Detecção Automática de Vulnerabilidades em Código Protegido por Canários*

Izabela Karennina Travizani Maffra[1], **Fernando Magno Quintão Pereira** [1], **Leonardo Barbosa Oliveira** [1]

[1]Departamento de Ciência da Computação – UFMG

{karennina, fernando, leob}@dcc.ufmg.br

Resumo. *O estouro de arranjos é um tipo de ataque que custa, todo ano, milhões de dólares aos usuários de* software. *Canários são uma das formas mais conhecidas de prevenção contra esse tipo de ataque. Apesar de serem muito efetivos na prática, canários não são a solução ideal. Em alguns casos, eles não impedem a sobrescrita das chamadas variáveis locais. Tal omissão pode permitir que um adversário tome o controle do programa, desde que haja escalares posicionados após um arranjo vulnerável. A detecção desse tipo de vulnerabilidade é difícil, pois demanda grande familiaridade com o código sob análise. O objetivo deste artigo é resolver esse problema de forma automática. Tal feito pode ser conseguido via análises estáticas de código, implementadas a nível do compilador. Nossa técnica não requer qualquer tipo de intervenção do usuário e é notavelmente precisa. Esses algoritmos, implementados no compilador LLVM, foram testados em benchmarks que juntos nos deram mais de 1,4 milhões de linhas de código C. Nossa análise é prática e eficiente. Por exemplo, detectamos 836 potenciais vulnerabilidades em 17 programas presentes em SPEC CPU 2006.*

1. Introdução

Canários são um dos mecanismos mais conhecidos para impedir que usuários maliciosos tomem controle de um programa via ataques baseados em estouro de arranjo. Um canário é, em sua forma mais geral, uma constante que o compilador insere logo antes do endereço de retorno de uma função. Um estouro de arranjo é uma técnica de ataque de *software* que consiste em preencher um vetor com uma quantidade de bits maior que a sua capacidade. Ataques desse tipo permitem que adversários sobrescrevam dados de forma ilegal. Contudo, tentativas de sobrescrever o endereço de retorno de uma função protegida com canários irão inevitavelmente sobrescrever também essa constante. Em conjunto com o canário, o compilador também insere uma asserção ao final do código da função. Essa asserção verifica se a constante foi modificada, e, constatando tal fato, desvia o fluxo do programa para uma rotina de tratamento de erro, o que muitas vezes implica no encerramento do programa. Essa rotina de verificação pode ser implementada com um pequeno número de instruções *assembly*, o que implica num *overhead* praticamente desprezível. Testemunho da efetividade de canários é o fato de eles serem adotados por vários compiladores de C ou C++, tais como `gcc`, LLVM e `icc`.

Ainda que canários mostrem-se um mecanismo bastante efetivo de segurança computacional, eles não são a derradeira solução contra ataques de estouro de arranjos.

*Esse projeto é financiado pela Intel e pelo CNPq.

 ©2013 SBC — Soc. Bras. de Computação

Em particular, a proteção baseada em canários é ineficaz contra a sobrescrita de variáveis locais. Tais variáveis são armazenadas no registro de ativação das funções. Dependendo da maneira como o compilador dispõe dados em memória, é possível que um arranjo vulnerável a estouro de dados ocupe endereços inferiores a endereços de algumas daquelas variáveis. Essas variáveis podem, assim, ser modificadas por um adversário que seja capaz de criar os dados que serão armazenados no arranjo. Dois exemplos detalhados de tal vulnerabilidade são descritos na seção 2 do presente trabalho.

Este artigo descreve uma análise estática de código que identifica programas que apresentam a vulnerabilidade descrita acima. Nossa análise encontra funções que apresentam duas propriedades que, combinadas, podem permitir que um adversário venha a comprometer o valor das variáveis locais àquela função. Essas duas propriedades são descritas a seguir:

1. A função declara, estaticamente, um arranjo que pode ser escrito com dados provenientes de uma entrada à qual o adversário possui acesso.
2. A função contém variáveis locais que o compilador aloca em regiões posteriores àquela em que o arranjo vulnerável está alocado.

O primeiro desses itens existe devido à lógica usada para implementar o programa. Se um arranjo é usado para armazenar dados que um adversário pode manipular, e escritas nesse arranjo não são guardadas contra acessos fora de limite, como ocorre em linguagens fracamente tipadas como C, então esse arranjo é vulnerável. Nós identificamos esse tipo de situação via uma técnica conhecida como análise de fluxos contaminados [Denning and Denning 1977, Jovanovic et al. 2006, Rimsa et al. 2011]. O segundo de nossos itens é, muitas vezes, resultado de uma má alocação de dados feita pelo compilador. Em geral, compiladores previnem a segunda condição necessária para a existência de vulnerabilidade dispondo arranjos após outros tipos de dados. Contudo, há situações, conforme discutiremos na seção 2, em que tal cuidado é impossível. Nós identificamos essas situações observando o padrão de declarações de variáveis locais na pilha de funções.

A análise que propomos neste trabalho foi implementada no compilador LLVM [Lattner and Adve 2004]. LLVM é um compilador de C e C++ de qualidade industrial, atrás somente de `gcc` em número de usuários. A fim de validar nossa proposta, fomos capazes de analisar um conjunto de 445 programas bem conhecidos, com mais de 1,4 milhões de linha de código C. A ferramenta que criamos para validar nossa técnica é totalmente automática, e é capaz de apontar cadeias de dependências de dados, dentro do código fonte do programa, que um adversário pode usar para alterar informações críticas. Obtivemos 836 avisos de caminhos vulneráveis entre os 17 programas presentes em SPEC CPU 2006. A ferramenta, chamada `unsmasher` está publicamente disponível. Embora ela analise programas *assembly*, seus avisos referem-se a linhas de código C. Sua principal utilidade é indicar ao desenvolvedor possíveis vulnerabilidades a ataques de estouro de *buffer*. De posse desse conhecimento, o programador pode sanear código vulnerável via verificações de limites de arranjos.

2. Exemplo de Ataque a Código Protegido por Canários

A vasta maioria das linguagens de programação hoje disponíveis ao desenvolvedor de *software* utiliza a abstração de *funções* ou *procedimentos*. Uma função precisa de alguns dados para poder ser ativada. Esses dados, a saber, valores de parâmetros, variáveis locais e endereço de retorno são armazenadas em uma área conhecida como *registro de*

 ©2013 SBC — Soc. Bras. de Computação

ativação. Desde o surgimento de Algol, no final da década de 50, registros de ativação são armazenados em uma estrutura chamada *pilha*. Assim, a última função chamada é a primeira a retornar. Ao fim da execução de uma função f, o fluxo do programa é desviado para a instrução seguinte àquela que invocou f.

Um arranjo, ou *buffer*, é uma região contígua alocada em memória. Algumas linguagens, chamadas *fortemente tipadas*, garantem que acessos a essa região, sejam eles de escrita ou leitura, somente acontecerão dentro dos limites alocados pelo programa. Java, C# e Python são linguagens que pertencem a essa categoria. Existem, contudo, linguagens que não provêem tais garantias. Os dois membros mais notórios dessa segunda categoria são C e C++. O fato de que essas linguagens não verificam acessos fora de limites pré-alocados dá margem à existência de uma grande quantidade de vírus e *worms* que hoje contaminam milhões de máquinas em todo o mundo. O mecanismos básico de ataque desse tipo de *malware* é o *estouro de arranjo*. Um estouro de arranjo acontece quando um buffer é escrito com uma quantidade de dados maior que sua capacidade. Ao usar dados suficientes para sobrescrever o endereço de retorno de uma função, um adversário pode tomar conta de um programa. Esse ataque é feito, por exemplo, desviando-se o fluxo do programa para funções de sistema, tais como `telnet` ou `shell`.

O principal mecanismo de defesa contra ataques de estouro de arranjos são os canários. Um canário é uma constante, inserida logo antes do endereço de retorno de uma função. Tentativas de sobrescrever esse endereço terminam por modificar o canário. Antes de a função atacada retornar, o valor armazenado em seu canário é verificado. A constatação de alteração, naturalmente, leva à invocação de uma rotina de tratamento de erros. A figura 1 ilustra esses princípios. A constante `0x000aff0d` é conhecida como *canário terminador*. Ela contém `0x00`, um caractere que termina a função `strcpy`, e `0x0a`, que para `gets`, duas rotinas normalmente usadas para o preenchimento de arranjos em C. Qualquer constante diferente terminaria por disparar o código de erro, uma vez que `account` retornasse. Canários, todavia, não previnem certos tipos de ataques, os quais descreveremos a seguir.

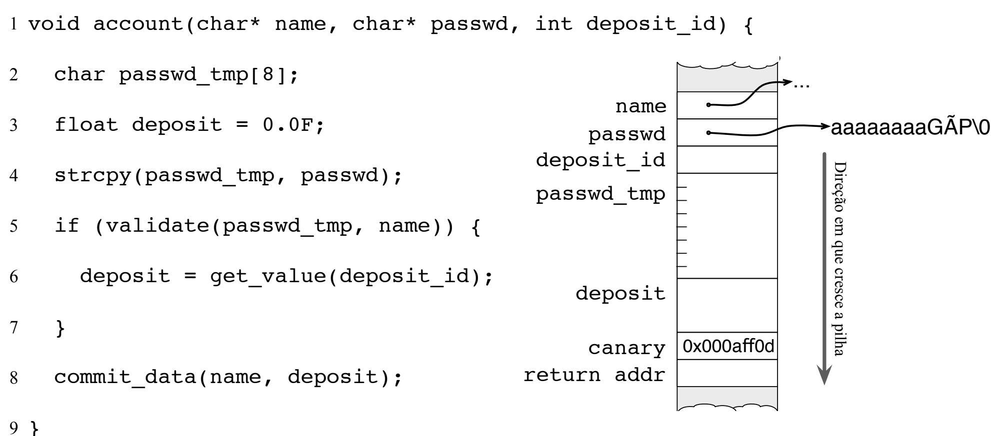

Figura 1. Um exemplo de código protegido por canário.

Exemplo de Código Vulnerável, ainda que Protegido por Canário. A figura 1 mostra um exemplo em que um usuário malicioso poderia sobrescrever uma informação sigilosa

do programa. Nesse caso particular, o adversário está interessado em modificar o valor da variável `deposit`, que, nesse exemplo hipotético, representa uma quantidade de dinheiro a ser transferida para a conta de um cliente. Ao fornecer uma senha incorreta, o adversário garante que o teste na linha cinco do exemplo não será verdadeiro. Por outro lado, essa mesma senha contém dados suficientes para sobrescrever o conteúdo de `deposit`. Tal sobrescrita ocorre por que essa variável – `deposit` – foi declarada logo após o arranjo `passwd_tmp`, usado para fazer a validação da senha. Vários compiladores de qualidade industrial, tais como LLVM 3.1, mantido pela Apple, respeitam a ordem de declaração de variáveis ao alocá-las em memória. Se a variável `deposit` for alocada antes do arranjo `passwd_tmp`, então o ataque não é possível.

O ataque anterior altera dados do programa vulnerável. Entretanto, alterações de seu fluxo de controle também são possíveis, ainda que o endereço de retorno da função esteja protegido. A figure 2 mostra um exemplo de tal situação. Nesse caso, um adversário pode alterar um ponteiro para função, fazendo com que ele aponte para uma das rotinas de sistema. Como o ponteiro de função sobrescrito, `hamming`, possui a mesma assinatura que a função `telnet`, disponível na biblioteca padrão C (`libc`), esse é o alvo do ataque. De posse do endereço dessa função, o adversário pode criar uma string grande o suficiente para preencher o arranjo comprometido – `tmp` nesse exemplo – contendo, ao final, o endereço da função a ser capturada. Quando o ponteiro `hamming` for derreferenciado, o fluxo de execução do programa será desviado para uma chamada de sistema, a qual dará ao usuário malicioso um terminal com os mesmos privilégios do programa atacado.

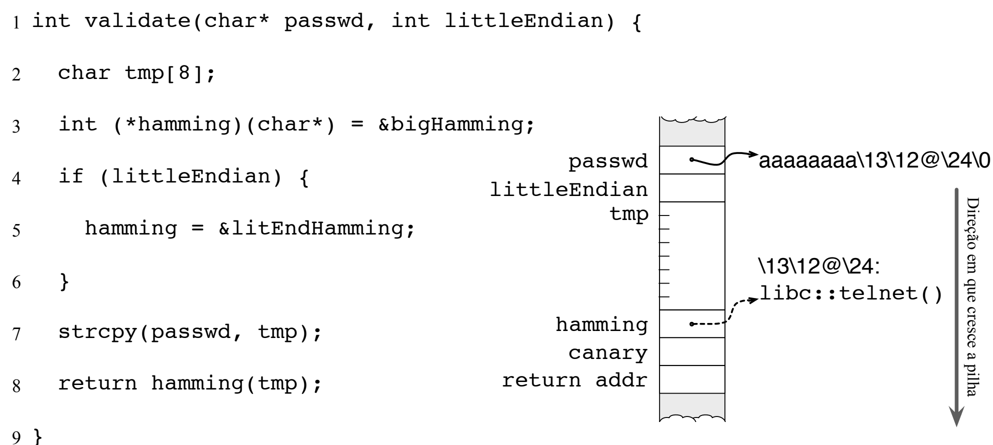

Figura 2. Outro exemplo de código vulnerável, mesmo com canários: o adversário pode tomar controle do sistema sobrescrevendo um ponteiro de função.

Os dois tipos de ataques vistos anteriormente podem ser prevenidos se o compilador alocar os arranjos depois dos escalares, i.e., os arranjos posicionam-se de tal forma a estarem o mais próximo possível do canário. Existem, contudo, situações onde essa solução não se aplica. Em particular, funções que declaram localmente dois arranjos são inerentemente vulneráveis se ambos os arranjos puderem ser escritos por um adversário. Além disso, arranjos declarados no escopo de registros (`structs`) não podem ser realocados pelo compilador. Tal impossibilidade existe por que o padrão C permite que programadores iterem sobre uma estrutura via aritmética de ponteiros, fazendo suposições sobre a ordem em que cada elemento daquele registro está disposto em memória.

 ©2013 SBC — Soc. Bras. de Computação

3. Detecção Estática de Código Vulnerável

Detectamos código vulnerável em três etapas. Primeiramente, é realizada a análise de ponteiros sobre o programa, cujo propósito é garantir a correção de nossos algoritmos. Em segundo lugar, é feita a análise de fluxos contaminados sobre o programa. Essa análise determina o conjunto de *arranjos alcançáveis*. Esses são buffers que podem ser escritos com os dados que um adversário consegue manipular. Finalmente, ocorre a análise de disposição de dados: o código intermediário é percorrido em busca de variáveis locais declaradas antes de arranjos alcançáveis. Essa seção explica cada uma dessas fases.

3.1. Análise de Ponteiros

Esta etapa determina quais ponteiros dereferenciam regiões de memória que se sobrepõem. Seu papel é capturar dependências que a sintaxe do programa não mostra, evitando, assim, *falsos negativos*. Um falso negativo ocorre se indicamos, erroneamente, que um arranjo é seguro. O programa ao lado ilustra essa situação. O arranjo local `a` recebe informação proveniente do arranjo `in`, passado como entrada da função `f`. Contudo, uma busca sintática no programa não revela esse fato, uma vez que `a` é escrito via o ponteiro `b`, seu sinônimo.

```
1 void f(char* in) {
2   char a[SIZE];
3   char* b = a;
4   strcpy(b, in);
5   printf(b);
6 }
```

A literatura de linguagens de programação descreve uma gama diversa de algoritmos que encontram ponteiros sobrepostos. Nós optamos por usar a análise de *Andersen* [Andersen 1994], pois ela nos dá um compromisso razoável entre precisão e eficiência. A fim de torná-la mais eficiente, nós usamos a heurística de *eliminação tardia de ciclos*, proposta por Hardekopf e Lin em 2007 [Hardekopf and Lin 2007]. Uma vez que esses algoritmos são bem estabelecidos na literatura, omitimos detalhes de sua implementação neste trabalho.

3.2. Análise de Fluxo Contaminado

A análise de fluxos contaminados busca encontrar quais variáveis presentes no texto do programa são dependentes de entradas que um adversário pode manipular. Análises desse tipo já foram descritas na literatura anteriormente, em geral com o propósito de encontrar vulnerabilidades a ataques do tipo injeção de código SQL [Tripp et al. 2009]. A principal ferramenta usada para a realização dessa análise é o grafo de dependências de dados, uma estrutura originalmente descrita por Ferrante [Ferrante et al. 1987]. O grafo de dependências G foi construído de tal forma que $G = (V, E)$ é um conjunto de vértices V, no qual cada vértice é uma variável, um conjunto de posições de memória ou uma operação. Arestas, presentes em E, conectam variáveis e blocos de memória aos nós de operação. Uma aresta denota uma dependência de dados. Para cada instrução i que define uma variável v e utiliza uma variável u, existe um arco $u \to i$ e um arco $i \to v$. Notem que essas variáveis podem ser também blocos de memória. Todo arranjo, por exemplo, é um bloco de memória. A figura 3 mostra um exemplo de programa, e o correspondente grafo de dependências. Devido à nossa análise de ponteiros, um mesmo bloco de memória pode representar diversos nomes diferentes de arranjos, como `argv[2]` e `passwd` na figura 3.

 ©2013 SBC — Soc. Bras. de Computação

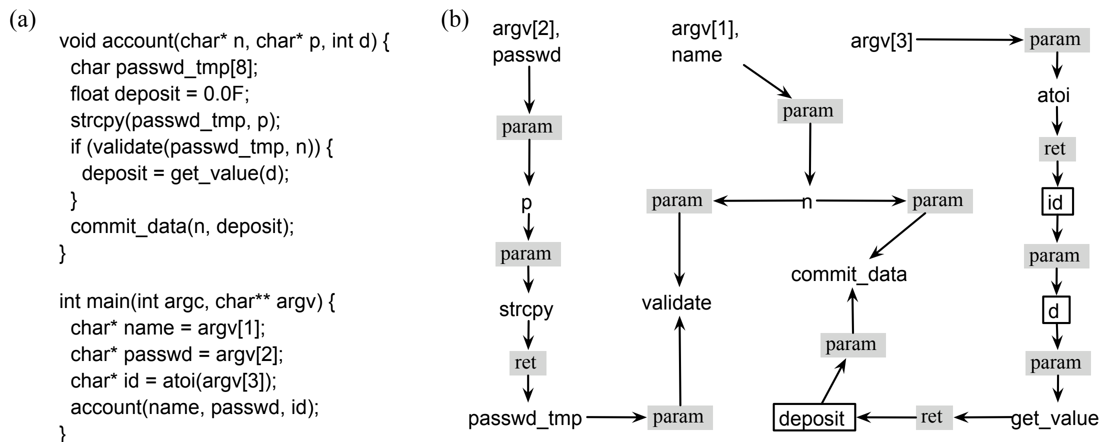

Figura 3. (a) Um programa C completo. (b) Seu grafo de dependências. Nós de operação são marcados em cinza, e nós de variáveis aparecem em caixas.

As entradas de dados são os elementos que um adversário pode usar para forçar estouros de *buffer* em programas. Uma questão importante que precisamos abordar é: "quais funções e variáveis constituem entrada de dados". A análise de fluxos contaminados irá procurar, no grafo de dependências dos programas, caminhos que conectem essas funções de entrada aos arranjos declarados localmente em funções. Neste trabalho nós consideramos entradas de dados os seguintes elementos:

- os argumentos do método `main`, isto é, as variáveis `argc` e `argv`;
- o resultado retornado por funções *externas*;
- ponteiros passados como argumento de funções *externas*.

Como funções externas, poderiam ser consideraradas todas as funções definidas externamente aos arquivos que compõem o programa compilado. Entretanto, esta abordagem mostrou-se ineficiente no sentido de que a frequência de funções de biblioteca é bastante alta, mas a maioria delas não representa uma real possibilidade de injetar dados em um programa. Buscando-se então um compromisso entre segurança e relevância dos resultados, optou-se por identificar uma lista de funções que permitem uma interação com o usuário que necessariamente lhe confere a capacidade de entrar com dados maliciosos. As funções utilizadas foram: `scanf`, `fscanf`, `gets`, `fgets`, `fread`, `fgetc`, `getc`, `getchar`, `recv`, `recvmsg`, `read`, `recvfrom`, `fread`.

A análise de fluxos contaminados começa por identificar, no grafo de dependências $G = (V, E)$, o conjunto $V_i \subseteq V$ de nós que representam entradas de dados. A seguir, essa análise identifica o conjunto $V_s \subseteq V$ de nós que representam arranjos declarados localmente em funções. Finalmente, para cada par $(i, s), i \in V_i, s \in V_s$, a análise de fluxos contaminados verifica se G contém algum caminho de i até s. O subgrafo G_{is} de G, formado por todos os caminhos de i até s constitui um *relatório de vulnerabilidade*. Dessa forma, a complexidade de nossa análise de fluxos contaminados é $O(|E| \times |V_i| \times |V_s|)$.

A nossa análise de fluxos contaminados é *interprocedural*. Isso significa que ela analisa o programa como um todo, em vez de ater-se a cada função em separado. Obtemos a interproceduralidade via a solução trivial de adicionar dependências entre parâmetros reais de funções, e seus parâmetros formais. Na figura 3, essas dependências existem, por

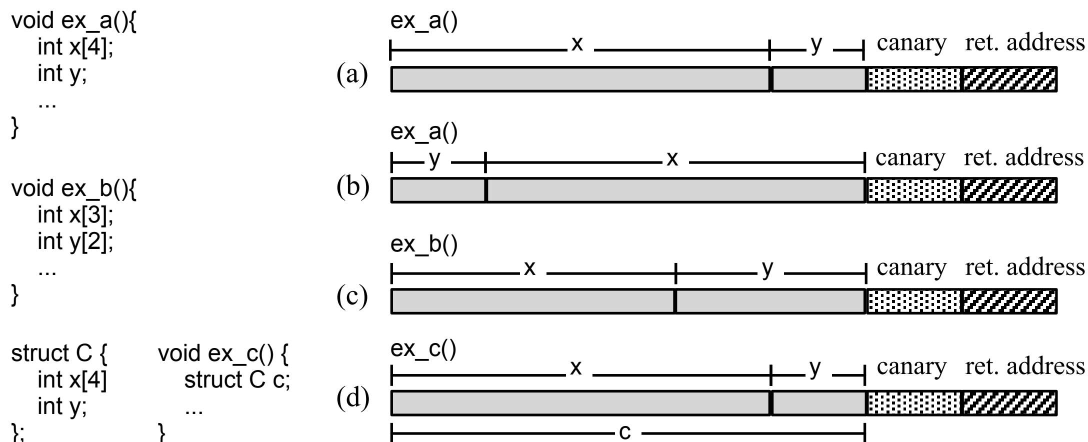

Figura 4. (a) Disposição insegura de dados: estouro de `x` pode sobrescrever variável local `y`. (b) Disposição segura de dados: arranjo alcançável é colocado após escalares. (c) Situação inerentemente vulnerável com dois arranjos. (d) Situação inerentemente vulnerável com arranjo em registro.

exemplo, entre `name`, um parâmetro real, e `n`, um parâmetro formal da função `account`. A interproceduralidade torna nossa análise muito mais efetiva, pois lhe confere uma visão holística do programa. Concluindo nosso exemplo, a cadeia que vai de `argv[2]` até `passwd_tmp` constitui um caminho vulnerável.

3.3. Análise de Disposição de Dados

A última etapa de nosso método analisa o código de cada função compilada, buscando por declarações de dados *inerentemente vulneráveis*. Dizemos que uma função declara dados inerentemente vulneráveis quando ela encaixa-se em pelo menos um dos casos abaixo:

1. A função declara localmente dois ou mais arranjos alcançáveis a partir de entrada. Um ou mais desses arranjos podem ser declarados no escopo de registros (`struct`).
2. A função declara localmente um registro que contém pelo menos um arranjo alcançável antes da declaração de alguma outra área de dado.

Esses cenários são perigosos porque não há forma de dispor dados que venha a impedir a sobrescrita das variáveis locais. A figura 4 ilustra cada um desses casos. A função `ex_a` declara localmente um arranjo `x` e uma variável local `y`. O compilador pode armazenar `x` após `y`, de forma que estouros no arranjo não venham a sobrescrever o escalar. Por outro lado, tal solução não é possível em `ex_b`. Se o arranjo `x` é alcançável, então escritas além de seu limite poderão comprometer os dados do arranjo `y`. Finalmente, a função `ex_c` declara um registro localmente. O compilador não tem liberdade para rearranjar dados em uma estrutura. O padrão ANSI C reza que esses dados sejam dispostos, na área alocada para a estrutura, na ordem em que eles são declarados. Portanto, nossa análise de disposição de dados apontaria as funções `ex_b` e `ex_c` como inerentemente vulneráveis.

4. Resultados Experimentais

Nós implementamos nossas técnicas no compilador LLVM, versão 3.1, disponível em Maio de 2013. Os experimentos foram realizados no sistema operacinal Linux Ubuntu

 ©2013 SBC — Soc. Bras. de Computação

12.04, sobre uma CPU de quatro núcleos, modelo Intel Core i5 com clock de 1.7GHz e 3.7GB de RAM. Testamos nossa análise sobre os benchmarks inteiros presentes em SPEC CPU 2006, um benchmark tradicionalmente utilizado na área de compiladores. Nós a usamos também sobre os 435 programas que compõem o arcabouço de testes de LLVM. No total, compilamos e analisamos mais de 1,4 milhões de linhas de código C. Esta seção discute os resultados obtidos.

Proporção de Funções Vulneráveis: A quantidade de funções inerentemente vulneráveis, segundo a definição da seção 3.3 é relativamente alta. Essa é uma observação surpreendente, pois poder-se-ia pensar que as propriedades enumeradas na seção 3.3 são muito restritivas. A figura 5 mostra esse fato, para os programas em SPEC 2006. Esse gráfico classifica todas as funções encontradas nos programas analisados em uma de três categorias. Funções que contém *somente escalares* não podem sofrer ataques de estouro de arranjos, uma vez que elas não possuem *buffers* declarados localmente. Funções que contém *arranjos locais* podem ser protegidas por uma disposição adequada de dados, em que o compilador coloca *buffers* após as variáveis escalares. Finalmente, as funções *inerentemente vulneráveis* possuem registros com arranjos, ou duas declarações locais de arranjos. *Benchmarks* que utilizam arranjos intensivamente, como `444.namd`, tendem a apresentar maior proporção de funções vulneráveis. Mesmo programas com número muito grande de funções apresentam alto índice de rotinas inerentemente vulneráveis. Essa análise nos leva a concluir que conjunturas como aquelas descritas na seção 3.3 são comuns. Portanto, elas precisam ser levadas em consideração para o desenvolvimento de programas seguros.

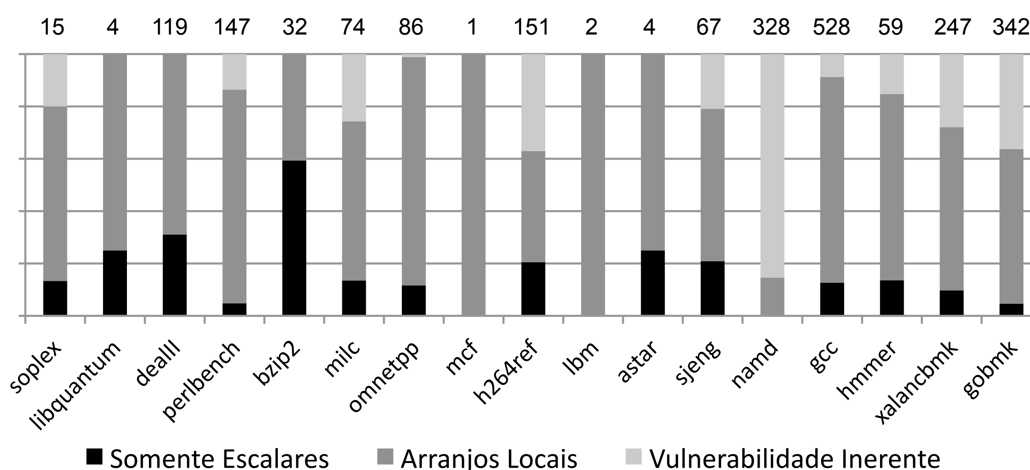

Figura 5. Proporção de funções vulneráveis encontradas nos programas presentes em SPEC CPU 2006. Valores sobre as barras informam o número total de funções em cada *benchmark*.

A próxima pergunta que julgamos relevante é: “qual a natureza das vulnerabilidades inerentes?” Em outras palavras, tais vulnerabilidades devem-se a múltiplas declarações de arranjos alcançáveis, ou a arranjos alcançáveis declarados em registros? A figura 6 responde essa questão. A figura mostra, para cada benchmark no qual foi encontrada alguma vulnerabilidade, a sua natureza. O caso de *buffers* declarados contiguamente é o mais frequente. Por outro lado, vulnerabilidades devido a registros não

 ©2013 SBC — Soc. Bras. de Computação

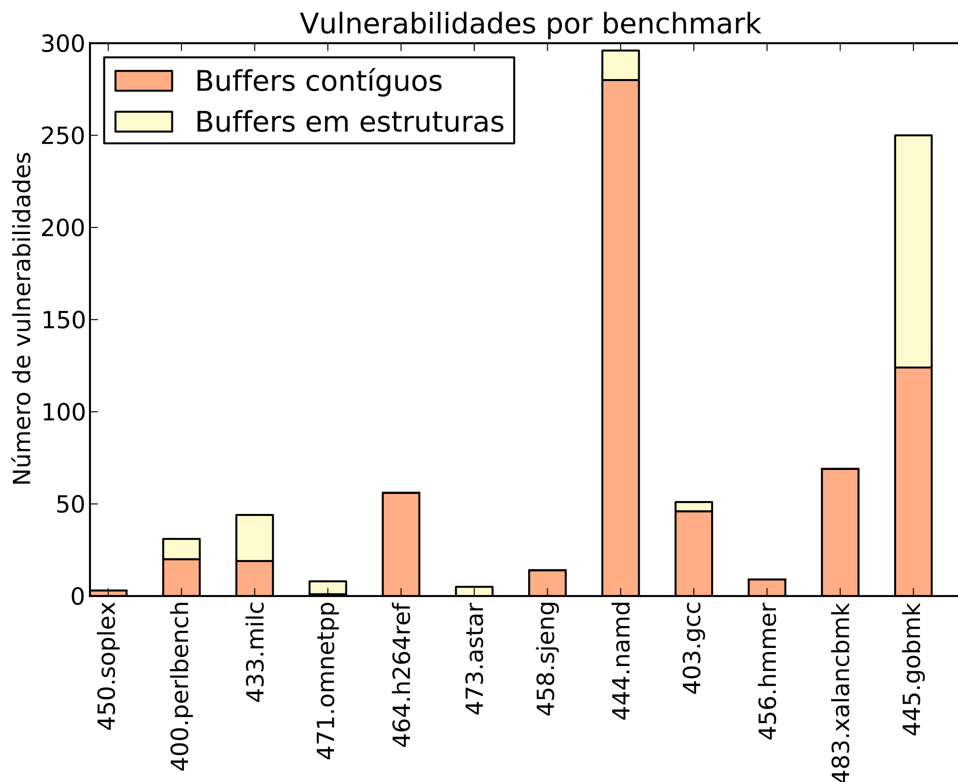

Figura 6. Natureza das vulnerabilidades encontradas.

podem ser desprezadas. Por exemplo, `445.gobmk` possui 123 funções que declaram, localmente, registros contendo arranjos dispostos antes de outros dados.

Análise de Complexidade. A figura 7 compara o tempo necessário para executar nossa análise com o tamanho do programa de entrada. Nesse experimento vários *benchmarks* presentes no arcabouço de testes de LLVM, além dos programas em SPEC CPU 2006. A maior quantidade de pontos permite-nos estimar a complexidade de nosso algoritmo empiricamente. Mostramos resultados para os 100 *benchmarks* que contêm arranjos declarados localmente cuja análise demorou mais tempo. Como medida de tamanho do programa, consideramos o produto entre o número de *buffers* que ele contém, e o seu número de variáveis. A quantidade de *buffers* afeta o número de buscas que são disparadas no grafo de dependências durante nossa análise de fluxos contaminados. O número de variáveis dá-nos uma medida do tamanho do grafo que precisa ser atravessado. Cada ponto do eixo x da figura 7 corresponde a um benchmark. O menor dentre esses benchmarks, `Trimaran/enc-rc4`, tem 113 variáveis e é analisado em apenas 32ms. O maior benchmark, `403.gcc`, que possui um grafo de dependências com 287.889 variáveis, é analisado em 66 minutos.

Para esse conjunto de dados, o coeficiente de determinação R^2 é 0,923, indicando fortemente que a análise possui complexidade assintótica proporcional ao produto entre *buffers* e número de variáveis. Note que nossa análise responde à pergunta: “quais pares, função de entrada $\times$ arranjo, são vulneráveis?” Se quiséssemos responder somente à

 ©2013 SBC — Soc. Bras. de Computação

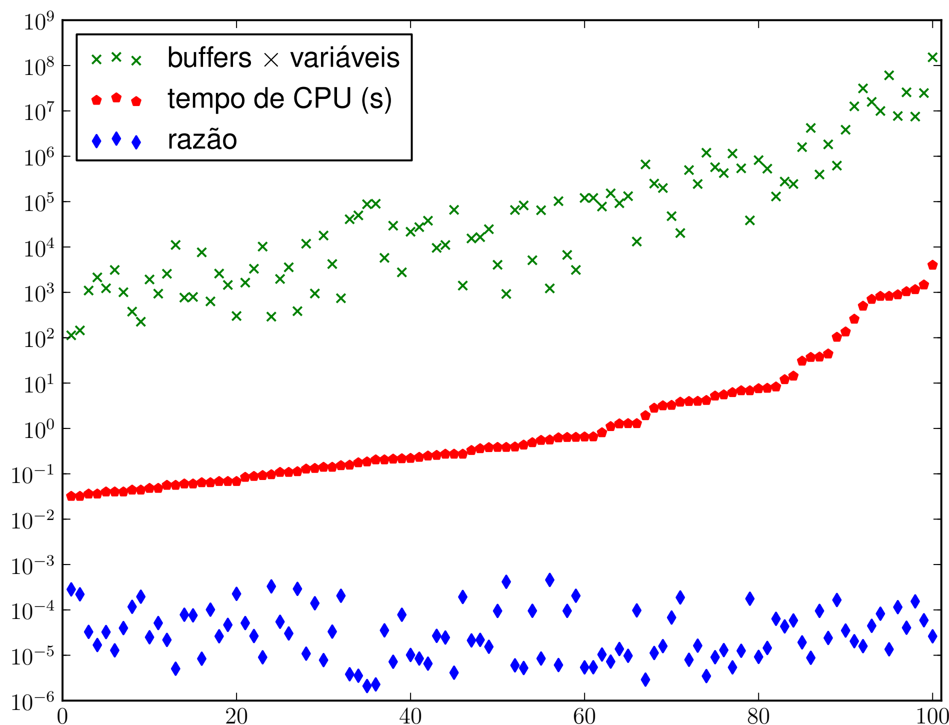

Figura 7. Análise empírica da complexidade de nosso algoritmo.

pergunta: "quais arranjos podem ser alcançados a partir de funções de entrada?" então teríamos um algoritmo linear no tamanho do grafo de dependências do programa. Cada vértice visitado, nesse caso, não precisaria ser visitado novamente.

A Estrutura do Grafo de Dependências. A tabela 1 provê estatísticas referentes à estrutura dos grafos de dependência que encontramos. O tamanho desses grafos é, evidentemente, proporcional ao tamanho do programa analisado, em número de instruções. Ressalta-se, entretanto, que várias instruções, como desvios condicionais e incondicionais, não contribuem para a construção desses grafos. Nossos grafos possuem mais vértices que representam operações que vértices que representam variáveis, pois a mesma variável pode ser usada em várias instruções, e nem toda instrução define uma variável. Instruções de carregamento (`store` e suas variações) são o melhor exemplo desse tipo de situação. Nossos grafos tendem a ser esparsos. Em geral temos 1,56 arestas por vértice. Esse fato é consequência direta do fato da maioria dos nós de operação terem somente uma aresta de saída, e no máximo duas de entrada. Note que não existem arestas ligando duas variáveis diretamente. Um último fato de nota é a grande quantidade de sorvedouros, isto é, operações de escrita em arranjos, quando comparado ao número de entradas. Entradas são funções que lidam com a interface entre usuário e programa. É natural que existam em menor número que as operações que carregam dados em arranjos.

Complexidade dos avisos reportados por nossa análise. Nossa análise informa ao desenvolvedor quais caminhos existem entre função de entrada e operação de escrita em

 ©2013 SBC — Soc. Bras. de Computação

benchmark	Instruções	Nós de variáveis	Nós de operações	Nós de memória	arestas	Ent.	Sourv.
mcf	2.556	922	2.013	905	6.013	5	0
lbm	4.100	3.341	3.868	628	10.629	2	0
libquantum	6.435	3.294	5.094	1.048	15.079	15	4
astar	8.646	4.121	6.657	2.006	20.464	41	8
omnetpp	91.146	48.895	67.872	31.971	237.635	7	31
soplex	63.044	26.403	45.082	22.300	152.435	2	52
hmmer	67.528	30.990	52.991	21.252	149.514	116	82
xalancbmk	588.502	246.341	434.670	251.631	1.459.581	4	107
milc	24.212	11.103	19.442	6.913	57.791	2	121
sjeng	30.012	17.850	22.438	2.536	62.058	5	135
bzip2	17.324	8.641	14.341	4.428	41.296	3	141
dealII	454.530	216.101	333.336	148.751	1.201.739	2	190
perlbench	283.594	107.460	214.986	100.107	606.512	14	214
gcc	808.795	287.889	607.428	328.236	1.798.105	8	491
gobmk	144.267	91.670	106.493	21.628	329.731	9	498
namd	68.820	38.424	59.720	12.006	161.724	2	806
h264ref	143.804	65.946	121.081	45.230	338.909	11	871

Tabela 1. Estatísticas relacionadas aos grafos de dependência.

arranjo. Esses caminhos são medidos em instruções, mas são reportados em linhas de código. LLVM permite-nos obter a linha de código que corresponde a uma certa instrução *assembly*. Idealmente gostaríamos que tais caminhos fossem curtos o suficiente para que pudessem ser inspecionados por desenvolvedores de *software*. A figura 8 mostra o tamanho desses caminhos. Um caminho vulnerável possui em média 22 instruções *assembly*. Cada instrução desse tipo corresponde a no máximo uma linha de código no programa fonte. Note, contudo, que a mesma linha pode ser reponsável por várias instruções. Para auxiliar o usuário, a ferramenta desenvolvida reporta também quais são os menores caminhos entre cada entrada e cada *buffer* vulnerável. Nesse gráfico, cada círculo (x, y, R) é determinado de tal forma que x representa um benchmark, y representa o tamanho médio dos caminhos mínimos entre uma vulnerabilidade e cada uma das entradas que a alcançam e R representa o número de vulnerabilidades reportadas com tal propriedade. No benchmark 444.namd, por exemplo, foram encontradas mais de 200 vulnerabilidades com caminho médio de tamanho igual a 30.

5. Trabalhos Relacionados

O problema discutido neste artigo – a existência de variáveis locais vulneráveis, apesar da presença de canários – insere-se no amplo contexto de vulnerabilidades relacionadas a estouro de arranjos na pilha. Esse problema se popularizou especialmente após a publicação do artigo *Smashing the Stack for Fun and Profit* [Levy 1996]. Esse artigo descreve de maneira detalhada como usar um estouro de arranjos em um programa C para sobrescrever o endereço de retorno de uma função com instruções maliciosas. Mais estreitamente relacionado ao nosso trabalho, Richarte mostra um ataque que sobrescreve o ponteiro de quadro (uma variável local), para contornar de maneira efetiva a proteção baseada em canários que é oferecida pelo StackGuard [Richarte 2002].

 ©2013 SBC — Soc. Bras. de Computação

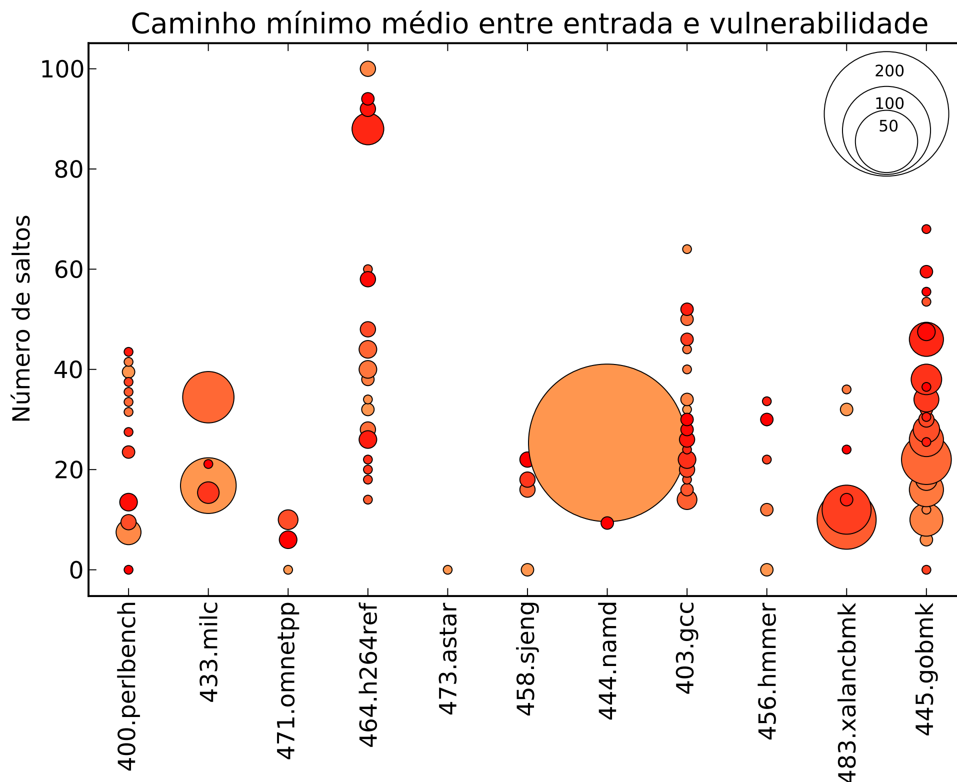

Figura 8. Tamanho de caminhos vulneráveis. O eixo y define quantas instruções existem por caminho. O tamanho de cada círculo informa o número de caminhos encontrados com um dado tamanho.

Mesmo sendo um problema antigo, ataques de estouro de arranjos são ainda tópico de pesquisa. Diferentes análises estáticas já foram desenvolvidas com o objetivo de identificar códigos vulneráveis. Para um apanhado delas, recomendamos a seção de trabalhos relacionados escrita por Shahriar *et al.* [Shahriar and Zulkernine 2010]. A maioria dessas análises tem como alvo a linguagem C, dada sua popularidade, e sua tipagem insegura. As primeiras análises propostas na literatura baseiam-se em *casamento de padrões*. De maneira geral, dado um conjunto de funções de biblioteca que conhecidamente geram vulnerabilidades, essas ferramentas percorrem o programa buscando por chamadas dessas operações. Entre os exemplos de analisadores nessa categoria, temos o trabalho de Vietas *et al.* [Viega et al. 2002] e FlawFinder [1]. Buscas também podem dar-se diretamente sobre o programa executável, como demonstrado por Tevis *et al.* [Tevis and Hamilton 2006]. De maneira geral, essas ferramentas têm uma capacidade de detecção inferior à que desenvolvemos, já que a análise restringe-se a padrões previamente identificados como causadores de vulnerabilidades. Assim, elas não realizam um estudo mais minucioso sobre os possíveis caminhos entre fontes de ataques e operações críticas no programa.

Uma outra estratégia usada no saneamento de código C é a análise baseada em restrições. Um parser gera essas restrições a partir de anotações que o programador insere no código auditado, como demonstrado - de forma independente - por

[1] `http://www.dwheeler.com/flawfinder/`

©2013 SBC — Soc. Bras. de Computação

Dor *et al.* [Dor et al. 2003], Evans *et al.* [Evans and Larochelle 2002] e Hackett *et al.* [Hackett et al. 2006]. Uma ferramenta automática, como a que desenvolvemos, oferece vantagens quando comparada com essas alternativas, pois requer muito menos esforço por parte do usuário. Além disso, nossa análise não depende da corretude e completude das informações que o usuário possui. É possível, contudo, que restrições sejam geradas automaticamente. Essa geração automática foi feita, por exemplo, por Weber *et al.* [Weber et al. 2001]. Entretanto, essa ferramenta, ao contrário de nosso trabalho, não dispõe de uma análise de *alias*. Conforme apontado pelos autores, essa omissão pode levar a falsos negativos.

Análises de fluxo contaminado similares à que empregamos já foram utilizadas na literatura, para detectar vulnerabilidades diversas, não se limitando ao estouro de arranjos. Em [Quadros and Pereira 2011], por exemplo, uma análise de fluxo contaminado é utilizada na detecção de vazamento de endereços. Já no trabalho de Sotirov [Sotirov 2005], é demonstrada uma análise mais similar à que apresentamos, sendo também identificados fluxos contaminados que permitam a um adversário sobrescrever arranjos com dados provenientes de uma entrada. Entretanto, tampouco este trabalho ou qualquer dos citados acima buscava vulnerabilidade em código protegido com canários. Embora a generalidade seja importante, nosso trabalho tem cunho bastante prático. Podemos apontar falhas de segurança em programas que não passaram por uma análise estática para detectar vulnerabilidades, mas que ainda assim são considerados seguros devido ao emprego de uma proteção dinâmica – o canário.

6. Conclusões e Observações Finais

Este artigo apresentou uma ferramenta que emprega um conjunto de análises de código com o intuito de detectar programas vulneráveis a ataques de estouro de arranjos. Essas vulnerabilidades existem mesmo em programas protegidos por canários. As nossas análises são práticas o suficiente para serem empregadas em um ambiente de produção industrial. Prova disso são os extensivos experimentos que pudemos apresentar neste trabalho. Fomos capazes de detectar vulnerabilidades não conhecidas, até mesmo em programas de uso estabelecido, como aqueles presentes em SPEC CPU2006.

A nossa técnica, embora efetiva e útil, possui duas limitações. A primeira delas é o fato de precisarmos do código fonte dos programas que analisamos. Essa não é uma limitação algorítmica, entretanto; trata-se mais de um artefato de nossa implementação. Desenvolvemos nossas idéias sobre um compilador, o qual, naturalmente, recebe código fonte, e não binários executáveis. A segunda de nossas limitações são os falsos positivos. A análise de fluxos contaminados, descrita na seção 3.2, não leva testes de limites de arranjos em consideração. Um teste desse tipo verifica se os valores usados para indexar arranjos são menores que a área de memória alocada para eles. Assim, muito embora arranjos sejam alcançáveis, essas guardas os tornam invulneráveis a ataques de estouro. Nosso objetivo imediato, a partir deste trabalho, é incorporar esses testes em nossa análise de fluxos contaminados, tornando os resultados reportados ao usuário mais precisos.

Reproducibilidade: a nossa ferramenta de detecção de vulnerabilidas está disponível em repositório público: `http://code.google.com/p/ecosoc/`. Esse repositório também contém vários exemplos de usos, e um tutorial sobre como usá-la.

 ©2013 SBC — Soc. Bras. de Computação

Referências

Andersen, L. O. (1994). *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen.

Denning, D. E. and Denning, P. J. (1977). Certification of programs for secure information flow. *Commun. ACM*, 20:504–513.

Dor, N., Rodeh, M., and Sagiv, M. (2003). CSSV: Towards a realistic tool for statically detecting all buffer overflows in c. In *PLDI*, pages 155–167. ACM.

Evans, D. and Larochelle, D. (2002). Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19:2002.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349.

Hackett, B., Das, M., Wang, D., and Yang, Z. (2006). Modular Checking for Buffer Overflows in the Large. In *ICSE*, pages 232–241. ACM.

Hardekopf, B. and Lin, C. (2007). The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299. ACM.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Symposium on Security and Privacy*, pages 258–263. IEEE.

Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.

Levy, E. (1996). Smashing the stack for fun and profit. *Phrack*, 7(49).

Quadros, G. S. and Pereira, F. M. Q. (2011). Static detection of address leaks. In *SBSeg*, pages 23–37.

Richarte, G. (2002). Four Different Tricks to Bypass StackShield and StackGuard Protection. *World Wide Web*, 1.

Rimsa, A. A., D'Amorim, M., and Pereira, F. M. Q. (2011). Tainted flow analysis on e-SSA-form programs. In *CC*, pages 124–143. Springer.

Shahriar, H. and Zulkernine, M. (2010). Classification of Static Analysis-Based Buffer Overflow Detectors. In *SSIRI-C*, pages 94–101. IEEE Computer Society.

Sotirov, A. (2005). Automatic Vulnerability Detection Using Static Source Code Analysis. Technical report.

Tevis, J.-E. J. and Hamilton, Jr., J. A. (2006). Static Analysis of Anomalies and Security Vulnerabilities in Executable Files. In *ACM-SE 44*, pages 560–565. ACM.

Tripp, O., Pistoia, M., Fink, S., Sridharan, M., and Weisman, O. (2009). TAJ: Effective taint analysis of web applications. In *PLDI*, pages 87–97. ACM.

Viega, J., Bloch, J. T., Kohno, T., and McGraw, G. (2002). Token-Based Scanning of Source Code for Security Problems. *ACM Trans. Inf. Syst. Secur.*, 5(3):238–261.

Weber, M., Shah, V., and Ren, C. (2001). A Case Study in Detecting Software Security Vulnerabilities Using Constraint Optimization. In *Workshop on Source Code Analysis and Manipulation*, pages 1–11.

 ©2013 SBC — Soc. Bras. de Computação