UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Um Verificador Seguro de Integridade de Arquivos

por

VINÍCIUS DA SILVEIRA SERAFIM

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Raul Fernando Weber
Orientador

Porto Alegre, março de 2002.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Serafim, Vinícius da Silveira
Um Verificador Seguro de Integridade de Arquivos / por Vinícius da Silveira Serafim – Porto Alegre: PPGC da UFRGS, 2002.
97f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2002. Orientador: Weber, Raul Fernando.
1.Segurança. 2.Integridade de arquivos. 3.Sistemas operacionais. 4.Linux. I. Weber, Raul Fernando. II. Título.

Agradecimentos

Agradeço a todos aqueles que de alguma forma contribuíram no desenvolvimento desse trabalho, em especial:

Aos amigos e revisores (coitados): Rafael, César (Japa) e Everaldo.

Aos amigos: Betemps, Cláudio, Eidy e Cris, Filipe (Portuga), Herrmann, João, Lichtler (Shmoo), Renato (o quebra ossos) e Vinícius.

Aos amigos e funcionários: Astrogildo, Elgio, Eliane, Eliene, Janaína, Leandro, Luis Otávio, Marion, e todos os outros que encontro no meu dia-a-dia aqui no Instituto.

À professora Taisy Weber, pelos livros que muito me foram úteis neste trabalho.

Ao professor João Netto, pelas dicas, mais do que suficientes, para as medidas de desempenho.

À Nica e ao Sérgio, pelo apoio incondicional em todos os momentos necessários, pela amizade e pelo carinho.

À minha família pelo amor e carinho ilimitados.

Ao meu orientador e amigo, Raul Fernando Weber, pelo constante interesse e pelo extremo envolvimento na realização deste trabalho.

Ao CNPq que, através do programa de bolsas de mestrado, tornou possível a realização deste trabalho.

Enfim, a todos os funcionários e professores do Instituto de Informática da UFRGS e a tantos outros dignos de meu agradecimento que não foram aqui citados.

Dedico este trabalho à minha esposa,
pelo amor, carinho e dedicação presentes
em cada um de seus atos e palavras.

Sumário

Lista de Abreviaturas e Siglas

AIDE	Advanced Intrusion Detection Environment
CNPq	Conselho Nacional de Desenvolvimento Científico e Tecnológico
CRC	Cyclic Redundancy Check
DAC	Discretionary Access Control
FS	File System
GPL	General Public License
HL	Hash List
LIDS	Linux Intrusion Detection System
LS	Leituras Seqüenciais
MAC	Message Authentication Code
MD4	Message Digest 4
MD5	Message Digest 5
MDC	Modification Detection Code
PL	Primeiras Leituras
RVFC	Recently Verified Files Cache
SACM	Standard Access Control Mechanisms
SHA1	Secure Hash Algorithm 1
SOFFIC	Secure On-the-Fly File Integrity Checker
TFL	Trusted Files List
VFS	Virtual File System

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho apresenta um modelo de mecanismo para o controle de integridade de arquivos em sistemas operacionais, com a capacidade de bloquear o acesso à arquivos inválidos de forma a conter os danos causados por possíveis ataques. O modelo foi inicialmente formulado a partir do estudo detalhado do processo de controle de integridade, revelando diversos pontos críticos de segurança nele existentes, e da avaliação dos mecanismos atualmente implementados nos sistemas operacionais que oferecem, mesmo que indiretamente, algum tipo de garantia de integridade dos arquivos do sistema.

Durante o seu desenvolvimento, a segurança do próprio modelo foi detalhadamente analisada de forma a enumerar os seus pontos críticos e possíveis soluções a serem empregadas, resultando na definição dos requisitos mínimos de segurança que devem ser encontrados nesse tipo de mecanismo. Em seguida, visando a validação do modelo especificado e decorrente disponibilização do mecanismo para uso prático e em estudos futuros, um protótipo foi implementado no sistema operacional GNU/Linux. Procurando confirmar a sua viabilidade, foram realizadas análises do impacto causado sobre o desempenho do sistema.

Por fim, foi confirmada a importância e viabilidade do emprego do modelo apresentado como mecanismo adicional de segurança em sistemas operacionais. Além disso, foi colocado em evidência um campo de pesquisa ainda pouco explorado e portanto atrativo para a realização de novos estudos.

Palavras-chave: segurança, integridade de arquivos, sistemas operacionais, Linux.

TITLE: “A SECURE FILE INTEGRITY CHECKER”

Abstract

This study proposes a model for a file integrity control mechanism, applied to operating systems, that is able to prevent the use of invalid files and to provide containment of damages arisen from possible attacks. The model was primarily defined based on detailed studies on the process of integrity control, showing up many security critical points on it, and on the assessment of the existent mechanisms implemented in operating systems which provides some kind of file integrity assurance.

During its development, the security of the model itself was carefully analyzed to enumerate the critical points and possible solutions, resulting in the definition of the minimal security requirements that must be found in this kind of mechanism. Next, aimed to validate the proposed model and to the decurrent mechanism availability for practical use and for future studies, a prototype was implemented in the GNU/Linux operating system. To confirm its validity, the impact over the system performance was analyzed.

As a result of this study, the significance and feasibility of the model’s use as an additional mechanism for operating systems security was confirmed. In addition, a weak explored research field, and therefore attractive for new studies, was exposed.

Keywords: security, file integrity, operating systems, Linux.

1 Introdução

Segurança não é uma palavra recente no contexto dos sistemas computacionais, haja visto que o desenvolvimento dos computadores foi substancialmente impulsionado pela necessidade de algumas nações garantirem sua própria segurança. Exemplo concreto disso foi a criação e emprego de computadores durante a segunda guerra mundial.

Desde então os computadores foram sendo aperfeiçoados e produzidos em maior número, mas ainda se mantiveram como recursos de custo muito elevado e portanto utilizados por um grupo muito restrito de usuários. Entre esses grupos, encontravam-se apenas alguns centros de pesquisa ligados direta ou indiretamente às forças armadas de alguns poucos países, principalmente dos Estados Unidos. Nessa época os computadores ainda eram utilizados de forma isolada, ou seja, não era possível acessá-los de outro lugar senão a partir de seu console. Dessa forma, o computador e seus recursos ficavam fisicamente isolados e o número de operadores – aqueles que realmente acessavam o console – era bastante reduzido. Isso possibilitava que o controle de acesso a esses recursos pudesse ser facilmente realizado através de mecanismos físicos de segurança já bastante comuns, como por exemplo: vigias, cartões de identificação, trancas e chaves. Nesse cenário um atacante teria que invadir fisicamente as instalações onde o computador se encontrava para obter acesso aos seus recursos e informações através dele disponibilizados.

Na década de sessenta, ainda nos Estados Unidos, a presença dos computadores em centros de pesquisa, tanto militares quanto acadêmicos, tornou-se de certa forma comum. Mas ainda devido ao seu alto custo, os recursos computacionais não eram distribuídos de forma homogênea, resultando na sua centralização em alguns poucos locais. Pelo mesmo motivo causador da centralização, era desejável que o nível de utilização desses recursos se mantivesse o mais alto possível durante todo o tempo, desencadeando assim o desenvolvimento de diversas tecnologias para compartilhamento desses recursos. Entre essas tecnologias estão o timesharing – permitindo o uso de um computador por mais de um usuário através de terminais – e as redes – possibilitando o compartilhamento de recursos entre computadores.

À medida que essas tecnologias eram desenvolvidas, a eficácia dos mecanismos físicos de controle de acesso era colocada a prova e cada vez mais esses mecanismos se mostravam insuficientes para garantir a segurança do computador, de seus recursos e das informações nele contidas. Inicialmente essas tecnologias ainda eram utilizadas dentro de ambientes fisicamente controlados, onde o emprego de mecanismos lógicos de controle de acesso já se fazia necessário para evitar que usuários causassem, de forma intencional ou não, danos aos dados e à processos que não eram de sua propriedade. É importante notar que, nesses ambientes, não obstante o uso dos mecanismos lógicos, a segurança era ainda substancialmente implementada através dos mecanismos físicos.

No final da década de sessenta, isso veio a mudar com o projeto Advanced Research Projects Agency, do Departamento de Defesa dos Estados Unidos, mais tarde conhecido como DARPA. Esse projeto teve como seu início o financiamento da criação de uma rede de longa distância interligando vários centros de pesquisa, a ARPANet.

Agora não mais era necessário que o usuário tivesse acesso físico ao console do computador nem a um dos seus terminais para fazer uso de seus recursos, bastava utilizar um terminal remoto em qualquer computador conectado à ARPANet. Nesse momento os mecanismos lógicos de controle de acesso passaram a desempenhar papel de grande importância na segurança dos sistemas computacionais.

Esse cenário não se manteve imutável ao longo do tempo. Ao contrário, o desenvolvimento de novas tecnologias de rede, como o TCP/IP na década de oitenta, e o relativo barateamento dos computadores, possibilitaram a interligação de diversas redes locais de computadores resultando em um aumento significativo da disponibilidade do acesso à recursos computacionais para grande parte da comunidade acadêmica nos Estados Unidos. Essa foi a origem da Internet, rede acadêmica com controle descentralizado e tendo como maior objetivo o compartilhamento de recursos e informações. Ao longo da década de noventa a Internet teve o seu maior período de expansão, deixando de ser uma rede confinada aos limites políticos dos Estados Unidos para ser uma rede com alcance mundial.

Atualmente, com a ampla disponibilidade da Internet e dos computadores, esses recursos são empregados nos mais diversos setores da sociedade com o objetivo de fornecer, relacionar e trocar informações, tornando-a altamente dependente do bom funcionamento deles. Nesse novo e atual ambiente, onde não existem limites físicos e muito menos um ponto de controle centralizado de acesso, a deficiência dos mecanismos físicos de segurança ficou definitivamente evidenciada, tornando o emprego de mecanismos lógicos uma exigência em qualquer sistema computacional.

1.1 Atributos de Segurança

Não obstante a preocupação com segurança desde o advento do computador eletrônico, ela foi de certa forma negligenciada, resultando daí grande parte dos problemas hoje existentes de invasão de sistemas computacionais e decorrente roubo, deturpação e destruição de informações. Segundo Russel, em [RUS 91], a segurança de sistemas computacionais compreende os quatro seguintes atributos:

a) autenticidade: entidades, como usuários e processos, devem ter sua identidade devidamente certificada a fim de possibilitar o emprego de controles de acesso eficientes aos recursos de um sistema computacional, bem como permitir a realização de auditorias;

b) confidencialidade: um sistema computacional seguro deve evitar que qualquer informação seja revelada para entidades que não possuam autorização para acessá-la;

c) integridade: o sistema deve impedir que as informações nele contidas sofram modificações não autorizadas, sejam estas acidentais ou intencionais; e

d) disponibilidade: o sistema deve manter as informações disponíveis para os seus usuários legítimos.

Cada um desses atributos exerce uma influência determinante no cumprimento de todos os outros. Se a integridade não puder ser garantida, por exemplo, a

confidencialidade pode ser comprometida por alterações nos direitos de acesso à recursos do sistema e a autenticidade por alteração dos seus respectivos mecanismos. Como conseqüência a disponibilidade pode ser prejudicada através do esgotamento ou destruição de recursos. Dessa forma, cada um dos atributos citados acima é de fundamental importância para a segurança de um sistema computacional, não sendo possível desconsiderar qualquer um deles sob pena de comprometê-la seriamente.

Esses atributos são incorporados aos sistemas através do emprego de mecanismos de segurança, os quais devem ser cuidadosamente modelados e implementados, a fim de não representarem novos riscos ao invés de uma solução. Tais mecanismos são naturalmente implementados nos sistemas operacionais, já que esses últimos são o núcleo de qualquer tipo de computador, ou seja, são os responsáveis por tarefas como: fornecer meios para execução de programas, fornecer acesso aos mais diversos periféricos, permitir a manipulação de arquivos, entre outros [OLI 2000]. Sendo assim, o sistema operacional é a base sobre a qual as mais diversas aplicações são utilizadas e por isso, quaisquer deficiências de segurança existentes no sistema operacional utilizado são refletidas na segurança das próprias aplicações.

1.2 Mecanismos de Controle de Acesso

Na busca pelo cumprimento dos requisitos de segurança anteriormente citados, vários mecanismos foram projetados e implementados nos sistemas operacionais. Infelizmente, a grande maioria desses mecanismos implementam apenas arquiteturas básicas de autenticação e de controle de acesso, procurando identificar usuários e determinar quais recursos podem ser utilizados por eles e de que forma, não abordando diretamente o requisito integridade.

O mecanismo de controle de acesso mais comum é o Discretionary Access Control (DAC), que se baseia na identidade de usuários e/ou grupos para permitir o acesso a objetos contidos no sistema (ex.: arquivos, sockets, dispositivos). O DAC pode ser representado de forma abstrata por uma matriz, onde os usuários estão dispostos nas linhas e os objetos nas colunas. Cada posição na matriz, definida pelo par (linha, coluna) especifica os direitos de acesso (ex.: read, write, execute, owner) a um objeto que são garantidos a um determinado usuário [SIL 98]. Os direitos de acesso somente podem ser alterados pelo dono do objeto, ou seja, ele pode compartilhar o objeto com quaisquer outros usuários, independente da política de segurança assumida no sistema. Essa é a principal característica do DAC, diferenciando-o completamente de outro mecanismo bastante conhecido mas raramente aplicado, o Mandatory Access Control (MAC).

O MAC controla o acesso aos recursos do sistema baseado nos níveis de segurança atribuídos à usuários e objetos [RUS 91]. Pode-se definir, por exemplo, dois níveis de segurança: confidencial e público. No caso de um objeto ser rotulado como confidencial, nem mesmo o dono desse objeto pode torná-lo disponível para outros usuários, que somente possuem direitos para acessar o nível público de segurança. É notável a diferença entre o DAC e o MAC: enquanto no primeiro o controle de acesso à um objeto é confiado ao seu dono, no segundo o controle é feito primeiramente pelo sistema.

Tanto no DAC quanto no MAC, o sistema acaba por criar domínios de proteção, os quais são representados pelos usuários e seus direitos de acesso. Os processos em um sistema encontram-se nos domínios dos usuários que iniciaram cada um deles. Assim, o processo recebe todos os direitos de acesso garantidos ao seu usuário [SIL 98]. Como resultado disso, o princípio do *menor privilégio* é violado [RUS 91], ou seja, vários programas são executados com direitos de acesso muito mais amplos dos que os necessários para cumprirem suas tarefas, decorrendo daí uma das principais fontes dos problemas de segurança hoje enfrentados.

Outro fator que vem a agravar o problema da violação do princípio do *menor privilégio*, é que a grande maioria dos sistemas operacionais assume uma postura um tanto discutível no que se refere à confiança no usuário, a qual separa os usuários em duas classes: absolutamente confiáveis (ex.: administrador, root) e não confiáveis [LOS 2001]. Os usuários pertencentes à primeira classe podem realizar qualquer tipo de operação sobre os recursos do sistema, operações essas que vão desde a modificação de qualquer parte do sistema de arquivos até a alteração da memória do kernel (núcleo) do sistema operacional.

1.3 Integridade de Arquivos

Além de falhos, nenhum dos mecanismos citados na seção anterior fornece controles de integridade dos arquivos mantidos no sistema, não possibilitando a detecção de alterações não autorizadas dos mesmos, sejam estas intencionais ou não. Nesse caso, não só o requisito integridade é inadequadamente abordado, mas também a autenticidade. Uma vez que um arquivo seja indevidamente alterado ele deixa automaticamente de ser autêntico, influenciando no comportamento de processos em execução no sistema e mesmo do próprio sistema operacional, levando todo o sistema computacional envolvido a um estado onde não é possível confiar em seu comportamento e resultados.

Fica assim claramente exposta a ineficiência dos mecanismos de segurança atualmente empregados na grande maioria dos sistemas operacionais no cumprimento do requisito integridade e, conseqüentemente, da autenticidade que, conforme colocado anteriormente, pode levar ao comprometimento total ou parcial de todos os outros requisitos. Torna-se necessária a criação de novos mecanismos, com diferentes e inovadoras abordagens, para suprir as lacunas existentes na segurança dos sistemas operacionais.

O presente trabalho tem por objetivo a modelagem de um mecanismo específico para o controle de integridade de arquivos em sistemas operacionais, de forma a permitir o aprofundamento dos conhecimentos nesse campo de pesquisa, que, apesar de não ser novo, ainda se encontra pouco explorado. Para tanto, foi realizado o levantamento e estudo das principais técnicas de controle de integridade de arquivos e dos problemas envolvidos nesse processo, bem como das soluções atualmente existentes. Para validar o modelo proposto, um protótipo foi implementado no kernel do sistema operacional Linux.

2 Integridade de Arquivos

Entre os subsistemas contidos em um sistema operacional, talvez o sistema de arquivos, ou file system (FS), seja o mais importante. Nele são mantidos arquivos de usuários, arquivos de configuração dos mais diversos serviços e utilitários, arquivos executáveis, além de conter o sistema operacional em si. Assim, além de ser a parte do sistema operacional mais visível para os seus usuários, o sistema de arquivos está também intimamente relacionado a todo o processo de carga do sistema [OLI 2000].

Sendo assim, a partir da modificação de alguns dos arquivos mantidos pelo sistema, é possível alterar de forma significativa o seu comportamento podendo até provocar o comprometimento de toda a sua segurança de funcionamento. Entre alguns resultados da alteração de arquivos pode-se ter: subversão de mecanismos de autenticação, interrupção ou deturpação de serviços, subversão de mecanismos de controle de acesso, paralisação total do sistema, entre outros.

Devido a esse fato, os FSs são um dos principais alvos de ataque de agentes maliciosos. Uma vez que um agente malicioso obtenha algum tipo de acesso a um sistema, dependendo dos privilégios conseguidos, ele pode alterar, substituir ou mesmo incluir novos arquivos com o intuito de esconder os traços da invasão, garantir futuros acessos ou ainda alterar configurações de diferentes serviços a fim de obter maiores privilégios de acesso aos recursos do sistema.

Exemplos práticos desse tipo de atividade maliciosa são os backdoors e rootkits. Os primeiros são utilizados pelos atacantes para a criação de outros meios de entrada no sistema invadido, além do meio inicialmente utilizado. Com isso, o atacante procura garantir o seu acesso indevido ainda que a vulnerabilidade explorada seja corrigida pelos administradores. Já os rootkits são ferramentas mais avançadas incluindo, além dos backdoors, versões modificadas dos programas de monitoramento e diagnóstico do sistema, que escondem as atividades do atacante, tornando sua detecção ainda mais difícil.

Não bastasse a criação de ferramentas cada vez mais avançadas para a invasão de sistemas, a constante criação de novas aplicações e a inserção ou melhoramento das funcionalidades das aplicações já existentes assim como as dos sistemas operacionais, dificultam ainda mais a detecção de modificações no sistema de arquivos, uma vez que o administrador é obrigado a manter um número já grande e ainda crescente de arquivos de configuração e executáveis.

Neste contexto o controle de integridade de arquivos passa a ser um recurso de extrema importância, já que sem ele a inserção de códigos maliciosos no sistema (e.g. backdoors) e as alterações não autorizadas de arquivos dificilmente serão detectadas e o agente malicioso obterá sucesso em sua investida.

2.1 Controle de Integridade de Arquivos

O processo chave para o controle de integridade de arquivos é a reconciliação de objetos, ou simplesmente a comparação de objetos. Para tanto, é necessário que o objeto, cuja integridade deverá ser futuramente verificada, seja copiado (duplicado). Num momento seguinte, quando é necessária a checagem da integridade do objeto em questão, compara-se a versão original desse com a cópia armazenada. Cabe aqui colocar que a cópia dos objetos para fins de controle de integridade é comumente chamada de instantâneo, ou snapshot.

Esse processo pode ser facilmente aplicado aos arquivos de um FS, para isso, basta a realização de uma cópia dos arquivos, e posterior comparação dos originais com a cópia previamente realizada. Assim, do processo de comparação pode-se obter os seguintes resultados:

a) nenhuma violação detectada: confirma que *no momento* em que a verificação foi realizada, os arquivos monitorados encontravam-se íntegros, autênticos;

b) violações não autorizadas detectadas: denunciam a presença de arquivos indevidamente alterados; esses arquivos podem ser facilmente substituídos a partir do snapshot ou das mídias originais; e

c) violações legítimas detectadas: indicam a necessidade de atualização do snapshot, para refletir um novo estado íntegro dos arquivos.

Caso seja feita a aplicação deste processo tal como descrito, ter-se-á certamente problemas de espaço de armazenamento no sistema, visto que ocorreria a duplicação do espaço ocupado por cada arquivo monitorado. Além disso, o processo de comparação de dois arquivos byte a byte ocasionaria um desperdício de tempo razoável.

A forma atualmente mais apropriada para a solução desses problemas é a utilização de funções matemáticas que, quando aplicadas ao conteúdo de um arquivo, geram um resumo matemático desse. Esse resumo matemático geralmente possui um tamanho pequeno e fixo de bits, que depende da função utilizada e normalmente varia entre 16 e 256 bits. Tem-se assim, uma forma de identificação de diversos conjuntos de bytes (arquivos), diferentes tanto em conteúdo quanto em tamanho, através da sua representação por uma seqüência singular e de tamanho fixo de bits. A esta seqüência dá-se o nome de digital, ou *fingerprint*.

Através do uso dessas funções matemáticas, o snapshot não precisa ser constituído de uma cópia fiel de cada arquivo, cujo tamanho é variável, mas apenas das digitais de cada um deles, cujo tamanho é sempre constante e pequeno (cerca de 2 a 32 bytes), proporcionando uma economia substancial de espaço de armazenamento do snapshot. Apesar desse ganho de espaço, a maior vantagem é a velocidade nos processos de geração do snapshot e de verificação da integridade dos arquivos, o que torna possível a criação de mecanismos de controle de integridade eficazes e realmente aplicáveis, conforme será visto ao longo deste trabalho.

Independente da forma como o snapshot é composto, o processo de controle de integridade de arquivos possui uma série de pontos sensíveis e potencialmente vulneráveis sob o ponto de vista da segurança. O tratamento inadequado destes pontos,

pode proporcionar a um agente malicioso a chance de intervir no processo, invalidando completamente todo o controle de integridade desejado.

Dessa forma, é de extrema importância que esses pontos sejam destacados e devidamente analisados. Nas seções seguintes, cada ponto sensível do processo é colocado, discutido e ainda algumas possíveis soluções são comentadas.

2.2 Estado Inicial dos Arquivos

O momento em que o snapshot vai ser gerado é crucial para garantir o sucesso do controle de integridade, pois daí resultará o parâmetro de comparação para a identificação de arquivos violados, a base de todo o processo. Caso o snapshot seja gerado a partir de um sistema já comprometido, todo o processo é invalidado, já que as possíveis alterações feitas pelo agente malicioso não poderão ser detectadas.

O que torna este primeiro passo ainda mais crítico, é o fato do administrador ter que confiar na integridade do sistema, o que acaba exigindo que ele estenda esta confiança aos desenvolvedores e distribuidores dos programas em uso, já que não é praticável, e as vezes nem possível, a auditoria dos códigos fonte de cada programa existente no sistema (inclui-se aí o próprio sistema operacional).

No caso da confiança cega e inevitável, pode-se reduzir de forma sensível os riscos optando-se por produtos de desenvolvedores e distribuidores mais conhecidos no mercado, levando-se em conta o tempo de existência e o número de usuários. Alguns distribuidores fornecem digitais de seus produtos, permitindo assim que a integridade seja verificada logo antes, ou mesmo após, a instalação dos mesmos, procedimento este, altamente recomendado.

O momento mais adequado para a geração do snapshot de um sistema é logo após sua instalação, pois, se realizada em ambiente adequado, nenhum agente malicioso terá a possibilidade de comprometer qualquer parte do sistema antes ou durante a geração do snapshot. Como ambiente adequado entende-se que:

a) há controle do acesso físico ao equipamento onde a instalação está sendo realizada;

b) não há nenhuma possibilidade de acesso remoto ao sistema, seja por usuários autorizados ou não;

c) todos os programas a serem instalados têm sua origem e integridade verificadas; e

d) após a instalação o sistema é varrido por programas de detecção de códigos maliciosos, tais como um antivírus.

Nos casos, muitas vezes freqüentes, em que o sistema já está em funcionamento, a dificuldade de obter-se um alto nível de confiança na integridade dele aumenta consideravelmente, pois deve-se então levar em conta os níveis de segurança e de monitoramento implementados no sistema desde a sua instalação, além do histórico de incidentes ocorridos e a forma como foram tratados. Tais procedimentos de avaliação

não estão compreendidos no escopo do presente trabalho e por isso não serão discutidos com maior profundidade, sem que isso diminua sua importância no processo.

Levando-se a efeito os procedimentos aqui comentados, obter-se-á um nível razoável de confiança na integridade do sistema, diminuindo-se consideravelmente os riscos de geração de um snapshot já comprometido.

2.3 Armazenamento do Snapshot

O snapshot não é somente vulnerável no momento da primeira geração, mas também durante o período em que é armazenado. Armazenar o snapshot de forma adequada é o procedimento básico para possibilitar o controle de integridade de arquivos.

Em caso de perda do snapshot, sua disponibilidade é comprometida tornando impossível a verificação da integridade dos arquivos monitorados. Ainda pior do que o comprometimento da disponibilidade do snapshot, é a alteração deste realizada por um agente malicioso, a fim de refletir as modificações por ele realizadas nos arquivos do sistema, evitando assim qualquer detecção por parte do mecanismo de controle de integridade. É preferível a perda ou a invalidação total do snapshot, pois, no segundo caso, a realização de uma verificação de integridade irá assegurar ao administrador que o sistema encontra-se íntegro, quando na realidade não está.

Sendo assim, o armazenamento adequado do snapshot, visando garantir a sua disponibilidade e integridade, é um fator crítico em qualquer método de controle de integridade de arquivos, tornando necessário o emprego de mecanismos para garantir a proteção do snapshot armazenado.

A seguir são descritos alguns desses mecanismos que podem ser empregados para proteção do snapshot. É possível classificar cada um deles como físicos ou lógicos. A proteção física é geralmente mais fácil de ser implementada e mais segura do que a proteção lógica, já que na primeira não basta acesso remoto ao sistema para modificar o snapshot, mesmo quando esse é feito com os privilégios de acesso do administrador do sistema.

2.3.1 CD-Recordable

A gravação do snapshot em uma mídia como o CD não regravável garante a validade do mesmo, dificultando consideravelmente a ação de qualquer agente malicioso. Ataques que contam apenas com o acesso remoto ao sistema, mesmo com os privilégios do administrador, são impossíveis de serem realizados já que a própria mídia tem restrições físicas inalteráveis que impedem qualquer modificação do seu conteúdo.

Mesmo com acesso físico à mídia, os ataques possíveis são limitados. O ataque mais fácil de ser realizado é a destruição da mídia – comprometimento de sua disponibilidade – impedindo a realização da verificação de integridade caso não haja um outra mídia reserva. O outro ataque é substituição da mídia original por uma outra já modificada, para isso não só é preciso o acesso físico mas também lógico ao sistema.

Esse ataque pode ser facilmente impedido com o uso deste mecanismo físico em combinação com um mecanismo lógico, como as assinaturas digitais.

Em resumo, este mecanismo físico fornece um nível bastante alto de proteção ao snapshot, praticamente evitando que sejam utilizados snapshots inválidos em processos de reconciliação de objetos.

2.3.2 CD-RW em drive CD-ROM

Este mecanismo é apenas uma variação do anterior. Neste, o snapshot é gravado em um CD regravável, o que possibilita a sua atualização, mas quando em uso é mantido somente em drives CD-ROM, pois esses não possuem condições físicas para alterar a mídia neles inserida. Essa é a proteção fornecida.

Os ataques possíveis para este mecanismo são os mesmos do mecanismo anterior. Como vantagens tem-se a economia de mídias e a redução de tempo, já que não é necessário gerar novamente todo o snapshot para realizar qualquer atualização. Ainda, assim como no mecanismo anterior, é recomendado o uso em conjunto com um mecanismo lógico.

Apesar de simples, tanto este quanto o mecanismo da seção anterior, impõem sérias restrições às possíveis investidas de um agente malicioso contra o snapshot armazenado, resultando na obtenção de um nível de segurança relativamente alto.

Outras mídias que fornecem algum tipo de proteção podem também ser utilizadas, mas antes de o serem devem ter suas características avaliadas a fim de não dar origem a outros pontos fracos para o sistema. Um exemplo de uma mídia alternativa são os ZIP Disks, que podem ser protegidos contra gravação com o uso de uma senha.

2.3.3 Montagem do FS em modo read-only

O snapshot pode ser mantido em um FS que é montado em modo *read-only*, evitando-se assim que qualquer modificação seja realizada no FS em questão. A idéia aqui empregada é bastante semelhante à de manter um CD-RW em um drive de CD-ROM. A diferença crucial é que este mecanismo é lógico e não físico, resultando daí o seu ponto fraco: caso o atacante obtenha acesso (remoto ou não) como administrador do sistema, ele pode desmontar o FS e montá-lo novamente de forma que modificações possam ser realizadas. Ou seja, a proteção só funciona contra usuários comuns do sistema.

Mesmo sofrendo da deficiência acima descrita, quando utilizado em conjunto com outros mecanismos lógicos, essa abordagem pode constituir uma barreira significativa para as ações de um agente malicioso.

2.3.4 Message Authentication Codes

A segurança do snapshot não pode ser baseada no fato de que o atacante desconhece o método utilizado para gerá-lo, pois, uma vez que o mecanismo de controle de integridade esteja publicamente disponível, o seu método de geração do snapshot passa também a ser de conhecimento público.

Assim sendo, caso um atacante obtenha o necessário acesso à mídia onde está armazenado o snapshot, seja físico ou lógico, ele poderá facilmente substituir determinadas entradas nele existentes visando tornar válidas suas modificações em arquivos contidos no sistema. Não importa neste processo se o snapshot é composto de cópias dos arquivos ou das digitais destes. Em qualquer um dos casos o atacante tem em sua posse os dados necessários para inserir entradas válidas no snapshot. No primeiro caso basta o arquivo em si e no segundo, além do arquivo, a função matemática empregada na geração do resumo.

Este problema pode ser solucionado com a inserção de algum dado no processo que seja somente de conhecimento do administrador do sistema e do mecanismo de controle de acesso. Esse dado a ser inserido é chamado de Message Authentication Code, ou simplesmente MAC. A expressão matemática da geração da digital D de um arquivo A através de uma função matemática F, sem a adição de um MAC, é:

$$D = F(A)$$

Neste caso, a digital do arquivo é obtida simplesmente com a aplicação da função matemática F. Com a adição de um MAC K, a nova expressão ficaria assim:

$$D = F(A, K)$$

O que pode ser notado na expressão acima, é que a chave (K) é integrada ao conteúdo do arquivo antes da geração da digital. Desse modo, para uma digital válida ser gerada ou verificada, não basta somente conhecer a função F e o conteúdo do arquivo A, mas também a chave K. Um atacante ficaria assim impossibilitado de inserir uma digital válida em um snapshot já que não conhece a chave utilizada.

Visando dificultar a descoberta da chave por métodos de força bruta ou pelo uso de dicionários de senhas, os quais podem ser facilmente empregados por um atacante, a mesma deve ser cuidadosamente escolhida, tal como uma senha de acesso à um sistema ou serviço.

Com a aplicação deste mecanismo, a chave K deve ser informada no momento da geração do snapshot bem como na verificação. Assim, caso o processo de verificação seja automatizado, a chave K deve ser armazenada no sistema, seja em disco ou em RAM. Nesses casos têm-se um novo problema, a segurança da chave a ser armazenada, pois caso o atacante obtenha acesso à ela, ele poderá realizar modificações no snapshot sem que estas sejam detectadas.

Ainda, é importante destacar que, como será verificado no capítulo 3, o que determina a geração adequada de uma digital não é apenas o uso ou não de um MAC, mas sim as características particulares da função matemática utilizada.

2.3.5 Cifragem com chaves simétricas

Um algoritmo de cifragem de chaves simétricas exige que a mesma chave utilizada no processo de cifragem seja também utilizada na decifragem. Este mecanismo pode ser utilizado para garantir a confidencialidade do snapshot e de certa forma a sua integridade, pois caso o snapshot cifrado seja alterado, o resultado obtido de uma decifragem será completamente diferente do original, algo como uma massa de caracteres embaralhada e sem sentido. [SCH 96]

O único ponto sensível desses algoritmos é que a mesma chave utilizada na geração do snapshot deve estar de alguma forma disponível para o mecanismo de controle de integridade para que ele realize a verificação e, se ela for comprometida, um atacante pode facilmente modificar o snapshot. Essa dificuldade é inexistente nos algoritmos de chaves assimétricas, comentados na seção seguinte. [SCH 96]

2.3.6 Assinaturas digitais

O uso de assinaturas digitais pode garantir a confidencialidade e a autenticidade do snapshot. As assinaturas digitais são realizadas com o uso de algoritmos de chaves assimétricas, também ditos de chave pública.

O fato das chaves serem assimétricas significa, em outras palavras, que chaves diferentes são utilizadas no processo de cifragem e decifragem. Esse par de chaves é constituído de uma chave privada e outra pública, e a relação entre elas é a seguinte: tudo o que for cifrado com a chave pública, somente pode ser decifrado com o uso da chave privada correspondente; e tudo que for cifrado com a chave privada, só pode ser decifrado com a chave pública correspondente. [MEN 96]

Sendo assim, o administrador do sistema teria o seu par de chaves e com sua chave privada cifra (assina) o snapshot, ou apenas o seu resumo matemático. [MEN 96] Dessa forma, somente a chave pública precisa estar armazenada no sistema. Nesse caso, mesmo que o atacante obtenha acesso à ela, ele não é capaz de modificar o snapshot. Aqui, somente a autenticidade do snapshot é garantida.

Opcionalmente, o próprio sistema poderia ter um par de chaves, o que possibilitaria além de autenticidade a confidencialidade. O administrador primeiro assina o snapshot, e em seguida cifra-o com a chave pública do sistema.

O ataque mais prático aqui é a substituição de chaves, ou seja, o atacante gera um novo par de chaves, substitui a chave pública do administrador armazenada no sistema pela sua, em seguida altera o snapshot e assina-o. Dependendo da forma como a chave pública e o snapshot é armazenado, esse ataque pode se tornar muito difícil ou praticamente impossível de ser realizado, principalmente quando usado em conjunto com um mecanismo físico de proteção, tal como o CD-R.

2.3.7 Controles de acesso adicionais

Conforme colocado na seção 1.2, os mecanismos de acesso comumente empregados nos sistemas operacionais apresentam diversas falhas, tornando-os insuficientes para garantir a segurança do snapshot.

Felizmente, existem mecanismos opcionais de controle de acesso que podem ser adicionados ao sistema operacional. Alguns deles permitem até mesmo a limitação dos direitos de acesso do próprio administrador, impondo uma barreira relevante para o problema da violação do princípio do menor privilégio.

Pode-se então, com o uso de tais mecanismos adicionais, reforçar a segurança dos mecanismos anteriormente comentados, como por exemplo do uso de um FS em modo read-only. Neste caso específico, seria possível retirar do administrador os direitos de alterar as propriedades do FS, solucionando-se assim o ponto fraco desse mecanismo de proteção do snapshot.

2.4 Atualização do Snapshot

Ao longo do uso de um sistema, diversas tarefas administrativas acabam por causar a execução de alterações intencionais e não maliciosas, portanto válidas, no sistema de arquivos. Entre essas tarefas tem-se:

a) aplicação de correções em programas já existentes (patches);

b) atualização de programas já existentes (updates);

c) instalação de novos programas; e

d) customização de configurações executadas pelo próprio administrador.

Independentemente do tipo de tarefa que deu origem às modificações válidas no sistema de arquivos, o snapshot deve ser atualizado a fim de refletir esse novo estado íntegro do sistema. Este processo de atualização é tão sensível quanto o processo de criação do snapshot inicial do sistema, visto na seção 2.2. A falta de conhecimento e de controle do que realmente foi alterado ou adicionado ao sistema de arquivos, pode levar à consolidação de modificações inválidas, potencialmente intencionais e maliciosas no snapshot.

Nenhum mecanismo de proteção do snapshot é capaz de evitar a ocorrência deste tipo de incidente, já que este é resultante da imperícia do próprio administrador. Felizmente é possível reduzir consideravelmente o risco de comprometimento do snapshot, através da utilização do próprio mecanismo de controle de integridade de arquivos e do emprego de procedimentos adequados por parte do administrador. Abaixo são citados e comentados alguns passos genéricos que podem ser adotados como procedimento para a referida atualização:

a) *verificação da integridade dos arquivos do sistema*: este passo deve ocorrer antes da alteração dos arquivos, a fim de dar certeza do estado íntegro atual

dos arquivos do sistema; quaisquer violações detectadas devem ser imediatamente investigadas e solucionadas pelo administrador;

b) *execução das alterações*: executado o primeiro passo, pode-se então realizar as alterações necessárias;

c) *nova verificação da integridade dos arquivos do sistema*: essa segunda verificação tem por objetivo levantar todas as modificações realizadas nos arquivos do sistema; em seguida, todas as modificações identificadas devem ser investigadas e relacionadas às tarefas executadas no segundo passo; modificações não relacionadas não devem ser mantidas no sistema e devem ser tratadas como violações potencialmente intencionais e maliciosas, sendo imediatamente corrigidas; e

d) *atualização do snapshot*: finalmente o snapshot pode ser atualizado com segurança.

Ainda, visando um maior nível de segurança, o procedimento aqui descrito deve ser realizado em um ambiente similar ao exposto na seção 2.2.

Como foi visto, apesar de não ser possível resolver o problema descrito apenas com o uso de mecanismos de controle de integridade, a adoção dos procedimentos apropriados por parte do administrador pode reduzir substancialmente as chances de comprometimento do snapshot.

2.5 Restauração de Arquivos Violados

A detecção de uma violação no sistema de arquivos, o que pode sinalizar a execução de ações maliciosas por exemplo, não basta para fornecer segurança para um sistema. Necessita-se ainda de algum meio para corrigir a violação detectada, fazendo com que o sistema volte ao seu estado íntegro original.

Infelizmente, mecanismos de controle de integridade baseados em resumos matemáticos não são capazes de corrigir violações, uma vez que não é possível obter-se o arquivo original apenas partindo de seu resumo. Sendo assim, o uso de um mecanismo de controle de integridade deve ser combinado com uma prática já bem documentada e universalmente recomendada: a realização de cópias de segurança, ou backups.

As cópias de segurança são indicadas para arquivos customizados e particulares de cada sistema, já que arquivos de programas e também o sistema operacional podem ser facilmente obtidos a partir da mídia original de instalação. A inexistência de cópias de segurança ou das mídias originais pode levar o sistema a um estado sabidamente inválido e sem a possibilidade de correção das violações detectadas. Nesses casos, que devem ser considerados inadmissíveis tanto do ponto de vista da segurança quanto administrativo, pode-se ter que recorrer a paralisação e reinstalação de partes ou de todo o sistema envolvido.

Muito embora a cópia de segurança de arquivos esteja, de certa forma, intimamente relacionada ao controle de integridade, não faz parte do objetivo deste

trabalho o detalhamento das técnicas e procedimentos envolvidos na sua execução, já que é extremamente farta e difundida a documentação existente sobre o assunto.

2.6 Identificação de Ações Maliciosas

Os mecanismos de controle de integridade somente detectam violações nos arquivos cuja integridade é por ele monitorada, ou seja, ele não indica quando uma violação foi intencional ou não e muito menos se foi maliciosa.

Para tais classificações deve ser realizada uma investigação da violação detectada, utilizando-se para isso ferramentas específicas e o próprio conhecimento do administrador do sistema. Entre as diversas ferramentas disponíveis e conhecidas que podem ser empregas tem-se: programas que varrem o sistema a procura de códigos maliciosos como vírus, backdoors e rootkits, o próprio registro das operações do sistema (arquivos de log) e as ferramentas utilizadas para sua análise.

Mesmo com o emprego dessas e de outras ferramentas, o conhecimento e experiência do administrador são muitas vezes determinantes para o sucesso de uma investigação de violações ocorridas em arquivos do sistema. Mais uma vez, tal como colocado nas seções anteriores, o fator humano é de extrema importância para o funcionamento adequado de um mecanismo de controle de integridade.

2.7 Integridade do Mecanismo de Controle

Exatamente por ser um dos responsáveis pela segurança do sistema, o mecanismo de controle de integridade pode virar alvo da ação de um atacante, pois caso ele consiga comprometer o funcionamento do mecanismo, uma importante barreira para o sucesso de seus propósitos será eliminada. E ainda, dependendo de como o mecanismo é comprometido, ele pode ser convertido em ferramenta de apoio ao atacante fazendo com que o administrador e usuários tenham uma falsa sensação de segurança ao pensar que o mecanismo está funcionando corretamente e não investiguem qualquer comportamento estranho do sistema.

Chega-se então ao ponto em que a integridade do próprio mecanismo de controle deve ser regularmente verificada, tanto por processo automatizado quanto por processo manual. Este procedimento é necessário pois o comprometimento do mecanismo pode indicar que uma grave vulnerabilidade foi previamente explorada pelo atacante, expondo o sistema ao risco de perda total ou parcial dos arquivos nele contidos, bem como de vazamento e deturpação de informações.

Dessa forma, a segurança do próprio mecanismo deve ter destacada importância durante o seu processo de modelagem e implementação, sob o risco de sua aplicação ser não somente inútil mas também prejudicial para a segurança do sistema.

2.8 Intervalo entre Verificações

O intervalo de tempo entre duas verificações de integridade é o tempo que um atacante tem para agir sem que seja detectado, podendo assim alterar o conteúdo dos arquivos conforme seus propósitos. Como conseqüência, a relação entre intervalo de tempo e segurança é inversamente proporcional, ou seja, quanto maior esse intervalo, menor será a segurança fornecida e vice-versa.

Dessa forma, quando o objetivo é atingir um nível alto de segurança, tende-se a reduzir ao mínimo possível o intervalo de tempo. Infelizmente esta prática coloca em risco o desempenho do sistema como um todo, podendo prejudicar o fornecimento dos serviços que estão sendo oferecidos aos usuários e talvez determinado a desativação do mecanismo de controle de integridade. Desempenho é, portanto, um importante fator a ser levado em conta na modelagem e implementação de qualquer mecanismo deste tipo.

A medida que o intervalo de verificação aumenta, a perda de desempenho torna-se mais aceitável, ou seja, com intervalos grandes de tempo entre duas verificações qualquer perda de desempenho será momentânea e não deverá causar prejuízo às operações de rotina do sistema. No entanto, perde-se em relação à segurança.

A perda de desempenho e a segurança fornecida não são somente determinadas pelo intervalo de tempo utilizado entre duas verificações, mas também pela forma como o mecanismo é implementado, os momentos em que o mesmo realiza o controle de integridade e a quantidade de arquivos monitorados.

2.9 Momentos para a Verificação

Os momentos escolhidos para a realização da verificação da integridade de arquivos influenciam diretamente o nível de segurança fornecido pelo mecanismo empregado e o mínimo intervalo de tempo possível entre duas verificações. De modo geral, pode-se dizer que existem dois momentos para a realização da verificação de integridade: durante o isolamento do sistema e durante a sua operação normal.

O primeiro momento é encontrado em sistemas que podem ter suas atividades temporariamente suspensas em períodos regulares de tempo. Nesses casos a principal vantagem é que praticamente qualquer perda de desempenho é aceitável, pois não há o risco de prejuízo para as operações de rotina do sistema, possibilitando a monitoração de uma grande quantidade de arquivos. Ainda, evita-se nesses momentos até mesmo uma possível monitoração realizada por um atacante, caso ele já tenha obtido algum nível de acesso ao sistema, diminuindo suas chances de interferir no processo de verificação de integridade.

Embora pareça ser um momento ideal para a verificação, principalmente considerando a despreocupação com o desempenho, normalmente o isolamento do sistema não ocorre muito freqüentemente prejudicando a periodicidade da verificação, ou seja, o intervalo de tempo entre verificações tende a ser longo, fazendo com que as possíveis violações demorem a ser detectadas, comprometendo o nível de segurança fornecido.

Intervalos de tempo menores são obtidos de forma mais fácil quando a verificação é realizada durante a operação normal do sistema, uma vez que este último não precisa ter suas atividades paralisadas. Porém, possíveis perdas de desempenho são aqui fator limitante, pois quanto menor o intervalo, maiores serão os gastos de recursos, principalmente tempo de processador. A princípio, fica limitado também o número de arquivos a serem monitorados, já que quanto maior for este número, maiores serão os custos de uma verificação.

O modo como a verificação é executada durante a operação normal do sistema influencia fortemente o impacto resultante no desempenho deste último. Existem basicamente dois modos de execução:

a) *verificação disparada em um dado instante*: a verificação é executada em um dado instante escolhido pelo administrador, podendo ser disparada de forma interativa ou automatizada; nesse modo, a integridade de todos os arquivos é verificada de uma só vez, concentrando a perda de desempenho em um curto espaço de tempo e tornado-a sensível para os usuários do sistema; e

b) *verificação sob demanda*: a verificação é realizada cada vez que for requisitado acesso (e.g. leitura, escrita ou execução) a um arquivo monitorado; a perda de desempenho é aqui diluída ao longo do tempo, já que somente os arquivos utilizados são verificados, o que não acontece a todo momento; além dessa vantagem, este tipo de mecanismo evita que um arquivo violado seja utilizado sem que ocorra a detecção e ainda é possível que ele bloqueie o acesso ao arquivo violado, evitando o seu uso no sistema.

É clara a vantagem do segundo modo, que além de reduzir o impacto no desempenho do sistema, fornece um maior nível de segurança para o mesmo.

A existência dos pontos críticos anteriormente comentados não condenam a realização de verificações durante a operação normal do sistema, mas apenas tornam necessário um maior cuidado e atenção na modelagem e implementação do mecanismo de controle de integridade. Havendo tal preocupação é possível obter-se um equilíbrio satisfatório entre segurança e perda desempenho.

Vistas as vantagens e desvantagens de cada momento, é importante ressaltar que não é necessário que um mecanismo utilize apenas um deles para a verificação de integridade. Na verdade, o ideal é a possibilidade de utilização de ambos, aproveitando-se das vantagens de cada um para o fornecimento de um alto nível de segurança.

3 Funções de Integridade

Conforme colocado na seção 2.1, manter uma cópia integral de cada arquivo, cuja integridade deva ser controlada, não é um procedimento prático para o processo de controle de integridade. Não é prático devido a várias dificuldades impostas ao processo, entre elas:

a) *perda de tempo na leitura de arquivos*: a cada momento que a integridade de um arquivo for verificada, ter-se-á que ler na íntegra o arquivo original e a sua cópia existente no snapshot; este tipo de procedimento inviabiliza a implantação de mecanismos cujo o intervalo de tempo entre verificações não seja longo, pois causaria impacto sensível no desempenho no sistema;

b) *snapshot muito grande*: o tamanho do snapshot será dado pela soma do tamanho de cada arquivo verificado, podendo facilmente ultrapassar a barreira dos megabytes, dificultando o seu armazenamento e fazendo com que mídias, como os CDs, tornem-se inviáveis para tal; métodos de compressão de dados podem até serem utilizados para resolver este problema, mas como conseqüência a perda de desempenho é ainda maior uma vez que para a verificação de um arquivo sua cópia deve ser lida, descompactada e então comparada com o original.

Devido a tais limitações, esse procedimento não é empregado na grande maioria dos mecanismos de controle de integridade. Ao invés disso, esses mecanismos geralmente adotam o uso de resumos matemáticos dos arquivos.

Um resumo matemático representa o conteúdo de um arquivo utilizando um número pequeno e fixo de bits, solucionando já num primeiro momento o problema da armazenagem do snapshot e, por conseqüência, evitando um impacto muito grande no desempenho do sistema. O resumo é facilmente obtido através da aplicação de uma função matemática sobre o conteúdo do arquivo. Para verificar a integridade de um arquivo, basta aplicar novamente a função matemática sobre o arquivo atual e comparar o resumo resultante com o existente no snapshot. [SCH 96]

São várias as funções existentes para o controle de integridade de uma determinada massa de dados. Algumas são extremamente simples, porém de aplicação bastante restrita, enquanto outras mais complexas fornecem características bastante relevantes para o tipo de mecanismo estudado e implementado neste trabalho. Nas seções seguintes são relacionados os principais tipos de funções e alguns algoritmos são detalhadamente comentados.

3.1 Checksums

Os checksums são funções extremamente simples e foram criadas para a detecção de erros acidentais ou não maliciosos em dados transmitidos por canais sujeitos a ocorrência de erros durante a transmissão. [MEN 96]

Sendo notório o fato de que as necessidades para a detecção de erros causados por ruídos em canais de comunicação diferem substancialmente daquelas para detecção de erros intencionais, sendo este último o principal objetivo de um mecanismo de controle de integridade, os checksums serão tratados neste trabalho apenas com o objetivo de tornar mais claros os requisitos necessários para uma função matemática ser considerada adequada à solução do problema de controle de integridade. [MEN 96]

3.1.1 Bit de paridade

O uso de um bit de paridade em uma massa de dados é um exemplo bem simples de checksum. Nesse caso o bit de paridade é escolhido de forma a tornar o número de bits 1 existente na massa de dados, par ou ímpar, de acordo com o que foi previamente estabelecido pelo mecanismo. [TAN 96]

Assim, tomando-se como exemplo a seguinte massa de dados:

100110101110

Sendo a paridade par e o número de bits 1 igual a sete, o bit de paridade será 1, resultando, então, num número par de bits 1: oito. O bit de paridade é capaz de detectar erros em apenas um bit, ou seja, caso um bit 1 seja alterado para 0, ou vice-versa, então o número de bits 1 passa a ser ímpar, indicando a existência de um erro. Nos casos em que mais de um bit é alterado, mantendo-se o mesmo número de bits 1, o erro não será detectado. [TAN 96]

O bit de paridade não é adequado para o controle de integridade de arquivos, visto que a possibilidade de um erro não ser detectado, mesmo no caso de erros acidentais causados por ruídos, é muito grande. Em erros intencionais e maliciosos, onde o atacante poderia facilmente alterar um arquivo mantendo o mesmo valor do bit de paridade, esse problema fica ainda mais claro.

3.1.2 Exclusive OR (XOR)

Outra função que permite a detecção de erros pode ser construída através de operações simples de XOR. Neste caso a massa de dados é dividida em blocos de mesmo tamanho, e em seguida é realizado um XOR de um bloco com o bloco seguinte, constituindo um laço de execução que irá processar até o último bloco de dados. O resultado da última operação XOR é o checksum. [MEN 96]

O esquema colocado na figura a seguir representa a função descrita:

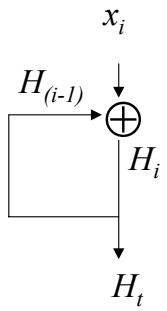


FIGURA 3.1 – Checksum usando operações XOR

Supondo uma implementação que utilize blocos de 8 bits, e uma massa de dados contendo 24 bits como entrada, o algoritmo seria:

1) o tamanho de *m*, caso seja necessário, é ajustado com o número de bits restantes para torná-lo múltiplo do tamanho dos blocos utilizados (neste exemplo, 8 bits);

2) a massa de dados *m* é dividida em *t* blocos *x* de 8 bits cada, *H* e *i* são inicializados: $t = 24 / 8 = 3$; $H_1 = x_1$ e $i=2$;

3) as iterações ocorrem para $2 \leq i \leq t$; a saída *H* de cada iteração é então dada por: $H_i = x_i \oplus H_{(i-1)}$;

4) o resultado da última iteração é o checksum da massa de dados usada como entrada.

Aplicando-se este algoritmo a: 00100011 01100001 11010110, tem-se:

a) $x_1 = 00100011$; $x_2 = 01100001$; $x_3 = 11010110$; $t = 3$; $H_1 = x_1$; $i = 2$;

b) para $i = 2$; $H_2 = x_2 \oplus H_1 = 00100011 \oplus 01100001 = 01000010$;

c) para $i = 3 = t$; $H_3 = x_3 \oplus H_2 = 11010110 \oplus 01000010 =$ **10010100**.

O resultado da última iteração, passo c, é o valor do checksum. Para verificar-se a integridade da massa de dados *m*, basta calcular novamente o checksum e comparar com o originalmente obtido.

Comparado com o uso de bits de paridade, o XOR é bem mais eficiente na detecção de erros não intencionais mas, infelizmente, também não detecta a inserção de erros intencionais e maliciosos. Para um atacante modificar ou mesmo criar uma nova massa de dados, mantendo o mesmo valor de checksum, basta ele acrescentar um bloco adicional para corrigir o checksum para o valor desejado. [MEN 96]

Tomando-se a seguinte massa de dados criada por um suposto atacante, a fim de substituir a do exemplo anterior: 11011010 01100011 10101111, os passos por ele realizados seriam:

a) calcular o checksum dos novos dados, com a aplicação do mesmo algoritmo anteriormente descrito; o atacante então obteria $H_t' = 00010110$; tal resultado

é completamente diferente do checksum da mensagem original, H_t = 10010100;

b) a fim de fazer com que H_t' se iguale a H_t, o atacante cria um novo bloco *b* para fazer a correção de H_t'; o bloco é criado com: $b = H_t' \oplus H_t$; b = 00010110 ⊕ 10010100 = 10000010;

c) o novo bloco *b* é então adicionado à massa de dados do atacante, ficando esta com quatro blocos: 11011010 01100011 10101111 *10000010*.

Após tal procedimento, o checksum da nova massa de dados (H_t') é 10010100, ficando então $H_t' = H_t$. Dessa forma um atacante consegue substituir uma massa de dados válida, por outra completamente diferente, mas mantendo o valor do checksum.

Tendo-se em vista tais aspectos de uma função de checksum utilizando operações XOR, pode-se considerá-la inapropriada para a implementação de mecanismos de controle de integridade de arquivos, uma vez que um arquivo poderia ser maliciosamente substituído por um outro completamente diferente, sem a menor possibilidade de detecção. [MEN 96]

3.1.3 Cyclic Redundancy Check (CRC)

O CRC é também conhecido como código polinomial, ou seja, é empregada a divisão de polinômios para o cálculo do valor do checksum, sendo este último o resto da divisão. Para tanto, a massa de dados é tratada como sendo uma representação de um polinômio cujos coeficientes possíveis são somente 0 e 1. Assim, uma massa de dados contendo *k* bits representa um polinômio cujos termos vão de $x^{(k-1)}$ a x^0 e cujo grau é *k*-1. [TAN 96]

Tomando-se como exemplo a seqüência de bits 10111, o seguinte polinômio é por ela representado:

$$1x^4 + 0x^3 + 1x^2 + 1x^1 + 1x^0,$$

ou simplificando:

$$x^4 + x^2 + x + 1$$

A massa de dados é então tomada como dividendo e um polinômio gerador, a ser escolhido, é utilizado como divisor. O resultado da divisão é descartado, só interessando o valor do resto. Assim, o mesmo polinômio gerador deve ser utilizado tanto no cálculo do checksum quanto na sua verificação. [WIL 93]

A aritmética polinomial é feita em módulo 2, não havendo então carry ou borrow. Assim as operações de soma e subtração ficam idênticas à operação binária XOR. [TAN 96]. Seguem alguns exemplos:

10011011 + 11001010 = 01010001

11110000 – 10100110 = 01010110

Sendo a divisão uma série de subtrações, a divisão de 10101100 por 1011 seria assim executada:

```
10101100 | 1011
1011     |------
0001110  | 10011
   1011
   01010
    1011
    0001
```

FIGURA 3.2 – Divisão polinomial módulo 2

A partir desses princípios, o CRC é dado pela aplicação do seguinte algoritmo [TAN 96]:

1) um polinômio gerador G(x) deve ser escolhido, onde tanto o bit mais significativo quanto o menos significativo de G(x) deve ser 1;

2) considerando que *g* seja o grau de G(x), são acrescentados *g* bits 0, no lado menos significativo da massa de dados *m*; e

3) em seguida, *m* é divida por G(x); o resto desta divisão é o valor do checksum.

Entre os polinômios geradores tomados como padrão, tem-se [TAN 96]:

a) CRC-12 = $x^{12} + x^{11} + x^3 + x^2 + x^1 + 1$

b) CRC-16 = $x^{16} + x^{15} + x^2 + 1$

c) CRC-CCITT = $x^{16} + x^{12} + x^5 + 1$

O CRC-12 é utilizado quando o tamanho de um caracter é 6 bits, enquanto os outros dois são projetados para caracteres de 8 bits. Um checksum de 16 bits, tal como o CRC-16 e o CRC-CCITT, detecta qualquer erro simples (mudança de um único bit) ou dobrado (mudança de dois bits), todos os erros com número impar de bits e todas as rajadas de erros com 16 ou menos bits. [TAN 96]

Aplicando-se o algoritmo à massa de dados *m* = 10101100 e utilizando-se um polinômio gerador G(x) = $x^3 + x^1 + x^0$ = 1011, tem-se:

a) sendo *g = 3*, são adicionados três bits 0 ao final de *m*, resultando em *m* = 10101100*000*;

m é então dividida por G(x) conforme a figura a seguir:

```
10101100000 | 1011
1011          ------
------        10011001
0001110
   1011
   ----
   01010
    1011
    ----
    0001000
       1011
       ----
       0011
```

FIGURA 3.3 – Cálculo do CRC Checksum

O resto 0011 é o checksum de *m*. Normalmente, na transmissão de dados, o checksum é adicionado a *m* e, ao chegar no destino, *m* é novamente dividida pelo polinômio G(x). Caso não tenham ocorrido erros durante a transmissão, não haverá resto. No caso de controle de integridade de arquivos, somente o checksum é armazenado, e no momento da verificação, o checksum é novamente calculado e então comparado com o existente no snapshot.

Independentemente do polinômio gerador escolhido, um checksum CRC pode também ser facilmente manipulado por um atacante, bastando para isso que as modificações sejam sempre um múltiplo do polinômio G(x), não alterando assim o valor do resto. Como resultado, um atacante poderia substituir ou alterar com sucesso um determinado arquivo sem que tal ação fosse detectada, subvertendo o mecanismo de controle de integridade.

3.2 Funções de Hash Unidirecionais

As funções de hash unidirecionais, também conhecidas como funções criptográficas de hash ou simplesmente funções de hash, desempenham um importante papel na criptografia moderna, fazendo parte dos mais diversos protocolos criptográficos existentes. Elas funcionam de maneira semelhante às funções anteriormente comentadas, uma vez que mapeiam uma massa de dados de tamanho variável para uma seqüência de tamanho fixo de bits. Em favor de uma maior clareza do texto, as funções de hash unidirecionais serão aqui referenciadas apenas como funções de hash. Ainda, para a busca de maiores detalhes, recomenda-se a consulta de [MEN 96] e [SCH 96].

O resultado do uso de uma função de hash, chamado de digital (fingerprint), message digest (MD) ou simplesmente hash, pode ser inicialmente comparado a um checksum, ou seja, ambos permitem a detecção de alterações dos dados utilizados para gerá-los, mas não fornecem qualquer recurso para a correção das mesmas. [SCH 96]

De forma mais abrangente, uma função de hash deve ter no mínimo as seguintes características:

a) compressão: uma entrada *x* de tamanho arbitrário é mapeada para uma saída *h(x)* de tamanho fixo; e

b) facilidade de cálculo: dada uma função *h* e uma entrada *x*, deve ser fácil calcular *h(x)*.

Considerando-se apenas estas duas características é até possível classificar uma função de checksum XOR, por exemplo, como sendo uma função de hash. Mas apesar da aparente semelhança, as funções de checksum não dispõem das propriedades adicionais que fazem com que uma função de hash seja unidirecional. Essa diferenciação é facilmente justificada uma vez que funções de checksum foram projetadas para detectar alterações causadas por ruídos existentes em canais de transmissão, e não alterações intencionais e maliciosas. Já as funções de hash foram especificamente projetadas para evitar a ocorrência de qualquer modificação que não possa ser detectada, seja esta intencional ou não. [MEN 96]

Sendo assim, além da compressão e facilidade de cálculo, as funções de hash têm as seguintes propriedades adicionais:

a) unidirecionalidade: dado um hash de uma entrada não conhecida, *h(x)*, deve ser muito difícil obter a entrada *x* em parte ou na íntegra; essa dificuldade deve manter-se mesmo que parte de *x* seja conhecida;

b) resistência a colisões fracas: dado uma determinada entrada *x*, deve ser muito difícil encontrar outra entrada *x'*, tal que *h(x') = h(x)*;

c) resistência a colisões fortes: deve ser muito difícil encontrar duas entradas quaisquer, *x* e *x'*, tal que *h(x') = h(x)*; a principal diferença desta em relação as duas anteriores é que o atacante tem liberdade total para escolher tanto *x* quanto *x'*;

d) resistência a colisões próximas: deve ser difícil encontrar duas entradas *x* e *x'* tal que *h(x)* e *h(x')* difiram em apenas um número pequeno de bits; e

e) inexistência de correlação: os bits da entrada *x* e da saída *h(x)* não devem ser correlacionados; a idéia aqui é que cada bit da entrada afete todos os bits da saída; isto é conhecido como efeito avalanche.

Aqui o termo "difícil", conforme comumente utilizado na literatura, não é algo facilmente mensurável, mas normalmente refere-se à necessidade de emprego de esforços que excedem, em muito, os recursos atualmente disponíveis (e.g. poder de processamento). A segurança está, então, essencialmente baseada nesta limitação de recursos.

Considerando-se uma função de hash que tenha as propriedades acima relacionadas e que tenha sido cuidadosamente implementada, o melhor ataque é por meio da força bruta. Assim, caso um atacante queira comprometer as propriedades de unidirecionalidade e resistência a colisões fracas, encontrando uma entrada *x'* qualquer que resulte em um determinado hash *h(x)*, ele deverá realizar 2^n tentativas, onde *n* é o número de bits do hash. No ataque à terceira propriedade, resistência a colisões fortes, onde basta encontrar duas entradas quaisquer que resultem em um mesmo hash, o atacante teria de realizar $2^{n/2}$ tentativas.

Uma função de hash cuja saída tem 64 bits, exigiria 2^{64} tentativas do atacante para realizar o primeiro ataque descrito no parágrafo anterior. Caso ele consiga realizar

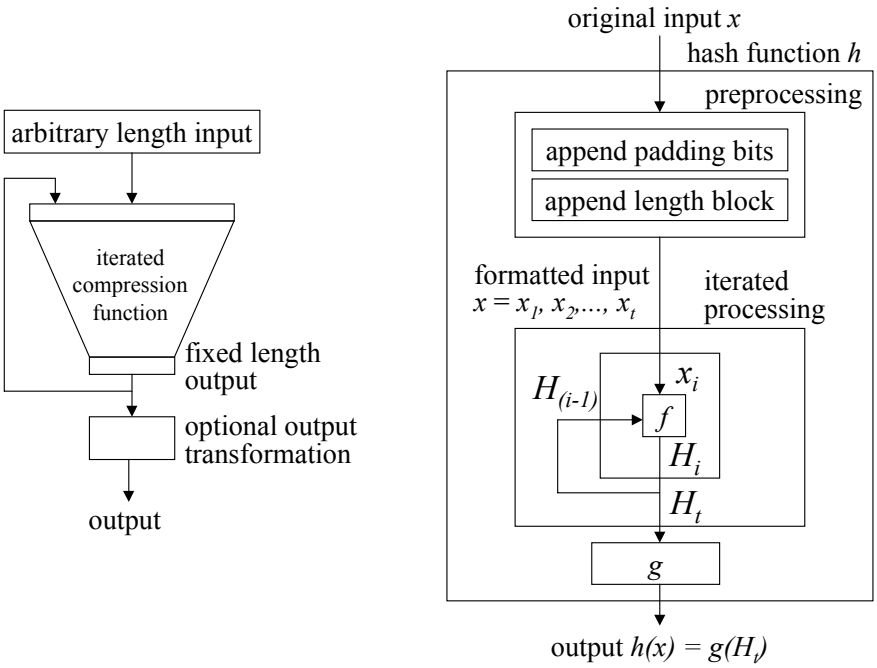
um milhão de tentativas por segundo, serão necessários 600.000 anos para achar uma entrada *x'* que corresponda ao hash determinado. Já no segundo ataque, apenas $2^{64/2}$ tentativas são necessárias. Assim, um atacante com os mesmos recursos empregados no ataque anterior levaria cerca de uma hora para encontrar duas entradas quaisquer que resultem em um mesmo hash. Devido a isso, o tamanho mínimo recomendado para *n* é 128 bits, o que é praticamente um padrão entre as funções de hash atuais. Aplicados às funções de checksum, como foi visto na seção 3.1, esses mesmos ataques não encontram nenhuma dificuldade de serem realizados.

Todos esses aspectos comentados tornam as funções de hash ferramentas extremamente úteis para o controle de integridade de arquivos, permitindo a criação de uma representação compacta e única de uma determinada massa de dados.

Nas seções seguintes serão tratados aspectos genéricos da implementação das funções de hash, os tipos de funções existentes (MDC e MAC), bem como alguns algoritmos cujo uso é amplamente difundido.

3.2.1 Modelo genérico de uma função de hash

As funções de hash são normalmente projetadas como processos iterativos que, recebendo uma entrada de tamanho arbitrário, a dividem em blocos de tamanho fixo que em seguida são sucessivamente processados [MEN 96]. A figura abaixo ilustra o funcionamento genérico de uma função de hash.



(a) high-level view (b) detailed view

FIGURA 3.4 – Modelo genérico de uma função de hash
Fonte: [MEN 96], p.332

Os seguintes itens enumeram as etapas que compõem um algoritmo de hash:

1) inicialização: H_o é inicializado com um valor predefinido, initializing value (IV);

2) pré-processamento (*padding*): dada uma entrada x de tamanho arbitrário e finito o seu tamanho é ajustado acrescentando-se o número de bits necessários para torná-lo um múltiplo do tamanho de bloco (*r)* utilizado pelo algoritmo;

3) pré-processamento (length block): é adicionado ao final da entrada x um bloco com a representação binária do tamanho original de x; ao final desta etapa, x é dividida em blocos de r bits (x_1, x_2, ..., x_i), que por sua vez são passados como entrada para o processamento iterativo;

4) função de compressão: cada bloco x_i serve de entrada para uma função de compressão f, também chamada de função de hash interna; a cada iteração, f toma como entrada o resultado da última interação realizada ($H_{(i-1)}$), e o próximo bloco x_i, obtendo-se daí um resultado intermediário H_i de tamanho fixo n bits; essa iteração ocorre para $1 \leq i \leq t$

5) função de transformação: esta etapa é opcional e portanto não é encontrada em todos os algoritmos; ao final do processamento iterativo, um resultado H_t é obtido e fornecido como entrada para uma função de transformação g; o hash $h(x)$ é então o resultado de $g(H_t)$; para os algoritmos que não utilizam esta etapa $h(x) = H_t$.

Todo este processo pode ser representado da seguinte forma:

$$H_o = \text{IV}; \quad H_i = f(H_{(i-1)}, x_i),\ 1 \leq i \leq t; \quad h(x) = g(H_t).$$

Visando uma melhor clareza alguns pontos interessantes e importantes desse processo são destacados nos parágrafos seguintes.

Existem dois modos normalmente utilizados para a realização do padding (etapa 2). No primeiro, são utilizados apenas bits 0, e são acrescentados no menor número possível para ajustar o tamanho da entrada x. Caso este tamanho já seja múltiplo de r, então nenhum bit é acrescentado. Já no segundo, um bit 1 é sempre acrescentado ao final de x, que então é seguido de tantos bits 0 quantos forem necessários para tornar o tamanho de x um múltiplo de r.

A diferença entre esses dois métodos de padding é que o primeiro é ambíguo, ou seja, não é possível saber onde termina x e onde começa o padding. Já no segundo é fácil determinar este limite, bastando, para isto, verificar qual é o último bit 1 em x, a partir do qual o padding inicia. Ainda no segundo modo, já tendo x um tamanho adequado, um bit 1 é mesmo assim acrescentado resultando na adição de um novo bloco a x. Este bloco iniciaria com um bit 1 e seria seguido de $r-1$ bits 0. [MEN 96] É importante ressaltar que esses métodos de padding são genéricos, sendo possível encontrar variações nas mais diversas implementações do algoritmo.

Para ser possível uma futura verificação da integridade de uma entrada *x*, tanto o modo de realização do padding quanto o valor IV devem ser conhecidos, fazendo com que eles sejam padronizados em cada algoritmo de hash.

A etapa três pode parecer, num primeiro momento, sem sentido, mas a mesma é realizada a fim de evitar um eventual problema de segurança, decorrente da possibilidade de duas entradas de tamanhos diferentes produzirem o mesmo valor de hash. Essa técnica é referenciada por alguns autores como MD-strengthenig. [SCH 96]

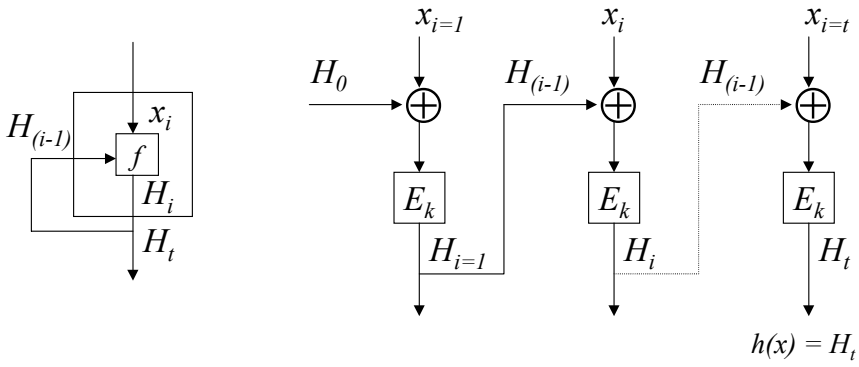
Entre as funções de hash, existem dois tipos específicos: Modification Detection Codes (MDC) e Message Authentication Codes (MAC). As funções do tipo MDC são específicas para o controle de integridade, enquanto as funções MAC incluem também recursos para a autenticação da massa de dados, e por isso acabam tendo propriedades adicionais àquelas existentes nos MDCs.

3.2.2 Modification Detection Codes (MDC)

Esse é o tipo de função de hash mais comumente implementado. Embora seja específico para a realização de simples controle de integridade, na prática, é também bastante utilizado para a obtenção de MACs. Existem diversas formas de se criar um MDC. Entre estas, as principais são comentadas a seguir.

Uso de cifras de bloco

Pode-se construir um MDC a partir de um algoritmo de chaves simétricas que trabalhe com blocos, tal como uma função de hash. A idéia é utilizar o algoritmo em modo Cipher Block Chaining (CBC), já que neste modo, o resultado da cifragem de um bloco é utilizado na cifragem do bloco seguinte, resultando daí um efeito avalanche idêntico àquele das funções de hash. O resultado da cifragem do último bloco é o valor de hash. Como pode ser visto na figura abaixo, onde E_k representa a cifragem em função da chave *k*, as iterações de uma cifra de bloco em modo CBC são bastante semelhantes às de uma função interna de hash.



(a) função interna de hash

(b) iterações de uma cifra de bloco utilizando o modo CBC

FIGURA 3.5 – Funções de hash com cifras de bloco CBC

Fonte: a) [MEN 96], p.332. b) [SCH 96], p.194.

O algoritmo normalmente utilizado para este fim é o Data Encryption Standard (DES). Ainda, é possível utilizar a cifra também em modo Cipher-Feedback (CFB), uma vez que, assim como no CBC, o resultado de cada etapa interfere diretamente no da etapa seguinte. [SCH 96]

Apesar das semelhanças com as funções de hash, existem alguns problemas no uso de cifras de bloco para a geração de hashes:

a) os algoritmos de chave simétrica trabalham normalmente com blocos de 64 bits, valor inferior aos 128 bits recomendados na seção 3.2, o que, mesmo com a proposição de vários esquemas para o ajuste do tamanho do hash, os tornam suscetíveis a ataques;

b) o gerenciamento da chave *k* utilizada para gerar o hash é outro problema, pois a mesma chave deve ser utilizada sempre quando houver necessidade de verificação; o fato da chave ter que ser conhecida gera novas vulnerabilidades a serem exploradas por atacantes. [SCH 96]; e

c) outro problema conhecido é a baixa velocidade em relação às funções customizadas de hash. [MEN 96]

Devido a essas dificuldades e ao grande número de propostas buscando solucionar cada uma delas, existem várias dúvidas pairando sobre o uso de cifras de bloco para fins de geração de hashes, exigindo dessa forma, que uma análise cuidadosa do problema a ser resolvido seja realizada antes do emprego de um hash obtido através de cifras de bloco.

Uso de algoritmos de chaves assimétricas

É também possível utilizar um algoritmo de criptografia de chaves assimétricas para gerar um hash. Para tanto, basta que um par de chaves, K_{pv} e K_{pb}, seja gerado e que a chave privada K_{pv} seja imediatamente destruída. Pode-se, então, cifrar uma determinada massa de dados com K_{pb}. O resultado desta cifragem é o valor do hash. [SCH 96]

Tendo-se como base as propriedades inerentes desses algoritmos, a unidirecionalidade do processo é garantida com a destruição de k_{pv}. Sendo assim, atacar este hash é tão difícil quanto tentar ler uma mensagem cifrada sem ter a chave privada correspondente.

As únicas desvantagens que tornam este método não recomendável, é o seu desempenho, uma vez que ele é ainda mais lento que o uso de cifras de bloco e o tamanho do hash, que varia de acordo com o tamanho da massa de dados. [SCH 96]

Uso de aritmética modular

O princípio neste tipo de solução é construir uma função de hash a partir de uma função de compressão f baseada em aritmética de módulo M. Embora seja

aparentemente simples de ser implementado, este tipo de função não é muito popular devido a um histórico de propostas de implementação inseguras, além de também serem lentos em relação às funções customizadas. [MEN 96]

Funções de hash customizadas

Estas funções foram projetadas desde o seu início para o propósito de geração de hashes. Sendo assim, elas não sofrem das limitações impostas pelo uso de mecanismos já existentes, como a cifragem de blocos e a aritmética modular, tornando-as também independentes da existência desses subcomponentes para sua aplicação. Dessa forma, requisitos como o tamanho do hash gerado e a velocidade, puderam ser priorizados, resultando em importantes vantagens sobre as outras funções aqui colocadas.

Entre os diversos algoritmos existentes, o Message Digest 4 (MD4) pode ser tido como um marco histórico, pois é nele que são baseados os algoritmos de hash atualmente mais populares, que compõem praticamente o padrão em relação à eficiência e segurança para a geração de hashes. Entre estes últimos, têm-se: MD5 e SHA1.

Mais detalhes do MD4 estão compreendidos na seção 3.2.4, enquanto o MD5 e o SHA1 são comentados, respectivamente, nas seções 3.2.5 e 3.2.6. Outros algoritmos, como o RIPEMD e HAVAL, não serão aqui tratados uma vez que foram considerados de baixa relevância para o desenvolvimento deste trabalho, principalmente pela pouca popularidade e conseqüente baixa disponibilidade de implementações.

3.2.3 Message Authentication Codes (MAC)

As funções de hash do tipo MAC também possuem as mesmas propriedades das MDCs, mas permitem não só o controle de integridade de uma massa de dados como também a sua autenticação. Estas funções recebem como entrada, além da massa de dados x, uma chave k. A chave é simétrica, uma vez que a mesma deve ser utilizada para a verificação da massa de dados. [SCH 96]

Dessa forma, a propriedade adicional de uma MAC é a impossibilidade, ou dificuldade, de gerar um hash válido sem que a chave utilizada seja conhecida. Esta propriedade pode ser facilmente utilizada para proteger o snapshot, conforme colocado na seção 2.3.4, já que um atacante seria impossibilitado de gerar uma entrada válida no snapshot sem o conhecimento da chave secreta utilizada pelo mecanismo de controle de integridade.

Assim como nas MDCs, há vários meios de se obter uma MAC. Talvez o mais óbvio é o uso de cifras de bloco. A implementação é idêntica à de um MDC, só que a chave k utilizada no processo é mantida secreta. O problema da velocidade da cifra de bloco também é herdado pela MAC. A seguir, são comentadas de forma destacada outros dois meios de obtenção de MACs.

MACs a partir de MDCs

Apesar de não terem sido criadas especificamente para geração de MACs, as MDCs se apresentam como base comumente utilizada para a implementação desse tipo de função. O princípio aqui é fornecer a chave secreta *k* junto com a massa de dados *x* como entrada para a função MDC. O que num primeiro momento parece ser simples de ser realizado, é uma tarefa bastante delicada, pois caso seja realizada de maneira inadequada pode comprometer toda a segurança do MAC.

A seguir são comentados os métodos que podem ser utilizados para inserção de uma chave secreta em um MDC. Em todos eles a chave *k* deve ter um tamanho igual ao do bloco processado pelo algoritmo. Os dois métodos mais óbvios de inserir uma chave secreta em um MDC são [MEN 96]:

a) chave como prefixo: neste caso o MAC *M* é dado por $M = h(k,x)$, onde *k* é concatenada ao início de *x*; o que parece ser seguro na verdade não é, pois considerando uma função de hash sem uma função de transformação *g* (ver FIGURA 3.4), é possível a alteração de *M* mantendo-o válido sem ter conhecimento da chave *k*; isto é possível uma vez que o novo MAC *M'* seria dado por $M' = h(k,x,y)$, onde *y* são os dados adicionais inseridos pelo atacante; o atacante não tem *k* nem *x*, mas tem *M* que corresponde a $h(k,x)$, assim $M' = h(M,y)$; isto é semelhante a tomar um *hash* incompleto e mais tarde concluí-lo processando os blocos restantes de *x*; e

b) chave como sufixo: $M = h(x,k)$, este método também é sucessível a ataques; um atacante que tenha a possibilidade de escolher a entrada *x* a ser utilizada, pode tentar gerar colisões fortes (seção 3.2), mesmo sem conhecimento de *k*.

Partindo-se dessas duas análises, pode-se chegar a sugestão de utilizar um método híbrido, ou seja, usar a chave secreta *k* tanto no início quanto no final da geração do MAC. [MEN 96] Assim, pode-se fazer:

$$M = h(k,x,k)$$

Outro método, considerado ainda mais seguro em [MEN 96] é:

$$M = h(k,h(k,x))$$

Neste caso, primeiro é calculado um MAC intermediário com a chave prefixada à entrada *x*, $M' = h(k,x)$. Em seguida, o MAC final é calculado com a mesma chave *k* sendo prefixada ao MAC intermediário, $M = h(k,M')$. Além destas duas construções uma outra variação é proposta em [SCH 96]:

$$M = h(k_1,h(k_2,x))$$

Esta variação é bastante semelhante à anterior. A única diferença é a utilização de chaves diferentes no cálculo do MAC intermediário e no cálculo do MAC final. *M'* seria então dado por $h(k_2,x)$, e *M* por $h(k_1,M')$.

Tanto em [SCH 96] quanto em [MEN 96], o penúltimo método é considerado suficientemente seguro para a inserção de uma chave em um MDC. Apesar de exigir que a função de hash seja aplicada duas vezes não há perda significativa de

desempenho, visto que a segunda aplicação processa somente dois blocos, ou no máximo três, dependendo da forma como o padding é realizado pelo algoritmo empregado.

Funções MAC customizadas

Embora sejam poucas, existem algumas funções deste tipo. São encontradas em menor número devido à eficiência de MACs obtidas a partir de MDCs, as quais já se encontram largamente difundidas. [MEN 96]

Entre os algoritmos existentes para MACs, tem-se o Message Authenticator Algorithm (MAA), que apesar de não ser recente, datando de 1983, não teve a mesma aceitação que algoritmos de hash como o MD4 e MD5 tiveram. Um dos seus problemas é a velocidade, cerca de duas vezes mais lento que o MD4, e outro é a segurança, pois ainda existem divergências sobre sua construção. [SCH 96]

3.2.4 Message Digest 4 (MD4)

O número 4 em uma série de algoritmos criados por Ronald Rivest, no MIT, o MD4 é um MDC especificamente projetado para ser extremamente eficiente em máquinas com arquitetura de 32 bits, gerando hashes de 128 bits. [RIV 92]

O MD4 opera sobre blocos de 512 bits, os quais são subdivididos em 16 blocos de 32 bits, resultando daí um conjunto de 4 blocos de 32 bits cada, que, concatenados, formam o hash de 128 bits. As operações utilizadas internamente compreendem soma, deslocamento, XOR e operações lógicas OR e AND. O MD4 não será aqui detalhado uma vez que o MD5, sucessor do MD4, o será na seção 3.2.5.

Após sua publicação, alguns pesquisadores conseguiram atacar algumas das etapas da operação do MD4, mas não o algoritmo como um todo. Apesar destes ataques não poderem ser estendidos para o algoritmo inteiro, Rivest realizou em seguida algumas modificações no algoritmo de forma a torná-lo mais seguro, resultando na criação do MD5. [SCH 96]

3.2.5 Message Digest 5 (MD5)

O MD5, assim como o MD4, é um Modification Detection Code, e segundo o próprio autor, em [RIV 92a]:

“The MD5 algorithm is an extension of the MD4 message-digest algorithm. MD5 is slightly slower than MD4, but is more "conservative" in design. MD5 was designed because it was felt that MD4 was perhaps being adopted for use more quickly than justified by the existing critical review; because MD4 was designed to be exceptionally fast, it is "at the edge" in terms of risking successful cryptanalytic attack. MD5 backs off a bit, giving up a little in speed for a much greater likelihood of ultimate security. It incorporates some suggestions made by various reviewers, and contains additional optimizations.”

O MD5, de forma semelhante ao MD4, opera com blocos de 512 bits, subdivididos em 16 blocos de 32 bits cada, tal como mostrado na FIGURA 3.6. A operação do MD5 é descrita nos itens abaixo.

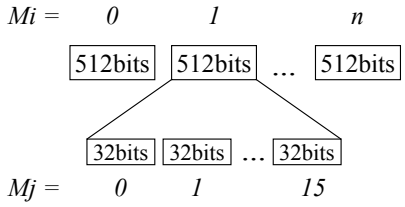


FIGURA 3.6 - Blocos utilizados no MD5

1) inicialização: quatro variáveis de 32 bits, *A*, *B*, *C* e *D*, são inicializadas com os respectivos IVs fixados pelo algoritmo:

   A = 0x01234567
   B = 0x 89abcdef
   C = 0xfedcba98
   D = 0x76543210

2) pré-processamento (*padding*): o tamanho da entrada *M* é ajustado com a concatenação de um bit 1, seguido de tantos bits 0 quantos forem necessários para fazer com que faltem apenas 64 bits para obter-se um múltiplo de 512; tomando-se como exemplo uma entrada com tamanho de 904 bits, seria acrescentado um bit 1 seguido de 55 bits 0, ficando no total 960 bits;

3) pré-processamento (length block): uma representação de 64 bits do tamanho de *M*, antes de sofrer o padding, é concatenada à mensagem, completando os 64 bits restantes para tornar o tamanho da mensagem um múltiplo de 512; trazendo o exemplo do item anterior, 960 + 64 = 1024 bits;

4) laço principal do algoritmo: realizado o ajuste no tamanho da entrada, a mesma é dividida em blocos de 512 bits (M_0, M_1, M_2, ...); a partir daí, o laço principal (FIGURA 3.7) é executado uma vez para cada bloco M_i existente;

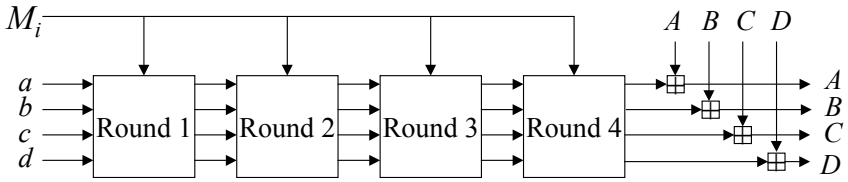


FIGURA 3.7 – Laço principal do MD5
Fonte: [SCH 96], p.437.

   a) os valores de *A*, *B*, *C* e *D* são copiados respectivamente para as variáveis *a*, *b*, *c* e *d*, que então são utilizadas em cada etapa (*round*) do laço principal;

   b) em cada etapa, que são quatro, o bloco M_i recebido é dividido em 16 blocos de 32 bits cada (Mj, $0 \leq j \leq 15$), e então cada um desses blocos é

processado, resultando na execução de 16 passos em cada etapa, o que pode ser chamado de laço interno do algoritmo;

c) em cada um dos 16 passos, três das variáveis (*a*,*b*,*c*,*d*) são utilizadas como entrada para uma função não linear; ao resultado dessa função é somado o valor da variável não utilizada por ela, o bloco de 32 bits *Mj* e uma constante t_p; este resultado intermediário sofre um deslocamento de *s* bits e então é somado a ele o valor de uma das variáveis (*a*,*b*,*c*,*d*); finalmente, o valor de uma dessas variáveis é substituído pelo resultado dessas operações; a figura abaixo ilustra a execução de um passo:

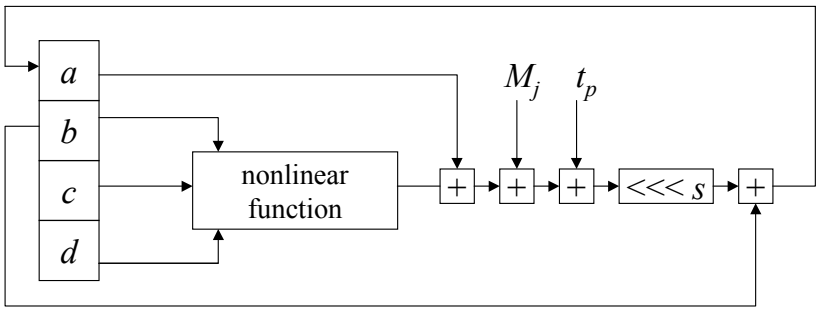


FIGURA 3.8 – Exemplo de um passo do MD5
Fonte: [SCH 96], p.438.

d) encerrada uma etapa, as variáveis (*a*,*b*,*c*,*d*) são passadas para a etapa seguinte;

e) após a realização das quatro etapas, os valores (*A*,*B*,*C*,*D*) são atualizados somando-se a eles os valores de (*a*,*b*,*c*,*d*) respectivamente; assim:

$$A = A + a$$
$$B = B + b$$
$$C = C + c$$
$$D = D + d$$

5) após todos os blocos M_i de 512 bits terem sido processados, o valor do hash é dado pela concatenação de A, B, C e D, resultando em um hash de 128 bits.

Tendo-se visto o algoritmo MD5 de uma forma genérica, é importante que alguns pontos sejam detalhados:

a) as operações de soma sempre são realizadas em módulo 2^{32};

b) as funções não lineares são quatro, uma para cada etapa:

Etapa 1: $F(x,y,z)$ = (x and y) *or* (not(x) and z)
Etapa 2: $G(x,y,z)$ = (x and z) *or* (y and not(z))
Etapa 3: $H(x,y,z) = x$ xor y xor z
Etapa 4: $I(x,y,z) = y$ xor (x or not(z))

c) a partir destas funções, a expressão genérica de um passo de uma etapa seria:

$$ff(v,x,y,z,M_j,s,t_p): v = ((f(x,y,d) + v + M_j + t_p) <<< s) + x$$

onde *f* é uma das funções não lineares, e as variáveis *v*, *x*, *y* e *z* são substituídas por *a*, *b*, *c* e d, não necessariamente nesta mesma ordem; na verdade esta ordem muda a cada passo de uma etapa; a ordem inicial é (*a*,*b*,*c*,*d*), seguida de (*d*,*a*,*b*,*c*), (*c*,*d*,*a*,*b*), (*b*,*c*,*d*,*a*) e então volta-se para (*a*,*b*,*c*,*d*) iniciando novamente a seqüência.

d) a ordem de acesso aos blocos de 16 bits M_j não é necessariamente seqüencial, variando a cada etapa; *j* então varia assumindo os seguintes valores na ordem em que são apresentados:

Etapa 1: { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
Etapa 2: { l, 6, 11, 0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12}
Etapa 3: { 5, 8, 11, 14, 1, 4, 7, 10, 13, 0, 3, 6, 9, 12, 15, 2}
Etapa 4: { 0, 7, 14, 5, 12, 3, 10, 1, 8, 15, 6, 13, 4, 11, 2, 9}

e) a constante t_p é diferente em cada passo; assim, tendo o algoritmo quatro etapas e cada etapa realizando 16 passos, tem-se no laço principal um total de 64 passos; assim, o índice *p* de *t* indica o passo atual do laço principal, e então *t* é dado por: $t = 2^{32} * \text{abs}(\sin(p))$, $1 \leq p \leq 16$; e

f) o valor de *s* também varia a cada passo, mas seguindo uma lógica um pouco diferente; para cada etapa há uma seqüência de quatro valores para *s*; a cada um dos 16 passos realizados (um laço interno), um valor de *s* é utilizado; chegando-se ao último valor, volta-se a utilizar o primeiro; as seqüências de valores de *s* para cada etapa são:

Etapa 1: { 7, 12, 17, 22}
Etapa 2: { 5, 9, 14, 20}
Etapa 3: { 4, 11, 16, 23}
Etapa 4: { 6, 10, 15, 21}

No Anexo 1 encontra-se a listagem completa das quatro etapas, ou 64 passos, do MD5. Descrito e detalhado o MD5, pode-se agora destacar as diferenças entre ele e seu antecessor, o MD4:

a) uma quarta etapa foi adicionada; no MD4 eram somente 3;

b) cada passo tem uma constante aditiva única (*t*); no MD4 as constantes são reutilizadas;

c) a função da etapa dois *G* = ((*x* and *y*) or (*x* and *z*) or (*y* and *z*)) foi trocada por G = (*x* and *z*) *or* (*y* and not(*z*)); o motivo era torná-la menos simétrica;

d) cada passo adiciona o resultado do passo anterior, promovendo um efeito avalanche mais rápido;

e) a ordem em que os blocos de 32 bits M_i eram acessados nas etapas 2 e 3 foi trocada para fazer com que ficassem menos parecidas; e

f) os valores de deslocamento *s* foram otimizados para causar um efeito avalanche melhor; esses valores são diferentes em cada etapa.

Apesar dessas melhorias, alguns estudos descritos em [BOE 93], apontaram a existência de fraquezas no algoritmo que permitiram a geração de colisões em algumas situações especiais (restritas). Felizmente, tal ataque, na prática, não representa qualquer problema para o MD5, nem mesmo serve para afirmar que sua propriedade de resistência a colisões tenha sido violada. [SCH 96] De qualquer forma, o MD5 é hoje um dos algoritmos mais difundidos, estando presente na grande maioria das aplicações que utilizam criptografia para obter algum tipo de segurança, como, por exemplo, Secure Sockets Layer (SSL), Secure Shell (SSH) e Pretty Good Privacy (PGP).

Existe também uma variação chamada MD5-MAC, que nada mais é do que uma adaptação do MD5 para a criação de um MAC. A idéia básica no MD5-MAC é fazer com que a chave *k* interfira em todos os passos do MD5, resultando em um MAC mais seguro. Apesar disso, o MD5-MAC é bem pouco difundido e a literatura consultada não o coloca em destaque.

3.2.6 Secure Hash Algorithm (SHA1)

O SHA1 foi desenvolvido pelo NIST e pela NSA para ser utilizado como o algoritmo de hash padrão para algumas aplicações do governo dos Estados Unidos. [NIS 95]

Também baseado no MD4, opera com blocos de 512 bits, subdivididos em 16 blocos de 32 bits cada. Seu algoritmo é por conseqüência bastante parecido com o do MD5, com o pré-processamento (padding e length block) praticamente idênticos, por exemplo.

Segue abaixo uma comparação das melhorias do MD5 em relação ao MD4 com as modificações existentes no SHA1 [SCH 96]:

a) no SHA1 também foi adicionada uma quarta etapa, mas a segunda e quarta rodadas utilizam a mesma função *f*;

b) no caso da constante *t*, o SHA1 manteve o esquema do MD4, reutilizando as constantes a cada 20 rodadas;

c) o SHA1 não modificou a função *G*, mantendo a versão do MD4;

d) “cada passo adiciona o resultado do passo anterior, promovendo um efeito avalanche mais rápido”; o mesmo foi feito no SHA1, mas uma quinta variável *e* foi adicionada, fazendo com que os ataques ao MD5 descritos em [BOE 93], não possam ser aplicados ao SHA1;

e) o SHA1 implementa o acesso aos blocos de entrada de uma forma completamente diferente do MD4 e do MD5, utilizando um código cíclico de correção de erros; e

f) os valores de deslocamento do MD4 foram mantidos no SHA1.

Ainda, a diferença mais visível do SHA1 é o tamanho do hash gerado, 160 bits ao invés dos 128 bits do MD5 e do MD4. Isso faz com que o SHA seja mais resistente a

ataques, e talvez como conseqüência disso, não há ataques criptográficos conhecidos que tenham sido aplicados com sucesso contra ele. [SCH 96]

Maiores detalhes sobre o algoritmo e sua implementação, podem ser obtidas em [NIS 95].

4 Estado da arte

Vários projetos têm sido desenvolvidos buscando suprir a falta de mecanismos de controle de integridade e de autenticidade de arquivos nos sistemas operacionais. Infelizmente, com raras exceções, esses projetos são muito pouco documentados, ficando limitados à sua implementação e simples manuais de uso.

Todos os projetos estudados durante a realização deste trabalho têm como base do seu mecanismo de controle de integridade o uso de funções de hash, embora alguns forneçam a opção de utilizar também checksums. Outra característica comum é o fato dos mecanismos serem externos ao kernel do sistema, ou seja, são implementados como aplicações que são executadas no espaço de usuário. A única exceção nesta característica é o SecureBSD[†].

Nas seções seguintes são comentados os principais projetos estudados e, na última seção deste capítulo, alguns comentários são tecidos.

4.1 Tripwire

Talvez um dos utilitários mais populares para o controle de integridade de arquivos, o Tripwire teve a sua primeira implementação em 1992, resultante de estudos realizados no Departamente de Ciências da Computação da Universidade de Purdue, Estados Unidos. E desde então, sua estrutura tem servido de modelo para vários outros projetos.

Entre os algoritmos de hash disponibilizados no Tripwire, têm-se: Message Digest 2 (MD2), Message Digest 4 (MD4), Message Digest 5 (MD5), Snefru, HAVAL-128bits, Secure Hash Algorithm (SHA1), CRC-16 e CRC-32. Uma característica interessante do Tripwire é a possibilidade de utilização de algoritmos diferentes para cada arquivo, ou conjunto de arquivos, permitindo que o administrador do sistema escolha o nível de segurança para os arquivos que deseja proteger. Pode-se então, por exemplo, utilizar o CRC-32, que é extremamente rápido, para arquivos de menor importância e o SHA1 para arquivos críticos do sistema. [KIM 95]

Nas primeiras implementações, o snapshot gerado pelo Tripwire não tinha nenhum mecanismo de proteção, tornando-se um ponto extremamente vulnerável. [KIM 95] Este problema foi resolvido nas implementações mais recentes com o uso de assinaturas digitais. Assim, após gerado, o snapshot é assinado digitalmente pelo administrador.

Atualmente existem duas versões do Tripwire[‡], uma de livre distribuição e outra comercial, esta última com recursos bem mais avançados, que permitem, entre outras coisas, a integração de instâncias do Tripwire distribuídas em uma rede de computadores. Mas infelizmente, a despeito do período inicial do projeto, não há

[†] Disponível em http://www.securebsd.org/
[‡] Disponível em http://www.tripwire.org e http://www.tripwire.com

publicações relevantes e recentes originárias deste para o desenvolvimento do presente trabalho.

4.2 Advanced Intrusion Detection Environment (AIDE)

O AIDE é uma iniciativa isolada para a criação de um utilitário livre que substitua o Tripwire. Os seus autores são: Rami Lehti e Pablo Virolainen. Segundo os próprios autores, em [LET 2001], o AIDE implementa funcionalidades adicionais às implementadas na versão livre do Tripwire.

Na verdade o AIDE funciona de maneira semelhante ao Tripwire, utilizando os mesmos algoritmos de hash, mas não implementa qualquer tipo de proteção ao snapshot nem mesmo assinaturas digitais

Visto estes pontos negativos, o AIDE só foi aqui incluído devido a sua relativa popularidade entre os usuários de sistemas operacionais da família Unix.

4.3 SecureBSD

O SecureBSD é um patch para o kernel BSD que busca reforçar a segurança desse kernel. Uma das funcionalidades adicionadas pelo SecureBSD ao kernel BSD é a verificação de integridade de arquivos sob demanda (seção 2.9). Esta característica o diferencia completamente das outras soluções até aqui abordadas, as quais devem ser utilizadas em determinados instantes, sendo disparadas de forma interativa ou automatizada.

A solução apresentada pelo SecureBSD emprega somente o algoritmo de hash MD5 e não protege o snapshot com assinaturas digitais. De fato ela é bastante simples, não havendo preocupações com pontos particulares da segurança e desempenho para verificação de integridade de arquivos.

4.4 Outros Projetos Estudados

Segue abaixo uma lista de outros projetos estudados, mas que, devido a sua semelhança com os já citados ou pouca relevância para o desenvolvimento deste trabalho, não serão aqui detalhados.

- Dragon Squire: é um software comercial que não é específico para o controle de integridade de arquivos, mas sim um sistema de detecção de intrusão onde o controle de integridade é apenas mais um módulo; o algoritmo de hash utilizado é o MD5; disponível em http://www.enterasys.com/ids/squire/;

- Integrit: utiliza o algoritmo de hash MD5, mas não oferece proteção alguma para o snapshot; disponível em http://integrit.sourceforge.net/

- L6: é implementado em Perl, e pode utilizar tanto o MD5 quanto o SHA para a geração de hashes; não oferece qualquer tipo de proteção ao snapshot; disponível em http://www.pgci.ca/l6.html;

- Rocksoft Veracity: software comercial bastante semelhante à versão comercial do Tripwire, embora não trazendo todas as funcionalidades deste último; disponível em http://www.veracity.com/;

- Sentinel: é baseado no algoritmo RIPEMD-MAC, de 160 bits, para a geração de hashes, fornecendo assim algum nível de autenticação ao snapshot; disponível em http://zurk.netpedia.net/zfile.html.

Apesar de não terem sido citadas, diversas outras soluções foram preliminarmente estudadas, sendo que a grande maioria foi considerada irrelevante pela superficialidade com que tratam o problema do controle de integridade, bem como pela falta, por parte de seus autores, de fundamentação teórica sobre os conceitos básicos desse tipo de mecanismo.

4.5 Linux Intrusion Detection System (LIDS)

O LIDS, apesar do nome, é um patch para o kernel do sistema operacional Linux que implementa uma série de controles adicionais sobre os recursos do sistema. Entre esses controles estão [BRE 2001]:

a) proteção de arquivos: arquivos protegidos pelo LIDS não podem ser modificados por nenhum usuário, inclusive o root; arquivos podem ser escondidos do resto do sistema;

b) proteção de processos: processos podem ser protegidos inclusive da ação do root, e, assim como os arquivos, podem ser escondidos;

c) controle granular sobre o acesso a recursos do sistema através de Access Control Lists (ACLs); e

d) bloqueio de acesso a vários recursos do sistema como portas TCP/IP, acesso direto a dispositivos (e.g. /dev/hda), entre vários outros.

Embora não implemente nenhum controle de integridade de arquivos, o LIDS possui características bastante interessantes para a garantia da segurança do snapshot, uma vez que permite a limitação dos poderes do usuário root, resolvendo alguns problemas citados na seção 1.2 (violação da lei do menor privilégio e excesso de direitos concedidos à conta root).

Boa parte da implementação do presente trabalho foi baseada no código fonte do LIDS, bem como em informações obtidas através de contatos com os seus autores.

4.6 Comentários

Entre todas as soluções estudadas, o Tripwire é a mais completa atualmente e é amplamente disponível para a comunidade. Todas as outras, com exceção do SecureBSD, implementam apenas um subconjunto das funcionalidades do Tripwire.

Tanto no Tripwire quanto nos seus semelhantes é difícil evitar um impacto no desempenho do sistema sem que o intervalo de tempo entre cada verificação seja relativamente longo. Este é um dos problemas evitados pelo SecureBSD, o único que implementa uma solução de controle de integridade com verificação sob demanda.

O LIDS, apesar de parecer deslocado no contexto deste trabalho, implementa uma série de modificações no kernel do Linux que acabam sendo semelhantes àquelas que devem ser realizadas para a inserção de um mecanismo de controle de integridade. Ainda, algumas de suas funcionalidades, como o bloqueio total do acesso a determinados arquivos, podem ser facilmente estendidas para fornecer um maior nível de segurança ao mecanismo a ser implementado.

5 Secure On-the-Fly File Integrity Checker

Como resultado dos estudos realizados neste trabalho, foi modelado um mecanismo de controle de integridade que, assim como os citados no capítulo anterior, é baseado no uso de algoritmos de hash. Embora a base seja a mesma, as necessidades de segurança e de desempenho foram aqui priorizadas, resultando em um modelo de um mecanismo seguro de controle de integridade de arquivos.

A este modelo foi dado o nome de Secure On-the-Fly File Integrity Checker (SOFFIC). A verificação da integridade dos arquivos é realizada cada vez que o acesso a eles é requisitado, ou seja, sob demanda (On-the-Fly). Ainda, o SOFFIC é capaz de interceptar o uso de um arquivo inválido, evitando assim a propagação de danos de um possível ataque. Essa é, juntamente com os mecanismos de autoproteção e a verificação sob demanda, uma das principais características que diferenciam o modelo apresentado neste trabalho da maioria das soluções atualmente existentes.

5.1 Objetivos

O SOFFIC deve ser capaz de interceptar requisições para leitura e execução de arquivos de modo que, após realizar a verificação de integridade dos mesmos, possa permitir ou não o acesso.

Para garantir a efetividade do SOFFIC, alguns mecanismos de autoproteção precisam ser utilizados em cada um dos seus componentes. Os requerimentos mínimos desejados para a segurança são definidos com base na seguinte afirmação: “o SOFFIC não deve confiar na conta administrador/root, ao menos não todo o tempo”. Isto pode ser facilmente justificado pelo fato de que a maioria das vulnerabilidades exploradas por agentes maliciosos resultam em comprometimento direto ou indireto da conta administrador/root [ANO 2000]. Ainda, se essa conta fosse completamente segura, os mecanismos de segurança padrão encontrados na família de sistemas operacionais unix, por exemplo, seriam suficientes para garantir a integridade de arquivos importantes, e o projeto SOFFIC não seria necessário.

Considerando os requisitos básicos de operação, o SOFFIC deverá satisfazer os objetivos traçados ao mesmo tempo que garante sua própria segurança e mantém taxas aceitáveis de desempenho.

5.2 Ambientes Ideais de Aplicação

A definição do tipo de ambiente ideal para a aplicação do SOFFIC é fundamental para o pleno entendimento de seus objetivos. Quanto mais estável o ambiente, em termos de criação e modificação de arquivos (principalmente executáveis e de configuração), mais fácil será a aplicação do SOFFIC. Alguns ambientes considerados estáveis e portanto ideais:

a) firewalls;

b) servidores HTTP/FTP;

c) servidores de e-mail; e

d) servidores de Banco de Dados.

A característica comum entre esses ambientes é que não são ambientes de desenvolvimento, pelo menos quando bem administrados.

Uma aplicação interessante do SOFFIC se dá em servidores HTTP, onde ele pode proteger um site contra ataques cujo objetivo seja a alteração de páginas publicamente disponíveis. Nesses casos, cada vez que o servidor HTTP for acessar um arquivo, o SOFFIC irá verificar a sua integridade e, no caso de uma violação ser detectada, o acesso será negado. Assim, ao invés da página modificada pelo atacante ser exibida para o cliente, o servidor irá retornar um erro relatando que o acesso não foi permitido, resguardando a reputação da organização. De forma bastante semelhante, o SOFFIC se aplica aos servidores FTP, evitando a distribuição de arquivos violados.

Um ambiente ideal para o SOFFIC é basicamente aquele onde o usuário não é desenvolvedor. Desse modo, não são criados a todo momento novos arquivos executáveis ou de configuração, o que exigiria uma constante atualização dos dados do SOFFIC para que esses novos arquivos pudessem ser acessados. Ainda, a simples ausência de compiladores, normalmente desnecessários para a operação normal desses sistemas, já constitui mais uma barreira a dificultar as atividades de um agente malicioso.

5.3 Arquitetura

Os componentes do SOFFIC estão presentes em diferentes partes do sistema, alguns no kernel e outros no sistema de arquivos. A disposição desses componentes é ilustrada na figura abaixo.

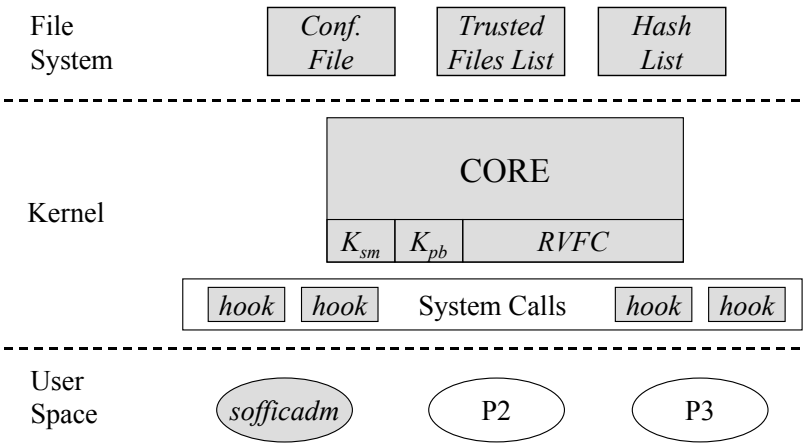


FIGURA 5.1 – Arquitetura do SOFFIC

a) Core: é o núcleo do SOFFIC, nele estão inseridas as funções responsáveis pela carga da ferramenta, tratamento dos *hooks* (ganchos) inseridos nas chamadas de sistema, realização da verificação de integridade, entre outras;

b) Configuration File: arquivo contendo as configurações básicas do SOFFIC;

c) Hash List (HL): um arquivo contendo a lista de hashes a ser utilizada no processo de verificação, ou seja, o snapshot;

d) Trusted Files List (TFL): uma lista de arquivos que podem ser lidos ou executados sem qualquer verificação. Arquivos que não estão presentes na HL e nem na TFL, a princípio, não podem ser lidos ou executados;

e) par de chaves assimétricas (K_{pb}, K_{pv}): K_{pv} fica sob poder do administrador do SOFFIC e é utilizada para a geração de assinaturas. Somente a chave pública K_{pb} fica disponível ao SOFFIC;

f) chave simétrica (K_{sm}): chave utilizada tanto pelo administrador quanto pelo SOFFIC para a composição de MACs ou para o uso em algoritmos de chaves simétricas;

g) Recently Verified Files Cache (RVFC): uma cache para evitar a execução de um grande número de verificações de um mesmo arquivo em um curto espaço de tempo, beneficiando o desempenho;

h) ganchos nas chamadas de sistema (hooks): são modificações em determinadas chamadas de sistema que desviam o controle para o SOFFIC;

i) sofficadm: uma aplicação através da qual o administrador do SOFFIC executa operações administrativas, como por exemplo: geração da HL e assinaturas digitais.

Alguns desses componentes são detalhados nas seções seguintes.

5.3.1 Core (núcleo do SOFFIC)

O Core é inserido no kernel do sistema, onde desempenha as seguintes funções básicas do SOFFIC:

a) abertura/leitura de arquivos;

b) geração de *hashes* de arquivos ou qualquer outra massa de dados;

c) cifragem/decifragem com algoritmos de chaves simétricas; e

d) verificação de assinaturas digitais;

e algumas funções mais específicas:

e) inicialização do SOFFIC;

f) operações na HL (inicialização, pesquisa);

g) operações na TFL (inicialização, pesquisa);

h) operações na RVFC (inicialização, pesquisa, manutenção);

i) tratamento dos *hooks* implementados; e

j) controle de acesso aos seus arquivos de configuração.

Este último item garante mais um nível de proteção para todos os arquivos de configuração do SOFFIC contidos no seu diretório. Quando acionada essa proteção, somente o próprio kernel consegue listar ou acessar qualquer arquivo neste diretório.

5.3.2 Trusted Files List (TFL)

O objetivo da TFL é tratar arquivos que não devem ser controlados através da Hash List, uma vez que sofrem inúmeras mudanças durante a operação normal do sistema. Entre esses, estão os arquivos temporários e de logs.

A TFL pode ainda referenciar um diretório, fazendo com que qualquer arquivo nele contido possa ser acessado sem verificação alguma. Tal característica permite tratar diretórios como o /tmp, onde novos arquivos são criados e acessados a todo momento. É importante ressaltar que isso é altamente inseguro, podendo facilmente comprometer a segurança do sistema, caso um erro de configuração seja cometido pelo usuário.

5.3.3 Configuration File

No arquivo de configuração, várias opções que controlam o comportamento do SOFFIC podem ser definidas pelo usuário. A tabela abaixo descreve as opções e os valores possíveis para cada uma delas:

TABELA 5.1 – Opções do arquivo de configuração

Descrição	**Valores**
Ação padrão para arquivos não constantes na HL e nem na TFL	DENY/PERMIT
Localização da HL	Caminho completo do arquivo
Localização da TFL	Caminho completo do arquivo
Número de entradas da RVFC	0 (desativada) a 65536

Essa lista não é tida neste momento como completa, pois várias outras opções deverão ser requisitadas a partir do momento que o SOFFIC for sendo utilizado em situações práticas.

5.3.4 Recently Verified Files Cache (RVFC)

A RVFC é composta por uma lista de índices dos arquivos verificados pelo SOFFIC e considerados válidos. A composição desta lista é detalhada no capítulo 6. Como estratégia de gerenciamento é utilizada a Least-Recently-Used (LRU), a qual é amplamente conhecida e aplicada em diversos outros tipos de cache. A LRU faz com que os índices dos arquivos mais freqüentemente utilizados sejam mantidos por mais tempo na RVFC, enquanto os menos utilizados vão sendo mais rapidamente eliminados para dar lugar a novos índices [FOL 92].

5.3.5 Sofficadm

Essa ferramenta é a principal interface de gerenciamento do SOFFIC para o seu administrador, oferecendo meios para a execução das seguintes operações:

a) geração de par de chaves assimétricas;

b) cifragem e decifragem de arquivos (chave simétrica);

c) assinatura e verificação de assinatura de arquivos (chaves assimétricas); e

d) geração das listas HL e TFL.

Assim como no caso do arquivo de configuração (Configuration File), as funcionalidades do sofficadm podem ser expandidas de acordo com as novas necessidades advindas do seu uso em situações práticas.

5.4 Mecanismos de proteção

Durante a sua modelagem, cada um dos componentes do SOFFIC teve os seus requisitos de segurança analisados e avaliados com o objetivo de definir os mecanismos de segurança a serem empregados em cada um deles. Como primeiro resultado dessa análise, foi obtida a indicação das proteções necessárias para cada componente do SOFFIC (TABELA 5.2).

Os componentes que necessitam de sigilo não podem ser expostos a nenhuma outra entidade senão ao próprio SOFFIC e ao seu administrador. Pode-se traduzir o sigilo como proteção contra leitura não autorizada. A integridade, exigida em todos os componentes, visa garantir que nenhum deles seja modificado ou substituído, e para tanto, deve-se ter proteção contra escrita. A última coluna da tabela, denota a necessidade de autenticação, ou seja, antes de um componente ser efetivamente utilizado, não só a sua integridade deve ser conferida mas também sua origem deve ser autenticada.

TABELA 5.2 – Requisitos de segurança dos componentes do SOFFIC

Componente	**Requisitos de segurança**		
	Sigilo	Integridade	Autenticação
Core em disco		•	•
Core em memória		•	
Configuration File		•	•
Hash List		•	•
Trusted Files List	•	•	•
Chave pública K_{pb}		•	
Chave privada K_{pv}	•		
Chave simétrica K_{sm}	•	•	
RVFC		•	•
Sofficadm		•	•

A partir da indicação das proteções necessárias para cada componente, uma lista dos possíveis mecanismos aplicáveis a cada um deles foi elaborada, resultando na tabela abaixo:

TABELA 5.3 – Mecanismos de proteção aplicáveis aos componentes do SOFFIC

Componentes	**Mecanismos de proteção**				
	MACs	Cifragem Simétrica	Assinatura digital	Mídias seguras	Controle de acesso
Core em disco	•		•	•	•
Core em memória					•
Configuration File			•	•	•
Hash List	•		•	•	•
Trusted Files List		•	•	•	•
Chave pública K_{pb}		•		•	•
Chave privada K_{pv}		•		•	•
Chave simétrica K_{sm}				•	•
RVFC	•				•
Sofficadm			•	•	•

A última coluna não faz referência somente aos mecanismos de controle de acesso existentes no sistema operacional, mas também àqueles adicionados pelo uso de patches do kernel, tal como o LIDS. Esses últimos, embora sejam de uso recomendado, não são essenciais para garantir a segurança do SOFFIC.

Nas seções seguintes, a segurança de cada componente é discutida em maiores detalhes. A segurança do Core é tratada em duas seções, uma para cada estado (em memória e em disco).

5.4.1 Core em disco

O Core é o componente vital do SOFFIC. Nele estão contidos diversos componentes críticos conforme visto na seção 5.3.1, e por sua vez, ele está contido no kernel. Dessa forma, o objeto a ser protegido é a imagem do kernel existente em disco. Caso um atacante consiga modificar a imagem do kernel, ele pode comprometer o funcionamento dos diversos mecanismos de segurança, incluindo-se aí o SOFFIC. Este não é um ataque fácil de ser realizado, porém é perfeitamente possível.

Buscando proteger a imagem do kernel, uma entrada para ela é inserida na HL. Como pode ser constatado na seção 5.4.4, a imagem do kernel é de fato assinada digitalmente.

É claro que se o Core do SOFFIC for alterado pelo atacante, a proteção pode eventualmente ser desativada. Ainda, também é possível que a imagem do kernel seja substituída por outra sem o SOFFIC. Nesses casos, não há proteção que o SOFFIC possa fornecer e, portanto, outros mecanismos devem ser adotados, como por exemplo, a gravação da imagem do kernel em uma mídia segura (CD-ROM), a partir de onde a mesma deve ser carregada durante o processo de boot.

5.4.2 Core em memória

Após ter sido carregado e a assinatura digital de sua imagem em disco ter sido verificada pelo SOFFIC, o kernel do sistema permanece todo em memória, não havendo mais consultas a sua imagem em disco. Assim, caso um atacante consiga efetuar a difícil tarefa de manipular o conteúdo do kernel em memória, ele pode:

a) causar a paralisação do kernel ou de alguns dos seus subsistemas;

b) promover o funcionamento inadequado de qualquer subsistema; e

c) provocar alterações cuidadosas a fim de subverter o funcionamento de mecanismos de segurança, sem causar problemas ao resto do kernel.

Esse ataque é similar ao comentado na seção anterior e, da mesma forma, embora o nível de dificuldade seja considerável, sua implementação é possível. O SOFFIC somente consegue oferecer algum nível de segurança contra esse tipo de ataque em casos onde apenas arquivos monitorados possam ser executados, ou seja, arquivos presentes na HL. Dessa forma, a ferramenta desenvolvida pelo atacante para comprometer a área de memória do kernel, mesmo que fosse compilada, não poderia ser executada. Para tanto, o atacante precisaria subverter, antes, os mecanismos de proteção da HL, os quais estão descritos na seção 5.4.4.

5.4.3 Arquivo de Configuração

Alterações não autorizadas nesse arquivo podem resultar no mau funcionamento do SOFFIC, comprometendo assim a segurança por ele fornecida. Não havendo informações consideradas sigilosas no arquivo de configuração, bastaria que o mesmo

fosse resguardado de procedimentos de escrita não autorizados. Por outro lado, buscando incorporar o princípio do menor privilégio no SOFFIC, esse arquivo é protegido também contra leituras indevidas.

A proteção contra escrita é aqui fornecida através do mesmo método aplicado ao Core em disco, ou seja, também é criada uma entrada para o arquivo de configuração na HL. Já a proteção contra leitura é obtida com a cifragem do arquivo, utilizando-se a chave simétrica K_{sm}.

5.4.4 Hash List

Cada entrada da Hash List é por si só segura, do ponto de vista da leitura indevida, pois, como foi visto anteriormente, na seção 3.2, um atacante não pode atacar de outra maneira o hash que não seja por força bruta. Já no caso do atacante obter acesso de escrita, ele poderia comprometer a Hash List substituindo ou inserindo hashes de arquivos por ele modificados e ou criados.

A Hash List é assinada digitalmente com a chave privada K_{pv} e adicionalmente, pelos mesmos motivos comentados na seção anterior, ela é cifrada com a chave simétrica K_{sm}.

O fato da Hash List ser assinada faz com que todos os arquivos cujos hashes estão nela contidos sejam indiretamente assinados, uma vez que o hash representa de forma única o conteúdo de cada um deles.

5.4.5 Trusted Files List

A Trusted Files List é um componente bem mais sensível que a HL, já que as suas entradas fazem referência direta a arquivos e diretórios de arquivos que podem ser lidos e/ou executados sem verificação.

Um atacante que consiga apenas ler a TFL, pode utilizar os dados nela contidos para procurar um arquivo a ser modificado ou mesmo um diretório cujos arquivos são executados sem verificação. Caso um desses diretórios seja encontrado, o atacante poderia lá inserir os seus programas e, a partir daí, dar seqüência ao seu ataque.

A TFL deve ser protegida tanto contra escrita quanto contra leituras indevidas, assim, ela é obrigatoriamente cifrada com a chave simétrica K_{sm}, logo após ter sido criada uma entrada para ela na HL.

5.4.6 Par de chaves assimétricas (K_{pv}, K_{pb})

Cada uma dessas chaves tem requisitos diferentes de segurança, como pôde ser visto anteriormente na TABELA 5.2.

A chave privada K_{pv} é mais facilmente protegida por ser necessária apenas nos momentos de manutenção do SOFFIC e, portanto, mesmo que ela seja destruída ou

invalidada, o SOFFIC continuará funcionando normalmente. Apenas o administrador terá que obter a chave privada a partir de uma cópia de segurança, ou gerar um novo par de chaves.

A única proteção necessária para a K_{pv} é contra leitura, ou melhor, uso indevido para geração de assinaturas. Essa proteção é facilmente obtida uma vez que em praticamente todos os esquemas de assinatura digital, a chave privada é por padrão protegida por uma chave simétrica, de conhecimento exclusivo do seu detentor. Essa chave é independente da chave K_{sm} utilizada pelo SOFFIC e, por questões de segurança, elas não devem ser iguais.

Não podendo o atacante fazer uso da chave privada sem o conhecimento da chave que a protege, só resta a ele o ataque da força bruta. Ainda, embora a destruição ou invalidação da chave privada não comprometa o funcionamento do SOFFIC, é recomendável que o administrador a mantenha em uma mídia off-line.

Já a chave pública K_{pb} pode ser lida por qualquer entidade, seja um usuário do sistema ou um atacante, sem que o processo de verificação de assinaturas seja comprometido. No entanto, existem dois ataques possíveis à chave pública K_{pb}: a destruição ou invalidação e a substituição. Como conseqüência do primeiro ataque o procedimento de carga do SOFFIC é comprometido, fazendo com que o sistema não possa ser reinicializado até que a chave pública original seja restaurada. O segundo ataque seria mais crítico para o SOFFIC, pois caso um atacante consiga substituir a chave pública original por outra, para a qual ele tem a chave privada correspondente, ele poderá gerar assinaturas válidas, como se fosse o administrador.

Antes de ter sido carregada, a proteção empregada para a chave pública K_{pb} pode variar segundo o local onde ela se encontra armazenada. Caso ela tenha sido inserida no Core no momento da compilação do kernel, ela também é protegida pelos mecanismos empregados no Core em disco (seção 5.4.1). Nos casos em que a chave pública K_{pb} é mantida em arquivo, externa ao Core até o momento da carga, ela é cifrada com a chave simétrica K_{sm}. O atacante então somente poderia substituí-la em disco conhecendo a chave K_{sm} utilizada. Ainda, é recomendado o uso de uma mídia segura, tal como um CD-ROM ou outra cujos mecanismos de proteção não dependam do sistema operacional.

Após a carga, não sendo a chave pública K_{pb} necessária para a operação normal do SOFFIC, ela é eliminada da memória.

5.4.7 Chave simétrica (K_{sm})

A chave simétrica K_{sm}, como pode ser constatado na seção 5.4.10, é componente necessário, direta ou indiretamente, para o Core obter acesso a todos os outros componentes. Sendo assim, ela deve estar diretamente disponível a ele.

Sendo ela necessária apenas nos momentos de carga e de manutenção do SOFFIC, a mesma não precisa ficar permanentemente disponível. Assim, em favor de uma maior segurança, a chave simétrica K_{sm} é de conhecimento exclusivo do administrador do SOFFIC, não sendo mantida qualquer cópia no sistema. No momento

do boot, a chave é informada, utilizada pelo Core para acessar os arquivos necessários, e em seguida destruída.

Evita-se, assim, qualquer tentativa de um atacante obter a chave através de pesquisas na memória do kernel. A chave simétrica somente pode ser comprometida por descuido do próprio administrador.

5.4.8 Recently Verified Files Cache

A RVFC, ao mesmo tempo que beneficia o desempenho do SOFFIC, constitui um ponto crítico para a segurança. Um dos ataques que poderia ser empregado contra o SOFFIC é a alteração e utilização de um arquivo para o qual existe uma entrada na RVFC. Para proteger o SOFFIC desse tipo de ataque, as modificações de arquivos são monitoradas e, quando um arquivo correspondente a uma entrada existente na RVFC é modificado, ela é imediatamente eliminada.

Assim como a HL, a RVFC pode ser livremente lida sem que a segurança seja comprometida. Sendo a RVFC um componente interno ao Core, ela é protegida pelos mesmos meios que o Core em memória.

5.4.9 Sofficadm

A integridade do sofficadm precisa ser garantida, pois caso um atacante consiga alterá-lo, backdoors podem ser facilmente inseridos de forma a revelar ao atacante a chave simétrica K_{sm} ou mesmo a chave de proteção da chave privada K_{pv}, que então poderia, posteriormente, modificar sem restrições os arquivos do SOFFIC.

Para evitar esse tipo de ataque, uma entrada para o sofficadm é também adicionada a HL, e ainda, como ele só é necessário para a manutenção do SOFFIC, o sofficadm pode ser mantido em uma mídia off-line juntamente com a chave privada K_{pv}.

5.4.10 Interdependência de componentes

A fim de tornar mais claras as relações de dependência dos componentes do SOFFIC em relação ao fornecimento de segurança, é apresentado na figura a seguir um diagrama que ilustra essas dependências:

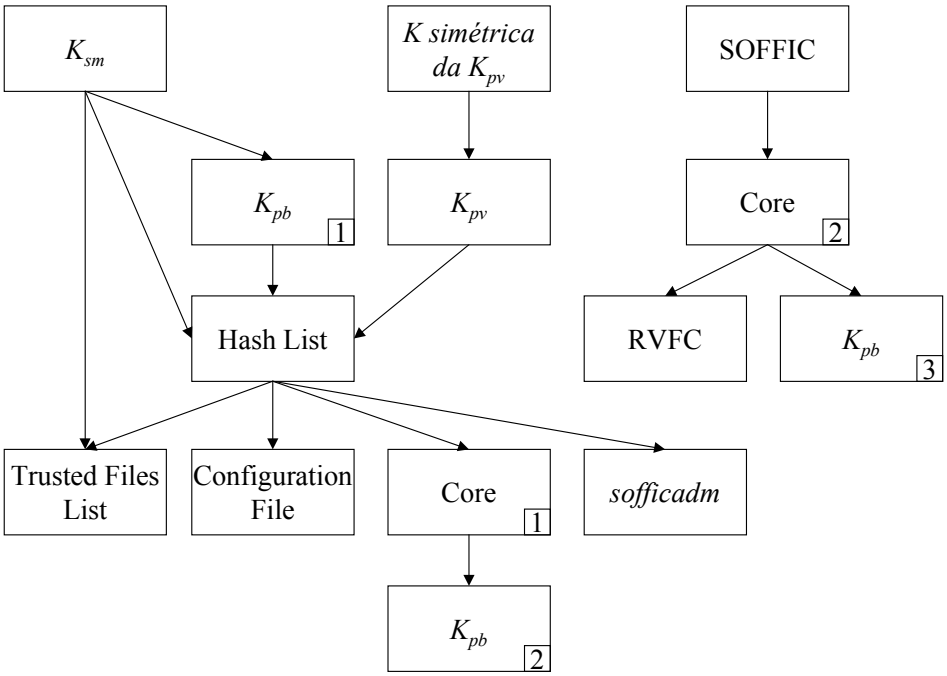


FIGURA 5.2 – Diagrama de dependência de segurança entre componentes

Na figura, o sentido da seta indica o componente dependente. Componentes que são protegidos de várias formas, variando de acordo com uma situação em particular (e.g. se estão em memória ou em disco), foram replicados. A replicação é indicada pelo número no canto inferior direito de cada bloco replicado. Finalmente, o retângulo indicando SOFFIC, significa que a segurança fornecida é dependente do funcionamento normal do mesmo.

5.5 Aspectos Funcionais

Nesta seção são descritos alguns aspectos do funcionamento do SOFFIC, a fim de fornecer maiores recursos para a compreensão de suas diversas operações e conseqüentemente da implementação realizada. É importante ressaltar que esses aspectos, assim como os descritos nas seções anteriores, são independentes da plataforma de aplicação do SOFFIC.

5.5.1 Preparação para uso

Para colocar o SOFFIC em funcionamento é necessário que o administrador execute uma série de passos a fim de arranjar o ambiente de operação. Assim, considerando as relações de dependência dos componentes do SOFFIC, os passos para configurar o ambiente devem respeitar a seguinte ordem:

a) criação da chave assimétrica (K_{sm}): este componente depende somente do administrador do SOFFIC, e deve ser cuidadosamente escolhida, tal como uma senha de acesso a um sistema;

b) geração do par de chaves assimétricas (K_{pv}, K_{pb}): o par de chaves é gerado através do sofficadm; no momento da geração a chave simétrica que protege a chave privada K_{pv} deve ser escolhida pelo administrador;

c) edição e cifragem do arquivo de configuração;

d) geração da Trusted Files List: essa lista é gerada através do sofficadm;

e) geração da Hash List: assim como a TFL, o *sofficadm* é também utilizado para sua geração; na HL além dos arquivos comuns do sistema, devem constar os hashes dos componentes citados na seção 5.4: TFL, Configuration File, Core e o sofficadm; e

f) cifragem dos componentes com k_{sm}: gerados os arquivos acima, pode-se realizar o último passo, a cifragem deles com a chave simétrica k_{sm}; este passo também é executado com o uso do sofficadm.

É importante ressaltar a importância dos procedimentos comentados no capítulo 2, que visam garantir a segurança para o controle de integridade. Eles são válidos tanto para o SOFFIC quanto para qualquer outro mecanismo de controle de integridade de arquivos.

5.5.2 Procedimento de carga do SOFFIC

O SOFFIC entra em operação durante a carga do kernel, logo após os sistemas de arquivos terem sido inicializados. Cada um dos passos de carga do SOFFIC são descritos nos itens abaixo:

a) inicialização das estruturas de memória e obtenção da chave simétrica K_{sm}: a chave é informada pelo administrador;

b) decifragem dos arquivos: nesta etapa os arquivos cifrados com K_{sm} são decifrados, entre eles: chave pública K_{pb} (quando mantida em arquivo), HL, TFL e Configuration File; por fim, K_{sm} é eliminada da memória;

c) verificação da assinatura e carga da HL em memória: sendo a assinatura inválida a carga do sistema é incondicionalmente paralisada, exigindo intervenção do administrador para recuperar o sistema;

d) verificação da integridade do Core em disco: sendo inválido, o mesmo procedimento de paralisação adotado no item anterior é aqui aplicado;

e) verificação da integridade do Configuration File: sendo detectada uma violação deste arquivo, as opções padrão são assumidas;

f) verificação da integridade e carga da TFL: somente carregada se for válida, caso contrário é ignorada e somente os arquivos mantidos na HL poderão ser acessados; e

g) inicialização da RVFC: inicializada a cache o SOFFIC é definitivamente habilitado e dá-se continuidade à carga do sistema.

Tanto o sucesso quanto a falha de todos os procedimentos são registrados em logs. Os comportamentos de paralisação foram definidos neste primeiro momento de forma a fornecer uma maior segurança da integridade do sistema, e não de sua disponibilidade. O tratamento de tal aspecto certamente exigirá algum nível de flexibilidade dos comportamentos do SOFFIC em casos de falhas. Embora a disponibilidade seja um aspecto crítico em vários sistemas, o presente trabalho prioriza a integridade dos mesmos.

5.5.3 Verificação de integridade e bloqueio de acesso

Uma vez instalado e carregado, o SOFFIC entra imediatamente em operação. Assim, quando um processo (P2 na FIGURA 5.3) requisita a abertura de um arquivo ao sistema operacional, o kernel aplica os mecanismos padrão de controle de acesso (SACM) implementados pelo sistema de arquivos utilizado e, se o acesso for autorizado, ocorre o desvio do fluxo normal da chamada de sistema responsável pela abertura do arquivo, passando o controle para o SOFFIC.

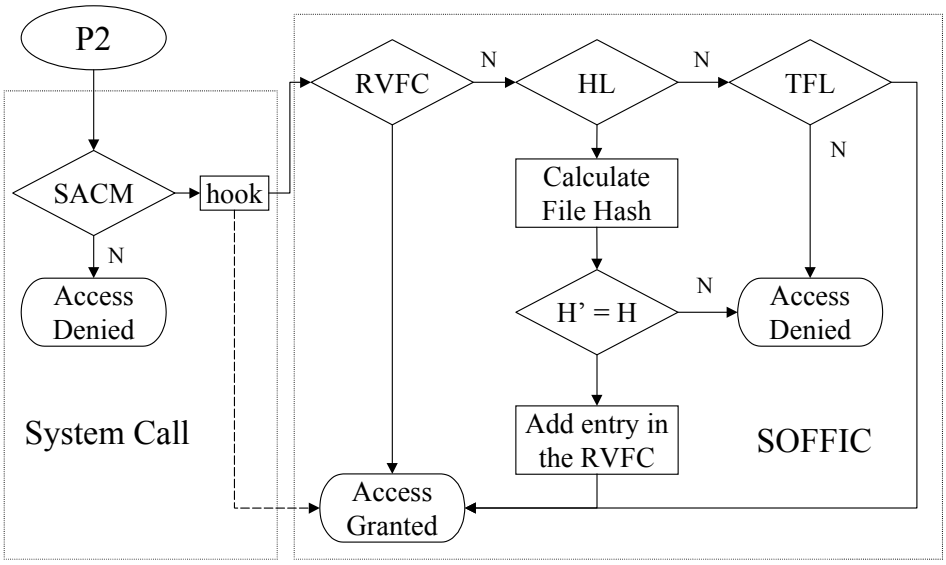


FIGURA 5.3 – Procedimento de verificação de integridade

É importante destacar que o SOFFIC somente entra em ação depois dos mecanismos de controle de acesso do sistema, dessa forma, evita-se um maior consumo de recursos com verificações desnecessárias.

Uma vez acionado o SOFFIC, a RVFC, a HL e a TFL, são nesta exata ordem pesquisadas a procura de uma entrada para o arquivo requisitado. Se uma entrada na RVFC é encontrada, o acesso é permitido sem verificação. Caso contrário, a HL é pesquisada. Sendo localizada uma entrada na HL, o SOFFIC calcula o hash do arquivo em disco e compara-o com o hash armazenado. Caso o resultado da comparação seja positivo o acesso é permitido, senão, negado. No primeiro caso (acesso permitido), é criada uma entrada na RVFC para o arquivo utilizado.

Se, no exemplo citado no parágrafo anterior, o arquivo requisitado pelo processo P2 estivesse na TFL, então nenhuma verificação seria realizada e o acesso seria permitido. Em casos incomuns, mas possíveis, em que um determinado arquivo é listado tanto na HL quanto na TFL, a primeira tem precedência sobre a segunda. Finalmente, não existindo entradas em ambas as listas para um determinado arquivo, o acesso é negado.

6 Protótipo

Buscando validar o modelo apresentado quanto à sua eficiência e aplicabilidade, um protótipo foi implementado utilizando-se o sistema operacional GNU/Linux. Este sistema foi escolhido devido ao fato do kernel Linux ter seu código completamente aberto e licenciado sob a General Public License (GPL). Desenvolvido o protótipo, análises de desempenho foram realizadas a fim de verificar o impacto do uso do SOFFIC no funcionamento do sistema.

No início dos primeiros estudos, a versão estável mais atual do kernel[†] do Linux era a 2.4.3. Desde então, visando evitar potenciais dificuldades advindas de atualizações (modificações) existentes nas versões seguintes do kernel, o desenvolvimento foi mantido até o momento sobre a versão inicialmente utilizada.

Antes do início da implementação, foi necessária a realização de estudos sobre a estrutura do kernel, bem como das técnicas e práticas de programação mais comumente nele aplicadas. Isso foi realizado com o intuito de fornecer um maior embasamento para programação dentro do kernel, utilizando de forma eficiente os recursos nele disponibilizados. Além disso, os fontes do kernel constituem em si uma documentação bastante importante sobre como programar, organizar e integrar um novo conjunto de funcionalidades dentro dele.

A linguagem de programação adotada para a implementação do protótipo foi a ANSI C, já que o kernel é também escrito nessa linguagem. Desse modo, todas as listagens de códigos fonte encontradas neste trabalho estão em ANSI C.

Neste capítulo o protótipo é detalhadamente descrito, colocando-se em evidência as principais dificuldades encontradas em sua implementação, as soluções estudadas e adotadas. Na seção 6.7 a interação do protótipo com o kernel é descrita, indicando a localização de seus componentes e sua relação com o kernel.

6.1 Suporte criptográfico

Para o desempenho de suas atividades o SOFFIC precisa de três tipos de algoritmos criptográficos disponibilizados no kernel:

a) algoritmos de hash;

b) algoritmos de chaves simétricas; e

c) algoritmos de chaves assimétricas (somente verificação de assinatura).

[†] Disponível em http://www.kernel.org

Os algoritmos de hash e de chaves simétricas foram facilmente obtidos através da aplicação do *The International Kernel Patch*[†], um patch para o kernel que implementa vários desses algoritmos, entre eles:

a) algoritmos de hash: MD5 e SHA1; e

b) algoritmos de chaves simétricas: Blowfish, IDEA, 3DES, DES, Rijndael, Twofish, MARS e Serpent.

Esses algoritmos puderam ser facilmente empregados, eliminando dessa forma maiores preocupações com a sua implementação.

Ao contrário dos algoritmos de hash e de chaves simétricas, os algoritmos de chaves assimétricas precisam executar operações aritméticas que operam com grandes números, acima de 512 bits, o que não é suportado nativamente pelo kernel. Fez-se então necessária a inserção, no kernel, de um biblioteca de funções para tratar e operar com tais números, bem como para fornecer os algoritmos de chaves assimétricas. Infelizmente, o único patch encontrado para suprir tais necessidades foi o MIRACL[‡], e este, por sua vez, possui uma séria limitação quanto ao seu uso fora do ambiente acadêmico, impondo uma grande barreira a um dos principais objetivos deste trabalho: a livre distribuição do código fonte do SOFFIC, sob a General Public Licence (GPL).

Sendo assim, ainda procurando evitar os esforços com a implementação desses algoritmos, uma alternativa foi levantada: integrar ao kernel uma biblioteca criptográfica já disponível para uso no espaço de usuário, distribuídas sem restrições de uso. Entre as bibliotecas analisadas estão: a Beecrypt, a GPGMe e a OpenSSL. A primeira não implementa o suporte desejado para a verificação de assinaturas RSA, embora tenha sido anunciado no seu site. Já a segunda era específica para o uso em aplicações, não podendo ser facilmente portada para o kernel. A OpenSSL é a mais completa delas, oferecendo todas as funções de geração de chaves assimétricas RSA, bem como de assinatura e verificação. Mas, talvez devido a esse motivo, a sua estrutura é demasiadamente grande para ser inserida na íntegra no kernel e ao mesmo tempo complexa para ser desmembrada.

Outras bibliotecas ainda foram cogitadas, mas nenhuma reunia todas as características desejadas de forma de distribuição e simplicidade para inserção no kernel. Como resultado disso, uma vez que somente a verificação de assinatura RSA é necessária, optou-se pelo desenvolvimento de uma biblioteca própria para esse fim. Entre as vantagens dessa decisão estão a simplicidade desejada para inserção no kernel e eliminação de qualquer limitação de distribuição.

A simplicidade foi obtida graças ao fato da biblioteca de grandes números ter sido criada especificamente para a verificação de assinaturas e da simplicidade do algoritmo para verificação de uma assinatura RSA, o qual deve calcular o resultado da seguinte expressão:

$$z = x^b \bmod n,$$

[†] Disponível em http://www.kernel.org
[‡] Disponível em http://indigo.ie/~mscott/

onde x é a mensagem cifrada, b e n compõe a chave pública e finalmente z é a mensagem decifrada. A mensagem em questão é o hash da massa de dados assinada, que então pode ser utilizado para verificar a sua integridade. A autenticidade é obtida pela correta decifragem do hash com a chave pública K_{pb} do administrador do SOFFIC.

A essa biblioteca deu-se o nome de KBigNum (Kernel Big Numbers). Ela é capaz de realizar operações com números acima 512 bits, necessários para operação com chaves RSA, cujos tamanhos mais comuns variam entre 1024 e 2048 bits. De fato, o limite para o tamanho do número é a memória disponível para armazená-lo. Entre as operações implementadas estão: soma, subtração, deslocamento de bits, divisão, resto de divisão e comparação.

Com isso, os três tipos de algoritmos criptográficos estão disponíveis no kernel para a plena implementação do SOFFIC.

6.2 Hash List

Cada entrada da HL, conforme pode ser visto na figura abaixo, é composta de apenas quatro elementos:

a) device e inode: esse par de informações identifica de forma única um arquivo no sistema e é por isso utilizado como índice para pesquisas;

b) hash: é gerado apenas a partir do conteúdo do arquivo, mas também poderiam ser inseridos no cálculo diversos atributos do mesmo com o objetivo de também verificá-los (e.g. propriedade e permissões de acesso); e

c) flags: é um mapa de bits utilizado para indicar determinadas características de uma entrada, como, por exemplo, o algoritmo de hash utilizado.

device	inode	hash	flags
773	21238	593b7961645f383520ea	MD5
...	...	...	...
773	38374	7b5637c5070160ab2990	SHA1
774	2763	2d77d2b2aa9811bbb20b	MD5, CORE

FIGURA 6.1 – Estrutura da Hash List

A pesquisa na HL é realizada em duas etapas: uma pesquisa seqüencial por um índice de devices e outra binária em um índice de inodes (FIGURA 6.2). A pesquisa binária é extremamente eficiente, sendo que o maior número de comparações para localizar uma determinada entrada, ou para verificar a inexistência da entrada procurada, pode ser calculado a partir da seguinte fórmula:

$$c = \log_2 n + 1,$$

onde n é o número de entradas e c o número de comparações necessárias [FOL 92]. Assim, sendo n = 1024, seriam necessárias, no pior caso, apenas 11 comparações para encontrar uma determinada entrada. Ainda, a medida que n é dobrado c aumenta em apenas 1.

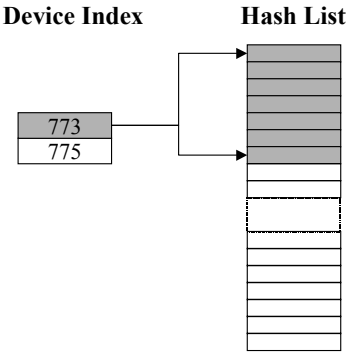


FIGURA 6.2 – Pesquisa na Hash List

A criação de um índice de devices (Device Index), foi devida a uma preocupação com a grande possibilidade de determinados devices terem apenas algumas poucas entradas na HL, fazendo com que a pesquisa fosse realizada sobre o número total de entradas da mesma. O índice de devices tende a ser bastante pequeno, possuindo cerca de 3 a 5 entradas, daí a pesquisa seqüencial ter sido empregada. Cada uma das entradas aponta para os endereços iniciais e finais da HL onde se encontram os inodes de cada device (FIGURA 6.2), a partir daí a pesquisa binária é utilizada.

Para esse método de pesquisa funcionar, a HL deve estar previamente ordenada pelo valor do device seguido pelo valor do inode, o que é realizado no momento da sua geração pelo sofficadm (seção 6.9).

6.3 Trusted Files List

A TFL possui uma estrutura bastante simples, já que o SOFFIC só a utiliza para saber quais arquivos podem ser acessados sem verificação de integridade. É composta somente pelo par device/inode, novamente utilizado como índice, e pelo campo flags (FIGURA 6.3).

device	inode	flags
773	10232	L,E
...	...	...
774	5393	L
774	21923	L

FIGURA 6.3 – Estrutura da Trusted Files List

Diferentemente da HL, a TFL pode incluir pares de device/inode não só de arquivos mas também de diretórios, suportando as características descritas na seção 5.3.2. Os flags da TFL indicam apenas o tipo de acesso liberado para uma determinada entrada: Leitura e/ou Execução

Somente é utilizada a pesquisa binária na TFL sem o uso de qualquer outro índice auxiliar, uma vez que ela não foi projetada para conter um número grande de entradas. Tal como a HL, a TFL, já organizada pelo device/inode de cada entrada, é gerada pelo sofficadm.

6.4 Recently Verified Files Cache

Os únicos dados da HL a serem armazenados na RVFC são o device e o inode de arquivos recentemente verificados e válidos. Arquivos compreendidos pela TFL não são colocados na RVFC.

A RVFC foi implementada com base em quatro estruturas, são elas:

a) vetor de entradas (slots): vetor com o número de entradas que a RVFC deverá conter; atualmente esse número é definido no momento da compilação do kernel;

b) lista de slots livres (free_slots): evita uma procura seqüencial no vetor de entradas por um slot livre;

c) lista de slots ocupados (used_slots): utilizada para a remoção de slots antigos; e

d) hash table: utilizada para busca por hash dos slots, evitando uma busca seqüencial na used_slots; cada entrada da hash table é apenas um ponteiro para um slot.

Para um maior entendimento da implementação da RVFC, se fazem necessários, neste ponto, alguns comentários sobre a estrutura das listas nela utilizadas. O kernel já fornece um conjunto de funções, macros e tipos de dados para suporte à implementação de listas duplamente encadeadas, as quais são comumente utilizadas na implementação de outros tipos de cache no kernel. Não há diferenciação entre cabeçalhos e nodos, todos têm a mesma estrutura constituída apenas de dois membros: *next* e *prev*. As funções mais relevantes utilizadas na RVFC são descritas na tabela a seguir, as demais, bem como as macros citadas, podem ser encontradas no arquivo include/linux/list.h, na raiz dos fontes do kernel.

TABELA 6.1 – Principais funções de manipulação de listas na RVFC

Nome	**Função**
list_add(node, head)	Insere um nodo imediatamente *após* o cabeçalho especificado;
list_add_tail(node, head)	Insere um nodo imediatamente *anterior* ao cabeçalho especificado
list_del(node)	Remove o nodo especificado da lista a qual está encadeado
list_entry(node, type, member)	Retorna o endereço da estrutura tipo *type*, da qual o nodo especificado faz parte sob o nome *member*

A última função é talvez a mais importante para a RVFC, ela permite a obtenção do endereço dos slots a partir do endereço de um nodo, sem que, para isso, este último tenha em sua estrutura o endereço do referido slot.

Cada slot da RVFC tem a estrutura apresentada na figura abaixo. Os membros device e inode já foram comentados, os dois últimos, *state* e *list*, são nodos a serem utilizados respectivamente para controle de estado do slot (livre ou ocupado) e para o gerenciamento de colisões na hash table.

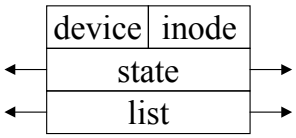


FIGURA 6.4 – Estrutura de uma entrada da RVFC (slot)

Logo na inicialização, todos os slots têm seu membro *state* encadeado à lista free_slots (FIGURA 6.5). A ordem em que os slots são encadeados à esta lista é irrelevante, uma vez que qualquer um deles pode ser ocupado sem nenhuma conseqüência para quaisquer outros procedimentos de manipulação dos slots.

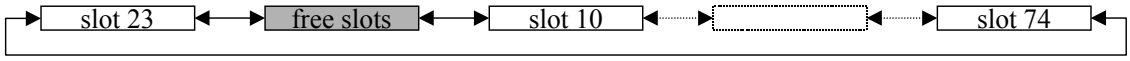


FIGURA 6.5 – Lista free_slots da RVFC

6.4.1 Inserção de uma nova entrada

Havendo a requisição para inserção de uma nova entrada na RVFC, a lista free_slots é verificada. Caso seja encontrado um slot disponível, o seu membro *state* é removido da lista free_slots (FIGURA 6.6).

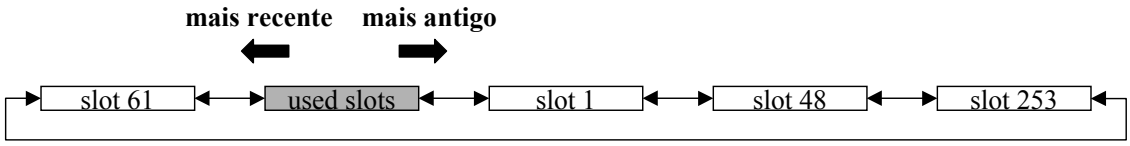


FIGURA 6.6 – Lista used_slots da RVFC

Estando a free_slots vazia, um slot em uso deve ser desocupado para a inserção da nova entrada na RVFC. A ordem de encadeamento dos nodos na used_slots é fundamental para a escolha do slot a ser desocupado, uma vez que a estratégia implementada para gerenciamento da RVFC é a LRU, descrita anteriormente na seção 5.4.8. Desse modo, os primeiros slots a serem desocupados devem ser sempre os mais antigos, assim sendo, o nodo imediatamente seguinte ao cabeçalho da lista é removido e encadeado à free_slots.

O slot livre obtido recebe os valores device e inode da entrada, em seguida é calculado o hash para a inserção do slot na hash table. O referido hash não é um hash criptográfico, mas apenas serve para indicar o índice da hash table sob o qual o slot será mantido. O hash tem um tamanho de 16 bits, ou seja, é capaz de endereçar todas as 65536 entradas da hash table, e é calculado somente sobre o valor do inode, cujo

tamanho é 32 bits. A seguir a função de cálculo desse hash é apresentada em código fonte.

```
_rvfc_hash hash = 0;

for (i=0; i<4; i++) hash += (_rvfc_hash) ino >> i;
```

O tipo _rvfc_hash é apenas um inteiro de 16 bits não assinalado. Toda a função está contida no laço *for* que soma os 16 bits menos significativos do inode deslocado *i* posições para a direita. Essa função foi obtida através de experimentos onde foram também testadas diversas outras funções simples de hash, entre as quais foi escolhida devido ao baixo número de colisões, cerca de 2 em cada 100 entradas. Apesar disso, ela não pode ser considerada uma função ideal e muito menos definitiva para o SOFFIC, uma vez que os ambientes onde a mesma foi testada, incluindo os ambientes de teste da seção 7.1, não apresentavam muitas repetições de inodes, ou seja, um mesmo número de inode utilizado em devices diferentes.

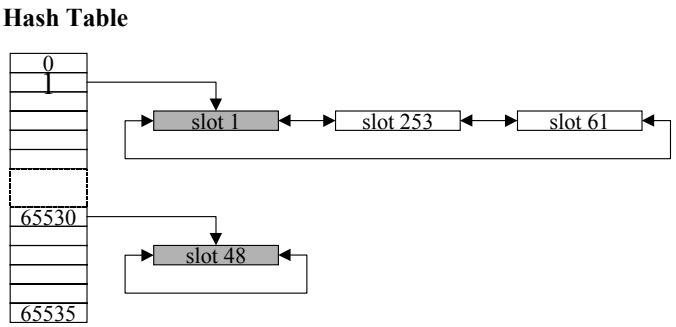


FIGURA 6.7 – Hash Table e colisões na RVFC

Calculado o hash, a entrada da hash table por ele indicada recebe o endereço do slot, isso pode ser observado na figura acima para os slots 1 e 48 (marcados em cinza). Nos casos em que a entrada da hash table já estiver apontando para um slot há uma colisão, e então os membros *list* dos slots são utilizados. O nodo *list* do primeiro slot, apontado pela hash table, é tratado então como cabeçalho de uma lista encadeada, onde então o novo slot será encadeado através do seu membro *list*.

Incluído o slot na hash table, o mesmo é encadeado ao final da lista used_slots através do seu membro *state*. Com isso a inclusão de uma nova entrada na RVFC é concluída.

6.4.2 Pesquisa

O primeiro passo para a pesquisa é localizar a entrada da hash table na qual está, ou deveria estar, inserido o slot contendo o par device/inode procurado. Isso é feito rapidamente com o cálculo do hash sobre o valor do inode. No momento seguinte, o conteúdo da entrada da hash table indicada pelo hash calculado é verificado; a ausência de um endereço de um slot indica imediatamente a inexistência do par device/inode procurados. Se, ao contrário, um endereço de um slot for encontrado, os valores de seus membros device e inode são comparados aos valores procurados. Não sendo satisfeita a

procura e existindo outros slots encadeados através dos membros *list* ao primeiro slot localizado, é realizada uma pesquisa seqüencial através dessa nova lista. Essa última situação pode ser visualizada no slot 1 da figura anterior.

Em qualquer ponto do processo, sendo encontrado o par device/inode procurado, a pesquisa retorna imediatamente indicando sucesso e o slot é movido para o fim da lista used_slots. Como pode ser constatado, a pesquisa na RVFC é extremamente eficiente devido à hash table e ao baixo número de colisões obtidas, fazendo com que o número de pesquisas seqüências realizadas seja bastante pequeno.

6.4.3 Casos especiais de remoção de entradas

Além do caso em que uma entrada antiga da RVFC é removida para dar lugar a uma nova, existem dois outros, ditos especiais:

a) invalidação da entrada de um arquivo alterado em disco; e

b) invalidação de toda a cache devido a uma escrita direta em um dispositivo (raw IO).

Esses dois casos foram implementados visando evitar o ataque descrito na seção 5.4.8, onde um atacante poderia alterar o conteúdo de um arquivo para o qual existe uma entrada na RVFC. Em um novo acesso, caso não houvesse o controle aqui discutido, o arquivo seria dado como íntegro.

No caso *b* a invalidação de toda a cache se faz necessária, pois um acesso direto ao dispositivo não permite ao SOFFIC determinar, com segurança, quais arquivos foram efetivamente modificados. Além disso, o acesso direto à dispositivos não é um evento comum no funcionamento do sistema

6.5 Desvios das chamadas de sistema (hooks)

Apenas quatro chamadas de sistema foram modificadas para desviar o seu fluxo de execução para o SOFFIC. Os nomes das chamadas modificadas, bem como os arquivos onde estão localizadas e as suas funções, estão descritos na tabela a seguir:

TABELA 6.2 – Desvios das chamadas de sistema

Arquivo*	Chamada	Função
fs/exec.c	open_exec()	Abertura de arquivos para execução
fs/namei.c	open_namei()	Abertura de arquivos para leitura
fs/read_write.c	sys_write()	Escrita em arquivo ou em dispositivo
fs/open.c	do_truncate()	Trunca o arquivo sem passar pela sys_write()

* o caminho é dado em relação a raiz dos fontes do kernel.

As chamadas open_exec() e open_namei() são as responsáveis por disparar o processo de verificação do SOFFIC. As duas últimas apenas são utilizadas para a invalidação de entradas na RVFC. A sys_write intercepta qualquer operação de escrita, seja em arquivo ou dispositivo (raw IO), e em ambos os casos RVFC é adequadamente atualizada segundo o procedimento descrito na seção anterior. A chamada do_truncate() executa a única operação que modifica o conteúdo de um arquivo sem o uso da sys_write, desse modo, foi necessário o desvio de fluxo dessa chamada para também invalidar entradas na RVFC.

Em todos os desvios são apenas inseridos os códigos estritamente necessários para fazer a chamada de uma das funções disponibilizadas ao kernel através da sofficapi, além de eventualmente alocar e/ou liberar algumas estruturas de dados. As funções disponibilizadas pela sofficapi são descritas na tabela a seguir:

TABELA 6.3 – Funções disponibilizadas na sofficapi

Sofficapi	**Chamada de sistema onde é utilizada**	**Função**
soffic_verify_read()	open_namei()	Verificar a integridade de arquivos
soffic_verify_exec()	open_exec()	Verificar a integridade de arquivos
soffic_rvfc_remove()	sys_write(), do_truncate()	Invalidar uma determinada entrada da RVFC
soffic_rvfc_flush()	sys_write()	Invalidar todas as entradas da RVFC

Somente as duas primeiras funções da sofficapi retornam um valor indicando a necessidade de bloqueio ou não do acesso. As funções relacionadas a RVFC nunca retornam valor algum. No anexo 6 pode ser encontrado, como exemplo prático, o código fonte completo da chamada open_exec(). Nele, a alteração (desvio) relacionada ao soffic encontra-se destacada em fundo cinza.

6.6 Flags e estatísticas do SOFFIC

A fim de facilitar o controle do SOFFIC durante o seu desenvolvimento, foram implementados vários flags que especificam a habilitação ou não de determinados recursos do mesmo. Cada um desses flags é composto de apenas 1 bit pertencente a uma variável de 32 bits cujo nome é soffic_flags. Na tabela abaixo são listados os flags implementados e suas funções.

TABELA 6.4 – Flags do SOFFIC

Flag	**Valor**	**Função**
SOF_BF_*DBGMESS*	2^1	Mensagens de debug
SOF_BF_*EXECVRFY*	2^2	Verificar arquivos na execução
SOF_BF_*READVRFY*	2^3	Verificar arquivos na leitura
SOF_BF_*EXECDENY*	2^4	Impedir execução de arquivos inválidos
SOF_BF_*READDENY*	2^5	Impedir leitura de arquivos inválidos

SOF_BF_*RVFC*	2^6	Habilitar a RVFC
SOF_BF_*SOFFIC*	2^{15}	Habilitar o SOFFIC

No momento da compilação do kernel, o usuário pode definir o valor padrão dos flags SOF_BF_EXECDENY e SOF_BF_READDENY. Os demais flags são por padrão acionados, com exceção do SOF_BF_DBGMESS, não sendo oferecida ao usuário a opção de alterá-los no momento da compilação.

Já no momento do boot do sistema, pode-se alterar o valor de qualquer um dos flags. Por exemplo, desejando-se acionar as mensagens de debug e o bloqueio da execução de arquivos inválidos, os seguintes parâmetros devem ser passados para o boot loader (normalmente o LILO):

```
soffic=dbgmess:1,execdeny:1
```

Outra opção para alterar os flags, após o sistema ter sido inicializado, é fornecida através do sistema de arquivos Proc. Na prática, o Proc não existe sob forma de uma partição em disco, ele é apenas constituído na memória RAM e foi criado como uma interface para facilitar a troca de informações entre o kernel e os processos do espaço de usuário. Os arquivos nele existentes podem ser lidos e/ou escritos conforme as operações implementadas para cada um deles. [BEC 98]

Fazendo uso dos recursos oferecidos pelo Proc, foi criado um diretório para o soffic (/proc/fs/soffic/), onde são inseridos arquivos para cada um dos flags anteriormente citados. O nome dos arquivos é dado pelo nome dos flags sem o prefixo SOF_. Dessa forma, eles podem ser facilmente acessados e alterados pelo usuário, utilizando, por exemplo, as seguintes linhas de comando:

```
# echo 0 > /proc/fs/soffic/bf_dbgmess

# echo 1 > /proc/fs/soffic/bf_execdeny

# cat /proc/fs/soffic/bf_execdeny
```

O primeiro comando desliga o flag dbgmess, o segundo ativa o execdeny e a terceira apenas mostra o valor desse último: 1.

Outro recurso implementado no SOFFIC foi a contabilização de determinadas operações e resultados, de forma a facilitar o acompanhamento de seu funcionamento. Os valores contabilizados são disponibilizados ao usuário através de um outro arquivo inserido no sistema de arquivos Proc: o stat. Esse arquivo somente pode ser lido, e para isso o programa *cat* pode ser novamente utilizado, conforme exemplificado abaixo:

```
# cat /proc/fs/soffic/stats
execvrfy   394   (verificações na execução)
readvrfy  2622   (verificações na leitura)
notfound   303   (não encontrados na HL e TFL)

RVFC:
found     2552   (arquivos encontrados na RVFC)
used       151   (entradas em uso)
released     0   (entradas substituídas)
```

```
killed       0   (entradas invalidadas)
flush        2   (todas entradas invalidadas)

HL:
found      161   (arquivos encontrados na HL)
valid      153   (arquivos válidos)
invalid      8   (arquivos inválidos)

TFL:
found        0   (arquivos compreendidos pela TFL)
```

Os comentários, colocados entre parênteses em cada linha, não fazem parte da saída do comando, mas foram acrescentados aqui para uma melhor clareza do papel de cada elemento.

Os flags foram criados, num primeiro momento, apenas para facilitar os testes de implementação e desempenho do SOFFIC, dessa forma, os seus aspectos de segurança não foram considerados até o momento. Como conseqüência disso, os mesmos deverão ser retirados em uma instalação de um ambiente prático, pelo menos até que sejam adequadamente analisados.

6.7 Interação com o kernel

A operação do SOFFIC é independente do tipo de sistema de arquivos utilizado (e.g. ext2, FAT, etc.). Isso é possível graças ao Virtual File System (VFS) do Linux, que abstrai completamente o acesso aos sistemas de arquivos suportados pelo kernel [BEC 98]. Sem o VFS não seria possível, por exemplo, utilizar devices e inodes a partir do sistema de arquivos FAT, uma vez que esse tipo de sistema de arquivos não os utiliza. Dessa forma, o VFS facilita consideravelmente a implementação do SOFFIC, eliminando a necessidade de estudo de cada sistema de arquivos a ser suportado pelo mesmo e potenciais limitações de uso.

A interação do SOFFIC com o VFS é apresentada na figura a seguir. Nela, fica claro o papel do VFS, podendo-se notar que, em nenhum momento, ocorre interação alguma do SOFFIC diretamente com os sistemas de arquivos ou dispositivos. As interações entre os processos e o VFS não foram alteradas, pois, conforme colocado na seção 5.5.3, primeiro o sistema deve aplicar os seus próprios mecanismos de controle de acesso, para só depois o SOFFIC desempenhar suas tarefas. O SOFFIC é então disparado pelo próprio VFS através dos hooks implementados.

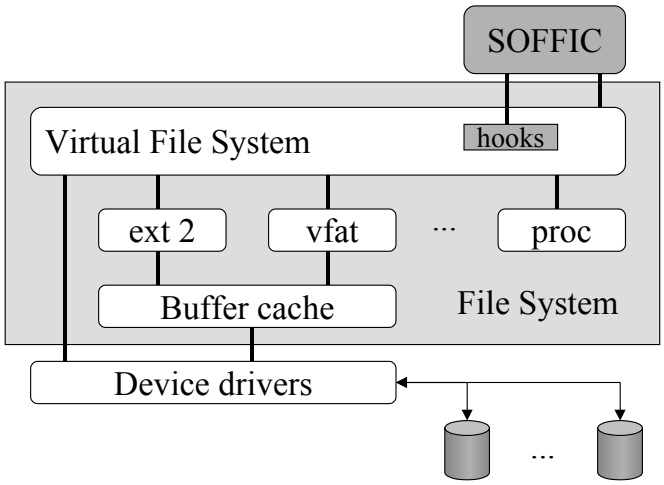


FIGURA 6.8 – Inserção do SOFFIC no kernel
Fonte: baseada em [BEC 98], p. 149.

Internamente o SOFFIC é composto por vários elementos (FIGURA 6.9). O digest, kbignum e rsacheck implementam toda estrutura para uso dos algoritmos criptográficos colocados na seção 6.1. No misc são implementadas funções diversas para manipulação de arquivos e alocação de memória. A HL e a TFL estão inseridas em lists e a RVFC é implementada separadamente.

Apenas três elementos do SOFFIC têm contato direto com o kernel: o proc, sofficapi e core. O primeiro apenas fornece ao VFS as chamadas para leitura e escrita dos arquivos do SOFFIC no Proc. A sofficapi, já comentada anteriormente, implementa uma interface para que o VFS tenha acesso às funções internas do SOFFIC. O core, ao contrário da sofficapi, tem uma relação bem mais tênue com o kernel, interagindo com o mesmo somente no momento da carga do sistema, quando o SOFFIC é inicializado.

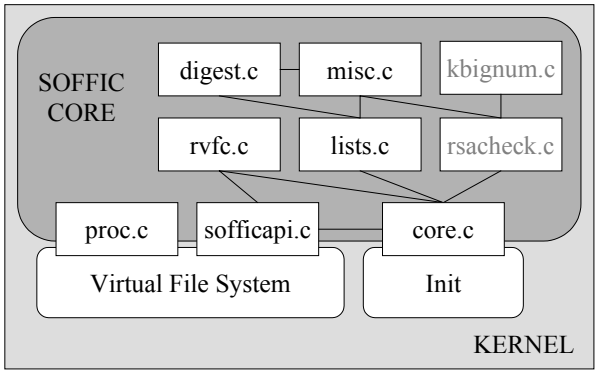


FIGURA 6.9 – Componentes do SOFFIC e interação com o kernel

Os elementos kbignum e rsacheck são apresentados em letras cinzas na figura anterior pois, apesar de terem sido implementados, ainda não foram inseridos no SOFFIC.

6.8 Instalação do SOFFIC no kernel

Dos fontes modificados do kernel, é criado um patch que pode ser utilizado por qualquer outro usuário para a inserção do SOFFIC num novo kernel. Assim como acontece com outros patches para o kernel do Linux, não é recomendada a aplicação desse patch num kernel cuja versão é diferente da qual onde ele foi gerado, portanto, o SOFFIC, no presente momento, só deve ser utilizado no kernel versão 2.4.3. Ainda, antes da aplicação do patch do SOFFIC, deve ser aplicado o *The International Kernel Patch* citado na seção 6.1.

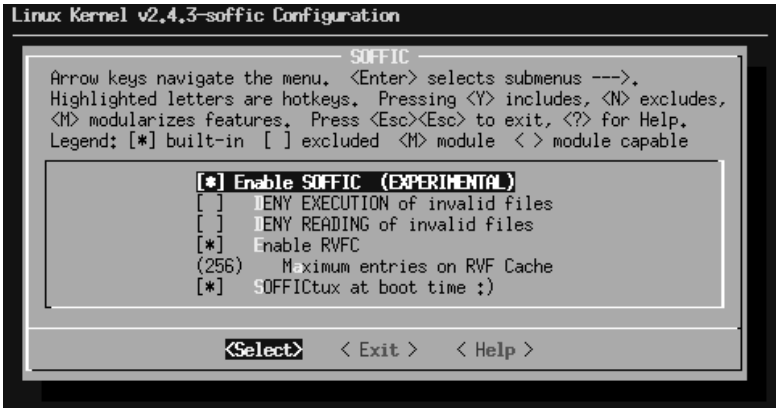


FIGURA 6.10 – Opções do SOFFIC para compilação do kernel

Na configuração do kernel, o SOFFIC deve ser selecionado para que seja compilado e para que as suas opções de compilação sejam disponibilizadas ao usuário (FIGURA 6.10). Atualmente, através dessas opções, o usuário pode:

a) determinar o valor padrão dos flags EXECDENY e READDENY;

b) compilar ou não a RVFC; e

c) determinar o tamanho da RVFC.

Essas opções foram criadas, assim como os flags, para facilitar o desenvolvimento do SOFFIC, não sendo portanto opções definitivas de compilação.

6.9 Sofficadm

Foram implementadas no sofficadm, a geração da HL e da TFL. Os algoritmos de hash (SHA1 e MD5) e de chave assimétrica foram obtidos com o uso da biblioteca OpenSSL. A mesma biblioteca fornece também toda a estrutura para geração de chaves assimétricas e de assinaturas RSA, a qual deverá ser utilizada no sofficadm. Até o presente estágio de desenvolvimento do sofficadm as operações com chaves assimétricas RSA são realizadas através do programa openssl.

A HL é gerada a partir de um arquivo de entrada contendo a lista de diretórios ou arquivos a serem nela incluídos. O conteúdo do arquivo de entrada obedece a seguinte estrutura:

```
Nome-do-grupo
(
<entradas>
)
{
<entradas>
}
```

O arquivo pode conter vários grupos, cada grupo é declarado com o seu nome seguido de subgrupos de entradas, delimitados pelo uso de parênteses ou chaves. O uso dos primeiros indicam que as pesquisas em diretórios não são recursivas, ou seja, somente os arquivos de cada diretório especificado são incluídos na HL, ignorando-se os subdiretórios. Já as chaves são utilizadas quando a recursividade é desejada. Cada grupo pode conter tantos subgrupos quantos forem necessários. Segue abaixo um exemplo prático de um grupo do arquivo de entrada da HL:

```
basic system binaries
{
/bin
/sbin
/usr/bin
/usr/sbin
}
```

As opções para geração da HL disponíveis no sofficadm são as seguintes:

```
sofficadm v0.3
usage: sofficadm -H [options]

 -g Generate
 -d <md5/sha1>    digest algorithm (md5)
 -i <file>        input file (must be specified)
 -o <file>        output file (/etc/soffic/soffic.hl)

 -l List contents
 -i <file>    input file (/etc/soffic/soffic.hl)
 -o <file>    output file (stdout)
```

A TFL é gerada de forma praticamente idêntica a HL, apenas com a diferença de que a recursividade indica a inclusão de cada arquivo existente em um diretório especificado e ao invés do diretório em si. Não há portanto, a inclusão de subdiretórios nas pesquisas recursivas para composição da TFL. Um exemplo de um grupo da TFL é apresentado a seguir.

```
Temp dirs
(
/tmp
/usr/tmp
)
```

As opções do sofficadm para a TFL são também idênticas, apenas com a ausência do parâmetro –d.

7 Análise de Desempenho

A análise realizada teve como principal objetivo fornecer uma indicação aproximada do impacto do SOFFIC no desempenho do sistema e do benefício da implementação da RVFC, indicando os casos de maior e de menor impacto. Uma análise inicial realizada sobre a arquitetura do SOFFIC, indicou como caso de maior impacto a primeira leitura de um arquivo para o qual existe uma entrada na Hash List, exigindo assim o cálculo do seu hash. O caso de menor impacto ocorreria quando um arquivo para o qual já existe uma entrada na RVFC fosse acessado.

Assim, buscou-se com a realização destes testes a confirmação dessas hipóteses e a possível indicação de pontos a serem aperfeiçoados no modelo. Nesta seção são descritos e comentados os ambientes de teste, a metodologia e os resultados obtidos. Nos anexos 2 a 5, estão contidas as tabelas dos resultados dos testes em cada um dos ambientes descritos a seguir.

7.1 Ambientes de avaliação

Foram utilizados dois ambientes idênticos em relação ao sistema operacional, mas distintos em termos de características do equipamento. O sistema operacional utilizado foi a distribuição RedHat 7.1 do GNU/Linux, que logo após a instalação padrão *servidor* teve os seus pacotes atualizados e seu kernel versão 2.4.2 substituído pela versão 2.4.3 com o SOFFIC já instalado.

Os equipamentos utilizados variam em dois pontos principais: processador e cache de disco. Uma tabela descritiva deles é apresentada a seguir, denominados aqui simplesmente como *A* e *B*.

TABELA 7.1 – Descrição dos ambientes de teste de desempenho

Característica	Ambiente A	Ambiente B
Processador	Intel Pentium III	Intel Pentium II Celeron
Freqüência	1 GHz	400 MHz
RAM	64 MB	64 MB
Disco – Capacidade	20 GB	6 GB
Disco – Cache	2 MB	-
Itens na Hash List	≅ 11000	≅ 11000
Tamanho da RVFC	256	256

7.2 Metodologia

Como elemento básico de todo o teste foi criado um programa, o sofficbench, que realiza um determinado número de leituras de um mesmo arquivo e mede o tempo de cada uma delas. O sofficbench, apesar de ser bastante simples, preenche de forma satisfatória as necessidades dos testes realizados. A seguir o principal trecho do sofficbench é apresentado:

```
#define BUFFSIZE 1024

gettimeofday(&begin, NULL);
fp = fopen(file, "r");

if (fp) {
      while (!feof(fp) && (fread(buffer,1,BUFFSIZE,fp)>0));
      fclose(fp);
      gettimeofday(&end, NULL);
}

time = elapsed_time(&begin, &end);
```

Este trecho é repetido tantas vezes quantas forem requisitadas para um determinado arquivo. A função gettimeofday(), utilizada para obter a hora atual da máquina em microsegundos, pode influenciar nas medidas em que o espaço tempo é muito pequeno, já que ela mesma consome também uma fatia de tempo para ser executada. Procurando-se verificar a intensidade dessa influência, foi criado um laço de calibragem no sofficbench. Este laço consiste em apenas chamar a função gettimeofday() repetidas vezes e calcular o tempo transcorrido entre cada uma delas. O principal trecho dessa rotina é listado a seguir.

```
for (i=0; i<5001; i++)
      gettimeofday(&calib[i], NULL);

tot = 0;
for (i=1; i<5001; i++)
      tot += elapsed_time(&calib[i-1], &calib[i]);

avg = tot / 5000;
```

O primeiro laço *for* chama a função gettimeofday() 5001 vezes e guarda os resultados em um vetor. O laço seguinte soma as diferenças de tempo entre cada uma das 5001 chamadas e em seguida a média é dada pela soma das diferenças dividida por 5000 (número de diferenças, não de medidas).

Criado o sofficbench, o mesmo foi aplicado em cinco configurações de ambiente. Cada uma dessas configurações é caracterizada na tabela a seguir:

TABELA 7.2 – Configurações de ambiente utilizadas

Nome	SOFFIC	HL	RVFC
SEM-SOFFIC	Desabilitado	-	-
MD5	Habilitado	MD5	Desabilitada
MD5-C	Habilitado	MD5	Habilitada
SHA1	Habilitado	SHA1	Desabilitada
SHA1-C	Habilitado	SHA1	Habilitada

A HL foi composta sempre pelos mesmos arquivos, em todas as configurações. Entre esses arquivos, seis foram criados especificamente para serem utilizados nos

testes. A única característica relevante desses seis arquivos para os testes realizados é o tamanho de cada um deles (TABELA 7.3). A TFL não foi utilizada nos testes, uma vez que ela não influência nos casos de pior e melhor desempenho do SOFFIC.

TABELA 7.3 – Arquivos utilizados nos testes

Nome do arquivo	Tamanho em KB
file-256k	256
file-1024k	1024
file-2048k	2048
file-5124k	5124
file-10248k	10248
file-20496k	20496

Foram realizados dois tipos de testes. O primeiro, Leituras Seqüenciais (LS), consistiu em executar o sofficbench para cada arquivo, realizando 20 leituras seguidas. Este procedimento foi repetido 20 vezes, resultando em 400 leituras de cada arquivo, para cada configuração de ambiente. O sistema foi sempre reinicializado a cada mudança de configuração. Nas configurações com a RVFC habilitada, a mesma era reinicializada a cada execução do sofficbench. Este teste objetivou o levantamento do impacto do SOFFIC no sistema e da eficácia da RVFC.

Aplicado o primeiro teste, foi constatada a influência da cache do próprio sistema operacional, e também da cache de disco do ambiente A, nas 19 últimas medidas de tempo de cada arquivo, fazendo com que o valor da primeira medida de cada execução do sofficbench fosse bastante superior aos valores das medidas consecutivas (essa influência é discutida na seção 7.3). Dessa forma, um segundo teste, Primeiras Leituras (PL), foi elaborado com o objetivo de obter o real impacto do SOFFIC, e da RVFC, numa primeira leitura de um arquivo. Assim, o teste consistiu na execução do sofficbench para cada um dos arquivos, realizando somente uma leitura. Este procedimento foi executado 10 vezes para cada configuração de ambiente, sendo que o sistema era reinicializado a cada execução do sofficbench.

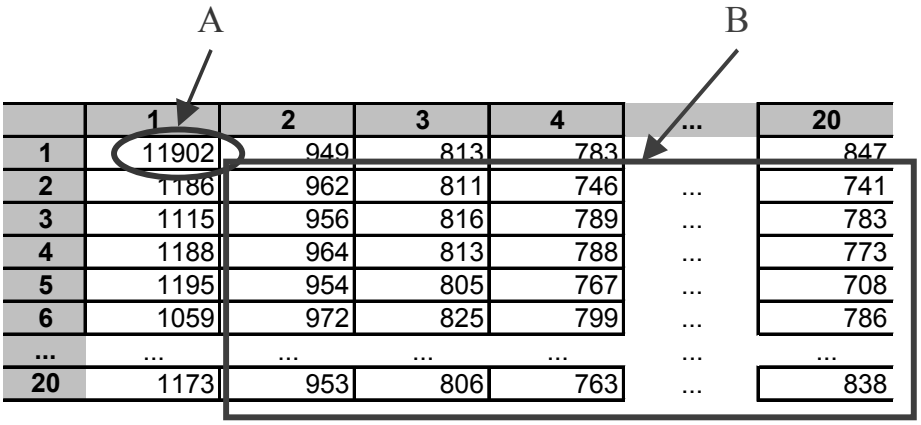
Em todas etapas de todos os testes, o sistema operacional era sempre inicializado em modo *single user*, já que após os primeiros experimentos foi constatada uma influência aleatória de processos em execução no sistema.

7.3 Resultados

O primeiro passo na análise de desempenho foi verificar o tempo médio de execução da chamada gettimeofday() através do laço de calibragem do sofficbench. Os resultados obtidos no ambiente A e B foram respectivamente 0 e 1 microsegundo, o que indica uma influência insignificante da chamada gettimeofday() no processo, sendo então ignorada.

Dos dados resultantes da aplicação do primeiro teste, Leituras Seqüenciais, foram constituídas tabelas para cada arquivo em cada configuração de ambiente. Na

figura a seguir, é apresentado um trecho da tabela gerada para o arquivo file-256k na configuração SEM-SOFFIC do ambiente A. Cada linha da tabela representa uma execução do sofficbench e cada coluna uma leitura do arquivo. Conforme colocado na seção anterior, todos os valores são dados em microsegundos.



	1	2	3	4	...	20
1	11902	949	813	783		847
2	1186	962	811	746	...	741
3	1115	956	816	789	...	783
4	1188	964	813	788	...	773
5	1195	954	805	767	...	708
6	1059	972	825	799	...	786
...	...	...	...	...	...	...
20	1173	953	806	763	...	838

FIGURA 7.1 – Exemplo de tabela resultante dos primeiros testes

Na figura é possível identificar a influência da cache do sistema operacional, e também da cache de disco, nas leituras realizadas pelo sofficbench. O círculo *A* destaca o valor da primeira leitura da primeira execução do sofficbench, o qual é evidentemente muito superior ao valor das leituras seguintes. Ainda, é possível notar a mesma influência nos valores das primeiras leituras das linhas 2 a 20, embora não na mesma magnitude. Essas mesmas influências foram também identificadas em todas as demais tabelas geradas pelo teste.

A partir dessas primeiras tabelas foram calculadas as médias que constituem as colunas ‘LS 1’ e ‘LS 2..20’ das tabelas resumo (TABELA 7.4). Tanto para o cálculo de uma quanto de outra, os valores da primeira execução do sofficbench foram descartados devido a discrepância do valor da primeira leitura. As colunas ‘LS 1’ são compostas pelas médias das primeiras leituras (coluna 1 na FIGURA 7.1), as quais foram calculadas separadamente devido a influência da cache comentada no parágrafo anterior.

TABELA 7.4 – Exemplo de tabela resumo

SEM-SOFFIC			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	8467	1172	787
1024k	42429	3716	3338
2048k	63131	6968	6585
5124k	159389	16727	16324
10248k	293401	32894	32545
20496k	592664	65474	65425

A composição das colunas ‘LS 2..20’ deu-se pelo cálculo das médias das colunas 2 a 20 (indicadas pelo retângulo *B* na FIGURA 7.1), seguido pelo cálculo da média dessas últimas. Da aplicação do segundo teste, Primeiras Leituras, obteve-se a média de tempo de dez leituras dos arquivos que por fim constituem a coluna PL das tabelas resumo.

As tabelas resumo do ambiente A encontram-se no anexo 2, e do ambiente B no anexo 4. Foram também criadas tabelas resumo com os valores normalizados em relação ao ambiente SEM-SOFFIC (anexos 3 e 5), as quais são efetivamente utilizadas na análise dos resultados.

7.3.1 Análise dos resultados

Para uma primeira análise foram elaborados gráficos procurando-se destacar as condições de menor e maior impacto do SOFFIC. Seguindo a hipótese colocada no primeiro parágrafo da seção 7, segundo a qual o maior impacto deve ocorrer no primeiro acesso a um arquivo listado na Hash List e o menor quando forem realizadas leituras subseqüentes de um arquivo para o qual existe uma entrada na RVFC, foram traçados gráficos para os ambientes A (FIGURA 7.2) e B (FIGURA 7.3), com base nos valores das colunas PL (maior impacto) e ‘LS 2..20’ (menor impacto) das tabelas resumo normalizadas. Ao passo em que todas as configurações de ambiente tiveram as suas seqüências de dados PL inseridas no gráfico, somente as configurações com a RVFC habilitada tiveram suas seqüências ‘LS 2..20’ incluídas.

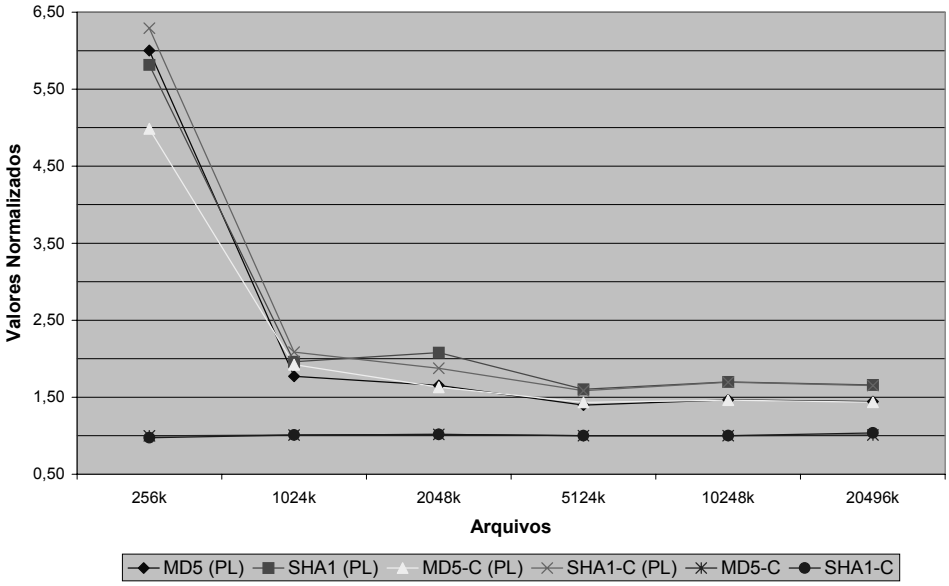


FIGURA 7.2 – Gráfico comparativo PL e ‘LS 2..20’ do Ambiente A

Na figura acima pode-se notar um impacto bastante grande nas primeiras leituras do menor arquivo do teste, isso ocorre devido ao fato dos intervalos de tempo para pesquisa e leitura desse arquivo serem extremamente pequenos no ambiente A, fazendo com que o tempo gasto no cálculo do hash venha a ser destacado. Nos arquivos seguintes o impacto tende a diminuir bruscamente, ficando aproximadamente entre 1,50

e abaixo de 2,00. Já na figura seguinte, as linhas das primeiras leituras são bem mais regulares entre si. Isso se deve a ausência da cache de disco, fazendo com que o tempo de pesquisa no mesmo seja mais elevado do que no ambiente A.

No gráfico do ambiente B nota-se também a clara influência da velocidade do algoritmo de hash utilizado nas primeiras leituras, onde o algoritmo MD5 tende a ser sempre mais rápido que o SHA1. Esse fato não era inesperado já que é documentado em diversas obras da área, entre elas [SCH 96] e [MEN 96]. E também de forma similar ao ocorrido no gráfico do ambiente A, existe um alteração brusca na linha do 'MD5-C (PL)' para o arquivo de 256KB.

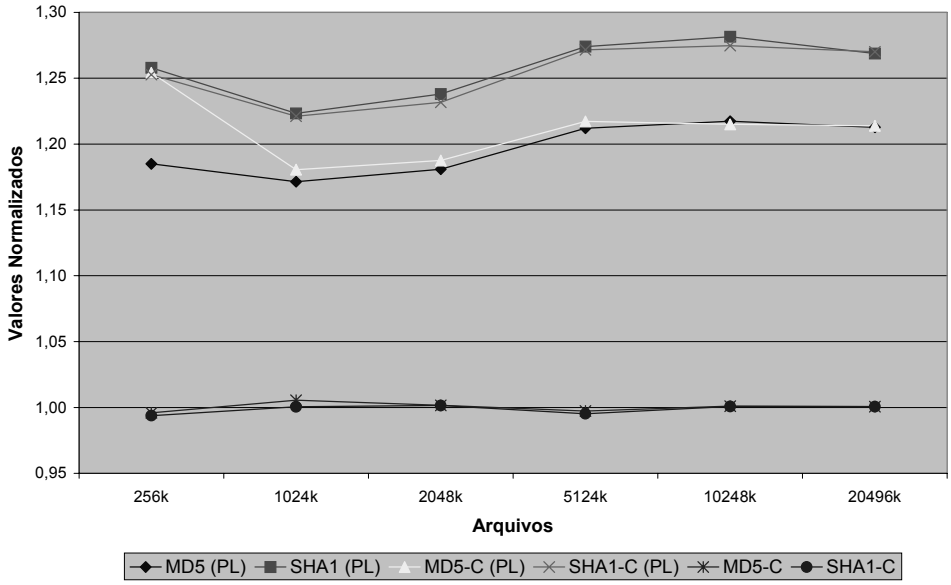


FIGURA 7.3 – Gráfico comparativo PL e 'LS 2..20' do Ambiente B

O impacto do SOFFIC na velocidade das primeiras leituras variou bastante entre os ambientes A e B, e também em outros ambientes experimentais não citados aqui simplesmente por não contribuírem significativamente nesses primeiros estudos de desempenho. Essas variações, em sua maioria bastante bruscas, são devidas a relação entre a velocidade do processador (quanto mais rápido menor o tempo de cálculo dos hashes) e do disco (quanto mais rápido menor o tempo de localização e leitura de um arquivo). Essa relação não foi detalhadamente estudada e desenvolvida neste trabalho uma vez que o objetivo era apenas realizar medidas simples de desempenho para, num primeiro momento, avaliar o modelo implementado. Ainda, o estudo de tais aspectos exigiriam esforços dignos de um novo trabalho neles enfocado, já que uma grande variedade de ambientes deveria ser incluída nos experimentos.

Muito embora pareça o contrário, as medidas PL não refletem o pior caso conforme estudado inicialmente. O pior caso é evidenciado pelas medidas 'LS 1' das tabelas resumo em que a RVFC está habilitada. Como os testes implicavam em uma nova inicialização da RVFC a cada execução do sofficbench, um arquivo que saía da RVFC continuava na cache do sistema. Assim, o tempo de uma nova pesquisa e leitura eram minimizados enquanto o tempo de cálculo do hash permanecia o mesmo, resultando em impactos maiores do que os indicados pela PL, variando na ordem de 8,00 a 12,00 em valores normalizados, bem acima da variação da PL: entre 1,00 e 2,00.

Desse resultado, conclui-se algo bastante interessante para o dimensionamento da RVFC: ela deve ser grande o suficiente para garantir que, em condições normais, nenhum arquivo ainda constante na cache do sistema, ou do disco, seja dela retirado. Caso isso acontecesse freqüentemente, o SOFFIC causaria uma demora de 8 a 12 vezes maior do que o normal para o acesso e leitura de um arquivo já contido em uma dessas caches do ambiente.

Já a hipótese do melhor caso foi visivelmente confirmada pelos gráficos, indicando a eficácia da RVFC, uma vez que o impacto do SOFFIC é praticamente desprezível em se tratando de um arquivo já verificado. Nos testes sem a RVFC os resultados foram idênticos aos das colunas 'LS 1' (maior impacto), conforme pode ser verificado nos anexos 2 a 5.

Apesar do impacto do SOFFIC indicado pelos valores normalizados, pode-se constatar que, de fato, os valores de tempo trabalhados na análise são bastante pequenos, e o real impacto nas operações normais de um sistema vão variar bastante de acordo com o papel desempenhado por ele. Nesses casos, diversos outros fatores, como número de usuários e tipos de serviços oferecidos, poderão ser determinantes para a adoção ou não do SOFFIC.

8 Conclusões

O desenvolvimento do presente trabalho colocou em evidência, mais uma vez, a deficiência dos atuais mecanismos de segurança implementados nos sistemas operacionais, que tratam a integridade de arquivos apenas de forma marginal. É fato, comprovado pela proliferação de diversos vírus e rootkits, por exemplo, que tal abordagem é extremamente ineficiente.

Várias propostas já foram apresentadas para suprir tal deficiência, apesar disso, a grande maioria delas segue um mesmo modelo, onde assumem papel secundário na segurança do sistema, auxiliando apenas na detecção de violações já ocorridas. O que, em muitas situações, pode levar à tratamentos tardios dos danos causados por ataques contra o sistema e decorrente comprometimento de sistemas vizinhos.

Foi apresentado neste trabalho, um modelo de mecanismo de controle de integridade que busca suprir as deficiências apontadas na segurança dos sistemas operacionais, sem, no entanto, apresentar as mesmas deficiências das propostas já existentes. No modelo proposto, uma das principais intenções foi realizar a contenção dos danos causados pela violação da integridade dos arquivos do sistema, detectando-a imediatamente antes que um arquivo inválido seja utilizado por um usuário. Também esteve constantemente presente na elaboração desse modelo a preocupação com a segurança do próprio mecanismo de controle de integridade, bem como com o seu desempenho.

O protótipo apresentado comprovou a viabilidade da implementação do modelo proposto no sistema operacional GNU/Linux, reafirmando, ao longo dos testes realizados, o valor da adição de um mecanismo de controle de integridade aos mecanismos de segurança já existentes nos sistemas operacionais. A eficiência do modelo foi em parte atestada, já que o mesmo ainda se encontra na fase de implementação e testes. Mesmo assim, os resultados obtidos até o momento são bastante promissores.

A preocupação com o desempenho mostrou-se fortemente fundamentada através das análises realizadas sobre o protótipo, indicando a certa inviabilidade de aplicação do modelo caso esse aspecto não fosse tratado. Nessa tarefa, as estratégias definidas para reduzir o impacto do modelo sobre o desempenho normal do sistema, foram empregadas de forma bastante satisfatória, embora ainda existam pontos a serem aperfeiçoados.

Um dos principais resultados deste trabalho foi a clara definição dos pontos sensíveis de qualquer mecanismo ou ferramenta de controle de integridade de arquivos, mais particularmente daqueles cujo funcionamento é integrado ao kernel do sistema operacional, e também a proposição de possíveis soluções a serem empregadas em cada um deles. Outro resultando bastante importante é a disponibilização de um protótipo útil para a realização de estudos e para aplicações práticas, em regime de experiência, para usuários finais.

Não obstante a importância dos resultados citados, talvez a contribuição mais relevante dos estudos realizados seja a exposição de um campo de pesquisa, que apesar de não ser novo, ainda foi pouco explorado: sistemas operacionais seguros. Esse campo

compreende não só a implementação de mecanismos de segurança já existentes, mas também o estudo e desenvolvimento de mecanismos novos e mais eficientes.

8.1 Trabalhos Futuros

Como trabalho futuro, mas já em andamento, o SOFFIC deverá ser implementado na íntegra, refletindo na sua implementação os requisitos de funcionamento definidos no modelo, permitindo a sua plena aplicação na prática. Incluí-se aqui, também, a substituição da função de hash da RVFC por outra mais adequada à ambientes com várias partições, mas que mantenha ou diminua a taxa de colisões obtida com a função atualmente implementada.

A seguir são colocados, sob forma de tópicos, outros trabalhos a serem realizados a partir do protótipo implementado:

a) verificar pontos a serem modificados no protótipo para operar em sistemas com múltiplos processadores (SMP);

b) ampliar os recursos para descrição da HL a ser criada, por exemplo: incluir diretivas para exclusão de arquivos e permitir a indicação de atributos dos arquivos a serem incluídos no hash criptográfico; e

c) implementar e avaliar o uso do algoritmo MD4 para geração de hashes criptográficos: apesar de possíveis fraquezas apontadas nesse algoritmo, ele, de fato, ainda não teve o seu funcionamento comprometido.

As indicações de trabalhos futuros aqui realizadas não tiveram a pretensão de esgotar todas as possibilidades, ao contrário, espera-se que diversas outras sejam obtidas a partir do estudo do trabalho apresentado, de forma a contribuir para a evolução do modelo e da implementação do SOFFIC.

Anexo 1 Listagem dos 64 passos do MD5

Etapa 1

FF(*a*,*b*,*c*,*d*,M_0,7,0xd76aa478)
FF(*d*,*a*,*b*,*c*,M_1,12,0xe8c7b756)
FF(*c*,*d*,*a*,*b*,M_2,17,0x242070db)
FF(*b*,*c*,*d*,*a*,M_3,22,0xc1bdceee)
FF(*a*,*b*,*c*,*d*,M_4,7,0xf57c0faf)
FF(*d*,*a*,*b*,*c*,M_5,12,0x4787c62a)
FF(*c*,*d*,*a*,*b*,M_6,17,0xa8304613)
FF(*b*,*c*,*d*,*a*,M_7,22,0xfd469501)
FF(*a*,*b*,*c*,*d*,M_8,7,0x698098d8)
FF(*d*,*a*,*b*,*c*,M_9,12,0x8b44f7af)
FF(*c*,*d*,*a*,*b*,M_{10},17,0xffff5bb1)
FF(*b*,*c*,*d*,*a*,M_{11},22,0x895cd7be)
FF(*a*,*b*,*c*,*d*,M_{12},7,0x6b901122)
FF(*d*,*a*,*b*,*c*,M_{13},12,0xfd987193)
FF(*c*,*d*,*a*,*b*,M_{14},17,0xa6794383)
FF(*b*,*c*,*d*,*a*,M_{15},22,0x49b40821)

Etapa 2

GG(*a*,*b*,*c*,*d*,M_1,5,0xf61e2562)
GG(*d*,*a*,*b*,*c*,M_6,9,0xc040b340)
GG(*c*,*d*,*a*,*b*,M_{11},14,0x265e5a51)
GG(*b*,*c*,*d*,*a*,M_0,20,0xe9b6c7aa)
GG(*a*,*b*,*c*,*d*,M_5,5,0xd62f105d)
GG(*d*,*a*,*b*,*c*,M_{10},9,0x02441453)
GG(*c*,*d*,*a*,*b*,M_{15},14,0xd8a1e681)
GG(*b*,*c*,*d*,*a*,M_4,20,0xe7d3fbc8)
GG(*a*,*b*,*c*,*d*,M_9,5,0x21e1cde6)
GG(*d*,*a*,*b*,*c*,M_{14},9,0xc33707d6)
GG(*c*,*d*,*a*,*b*,M_3,14,0xf4d50d87)
GG(*b*,*c*,*d*,*a*,M_8,20,0x455a14ed)
GG(*a*,*b*,*c*,*d*,M_{13},5,0xa9e3e905)
GG(*d*,*a*,*b*,*c*,M_2,9,0xfcefa3f8)
GG(*c*,*d*,*a*,*b*,M_7,14,0x676f02d9)
GG(*b*,*c*,*d*,*a*,M_{12},20,0x8d2a4c8a)

Etapa 3

HH(*a*,*b*,*c*,*d*,M_5,4,0xfffa3942)
HH(*d*,*a*,*b*,*c*,M_8,11,0x8771f681)
HH(*c*,*d*,*a*,*b*,M_{11},16,0x6d9d6122)
HH(*b*,*c*,*d*,*a*,M_{14},23,0xfde5380c)
HH(*a*,*b*,*c*,*d*,M_1,4,0xa4beea44)
HH(*d*,*a*,*b*,*c*,M_4,11,0x4bdecfa9)
HH(*c*,*d*,*a*,*b*,M_7,16,0xf6bb4b60)
HH(*b*,*c*,*d*,*a*,M_{10},23,0xbebfbc70)
HH(*a*,*b*,*c*,*d*,M_{13},4,0x289b7ec6)
HH(*d*,*a*,*b*,*c*,M_0,11,0xeaa127fa)
HH(*c*,*d*,*a*,*b*,M_3,16,0xd4ef3085)
HH(*b*,*c*,*d*,*a*,M_6,23,0x04881d05)
HH(*a*,*b*,*c*,*d*,M_9,4,0xd9d4d039)
HH(*d*,*a*,*b*,*c*,M_{12},11,0xe6db99e5)
HH(*c*,*d*,*a*,*b*,M_{15},16,0x1fa27cf8)
HH(*b*,*c*,*d*,*a*,M_2,23,0xc4ac5665)

Etapa 4

II(*a*,*b*,*c*,*d*,M_0,6,0xf4292244)
II(*d*,*a*,*b*,*c*,M_7,10,0x432aff97)
II(*c*,*d*,*a*,*b*,M_{14},15,0xab9423a7)
II(*b*,*c*,*d*,*a*,M_5,21,0xfc93a039)
II(*a*,*b*,*c*,*d*,M_{12},6,0x655b59c3)
II(*d*,*a*,*b*,*c*,M_3,10,0x8f0ccc92)
II(*c*,*d*,*a*,*b*,M_{10},15,0xffeff47d)
II(*b*,*c*,*d*,*a*,M_1,21,0x85845dd1)
II(*a*,*b*,*c*,*d*,M_8,6,0x6fa87e4f)
II(*d*,*a*,*b*,*c*,M_{15},10,0xfe2ce6e0)
II(*c*,*d*,*a*,*b*,M_6,15,0xa3014314)
II(*b*,*c*,*d*,*a*,M_{13},21,0x4e0811a1)
II(*a*,*b*,*c*,*d*,M_4,6,0xf7537e82)
II(*d*,*a*,*b*,*c*,M_{11},10,0xbd3af235)
II(*c*,*d*,*a*,*b*,M_2,15,0x2ad7d2bb)
II(*b*,*c*,*d*,*a*,M_9,21,0xeb86d391)

Anexo 2 Tabelas dos testes de desempenho
Ambiente A

Valores em Microsegundos

SEM-SOFFIC			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	8467	1172	787
1024k	42429	3716	3338
2048k	63131	6968	6585
5124k	159389	16727	16324
10248k	293401	32894	32545
20496k	592664	65474	65425

MD5			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	50828	8005	7508
1024k	75186	31974	31909
2048k	104277	63753	63669
5124k	222654	158250	158092
10248k	430410	316513	316111
20496k	855917	631939	632115

SHA1			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	49246	9721	9363
1024k	83122	38511	38432
2048k	131182	76761	76685
5124k	255446	190666	190571
10248k	498530	381220	381182
20496k	982763	761821	759816

MD5-C			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	42219	8014	786
1024k	81517	32017	3368
2048k	102849	63759	6695
5124k	228378	158083	16336
10248k	429043	315984	32571
20496k	850189	632178	65989

SHA1-C			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	53275	9689	765
1024k	88442	38423	3364
2048k	118461	76766	6689
5124k	252444	190584	16326
10248k	496873	380837	32548
20496k	978647	729590	67793

Anexo 3 Tabelas normalizadas dos testes de desempenho Ambiente A

MD5			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	6,00	6,83	9,54
1024k	1,77	8,60	9,56
2048k	1,65	9,15	9,67
5124k	1,40	9,46	9,68
10248k	1,47	9,62	9,71
20496k	1,44	9,65	9,66

SHA1			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	5,82	8,29	11,89
1024k	1,96	10,36	11,51
2048k	2,08	11,02	11,65
5124k	1,60	11,40	11,67
10248k	1,70	11,59	11,71
20496k	1,66	11,64	11,61

MD5-C			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	4,99	6,84	1,00
1024k	1,92	8,62	1,01
2048k	1,63	9,15	1,02
5124k	1,43	9,45	1,00
10248k	1,46	9,61	1,00
20496k	1,43	9,66	1,01

SHA1-C			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	6,29	8,27	0,97
1024k	2,08	10,34	1,01
2048k	1,88	11,02	1,02
5124k	1,58	11,39	1,00
10248k	1,69	11,58	1,00
20496k	1,65	11,14	1,04

Anexo 4 Tabelas dos testes de desempenho Ambiente B

Valores em Microsegundos

SEM-SOFFIC			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	86510	1740	1610
1024k	317737	6181	6012
2048k	637600	12070	11887
5124k	1477260	29870	29713
10248k	2926337	59036	58924
20496k	5880976	117452	117655

MD5			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	102513	18109	18051
1024k	372221	71497	71482
2048k	753001	142676	142706
5124k	1790483	356570	356650
10248k	3562019	713397	713549
20496k	7132390	1425280	1425019

SHA1			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	108818	22312	22249
1024k	388661	88178	88114
2048k	789341	175849	175885
5124k	1882138	880207	879354
10248k	3750125	880207	879354
20496k	7461895	1758077	1757096

MD5-C			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	108496	18111	1604
1024k	375110	71481	6045
2048k	757201	143033	11906
5124k	1798134	356745	29635
10248k	3555416	713499	58984
20496k	7139046	1427790	117722

SHA1-C			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	108366	22306	1600
1024k	387994	88121	6014
2048k	785323	175995	11905
5124k	1878434	439717	29573
10248k	3730195	879893	58971
20496k	7469507	1758763	117727

Anexo 5 Tabelas normalizadas dos testes de desempenho Ambiente B

MD5			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	1,18	10,41	11,21
1024k	1,17	11,57	11,89
2048k	1,18	11,82	12,01
5124k	1,21	11,94	12,00
10248k	1,22	12,08	12,11
20496k	1,21	12,14	12,11

SHA1			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	1,26	12,82	13,82
1024k	1,22	14,27	14,66
2048k	1,24	14,57	14,80
5124k	1,27	29,47	29,59
10248k	1,28	14,91	14,92
20496k	1,27	14,97	14,93

MD5-C			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	1,25	10,41	1,00
1024k	1,18	11,57	1,01
2048k	1,19	11,85	1,00
5124k	1,22	11,94	1,00
10248k	1,21	12,09	1,00
20496k	1,21	12,16	1,00

SHA1-C			
Arquivo	**PL**	**LS 1**	**LS 2..20**
256k	1,25	12,82	0,99
1024k	1,22	14,26	1,00
2048k	1,23	14,58	1,00
5124k	1,27	14,72	1,00
10248k	1,27	14,90	1,00
20496k	1,27	14,97	1,00

Anexo 6 Código fonte da chamada de sistema open_exec()

```
struct file *open_exec(const char *name)
{
  struct nameidata nd;
  struct inode *inode;
  struct file *file;
  int err = 0;

  if(path_init(name,LOOKUP_FOLLOW|LOOKUP_POSITIVE, &nd))
    err = path_walk(name, &nd);

  file = ERR_PTR(err);
  if (!err) {
    inode = nd.dentry->d_inode;
    file = ERR_PTR(-EACCES);

    if (!IS_NOEXEC(inode) && S_ISREG(inode->i_mode)){
      int err = permission(inode, MAY_EXEC);

      file = ERR_PTR(err);
      if (!err) {
        file=dentry_open(nd.dentry,nd.mnt, O_RDONLY);

        if (!IS_ERR(file)) {
          err = deny_write_access(file);
          if (err) {
            fput(file);
            file = ERR_PTR(err);
          }
#ifdef CONFIG_SOFFIC
          else if ((!soffic_domain) &&
                   (soffic_verify_exec(name,nd.dentry))) {

            allow_write_access(file);
            fput(file);

            file = ERR_PTR(-EACCES);
            /* Permission denied */
          }
#endif
        }
out:
        return file;
      }
    }
    path_release(&nd);
  }
  goto out;
}
```

Bibliografia

[AIV 2001] AIVAZIAN, Tigran. **Linux Kernel 2.4 Internals**. 2001. Disponível em: <http://www.moses.uklinux.net/patches/lki.sgml>. Acesso em: 20 de nov. 2001.

[ANO 2000] ANONYMOUS. **Maximum Linux Security**. Indianapolis: Sams Publishing, 2000.

[BEC 98] BECK, M. et al. **Linux Kernel Internals**. Harlow: Addison-Wesley, 1998.

[BEL 2001] BELLETTINI, Carlo et al. Role Based Access Control Models. **Information Security Technical Report**, [S.l.], v.6, n.2, p. 21-29, June 2001.

[BOE 93] BOER, B.; BOSSELAERS, A. Collisions for the Compression Function of MD5. In: EUROCRYPT, 1993. **Proceedings...** [S.l.:s.n.], 1993. p. 194-203.

[BRE 2001] BREMER, Steve. **LIDS FAQ**. 2001. Disponível em: <http://www.lids.org/lids-faq/LIDS-FAQ.html>. Acesso em: 20 de nov. 2001.

[CLA 2000] CLARK, Paul C. Policy-Enhanced Linux. In: NATIONAL INFORMATION SYSTEMS SECURITY CONFERENCE, NISSC, 23., 2000, Baltimore, Maryland. **Proceedings...** [S.l.:s.n.], 2000.

[DOB 96] DOBBERTIN, Hans; BOSSELAERS, Antoon; PRENEEL, Bart. RIPEMD-160: A Strengthened Version of RIPEMD. In: INTERNATIONAL WORKSHOP ON FAST SOFTWARE ENCRYPTION, 3., 1996. **Fast Software Encryption:** proceedings. Berlin: Springer-Verlag, 1996. (Lecture Notes in Computer Science, v. 1039).

[FOL 92] FOLK, Michael J.; ZOELLIC, Bill. **File Structures.** 2nd ed. Reading: Addison-Wesley, 1992.

[GAR 96] GARFINKEL, Simson; SPAFFORD, Gene. **Pratical Unix and Internet Security**. Sebastopol: O'Reilly & Associates, 1996.

[GRS 2001] GRSECURITY. **Getrewted General Security Guide**. 2001. Disponível em: <http://www.grsecurity.net/general.txt>. Acesso em: 20 de nov. 2001.

[GRS 2001a] GRSECURITY. **Grsecurity Features**. 2001. Disponível em: <http://www.grsecurity.net/features.htm>. Acesso em: 20 de nov. 2001.

[KIM 95] KIM, Gene H.; SPAFFORD, Eugene H. **The Design and Implementation of Tripwire:** A File System Integrity Checker. [S.l.]: Purdue University, 1993. (Technical Report CSD-TR-93-071).

[KRA 97] KRAWCZYK, H. et al. **HMAC: Keyed-Hashing for Message Authentication**: RFC 2104. [S.l.], 1997.

[LET 2001] LEHTI, Rami; VIROLAINEN, Pablo. **AIDE – Advanced Intrusion Detection Environment**. 2001. Disponível em: <http://www.cs.tut.fi/~rammer/aide.html>. Acesso em: 20 nov. 2001.

[LIN 2000] LIN, A.; BROWN, R. The application of security policy to role-based access control and the common data security architecture. **Computer Communications**, [S.l.], v.23, n.17, p. 1584-1593, Nov. 2000.

[LOS 2001] LOSCOCCO, Peter A.; SMALLEY, Stephen D. Integrating Flexible Support for Security Policies into the Linux Operating System. In: USENIX ANNUAL TECHNICAL CONFERENCE, 2001, Boston, Massachusetts. **Proceedings...**[S.l.:s.n.], 2001.

[LOS 2001a] LOSCOCCO, Peter A.; SMALLEY, Stephen D. Meeting Critical Security Objectives with Security-Enhanced Linux. In: OTTAWA LINUX SYMPOSIUM, 2001, Ottawa. **Proceedings...** [S.l.:s.n.], 2001.

[MED 2001] MEDUSA DS9. **History and Concepts**. 2001. Disponível em: <http://medusa.fornax.sk/English/project.shtml>. Acesso em: 20 nov. 2001.

[MEN 96] MENEZES, Alfred J. et al. **Handbook of Applied Cryptography**. Florida: CRC Press, 1996.

[NAT 87] NATIONAL COMPUTER SECURITY CENTER. **A guide to understanding discretionary access control in trusted systems**. Fort George G. Meade, Maryland, USA, 1987.

[NIS 95] NIST. **Secure Hash Standard.** FIPS PUB 180-1. Washington D.C., 1995.

[OLI 2000] OLIVEIRA, Rômulo S. et al. **Sistemas Operacionais**. Porto Alegre: Instituto de informática da UFRGS: Sagra Luzzatto, 2000.

[OTT 2001] OTT, Amon. Rule Set Based Access Control (RSBAC). In: SNOW UNIX EVENT, 2001, Waardenburg. **Proceedings...** [S.l.:s.n.], 2001.

[RIV 92] RIVEST, R. **The MD4 Message-Digest Algorithm**: RFC 1320. [S.l.], 1992.

[RIV 92a] RIVEST, R. **The MD5 Message Digest Algorithm**: RFC 1321. [S.l.], 1992.

[RSA 2000] RSA LABORATORIES. **Frequently Asked Questions about Today's Cryptography, vesion 4.1**. 2000. Disponível em <http://www.rsasecurity.com/rsalabs/faq/files/rsalabs_faq41.pdf>. Acesso em: 20 nov. 2001.

[RUS 91] RUSSEL, Deborah; GANGEMI, G. T. **Computer Security Basics**. Sebastopol: O'Reilly & Associates, 1991.

[SCH 2000] SCHNEIER, B; **Secrets and Lies:** Digital Security in a Networked Wolrd. New York: John Wiley & Sons, 2000.

[SCH 96] SCHNEIER, B; **Applied Cryptography.** 2nd ed. New York: John Wiley & Sons, 1996.

[SER 2001] SERAFIM, Vinícius S.; WEBER, Raul F. Um verificador seguro de integridade de arquivos. In: SIMPÓSIO SEGURANÇA EM INFORMÁTICA, 3., 2001, São José dos Campos, SP. **Anais...** São José dos Campos: CTA/ITA/IEC, 2001. p.169-174.

[SIL 98] SILBERSCHATZ, Abraham; GALVIN, Peter B. **Operating System Concepts**. Reading: Addison-Wesley, 1998.

[SKI 97] SKIENA, Steven S. **The Algorithm Design Manual**. New York: Springer-Verlag, 1997.

[TAN 96] TANNENBAUM, A. S. **Computer Networks.** 3rd ed. Englewood Cliffs: Prentice Hall, 1996.

[VXE 2001] VXE. **VXE Overview**. 2001. Disponível em: <http://www.intes.odessa.ua/vxe/Overview/overview.html>. Acesso em: 20 nov. 2001.

[WAT 2000] WATSON, Robert N. M. Introducing Supporting Infrastructure for Trusted Operating System Suport in FreeBSD. In: BSDCon, 2000. **Proceedings...** [S.l.:s.n.], 2000.

[WAT 2001] WATSON, Robert N. M. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In: USENIX ANNUAL TECHNICAL CONFERENCE, 2001, Boston, Massachusetts. **Proceedings...** [S.l.:s.n.], 2001.

[WIL 93] WILLIAMS, Ross N. **A Painless Guide to CRC Error Detection Algorithms**. 1993. Disponível em: <ftp://ftp.rocksoft.com/papers/crc_v3.txt>. Acesso em: 20 de out. 2001.

[WIL 94] WILLIAMS, Ross N. **An Introduction To Digest Algorithms**. 1994. Disponível em: <ftp://ftp.rocksoft.com/papers/digest10.ps>. Acesso em: 20 nov. 2001.

[WIL 94a] WILLIAMS, Ross N. **Data Integrity With Veracity**. 1994. Disponível em: <ftp://ftp.rocksoft.com/papers/vercty10.ps>. Acesso em: 20 nov. 2001.