessário uma única página física na memória do sistema operacional para todas as

páginas lógicas de uma mesma biblioteca. O sistema operacional somente mantém páginas diferentes no caso das páginas presentes nas áreas de dados que sofreram alterações em um dos processos, em que o mecanismo de *copy on write* é aplicado (conceito apresentado no capítulo 2).

O *dynamic linker* efetua os mapeamentos das bibliotecas que o programa precisa e de todas as bibliotecas que alguma biblioteca possa depender, e transfere o controle do fluxo de execução para o programa de forma transparente.

Quando o programa precisar efetivamente usar parte da biblioteca, é necessário saber em que endereço da memória do processo foram mapeadas as bibliotecas e podem ser encontrados os símbolos dentro das bibliotecas. O compilador cria para o programa uma tabela de símbolos, que deve ser atualizada pelo *dynamic linker* para conectar as suas funções com os pontos de entrada certos nas bibliotecas.

O *dynamic linker* pode efetuar a conexão dos símbolos de duas formas:

1. Efetuar a ligação de todos os símbolos antes de transferir o controle para o programa;

2. Efetuar a ligação somente quando o símbolo for utilizado.

A segunda forma de efetuar a ligação consiste no processo conhecido como *lazy binding* (ligação preguiçosa), em que a mesma é efetuada somente quando o processo utilizar o símbolo pela primeira vez. Se o símbolo não for utilizado, economiza-se o esforço de busca e conexão. Esta forma de funcionamento consiste em deixar na tabela de símbolos por padrão não o símbolo procurado pelo programa, e sim uma entrada que dispara o processo de conexão, transferindo o controle para o *dynamic linker*, que faz a conexão em benefício do processo.

A conexão é feita, a entrada da tabela é atualizada para referir-se ao símbolo, e o processo recebe o símbolo sem perceber que o *dynamic linker* teve que calcular o endereço do símbolo no espaço de memória mapeado para o processo. Como a tabela de símbolos foi atualizada na primeira busca, em todas as buscas subseqüentes o processo acessa diretamente o símbolo, sem acionar o *dynamic linker*.

Este processo de *lazy binding* reduz o esforço de conexão de símbolos, principalmente considerando que tipicamente os processos utilizam apenas uma pequena parte da biblioteca, e caso fosse feita a conexão de todos os símbolos, um grande esforço sempre seria feito desnecessariamente, atrasando a carga dos processos.

Assim como este processo de ligação é otimizado de forma oportunista para efetuar o mínimo esforço, o mecanismo de mapeamento de arquivos em memória e carga sob demanda é outra forma inteligente de funcionamento do sistema.

A carga sob demanda utiliza o mínimo de recursos, possibilitando a construção dos sistemas operacionais modernos, que são pesadamente baseados em bibliotecas.

Fazendo um paralelo entre estes dois processos e a forma de funcionamento proposta, conclui-se que o processo de verificação de assinatura digital oportunista é uma forma de combinar a verificação criptográfica com o sistema de carga sob demanda.

O sistema de verificação oportunista é uma forma otimizada de distribuir a carga (*overhead*) da criptografia, na mesma linha de funcionamento que o mecanismo de carga sob demanda distribui o esforço de *I/O* e que o mecanismo de *lazy binding* distribui o esforço de busca e conexão de símbolos.

O esforço computacional é feito sempre o mais tarde possível, quando necessário, se for efetivamente necessário.

5.4.1 Cabeçalhos de mapeamento de memória

O cabeçalho do tipo PT_LOAD é utilizado para representar um mapeamento entre sessões e a área de memória do processo. O padrão *ELF* permite que várias sessões do arquivo sejam combinadas em uma única área de memória (figura 5.3), reduzindo o número de mapeamentos necessários e possivelmente simplificando o processo de ligação.

O padrão *ELF* evita forçar o alinhamento das sessões com o alinhamento de páginas de duas formas:

1. Permitindo que as sessões mapeadas em memória contenham os próprios cabeçalhos e estruturas de controle do padrão *ELF*;

2. Possibilitando que uma mesma área no arquivo seja mapeada mais de uma vez na imagem do processo.

Ao evitar o alinhamento das sessões com o tamanho da página, evita-se o aumento desnecessário do arquivo com enchimento. O enchimento acontece nos mapeamentos duplicados (figura 5.3), mas como uma mesma página é carregada para ambas as áreas de memória, o esforço de *I/O* não é duplicado, e como o enchimento ocorre no espaço de memória

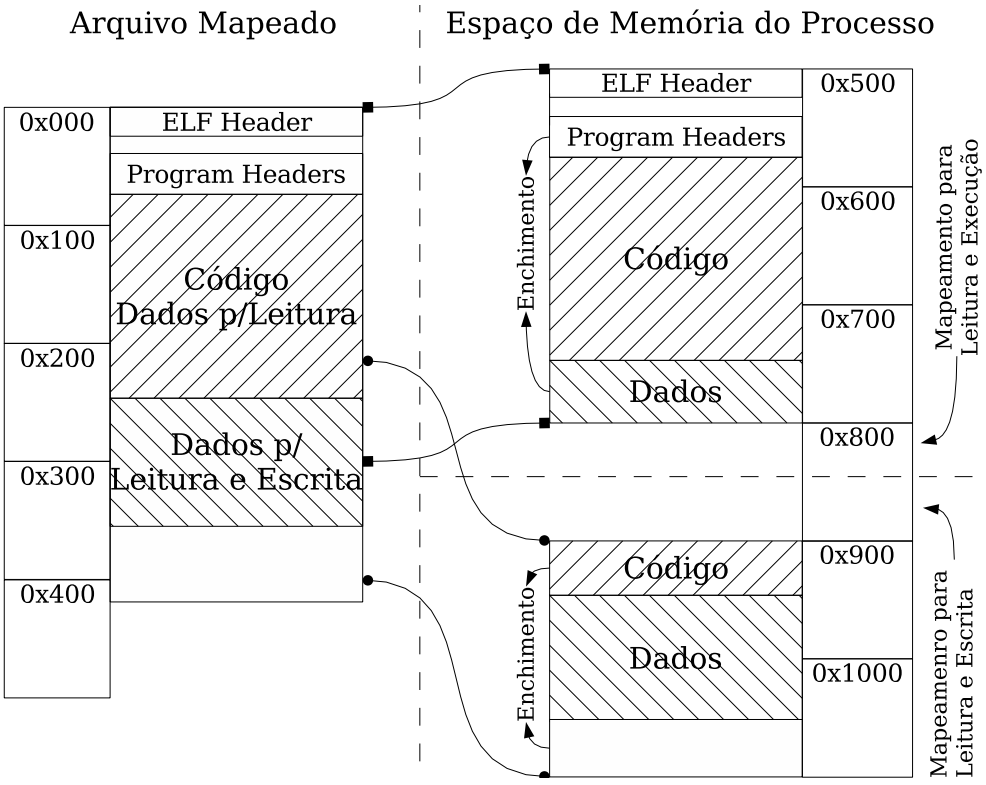

Figura 5.3: Correspondência entre arquivo mapeado em memória e a imagem do processo

de cada processo, a perda de espaço na realidade se limita à perda de um endereço de memória.

Desta forma, cada programa que efetua a montagem do arquivo (*linker*) pode escolher como dispor e combinar as sessões, dependendo das necessidades do sistema operacional ou da aplicação. No caso particular do *linker GNU ld*, utilizado na implementação, existe uma grande variedade possível de configurações, existindo ainda a possibilidade de criar uma configuração própria através de um arquivo que define como o programa deve efetuar a disposição das sessões.

A configuração padrão combina as sessões em apenas duas áreas mapeadas em memória, utilizando dois cabeçalhos de programa do tipo PT_LOAD (figura 5.3), simplificando o trabalho do *linker* dinâmico e com isso otimizando as cargas posteriores do *software*.

5.5 A proposta de extensão do padrão *ELF*

5.5.1 Objetivos da extensão

A extensão do padrão *ELF* tem como objetivo principal possibilitar ao sistema operacional o uso de uma verificação criptográfica do *software* antes de possibilitar ao mesmo o acesso aos recursos do sistema. Além do objetivo principal, existe uma série de requisitos adicionais, que devem ser amparados pela abordagem:

1. A inserção da assinatura digital dentro do arquivo, para facilitar a distribuição pelos desenvolvedores/fabricantes;
2. O uso de criptografia assimétrica, simplificando a gerência do processo pelos administradores do sistema operacional;
3. Utilizar todos os meios disponíveis pelo sistema operacional ou *hardware* para minimizar o impacto da verificação sobre o desempenho do sistema;
4. Possibilitar a compatibilidade com versões anteriores do sistema (sistema *backward compatible*).

No último ítem é especialmente importante a análise sob o ponto de vista da distribuição do sistema operacional para usuários que não desejam, ou não dispõem de um sistema com suporte à extensão.

Suponha que um usuário não deseja utilizar o sistema de criptografia, por exemplo devido ao custo em desempenho *versus* benefício em segurança não ser adequado à aplicação em questão. Se o sistema não for compatível com versões atuais do padrão *ELF*, o usuário precisaria instalar um sistema operacional construído sem assinatura em nenhum arquivo.

Isto causaria um custo de construção e distribuição dobrado para o fabricante do sistema operacional, que deveria disponibilizar dois sistemas.

O mesmo se aplica para desenvolvedores individuais, que deveriam disponibilizar um pacote com suporte a assinatura e outro sem suporte. Isto torna o sistema menos transparente tanto para os usuários quanto para os desenvolvedores e fabricantes de sistemas operacionais.

Com um sistema que tenha compatibilidade com versões anteriores, todo o processo fica mais simples e transparente, bastando para isso o desenvolvedor construir apenas o

pacote com assinatura e o fabricante distribuir o sistema operacional com todos os arquivos assinados, cabendo ao usuário ou administrador apenas ligar o sistema de verificação ou não. Caso não seja utilizado o sistema de assinatura, o único peso para o usuário é o espaço gasto com arquivos ligeiramente maiores.

5.5.2 Mudanças e adições ao padrão *ELF*

A proposta de extensão do protocolo *ELF* apresentada neste trabalho tem como objetivo proteger o *software*, possibilitando ao sistema operacional detetar modificações tanto na futura imagem do processo quanto nas estruturas de controle utilizadas para a carga do mesmo.

Devido ao compromisso de uma extensão que seja compatível com a versão original do padrão *ELF*, todas as alterações possíveis foram feitas para que o padrão estendido respeite o padrão *ELF* original.

A **primeira mudança** do padrão *ELF* estendido para suportar assinaturas digitais consiste em alterar o protocolo pelo qual o sistema operacional passa o controle para o *linker* dinâmico. Pela especificação original, o sistema operacional pode passar o processo com as sessões necessárias para a ligação dinâmica (*dynamic linking*) mapeadas na memória do processo (que o *dynamic linker* compartilha) ou passar para o mesmo um descritor de arquivo do *software* em questão já aberto, para que o *linker* leia ou mapeie as sessões em questão para a memória do processo.

A alteração na especificação restringe o protocolo para passar sempre as sessões previamente mapeadas em memória, impedindo assim que seja possível para o *linker* ler informações diretamente do arquivo, evitando o mecanismo de verificação de *softwares*. Esta mudança tem a vantagem de trabalhar de forma segura e transparente com o *linker* dinâmico, desde que o mesmo trabalhe diretamente na memória do processo.

A **segunda mudança** consiste em incluir dados dentro do arquivo *ELF*. Assim como é necessário que exista um cabeçalho de programa específico para o sistema de relocação e para o sistema de ligação dinâmica, possibilitando a montagem do processo apenas consultando os cabeçalhos de programa, é proposto a adição de um cabeçalho de programa específico. Este cabeçalho indica a posição no arquivo onde está a área que contém as estruturas de controle para a criptografia (figura 5.4). Este cabeçalho de programa adicional, definido como PT_SIGNATURE, pode ter valor definido como 8, que é um número reservado pelo padrão *ELF* para futuras extensões, mas ficando em desacordo com a

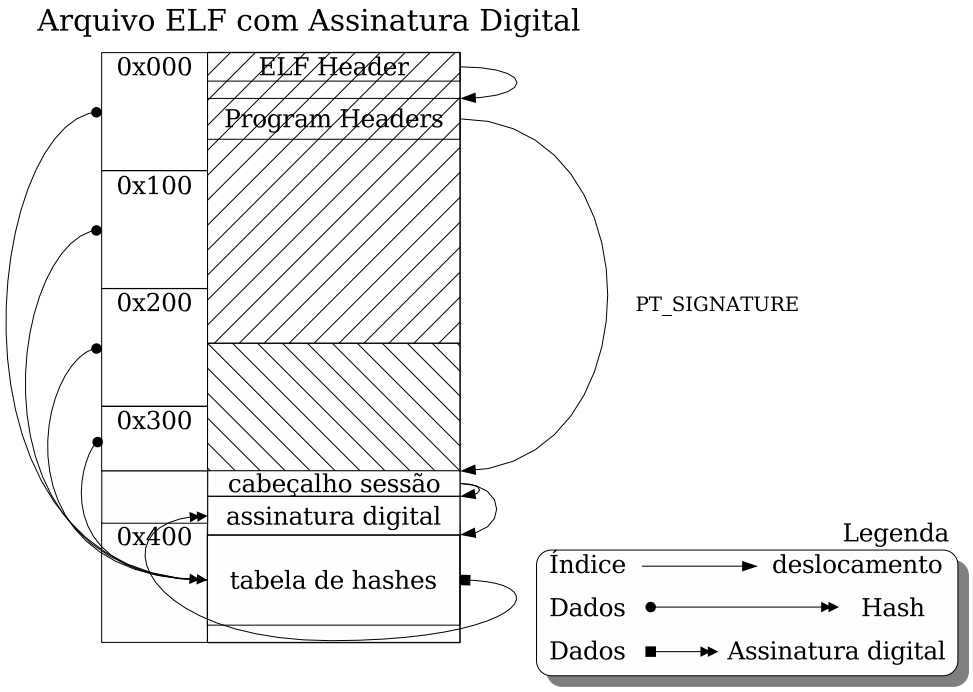

Figura 5.4: Exemplo de um arquivo *ELF* com assinatura digital embutida

especificação atual, ou algum número na faixa entre PT_LOPROC e PT_HIPROC (reservado para uso específico do processador em questão), de forma a ficar completamente compatível com a especificação *ELF* básica.

Os dados de controle da criptografia devem garantir a verificação, tanto dos dados utilizados para a verificação posterior das páginas, quanto a verificação do cabeçalho principal do *ELF* e dos cabeçalhos de programa, porque os cabeçalhos influem na carga do *software*. Desta forma, o sistema proposto efetua uma assinatura digital combinada da área contínua onde se encontra o cabeçalho principal do *ELF*, da área contínua dos cabeçalhos de programa e da área onde se encontra a sessão ".signature".

Esta assinatura digital combinada tem como característica utilizar apenas uma transformada criptográfica assimétrica para assinar mais de um valor de *hash*, aproveitando a diferença de tamanho entre o bloco da criptografia assimétrica e o tamanho da saída da função de *hash* para realizar apenas uma operação de chave pública (que é a transformada criptográfica mais pesada de todo o processo) para assinar três áreas distintas.

Esta área é colocada dentro de uma sessão especial no arquivo, com o nome reservado ".signature". Dentro desta sessão existe um cabeçalho próprio para o mecanismo de assinatura digital, um espaço onde é feita a transformada *RSA*, e um espaço onde é colocada

a informação a respeito da tabela de *hash* das páginas do arquivo (figura 5.5).

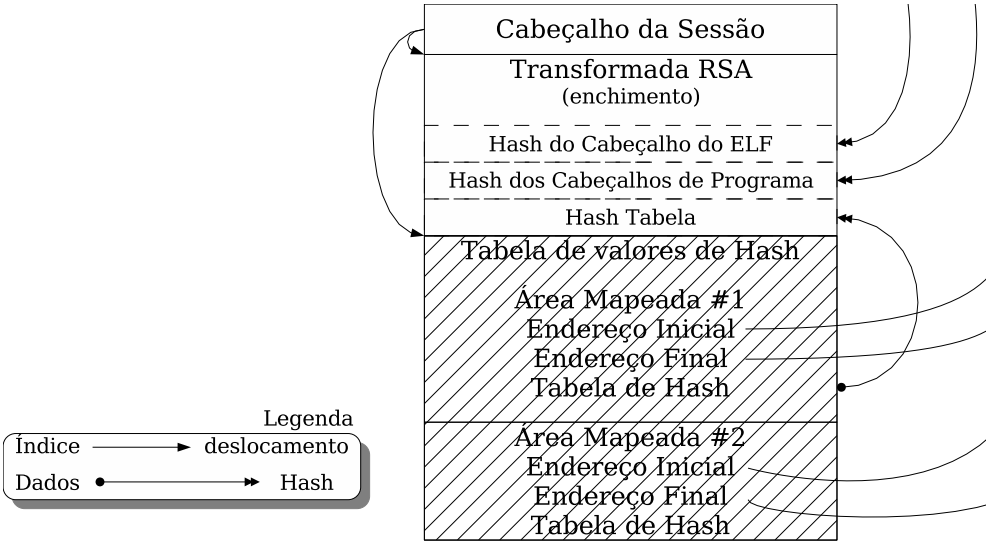

Figura 5.5: Formato da sessão com as estruturas de controle da assinatura digital

Como o cabeçalho da sessão contém apenas o *offset* e o tamanho da assinatura digital e da tabela de *hash* das páginas, qualquer adulteração na mesma implica em uma falha de verificação da assinatura digital. A única forma de adulterar a mesma seria apontar para outro espaço dentro da sessão ".signature", que verifique a assinatura digital, ou que tenha o mesmo valor de *hash* que a própria tabela de *hash*, fato que implicaria na quebra ou da transformada de criptografia assimétrica utilizada, ou da transformada de *hash*, portanto a premissa de que o sistema é tão seguro quanto as transformadas criptográficas aplicadas continua verdadeira.

Este formato de arquivo proposto deve ser obrigatoriamente criado durante a montagem do arquivo do programa ou biblioteca, porque o *hash* das páginas envolve os próprios cabeçalhos do executável, ou seja, a partir do começo do processo de assinatura digital os cabeçalhos não podem ser alterados sem a perda da assinatura.

Existem duas formas de executar este processo:

1. montando a estrutura do arquivo com o formato estendido, e fazendo a assinatura em uma etapa posterior;

2. fazer a assinatura imediatamente após a disposição de sessões, obrigatoriamente durante a montagem do arquivo.

Na implementação realizada, optou-se por adaptar o programa que monta os programas e bibliotecas (*linker GNU ld*) para reservar o espaço, e criar um programa externo para efetuar a assinatura (*elfsign*).

Devido à transparência no processo de compilação, foi possível personalizar de forma simples o processo de geração de pacotes binários *RPM* [Bailey, 1997, Foster-Johnson, 2003] para gerar pacotes já assinados, simplificando a etapa de construção do sistema operacional por parte dos fabricantes/distribuidores de sistemas operacionais.

5.6 Uso da assinatura digital integrada pelo sistema de verificação

A assinatura digital apresentada neste capítulo é gerada pelo programador ou distribuidor de software na etapa de *linking* do programa ou biblioteca. Esta assinatura digital será verificada sob demanda pelo sistema operacional, durante a execução do processo que depende deste componente de *software*, utilizando o mecanismo descrito no capítulo 4.

O suporte à geração e verificação de assinaturas foi implementado no sistema operacional *GNU/Linux*. Este sistema foi submetido a uma série de testes (descritos no capítulo 6), com o objetivo de avaliar o impacto da verificação criptográfica no desempenho do sistema operacional.

Capítulo 6

Impacto no Desempenho do Sistema Operacional

6.1 Objetivo dos testes

A avaliação do impacto no desempenho apresentada a seguir tem como objetivo avaliar a sobrecarga causada pela verificação de assinaturas digitais em um sistema operacional completo. Além do sistema operacional, um grupo de aplicativos foi escolhido para representar padrões de uso similares à realidade da maioria dos microcomputadores utilizados atualmente.

6.2 Hardware

O computador utilizado para a realização dos testes de desempenho foi um PC com um único microprocessador *AMD Athlon^{TM}XP 2400+*, trabalhando a uma freqüência de 1991,61MHz, com um cache de acesso à memória de 256kB. A quantidade de memória disponível para o sistema operacional totaliza 483352Mb, trabalhando a uma freqüência de 333Mhz. O sistema conta com três discos de 40GB cada, com uma taxa de transferência para leitura sem o uso do *buffer* de disco de aproximadamente 38MB/s. O sistema de arquivos está organizado com o sistema operacional em um único disco, utilizando o sistema de arquivos *EXT3*, e os outros dois discos estão sendo utilizados para compor um *RAID* nível 1 (espelhamento simples), dentro do qual está colocado o diretório de usuários, onde apenas o resultado dos testes é armazenado.

6.3 Sistema Operacional

Para realizar a análise foi compilado um sistema operacional *GNU/Linux* baseado na distribuição *Fedora Core 2*, e no *kernel* versão 2.6.7. O *kernel Linux* utiliza páginas de 4KB para arquiteturas derivadas do processador *i386*. O sistema foi completamente recompilado a partir dos fontes, através do sistema de pacotes *RPM* [Bailey, 1997, Foster-Johnson, 2003], e foi instalado em um microcomputador *desktop* padrão. Os pacotes instalam um total de 5237 programas e bibliotecas assinadas digitalmente, incluindo as bibliotecas básicas do sistema operacional.

6.4 Verificação de assinaturas

O mecanismo de verificação de assinaturas foi implementado em um módulo do *kernel*, utilizando uma chave pública fixa.

Observe que o sistema apresentado não protege os módulos do *kernel*. Para tratar desta vulnerabilidade, ou utiliza-se um sistema auxiliar que verifica os módulos, ou desabilita-se completamente o suporte a módulos ou todos os módulos devem ser carregados de um meio seguro durante o *boot* e futuras cargas de módulos devem ser desabilitadas.

A verificação criptográfica foi baseada em uma rotina de transformada criptográfica *RSA* [Menezes et al., 1996] do próprio módulo, e da interface de criptografia (*CryptoAPI*) do próprio *kernel*.

O módulo utiliza uma versão do sistema de segurança *Linux Security Modules (LSM)* [Loscocco and Smelley, 2001, Morris et al., 2002a, Morris et al., 2002b], para efetuar duas chamadas para o módulo, uma quando o *kernel* pergunta para o sistema de segurança se o mapeamento de arquivo em memória é autorizado, e outra, que consiste na adaptação do *LSM* para verificar se é permitido que uma página faça parte da memória de um processo. Esta adaptação envolve a verificação em um processo do *kernel*, evitando que seja efetuado um processamento pesado dentro de uma rotina de atendimento de interrupções. Outra vantagem consiste na possibilidade do processo de verificação ser escalonado, liberando o processador para o resto do sistema, que é conveniente sob o ponto de vista do desempenho. O módulo em questão foi programado para impedir a execução de qualquer binário ou parte do mesmo que não esteja conforme com o padrão de assinaturas, sendo que esta não conformidade pode ocorrer em qualquer das seguintes circunstâncias:

1. Falta de assinatura;
2. Assinatura inválida ou corrompida;
3. Página corrompida;
4. Página carregada fora do escopo da assinatura.

Nos dois primeiros casos, a verificação é feita durante qualquer chamada de mapeamento do programa ou biblioteca. A verificação é feita pelo próprio *kernel* do sistema operacional (no mapeamento inicial do arquivo do programa) ou pelo *dynamic linker* (no mapeamento das bibliotecas dinâmicas), trabalhando em benefício do programa durante a execução inicial do processo.

A assinatura é preservada em memória enquanto existir um mapeamento para o arquivo em questão. Se existe um processo utilizando um arquivo de programa ou biblioteca, todos os outros mapeamentos, tanto para o mesmo processo quanto para processos distintos, utilizarão uma única assinatura digital, economizando recursos para verificar e armazenar a assinatura em memória.

No terceiro caso, a verificação detectou uma modificação em alguma parte do programa ou biblioteca, seja uma área de dados (variáveis com valores padrão, por exemplo) ou uma área que contenha o próprio código binário.

O quarto caso corresponde a uma página que é carregada fora da área protegida por assinatura. Neste caso, assim como no terceiro, o *kernel* marca a página como inválida, como se ocorresse um problema físico que corrompeu a mesma. Caso o processo acesse um endereço correspondente a uma página inválida, o sistema operacional envia um sinal de problema de acesso à memória, definido pelo padrão *POSIX* como *SIGBUS*, cuja ação padrão é terminar o processo e gerar um arquivo com a cópia da memória do processo (*core dump*), e o processo termina.

Observe que o terceiro e quarto casos utilizam a adaptação no *LSM* para efetuar a liberação da página em contexto de usuário.

O funcionamento do módulo envolve, portanto, verificar a assinatura, carregar a mesma para a memória do sistema operacional, e finalmente aguardar a chegada de *page faults*. Ao ocorrer um *page fault*, o sistema operacional verifica se a página em questão faz parte de um processo com assinatura, e se a página está dentro da região protegida pela mesma. Em caso positivo, o sistema operacional calcula o *hash* da página e completa a verificação da assinatura digital para aquela página em particular.

6.4.1 O mecanismo de carga preditiva do *Kernel*

A máquina de memória virtual do sistema operacional *Linux* apresenta uma série de mecanismos para otimizar a velocidade de carga de aplicativos. Como o tempo de acesso a disco é de uma magnitude milhares de vezes mais lenta que a memória principal, tipicamente o processo é bloqueado seqüencialmente esperando por páginas serem carregadas. Além desta diferença de desempenho, os meios de armazenamento utilizados atualmente são baseados em discos rotativos, onde as informações são gravadas por meio ótico ou magnético, em partes específicas chamadas de setores. Os setores são acessados por um dispositivo que posiciona um leitor, com relação ao disco, na posição em que um dado setor vai futuramente passar. O disco rígido então espera o setor passar, lê o mesmo e envia para o sistema operacional. Devido ao sistema operacional tipicamente ler uma série de setores em uma ordem qualquer, o disco posiciona o dispositivo de leitura de forma a ler os setores de forma otimizada sob o ponto de vista de quantidade de dados, ignorando a ordem em que os setores foram pedidos.

Desta forma, os discos rígidos respondem de forma não determinística, e portanto inserem entropia nos testes realizados. Esta característica dos discos é utilizada para alimentar uma reserva de entropia do sistema operacional, que é empregada pelo gerador de números aleatórios, aplicado, dentre outras finalidades, para a geração de chaves de criptografia.

Devido esta diferença brutal de velocidade entre a memória principal e o disco, a máquina de memória virtual do *kernel* implementa um mecanismo chamado de *lookahead* (ou *read-ahead*). Este efetua, junto com cada pedido de leitura de páginas para a memória, a leitura das páginas consecutivas à pagina requisitada, possivelmente antecipando leituras futuras. Desta forma evita-se que um processo espere por operações de *I/O*. O mecanismo conta com uma quantidade de páginas a serem lidas por padrão, e pode alterar esta quantidade dinamicamente dependendo do comportamento do processo. Esta quantidade variável de páginas é chamada de janela de leitura preditiva.

Como efeito ao sistema de assinaturas, esta leitura preditiva faz com que as páginas que irão fazer parte da memória do processo sejam carregadas, verificadas e destravadas antes que o processo acesse o espaço de memória virtual correspondente, evitando uma falta de página e portanto aumentando a velocidade do programa.

Em contrapartida, é comum que o *kernel* ordene a leitura de páginas além das áreas necessárias para o processo. Neste caso, o sistema de assinaturas marca as páginas como

inválidas, mas como as mesmas não serão efetivamente acessadas, o processo não é afetado.

Outro efeito da carga preditiva de páginas é que as páginas podem ser carregadas antes mesmo da verificação de assinatura, que é realizada durante o mapeamento. Quando o sistema operacional lê a primeira página do executável, durante o começo da montagem do processo e antes de efetuar qualquer mapeamento, o mecanismo de *lookahead* já disparou a leitura preditiva de páginas. Desta forma, o módulo do *kernel* não tem como saber se deve verificar uma página que está chegando ou não. Esta informação somente será disponível no futuro.

A solução encontrada foi liberar toda página que não corresponde a uma biblioteca ou programa mapeado em memória com o dispositivo de segurança. Durante o mapeamento, é verificado todas as páginas que já foram carregadas pelo mecanismo de *lookahead*. Como o processo não acessa nenhuma página antes da área ser mapeada (uma tentativa de acesso a uma área de memória não mapeada dispara o mecanismo de exceção do microprocessador), é garantido que páginas não verificadas não são utilizadas pelo processo sem passar pela verificação.

6.5 Forma de medida

A medida do desempenho do sistema depende da forma com que o mesmo é utilizado. Para evitar a dependência de interação humana, que poderia gerar erros nas medidas devido ao comportamento não determinístico do usuário, foi construído um sistema de testes automático, que tenta simular algumas formas de utilização do computador.

O sistema foi programado para ligar com e sem a verificação de assinaturas, de forma alternada, carregando o sistema operacional, efetuando uma série de testes, anexando o resultado de cada teste a um conjunto de dados estatísticos e reiniciando o computador. Em cada reinicialização é feita uma nova bateria de testes.

Após 256 iterações com o uso do sistema de assinaturas intercaladas com 256 iterações sem o uso do sistema, este conjunto foi processado para extrair as informações relevantes a este estudo.

6.5.1 Inicialização do sistema

A primeira etapa do sistema consiste na medida do tempo que o sistema operacional demora para ligar, carregar os serviços básicos como suporte a rede, servidor de impressão, serviço de *log* de eventos, *firewall* e servidor *http*. O tempo foi medido através da interface '/proc', que disponibiliza informações relativas ao *kernel* do sistema operacional como arquivos texto. A informação utilizada foi o número de interrupções do relógio do computador desde que o sistema ligou. Através de uma simples divisão pode-se obter uma medida precisa do tempo que passou desde que o *kernel* do sistema operacional começou a funcionar. As medidas obtidas estão representadas na tabela 6.1 e na figura 6.1. Este

Sobrecarga	Tempo sem verificação		Tempo com verificação	
	Média	Desvio	Média	Desvio
1.19%	37.280s	0.963s	37.722s	0.312s

Tabela 6.1: Resultado dos testes para o tempo de *boot*

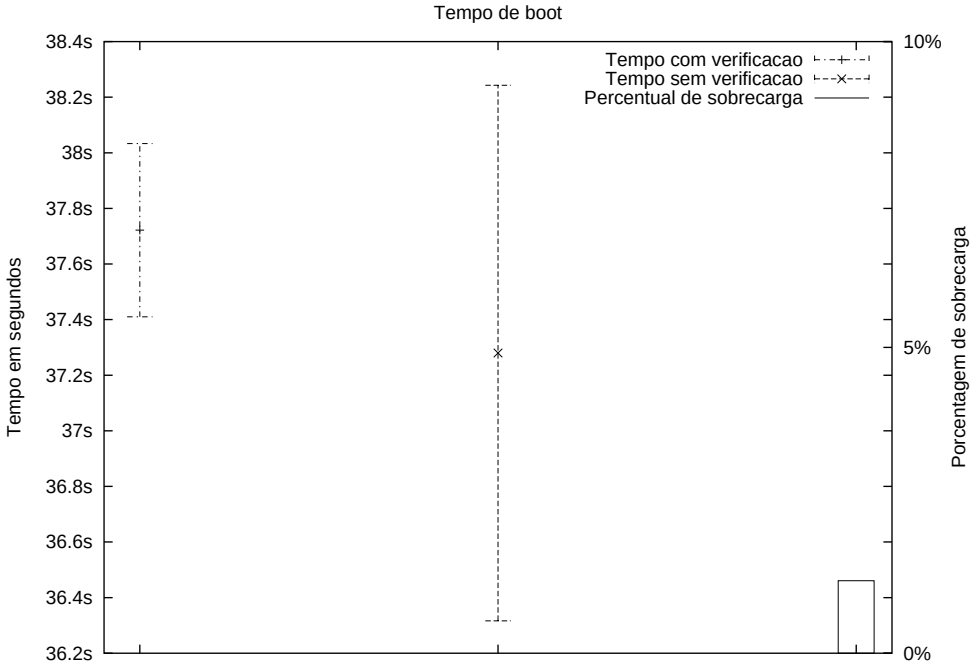

Figura 6.1: Gráfico de impacto sobre a velocidade de *boot* do sistema

gráfico foi construído com o centro no valor correspondente à média dos tempos de inicialização com e sem verificação de assinaturas, e a linha corresponde ao desvio padrão da amostra. O eixo das ordenadas do lado esquerdo do gráfico corresponde ao tempo em segundos, e o eixo das ordenadas no lado direito está relacionado com um gráfico de

barra, com o percentual de sobrecarga. Esta medida corresponde ao atraso esperado que o sistema de verificação de assinaturas acrescenta ao tempo de iniciar o computador e carregar o sistema operacional.

6.5.2 Programas avaliados

Para medir o impacto sobre o tempo de execução e desempenho dos programas, o sistema foi alterado para entrar em um modo de teste ao ser ligado. O primeiro passo consiste medir o tempo de inicialização. Posteriormente, espera-se o servidor gráfico carregar e estar pronto para aceitar conexões de clientes. A partir deste momento, automaticamente o sistema começa a simular a execução de alguns programas, de forma a simular algumas situações específicas.

Todos os programas são executados de forma seqüencial cinco vezes, de forma a avaliar a primeira carga do programa, a qual é especialmente interessante, considerando que as páginas necessárias para a execução do mesmo não estão disponíveis em *cache*. A primeira iteração constitui o pior caso tanto para a máquina de memória virtual (que deve disparar *I/O* para preencher a memória do processo) quanto para o sistema de assinaturas (que deve verificar todas as páginas transferidas).

Observe que para o caso das bibliotecas *libc* e do *dynamic linker*, mesmo na primeira iteração as páginas já estão em *cache* e as estruturas de segurança já foram carregadas, devido ao uso dos mesmo pelos programas que inicializam o sistema operacional e o servidor gráfico.

As iterações posteriores servem apenas para testar e avaliar os casos em que todas as páginas estão em *cache*, e as estruturas de segurança já estão anexadas aos respectivos *inodes* (descritores de arquivo) O *inode* é utilizado pelo *kernel* como suporte para todas as páginas da máquina virtual, e pelo sistema de assinaturas como suporte às estruturas de controle de assinatura digital.

Para o caso de mapeamento anônimo (memória sem arquivo mapeado), o dispositivo de suporte utilizado pelo *kernel* é o espaço de *swap*.

Observe que à medida que as bibliotecas são utilizadas pelo sistema, as mesmas são progressivamente verificadas e passam a fazer parte do *cache* de páginas. O *cache* de assinaturas só funciona caso exista alguma biblioteca utilizada por um programa no momento em que outro programa que dependa da mesma é executado. Portanto, quase todas as

bibliotecas compartilhadas tem a sua assinatura verificada a cada iteração, considerando que neste teste em particular não existe nenhum programa que esteja usando concorrentemente nenhuma biblioteca, exceto o *linker* dinâmico e a bibioteca *libc*.

À medida em que o teste progride, as bibliotecas utilizadas e verificadas pelos aplicativos executados anteriormente são disponibilizadas, a um custo de carga menor, para execuções futuras de outros programas. Desta forma, o teste simula o uso normal e progressivo da máquina de memória virtual do sistema operacional.

Execução do programa *md5sum*

O primeiro programa a ser executado é o *md5sum*, para estudar o impacto em um programa pequeno que depende de um mínimo de bibliotecas. O *md5sum* computa o *hash MD5* de arquivos, com o objetivo de avaliar modificações nos mesmo. O programa é executado para efetuar o *hash MD5* de um arquivo de teste de aproximadamente 5MB, que contém música comprimida. Cinco execuções são realizadas sobre o mesmo arquivo, e o resultado pode ser observado na figura 6.2.

	Tempo Total		CPU Kernel		CPU Usuário		PF Disco		PF Cache		Preemp. Inv.		Preemp. Vol.	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
1	0.299s	0.078s	0.007s	0.005s	0.015s	0.005s	0	0	134	0	6	7	75	16
2	0.030s	0.000s	0.004s	0.005s	0.016s	0.005s	0	0	134	0	1	0	1	0
3	0.030s	0.000s	0.004s	0.005s	0.017s	0.005s	0	0	134	0	1	0	1	0
4	0.030s	0.000s	0.004s	0.005s	0.017s	0.005s	0	0	134	0	1	0	1	0
5	0.030s	0.000s	0.004s	0.005s	0.017s	0.005s	0	0	134	0	1	0	1	0
e 1	0.343s	0.075s	0.009s	0.004s	0.016s	0.005s	0	0	134	0	29	10	76	16
e 2	0.030s	0.000s	0.006s	0.005s	0.017s	0.004s	0	0	134	0	22	4	1	0
e 3	0.030s	0.000s	0.006s	0.005s	0.018s	0.004s	0	0	134	0	23	2	1	0
e 4	0.030s	0.000s	0.006s	0.005s	0.017s	0.004s	0	0	134	0	24	2	1	0
e 5	0.030s	0.000s	0.005s	0.005s	0.017s	0.004s	0	0	134	0	24	2	1	0

Tabela 6.2: Resultado dos testes para o comando md5sum

A tabela 6.2 consiste de uma medida feita através da chamada de sistema *wait3*. As linhas são dispostas começando com o programa executado em um ambiente sem verificação de assinaturas, com as linhas numeradas de 1 a 5, e depois o resultado com verificação de assinaturas são apresentados nas linhas numeradas de e1 a e5.

As colunas apresentam, em pares de média das medidas e desvio padrão, as seguintes informações:

1. Tempo Total: corresponde ao tempo desde o começo da execução do processo até o fim;

2. CPU *Kernel*: significa quanto tempo o *kernel* trabalhou em benefício do processo;
3. CPU Usuário: quanto tempo o processo teve de processador disponível para executar;
4. PF Disco: número de vezes que um processo precisou de uma página em disco para compor seu espaço de memória, e foi escalonado enquanto a página não chegou;
5. PF Cache: número de vezes que um processo precisou de uma página que já estava em *cache* de memória;
6. Preemp. Inv: contagem de quantas vezes o processo foi escalonado porque sua cota de tempo de processador expirou;
7. Preemp. Vol: número de vezes que o processo foi escalonado voluntariamente, tanto por requisitar *I/O*, quanto por outra chamada de sistema.

Observe que o tempo de verificação, por se tratar de um processo do próprio *kernel*, não é adicionado a nenhum dos tempos medido anteriormente (este tempo é medido indiretamente através do tempo total, enquanto o *kernel* verifica as páginas o processo permanece bloqueado). As faltas de página medidas no quarto item correspondem às faltas que o processo gerou, não sendo computadas as geradas pelo mecanismo de *lookahead*.

No caso específico do comando *md5sum*, o processo não causou nenhuma falta de páginas que necessitou de *I/O*. A única forma disto ocorrer consiste na execução do processo disparar o mecanismo de *lookahead*, que carregou todas as páginas que o processo precisou.

Verificando o executável para examinar esta hipótese, observa-se que a imagem do processo é composta de 6 páginas de código e uma de dados. Através da interface */proc*, pode-se confirmar que o processo usa 28kB para código do programa e 1088kB de bibliotecas, considerando os mapeamentos de leitura e execução. Analisando o mecanismo de *lookahead* do *kernel*, percebe-se que a janela padrão de leitura preditiva começa com dez páginas por padrão. Portanto, quando o *kernel* executa o processo, o próprio *kernel* dispara o mecanismo de *lookahead*, que preenche completamente a memória do processo. O mecanismo de assinaturas verifica, neste caso específico, todas as páginas relacionadas com o arquivo de programa apenas no final do mapeamento.

Além do processo não ter efetuado nenhuma falta de página do arquivo do programa, para a hipótese ser verdadeira o processo não pode ter efetuado nenhuma falta de página

relacionada com arquivos de bibliotecas mapeadas. Analisando as dependências de bibliotecas do programa em questão, observa-se que o mesmo está ligado dinamicamente apenas com o *dynamic linker* e com a biblioteca padrão *libc*.

Como todos os outros programas que iniciaram o sistema operacional anteriormente dependem deste par de bibliotecas, as páginas das bibliotecas já estavam verificadas e em *cache*.

Além disto, como os *daemons* em execução mantém este par de bibliotecas mapeadas no momento dos testes, as assinaturas das bibliotecas ficaram em memória, já anexadas ao *inode*, e portanto não foi necessário carregar nenhuma assinatura de biblioteca do disco nem efetuar transformadas criptográficas assimétricas, que correspondem ao processo mais pesado da assinatura digital.

A situação em que um programa depende de uma biblioteca que já está em uso por outro programa é o caso em que o sistema de *cache* da máquina virtual do *kernel* e do sistema de assinaturas obtém a máxima eficiência, porque páginas utilizadas por outro programa tem maior probabilidade de estar em *cache*, evitando o esforço de *I/O* e de *hash* das páginas. Além disto, evita-se a sobrecarga da verificação de assinatura digital da biblioteca, porque a assinatura já está carregada e apoiada sobre o *inode* da biblioteca, o qual é compartilhado com todos os mapeamentos.

O gráfico de desempenho do programa *md5sum* (figura 6.2) é apresenta o número da iteração no eixo das abscissas, e existem linhas representando a variação da média e do desvio padrão ao longo das iterações. Estas linhas são colocadas apenas para facilitar a avaliação da diferença entre iterações. O percentual de sobrecarga está desenhado sobre o eixo das abscissas como barras, e a graduação de sobrecarga encontra-se no eixo das ordenadas do lado direito do gráfico.

Execução do programa *tar*

Seqüencialmente às cinco execuções do programa *md5sum*, o sistema de testes procede a execução do programa *tar* (figura 6.3), que trabalha em conjunto com o programa *bzip2* para empacotar e comprimir o mesmo arquivo de testes utilizado pelo programa *md5sum*.

Os dados coletados, apresentados na tabela 6.3, representam fielmente o tempo total de execução do par de programas, mas os outros dados, inclusive os tempo de *CPU* e *kernel* referem-se exclusivamente ao comando *tar*.

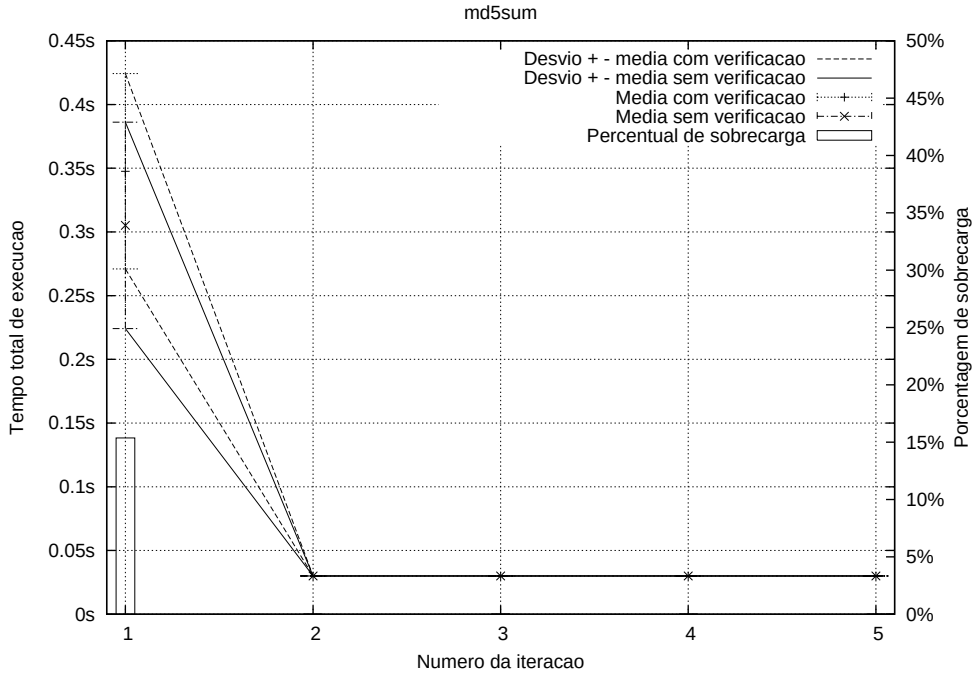

Figura 6.2: Gráfico de impacto sobre a velocidade do programa *md5sum*

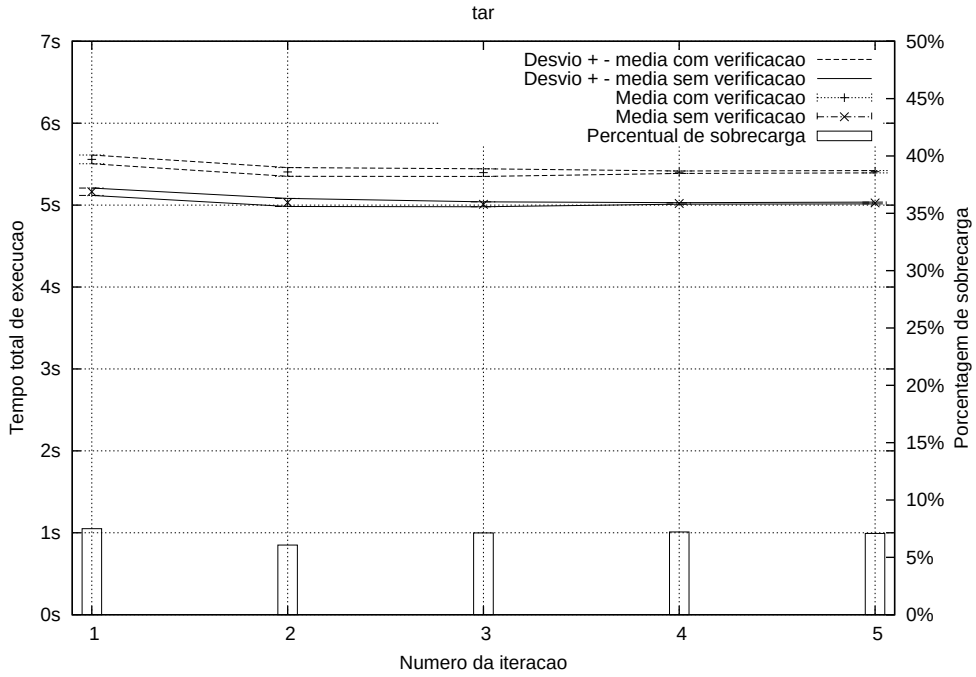

Figura 6.3: Gráfico de impacto sobre a velocidade do programa *tar*

Comparando o comando *tar* ao comando *md5sum*, o *tar* é ligeiramente maior, com um total de 36 páginas de código e 2 de dados, mas assim como o comando *md5sum* somente depende de duas bibliotecas, do *dynamic linker* e da biblioteca padrão *libc*. Através da interface */proc*, descobre-se que o processo usa 144kB para código do programa e 1088kB de bibliotecas (valor idêntico ao do comando *md5sum*), considerando os mapeamentos de leitura e execução. Apesar do comando *tar* depender das mesmas bibliotecas que o comando *md5sum*, o comando *bzip2* depende de uma terceira biblioteca, a *libbz2*, fazendo com que o programa use 1144kB com código executável de bibliotecas. Devido ao fato do programa *bzip2* ser o único que usa naquele instante a biblioteca *libbz2*, a cada iteração o sistema deve verificar a assinatura digital da biblioteca. Devido a este fato, o desempenho das iterações subseqüentes à primeira é menor do que no caso do *md5sum*, em que todas as bibliotecas já estão em uso.

	Tempo Total		CPU Kernel		CPU Usuário		PF Disco		PF Cache		Preemp. Inv.		Preemp. Vol.	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
1	5.164s	0.045s	0.044s	0.007s	4.918s	0.007s	7	0	1949	0	1315	9	1277	4
2	5.032s	0.049s	0.042s	0.006s	4.953s	0.007s	0	0	1956	0	1272	12	1264	12
3	5.010s	0.030s	0.042s	0.006s	4.955s	0.008s	0	0	1956	0	1265	17	1265	18
4	5.021s	0.010s	0.040s	0.007s	4.974s	0.012s	0	0	1956	0	1264	13	1264	13
5	5.026s	0.011s	0.041s	0.006s	4.978s	0.012s	0	0	1956	0	1266	11	1262	11
e 1	5.558s	0.053s	0.059s	0.008s	5.071s	0.025s	5	0	1951	0	4575	48	1277	5
e 2	5.405s	0.053s	0.060s	0.009s	5.087s	0.026s	0	0	1956	0	4710	325	1268	21
e 3	5.397s	0.046s	0.053s	0.008s	5.165s	0.045s	0	0	1956	0	4908	277	1264	22
e 4	5.401s	0.015s	0.048s	0.007s	5.252s	0.053s	0	0	1956	0	5005	36	1262	11
e 5	5.408s	0.014s	0.048s	0.007s	5.259s	0.051s	0	0	1956	0	5012	44	1258	11

Tabela 6.3: Resultado dos testes para o comando tar

Analisando o desempenho do comando *tar* (tabela 6.3), percebe-se que o número de *page faults* que geram *I/O* causado pelo processo é ligeiramente menor para o caso da verificação de assinaturas digitais. Este comportamento, que se repete para os outros programas testados a seguir, pode ser explicada pela hipótese de que o mecanismo de *lookahead* compete com o processo para trazer páginas para a memória do processo. Como o processo passa mais tempo bloqueado se o sistema de verificação de assinaturas estiver ligado, o processo fica um tempo ligeiramente maior esperando as páginas serem verificadas e desbloqueadas, e neste tempo as páginas que o mecanismo de *lookahead* requisitou para *I/O* já chegaram ao *cache*. Desta forma, o acesso do processo gera apenas uma falta de página resolvida com uma página em *cache*.

Teste com o programa *gimp*

Os dois comandos executados anteriormente possibilitam uma análise do desempenho do sistema de assinaturas para executáveis pequenos, que dependem de um mínimo de bibliotecas. Tipicamente, os programas orientados a eventos que utilizam uma *interface* gráfica para interação com usuários dependem pesadamente de bibliotecas, que tendem a ser compartilhadas com vários programas.

Para avaliar o desempenho deste tipo de programa, foi escolhido o comando *gimp*, que é um editor de imagens profissional de código aberto, baseado na biblioteca para criação de *interfaces* gráficas *GTK+*.

	Tempo Total		CPU Kernel		CPU Usuário		PF Disco		PF Cache		Preemp. Inv.		Preemp. Vol.	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
1	8.200s	0.200s	0.323s	0.019s	1.904s	0.019s	150	0.09	8926	3	9953	120	9862	55
2	3.877s	0.023s	0.286s	0.019s	1.906s	0.019s	0	0	9087	1	15054	2094	14197	1962
3	3.834s	0.039s	0.286s	0.017s	1.904s	0.019s	0	0	9087	1	14995	2163	14148	2037
4	3.854s	0.037s	0.285s	0.017s	1.903s	0.019s	0	0	9087	1	14754	2437	13913	2308
5	3.843s	0.033s	0.284s	0.017s	1.907s	0.018s	0	0	9087	0.6	15114	2038	14244	1905
e 1	8.578s	0.224s	0.348s	0.020s	1.933s	0.020s	135	0.2	8925	3	11318	162	9834	129
e 2	4.095s	0.038s	0.308s	0.017s	1.930s	0.024s	0	0	9071	2	11441	2349	9516	2237
e 3	4.065s	0.045s	0.307s	0.019s	1.927s	0.024s	0	0	9071	3	11101	1812	9194	1726
e 4	4.085s	0.068s	0.306s	0.019s	1.921s	0.024s	0	0	9071	2	10807	1242	8930	1171
e 5	4.077s	0.074s	0.307s	0.017s	1.919s	0.023s	0	0	9071	1	10759	1033	8886	974

Tabela 6.4: Resultado dos testes para o comando gimp

O editor de imagem em questão mapeia 629 páginas de código e 69 páginas de dados para a imagem do processo, e depende de 38 bibliotecas dinâmicas. Através da interface */proc*, pode-se confirmar que o processo usa 2516kB para código do programa e 9016kB de bibliotecas, considerando os mapeamentos de leitura e execução.

O programa foi configurado para iniciar e terminar imediatamente, de forma a avaliar o tempo de carga do programa, que constitui o momento crítico, tanto sob a ótica da espera do usuário por uma resposta do sistema, quanto ao esforço da máquina de memória virtual e do sistema de segurança por verificação de assinaturas.

Os resultados observados na tabela 6.4 e na figura 6.4 provam que o sistema continua eficiente mesmo para programas grandes, com utilização intensiva de bibliotecas.

Observe que o impacto sobre a velocidade da carga continua em índices similares aos encontrados em executáveis pequenos. Isto é especialmente importante se comparado com sistemas de verificação de assinaturas baseados na verificação completa do binário antes da execução [Arbaugh et al., 2003] (conforme apresentado no capítulo 2).

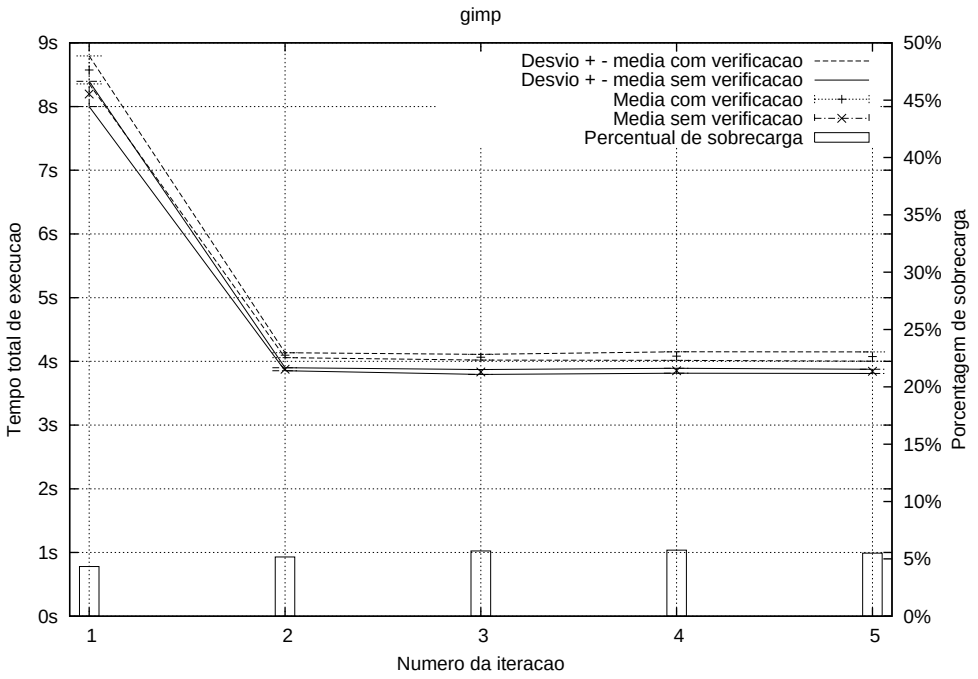

Figura 6.4: Gráfico de impacto sobre a velocidade do editor de imagens *gimp*

Open Office

A verificação de desempenho do pacote *office Open Office* (tabela 6.5 e figura 6.5) tem o mesmo objetivo dos testes realizados com o *gimp*, mas com a diferença que o pacote *office* tem uma imagem em memória ainda maior, e depende de mais bibliotecas.

	Tempo Total		CPU Kernel		CPU Usuário		PF Disco		PF Cache		Preemp. Inv.		Preemp. Vol.	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
1	24.488s	0.233s	0.231s	0.015s	3.973s	0.018s	1581	0.2	17590	2e+01	1400	73	3720	99
2	10.088s	0.382s	0.151s	0.011s	4.017s	0.014s	0	0	19124	3e+01	1763	70	1539	137
3	9.203s	0.227s	0.148s	0.012s	4.024s	0.014s	0	0	19163	2e+01	1819	36	1513	130
4	9.194s	0.228s	0.149s	0.011s	4.025s	0.013s	0	0	19124	3e+01	1824	40	1499	136
5	9.222s	0.249s	0.149s	0.012s	4.024s	0.012s	0	0	19115	2e+01	1816	37	1503	136
e 1	26.126s	0.325s	0.306s	0.014s	3.881s	0.020s	1495	0.06	17500	3e+01	4606	80	3893	110
e 2	10.205s	0.473s	0.170s	0.013s	3.939s	0.026s	0	0	18931	2e+01	3773	144	1475	119
e 3	9.455s	0.256s	0.172s	0.011s	3.966s	0.033s	0	0	18987	3e+01	4074	100	1443	131
e 4	9.562s	0.347s	0.173s	0.013s	3.972s	0.028s	0	0	18930	2e+01	4165	49	1395	146
e 5	9.639s	0.417s	0.174s	0.013s	3.955s	0.032s	0	0	18939	3e+01	4085	100	1388	141

Tabela 6.5: Resultado dos testes para o comando oowriter

O arquivo executável principal do *Open Office* tem 111 páginas de código e 13 páginas de dados apenas. O executável ocupa, com páginas carregadas para a memória, apenas 444kB originário do arquivo de programa. Em contrapartida, depende de 60 bibliotecas dinâmicas que ocupam em páginas físicas na memória 77620kB, apenas para código

executável.

O teste foi realizado através do comando imprimir arquivo, que é equivalente ao tempo de carga do *software*, assim como no teste do editor de imagens *gimp*. O *open office* foi executado para imprimir um arquivo texto vazio e terminar imediatamente, e os resultados estão representados na figura 6.5.

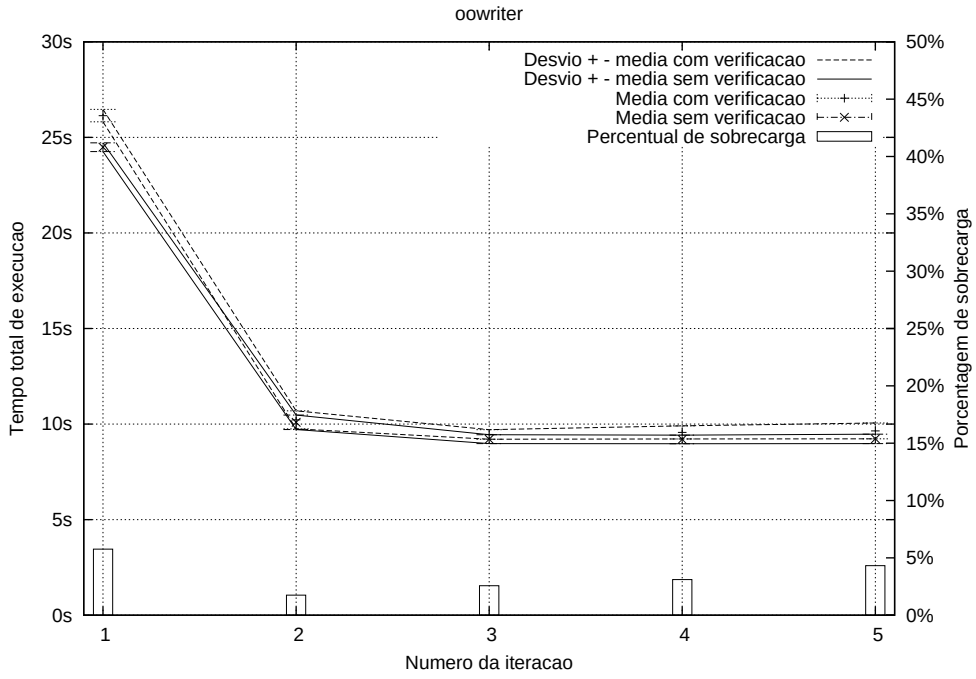

Figura 6.5: Gráfico de impacto sobre a velocidade do programa Open Office

Observe que, apesar do programa utilizar uma quantidade muito grande de memória mapeada de executáveis e gerar um número expressivo de *page faults*, a sobrecarga da verificação de assinaturas se mantém constante, e não apresenta relação com o tamanho do programa utilizado, nem com a quantidade de bibliotecas utilizadas.

6.5.3 Tempo total de execução dos testes

Além dos testes que objetivam analisar a sobrecarga da verificação nos momentos mais críticos de execução de alguns programas, foi medido o tempo que o sistema leva para executar um ciclo completo de testes.

Este tempo está relacionado com o atraso na execução da cada programa, mas o impacto em desempenho é distribuído ao longo da execução dos testes.

O tempo e a sobrecarga são apresentados na tabela 6.6 e na figura 6.6, da mesma forma que o tempo de inicialização do sistema é apresentado.

Sobrecarga	Tempo sem verificação		Tempo com verificação	
	Média	Desvio	Média	Desvio
4.37%	153.773s	1.286s	160.497s	0.952s

Tabela 6.6: Resultado do desempenho para o conjunto de testes

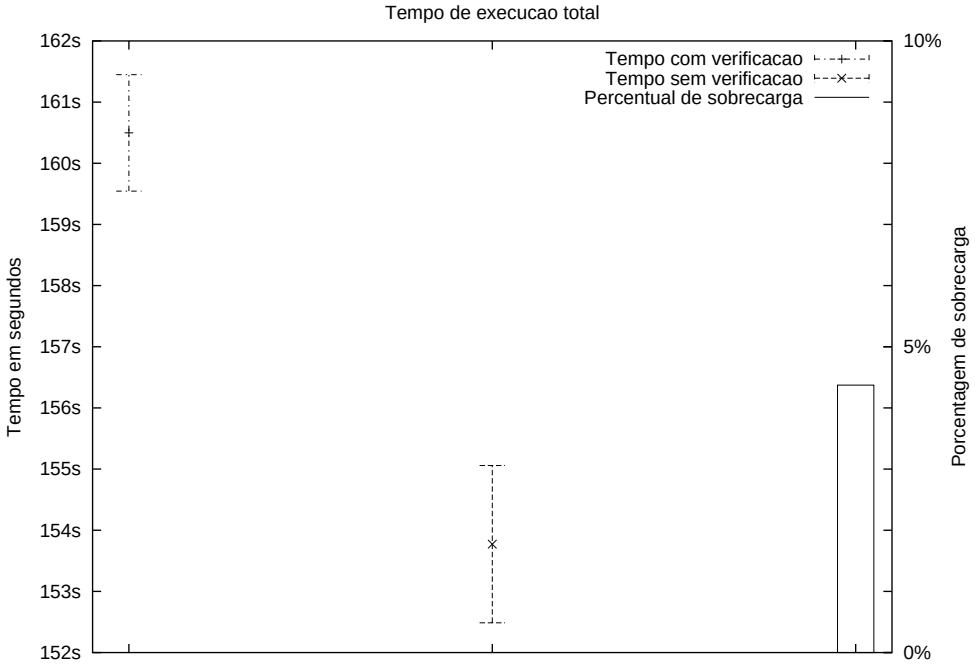

Figura 6.6: Gráfico de impacto sobre o tempo de todo o ciclo de testes

Note que o impacto sobre o desempenho é maior que no caso da inicialização do sistema, porque a sobrecarga dos testes de desempenho é distribuída pelo tempo de inicialização e pelo tempo total dos testes.

Capítulo 7

Conclusões

A implementação e validação do sistema de assinatura digital combinado com o mecanismo de carga sob demanda proposto neste trabalho mostrou que o mesmo é uma forma viável e transparente de aplicar proteção criptográfica a todos os pacotes de um sistema operacional, assim como aos aplicativos instalados.

Esta proteção criptográfica é uma forma de proteger os programas e bibliotecas do sistema contra adulterações. Este sistema deve ser combinado com sistemas de verificação dos *drivers* do *kernel*, e sistemas que proporcionem a verificação de integridade a outros arquivos que podem representar pontos de vulnerabilidade, como por exemplo *scripts* e arquivos de configuração.

Além da integridade, é essencial que o Sistema Operacional tenha mecanismos para prover a privacidade das informações e a disponibilidade dos serviços.

A proposta de extensão do padrão *ELF* utilizada na implementação valida o princípio de transparência do sistema de segurança, evitando que os usuários deixem de utilizar o sistema devido a uma necessidade de intervenção, e simplifica muito o processo de distribuição dos pacotes que compõe o sistema operacional, tanto sob o ponto de vista da instalação do sistema como sob o ponto de vista de atualizações posteriores. Esta implementação não contempla o uso de hierarquia de chaves públicas, mas pode ser estendido para comportar este uso. Outro ponto positivo consiste na simplicidade para o desenvolvedor assinar programas e bibliotecas e fazer a distribuição destes *softwares*.

A verificação por página carregada para a memória do sistema garante que a verificação será feita o mais tarde possível, impossibilitando a corrupção e comprometimento de um arquivo em disco. Observe que o sistema proposto não protege de outros tipos de ataques,

como estouros de *buffer* ou adulterações da memória do processo.

A integração com o padrão *ELF*, utilizado pelo sistema operacional *Linux*, garante que os arquivos gerados pelo sistema de assinatura digital sejam compatíveis com sistemas operacionais de versões anteriores, possibilitando a distribuição de um sistema operacional completamente assinado, que pode ser utilizado sem a verificação criptográfica se for conveniente para o usuário.

Como os arquivos são os mesmos para uso com verificação de assinaturas ou sem, o fabricante pode optar por fornecer o sistema completamente assinado sempre, com um aumento muito pequeno do tamanho do sistema, evitando um custo maior de montagem e distribuição. Este procedimento pode ser aplicado por um desenvolvedor individual de forma semelhante.

Os testes de desempenho evidenciam que a sobrecarga varia dependendo do tipo de aplicativo e da utilização do mesmo, tipicamente ficando em torno de 5% de sobrecarga, embora em um único caso (na primeira iteração do menor programa, o *md5sum*, capítulo 6) esta sobrecarga medida chegou a 15%.

Capítulo 8

Trabalhos Futuros

8.1 Usos alternativos para o sistema

Este sistema de segurança pode ser utilizado para construir um sistema operacional robusto. O mecanismo de verificação impede que um *software* adulterado seja executado, garantindo que nenhum *software* adulterado no sistema de arquivos possa ganhar acesso ao sistema operacional. Devido ao ponto em que é feita a verificação de integridade, é possível garantir que nenhum corrompimento de *softwares* no sistema de arquivos passe desapercebido.

Pode-se construir um sistema que apenas verifica que uma invasão está ocorrendo, com o objetivo de analisar como o invasor utiliza *softwares* do sistema para ganhar privilégios.

8.2 Uso de outras formas matemáticas de verificação

O uso de criptografia assimétrica garante a autenticidade e integridade de um dado código objeto, provando que um dado *software* do sistema foi desenvolvido por um dado indivíduo, mas não que este código não contenha alguma característica indesejável ou danosa ao sistema. O uso de métodos para garantir que um *software* não é nocivo pode ser desenvolvido como uma extensão ao trabalho atual, com a aplicação de algumas idéias de trabalhos apresentados no capítulo 2, como as provas de segurança.

8.3 Estudo do modelo de verificação preditiva

Um estudo posterior de grande relevância consiste numa análise de desempenho do sistema operacional para métodos alternativos de verificação preditiva. Estes métodos, que foram comentados no capítulo 4, devem ser comparados entre si para estabelecer tanto a forma mais eficiente de verificação preditiva quanto a correlação ou não da eficiência dos mecanismos para a aplicação tanto sob os aspectos de gerenciamento de memória quanto sob o aspecto da verificação criptográfica.

8.4 Estudo do problema de inversão de prioridades

Outra linha de estudos subseqüente consiste na pesquisa do problema de inversão de prioridades (descrito no capítulo 4), e a eficiência da solução proposta neste trabalho, assim como soluções alternativas e seus respectivos impactos.

8.5 Estudo de um padrão expansível de controle de chaves e assinatura digital

A validação deste trabalho foi realizada em um sistema de assinatura digital que trabalha com algoritmos fixos para compor a assinatura digital, tanto em função do algoritmo usado quanto pelo tamanho das chaves. Uma extensão interessante para este trabalho seria o estudo de uma forma de possibilitar o uso de várias transformadas criptográficas diferentes, assim como vários tamanhos de chaves. O sistema seria então configurado para um nível mínimo de segurança, tanto pelo tamanho das chaves, quanto pela segurança do algoritmo usado. Este estudo deve considerar a possibilidade de aplicar certificados de revogação, e a gerência dos mesmos pelo sistema operacional.

Referências Bibliográficas

[Albers, 1993] Albers, S. (1993). The influence of lookahead in competitive paging algorithms (extended abstract). In *European Symposium on Algorithms*, pages 1–12.

[Apvrille et al., 2004] Apvrille, A., Gordon, D., Hallyn, S., Pourzandi, M., and Roy, V. (2004). Digsig: Run-time authentication of binaries at kernel level. *LISA XVIII*, pages 59–66.

[Arbaugh et al., 2003] Arbaugh, W. A., van Doorn, L., and Ballintijn, G. (2003). Signed executables for linux. http://www.cs.umd.edu/%7Ewaa/pubs/cs4259.ps.

[Bailey, 1997] Bailey, E. (1997). *Maximum RPM*. MacMillan Publishing Company.

[Bellovin et al., 2002] Bellovin, S. M., Cohen, C., Havrilla, J., Hernan, S., King, B., Lanza, J., Pesante, L., and Pethia, R. (2002). Results of the security in activex workshop. CERT Website. http://www.cert.org/reports/activeX_report.pdf.

[Borchardt and Maziero, 2001a] Borchardt, M. A. and Maziero, C. A. (2001a). Uma arquitetura para a autenticação dinâmica de arquivos. In *III Simpósio de Segurança em Informática*, volume 1 of *Anais do Simpósio de Segurança da Informação 2001*, pages 101–108, Rua Imaculada Conceição, 1155 - Prado Velho CEP 80215-901, Curitiba, PR, Brazil. Pontifícia Universidade Católica do Paraná.

[Borchardt and Maziero, 2001b] Borchardt, M. A. and Maziero, C. A. (2001b). Verificação da integridade de arquivos no kernel do sistema operacional. In *Workshop em Segurança de Sistemas Computacionais*, volume 1 of *Anais do Workshop em Segurança de Sistemas Computacionais*, pages 31–36, Rua Imaculada Conceição, 1155 - Prado Velho CEP 80215-901, Curitiba, PR, Brazil. Pontifícia Universidade Católica do Paraná, Sociedade Brasileira de Computação.

[Borchardt et al., 2003] Borchardt, M. A., Maziero, C. A., and Jamhour, E. (2003). An architecture for on-the-fly file integrity checking. In *1ST Latin American Symposium on Dependable Computing*, volume 2847 of *Lecture Notes in Computer Science*, pages 117–126, Rua Imaculada Conceição, 1155 - Prado Velho CEP 80215-901, Curitiba, PR, Brazil. Pontifícia Universidade Católica do Paraná, Springer Verlag.

[Catuogno and Visconti, 2002] Catuogno, L. and Visconti, I. (2002). A format-independent architecture for run-time integrity checking of executable code. In Springer-Verlag, editor, *Lecture Notes in Computer Science Vol. 2576*, Third Conference on Security in Communication Networks, pages 219–233, Via S. Allende, 84081 Baronissi (SA), Italy. Dipartimento di Informatica ed Applicazioni, Università di Salerno, Academic Press.

[Copeland et al., 1999] Copeland, M., Grahn, J., and Wheeler, D. A. (1999). *The GNU Privacy Handbook*. Free Software Foundation.

[Devanbu, 1999] Devanbu, P. (1999). The ultimate reuse nightmare: Honey, i got the wrong dll. *ACM SSR'99 Panel Statement*. http://citeseer.nj.nec.com/devanbu99ultimate.html.

[Devanbu and Stubblebine, 1997] Devanbu, P. and Stubblebine, S. (1997). Research directions for automated software verification: Using trusted hardware. In *12th IEEE Int'l Conference on Automated Software Engineering – ASE'97*, Florham Park, NJ 07932, USA. IEEE Computer Society. http://citeseer.nj.nec.com/devanbu97research.html.

[Foster-Johnson, 2003] Foster-Johnson, E. (2003). *Red Hat RPM Guide*. John Wiley and Sons, 1st edition.

[Fritzinger and Mueller, 1996] Fritzinger, J. S. and Mueller, M. (1996). Java security. http://java.sun.com/security/whitepaper.ps.

[Garfinkel, 1994] Garfinkel, S. (1994). *PGP : Pretty Good Privacy*. O'Reilly and Associates.

[Ghosh and Voas, 1999] Ghosh, A. K. and Voas, J. M. (1999). Inoculating software for survivability. *Communications of the ACM*, 42(7):38–44. http://citeseer.nj.nec.com/ghosh99inoculating.html.

[Intel, 2003a] Intel (2003a). *IA32 Intel Architecture Software Developer's Manual - Basic Architecture*, volume 1. Intel.

[Intel, 2003b] Intel (2003b). *IA32 Intel Architecture Software Developer's Manual - Instruction Set Reference*, volume 2. Intel.

[Intel, 2003c] Intel (2003c). *IA32 Intel Architecture Software Developer's Manual - System Programming Guide*, volume 3. Intel.

[ITU, 2000] ITU, I. T. U. (2000). *X.509*. Series X: Data Networks and Open System Communications, The Directory: Public-key and attribute certificate frameworks. International Telecommunication Union ITU-T. http://www.itu.int/.

[Kamel et al., 1998] Kamel, M., Keast, J. P. D., and Pal, C. (1998). Concrete architecture of the linux kernel. Technical report, University of Waterloo, Waterloo, Ontario, N2L 3G1. Department of Electrical and Computer Engineering, Department of Computer Science.

[Kim, 2002] Kim, G. (2002). Advanced applications of tripwire for servers: Detecting intrusions, rootkits and more... Technical report, 326 SW Brodway, 3rd Floor Portland, Oregon 97205 USA. http://www.tripwire.com.

[Lee and Kim, 1999] Lee, B. and Kim, K. (1999). Software protection using public key infraestruture. SCI'99 The 1999 Symposium on Criptography and Information Security, Kobe, Japan.

[Loscocco and Smelley, 2001] Loscocco, P. and Smelley, S. (2001). Integrating flexible support for security policies into the linux operating system. http://www.nsa.gov/selinux/papers/freenix01/freenix01.html.

[MacDonald, 1998] MacDonald, J. (1998). On program security and obfuscation. http://citeseer.nj.nec.com/macdonald98program.html.

[Malkhi and Reiter, 2000] Malkhi, D. and Reiter, M. K. (2000). Secure execution of java applets using a remote playground. volume 26, pages 1197–1209.

[Menezes et al., 1996] Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. (1996). *Handbook of Applied Cryptography*. CRC Press, fifth edition. ISBN: 0-8493-8523-7.

[Microsoft, 2003] Microsoft (2003). *Introduction of Code Signing*. Microsoft. http://msdn.microsoft.com/workshop/security/authcode/intro_authenticode.asp.

[Morris et al., 2002a] Morris, J., Kroah-Hartman, G., Wright, C., Cowan, C., and Smalley, S. (2002a). Linux security module framework. http://www.intercode.com.au/jmorris/lsm-ols2002-html/.

[Morris et al., 2002b] Morris, J., Kroah-Hartman, G., Wright, C., Cowan, C., and Smalley, S. (2002b). Linux security modules: General security support for the linux kernel. In USENIX, editor, *11th USENIX Security Symposium*, San Francisco, CA, USA. USENIX. http://www.usenix.org/events/sec02/full_papers/wright/wright_html/index.html.

[Necula and Lee, 1997] Necula, G. C. and Lee, P. (1997). Proof-carrying code. In *24th Annual ACM Sigplan-SIGACT Symposium on Principles of Programming Languages (POPL 97)*, Pittsburgh, PA 15213. Carnegie Mellon University, School of Computer Science.

[Patil et al., 2004] Patil, S., Kashyap, A., Sivathanu, G., and Zadok, E. (2004). I3fs: An in-kernel integrity checker and intrusion detection file system. In *18th USENIX Large Installation System Administration Conference*. Stony Brook University, LISA.

[Romans and Ratliff, 2001] Romans, R. and Ratliff, E. (2001). Linux security state of the union. Technical report, IBM Linux Technology Center.

[Rosu and Segerlind, 1999] Rosu, G. and Segerlind, N. (1999). Proofs on safety for untrusted code. Technical Report CS1999-0633, University of California at San Diego. http://citeseer.nj.nec.com/501819.html.

[Shin et al., 2001] Shin, I., Chander, A., and Mitchell, J. (2001). Mobile code security by java bytecode instrumentation. In IEEE, editor, *DISCEX '01. Proceedings*, DARPA Information Survivability Conference and Exposition II. IEEE.

[Thimbleby et al., 1998] Thimbleby, H., Anderson, S., and Cairns, P. (1998). A framework for modeling trojans and computer virus infection. *Computer Journal*, 41(7):444–458. http://www3.oup.co.uk/computer_journal/hdb/Volume_41/Issue_07/Thimbleby.pdf.

[TIS, 1993] TIS, T. I. S. C. (1993). *Executable and Linkable Format*. Application Binary Interface Version 1.1. http://www.cs.princeton.edu/courses/archive/spring05/cos217/reading/elf.pdf.

[Tripwire, 2002] Tripwire (2002). Ensuring integrity and trustworthiness of electronic clinical data. Technical report, 326 SW Brodway, 3rd Floor Portland, Oregon 97205 USA. http://www.tripwire.com.

[Weber, 2000] Weber, A. (2000). Full bindingness and confidentiality. In *Requirements for Secure Computers, and Design Options*. University of Freiburg, Germany, ECIS. http://citeseer.nj.nec.com/457223.html.

[Wilhelm, 1997] Wilhelm, U. G. (1997). Criptographically protected objects. Technical report, Ecole Polytechnique Fédérale de Lausanne, CH-1015 Lausanne (Switzerland). http://citeseer.nj.nec.com/74363.html.

[Williams, 2002] Williams, M. A. (2002). Anti-trojan and trojan detection with in-kernel digital signature testing of executables. http://www.trojanproof.org/.

[Zovi, 2001] Zovi, D. D. (2001). Kernel rootkits. *SANS Institute*. http://www.sans.org/rr/whitepapers/threats/449.php.