Capítulo 3

Transformações de código para proteção de software

Davidson Rodrigo Boccardo, Raphael Carlos Santos Machado e Luiz Fernando Rust da Costa Carmo

Abstract

This chapter presents software transformation techniques for protection of intellectual property and security of sensitive data against 'Man-At-The-End' attacks. Security is normally associated with the confidentiality of the communication channel based on cryptography. Security based on software protection is associated with code obfuscation, watermarking and tamper-proofing techniques. Code obfuscation aims to make harder for a reverse engineer to understand the inner properties of the code. Watermarking is a defense against piracy, in which a mark is embedded inside the code for further tracking of legal authorities. Tamper-proofing aims to ensure that the program executes as expected affront of monitoring and/or changing. The goal of all these techniques is to maintain the security of the software code running at the end user, and its protection is extremely important to ensure that the running software is executing as its purpose.

Resumo

Este capítulo apresenta técnicas de transformação de código para proteção da propriedade intelectual e segurança dos dados contra ataques 'Man-At-The-End'. A segurança convencional é normalmente atrelada com à confidencialidade do canal de comunicação, e esta é baseada em métodos de criptografia. Segurança baseada em proteção de software está associada a técnicas de ofuscação, marca d'água e 'tamper-proofing'. As técnicas de ofuscação de código visam a dificultar o entendimento do código quando neste é aplicado engenharia reversa. As técnicas de marca d'água são defesas contra pirataria, em que uma marca é embarcada no código do programa para rastreabilidade a posteriori em uma ação legal. As técnicas de 'tamper-proofing' garantem que o programa execute como esperado diante de modificação e monitoração. O objetivo de todas estas técnicas é dar maior credibilidade ao software em execução no sistema final assegurando que o software em execução comporte-se de maneira legítima.

3.1. Introdução

Tradicionalmente, segurança de computadores está relacionada ao cenário em que um computador está sob ataque por um adversário ou por um código malicioso criado por este adversário. A pesquisa nesta área envolve o desenvolvimento de técnicas para prevenir que um adversário tenha acesso a um outro computador, ou que emita um alerta em uma situação de ataque. Estas técnicas visam a prevenir, detectar e/ou interromper um ato malicioso. Neste contexto temos *firewalls* que permitem restringir o acesso a um determinado computador; sistemas de detecção de intrusão, que alertam o usuário de um computador diante de um padrão de acesso à rede suspeito; e os varredores de código malicioso, que bloqueiam a execução de código caso apresente comportamento malicioso.

Em ataques *Man-At-The-End*, em que um adversário tem acesso físico ao software/hardware com poder de comprometê-lo, seja por inspeção ou por modificação, modelos tradicionais de segurança não são suficientes. Uma técnica comum para esconder informação dentro de um programa consiste na utilização de técnicas de criptografia. Nesta situação, um desenvolvedor esconderia o segredo de seu software — algoritmo, chave criptográfica ou número serial — usando algum algoritmo criptográfico. Contudo, neste cenário temos que proteger a chave de criptografia, pois caso desvendada o segredo seria revelado. Além disso, para que este código fosse executável, este precisaria ser decriptografado em algum instante, o que possibilitaria de um adversário extrair o código no momento que este estivesse em texto claro. Com acesso ao código, é só uma questão de tempo e motivação para que um adversário consiga extrair o segredo de um programa.

Avanços em análise de programas e tecnologias de engenharia de software têm levado ao aperfeiçoamento das ferramentas para análise de programas e desenvolvimento de software. Estes fornecem recursos para que os softwares sejam mais eficientes e livres (o quanto possível) de vulnerabilidades. No entanto, estes mesmos avanços incrementam o poder dos softwares para engenharia reversa com o objetivo de descobrir vulnerabilidades, realizar alterações indevidas, ou furtar propriedade intelectual. A engenharia reversa exige a reconstrução do código executável para alguma estrutura de alto nível. Por exemplo, para identificação de vulnerabilidades em um software, um adversário teria que primeiro identificá-las para depois atacá-las. Analogamente, para furtar um trecho de código na presença de um direito autoral ou marca d'água, um adversário teria que analisar o código para que fosse possível modificá-lo sem afetar a funcionalidade do programa.

Neste texto, exploramos técnicas de transformação de código para proteção de software com o intuito de dificultar ataques *Man-At-The-End*. Atualmente, existe pouco conhecimento, entre os desenvolvedores de software, sobre a importância do uso destas técnicas. As técnicas de proteção de software diferem das técnicas de segurança de computadores convencional, pois uma vez que um adversário tenha acesso ao programa, não existem limites do que o mesmo pode fazer. Uma técnica de proteção de software pode ser definida como um conjunto de procedimentos para dificultar um adversário de obter ganhos sobre o software, seja por motivos financeiros, sabotagem ou vingança. O conhecimento destas técnicas exerce maior impacto em cenários que podem afetar infra-estruturas nacionais caso atacadas, como *Smart Grids* do setor elétrico e redes de sensores sem fio da área militar. Um cenário de ataque em *Smart Grids* envolve perpetuar ataques na infra-estrutura de medição de forma a causar um *denial of service* na rede elétrica.

Na área militar um adversário poderia modificar os sensores para que estes enviassem informações distorcidas, por exemplo quanto ao movimento das tropas, para a central de comando.

Ataques em software normalmente envolvem duas etapas: análise e modificação. Na etapa de análise, um adversário busca compreensão do software, enquanto na etapa de modificação um adversário subverte o software de acordo com seu objetivo. Um ataque típico envolve fazer a engenharia reversa do software da concorrência para extrair conhecimento de algum módulo ou do software como um todo, para *a posteriori*, modificá-lo para revenda.

Existem diferentes abordagens de engenharia reversa e definir qual abordagem mais apropriada depende da aplicação alvo, da plataforma em que roda a aplicação, onde foi desenvolvida e qual tipo de informação deseja extrair. Existem duas metodologias de engenharia reversa: análise estática e análise dinâmica.

A análise estática de programas é uma técnica utilizada para coletar informações sobre o programa sem a necessidade de executá-lo. A análise envolve o uso de um *disassembler* que faz a conversão do código binário para código *assembly*, cujo formato é mais compreensível por um humano do que uma sequência de bits. A análise estática exige um melhor entendimento do programa (comparado com a análise dinâmica) pois o fluxo da execução é somente baseado no código do programa. Ferramentas de análise de fluxo de controle coletam informações para uma melhor compreensão sobre o fluxo de execução do programa. Como o fluxo pode ser dependente dos valores contidos em posições de memória, análise do fluxo de dados predizem o conjunto de valores que determinada memória pode vir a assumir durante a execução do programa. Outra ferramenta que se enquadra em análise estática são os decompiladores que tentam reverter o processo de compilação para produção de um código de alto nível. Contudo, na maioria das arquiteturas, a recuperação do código de alto nível não é realmente possível, pois existem elementos significativos que são removidos durante a compilação uma vez que o âmbito deste é otimização e não a legibilidade do código.

A análise dinâmica de programas é uma técnica em que a coleta de informações é feita em tempo de execução diante de uma determina entrada. Nesta análise, é possível observar como a entrada interage como o fluxo de execução e os dados do programa. A análise basicamente envolve o uso de um depurador (*debugger*) que permite um analista observar o programa durante sua execução. Um depurador possui tipicamente duas características básicas: habilidade de ajustar pontos de parada (*breakpoints*) no programa e capacidade de verificar o estado atual do programa (registradores, memória, conteúdo da pilha). A análise dinâmica também pode ser feita através de emulação de código. Nesta análise, o código é executado em um emulador que provê funcionalidades similares ao sistema original, assim como um arcabouço para auxiliar um analista no entendimento do código. Técnicas de *profiling* e *tracing* também são consideradas como dinâmicas. A primeira é utilizada para contagem do números de execuções, ou a quantidade de tempo despendida, de diferentes partes do código; a segunda em vez de realizar somente a contagem, registra também quais partes do código do programa foram executadas.

As técnicas de proteção de um software visam a dificultar a engenharia reversa, assegurando que o software executa como esperado e protegendo o software de pira-

taria [18, 19, 16, 17, 30]. Entre as técnicas de proteção de software, destacamos três: ofuscação, marca d'água e *tamper-proofing*. Técnicas de ofuscação de código dificultam engenharia reversa através de um aumento no grau de ininteligibilidade do código. Este aumento se caracteriza por modificações sintáticas que dificultam a compreensão do software mantendo, porém, as funcionalidades originais do programa (sua semântica). Técnicas de *tamper-proofing* asseguram que o software executa como esperado, mesmo na tentativa de modificação ou monitoração por um adversário. Técnicas de marca d'água agem como uma defesa contra pirataria de software cujo papel é determinar a origem de um software. Estas técnicas são baseadas na inclusão de informação no software em modo escondido que possa ser posteriormente reconhecido.

Técnicas de ofuscação de código adicionam confusão em um programa com a finalidade de dificultar o discernimento/extração/modificação de alguma propriedade desse programa, preservando, entretanto, sua funcionalidade. Desenvolvedores de software podem utilizar técnicas de ofuscação para esconder informações a respeito de um algoritmo por questões de propriedade intelectual — incluindo rotinas, chaves criptográficas e números seriais — assim como para incrementar a segurança do código contra a exploração maliciosa de vulnerabilidades, dada a dificuldade de se encontrar vulnerabilidades em um software de difícil discernimento. Por outro lado, ofuscação de software também tem sido utilizado por programadores maliciosos para difundir seus códigos maliciosos. Técnicas de ofuscação também podem ser utilizadas maliciosamente para distorcimento ou corrupção da marca d'água.

As transformações advindas de técnicas de ofuscação de código podem ocorrer tanto no código-fonte — que posteriormente é compilado de maneira a gerar um binário mais complexo — como diretamente no código binário. Alguns trabalhos propõem transformações a serem aplicadas sobre "código intermediário", visando, por exemplo, a proteção de *bytecodes* Java. Visto que sua compreensão é mais simples que a de código binário, e alvo de muitos decompiladores, sua proteção torna-se ainda mais crítica.

A ofuscação de um código busca dificultar a compreensão ou levar o adversário a desistir de analisar um determinado software. A engenharia reversa é o passo anterior à modificação deste software com o objetivo de obter alguma espécie de vantagem. Evitar violações à integridade do software e responder a violações — possivelmente impedindo a execução do software modificado ou redirecionando a execução para um código não funcional — são os objetivos das técnicas de *tamper-proofing*.

Estratégias comuns de *tamper-proofing* envolvem auto-verificação do *hash* de trechos de código ou a inspeção lógica do programa. Por exemplo, conhecendo previamente o *hash* de trechos de código ou o resultado que uma determinada variável deva possuir em um determinado ponto do programa, é possível verificar se os valores durante a execução do programa correspondem com os valores previamente conhecidos. Outra responsabilidade das técnicas de *tamper-proofing* diz respeito à ação a ser tomada diante de uma modificação. Estratégias comuns envolvem desviar o fluxo de execução para um código não funcional ou notificar o desenvolvedor que o código foi alterado. Outras estratégias menos amenas envolvem atacar o próprio sistema do adversário, por exemplo, apagando seus documentos pessoais. Para que as tarefas de verificação e resposta de uma técnica de *tamper-proofing* sejam menos "visíveis" para um adversário, estas costumam estar

dispersas no código do programa, assim como não executarem sequencialmente.

As técnicas de marca d'água de software são importantes para combate à pirataria de software. Embora marca d'água, por si só, não impeça um ato de violação de software, esta, no mínimo, desestimula tal ato e favorece a localização do violador do software. Marca d'água de software é caracterizada por uma informação de identificação codificada no código do programa que posteriormente possa ser reconhecida. Idealmente, uma marca d'água estaria tão atrelada ao software e ao cliente que qualquer região suficientemente grande do código possuiria tal informação o que inviabilizaria um processo de remoção.

Uma estratégia de marca d'água comum e de pouco impacto na execução do software envolve a inclusão do nome do autor em algum ponto do programa. Contudo, esta estratégia é de fácil localização e subversão um vez que um adversário pode modificar o nome do autor ou simplesmente substituir a porção de código do autor por um código não operante.

Dada a impossibilidade de esconder o código de um programa em cenários em que existe a possibilidade de violação física, métodos de ofuscação, *tamper-proofing* e marca d'água visam a tornar o processo de extrair conhecimento, piratear e modificar o código de um programa em um processo proibitivo quanto a custo, tempo e poder computacional. Outra possibilidade para esconder código, dados e execução do programa consiste em métodos baseados em hardwares especializados [20]. Contudo, estes apresentam desvantagens de custo de desenvolvimento, desempenho e perda de flexibilidade. Com estas desvantagens, é difícil um desenvolvedor convencer um cliente da importância destes métodos uma vez que tais métodos visam à proteção contra o próprio cliente (no caso de um cliente mal intencionado). Além disso, sistemas de software baseado em hardware tem sido alvos de ataque [48, 60].

Uma estratégia comum para combate a pirataria baseada em hardware atrela o funcionamento de um programa com alguma característica externa, seja esta dependente do ambiente de trabalho do cliente (número serial do disco rígido, CPU, etc...) ou por um dispositivo externo enviado pelo próprio desenvolvedor (*dongle*). Uma desvantagem de anexar a funcionalidade de um programa com uma característica externa é o transtorno gerado e quantidade de tempo despendida em situações de troca de equipamento (*upgrade* ou defeito) ou perda. Para agravar ainda mais a situação, o desenvolvedor poderia ter decretado falência o que inutilizaria o programa.

Uma estratégia contra engenharia reversa seria atrelar a CPU do usuário com os programas através de um esquema de criptografia de chave pública. Neste caso, a CPU conteria uma chave privada enquanto os programas só executariam se fossem criptografados com a correspondente chave pública. As desvantagens são as mesmas supracitadas, ou seja, custo elevado, desempenho inferior e estresse em caso de troca.

Todos meios de proteção, sejam eles em software ou em hardware, são passíveis de ataques e podem ser subvertidos. Mecanismos baseados em hardware oferecem um nível de proteção mais elevado, porém representam um custo agregado maior. Já as técnicas de transformação de código podem adicionar diferentes camadas de proteção a um custo inferior e, nada impede que ambas formas de proteção coexistam. O balanceamento entre o nível de proteção com o *overhead* em desempenho e custo de desenvolvimento é uma

decisão que cabe ao desenvolvedor do produto.

O restante deste capítulo é estruturado da seguinte forma. Seção 3.2 apresenta técnicas para ofuscação de código. Seção 3.3 apresenta técnicas de *anti-tampering*. Seção 3.4 descreve técnicas de marca d'água. E por fim Seção 3.5 apresenta as conclusões pertinentes.

3.2. Ofuscação de código

Técnicas de ofuscação de código alteram a estrutura de instruções de um programa com a intenção de fazer o conjunto menos aparente, dificultando com isso, o processo de engenharia reversa por um adversário [28, 36]. Estas técnicas tem sido utilizadas para proteger aplicações em situações que um competidor pode utilizar de engenharia reversa para extrair módulos/algoritmos proprietários [14]. Assim, ofuscação de código tem sido provada como uma técnica útil para esconder informação importante dentro de um software. Ofuscação de código tem sido utilizado para prover:

- **Diversidade**, em que a criação de versões distintas de um código sintaticamente dificulta ataques de diferença ("*diffing*") e exploração de vulnerabilidades por agentes maliciosos;
- **Proteção de propriedade intelectual**, em que o entendimento e a extração de códigos proprietários ou dados sigilosos são dificultados;
- **Integridade**, visto que um código mais complicado de entender é mais complicado de ser modificado.

Por outro lado, como qualquer conhecimento, ofuscação de código pode ser utilizada em um contexto malicioso. Especificamente, desenvolvedores de código malicioso frequentemente fazem uso de ofuscação de código para prevenir análise por especialistas ou por ferramentas de análise, assim como para evasão de detecção por detectores de código malicioso. Os trabalhos de Fred Cohen em construção e detecção de códigos maliciosos e ofuscação para criação de diversidade foram pioneiros na área de ofuscação de código [4, 5, 6, 7].

Ofuscar um programa refere-se a produzir um novo programa semanticamente equivalente em que a extração de informações por engenharia reversa é mais difícil de ser realizada neste do que no programa original. O conceito de dificuldade está associado à quantidade de tempo, dinheiro e poder computacional despendidos a mais na análise do programa transformado em relação ao programa original. O impacto de uma determinada transformação é mensurável de acordo com o *overhead*, confusão e dificuldade de um adversário de reverter a ofuscação [14]. As transformações efetivadas por um ofuscador de código são ditadas pela quantidade de *overhead* que um desenvolvedor esta disposto a colocar em sua aplicação. A seguir, detalhamos alguns cenários em que técnicas de ofuscação podem ser utilizadas.

- **Engenharia reversa maliciosa**. Um adversário faz uso de engenharia reversa para obter alguma vantagem pessoal sobre um determinado programa. Este pode, por

exemplo, desvendar o número serial de um programa para que o use sem restrições funcionais. Contudo, o adversário também pode implementar um gerador de número serial ou um *patch* que uma vez distribuído para outros usuários causaria prejuízos para o desenvolvedor, pois um usuário com acesso ao gerador de número serial ou *patch* não necessitaria adquirir o programa. Um outro exemplo de adversário, envolve um desenvolvedor concorrente extrair conhecimento de um algoritmo/módulo proprietário para utilizá-lo em seus próprios produtos.

- **Gerenciamento de direitos digitais (DRM)**. DRM (Digital Rights Managements) é uma tecnologia que visa restringir a cópia de um conteúdo digital por parte dos usuários. Em sistemas DRM de aúdio tipicamente uma mídia é criptografada com uma determinada chave criptográfica que é embarcada em um aparelho reprodutor. Para ouvir a mídia a mesma tem que ser decriptografada pelo aparelho reprodutor. Um adversário faz uso de engenharia reversa para obter a chave criptográfica embarcada no aparelho, possibilitando-o converter uma mídia criptografada em uma mídia em texto claro. Com a mídia em texto claro, um adversário pode escutá-la em qualquer aparelho reprodutor como pode distribuí-la para outros usuários.

- **Infra-estrutura avançada de medição**. O número de dispositivos de medição baseados em software embarcado vem crescendo consideravelmente. Um risco crítico associado à presença de software em dispositivos de medição é o interesse na adulteração do software para se obter vantagens pessoais. Por exemplo, para medidores elétricos um adversário poderia utilizar de engenharia reversa para modificar o software, permitindo-o conectar e desconectar clientes em tempos específicos remotamente, modificar dados de medição ou constantes de calibração, alterar a frequência de comunicação e inutilizar o dispositivo de medição. Em uma rede fortemente interligada como em *Smart Grids*, um adversário poderia causar um colapso na rede elétrica nacional caso estes softwares fossem atacados [59].

- **Redes de sensores sem fio**. Softwares também estão presentes em redes de sensores sem fio. Analogamente ao cenário anterior, é crítico que estes softwares não sofram adulteração. Em um contexto militar, por exemplo, um ataque envolveria modificar os softwares destes sensores de tal forma que estes enviassem informações falsificadas quanto ao movimento das tropas, radioatividade, substâncias químicas, etc... para a central de comando.

Antes de proceder com a descrição de técnicas de ofuscação, descrevemos na Figura 3.1 as etapas de engenharia reversa e compilação. Compilação é o processo de tradução de um programa em uma linguagem fonte (alto nível) para uma linguagem de máquina (baixo nível). Este processo é realizado através de etapas sucessivas que convertem o código a cada etapa para um nível inferior até atingir o nível de linguagem de máquina (*assembly*). Engenharia reversa é o processo oposto, ou seja, visa reconstruir a estrutura semântica de uma linguagem de baixo nível. Em geral, este processo é dividido em duas partes: *disassembly*, que converte uma sequência de bits em uma linguagem *assembly*; e decompilação, que reconstrói a estrutura semântica de um código *assembly* [35].

Técnicas de ofuscação pode dificultar as duas fases da engenharia reversa: a de *disassembly* para que esta retorne uma linguagem *assembly* incorreta [36]; ou a de de-

compilação fazendo com que as ferramentas de análise estática retornem grandes aproximações, dificultando-se assim, a recuperação da semântica do código. O processo de decompilação pode ser dificultado pela inserção de expressões booleanas complexas, ponteiros, saltos indiretos ou espúrios e *aliasing* [22, 16, 37, 21, 26]. *Aliasing* descreve uma situação em que uma posição de um dado na memória pode ser acessado através de diferentes nomes simbólicos em um programa, em que a modificação de um dado por um nome implicitamente modificaria todos os outros nomes apontados.

As técnicas de otimização aplicadas por compiladores geram um código mais ofuscado. Entretanto, estas buscam o aumento de desempenho da aplicação através da reordenação/substituição de instruções e/ou bloco de instruções. Já as técnicas de ofuscação de código visam aplicar transformações de código para dificultar seu discernimento, mesmo que essas degradem o desempenho e/ou aumentem o tamanho do código final.

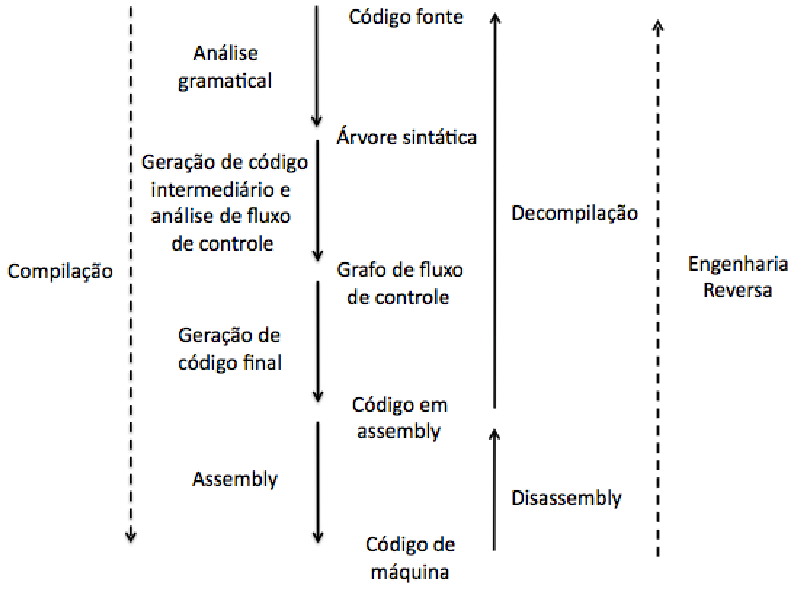


Figura 3.1. Etapas de compilação e engenharia reversa.

A base das ferramentas de análise de programas é o *disassembly* cuja função é a engenharia reversa de um código binário para recuperar o conjunto de instruções *assembly*. Contudo, como o processo de *disassembly* não é ciência exata, ferramentas de análise estática de programas que dependem de sua saída podem gerar resultados incorretos. Estas ferramentas são responsáveis pela abstração dos procedimentos, geração do grafo de fluxo de controle, análise do fluxo de dados e decompilação. Os *disassemblers* podem ser genericamente divididos em dois tipos: varredura linear e transversal recursivo [36].

Os *disassemblers* de varredura linear começam pelo processamento do segmento de texto (.text) da imagem do código binário obtido no cabeçalho do arquivo. O processo de *disassembly* é realizado sequencialmente até atingir o final do segmento de texto. Contudo, diante de dados inseridos no segmento de texto o processo de *disassembly* de varredura linear pode ser comprometido. O código da Figura 3.2 ilustra a situação em que o dado ‘db 0E8h’ seria interpretado como se fosse código. Caso os bytes coincidissem

```
Main:
      ...
      jmp     Func
      db      0E8h
Func:
      ...
```

Figura 3.2. Trecho de código contendo dado no segmento de texto de um binário.

com alguma codificação de alguma instrução *assembly*, o processo de *disassembly* converteria os bytes erroneamente. Em situações de não coincidência, estes *disassemblers* notificam através de pontos de interrogação os bytes não convertidos. Uma propriedade interessante destes *disassemblers* é que eventualmente ocorre a auto sincronização após um determinado número de instruções mal convertidas [36].

Um *disassembler* transversal recursivo visa a superar esta fraqueza fazendo com que o processo de *disassembly* acompanhe do fluxo de controle do programa. Caso houvesse dados no meio do fluxo de instruções (como o exemplo acima), o *disassembler* acompanharia o salto, e consequentemente, não interpretaria erroneamente o dado 'db 0E8h'. Contudo, o processo de *disassembly* é mais complexo, pois este tem que deduzir os possíveis destinos estaticamente, que nem sempre é uma tarefa trivial. Saltos indiretos e *aliasing* são desafios para que esta dedução seja feita corretamente.

Formalmente, dado um código C_σ com um "segredo" σ nele embarcardo, um ofuscador transforma este em um código C'_σ cuja extração do segredo σ é mais difícil de ser realizada em C'_σ do que C_σ. Para esta etapa de transformação temos uma biblioteca de transformações de código que preservam a semântica τ e um conjunto de ferramentas de análise de programas A que servem de auxílio para as transformações provendo informações estáticas e/ou dinâmicas sobre o programa C_σ. Informações dinâmicas são coletadas através da execução do programa diante de um determinado conjunto de entrada I. Figura 3.3 ilustra um processo de ofuscação, em que K é a parametrização de quais ofuscações foram utilizadas, assim como os pontos do código em que foram aplicadas.

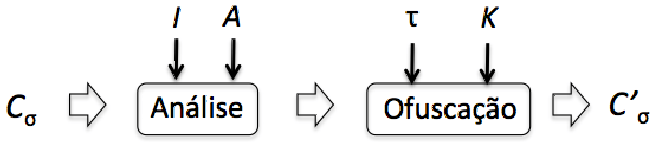


Figura 3.3. Processo de ofuscação.

As técnicas de ofuscação podem ser classificadas em estáticas ou dinâmicas. As estáticas podem ser: transformações de (*layout*) que removem informações de formatação de arquivos ou substituem identificadores; transformações de controle que distorcem o fluxo de controle pela inserção de código espúrio, reordenação do fluxo de execução das instruções através de saltos condicionais/incondicionais ou eliminação das estruturas de controle (laços de repetição e instruções condicionais); e transformações de dados que substituem as instruções ou os dados por novas representações. As dinâmicas transformam o código do programa em tempo de execução e por conseguinte, são mais resilientes

diante de análise puramente estática.

As transformações podem ser feitas tanto no código de alto-nível — que posteriormente é compilado de maneira a gerar um código binário ou um código intermediário mais complexo — como diretamente no código binário. Um código binário possui mais informação que pode ser ofuscada do que um código de alto-nível, porém o processo de modificar um binário é muito mais complexo. Outra vantagem é que a manipulação no binário não precisa se preocupar com o processo de otimização dos compiladores uma vez que a ofuscação é realizada diretamente no código final. A seguir, descreveremos técnicas aplicáveis no código de alto-nível e outras aplicáveis no código de baixo-nível.

3.2.1. Ofuscação estática

3.2.1.1. Transformações de *layout*

Técnicas de transformação de estrutura são as mais simples. Um exemplo é a remoção de informações de formatação de arquivos normalmente presentes em arquivos '.class' da linguagem Java, como é o caso do ofuscador Crema [10]. Outra técnica que se enquadra nas transformações de estrutura consiste na renomeação dos identificadores, o que dificulta a análise do código fonte. A renomeação sendo feita diretamente no código binário, no nível de registradores, é utilizada para criar cópias distintas com o âmbito de dificultar um ataque de diferença ou de assinatura. Esta técnica foi utilizada pelo vírus Win32/Regswap que, a cada replicação, faz a troca dos registradores utilizados. Figura 3.4 ilustra trechos de duas gerações deste vírus, em que a geração mais abaixo substitui o registrador 'edx' por 'eax', 'edi' por 'ebx', 'esi' por 'edx', 'eax' por 'edi' e'ebx' por 'esi'.

```
5A              pop  edx
BF04000000      mov  edi,0004h
8BF5            mov  esi,ebp
B80C000000      mov  eax,000Ch
81C288000000    add  edx,0088h
8B1A            mov  ebx,[edx]
899C8618110000mov  [esi+eax*4+00001118],ebx

58              pop  eax
BB04000000      mov  ebx,0004h
8BD5            mov  edx,ebp
BF0C000000      mov  edi,000Ch
81C088000000    add  eax,0088h
8B30            mov  esi,[eax]
89B4BA18110000mov  [edx+edi*4+00001118],esi
```

Figura 3.4. Atribuição de diferentes registradores em duas gerações do Win32/Regswap [25].

3.2.1.2. Transformações de controle

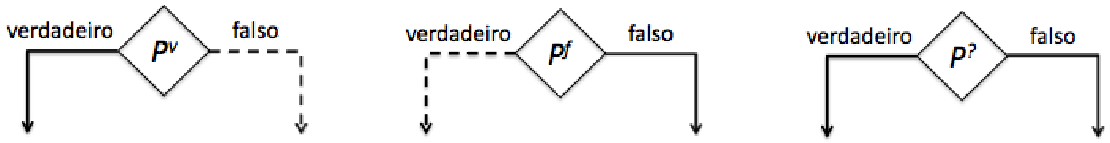
Transformações de controle visam a dificultar a análise do fluxo de controle. A análise do fluxo de controle é dependente do grafo de fluxo de controle que representa o fluxo de execução de todas possíveis execuções de um programa. Técnicas para ofuscar o fluxo de

controle consistem na inserção, reordenação ou eliminação de blocos básicos no fluxo de execução do programa.

A inserção de códigos espúrios no fluxo de controle degrada a precisão de uma análise de fluxo de controle. Esta técnica consiste na adição de saltos condicionais ou incondicionais para que estes códigos sejam incorporados no fluxo de controle do programa. Os saltos condicionais são normalmente vinculados com *predicados opacos*: expressões cujos valores são de conhecimento do programador ou ofuscador, porém de difícil discernimento para um analisador estático ou um analista [14]. Quanto mais difícil determinar o resultado de um predicado opaco, mais robusta será a técnica de ofuscação, pois uma ferramenta de engenharia reversa ou um analista que não consiga determinar estaticamente o valor de um predicado, tem que considerar ambos caminhos do fluxo de controle como possíveis, degradando, portanto, as informações coletadas.

Em outras palavras, predicados opacos provêem um modo para atacar a fase de geração do grafo de fluxo de controle de uma analisador estático. A utilização destas instruções de salto ofuscadas força um analisador adicionar arestas desnecessárias no grafo de fluxo de controle, fazendo com que os resultados sejam menos precisos.

Os predicados opacos podem ser classificados em verdadeiros, falsos ou indeterminados e são denotados respectivamente por P^v, P^f e $P^?$. Figura 3.5 ilustra como estes tipos de predicados opacos são representados graficamente. Para a construção de predicados opacos uma das opções triviais envolve a adição de tautologias para predicados verdadeiros e contradições para predicados falsos.

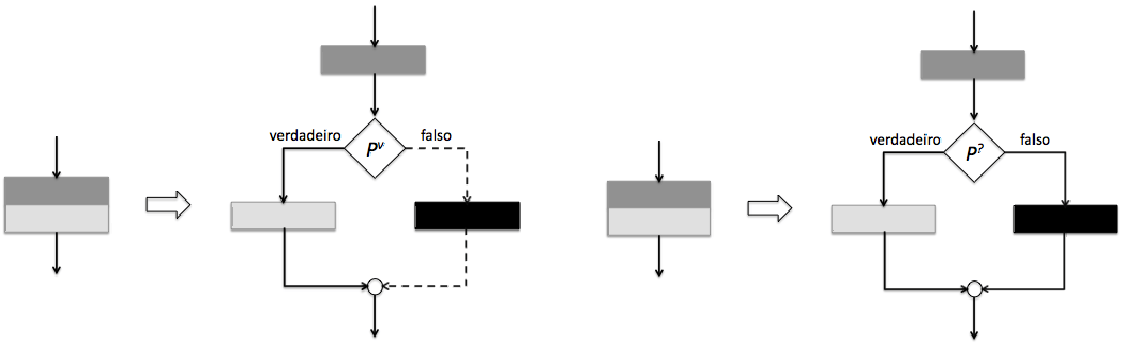


(a) Verdadeiro. (b) Falso. (c) Indeterminado.

Figura 3.5. Tipos de predicados opacos.

Estratégias para inserção de blocos espúrios no fluxo de controle envolvem a utilização de trechos de códigos que podem ser ou não executados [14, 16, 32]. Caso sejam executados, a semântica dos códigos espúrios deve ser equivalente à semântica dos blocos correspondentes. Blocos correspondentes são blocos que não foram executados, pois o fluxo de execução trilhou o caminho dos blocos espúrios. Figura 3.6(a) representa a situação em que o trecho de código em preto nunca será executado, pois o predicado sempre retornará um valor verdadeiro. Figura 3.6(b) representa a situação em que o trecho de código em preto pode ou não ser executado, pois o predicado é indeterminado, contudo a semântica do código preto, neste caso, é equivalente ao código em cinza claro.

A ofuscação por inserção de blocos espúrios no fluxo de controle visa a degradar a precisão de uma análise de fluxo de dados. Uma possível variação, exibida no código da Figura 3.7, seria a inserção de um dado ‘db 0E8h’ através de um predicado opaco ‘cmp eax, eax’ para degradar a etapa de *disassembly*. Nesta situação, *disassembler* poderia considerar a instrução da linha ‘4’ como se fosse código, pois a instrução ‘3’ é um salto incondicional para ela, porém é uma instrução que nunca vai ser executada, pois o salto



(a) Código em preto não executável.

(b) Código em preto executável com semântica equivalente ao código em cinza claro.

Figura 3.6. Inserção de informação espúria no fluxo de controle com predicados opacos.

da instrução ‘2’ sempre será executado dado que os dois operandos sendo comparados (‘eax’ e ‘eax’) na instrução ‘1’ são sempre iguais (predicado opaco).

```
Main:
1     cmp     eax,  eax
2     je      Func+1
3     jmp     Func
Func:
4     db       0E8h
5     push     0
6     call     ExitProcess
```

Figura 3.7. Ofuscação por inserção de dado espúrio por predicado opaco.

Figura 3.8 exibe o código gerado pelo depurador OllyDbg para a sequência de instruções da Figura 3.7. O depurador assumiu que ambos destinos dos saltos são válidos. Observe que o código gerado é muito diferente do código executável original. O depurador interpretou incorretamente o dado posicionado no começo de ‘Func’ como se fosse código. Como o dado ‘E8h’ coincide com o código de operação de uma instrução de chamada ‘call’, o depurador assumiu como se fosse uma instrução de chamada válida e prosseguiu o processo de conversão para instruções subsequentes de forma incorreta.

```
00401000  $ 3BC0          CMP EAX,EAX
00401002  .v74 03         JE SHORT sampleco.00401007
00401004  .vEB 00         JMP SHORT sampleco.00401006
00401006  > E8 6A00E800   CALL 01281075
0040100B  ? 0000          ADD BYTE PTR DS:[EAX],AL
0040100D  ? 00FF          ADD BH,BH
0040100F  ? 25 30304000   AND EAX,403030
```

Figura 3.8. Saída do depurador OllyDbg.

Outra técnica de transformação de controle consiste em achatar o fluxo de execução com o âmbito de dificultar a análise do fluxo de controle [13, 14, 26, 49]. O achatamento pode ser feito colocando cada bloco básico do programa dentro de um ‘*case*’

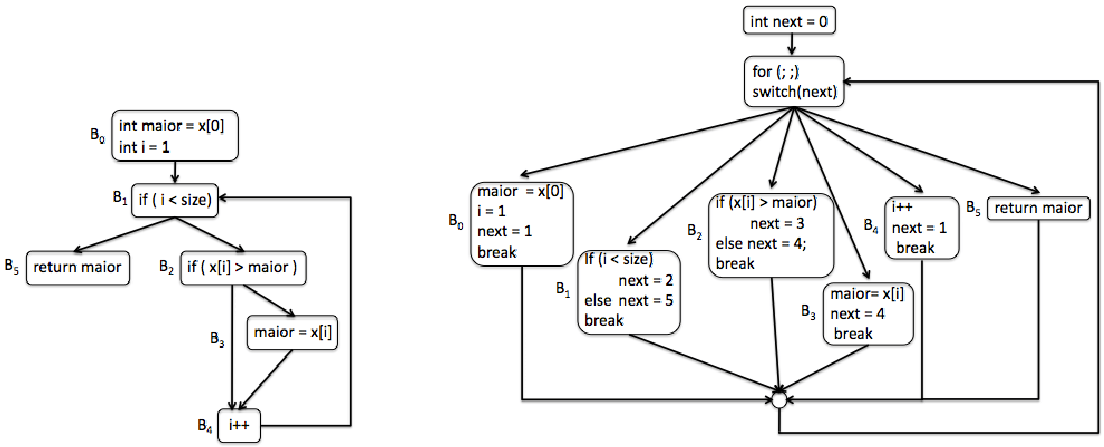
```
/* Código Original */

int maior(int x[], int size) {
     int maior = x[0];
     int i = 1;
     while ( i < size) {
          if ( x[i] > maior )
               maior = x[i];
          i++;
     }
     return maior;
}

/* Código Ofuscado */

int maior(int x[], int size) {
    int next = 0;
    for (;;){
          switch (next) {
               case 0: int maior = x[0]; int i =1; next = 1; break;
               case 1: if  (i < size) next = 2 else next = 5; break;
               case 2: if  (x[i] > maior) next = 3; else next = 4; break;
               case 3: maior = x[i]; next = 4; break;
               case 4: i++; next = 1 ; break;
               case 5: return maior;
          }
     }
}
```

Figura 3.9. Ofuscação por achatamento do fluxo de execução.

de uma estrutura '*switch*', e esta estrutura dentro de um laço infinito. Os códigos da Figura 3.9 exibem esta transformação de ofuscação por achatamento do fluxo de execução. Figura 3.10 ilustra os grafos de fluxo de controle para o código não ofuscado e ofuscado.

O código exemplificado é simples, contudo, oferece subsídios para a compreensão da importância das técnicas de transformação no aumento da dificuldade de análise por um adversário. Pode-se aumentar o grau de dificuldade com o uso de *aliasing* de ponteiros na variável 'next' [22, 16, 37, 21, 26], fazendo com que a análise estática retorne resultados pouco precisos quanto a variável 'next', dificultando-se portanto um analista de acompanhar o fluxo de execução do programa. Apesar de um grande número de trabalhos em algoritmos para análise de *aliasing*, nenhum deles mostrou ser escalável para programas com milhões de linhas de código, concentrando-se a maioria na ordem de cem mil linhas de código [55].

Outra técnica de transformação de controle envolve a reordenação de código cuja idéia é alterar a ordem sintática das instruções de um programa enquanto a ordem de execução do programa é preservada através da inserção de instruções de salto. O código da Figura 3.11 exibe a rotina original da Figura 3.9 ofuscada pela técnica de reordenação de código. Observe que os endereços dos saltos também foram ofuscados por um vetor de ponteiros, o que dificulta ainda mais a análise do código. Vale ressaltar que não é



(a) Código não ofuscado. (b) Código ofuscado por achatamento.

Figura 3.10. Grafo do fluxo de controle da rotina “maior”.

necessário o uso de saltos incondicionais para reordenação de código: uma estratégia mais robusta contra engenharia reversa seria o uso de instruções condicionais com predicados opacos.

```
int maior(int x[], int size) {
     int i, maior;
     char* jtab[ ]= {&&l0, &&l1, &&l3};
     goto *jtab[0];
l2:
     while ( i < size) {
          if ( x[i] > maior )
               maior = x[i];
     i++;
}
     goto *jtab[3];
l0:
     maior = x[0];
     goto *jtab[1];
l1:
     i = 1;
     goto l2;
l3:
     return maior;
}
```

Figura 3.11. Ofuscação por reordenação de código.

3.2.1.3. Transformações de dados

Transformações de dados envolvem a conversão de uma representação de uma estrutura de dados, tipo de dados ou instruções em outra representação que seja mais difícil para um atacante de entendê-la. Funções de criptografar/decriptografar em criptografia são meios

```
CODE:00401000 start           proc near
CODE:00401000                 push    1
CODE:00401002                 push    2
CODE:00401004                 call    sub_401010
CODE:00401009                 push    0
CODE:0040100B                 call    ExitProcess

CODE:00401010 sub_401010      proc near
CODE:00401010                 mov     eax, [esp+arg_0]
CODE:00401014                 mov     ebx, [esp+arg_4]
CODE:00401018                 add     eax, ebx
CODE:0040101A                 retn    8
```

(a) Chamada de procedimento não ofuscada.

```
CODE:00401000 start           proc near
CODE:00401000
CODE:00401000 var_8           = dword ptr -8
CODE:00401000 var_4           = dword ptr -4
CODE:00401000
CODE:00401000                 push    1
CODE:00401002                 push    2
CODE:00401004                 push    offset loc_40100F
CODE:00401009                 push    offset loc_401016
CODE:0040100E                 retn
CODE:0040100F loc_40100F:
CODE:0040100F                 push    0
CODE:00401011                 call    ExitProcess
CODE:00401016 loc_401016:
CODE:00401016                 mov     eax, [esp+4]
CODE:0040101A                 mov     ebx, [esp+10h+var_8]
CODE:0040101E                 add     eax, ebx
CODE:00401020                 retn    8
```

(b) Chamada de procedimento ofuscada.

Figura 3.12. Ofuscação de uma instrução de chamada ‘*call*’.

para transformação de dados. Contudo, para que uma operação seja realizada neste tipo de ofuscação, os dados devem ser primeiramente desofuscados (decriptografados), o que os torna acessíveis para um adversário de engenharia reversa.

Uma técnica de transformação de dados envolve ofuscar convenções relacionadas ao encapsulamento de funções de uma linguagem *assembly*. Este encapsulamento é feito através de convencões relacionadas com instruções de chamada ‘*call*’ e de retorno ‘*ret*’. Na maioria dos códigos gerados por compiladores, uma instrução de chamada ‘*call*’ é emparelhada com uma instrução de retorno ‘*ret*’, e a instrução de retorno exerce a função de transferir o controle para a instrução subsequente a instrução de chamada. Porém, esta convenção não precisa ser seguida, fazendo com que a fase de geração de fluxo de controle na etapa de decompilação retorne resultados imprecisos. A etapa de *disassembly* também pode ser prejudicada caso o *disassembler* seja transversal recursivo, pois o mesmo utiliza do comportamento do fluxo de controle para converter as instruções.

Instruções de chamada e retorno não são atômicas, assim as mesmas podem ser ofuscadas substituindo-as por instruções que exerçam semântica equivalente. Figura 3.12 ilustra esta situação, na coluna a esquerda uma função invoca a outra através de uma instrução normal de chamada de função; na coluna a direita temos um código com semântica equivalente que utiliza de uma sequência de duas instruções ‘*push*’ e uma instrução ‘*ret*’. A primeira instrução ‘*push*’ coloca o endereço da instrução após a instrução de chamada, a segunda instrução ‘*push*’ coloca o alvo da instrução de chamada e a instrução ‘*ret*’ transfere a execução para o endereço alvo da chamada.

A semântica das instruções '*call*' e '*ret*' clássicas pode ser divididas em duas partes: manipulação do endereço de retorno da pilha e transferência do controle do programa. Para ofuscar uma chamada de procedimento (ou um retorno de um procedimento) estas duas partes da semântica das instruções tem que ser separadas e efetuadas usando um outro conjunto de instruções. Este conjunto de instruções não precisa estar contíguo dentro do código do programa, podendo estar distribuído e/ou misturado com outras instruções. Instruções que modificam o ponteiro da pilha, tais como atribuição, incremento e decremento também poderiam ser utilizados para ofuscação. Por outro lado, instruções '*call*' ('*ret*') podem ser empregadas com outras finalidades do que fazer (retornar de) uma chamada de procedimento, como identificadas por Lakhotia *et al.* [45]:

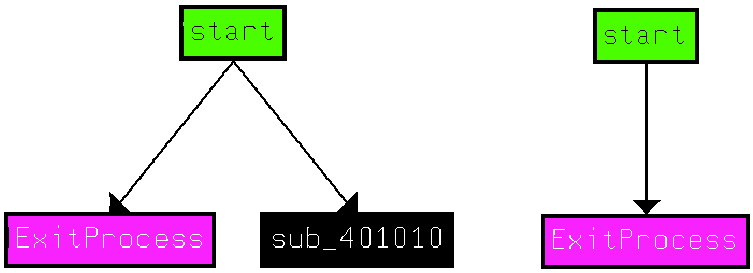
1. Uma chamada '*call*' simulada. A semântica de uma instrução '*call end*' é a seguinte: o endereço da instrução seguinte da instrução de chamada é colocada na pilha e o fluxo de controle é transferido para o endereço '*end*'. Esta semântica pode ser simulada através de uma combinação de duas instruções '*push*' e uma instrução '*ret*' como ilustrado na Figura 3.12. O código desta Figura exibe duas rotinas, uma que coloca dois valores na pilha (no caso '1' e '2') e faz a chamada para uma função 'sub_401010' que adiciona estes dois argumentos e retorna para a função principal. O mesmo efeito poderia ser adquirido através de uma instrução '*push*' que colocaria o endereço da instrução seguinte da instrução de chamada e uma instrução '*jmp*' que transferiria o fluxo de execução para o alvo da instrução de chamada. Esta situação está ilustrada na Figura 3.14(a).

2. Uma chamada '*call*' para transferir fluxo de controle. Aqui, a instrução de chamada seria utilizada para transferência de controle em que o endereço de retorno seria simplesmente descartado através da remoção do valor do topo da pilha.

3. Um retorno '*ret*' simulado. Uma instrução '*ret*' é complementar a uma instrução '*call*'. Esta retira o endereço de retorno (normalmente colocada por uma instrução '*call*') do topo da pilha e transfere o fluxo de controle para este endereço. Esta semântica pode ser adquirida através de uma instrução '*pop reg*' que retira o valor do topo da pilha colocando-o no registrador '*reg*' e uma instrução '*jmp reg*' que transfere o fluxo de controle para o endereço contido no registrador.

4. Um retorno '*ret*' para transferir fluxo de controle. Aqui, a instrução não retorna para um local de chamada. Uma instrução '*ret*' pode ser utilizada para transferir fluxo de controle para outra instrução sem que esta esteja relacionada com uma instrução de chamada. Por exemplo, uma instrução '*ret*' pode ser usada para simular uma instrução '*call*', como descrito anteriormente.

Estas ofuscações distorcem parâmetros importantes que são utilizados para análise automática e manual. Ainda que um programador experiente possa descobrir as ofuscações, o tempo que o mesmo leva pode ser muito precioso quando se trata de um código malicioso na Internet. A substituição de instruções de chamada, em particular, faz com que uma grande parte dos métodos automáticos para detecção de códigos maliciosos falhem uma vez que que esses dependem do reconhecimento de instruções de chamadas para identificar funções *kernel* utilizadas pelo programa. Por exemplo, o *disassembler*



(a) Código não ofuscado.

(b) Código ofuscado.

Figura 3.13. Grafo de chamadas para os códigos da Figura 3.12.

IDA Pro [61], vastamente utilizado pela indústria de antivírus, retorna resultados incorretos na geração do grafo de chamadas na presença desta técnica de ofuscação [45]. Figura 3.13 ilustra os grafos de chamadas gerados para o código não ofuscado à esquerda e o código ofuscado à direita. Nota-se que diante de ofuscação o *disassembler* não foi capaz de delimitar o procedimento 'sub_401016'.

Estas ofuscações de chamada e retorno de procedimento atingem duas fases importantes da análise de controle de fluxo interprocedural: identificação dos procedimentos e criação do grafo de chamadas. A maioria das linguagens *assembly* não fornecem mecanismos para encapsular procedimentos. Assim, *disassemblers* utilizam das instruções *call* e *ret* para determinar os limitantes das funções e para criar o grafo de chamadas [61]. Quando estas instruções são ofuscadas, as funções identificadas e o grafo de chamadas gerados podem ser questionáveis e qualquer análise subsequente duvidosa. Um método para análise sensível ao contexto de binários com instruções de chamada e retorno ofuscadas foi recentemente proposto por Boccardo [62]. O método é baseado na noção de contextos baseado na pilha que conseguem rastrear mudanças de contexto na pilha, e por conseguinte, gerar resultados mais precisos do que os métodos clássicos [3].

Outra transformação de dados em código *assembly* envolve instruções de salto incondicionais '*jmp*' [36]. A idéia consiste em substituir um salto incondicional por uma chamada de função e manipular o endereço de retorno da função para ser o endereço de destino do salto incondicional. Figura 3.14 exibe um ofuscamento de um salto incondicional. Figura 3.14(a) mostra um ofuscamento de chamada através do uso de uma instrução '*401004: push*' e uma instrução '*401009: jmp*'. Figura 3.14(b) ilustra o ofuscamento da instrução '*401009: jmp*' substituindo-a pela chamada de função '*40100B: call sub_401024*'. Instrução '*401024: xchg*' troca o conteúdo do registrador '*eax*' com o valor de retorno contido no topo da pilha '*esp+0*'. Instrução '*401027: add*' soma este valor com o deslocamento para o alvo do salto que foi colocado na pilha antes da chamada da função ('*401009: push*'). Instrução '*40102B: pop*' restaura o valor do registrador '*eax*' e instrução '*40102C: ret*' faz com que a execução seja transferida para o alvo do salto incondicional ('*401017: mov eax, [esp+4]*') .

As transformações de dados não são restritas as linguagens *assembly* como as técnicas de ofuscação de chamada e retorno de procedimento e de saltos incondicionais. Transformações podem ser feitas em uma linguagem de alto-nível para alterar o tipo de

```
CODE:00401000 start           proc near
CODE:00401000 var_8           = dword ptr -8
CODE:00401000 var_4           = dword ptr -4
CODE:00401000                 push    4
CODE:00401002                 push    2
CODE:00401004                 push    offset loc_40100B
CODE:00401009                 jmp     short loc_401012
CODE:0040100B loc_40100B:
CODE:0040100B                 push    0
CODE:0040100D                 call    ExitProcess
CODE:00401012 loc_401012:
CODE:00401012                 mov     eax, [esp+0Ch+var_8]
CODE:00401016                 mov     ebx, [esp+0Ch+var_4]
CODE:0040101A                 add     eax, ebx
CODE:0040101C                 retn    8
```

(a) Salto incondicional não ofuscado.

```
CODE:00401000 start           proc near
CODE:00401000                 push    4
CODE:00401002                 push    2
CODE:00401004                 push    offset loc_401010
CODE:00401009                 push    7
CODE:00401008                 call    sub_401024
CODE:00401010 loc_401010:
CODE:00401010                 push    0
CODE:00401012                 call    ExitProcess
CODE:00401012 start           endp
CODE:00401017                 mov     eax, [esp+4]
CODE:00401018                 mov     ebx, [esp+8]
CODE:0040101F                 add     eax, ebx
CODE:00401021                 retn    8
CODE:00401024 sub_401024      proc near
CODE:00401024                 xchg    eax, [esp+0]
CODE:00401027                 add     [esp+arg_0], eax
CODE:00401028                 pop     eax
CODE:0040102C                 retn
```

(b) Salto condicional ofuscado.

Figura 3.14. Ofuscação de um salto incondicional.

dados como inteiros, booleanos, *strings* ou estrutura de dados como vetores para que o código gerado seja mais complicado de ser entendido [17]. O tipo inteiro pode ser transformado alterando-se seu intervalo de valoração. Por exemplo, um inteiro 'i' pode ser representado por '$i' = c_1.i + c_2$', na qual 'c_1' e 'c_2' são constantes inteiras. Figura 3.15 ilustra este tipo de transformação utilizando $c_1 = 8$ e $c_2 = 3$. O único cuidado necessário ao utilizar esta técnica refere-se ao *overflow* uma vez que o intervalo de valores para o programa transformado é menor do que o intervalo de valores do programa original. O tipo booleano pode ser transformado utilizando-se valores múltiplos para representar os valores 'verdadeiro' e 'falso'.

Transformar *strings* em uma forma menos visível para um adversário é muito importante dado que *strings* tipicamente fornecem informações relevantes quanto o comportamento de um programa para um analista. Uma forma de mudar a representação de uma *string* consiste na implementação de uma rotina que produza a *string* desejada. Digamos que queremos ofuscar a string 'ofusca'. Figura 3.16 ilustra uma rotina para gerar esta *string*, passando a entrada de valor '1' para a função 'gera'.

Técnicas de transformação de vetores são baseadas no particionamento, fusão ou redimensionamento do número de dimensões. Figura 3.17 exibe uma transformação por particionamento de um vetor em dois sub-vetores. Figura 3.18 exibe uma transformação por fusão de dois vetores em um único vetor. Figura 3.19 exibe a transformação de

```
/* Código Original */

int i = 1;
while (i < 1000) {
   ... A[i] ...;
   i++;
}

/* Código Ofuscado */

int i = 11;
while (i < 8003) {
   ... A[(i-3)/8] ...;
   i+=8;
}
```

Figura 3.15. Transformação de inteiros.

```
String gera (int n) {
     String S;
     l1: if (n==1) {S[i++] = "o"; goto l2; }
     l2: if (n==2) {S[i++] = "f"; goto l3;}
     l3: if (n==3) {S[i++] = "u"; goto l4;}
     l4: if (n==4) {S[i++] = "s"; goto l5;}
     l5: if (n==5) {S[i++] = "c"; goto l6;}
     l6: if (n==6) {S[i++] = "a"; return S;}
}
```

Figura 3.16. Transformação de *strings*.

aumento do número de dimensões de um vetor. Figura 3.20 exibe a transformação de diminuição do número de dimensões de um vetor.

3.2.2. Ofuscação dinâmica

As técnicas de ofuscação de código até então abordadas são estáticas, na qual o código é transformado antes de sua execução. Apesar de estas aumentarem o grau de dificuldade de um adversário em descobrir alguma propriedade do código, as mesmas não são tão robustas diante de ferramentas de análise dinâmica. Técnicas de ofuscação dinâmica transformam o código do programa em tempo de execução e por conseguinte, tornam a tarefa de desvendar um segredo mais complicada para ambos tipos de análise: estática ou dinâmica. Em contrapartida, estas degradam desempenho, pois uma vez que o segmento de código é alterado, o *pipeline*[1] de instruções tem que ser esvaziado, o conteúdo do cache de dados tem que ser escrito na memória e o cache de instruções tem que ser invalidado.

Fred Cohen [6] aponta duas estratégias para ofuscação dinâmica. Ambas estratégias consistem de códigos auto modificáveis, porém a primeira utiliza-se de uma parte constante do código criptografada (ou compactada) que é decriptografada (ou descom-

[1]Técnica de hardware que possibilita a CPU buscar uma ou mais instruções além da próxima a ser executada.

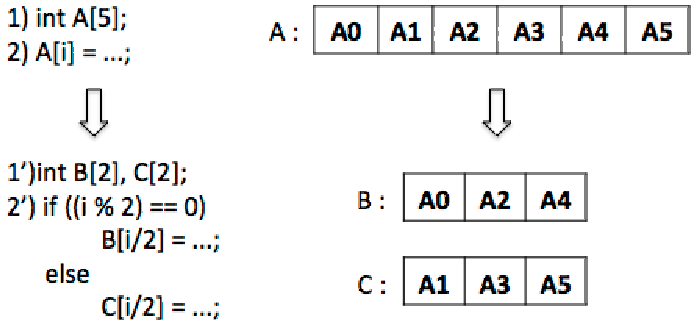


Figura 3.17. Transformação por particionamento de um vetor.

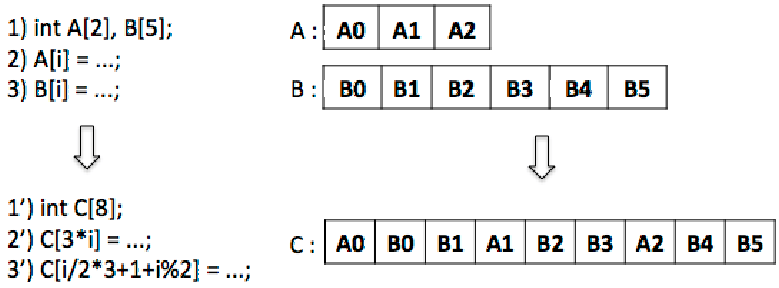


Figura 3.18. Transformação por fusão de um vetor.

pactada) por uma rotina embarcada no código que é executada durante a execução do programa; a segunda visa a transformar o fluxo de execução do programa constantemente.

A primeira estratégia tipicamente está atrelada a aplicação de criptografia (ou compactação) para esconder um trecho de código. Esta técnica de proteção, apesar de ser tradicional, não é considerada resiliente diante de ataques '*Man-At-The-End*, pois em algum momento de execução, o código torna-se visível para um adversário. Um adversário através de um depurador pode estabelecer critérios de parada e ter acesso ao código em texto claro. Outra possibilidade, ainda mais simples, é um adversário usar de emulação para ter acesso a todas instruções que foram executadas pelo programa. O código da Figura 3.21 ilustra esta técnica em que o trecho de código das linhas 3-9 encontram-se criptografados por um ou exclusivo (XOR) com a chave '34'. Durante a execução do programa, o código entre as linhas 'l3' e 'l9' é decriptografado pelo código das linhas 'l1' e 'l2' para posteriormente ser executado.

Kanzaki *et al.* [33] explora a idéia de mudar o fluxo de execução do programa constantemente e não somente durante algum ponto de execução como a técnica discutida anteriormente. Para isso, a técnica alterna uma sequência de instruções originais por uma sequência de instruções falsas e vice versa. O objetivo é deixar instruções originais o menor tempo possível na memória e disfarçá-las com instruções falsas para evitar que um analista descubra propriedades importantes do programa. Antes de executar uma instrução falsa, a instrução verdadeira é colocada no endereço da instrução falsa, e após executar a instrução verdadeira, a instrução falsa é recolocada no endereço da instrução verdadeira. Este procedimento é sucintamente ilustrado na Figura 3.22 em que a instrução falsa 'instf' do endereço '2' é substituída por uma instrução verdadeira 'instv' após exe-

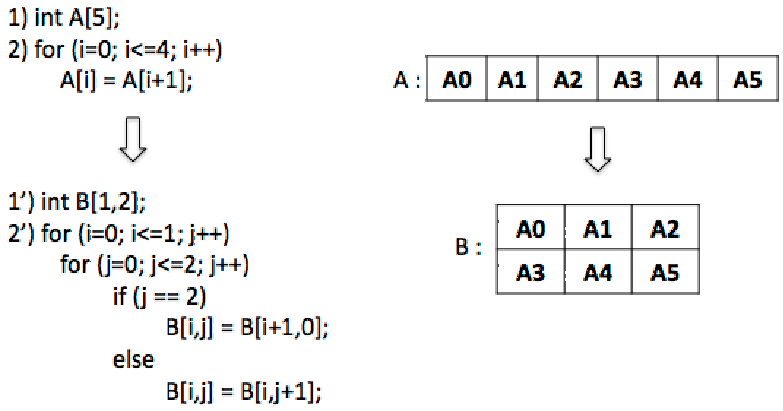


Figura 3.19. Transformação por aumento do número de dimensões de um vetor.

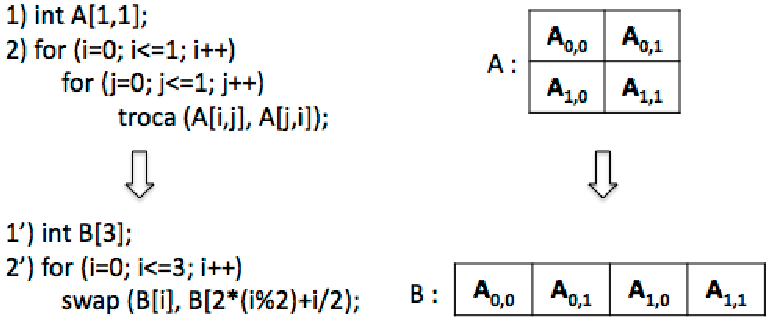


Figura 3.20. Transformação por diminuição do número de dimensões de um vetor.

cução da instrução do endereço '1'. Após a execução de 'instv', a instrução do endereço '3' substitui 'instv' do endereço '2' pela instrução falsa 'instf'. É claro que a técnica de criptografia (ou compactação) também poderia ser utilizada para constantemente alterar o fluxo de execução, ou seja, em um curto espaço de tempo o código estaria decriptografado (ou descompactado) na memória enquanto que na maior parte do tempo o código estaria inteligível.

Madou *et al.* [46] implementou um ofuscador dinâmico para fusão de código. A técnica envolve o compartilhamento de endereços de memória por duas ou mais funções distintas através da criação de um padrão de bytes comum a essas funções e da criação de *scripts* encarregados de substituir o padrão pela função a ser executada. Figura 3.23 ilustra essa técnica na qual é criado um padrão '*p*' para duas funções 'f_1' e 'f_2' cujos bytes diferem nas posições '2' e '3', e portanto, são representados por '?' (qualquer *byte*) no padrão 'p'. Os *scripts* 's_1' e 's_2' são encarregados de substituir os bytes correspondentes das funções em tempo de execução.

Vale ressaltar que um adversário pode interceptar estes *scripts* uma vez que movimentações de código em posições constantes (no padrão 'p') despertam a atenção de um adversário, auxiliando-o a desvendar propriedades do programa. Os autores desta técnica propõem o uso de criptografia nos *scripts* para aumentar a resiliência do método.

Aucsmith [9] desenvolveu uma técnica de ofuscação dinâmica que divide o pro-

```
int maior(int x[], int size) {
l1:     char* p=&&comeco;
l2:     while (p < (char *)&&fim)    *p++ ^=34; // xor com a chave 34
comeco:
l3:     int maior = x[0];
l4:     int i = 1;
l5:     while ( i < size) {
l6:          if ( x[i] > maior )
l7:               maior = x[i];
l8:          i++;     }
l9:    return maior;
fim:
}
```

Figura 3.21. Técnica de criptografia utilizada para proteger o código.

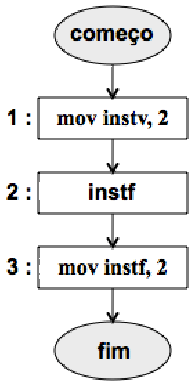


Figura 3.22. Ofuscação dinâmica através da intercalação de instruções falsas com instruções originais.

grama em blocos e permuta-os ciclicamente através de ou exclusivos (XORs). Antes da execução, os blocos são separados em duas regiões de memória: superior e inferior. A execução dos blocos é feita de maneira alternada e de forma que a cada iteração é feito um XOR de cada bloco da memória superior com cada bloco correspondente da memória inferior de modo que sempre exista um bloco em texto claro que vai ser executado na iteração subsequente. Nas iterações pares, partes da memória superior são mescladas com as partes da memória inferior, e nas iterações ímpares partes da memória inferior são mescladas com as partes da memória superior.

Figura 3.24 ilustra a permutação de dois trechos de código 'A' e 'B', lembrando que $A \oplus B = C$ e $C \oplus B = A$. Nota-se que a permutação de dois trechos de código envolve três etapas 'I', 'I_1' e 'I_2' e para que os trechos retornem a configuração inicial 'I_0' necessita-se de seis etapas: de 'I_0' até 'I_5'.

Figura 3.25 exibe como é a proteção de uma função constituída pelos blocos 'A' até 'F' pela técnica de Aucsmith. Observa-se que a função foi dividida em seis blocos e que foi configurado um estado inicial 'I_0' para os blocos do programa de modo que em todas iterações, o próximo bloco a ser executado esteja em texto claro. A sequência de estados configura-se de acordo com a intercalação dos XORs dos blocos da memória

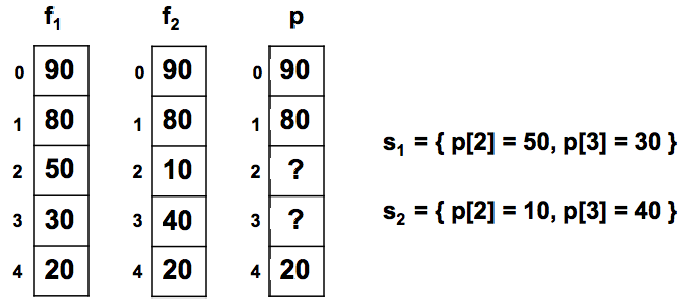


Figura 3.23. Ofuscação dinâmica por fusão de código.

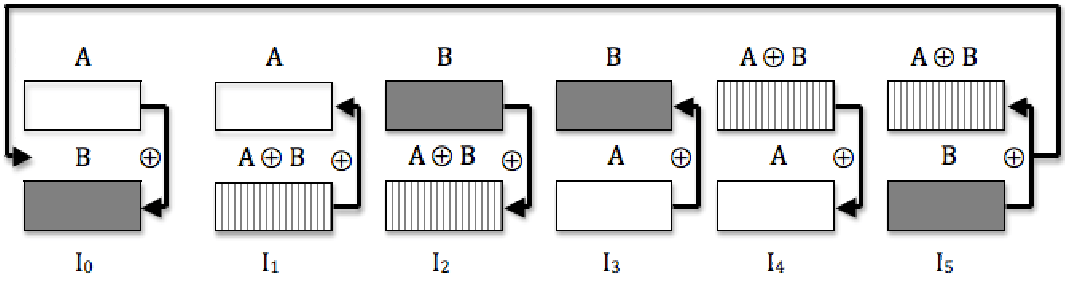


Figura 3.24. Permutação de trechos de código utilizando XOR.

superior com os blocos da memória inferior. O autor também propõe o uso de uma chave criptográfica para aumentar a robustez do método.

3.2.3. Ofuscação maliciosa

Programadores de código malicioso também aplicam técnicas de ofuscação de código e possuem a disposição deles um arsenal de ferramentas de ofuscação, tais como Mistfall [64] da comunidade *BlackHat*, assim como ferramentas comercialmente disponíveis como PECompact [65]. Estas modificações visam evadir detecção pelos varredores de código malicioso, pois uma boa parte da "inteligência" destes varredores é baseada no padrão de bits, chamado de assinatura[39]; e ofuscações alteram o padrão de bits e, consequentemente a assinatura.

Dois conceitos de ofuscação surgiram da comunidade Blackhat: polimorfismo e metamorfismo [57, 25]. Ambos conceitos se enquadram em ofuscação dinâmica. Um código polimórfico é constituído de duas partes: rotinas de criptografia e uma parte constante de código. A natureza polimórfica se dá através de transformações aplicadas no par de rotinas de criptografia, podendo gerar diferentes métodos para criptografar a parte constante do código a cada replicação do código do programa. Figura 3.26(a) exibe a representação de diferentes gerações de um vírus polimórfico e o correspondente código malicioso decriptografado. Observa-se que o corpo decriptografado do código malicioso permanece inalterado durante as gerações.

Um código metamórfico é constituído de uma rotina que transforma todo o código do programa, incluindo a rotina de transformação. Assim, não existem especificadamente rotinas de criptografia e uma parte constante de código como nos códigos polimórficos. Cada replicação de um código metamórfico pode gerar versões totalmente diferentes da

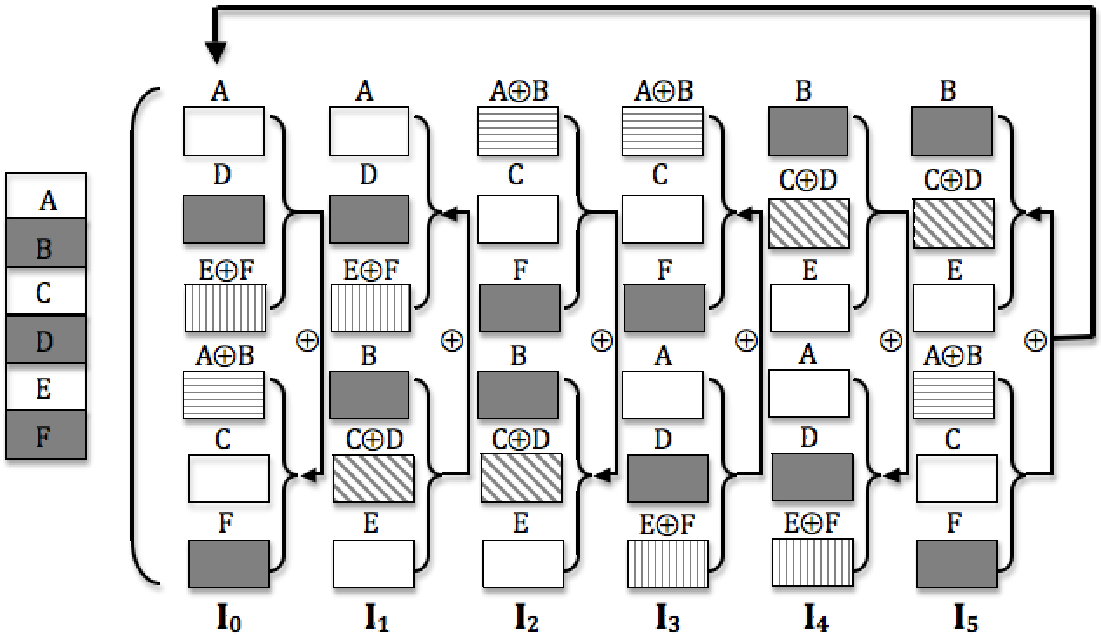


Figura 3.25. Técnica de Aucsmith.

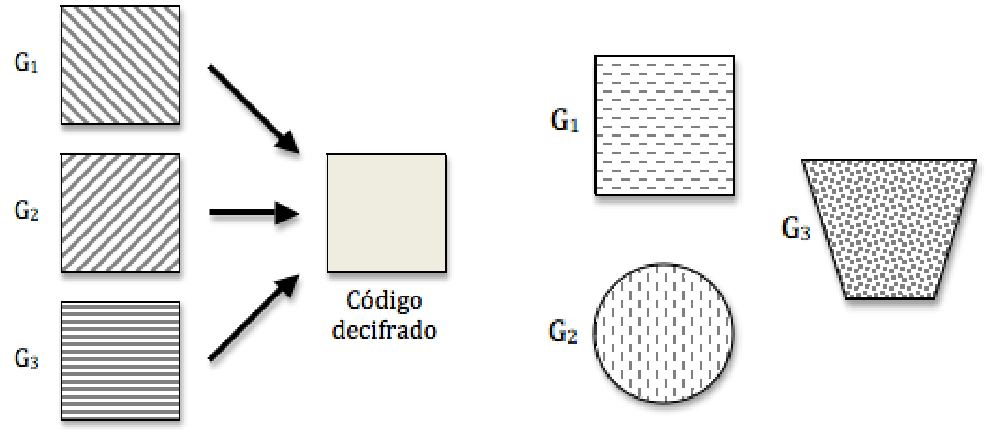
versão anterior, podendo inclusive, adicionar funcionalidades [25]. Figura 3.26(b) exibe a representação de diferentes gerações de um código metamórfico, na qual todo código do programa é transformado. O código malicioso W32/Evol é um exemplo de código metamórfico. Figura 3.27 exibe trechos de código extraídos dos pontos de entrada de três gerações deste código malicioso. Nota-se que da primeira geração para a segunda geração a instrução do endereço ‘401006’ foi substituída pelos códigos dos endereços de ‘1003acf’ até ‘1003ad8’. Em outro exemplo de código de segunda geração a instrução do endereço ‘40100f’ da primeira geração foi substituída pelas instruções dos endereços de ‘1002601’ até ‘100260a’.

3.3. Tamper-proofing

As técnicas de *tamper-proofing* visam a assegurar que um determinado programa executa como esperado, mesmo em situações de modificação ou monitoração. Estas técnicas, além de estarem atreladas à detecção violações de integridade, também possuem funcionalidades de resposta em situações de modificação ou monitoração.

Estas técnicas estão correlacionadas com as técnicas de ofuscação de código dado que para que um código seja modificado, o mesmo precisa ser entendido. As técnicas de ofuscação realizam transformações no código para que a engenharia reversa do mesmo seja mais complicada quanto ao tempo, custo ou poder computacional. Assim, pode-se dizer que a aplicação de ofuscação é uma etapa que garante maior resiliência ao método de *tamper-proofing*. O exemplo a seguir demonstra a importância da utilização de técnicas de ofuscação para dificultar violações a integridade em um programa.

Uma prática comum utilizada por desenvolvedores de software para atrair clientes consiste na distribuição de programas de avaliação com restrições na funcionalidade e/ou no tempo de uso. Programas comerciais frequentemente incluem uma verificação de



(a) Código polimórfico. (b) Código metamórfico.

Figura 3.26. Representação gráfica de diferentes gerações de códigos polimórficos e metamórficos.

tempo de uso, como o trecho de código da Figura 3.28.

Este código permite o usuário executar o programa até quinze vezes, após esta quantidade o programa terminaria sua execução assim que fosse verificada a condição exibida no código. Enquanto estas restrições resguardam o direito do desenvolvedor ao seu software, as mesmas motivam um adversário a modificar o software para remover as restrições. No exemplo acima, um adversário através de engenharia reversa poderia localizar o trecho de código representado e modificá-lo para que o mesmo possa ser executado a quantidade de vezes que ele achasse conveniente ou simplesmente nulificar a condição para que o programa executasse por tempo indeterminado.

As técnicas de *tamperproofing* também estão associadas as técnicas de marca d'água. Para elucidar esta questão, assuma que exista um sistema que um adversário nunca consiga modificar, assim, as técnicas de marca d'água poderiam ser as mais simples possíveis visto que um adversário não conseguiria remover/adulterar a marca d'água. Como este sistema não existe, temos que considerar técnicas para tornar as marcas d'água menos suscetíveis a remoção ou modificação.

Além de as técnicas de *tamper-proofing* estarem atreladas a modificação, qualquer ato de monitoração para extração de código ou dados de um programa também é considerado como uma forma de modificação. Um cenário comum para aplicação das técnicas *tamper-proofing* envolve sistemas de gerenciamento de direitos digitais (DRM) que possuem tipicamente dispositivos reprodutores com uma chave criptográfica embarcada e mídias criptografadas por esta chave criptográfica. Um adversário com acesso ao código do dispositivo reprodutor poderia modificá-lo para reproduzir mídias de outras regiões, assim como ter acesso a mídia em texto claro.

A modificação do código — por inserção e/ou por remoção — pode ser feita *offline*, ou seja, antes de executar o programa ou *online* (em tempo de execução) com a utilização de ferramentas externas como depuradores ou emuladores. Assumindo que o sistema possa ser violável fisicamente, três funções são atribuídas às técnicas de *tamper-proofing*: detecção de violações a integridade do código ou dos dados, verificação do

```
;primeira geração
  401000:       55                      push   ebp
  401001:       8b ec                   mov    ebp,esp
  401003:       83 ec 04                sub    esp,4
  401006:       8b 45 04                mov    eax,DWORD PTR [ebp+4]
  401009:       89 45 08                mov    DWORD PTR [ebp+8],eax
  40100c:       8b 45 00                mov    eax,DWORD PTR [ebp]
  40100f:       89 45 fc                mov    DWORD PTR [ebp-4],eax
  401012:       e8 aa 01 00 00          call   0x4011c1
;segunda geração
  1003ac9:       55                     push   ebp
  1003aca:       8b ec                  mov    ebp,esp
  1003acc:       83 ec 04               sub    esp,4
  1003acf:       56                     push   esi
  1003ad0:       89 ee                  mov    esi,ebp
  1003ad2:       83 c6 53               add    esi,83
  1003ad5:       8b 46 b1               mov    eax,DWORD PTR [esi-79]
  1003ad8:       5e                     pop    esi
  1003ad9:       89 45 08               mov    DWORD PTR [ebp+8],eax
  1003adc:       8b 45 00               mov    eax,DWORD PTR [ebp]
  1003adf:       89 45 fc               mov    DWORD PTR [ebp-4],eax
  1003ae2:       e8 73 02 00 00         call   0x1003d5a
;outra segunda geração
  10025f2:       55                     push   ebp
  10025f3:       8b ec                  mov    ebp,esp
  10025f5:       83 ec 04               sub    esp,4
  10025f8:       8b 45 04               mov    eax,DWORD PTR [ebp+4]
  10025fb:       89 45 08               mov    DWORD PTR [ebp+8],eax
  10025fe:       8b 45 00               mov    eax,DWORD PTR [ebp]
  1002601:       56                     push   esi
  1002602:       89 ee                  mov    esi,ebp
  1002604:       83 ee 1f               sub    esi,31
  1002607:       89 46 1b               mov    DWORD PTR [esi+27],eax
  100260a:       5e                     pop    esi
  100260b:       e8 e9 03 00 00         call   0x10029f9
```

Figura 3.27. Trechos de códigos de três gerações do código malicioso Win32/Evol.

```
if (numero_execucoes > 15) {
    printf ("Período de avaliação expirado");
    exit();
}
```

Figura 3.28. Trecho de código para verificar o número de execuções de um programa.

ambiente em que o código está sendo executado e tomada de ação, caso haja violações a integridade ou caso o ambiente de execução esteja hostilizado (programa executando em um sistema operacional corrompido ou sobre um emulador ou com um depurador anexado). Existem diversos métodos para monitorar se um ambiente está hostilizado, porém estes são específicos para cada ambiente. Uma abordagem ampla destes métodos podem ser encontrada em [44, 58].

Estratégias de *tamper-proofing* para detectar violações a integridade consistem na auto-verificação de trechos de código do programa ou na inspeção lógica do programa. A verificação pode ser classificada em estática em que a verificação é feita em tempo de carregamento do programa ou dinâmica em que a verificação é feita durante tempo de execução. A inspeção lógica pode ser feita nos dados do programa ou no fluxo de execução. Dado que um desenvolvedor tem conhecimento *a priori* dos possíveis estados que seu programa pode atingir, o desenvolvedor pode inserir rotinas para verificar se uma variável ou o resultado de uma função está em conformidade [8].

Entre as estratégias comuns para a ação (resposta) de um sistema de *tamper-proofing* temos: terminar a execução do programa, restaurar o programa substituindo os trechos de código adulterados por trechos originais, retornar resultados incorretos, degradar o desempenho da aplicação, contactar o desenvolver [52] ou até mesmo punir o sistema do adversário, por exemplo, apagando seus documentos pessoais [54]. Sejam quais forem a forma de verificação e a resposta a ser tomada, é importante que as mesmas estejam distantes em tempo (com execução não simultânea) e espaço (não próximas no código do programa) para aumentar a dificuldade de um adversário em subvertê-las.

3.3.1. Auto-verificação de código

Os métodos de detecção a violações a integridade são baseados na auto-verificação do código através do uso de *hash* (resumo criptográfico). O programa durante ou antes de sua execução calcula o *hash* de uma determinada região no segmento de código do programa e compara-o com um valor de *hash* esperado. Caso o resultado da comparação divirja, o programa pode acionar a resposta que lhe foi programada.

A técnica proposta por Chang e Atallah [27] explora uma metodologia distribuída através de uma rede de verificadores e respondedores, diferenciando-a de outras formas de um único ponto de verificação (facilmente subvertido). A criação desta rede permite proteção mútua entre os verificadores e respondedores, em que os verificadores além de verificar blocos de código do programa, também verificam verificadores e respondedores. Caso o resultado do verificador divirja do original, o respondedor tem a função de substituir o trecho de código adulterado por um trecho de código original.

Figura 3.29 exibe a estrutura de memória em que dois trechos de código, ‘C1’ e

‘C2’ estão protegidos por verificadores de *hash* e respondedores, que substituem o código adulterado pelo original caso haja modificação. Os trechos de código ‘C1’ e ‘C2’ encontram-se protegidos mutuamente por ‘G1’, ..., ‘G5’. O trecho de código ‘C1’ é corrigido por ‘G3’ antes de ser executado, e este é verificado por ‘G1’ e ‘G5’. A guarda ‘G5’ é verificada por ‘G2’ e reparada caso esteja modificada.

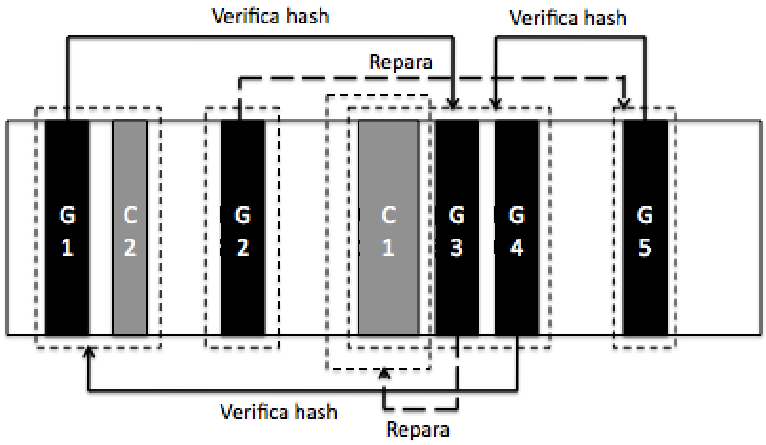


Figura 3.29. Estrutura de memória de um programa protegido.

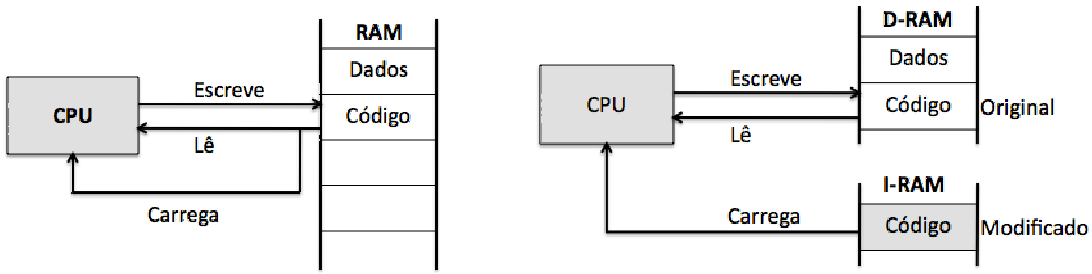
Uma técnica de ataque contra auto-verificadores de *hash* consiste em varrer o código a procura de instruções que comparam um cálculo de *hash* com um valor aleatório. Em programas que são distribuídos de forma distinta, um ataque de diferença poderia ser utilizado para localizar as exatas posições dos cálculos de *hash*. Uma estratégia para conter este problema consiste na duplicação de cada região que está sendo efetuado o cálculo de *hash* para que ao invés de comparar o resultado com um valor aleatório, comparar o resultado do cálculo de uma região com o cálculo da região duplicada. Uma desvantagem desta estratégia é a duplicação do código do programa.

As funções que calculam o *hash* também podem ser alvos de ataque de varredura por padrão, assim, é extremamente importante que estas funções estejam escondidas (o quanto possível) no código. Por exemplo, funções de criptografia tais como SHA-1 e MD5 apresentam assinaturas padrões que podem ser exploradas por um atacante. Horne *et al.* descreveu uma estratégia para dificultar estes ataques através da utilização de uma gama variada de funções de *hash*. Os autores utilizaram-se técnicas de otimização e ofuscação para gerar aproximadamente três milhões destas funções.

O trabalho de Horne *et al.* [24] envolve esconder os valores de *hash* para que estes sejam mais robustos diante de varredores de código que buscam instruções de comparação de um cálculo de *hash* com um valor aleatório. A idéia de seu trabalho consiste em diminuir o grau de visibilidade na instrução de comparação, fazendo com que as funções de cálculo de *hash* sempre retornem o valor zero, fazendo com que a comparação seja feita sobre um valor booleano (no caso 0), que é uma comparação típica na maioria dos programas. O algoritmo insere verificadores na forma “*if (hash(start_address, end_address)) respond();*” em que a resposta dada pela função “*respond()*” só executará se o cálculo do *hash* for diferente de zero em situações de modificação. Mais especificamente, o algoritmo faz uso de pontos para inserção de bits que fazem com que o cálculo do *hash*

seja nulo.

Os ataques aos métodos de auto-verificação de *hash*, como dito anteriormente, tipicamente envolvem análise para localização dos pontos de verificação para posterior modificação/anulação da verificação. Wurster *et al.* [50] desenvolveu um ataque aos métodos de verificação de *hash* baseado em modificações no ambiente, chamado de ataque de replicação de páginas. O ataque foca em uma deficiência dos algoritmos de auto-verificação de *hash* que assumem que instruções e dados devam compartilhar o mesmo espaço de endereçamento de memória (arquitetura von Neumann [1]) [43], ou seja, o código lido pelas rotinas de auto-verificação de *hash* é o mesmo que é carregado pelo processador para execução (vide Figura 3.30(a)). Contudo, um adversário pode violar esta suposição pela criação de uma arquitetura de memória virtual Harvard [2] em que memórias para instruções e dados são distintas (vide Figura 3.30(b)). Isso é feito através da modificação do sistema operacional para que este replique as páginas de memória das instruções tal que a leitura e o carregamento de instruções busquem valores de diferentes páginas de memória. Com isso, um adversário pode alterar o código na memória de instruções sem ser perceptível para as rotinas de verificação de *hash* que lêem a memória de dados.

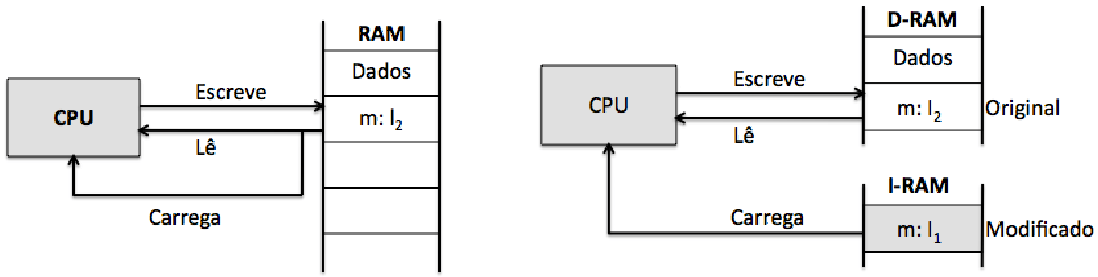


(a) Arquitetura von Neumann. (b) Arquitetura Harvard.

Figura 3.30. Ataque aos algoritmos de verificação de *hash*.

O trabalho de Giffin *et al.* [43], ao contrário do trabalho de Wurster *et al.* [50], apresenta uma técnica para detectar ataques de replicação de páginas. A técnica é baseada em códigos auto-modificáveis e envolve as seguintes etapas: sobreescrever uma instrução 'I_1' do endereço *m* com uma instrução 'I_2'; ler o valor contido no endereço *m*; executar a instrução do endereço *m*. Para que o ataque seja perceptível ao usuário é necessário que a instrução 'I_2' altere o fluxo de execução do programa.

Durante a execução do programa, se a instrução lida for 'I_2' e a execução do programa seguir o fluxo de execução de 'I_2' significa que o ambiente é regular ou que não está sobre ataque de replicação de páginas (vide Figura 3.31(a)). Caso contrário, estamos diante de um ataque de replicação de página em que a instrução lida é 'I_2' e a execução do programa não seguiu o fluxo de execução ditado pela instrução 'I_2', ou seja, a instrução carregada foi 'I_1' que significa que o ambiente está alterado ou sob ataque por replicação de páginas (vide Figura 3.31(b)).



(a) Arquitetura von Neumann. (b) Arquitetura Harvard.

Figura 3.31. Detecção de ataques de replicação de páginas.

3.3.2. Inspeção da lógica de execução

Os métodos de auto-verificação de *hash* possuem alguns desvantagens tais como a facilidade de detectar uma rotina de verificação dada a sua natureza atípica de operação, desde que a maioria dos programas não lêem seu próprio segmento de texto (código) e o fato de estes métodos não tratarem alterações no comportamento ou dados do programa. Por exemplo, um adversário pode temporariamente alterar o valor de um registrador que contém o resultado de retorno de uma função antes mesmo de a função retornar sem precisar alterar o código do programa. Métodos que inspecionam a corretude dos dados e do fluxo de controle visam a atender as deficiências dos métodos de auto-verificação de *hash*.

Chen *et al.* [31] definiu o conceito de *Oblivious Hashing* em que um adversário não tem a percepção que o código está sendo verificado. A idéia de Chen *et al.* é calcular o *hash* do traço de execução de um trecho de código, permitindo probabilisticamente e deterministicamente verificar o comportamento do programa em tempo de execução. Códigos que calculam o *hash* são inseridos no código fonte do programa que implicitamente calculam o *hash* no contexto de execução do código do programa. O objetivo é que os códigos para o cálculo do *hash* estejam misturados com o código original com a finalidade de dificultar a distinção destes por um adversário. Os *hashes* são basicamente calculados sobre valores de variáveis e no resultado de predicados de fluxo de controle. Outro trabalho que seguiu a idéia de *Oblivious Hashing* é o de Jacob *et al.* [56] que vincula a sobreposição de blocos básicos de instruções x86 com o cálculo de *hash*. Este cálculo é efetuado sem a necessidade de ler o código, e por isso, invulnerável a ataques de separação de memória.

Outra técnica de *tamper-proofing* consiste em remotamente detectar e responder a violações de integridade. Este cenário visa à proteção de um programa que executa em um local não confiável (lado do cliente) monitorado por comunicação com um programa que executa em um local confiável (lado do servidor). O servidor além de fornecer recursos aos pedidos do cliente, oferece mecanismos para verificar a integridade do programa e responder à violações. Em um sistema cliente-servidor em que o cliente espera recursos advindos do servidor, uma maneira fácil de responder a uma violação de integridade consiste na interrupção da comunicação. A tarefa de verificação não é básica quanto a tarefa de resposta pois o servidor não acessa diretamente o código do lado do cliente. O servidor poderia requisitar o *hash* de um determinado trecho de código através do canal de comunicação, porém nada garante que a resposta do cliente seria legítima. A resposta

pode ser forjada pela modificação do código cliente ou do ambiente de execução ou pela interceptação e substituição das mensagens de rede.

Soluções para verificar integridade de modo remoto são baseadas no compartilhamento de código entre o cliente e do servidor, levando em consideração que códigos mais críticos estejam no servidor. O balanceamento do compartilhamento reflete na carga computacional e na latência comunicação. Em situações que grande parte do código está contida no servidor significaria uma carga computacional alta no servidor e uma latência de comunicação alta para o cliente. Caso contrário, ou seja, a maior parte de código contida no cliente refletiria uma latência de comunicação baixa para o cliente e uma carga computacional baixa para o servidor. Contudo, nesta situação é difícil o servidor garantir que o código do cliente não foi modificado. As técnicas para *tamper-proofing* remoto são niveladas em um balanceamento intermediário entre carga computacional e latência de comunicação.

A técnica de Zhang e Gupta [38] automaticamente divide um programa em duas partes: uma aberta que executará no cliente e outra fechada que executará no servidor, podendo assim, proteger partes sensíveis do código. Esta técnica trata problemas como latência e largura de banda mantendo estruturas de dados grandes no lado do cliente e restringindo a execução do servidor e do cliente em uma rede local.

Outra técnica consiste na requisição de um valor de *hash* pelo servidor de um determinado trecho do código em execução no cliente, com o adicional de avaliar a legitimidade da resposta do cliente pela medição de determinadas propriedades do ambiente do cliente, como por exemplo, o tempo gasto para o cliente retornar o *hash* [51]. Esta técnica é baseada em algumas suposições para garantir que o código em execução no cliente não foi modificado. Estas suposições incluem: conhecimento da configuração de hardware do cliente, latência entre cliente e servidor, restringir a comunicação do cliente para que durante a verificação o cliente só mantenha comunicação com o servidor e o código do cliente tem que executar independentemente de outros códigos presentes no cliente.

A função para cálculo de *hash* no cliente é implementada de tal forma que se um adversário modificar o código, o mesmo retornará um valor de *hash* incorreto ou levará um tempo notável para efetuar o cálculo. A notabilidade de tempo justifica as suposições consideradas anteriormente, pois caso o servidor desconhecesse o poder computacional do cliente ou a velocidade de comunicação, o servidor não poderia estimar o tempo considerado “legítimo” para calcular o *hash*.

Outra idéia é de utilizar ofuscação dinâmica como um método de *tamper-proofing* remoto, fazendo com que o servidor envie versões modificadas (ofuscadas) para o cliente de modo que seja difícil para um adversário analisar o código. A técnica mantém uma representação do código do cliente tanto no lado do cliente como no lado do servidor. Este código é dividido em blocos de tal forma que a execução de cada bloco é feito através de requisições ao servidor. Paralelamente, o servidor contém um mecanismo de mutação que modifica os blocos do cliente e compartilha os blocos modificados com o cliente. Quando o cliente requisita ao servidor um bloco, este recebe um novo bloco do servidor caso o bloco esteja modificado. A técnica também pode enviar blocos para o cliente assim que modificados. Este processo de atender a requisições e enviar blocos é uma tarefa responsável por um processo escalonador executando no lado do servidor.

A robustez desta técnica diante de violações de integridade é relacionada com a taxa em que o servidor gera e envia blocos modificados para o cliente e a capacidade que um adversário tem de analisar um código que é constantemente alterado. É interessante que as transformações gerem códigos ofuscados diferentes continuamente e que o tempo de análise de um código por um adversário seja inferior a criação e envio dos blocos pelo servidor. Uma possibilidade de ataque consiste em ignorar os blocos enviados pelo servidor e copiar um estado fixo do programa contido na memória para análise fora do tempo de execução.

3.4. Marca d'água

Dizemos que um código C possui a propriedade σ como *marca d'água* se a propriedade σ pode ser verificada em C e em qualquer código C' obtido de C através de uma "pequena modificação". O conceito de "pequena modificação" pode ser definido de maneira rigorosa — por exemplo, o número de bits que devem ser alterados em C para se obter C' — de todo modo, uma modificação sobre C deveria ser considerada pequena se pudesse ser facilmente conduzida por um adversário. A propriedade σ pode ser definida tanto de maneira estática — por exemplo, um padrão de bits encontrado ao longo do código binário — quanto de maneira dinâmica — padrões somente verificáveis em tempo de execução.

A marca d'água fornece subsídios para se identificar o uso indevido de um software. Neste sentido, a marca d'água pode ser interpretada como o último estágio na defesa de um software: uma vez que a ofuscação falha em seu objetivo de **dificultar a análise** e as ferramentas de tamper-proofing falham em seu objetivo de **impedir a execução indevida**, ainda assim, é possível identificar que um determinado software, em execução em um dispositivo, está sendo utilizado indevidamente. Embora a marca d'água não impeça o uso indevido de um software, ela, ao menos, fornece meios de que outras ações — por exemplo, ações legais — sejam tomadas.

Exemplo 1. Um software possui uma área que contém o nome do seu desenvolvedor. O nome do desenvolvedor poderia ser inserido em uma área do programa jamais executada, por exemplo, depois de um salto incondicional. A verificação da marca d'água seria feita através da busca, no software, do nome do desenvolvedor, cuja presença indicaria sua autoria. Uma forma de remoção da marca d'água por um atacante seria simplesmente remover a área do código que indica sua autoria.

Exemplo 2. A técnica apresentada no Exemplo 1 é bastante singela e passível de ataque de remoção bastante rudimentar. Uma maneira de se dificultar o ataque de remoção descrito anteriormente seria inserir a informação de autoria em uma quantidade maior de posições do código. Além disso, para dificultar a "remoção automática", em que o atacante simplesmente busca e remove um padrão fixo de bits, a identificação de autoria do software poderia ser inserida através de identificadores variáveis — por exemplo, dependentes de posição de memória ou de instruções próximas. A remoção da marca d'água, neste caso, demandaria a localização e remoção de todas as instruções não executadas no código. Técnicas de ofuscação podem ser usadas para dificultar ainda mais a remoção da marca d'água.

Vimos, nos simples Exemplos 1 e 2, que duas operações estão associadas a uma técnica de marca d'água: a operação de *inserção* da marca d'água e a operação da *extra-*

ção da marca d'água para posterior verificação. A operação de inserção é a técnica de transformação de software que irá receber um software qualquer e dotá-lo de uma propriedade σ, posteriormente verificável. A operação de extração é aquela que irá tomar um software qualquer e nele buscar a existência da propriedade σ que irá evidenciar o uso devido ou indevido daquele software.

Os extratores podem, ainda, ser classificados como *blind* ou *informed*. Um detector é classificado como *informed* se, para extrair a marca d'água, é neessário estar de posse da "obra original", ou seja, do objeto sobre o qual foi inserido a marca d'água. Já um extrator *blind* é capaz de extrair a marca d'água de qualquer objeto, mesmo que a versão "original deste objeto não esteja disponível". Um exemplo de uso de extratores *informed* é o caso do rastreamento de transações em que o proprietário da obra original busca identificar quem distribuiu ilegalmente a obra — como o usuário do extrator é o proprietário da obra, então esta obra estará disponível, *à priori*, durante o processo de extração. Em outros casos, a obra não estará disponível, como é o caso, por exemplo, da extração da marca d'água de um vídeo para fins de verificação de autenticidade — neste caso, o vídeo é uma obra desconhecida, de forma que o extrator deverá ser capaz de obter a marca d'água sem o conhecimento da obra original. Na prática, extratores *informed* apresentam melhores resultados quando comparados a extratores *blinded* [63]. Isso porque a obra cuja marca dágua está sendo verificada pode sofrer uma série de alterações antes do processo de extração — tanto alterações maliciosas, visando à própria corrupção da marca d'água, por exemplo, quanto alterações lícitas, tal como a própria inclusão da marca d'água. Assim, a obra original torna-se uma boa referência para o algoritmo de extração da marca d'água.

Um sistema de marca d'água é dito *privado* quando apenas usuários autorizados são capazes de extrair a marca d'água; caso contrário, o sistema de marca d'água é *público*.

A *transparência* de uma marca d'água está associada ao grau de similaridade entre a obra original e a obra com marca d'água. Quanto menos perceptível for a marca d'água, mais transparente ela será considerada. A *capacidade* de um sistema de marca d'água diz respeito à quantidade de informação que este sistema é capaz de inserir na obra. A *robustez* de uma marca d'água é a sua resistência a manipulações da obra. O grau de robustez de uma marca d'água pode ser classificado como *seguro*, *robusto*, *semi-frágil* ou *frágil*, de acordo com o conjunto de manipulações da obra ao qual a marca d'água é capaz de resistir [63]. Na prática, observa-se um compromisso entre transparência, capacidade e robustez em sistemas de marca d'água: quanto mais informação a obra carrega na forma de marca d'água e quanto maior a sua robustez, menor será a transparência da marca d'água.

A Figura 3.32 ilustra um sistema de marca d'água genérico.

Antes de examinarmos técnicas mais sofisticadas para codificar uma marca d'água no software, vamos estabelecer as diferenças entre marca d'água e *fingerprinting* e explorar alguns possíveis ataques à marca d'água. Em um cenário de marca d'água todas as cópias do mesmo software são distribuídas de forma idêntica, ou seja, as marcas são codificadas igualmente. Em um cenário de *fingerprinting* toda cópia distribuída contém uma marca diferente associada ao cliente, possibilitando assim, um rastreamento do cliente

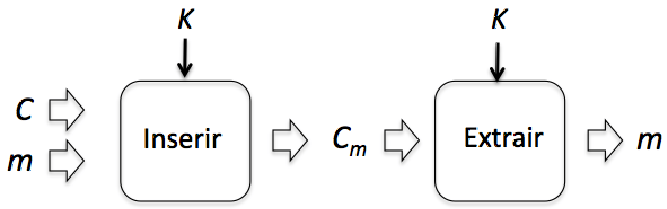


Figura 3.32. Sistema de *watermarking* em software.

diante de posse de uma cópia ilegal. Um problema deste cenário é a possibilidade de ataques de diferença na qual um adversário adquire cópias distintas de um mesmo software para identificar a localização da marca, facilitando sua remoção ou falsificação.

Marcas d'água são também passíveis de outros tipos de ataque. Ataque de distorção consiste na aplicação de técnicas de transformação de código que preservam a semântica do código, como otimização, portabilidade, compressão e ou ofuscação para para impedir que o processo de extração de um sistema de *watermarking* consiga reaver a marca. Ataque de adição envolve a adição de uma outra marca por um adversário com o objetivo de alegar em uma ação judicial que o software é de sua autoria. Ataques à marca d'água são considerados bem sucedidos se o processo de extração não consegue legitimar a marca original e a funcionalidade do software adulterado é a mesma do software original.

As técnicas de marca d'água de software podem ser classificadas em estáticas ou dinâmicas. Nas estáticas, as marcas são armazenadas no próprio código executável e o processo de inserção e extração é feito sem a necessidade do programa ser executado ou interpretado. Nas dinâmicas, o programa ou parte dele precisa ser executado ou interpretado para que a marca d'água seja extraída. O processo de extração de uma marca d'água dinâmica utiliza-se de uma sequência especial de entrada para extrair a marca de um determinado estado do programa. Estas marcas dinâmicas possuem a vantagem de serem mais resilientes diante de ataques de falsificação, remoção ou adição, pois as marcas estão associadas a um estado dinâmico do programa que por conseguinte, não são estaticamente visíveis para um adversário, contudo, são ainda suscetíveis a ataques de distorção.

3.4.1. Marcas d'água estáticas

A técnica de Peter Wayner [29] de inserir uma mensagem secreta em uma lista ordenada pode ser utilizada para inserir uma marca em qualquer lista de uma linguagem de programação que possa ser reordenada. A idéia consiste na codificação de um inteiro (marca d' água) através da reordenação das declarações de um programa ou atribuições desde que estas não sejam dependentes uma das outras. Por exemplo, assuma que a marca é '213', Figura 3.33 mostra como a marca é inserida por reordenação.

Neste caso, o extrator teria que ter acesso ao código original e ao programa com a marca d'água contida para extrair a marca d'água, ou seja, um extrator *informed*.

A reordenação também pode ser feita nos blocos básicos de um grafo de fluxo de controle [47]. A idéia é similar a técnica anterior ao ponto que ao invés de reordenar declarações de um programa reordena seus blocos básicos através de instruções incondi-

```
/* Código Original */
1: a = 5;
2: b = 8;
3: c = 4;

/* Código Reordenado*/
2: b = 8;
1: a = 5;
3: c = 4;
```

Figura 3.33. Reordenação das atribuições para inserção de marca d'água.

cionais ou condicionais através de predicados opacos.

Outra técnica consiste na renumeração dos registradores de um código binário para inserir a marca d'água. Vamos utilizar a mesma marca d'água da técnica anterior, ou seja, '213'. Figura 3.34 ilustra esta técnica.

```
/* Código Original */
1: r1 = 3;
2: r2 = r1 + 3;
3: r2 = r2 * 4;
4: r3 = r2 + r1;

/* Código Renumerado */
1: r2 = 3;
2: r1 = r2 + 3;
3: r1 = r1 * 4;
4: r3 = r1 + r2;
```

Figura 3.34. Renumeração dos registradores para inserção de marca d'água.

As técnicas até agora examinadas são consideradas frágeis, pois basta um adversário reordenar declarações ou blocos básicos ou renumerar registradores aleatoriamente para que as marcas não sejam mais válidas.

Moskowitz e Cooperman [11] desenvolveram uma técnica de marca d'água de software baseada no uso de imagens, vídeos ou áudio embarcados no software. Caso o software não possua um destes meios, é feita a inserção de um no código do programa. A idéia é ao invés de criar métodos para inserir e extrair marca d'água de software, utilizar algoritmos de inserção e extração já existentes para imagens, vídeos e áudio em software. A técnica embarca trechos de código essenciais para a execução do programa nestes meios, de tal forma que estes trechos de código só são executados diante de um processo de extração durante a execução. Isso caracteriza o método como sendo também um método de *tamper-proofing*, pois caso a marca seja distorcida de algum modo, não será possível extrair o código e o programa não apresentará a mesma funcionalidade. A distorção de uma imagem, por exemplo, pode ser feita através da ferramenta Stirmark [66]. Esta possui uma coleção de transformações que fazem grande parte dos algoritmos de marca d'água de imagem falharem no processo de extração.

```
/* Código Original */
...
    if ( x < y ) jmp b
a: bloco_de_instrução_a
b: bloco_de_instrução_b
...

/* Código Alterado */
...
    if ( x >= y ) jmp a
b: bloco_de_instrução_b
a: bloco_de_instrução_a
...
```

Figura 3.35. Inversão da condição em saltos condicionais.

Monden *et al.* [15] apresentou um algoritmo de marca d'água em que a marca é codificada através da modificação dos *opcodes* de uma função espúria. Esta função é adicionada ao programa e sua execução é disfarçada através de predicados opacos, fazendo com que a função nunca seja executada. Um ataque a este algoritmo basicamente envolve a construção do grafo de chamadas do programa e a utilização de *profiling* para efetuar a contagem de quantas vezes as funções estão sendo invocadas. Como a função espúria que contem a marca nunca é executada, um adversário trivialmente pode remover a função, apagando consequentemente a marca d'água do programa.

Venkatesan *et al.* [23] implementou um algoritmo de marca d'água que adiciona arestas redundantes no grafo de de fluxo de controle do programa, fazendo com que um adversário não consiga separar ou contar quais arestas são partes do programa e quais são pertencentes a marca d'água, sendo assim, mais resiliente à ataques de remoção auxiliado por *profiling*. A inserção é baseada na codificação de uma marca em um grafo que seja compatível com um grafo de fluxo de controle, ou seja, que seja redutível e que possua grau de conectividade 'um' ou 'dois' visto que as estruturas de controle ('if', 'for' e 'while') de um programa apresentam estas características. A estrutura 'switch' apesar de apresentar um grau de conectividade maior não é uma estrutura comumente utilizada na maioria dos programas. Uma possibilidade de ataque a esta técnica é alterar os saltos condicionais invertendo sua condição. Os códigos da Figura 3.35 ilustram esta inversão em que o código alterado inverte a condição do salto do código original. Contudo, a técnica propõe o uso de um grafo que seja resistente a estas inversões [40].

Outro algoritmo de marca d'água foi proposto por Cousot e Cousot [42] na qual a idéia é inserir uma marca d'água que possa ser extraída utilizando análise estática. A técnica proposta é baseada na propagação de constantes de um análise de fluxo de dados. Figura 3.36 ilustra como é o processo de inserção de uma marca d'água '2' na rotina maior. Para extração da marca d'água temos uma chave secreta '3' na qual durante a análise de propagação de constantes, os valores de 'w' dentro e fora do laço são constantes e resultam na marca d'água '2' se aplicados com o módulo da chave '3', *i.e.*, '11 mod 3 = 2' e '14 mod 3 = 2'. Os autores propõem o uso de polinômios no lugar de constantes para que a localização da marca d'água seja mais complicada para um adversário.

```
int maior(int x[], int size) {
     int maior = x[0];
     int i = 1;
     int w = 11;
     while ( i < size) {
          if ( x[i] > maior )
               maior = x[i];
          w = 14;
          i++;
     }
     return maior;
}
```

Figura 3.36. Marca d'água '2' contida na rotina maior.

Os algoritmos abordados até o momento basicamente fazem uso ou da codificação de um identificador como permutação do código original, ou da inserção de um novo código não funcional. Estes métodos são passíveis de ataque pelas mesmas transformações aplicadas para inserção da marca d'água, ou seja, um adversário pode aplicar transformações como reordenação e inserção para dificultar a extração da marca d'água. A seguir, examinaremos um algoritmo que extrai a marca da d'água através da execução ou interpretação do programa.

3.4.2. Marcas d'água dinâmicas

Algoritmos de marcas d'água dinâmicas são baseados na inserção da marca d'água em um estado do programa. A extração da marca d'água envolve a saída do programa, ou a estrutura de dados, ou um código dinamicamente gerado pelo programa. Este procedimento pode ser feito por exemplo, examinando a pilha, os *threads* ou os registradores do programa. Como um adversário não tem conhecimento de como o processo de extração é feito, marcas d'águas dinâmicas acabam sendo mais resilientes diante de ataques de distorção, falsificação ou remoção.

Collberg *et al.* [41] implementou um algoritmo para codificar uma marca d'água nos saltos do programa. A idéia envolve a adição de saltos condicionais no programa que diante de uma determinada entrada codifica o bit '1' caso a condição do salto for satisfeita ou codifica o bit '0' caso contrário. Figura 3.37 ilustra este algoritmo que diante de um vetor maior que '1' embarca a marca d'água '10010_2' nos saltos dos endereços da linha '6' até a linha '10'. É claro que que está técnica de inserção é facilmente subvertida através da inversão dos saltos condicionais (vide Figura 3.35).

Este trabalho propõe uma maneira para conter estes ataques de inversão de saltos. A idéia consiste na inserção de uma sequência de predicados e saltos em posições do código que são executadas várias vezes diante de uma determinada entrada. Na primeira execução destes códigos são identificadas as direções dos saltos para que nas execuções subsequentes seja gerada a codificação da marca d'água. Este algoritmo insere predicados que são dependentes das variáveis do programa, e a codificação da marca d'água é feita analisando o traço de execução do programa. Esta análise é feita para determinar predicados que são verdadeiros na primeira execução e falsos nas execuções posteriores.

```
/* Rotina Original */

int maior(int x[], int size) {
1:     int maior = x[0];
2:     int i = 1;
3:     while ( i < size) {
4:          if ( x[i] > maior )
5:               maior = x[i];
6:          i++;
     }
7:     return maior;
}

/* Rotina com marca d'água */

int maior(int x[], int size) {
1:     int maior = x[0];
2:     int i = 1;
3:     int u;
4:     int t = 0;
5:     while ( i < size) {
6:           if (t = 0) u++;
7:           if (t != 0) u++;
8:           if (t != 0) u++;
9:           if (t == 0) u++;
10:          if (t != 0) u++;
11:          if ( x[i] > maior )
12:               maior = x[i];
13:         i++;
     }
14:     return maior;
}
```

Figura 3.37. Inserção da marca d'água '10010_2' na rotina maior.

Figura 3.38 apresenta um código que extrai a marca d'água '10010_2' dado o vetor de entrada '$x = [0105]$'. Observe que dado este vetor, os códigos da linha '6' até a linha '11' entram em execução por duas vezes durante a execução desta rotina. Na primeira execução os valores de 'i = x[i] = maior = 1' e, portanto, todos os predicados são satisfeitos. Na segunda execução, o valor de 'i = 3' e 'x[i] = maior = 5', o que satisfaz os predicados das linhas '8', '9' e '11'. O bit '1' da marca d'água é codificado quando o predicado da segunda execução difere do predicado da primeira execução e a do bit '0' caso contrário. Através de uma análise de fluxo de dados, um adversário pode remover estas condições uma vez que a variável 'u' não altera o comportamento do programa. Um disfarce possível é atribuir 'u' a uma variável utilizada no programa, por exemplo, $if\ (P^f)\ x[i] = u$.

Collberg *et al.* [41] também propõe uma maneira de inserir uma marca d'água baseada na ofuscação de saltos incondicionais (vide Figura 3.14). A idéia é mapear cada bit da marca d'água em um salto incondicional '$a_i \mapsto b_i$' na qual $1 \leq i \leq n$ representa o

```
/* Rotina com marca d'água */

int maior(int x[], int size) {
1:     int maior = x[0];
2:     int i = 1;
3:     int u;
4:     while ( i < size) {
5:          if ( x[i] > maior ) {
6:               maior = x[i];
7:               if (i == x[i]) u++;
8:               if (x[i] == maior) u++;
9:               if (x[i] == maior) u++;
10:              if (i == x[i]) u++;
11:              if (x[i] == maior) u++; }
12:         i++;
     }
7:     return maior;
}
```

Figura 3.38. Codificação da marca d'água '10010_2' na rotina maior.

intervalo de endereços do programa. No exemplo da Figura 3.39, o bit '1' é codificado nos saltos para frente (*i.e.*, $a_i < b_i$) e o bit '0' nos saltos para trás (*i.e.*, $a_i > b_i$). Um salto incondicional $l_{comeco} \mapsto l_{fim}$ é incorporado em uma sequência de saltos incondicionais ofuscados a_0, a_1, a_2 e a_3 inseridos no programa. Este ofuscamento substitue um salto incondicional por uma chamada de função e manipula o endereço de retorno desta função para ser o endereço destino do salto incondicional. Este método também insere um salto incondicional antes das chamadas de função a_1, a_2 e a_3, pois estas não podem ser executadas exceto se advindas da execução de a_0. Estas marcas codificadas no próprio código executável são mais robustas contra ataques de adição e distorção uma vez que estes saltos ofuscados manipulam endereços absolutos e, por isso, qualquer alteração no endereçamento resulta em um programa não funcional.

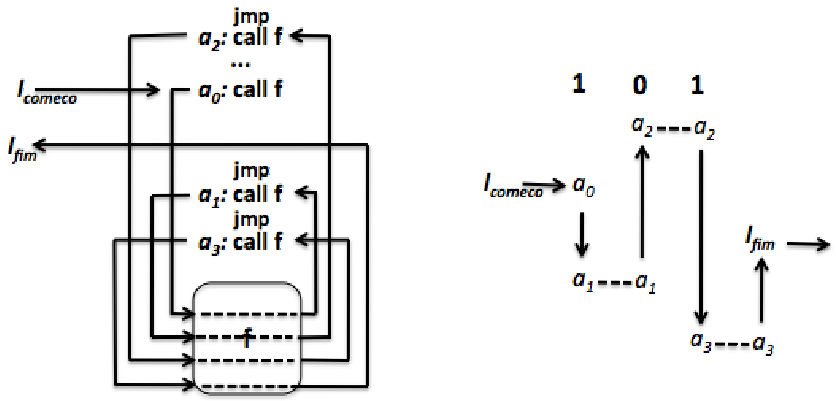


Figura 3.39. Codificação da marca d'água '101_2' através de saltos incondicionais ofuscados.

3.4.3. Marca d'água *versus* esteganografia

O leitor iniciado nas técnicas da esteganografia poderá ter se perguntado qual seria a diferença desta para a marca d'água. De uma maneira bastante concisa, uma técnica esteganográfica permite que se insira uma informação em um objeto de tal maneira que se torna muito difícil "detectar" a existência daquela informação no objeto. Não é objetivo primário da esteganografia que a informação embarcada em um objeto seja altamente resiliente a manipulações do objeto. Já a marca d'água busca associar uma informação a um objeto de tal maneira que seja "muito difícil" remover a informação daquele objeto, não se preocupando, primariamente, se a informação é ou não detectável.

A confusão observada algumas vezes na literatura entre os dois conceitos advém do fato de que as técnicas utilizadas tanto para esteganografia quanto para marca d'água atuam de maneira a "modificar" o objeto responsável pelo transporte da informação, sem que este objeto perca, no entanto, a sua funcionalidade inicial. Dessa forma, é possível o uso de ferramentas esteganográficas "puras" para a criação de marcas d'água. Deve-se ter claro, no entanto, que, uma vez descoberta, a marca d'água poderá ser facilmente removível. Além disso, observa-se que boa parte das ferramentas de marca d'água descritas na literatura inserem informação em um objeto efetuando sobre ele mudanças imperceptíveis, o que torna tais ferramentas passíveis de uso com finalidade esteganográfica.

Descrevemos um exemplo bastante concreto que contrapõe esteganografia e marca d'água. Suponha que se deseja transmitir uma determinada informação através de sua escrita sobre a superfície de um equipamento — por exemplo, um celular. As possíveis maneiras de se inserir a informação distinguem-se, entre outros, quanto ao tipo de tinta utilizado na escrita. Suponha que se utiliza uma tinta visível apenas sob radiação ultravioleta. Neste caso, tem-se um exemplo de esteganografia, já que a mensagem será invisível e "indetectável" sob iluminação convencional. Suponha, por outro lado, que se utiliza uma tinta de alta aderência — ou seja, de difícil remoção. Neste caso, tem-se um exemplo de marca dágua. O primeiro exemplo — da tinta invisível — seria usado, tipicamente, para a transmissão de uma informação sigilosa. No entanto, dado que a informação é de difícil detecção, o método poderia ser usado para inserir uma informação associada ao próprio equipamento, o que seria, em geral, objeto de uma marca d'água.

No mundo digital, técnicas esteganográficas podem ser aplicadas em áudio, imagem, texto e software. No contexto de áudio, uma técnica de marca d'água consiste em adicionar ecos nos bits menos significantes de som que são pouco perceptíveis para a audição humana. No contexto de imagem, pode-se aleatoriamente escolher dois bits da imagem e incrementar o brilho do primeiro *bit* em uma pequena quantidade e decrementar o segundo *bit* pela mesma quantidade. No contexto de textos, técnicas de *watermarking* envolvem alteração do espaçamento entre parágrafos ou palavras, assim como inserção de palavras sinônimas que codifiquem um padrão de bits. No contexto de software, pode-se inserir bits de informação em regiões do programa que nunca são executadas — por exemplo, após um salto incondicional. Em todos os casos a esteganografia será mais bem sucedida quanto menos perceptível for a modificação das características do objeto onde se insere a informação.

3.5. Conclusões

Todos meios de proteção, seja eles em software ou em hardware, são passíveis de ataques e podem ser subvertidos. Mecanismos baseados em hardware oferecem um nível de proteção mais elevado, porém apresentam desvantagens quanto ao custo, desempenho inferior e estresse para o usuário final em caso de extravio, atualização ou defeito. As técnicas de transformação de código degradam relativamente menos o desempenho e apresentam um custo inferior de desenvolvimento se comparado com os métodos de proteção baseado em hardware. Um outro ponto diz respeito ao fato que os hardwares possuirem um único ponto de falha, por exemplo, a proteção de uma chave em hardware, uma vez esta descoberta todo o sistema pode ser corrompido [48, 60]. As técnicas de transformação de código podem adicionar diferentes camadas de proteção dando maior robustez caso um adversário consiga burlar um nível de proteção. Porém nada impede que proteções de software e hardware coexistam. O balanceamento entre o nível de proteção com o *overhead* em desempenho e custo é uma decisão que cabe ao desenvolvedor do produto/programa.

Referências

[1] Neumann, J. von. . “First draft of a report on the EDVAC”,1945.

[2] Aiken, H.H. . “Proposed automatic calculating machine”. Unpublished manuscript, November, 1937. Also appeared in IEEE Spectrum, 1(8):62–69, Aug. 1964.

[3] Sharir, M.; Pnueli, A. . “Two approaches to interprocedural data flow analysis”. Program Flow Analysis: theory and applications. Englewood Cliffs: Prentice-Hall, 1981.

[4] Cohen F. . “Computer viruses—theory and experiments". In: IFIP-TC11, Computers and Security, pages 22–35, 1987.

[5] Cohen F. . “Current trends in computer viruses". In: International Symposium on Information Security, 1991.

[6] Cohen F. . “Operating system protection through program evolution". Computer Security, 12(6):565–584, 1993.

[7] Cohen F. . “A short course on computer viruses"(2nd ed.). John Wiley & Sons, Inc., New York, 1994.

[8] Blum M., Kannan S. . “Designing programs that check their work”. Journal of the ACM, 42(1):269–291, January 1995.

[9] Aucsmith, D. . “Tamper resistant software: An implementation”. In: Ross J. Anderson, editor, Information Hiding, First International Workshop, pp. 317–333, Cambridge, U.K., May 1996. Lecture Notes in Computer Science, Vol. 1174.

[10] Vliet, H. P. V. . “Crema — The Java obfuscator”. http://web.inter.nl.net/users/H.P.van.Vliet/mocha.html, January 1996.

[11] Moskowitz, S. A., Cooperman, Marc. . “Method for stega-cipher protection of computer code", US Patent 5,745,569, January 1996. Assignee: The Dice Company.

[12] Davidson, R. L., Myhrvold, N. . “Method and system for generating and auditing a signature for a computer program", US Patent 5,559,884, September 1996. Assignee: Microsoft Corporation.

[13] Majumdar, A., Thomborson, C. D., Drape, S. . “A Survey of Control-Flow Obfuscations”. In: Proceedings of the International Conference on Information Systems Security, 2006, pp. 353–356.

[14] Collberg, C.; Thomborson, C.; Low, D. . “A taxonomy of obfuscating transformations". Technical Report 148, Department of Computer Science, The University of Auckland, New Zealand, July 1997.

[15] Monden, A.; Iida, H.; Matsumoto, K.; Torii, K.; Ichisugi, Y. . “Watermarking method for computer programs”. In: Proc. of theSymposium on Cryptography and Information Security, 1998.

[16] Collberg, C.; Thomborson, C; Low, D. . “Manufacturing cheap, resilient, and stealthy opaque constructs”. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL’98, San Diego, January 1998.

[17] Collberg, C.; Thomborson, C; Low, D. . “Breaking abstractions and unstructuring data structures”. In: Proc. 1998 IEEE International Conference on Computer Languages, pages 28–38.

[18] Collberg, C.,Thomborson, C. . “Software watermarking: Models and dynamic embeddings", In: Principles of Programming Languages 1999, POPL’99, San Antonio, TX, January 1999.

[19] Collberg C.; Thomborson C. . “Watermarking, tamper-proofing, and obfuscation — tools for software protection”. Technical Report TR00-03, The Department of Computer Science, University of Arizona, February 2000.

[20] Lie, D.; Thekkath, C.; Mitchell, M.; Lincoln, P.; Boneh, D.; Mitchell, J.; Horowitz, M. . “Architectural support for copy and tamper resistant software”. In: Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), pp. 168–177, November 2000.

[21] Wang, C.; Hill, J.; Knight, J.; Davidson, J. .“Software tamper resistance: Obstructing static analysis of programs.” Technical Report CS-2000-12, 2000.

[22] Cho, W.; Lee, I.; Park, S. . “Against intelligent tampering: Software tamper resistance by extended control flow obfuscation”. In: Proc. World Multiconference on Systems, Cybernetics, and Informatics. International Institute of Informatics and Systematics, 2001.

[23] Venkatesan, R. , Vazirani, V., Sinha S. . “A graph theoretic approach to software watermarking". In: 4th International Information Hiding Workshop, Pittsburgh, PA, April 2001.

[24] Horne, B.; Matheson, L.; Sheehan, C.; Tarjan, R. E. . “Dynamic self-checking techniques for improved tamper resistance”. In: Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop DRM 2001, Philadelphia, November 2001. Springer-Verlag, LNCS 2320.

[25] Ször, P.; Ferrie, P. “Hunting for metamorphic”. In: Proc. of the 11th Virus Bulletin Conference, Prague, Czech Republic, pp. 123–144, 2001.

[26] Wang, C.; Hill, J.; Knight, J.; Davidson, J. . “Protection of software-based survivability mechanisms”. In: Proc. International Conference of Dependable Systems and Networks, July 2001.

[27] Chang, H.; Atallah, Mikhail J. . “Protecting Software Code by Guards”. In: ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management, 2002, pp 160–175.

[28] Collberg C.; Thomborson C. . “Watermarking, tamper-proofing, and obfuscation — tools for software protection”. In: IEEE Transactions on Software Engineering, New York, v. 28, n. 8, p. 735–746, 2002.

[29] Wayner, P. .“Disappearing Cryptography: Information Hiding: Steganography and Watermarking (2nd Edition)”. Morgan kaufmann Publishers Inc., San Francisco, CA, 2002.

[30] Wroblewski, G. . “General Method of Program Code Obfuscation”. PhD thesis, Wroclaw University of technology, Institute of Engineering Cybernetics, 2002.

[31] Chen, Y.; Venkatesan, R.; Cary, M.; Pang, R.; and Sinha, S.; and Jakubowski, M. H. . “Oblivious Hashing: A Stealthy Software Integrity Verification Primitive”. In: 5th International Workshop on Information Hiding, 2003, pp. 400–414.

[32] Christian Sven Collberg, Clark David Thomborson, and Douglas Wai Kok Low. . “Obfuscation techniques for enhancing software security”. U.S. Patent 6668325, December 2003.

[33] Kanzaki, Y.; Monden, A.; Nakamura, M; Matsumoto K. . “Exploiting self-modification mechanism for program protection”. In: Proc. of the 27th Annual International Conference on Computer Software and Applications, pp 170, Washington, USA, 2003.

[34] Horne, W. G.; Matheson, L. R.; Sheehan, C.; Tarjan, R. E. . “Software self-checking systems and methods”. U.S. Application 20030023856, January 2003. Assigned to InterTrust Technologies Corporation.

[35] Lakhotia, A.; Singh, P. K. . “Challenges in getting ‘formal’ with viruses”. Virus Bulletin, pp. 15–19, September 2003.

[36] Linn, C.; Debray, S. . “Obfuscation of executable code to improve resistance to static disassembly”. In: Proceedings of the 10th Computer and Communications Security (CCS), 2003, pp. 290–299.

[37] Ogiso, T.; Sakabe, Y.; Soshi, M.; Miyaji, A. . “Software obfuscation on a theoretical basis and its implementation”. In: IEEE Trans. Fundamentals, E86-A(1), January 2003.

[38] Zhang, X.; Gupta, R. . “Hiding program slices for software security”. In: Proc. of the International Symposium on Code Generation and Optimization, 2003, pp. 325–336, Washington, DC, 2003. IEEE.

[39] Christodorescu, M.; Somesh, J. . “Testing Malware Detectors”. In: Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA’04), July 11-14, 2004, Boston, Massachusetts, USA.

[40] Collberg, C.; Huntwork, A.; Carter, E.; Townsend G. . “Graph theoretic software watermarks: Implementation, analysis, and attacks”. In Workshop on Information Hiding, pp. 192-207, 2004. Springer-Verlag.

[41] Collberg, C.; Debray, S.; Carter, E.; Huntwork, A.; Linn, C.; Stepp, M. . “Dynamic Path-Based Software Watermarking”. In: Proc. SIGPLAN ’04 Conf. on Prog. Language Design and Implementation (PLDI 04), June 2004.

[42] Cousot, P.; Cousot, R. . “An abstract interpretation-based framework for software watermarking”. In: Proc. of Principles of Programming Languages, 2004, Venice, Italy. ACM.

[43] Giffin J. T.; Christodorescu M.; Kruger L. “Strengthening software self-checksumming via self-modifying code". In: 21st Annual Computer Security Applications Conference, 2005, pp. 23–32, IEEE Computer Society.

[44] Holz T.; Raynal F. . “Detecting honeypots and other suspicious environments”. In: Proceedings of the 2005 IEEE Workshop on Information Assurance and Security, U.S. Military Academy, West Point, NY, June 2005.

[45] Lakhotia, A.; Kumar, E. U.; Venable, M. . “A method for detecting obfuscated calls in malicious binaries”. In: IEEE Transactions on Software Engineering, Piscataway, v. 31, n. 11, pp. 955–968, 2005.

[46] Madou, M.; Anckaert, B.; Moseley, P.; Debray, S.; De Sutter, B.; De Bosschere, K. . “Software protection through dynamic code mutation”. In: 6th International Workshop in Information Security Applications, August 2005.

[47] Myles, G.; Collberg, C.; Heidepriem, Z.; Navabi, A. . “The evaluation of two software watermarking algorithms”. Software: Practice and Experience, 35(10):923-938,2005.

[48] Skorobogatov, S. P. . “Semi-invasive attacks – A new approach to hardware security analysis”. Technical Report 630 - University of Cambridge, 2005.

[49] Udupa, S. K; Debray, S. K.; Madou, M. . “Deobfuscation: Reverse engineering obfuscated code”. In: WCRE ’05: Proceedings ofthe 12th Working Conference on Reverse Engineering, pages 45–54, Washington, DC, 2005. IEEE.

[50] Wurster, G.; van Oorschot, P. C.; Somayaji, A. . “A generic attack on checksumming-based software tamper resistance”. In: IEEE Symposium on Security and Privacy, Oakland, CA, pp. 127–138, May 2005.

[51] Seshadri, A.; Luk, M.; Perrig, A.; van Doorn, L.; Khosla, P. . “Enternally verifiable code execution”. Commun. ACM, 49(9):45–49, 2006.

[52] Tan, G.; Chen, Y.; Jakubowski, M. H. . “Delayed and controlled failures in tamper-resistant systems”. In: Information Hiding, 2006. Springer-Verlag.

[53] Wong, W.; Stamp, M. . “Hunting for Metamorphic Engines”. Journal in Computer Virology (Department of Computer Science, San Jose State University).

[54] Farrell, N. . “Mac Display Eater kills home files”. The Inquirer (February 27 , 2007), www.theinquirer.net/default. aspx?article=37824.

[55] Hardekopf, B.; Lin, C. . “The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code”. SIGPLAN Not., 42(6):290–299, 2007.

[56] Jacob, M.; Jakubowski, M. H.; Venkatesan, R. . “Towards integral binary execution: implementing oblivious hashing using overlapped instruction encodings”. In: Proc. of the 9th Workshop on Multimedia and Security, pp. 129–140, New York, 2007. ACM.

[57] Srinivasan, R. . “Protecting anti-virus software under viral attacks”. M.Sc. Thesis, Arizona State University.

[58] Ferrie, P. . “Anti-Unpacker Tricks”. In: 2nd International CARO Workshop, 2008, The Netherlands.

[59] Baker, D. . “Making a Secure Smart Grid a Reality”. Journal of Energy Security 2009.

[60] Goodspeed *et al.*. “Low level Design Vulnerabilities in Wireless Control System Hardware”, S4 2009 papers.

[61] IDA Pro - Disassembler. 2009. Data Rescue, Liege, Belgium. Available from Internet: <http://www.datarescue.com. Último acesso July 2009>.

[62] Boccardo, D. R. . “Context-Sensitive Analysis of x86 Obfuscated Executables” PhD thesis, Universidade Estadual Paulista, Departamento de Engenharia Elétrica - FEIS, 2009.

[63] Oliveira, L. P. M. . “Marca d’água Frágil e Semi-frágil para Autenticação de Vídeo no Padrão H.264”. Dissertação de mestrado, Universidade Federal do Rio de Janeiro, 2009.

[64] z0mbie. “Automated reverse engineering: Mistfall engine.” Publicado online em http://z0mbie.host.sk/autorev.txt . Último acesso, Julho 2010.

[65] PECompact. “Windows executable compressor”. Bitsum Technologies. http://www.bitsum.com . Último acesso, Julho 2010.

[66] Petitcolas, F. A. P. . “Stirmark Benchmark 4.0”. http://www.petitcolas.net/fabien/watermarking/stirmark . Último acesso, Julho 2010.