República Federativa do Brasil
Ministério do Desenvolvimento, Indústria e do Comércio Exterior
Instituto Nacional da Propriedade Industrial.

(21) **PI0614089-0 A2**

BRPI0614089A2

(22) Data de Depósito: 30/03/2006
(43) Data da Publicação: 09/03/2011
(RPI 2096)

(51) Int.Cl.:
G06F 21/22

(54) Título: **MÉTODO PARA EVITAR ENGENHARIA REVERSA DE SOFTWARE, MODIFICAÇÃO NÃO AUTORIZADA E INTERCEPTAÇÃO DE DADOS DE TEMPO DE EXECUÇÃO**

(30) Prioridade Unionista: 06/08/2005 US 60/595.802

(73) Titular(es): Secured Dimensions Ltd.

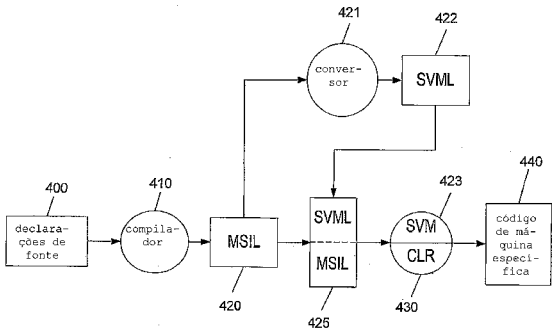
(72) Inventor(es): Boris Asipov, Keren Asipov

(74) Procurador(es): ALEXANDRE FERREIRA

(86) Pedido Internacional: PCT IL2006000398 de 30/03/2006

(87) Publicação Internacional: WO 2007/017856de 15/02/2007

(57) Resumo: MÉTODO PARA EVITAR ENGENHARIA REVERSA DE SOFTWARE, MODIFICAÇÃO NÃO AUTORIZADA E INTERCEPTAÇAO DE DADOS DE TEMPO DE EXECUÇAO Um método para evitar a modificação desautorizada de um software ou modificação desautorizada de dados de tempo de execução. De acordo com esse método, um conversor, que é capaz de converter o software em um código de máquina generalizado é fornecido. O conversor é projetado de tal modo que não possa sofrer engenharia reversa, pelo uso de um processo de conversão que causa perda de dados. Um intérprete, cujo conhecimento de seu método de processo é mantido restrito, também é fornecido. O intérprete interpreta o código de máquina geral em um código de máquina específico, enquanto reconstrói os dados perdidos durante o processo de interpretação.



"MÉTODO PARA EVITAR ENGENHARIA REVERSA DE SOFTWARE, MODIFICAÇÃO NÃO AUTORIZADA E INTERCEPTAÇÃO DE DADOS DE TEMPO DE EXECUÇÃO"

CAMPO DA INVENÇÃO

A presente invenção refere-se ao campo de proteção de software de computador. Mais particularmente, a invenção refere-se a um método para proteger software de computador contra engenharia reversa, modificação não autorizada e interceptação de dados de tempo de execução.

ANTECEDENTES DA INVENÇÃO

Muitas tentativas têm sido feitas nos últimos anos para proteger software de computador original contra duplicação e distribuição em massa. Um dos métodos utilizados atualmente envolve a necessidade de uma licença ou uma chave de seqüência, que é entrada manualmente pelo cliente, durante instalação ou durante tempo de execução. Outro método popular para evitar uso duplicata de software envolve a ativação do software após instalação. O processo de ativação requer que o software leia números seriais de ID de elementos de hardware no computador, como o número serial do processador ou número serial do cartão gráfico. Após leitura dos números seriais de ID de hardware, os mesmos podem ser enviados juntamente com o número de ID de software através da Internet para o vendedor. O vendedor armazena os números de ID e envia um código de licença para o programa através da Internet. O software pode ser programado para cessar funcionamento adequado sem um código de licença autenticado a partir do vendedor. Nesse caso, se o software for ilegalmente copi-

ado e instalado em um computador diferente, o software não pode ser ativado uma vez que a licença do software já está associada ao hardware da primeira instalação e o código de licença pode ser enviado somente para o computador tendo os mesmos perfis de hardware armazenados pelo vendedor. Entretanto, esses métodos não evitam que uma parte não autorizada realize engenharia reversa do código de software, modifique o mesmo para excluir essas ferramentas de proteção de software, e distribua em massa o software modificado.

Muitas ferramentas estão em uso atualmente para engenharia reversa de software, como o dumper hexadecimal, que imprime ou exibe os números binários de um código de software em formato hexadecimal. Conhecendo os padrões de bit que representam essas instruções, bem como os comprimentos de instruções, uma pessoa que deseja realizar engenharia reversa o software pode identificar certas porções de um código para ver como funcionam, e então modificar as mesmas. Outra ferramenta comum para engenharia reversa e modificação de código é o desassemblador. O desassemblador lê o código binário e então exibe cada instrução executável em um formato textual. Além disso, uma vez que o desassemblador não pode dizer a diferença entre uma instrução executável e os dados utilizados pelo código, um depurador pode ser utilizado. O depurador permite que o desassemblador evite desassemblagem as porções de dados de um código. Por exemplo, se o desassemblador ler um comando "ADD_INT8", que significa: "adicionar o número representado nos 8 bits seguintes", o depurador processa os 8 bits seguintes como a porção de dados do

comando "ADD_INT8", e o próximo grupo de bits é processado como um comando novo. Entretanto, essas ferramentas se baseiam no conhecimento publicado de como o código de instruções é construído, onde as informações residem na memória, quais registros são utilizados, e como a pilha (uma armazenagem de dados utilizada para armazenar solicitações que necessitam ser tratadas, na forma de uma lista de empurrar para baixo) é utilizada.

O problema de engenharia reversa e modificação de código por usuários não autorizados é ainda mais evidente ao lidar com linguagens de programação baseadas em intérprete, ao contrário de linguagens de programação baseadas em compilador. Uma descrição de linguagem de programação baseada em compilador pode ser encontrada na figura 1, que ilustra genericamente o processo de software da técnica anterior de linguagens de programação baseados em compilador, como C ou Pascal. Quando um programador programa em uma linguagem de alto nível utilizando um editor ou similar, as instruções de seu código 10, ou código de fonte, não podem ser lidas diretamente pelo hardware de computador. Portanto, o código de fonte 10 tem de se submeter a um processo de tradução conhecido como compilação pelo compilador 11. O compilador 11 compila código de fonte 10 em um Código de Máquina (MC) específico 12, o qual o hardware do computador é capaz de ler e executar. Uma vez que o MC 12 é especificamente compilado para uma certa plataforma, não pode ser transferido de uma plataforma para outra. Nas linguagens de programação baseadas em compilador o código de fonte 10 é compilado para cada

plataforma individualmente produzindo um MC específico, diferente, 12 para cada plataforma. Um exemplo de plataformas diferentes pode ser um PC baseado em Intel® com Windows® XP e Mac® OS X.

Uma descrição de linguagem de programação baseada em intérprete pode ser encontrada na figura 2A que é um fluxograma, genericamente ilustrando o processo de software da técnica anterior de linguagens de programação baseadas em intérprete, como JAVA. Similares às linguagens de programação baseadas em compilador, as linguagens baseadas em intérprete são gravadas em linguagem de alto nível, utilizando um editor ou similar, mencionado a seguir como declarações de fonte 20. Entretanto, de acordo com essa abordagem, o compilador 21 traduz as declarações de fonte de alto nível 20 para um Código de byte (BC) 22 que é um MC generalizado não limitado a uma certa plataforma. Não obstante, para executar o BC 22, um intérprete específico 23 é necessário para traduzir BC 22 em MC específico 24. O intérprete específico 23 é normalmente instalado juntamente com o sistema operacional. A principal vantagem dessa abordagem é que BC 22 pode ser distribuído para diferentes plataformas. Após BC 22 ser executado em uma certa plataforma, o intérprete específico 23 traduz somente uma instrução de BC 22 de cada vez, produzindo uma instrução de MC 24 específica para execução pelo hardware de computador. Entretanto, uma vez que o método de intérprete de processamento é um conhecimento comum, é relativamente fácil de ler, entender e modificar o BC 22 que é um conjunto de instruções para o intérprete 23. Um hacker

pode comprar uma cópia legal de um código gravado no BC, decifrar suas instruções e apagar ou modificar algumas das instruções originais do BC. Após modificação do BC, o mesmo pode ser copiado em massa e revendido.

Outro método utilizado por hackers é conhecido na técnica como "interceptação de dados de tempo de execução." Por interceptar e ler o fluxo de dados durante a execução de um programa legal pelo intérprete, o hacker pode simular o processo ao executar um programa ilegal.

Um método para evitar fácil entendimento e decifragem do comportamento de código utiliza criptografia do código, como descrito na US 2004/0015710. De acordo com essa abordagem, o código criptografado é vendido com uma chave de decriptografia para decriptografar o código. Cada instrução no código é primeiramente decriptografada e interpretada por um intérprete para execução pelo processador. Entretanto, após decriptografia de código, um hacker pode ler o código decriptografado para realizar engenharia reversa do código original. Além disso, o processo de decriptografia pode ser monitorado por um usuário para formular a chave de decriptografia. Além disso, após decriptografia de código, o mesmo é carregado desprotegido na memória do computador e pode ser copiado a partir da mesma, também.

Outro método para evitar modificação de um código de software é dividir o código em 2 partes, uma parte sensível que compreende a proteção de código, e uma parte menos sensível. A parte menos sensível do código é vendida ao usuário, como anteriormente, pronta para interpretação, ao pas-

so que a parte sensível do código é armazenada em produtos de hardware, como smart-cards. A interpretação da parte sensível do código é feita em hardware, como uma leitora de smart-card, onde não pode ser monitorada ou lida. Entretanto, em alguns dos casos, o hardware adicional pode ser caro, e redistribuição de atualizações de código geradas pelo provedor é complicada.

Um método para evitar modificação de um código de software é descrito em um artigo por Enriquillo Valdez e Moti Yung "DISSECT: Distribution for SECurity Tool" (G.I. Davida e Y. rankel (Eds.): ISC 2001, LNCS 2200, pág. 125-143, 2001. Springer-Verlag Berlin Heidelberg 2001). O método sugere a divisão do código em 2 partes, uma parte sensível e uma parte menos sensível. A parte menos sensível do código é vendida ao usuário, como anteriormente, pronta para interpretação, ao passo que a parte sensível do código é armazenada em um servidor seguro. A interpretação da parte sensível do código é feita em um servidor seguro, onde não pode ser monitorada ou lida. Entretanto, essa abordagem requer a manutenção de um contato direto com o servidor designado para executar o código.

Portanto, é um objetivo da presente invenção fornecer um método barato para evitar engenharia reversa de software, modificação não autorizada, e interceptação de dados de tempo de execução.

É outro objetivo da presente invenção fornecer um método para evitar modificação não autorizada de software, sem necessitar de hardware adicional.

É ainda outro objetivo da presente invenção fornecer um método que, por um lado, evite qualquer modificação por um usuário não autorizado e por outro lado, permite modificação e atualização pelo vendedor.

Outros objetivos e vantagens da invenção tornar-se-ão evidentes à medida que a descrição prossegue.

Sumário da invenção

A presente invenção é dirigida a um método para evitar a modificação desautorizada de um software ou modificação desautorizada de dados de tempo de execução. É fornecido um conversor, capaz de converter o software em um código de máquina generalizado que não pode sofrer engenharia reversa, pelo uso de um processo de conversão que cause perda de dados e um intérprete que pode ser compilado por um CLR. O conhecimento do método de processo de intérprete é mantido restrito. O código de máquina geral é interpretado pelo intérprete em uma máquina específica.

O software pode ser uma linguagem de alto nível (por exemplo, Java, Visual, J#, J#, C#, ou VB.NET ou uma linguagem baseada em compilador, por exemplo, C++, VB, ou Pascal), como uma linguagem baseada em intérprete, e pode ser dividido de tal modo que somente parte do software é convertida com o conversor e interpretada pelo intérprete.

A perda de dados durante conversão pode ser a remoção de metadados de estrutura de código ou a conversão de instruções em outras instruções que seu operando(s) correspondente(s) é determinado durante tempo de execução.

Breve descrição dos desenhos

Nos desenhos:

A figura 1 é um fluxograma ilustrando genericamente o processo de software da técnica anterior de linguagens de programação baseadas em compilador;

A figura 2a é um fluxograma ilustrando genericamente o processo de software da técnica anterior de linguagens de programação baseadas em intérprete;

A figura 2b é um fluxograma ilustrando genericamente o processo de software da técnica anterior de linguagens de programação baseadas em intérprete, principalmente JAVA;

A figura 3 é um fluxograma ilustrando genericamente o processo de software da técnica anterior de linguagens de programação de .NET, como Visual J#;

A figura 4 é um fluxograma ilustrando genericamente a implementação da invenção de acordo com uma das modalidades;

A figura 5 é um diagrama de blocos ilustrando uma das modalidades da invenção;

A figura 6 é um fluxograma ilustrando genericamente a implementação da invenção de acordo com outra modalidade da invenção; e

A figura 7 é um fluxograma ilustrando genericamente a implementação da invenção de acordo com uma das modalidades, para linguagens de programação baseadas em compilador.

Descrição detalhada das modalidades preferidas

Para fins de brevidade, os seguintes termos são

definidos explicitamente:

- Uma plataforma é o sistema operacional do computador que é construído no conjunto de instruções para o processador de computador, o hardware que executa operações lógicas e gerencia movimento de dados no computador.

- Código de máquina (MC) é o código que pode ser lido e executado diretamente pelo processador do computador.

- Código de máquina específica é o código que pode ser somente lido e executado por uma plataforma específica, ou um número de plataformas especificadas.

- Código de máquina generalizado é o código que não é limitado a uma plataforma específica.

- um compilador converte um conjunto de instruções em um código de máquina.

Descrição de processos bem conhecidos

A figura 2b é um fluxograma ilustrando genericamente o processo de software da técnica anterior de linguagens de programação baseadas em intérprete, como JAVA. Similar às linguagens de programação baseadas em compilador, as linguagens baseadas em intérprete são gravadas em linguagem de alto nível, utilizando um editor ou similar, mencionado a seguir como declarações de fonte 200. De acordo com essa abordagem, o compilador 210 traduz as declarações de fonte de alto nível 200 em um Código de Byte (BC) 220, que é um MC generalizado que não é limitado a uma certa plataforma. Não obstante, para executar o BC 220, um intérprete específico 230 é necessário para traduzir BC 220 em um MC específico 240. O intérprete específico 230 é normalmente instalado

juntamente com o sistema operacional. A principal vantagem dessa abordagem é que BC 220 pode ser distribuído para plataformas diferentes. Após BC 220 ser executado em uma certa plataforma, o intérprete específico 230 traduz um comando de BC de cada vez, desse modo produzindo um comando MC específico para o hardware de computador executar. Ao lidar com Sun Microsystems® Java, BC 220 é denominado Código de Byte Java e o intérprete 230 é denominado uma Máquina Virtual (VM). Em alguns casos, a VM vem junto com um compilador Just-in-time 250 e é utilizado opcionalmente. O compilador Just-in-time 250 compila BC de Java 220 em um MC específico 260 como se o programa tivesse sido compilado inicialmente para aquele programa específico. Nos dois casos de VM 230 e compilador Just-in-time 250, o hardware de computador lê seu MC específico pretendido. Entretanto, uma vez que o intérprete 230 traduz um comando de BC 220 de cada vez durante execução, pode rodar mais lento no computador.

A VM Java, que opera como um intérprete entre BC Java e um MC específico, é individual para cada plataforma. Após VM Java ter sido fornecido para uma plataforma, qualquer BC Java compilado pode ser rodado naquela plataforma. Portanto, quando um usuário tiver uma VM Java instalada em seu computador, ele pode adquirir qualquer programa em BC Java, e executar o mesmo em seu computador. Quando um programador programa em Java e compila o programa em um BC Java, ele pode distribuir o BC Java amplamente para qualquer usuário, visto que o BC Java é compatível para todas as plataformas populares. A VM Java é responsável por alocar memó-

ria, definir registros, empilhar, heap de "lixo", e área de método (área de método de uma VM Java é uma área lógica de memória que armazena todas as informações sobre os tipos carregados), para a execução de programa.

A figura 3 é um fluxograma ilustrando genericamente o processo de software da técnica anterior de programar linguagens que são projetadas para rodar em Microsoft®NET, como Visual J#. Dito em termos gerais, o ambiente .NET permite o uso de recursos da Web em vez dos recursos de computador para vários serviços. J# Visual ou J# permite que os programadores programem em linguagem "similar à Java" e rodem o programa em .NET. As declarações de fonte 300, gravadas em linguagem de alto nível Visual J#, são compiladas pelo compilador 310 em Linguagem Intermediária Microsoft (MSIL) 320, que é um MC geral que não é limitado a uma plataforma específica. O MSIL 320 é equivalente ao BC Java 220 em suas funções, e o BC Java 220 pode até mesmo ser convertido facilmente em MSIL 320. Similar ao processo descrito acima, o MSIL 320 é convertido em um MC específico 340 utilizando Tempo de execução de Linguagem comum (CLR) 330, que é equivalente à função de JAVA VM 230. Como entendido, outras linguagens de programação .NET como C# e VB.NET são submetidas a um processo similar a partir de declarações de fontes 300 para MSIL 320 para MC específico 340.

Deve ser observado que as alocações de memória do VM e CLR descritos são amplamente conhecidas pelos hackers, como a definição dos registros, pilha, heap de "lixo", e área de método do programa. Utilizando essa informação, o

hacker pode entender quais comandos do BC Java ou MSIL se referem à exigência de licença e modificar esses comandos.

Um metadado .NET Na estrutura .NET da Microsoft descreve o código CIL (Linguagem Intermediária comum) .NET. Um compilador de linguagem .NET gerará os metadados e armazenará esses na montagem contendo o CIL. Os metadados descrevem todas as classes e membros de classe que são definidos na montagem, e as classes e membros de classe que a montagem atual chamará a partir de outra montagem. Os metadados para um método contêm a descrição completa do método, incluindo a classe (e a montagem que contém a classe), o tipo de retorno e todos os parâmetros de método. Quando o CLR executa CIL, verifica que os metadados do método chamado são iguais aos metadados que são armazenados no método que chama. Isso assegura que um método possa somente ser chamado exatamente com o número certo de parâmetros e exatamente os tipos certos de parâmetros. Portanto, em ambientes como .NET e Java é mais fácil realizar engenharia reversa do código visto que o código e metadados são fornecidos juntos como parte do pacote redistribuível. Os metadados são necessários para compilação Just-in-time de código para a plataforma alvo. Entretanto, em linguagens baseadas em compilador como C++, os metadados são abandonados durante a compilação e estágios de link e não são redistribuídos para usuários finais.

Descrição geral da invenção

A essência da invenção é um intérprete, cujos métodos de operação e alocações de memória são não publicados. O novo intérprete não revelado ou "Secret VM", é mencionado

a seguir como SVM. Cada SVM é emparelhado com um conversor correlacionado, ou em outras palavras, cada SVM pode somente interpretar um código que foi produzido por um conversor correlacionado. Portanto, cada vendedor de software original que requer proteção de software pode adquirir um par correlacionado, exclusivo de conversor e SVM. Os métodos de operação, como codificação de instrução ou alocações de memória, podem variar entre SVMs diferentes.

A figura 4 ilustra uma implementação da invenção, de acordo com uma das modalidades, onde as declarações de fonte 400 são gravadas em linguagem de programação de alto nível de .NET. O compilador 410 compila declarações de fonte 400 em MSIL 420 como descrito na técnica anterior. Nesse ponto, o conversor 421 é utilizado para converter MSIL 420 em uma Linguagem de Máquina Virtual Secreta (SVML) 422. A SVML 422 é um MC geral, não limitado a uma certa plataforma. Entretanto, os comandos da SVML 422 são diferentes dos comandos de MCs gerais conhecidos como os comandos de BC Java ou MSIL 420. Portanto, a decifragem da SVML 422 é excepcionalmente complicada, uma vez que não existe desassemblador ou depurador conhecido para SVML 422. A SVML 422 pode ser distribuída juntamente com a SVM correspondente 423. A SVM 423 é compilada no computador designado utilizando o CLR local 430 para adicionar os dados referentes à plataforma específica do computador designado. Uma vez que a SVM 423 desempenha como intérprete, compreende não somente novos dados para interpretar SVML 422, como também dados referentes ao perfil de plataforma a partir de CLR 430. Desse modo, quando

SVML 422 é executado no computador designado, SVM 423 interpreta cada comando para o hardware para execução. Uma vez que o método de processamento da SVM 423 é desconhecido, um hacker achará difícil entender e modificar o código, ou tentar interceptar os dados durante tempo de execução.

Descrição geral dos atributos propostos do conversor

Um dos atributos designados do conversor envolve a produção de programas de SVML diferentes para a mesma entrada de MSIL (de outro modo conhecida como "morphing de código"). O morphing de código se baseia em uma redundância no conjunto de instruções de SVML, por exemplo, a instrução SUB pode ser substituída por instruções ADD e NEG. Esse atributo é principalmente eficaz para evitar tentativas de comparar o conjunto de instruções de MSIL com o conjunto de instruções de SVML equivalente. Esse atributo se baseia em uma redundância no conjunto de instruções de SVML, por exemplo, a instrução SUB pode ser substituída por instruções NEG e ADD.

I. Outro atributo designado do conversor é a possibilidade de codificação dinâmica de instruções, significando a alteração de padrão de bit correspondente, ou código de uma certa instrução. Ao contrário da MSIL, onde se espera que as mesmas instruções sejam codificadas similarmente, em SVML a mesma instrução pode aparecer em códigos diferentes. Por exemplo, a instrução pode ser codificada com seu endereço como mostrado nas seguintes tabelas:

Endereço de instrução	Instrução de MSIL	Código

70	ADD	20

Endereço de instrução	Instrução de SVML	Código
70	ADD	20+70 = 90
74	ADD	20 + 74 = 94

Portanto, mesmo se um hacker pudesse tentar encontrar padrões de repetição em um código SVML para deduzir instruções comuns, ele considerará mais complicado do que assumido.

II. O atributo projetado principal do conversor é a causa de perda de dados durante conversão por tornar o processo de conversão praticamente irreversível. Um exemplo de perda de dados é a remoção de metadados de estrutura de código, como declarações de método, uma vez que em .NET não é necessário quando um método é chamado somente por outros métodos transformados. O exemplo adicional de perda de dados é o seguinte: um conjunto de instruções amplamente conhecido compreende as seguintes instruções: ADD_INT8, ADD_INT16, e ADD_INT32. Essas instruções instruem o processador a somar os números 8, 16 ou 32 bits de acordo. Durante o processo de conversão, utilizando o conversor exclusivo, todas essas instruções são convertidas em instruções abertas "ADD". O tipo de operando e número de bits, que deve ser somado (8, 16 ou 32) é determinado durante tempo de execução. Portanto, a compilação reversa é impossível sem conhecer o número para somar na instrução "ADD". Uma vez que o processo de conversão é irreversível, o código não pode ser convertido de vol-

ta em um formato BC Java/MSIL padrão, e portanto não pode ser descompilado, desassemblado, depurado ou modificado utilizando ferramentas padrão.

Exemplo de arquitetura SVM e conjunto de instruções SVML correspondente

A figura 5 é um diagrama de blocos ilustrando um exemplo de arquitetura SVM, de acordo com uma das modalidades. A Unidade de Lógica aritmética 500 executa as operações lógicas em registros de operando 510 e 520, e armazena o resultado no registro 530. O registro de transferência de dados 560 é utilizado para transferir dados entre registros e Bancos de memória 540 e 550. Bancos de memória 540 e 550 são utilizados para armazenar variáveis locais e parâmetros de método. O registro seletor de banco 570 armazena o número de Banco de memória em uso.

Um exemplo de um subconjunto de instruções de SVML e seu significado:

Instrução de SVML	Descrição
MEM2TRANSFER	Copia conteúdo de endereço de memória especificado para o Registro de transferência de dados 560
SETMBANK1	Define o Registro seletor de banco 570 em 1
SETMBANK2	Define o Registro seletor de banco 570 em 2
TRANSFER2MEM	Copia o conteúdo do Registro de transferência de dados 560 para o endereço de me-

	mória especificado
TRANSFER2OP1	Copia o conteúdo do Registro de transferência de dados 560 para o Registro de Operando 510
TRANSFER2OP2	Copia o conteúdo do Registro de transferência de dados 560 para o Registro de Operando 520
RESULT2TRANSFER	Copia o conteúdo do Registro de Resultado 530 para o Registro de transferência de dados 560
ADD	Executa a operação aritmética de adição nos registros de Operando 1 e Operando 2 e armazena o resultado no Registro de Resultado 530
SUB	Executa a operação aritmética de subtrair nos registros de Operando 1 e Operando 2 e armazena o resultado no Registro de Resultado 530

Comparação entre um código de MC geral e código de SVML

Para fins de brevidade um exemplo não limitador é mostrado aqui abaixo comparando um código de programa de montagem do MC geral da técnica anterior com aquele da SVML. Nos dois casos, a tarefa dada exigiu processamento da equação 4+3-1.

Um código de programa de MC geral processando a equação 4+3-1:

LDC 4

LDC 3

ADD

LDC 1

SUB

Um código de programa de SVML processando a equação 4+3-1:

SETMBANK1 - seleciona o banco de memória 1

MEM2TRANSFER 23 - carrega "4", uma constante armazenada no endereço de memória 23

TRANSFER2OP1 - mover o valor "4" para o registro de operando 1

MEM2TRANSFER 45 - carrega "3", uma constante armazenada no endereço de memória 45

TRANSFER2OP2 - mover o valor "3" para o registro de operando 2

ADD - somar conteúdo dos registros de operando 1 e 2

RESULT2TRANSFER - mover o resultado "7" para o registro de transferência

TRANSFER2OP1 -mover o valor "7" para o registro de operando 1

MEM2TRANSFER 12 - Carregar "1", uma constante armazenada no endereço de memória 12

TRANSFER2OP2 -mover o valor "1" para o registro de operando 2

SUB - subtrair o conteúdo do operando 2 do operando 1

RESULT2TRANSFER - mover o resultado para o regis-

tro de transferência

TRANSFER2MEM 1 - armazenar o resultado no endereço de memória 1

Como mostrado no código de programa acima, os operandos processados (4, 3 e 1) nunca são representados explicitamente nas instruções. Um hacker que tenta engenharia reversa no programa não pode deduzir do presente conjunto de instruções quais são os valores de operandos na equação, visto que cada valor é lido da memória durante tempo de execução.

Modalidades adicionais da invenção

Em uma das modalidades, cada vendedor é equipado com seu próprio par de conversor e SVM. Portanto, conhecer o método de processo de um SVM não revela o método de processo de outras SVMs.

O método proposto pela presente invenção pode ser utilizado com qualquer linguagem baseada em intérprete. Por exemplo, para J# de .NET a SVM é compilada pelo CLR, para JAVA a SVM é compilada pela VM, e assim por diante. A invenção proposta pode ser utilizada para qualquer software quer seja uma linguagem de alto nível como C# ou VB.NET, um código de software, um código de fonte ou um código de máquina.

A figura 6 ilustra um exemplo de outra modalidade da invenção, onde as declarações de fonte 500 são gravadas em linguagem de alto nível, como Visual J#. O compilador 410 compila declarações de fonte 400 em MSIL 420 como descrito anteriormente. Entretanto, antes da conversão, as instruções da MSIL 420 são divididas em dois grupos, instruções sensí-

veis, que podem incluir as exigências de licença, e instruções insensíveis. As instruções sensíveis são convertidas pelo conversor 421 em SVML 422, ao passo que as instruções insensíveis não são convertidas. O MC geral 425, ou programa, que consiste em uma parte MSIL e uma parte SVML, pode ser distribuído juntamente com a SVM correspondente 423 para qualquer plataforma popular. Para executar o programa, SVM 423 é compilado por CLR 430 na plataforma designada. Durante execução, cada instrução é examinada em relação à compatibilidade com MSIL 420 ou SVML 422. As instruções de MSIL 420 são interpretadas diretamente por CLR 430, ao passo que as instruções de SVML 422 são interpretadas por SVM 423.

A figura 7 ilustra uma implementação da invenção, de acordo com uma das modalidades, para linguagens de programação baseadas em compilador. Como descrito nos antecedentes, o código de fonte 700 é compilado pelo compilador 710 em MC específico 720. O MC específico 720 é convertido utilizando um conversor designado 721 em uma SVML 722, onde a SVML 722 é dependente de plataforma, ou em outras palavras é um MC específico. A SVML 722 é distribuída com uma SVM 723 projetada para a plataforma específica da SVML 722. Uma vez que a SVM distribuída 723 já é especificada para uma plataforma designada, não requer compilação no computador designado. Portanto, a SVM 723 é capaz de traduzir a SVML 722 em MC específico 740, para o hardware do computador designado.

Em outra modalidade para linguagens de programação baseadas em compilador, somente as instruções sensíveis são convertidas pelo conversor 721 em SVML 722. A SVML 722 é

distribuída com a SVM 723 e as instruções restantes de MC específico 720. Durante execução a SVM 723 executa as instruções da SVML 722.

Embora algumas modalidades da invenção tenham sido descritas como ilustração, será evidente que a invenção pode ser colocada em prática com muitas modificações, variações e adaptações, e com o uso de inúmeros equivalentes ou soluções alternativas que estão compreendidas no escopo das pessoas versadas na técnica, sem se afastar do espírito da invenção ou exceder o escopo das reivindicações.

PI0614089-0

REIVINDICAÇÕES

1. Método para evitar a modificação desautorizada de um software ou modificação desautorizada de dados de tempo de execução, **CARACTERIZADO** por compreender as etapas de:

a. fornecer um conversor, capaz de converter o software em um código de máquina generalizado, que não pode sofrer engenharia reversa, utilizando um processo de conversão que causa perda de dados e

b. fornecer um intérprete, cujo conhecido de seu método de processo é mantido restrito,

c. interpretar, pelo intérprete, o código de máquina geral em um código de máquina específico enquanto reconstrói os dados perdidos durante o processo de interpretação.

2. Método, de acordo com a reivindicação 1, **CARACTERIZADO** pelo fato de que o intérprete é compilado por um CLR.

3. Método, de acordo com a reivindicação 1 ou 2, **CARACTERIZADO** pelo fato de que o software é dividido de tal modo que somente parte do software é convertida com o conversor e interpretada pelo intérprete.

4. Método, de acordo com a reivindicação 1, **CARACTERIZADO** pelo fato de que o software é uma linguagem de alto nível.

5. Método, de acordo com a reivindicação 4, **CARACTERIZADO** pelo fato de que a linguagem de alto nível é uma linguagem baseada em intérprete.

6. Método, de acordo com a reivindicação 5,

CARACTERIZADO pelo fato de que a linguagem de alto nível pode ser qualquer uma das seguintes: Java, Visual J#, J#, C#, ou VB.NET.

7. Método, de acordo com a reivindicação 4, **CARACTERIZADO** pelo fato de que a linguagem de alto nível é uma linguagem baseada em compilador.

8. Método, de acordo com a reivindicação 7, **CARACTERIZADO** pelo fato de que a linguagem de alto nível pode ser qualquer uma das seguintes: C++, VB ou Pascal.

9. Método, de acordo com a reivindicação 1, **CARACTERIZADO** pelo fato de que a perda de dados durante conversão é a remoção de metadados de estrutura de código.

10. Método, de acordo com a reivindicação 1, **CARACTERIZADO** pelo fato de que a perda de dados durante conversão é a conversão de instruções em outras instruções as quais seu(s) operando(s) correspondente(s) são determinados durante tempo de execução.

11. Método para evitar a descompilação e modificação de tempo de execução de um código de software .NET/CLI utilizando ferramentas que se baseiam em metadados .NET/CLI, **CARACTERIZADO** por compreender:

a. fornecer um conversor, capaz de converter o código de software .NET/CLI em um código de máquina geral, onde a conversão envolve perda de dados de metadados .NET/CLI;

b.fornecer um novo intérprete capaz de interpretar o código de máquina geral sem exigir metadados .NET/CLI, onde o novo intérprete é implementado como componente de software baseado em .NET/CLI para manter o mesmo nível de porta-

bilidade que o código de software .NET/CLI; e

c. interpretar o código de máquina geral em um código de máquina específico, pelo intérprete.

12. Método, de acordo com a reivindicação 11, **CARACTERIZADO** pelo fato de que o software é dividido de tal modo que somente parte do software seja convertida com o conversor e interpretada pelo novo intérprete.

13. Método para evitar violação de um mecanismo de execução de termos de licença de software em um ambiente .NET/CLI, de acordo com a reivindicação 11, **CARACTERIZADO** pelo fato de que as ferramentas de descompilação e modificação que se baseiam em metadados .NET/CLI se tornam inutilizáveis após aplicar o método.

14. Método, de acordo com a reivindicação 11, para criptografar um código .NET/CLI enquanto supera a ausência de suporte de criptografia/decriptografia de código em .NET CLR pela execução de uma decriptografia parcial para a função atualmente executada, **CARACTERIZADO** por compreender:

a. aplicar um processo de criptografia simétrico na representação de código de máquina geral de cada função convertida; e

b. decriptografar cada função de código de máquina geral antes da execução pelo novo intérprete.

15. Método, de acordo com a reivindicação 14, **CARACTERIZADO** pelo fato de que a chave de processo de criptografia simétrica é entregue ao usuário do software separadamente, desse modo tornando impossível a execução desautorizada de software.

16. Método, de acordo com a reivindicação 11, **CARACTERIZADO** pelo fato de que implementações diferentes do novo intérprete são fornecidas para vendedores de software, onde o novo intérprete é formado pela escolha de um subconjunto completo de instruções a partir de um conjunto de instruções redundantes, e por gerar dinamicamente um conversor e a montagem .NET/CLI da implementação do novo intérprete utilizando .NET Framework System.CodeDom API, System.Reflection.Emit API, ou geração de código de fonte de linguagem .NET direta.

17. Método, de acordo com a reivindicação 11, **CARACTERIZADO** pelo fato de que as implementações diferentes do novo intérprete são fornecidas a vendedores de software, pela aplicação de processo de obscurecimento com diferentes esquemas de renomear metadados na mesma implementação .NET/CLI de base do novo intérprete, para fazer com que os metadados pareçam diferentes em ferramentas de descompilação.

18. Método, de acordo com as reivindicações 16 ou 17, **CARACTERIZADO** pelo fato de que o novo intérprete é formado escolhendo um subconjunto de instruções completo a partir de um conjunto de instruções redundantes, e gerando dinamicamente um conversor e montagem .NET/CLI de implementação de intérprete novo utilizando .NET Framework System.CodeDom API, System.Reflection.Emit API, ou geração de código de fonte de linguagem .NET direta, e pela aplicação de processo de obscurecimento com diferentes esquemas de renomear metadados na mesma implementação de .NET/CLI de base

do novo intérprete, fazendo com que os metadados pareçam diferentes em ferramentas de descompilação.

19. Método para evitar a descompilação e modificação de tempo de execução de código de software Java utilizando ferramentas que se baseiam em metadados de classe Java, **CARACTERIZADO** por compreender:

a. fornecer um conversor capaz de converter o código de software Java em um código de máquina geral, onde a conversão envolve perda de dados de metadados Java;

b. fornecer um novo intérprete capaz de interpretar o código de máquina geral sem exigir metadados de classe Java, onde o novo intérprete é implementado com componente de software baseado em Java para manter o mesmo nível de portabilidade que o software de código de software Java; e

c. interpretar, pelo intérprete, o código de máquina geral em um código de máquina específico.

20. Método, de acordo com a reivindicação 19, **CARACTERIZADO** pelo fato de que o software é dividido de tal modo que somente parte do software seja convertida com o conversor e interpretada pelo novo intérprete.

21. Método para evitar a violação de um mecanismo de execução de termos de licença de software em um ambiente Java, de acordo com a reivindicação 11, **CARACTERIZADO** pelo fato de que as ferramentas de descompilação e modificação que se baseiam em metadados .NET/CLI se tornam inutilizáveis após aplicar o método.

22. Método, de acordo com a reivindicação 19, para criptografar um código Java enquanto supera ausência de su-

porte de criptografia/decriptografia de código em Java VM pela execução de uma decriptografia parcial para a função atualmente executada, **CARACTERIZADO** por compreender:

a. aplicar um processo de criptografia simétrico na representação de código de máquina geral de cada função convertida; e

b. decriptografar cada função de código de máquina geral antes da execução pelo novo intérprete.

23. Método, de acordo com a reivindicação 22, **CARACTERIZADO** pelo fato de que a chave de processo de criptografia simétrica é fornecida ao usuário de software separadamente, para tornar impossível execução desautorizada de software.

24. Método, de acordo com a reivindicação 19, **CARACTERIZADO** pelo fato de que implementações diferentes do novo intérprete são fornecidas a vendedores de software, onde o novo intérprete é formado escolhendo um subconjunto de instruções completo a partir de um conjunto de instruções redundantes, e gerando um conversor e implementação de intérprete novo.

25. Método, de acordo com a reivindicação 19, **CARACTERIZADO** pelo fato de que as diferentes implementações do novo intérprete são fornecidas a vendedores de software, por aplicação do processo de obscurecimento com diferentes esquemas de renomear metadados na mesma implementação Java de base do novo intérprete, para fazer com que os metadados pareçam diferentes em ferramentas de descompilação.

26. Método, de acordo com as reivindicações 24 ou

25, **CARACTERIZADO** pelo fato de que o novo intérprete é formado escolhendo um subconjunto de instruções completo a partir de um conjunto de instruções redundantes, e por gerar dinamicamente um conversor e implementação de intérprete novo, e aplicar processo de obscurecimento com diferentes esquemas de renomear metadados na mesma implementação de base do novo intérprete, para fazer com que os metadados pareçam diferentes em ferramentas de descompilação.

27. Método, de acordo com a reivindicação 11, **CARACTERIZADO** pelo fato de que o uso de ferramentas de instrumentação que se baseiam em metadados .NET/CLI para violar um mecanismo de execução de termos de licença de software é evitado pelo uso de arquitetura de intérprete novo para executar chamadas entre métodos convertidos.

28. Método, de acordo com a reivindicação 19, **CARACTERIZADO** pelo fato de que o uso de ferramentas de instrumentação que se baseiam em metadados de classe Java para violar um mecanismo de execução de termos de licença de software é evitado pelo uso de arquitetura de intérprete novo para executar chamadas entre métodos convertidos.

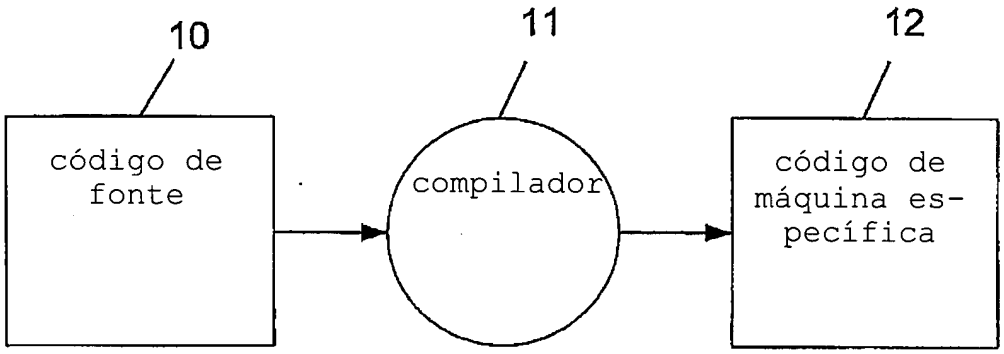


Fig.1

(estado da técnica)

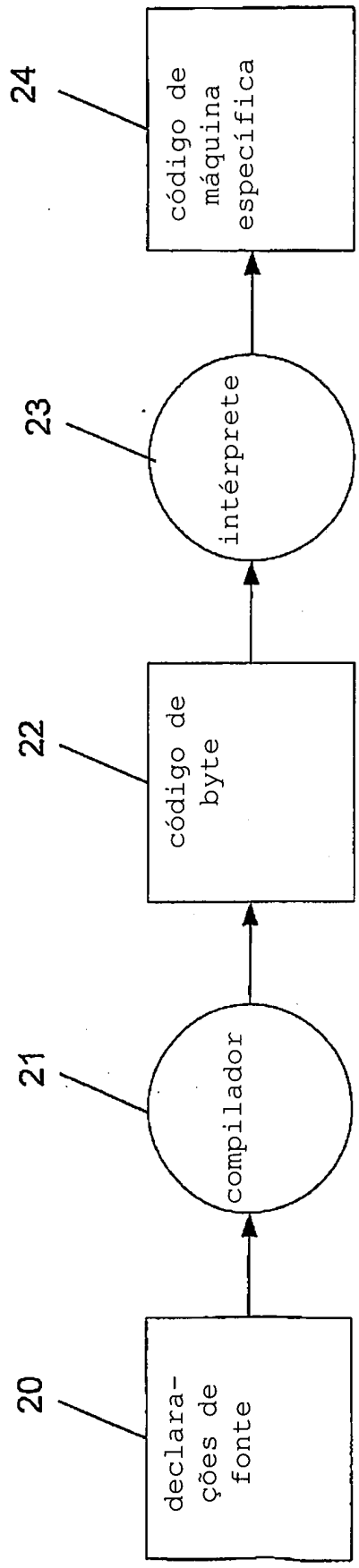


Fig. 2a
(estado da técnica)

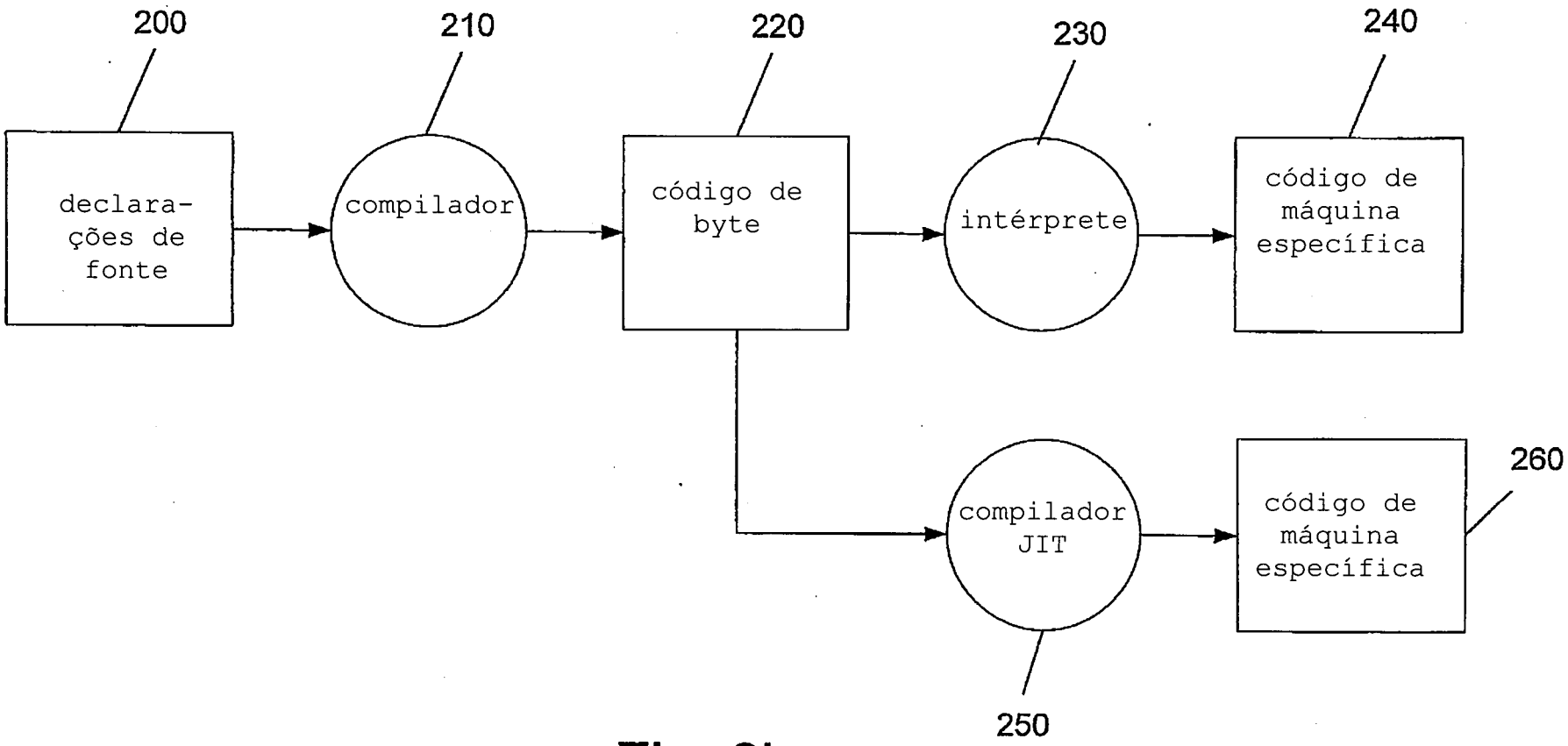


Fig. 2b

(estado da técnica)

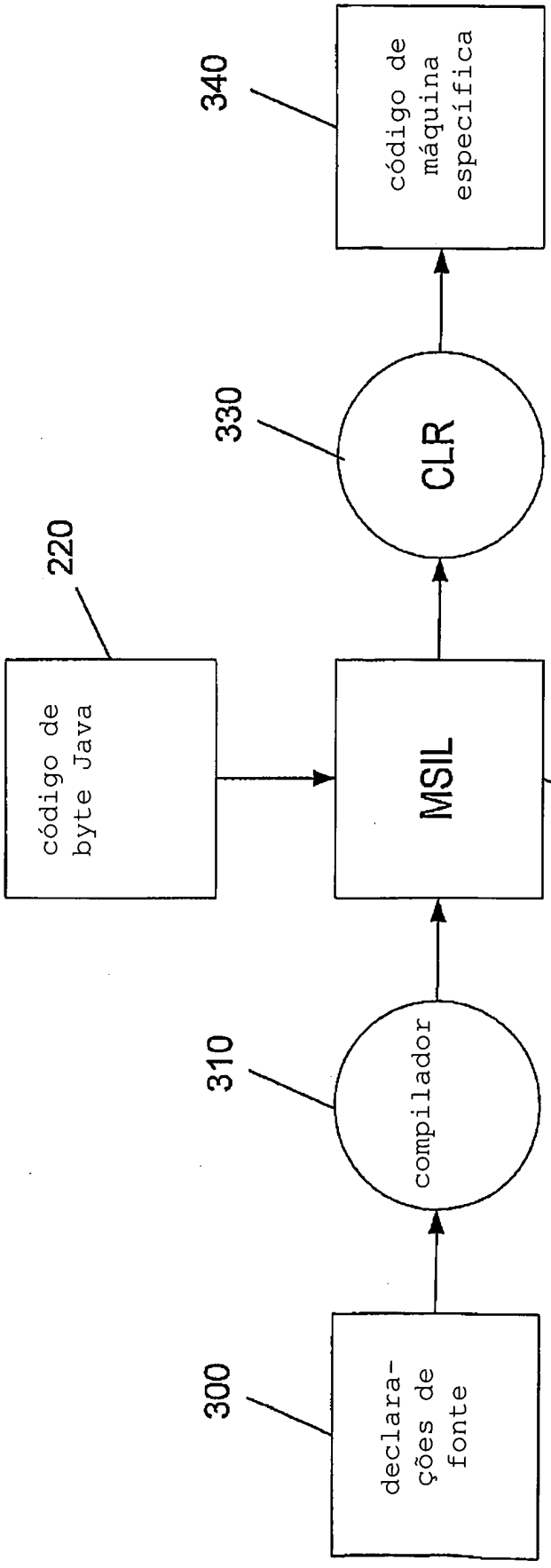


Fig. 3
(estado da técnica)

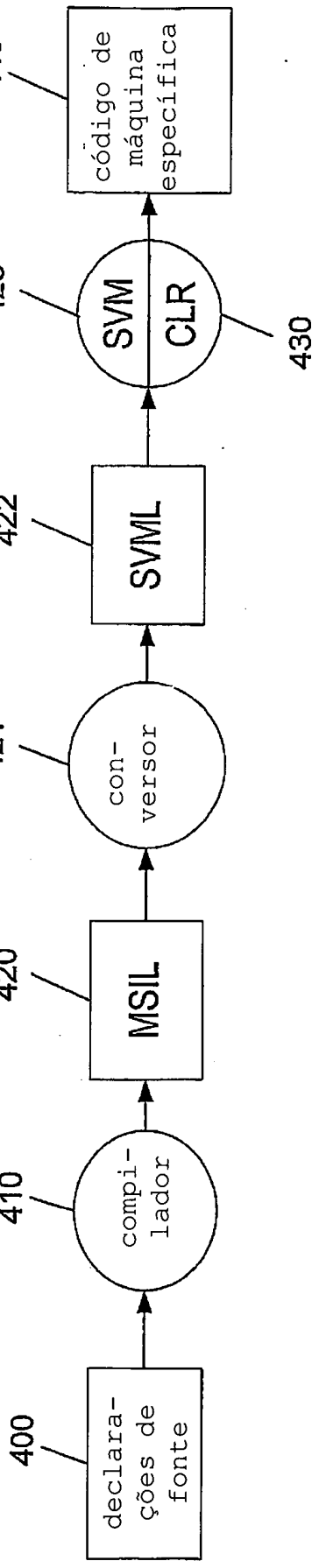

400
declara-
ções de
fonte
410
compi-
lador
420
MSIL
421
con-
versor
422
SVML
423
SVM
CLR
430
440
código de
máquina
específica

Fig. 4

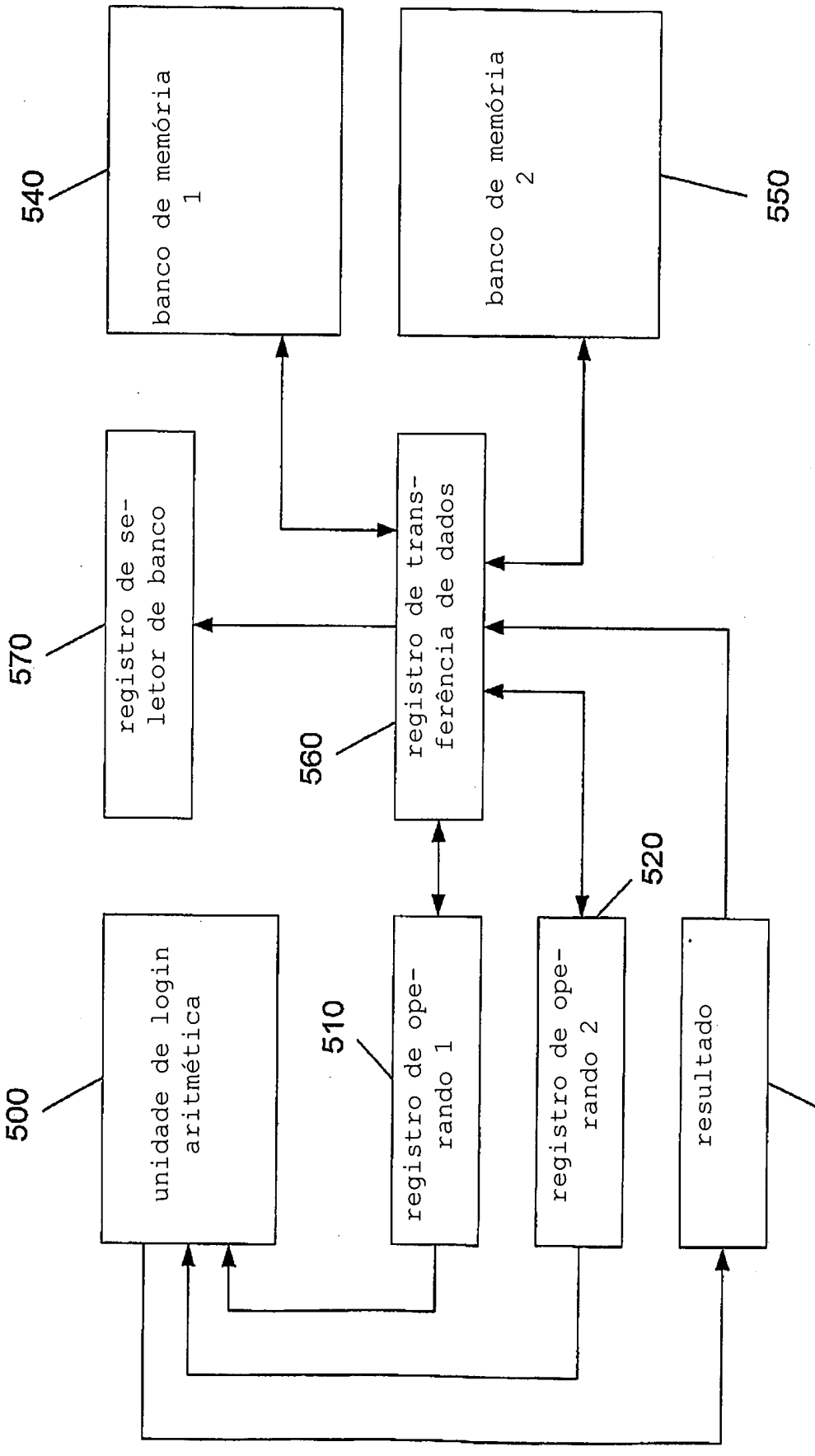

540
banco de memória 1
550
banco de memória 2
570
registro de seletor de banco
560
registro de transferência de dados
520
500
unidade de login aritmética
510
registro de operando 1
registro de operando 2
530
resultado

Fig. 5

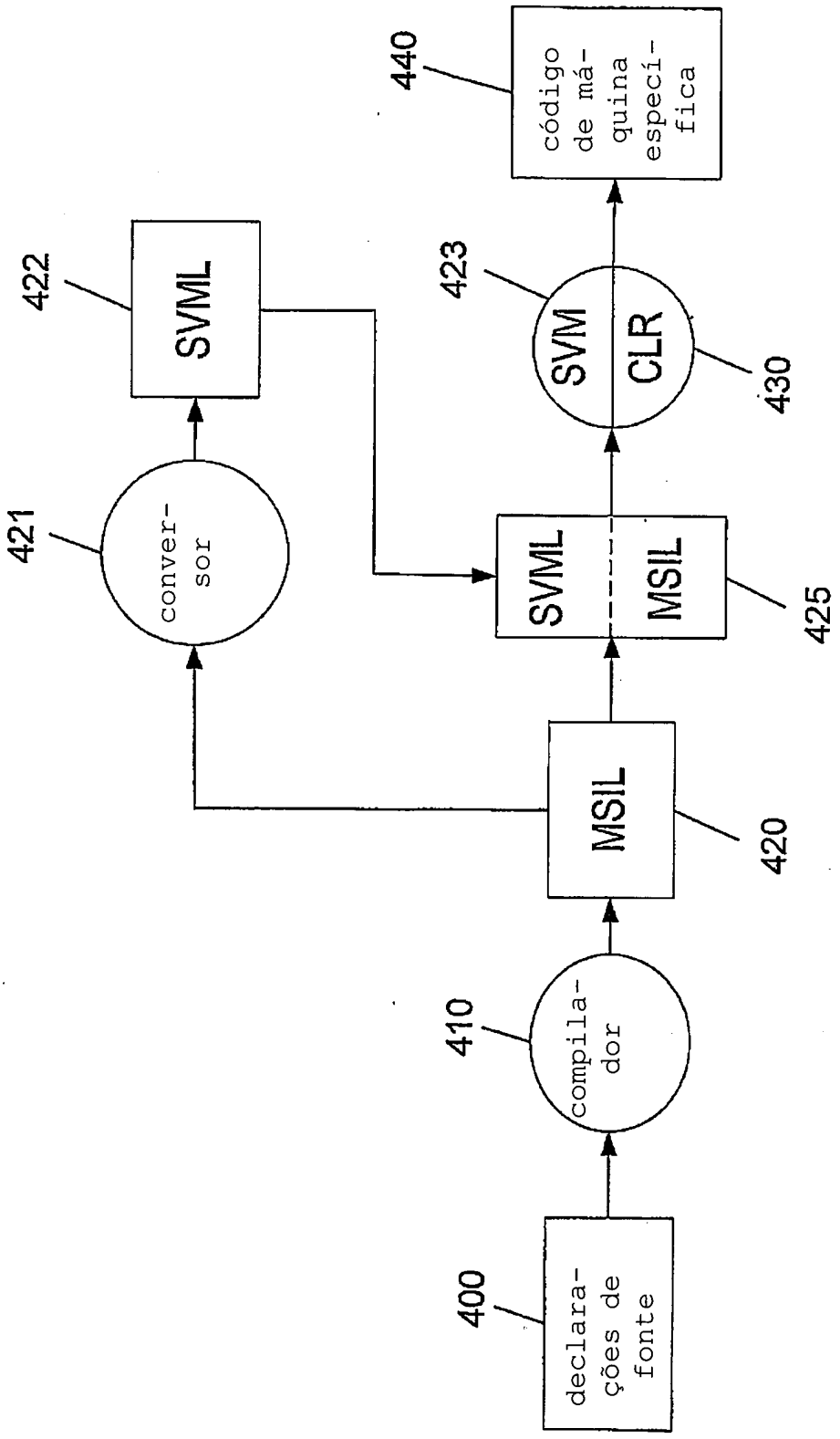

400
declara-
ções de
fonte
410
compila-
dor
420
MSIL
421
conver-
sor
422
SVML
423
SVM
CLR
430
425
SVML
MSIL
440
código
de má-
quina
especí-
fica

Fig. 6

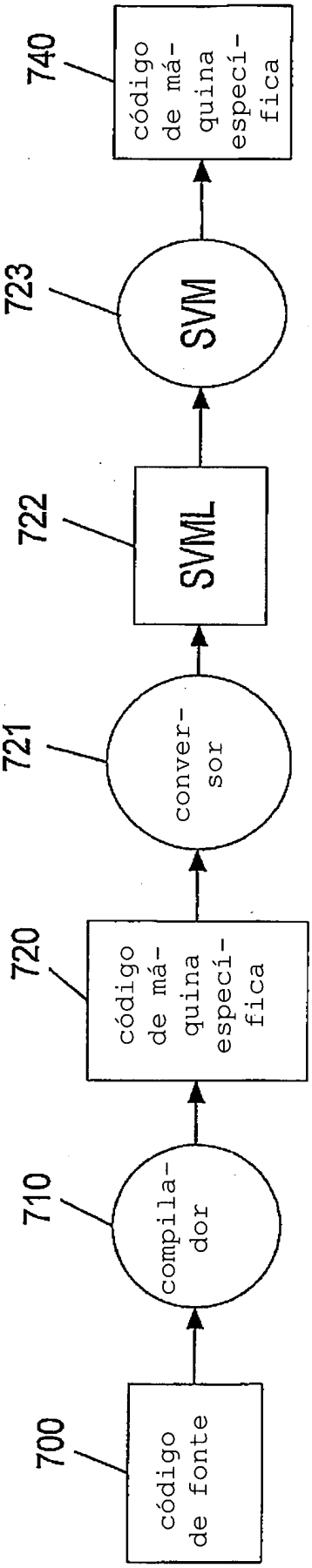

700
código de fonte
710
compila-dor
720
código de má-quina especí-fica
721
conver-sor
722
SVML
723
SVM
740
código de má-quina especí-fica

Fig. 7

PI0614089-0

RESUMO

"MÉTODO PARA EVITAR ENGENHARIA REVERSA DE SOFTWARE, MODIFICAÇÃO NÃO AUTORIZADA E INTERCEPTAÇÃO DE DADOS DE TEMPO DE EXECUÇÃO"

Um método para evitar a modificação desautorizada de um software ou modificação desautorizada de dados de tempo de execução. De acordo com esse método, um conversor, que é capaz de converter o software em um código de máquina generalizado é fornecido. O conversor é projetado de tal modo que não possa sofrer engenharia reversa, pelo uso de um processo de conversão que causa perda de dados. Um intérprete, cujo conhecimento de seu método de processo é mantido restrito, também é fornecido. O intérprete interpreta o código de máquina geral em um código de máquina específico, enquanto reconstrói os dados perdidos durante o processo de interpretação.