Verificação Estática de Acessos a Arranjos em C*

Henrique Nazaré Santos, Fernando Magno Quintão Pereira, Leonardo Barbosa Oliveira

[1]Departamento de Ciencia da Computação – UFMG
Av. Antônio Carlos, 6627 - 31.270-010 - Belo Horizonte - MG - Brazil

hnsantos,fernando,leonardo.barbosa@dcc.ufmg.br

Resumo. *Existem linguagens, tais como C ou C++, que permitem que arranjos sejam lidos fora de seus limites alocados. Essa característica, se por um lado aumenta o desempenho dessas linguagens, por outro permite que vírus e* worms *contaminem software de forma efetiva. O objetivo deste trabalho é remediar tal vulnerabilidade; contudo, sem diminuir sobremaneira a eficiência do código escrito em C ou C++. Com tal propósito, nós projetamos e testamos um conjunto de análises estáticas de programas que mostram que alguns acessos a arranjos são seguros, e aponta outros que não o são. Nossos algoritmos foram usados para estender a popular ferramenta* AddressSanitizer*, que protege arranjos contra ataques de estouro de* buffer*. Pudemos apontar que 40% dos acessos a arranjos em SPEC CPU 2006 são seguros. Tal informação permitiu-nos diminuir as responsabilidades de verificação impostas sobre* AddressSanitizer*, aumentando o desempenho dos programas protegidos em cerca de 10% – um feito animador, considerando-se a qualidade industrial daquela ferramenta.*

Abstract. *Some languages, such as C and C++, allow out-of-bounds accesses of array positions. This characteristic boosts the efficiency of these languages; however, it is also the core reason behind a plethora of malware that plagues the internet. The goal of this paper is to guard these arrays against illegal accesses with little slow down to programs written in these unsafe languages. To reach this goal, we have designed and tested a suite of static code analyses which we combine to show that some array accesses are always safe. We have tested these algorithms in the* AddressSanitizer *tool, which is widely used to guard arrays against buffer overrun attacks. We have been able to show that 40% of all array accesses in the programs available in SPEC CPU 2006 are safe. This knowledge lets us reduce the burden imposed by AddressSanitizer, increasing the performance of guarded code by 10% on average, reaching a maximum increase of 19%. This final result is remarkable, given that we got it atop an industrial quality tool.*

1. Introdução

Arranjos são um dos principais tipos estruturados de dados usados em linguagens de programação. Um arranjo é uma sequência contígua de posições de memória, todas de mesmo tamanho, indexadas a partir da primeira célula desse conjunto. A vantagem desses tipos sobre outras estruturas de dados é o acesso aleatório em tempo constante,

*Este projeto é financiado pela Intel e pelo CNPq.

consequência natural da alocação contígua e uniforme. As listas, tão presentes em linguagens funcionais, e mesmo as tabelas associativas, símbolo das linguagens dinamicamente tipadas, não conseguem prover tal eficiência. Esse alto desempenho de acesso, aliado à simplicidade de implementação, motivam o uso de arranjos com os mais diversos fins, em linguagens imperativas, tais como Java, C#, C++ e C.

Algumas dessas linguagens, por exemplo Java e C#, verificam se os valores usados para indexar um arranjo v de fato encontram-se dentro dos limites da memória alocada para v. Chamaremos tais linguagens de *fortemente tipadas* [Pierce 2004]. Outras linguagens, em particular C e C++, não provêem essa garantia, e portanto serão chamadas de linguagens *fracamente tipadas*. A principal vantagem da tipagem fraca é o desempenho. Verificações de limites de arranjo custam algum processamento, que embora de complexidade constante, não é irrisório. A grande desvantagem é a insegurança [Bhatkar et al. 2003]. Vários vírus de computador e *worms* exploram esse fato para sobrescrever dados críticos de um programa via ataques de *estouro de buffer*. A título de exemplo, o famoso *worm* de 1988 valia-se de uma vulnerabilidade de estouro de buffer no aplicativo `finger` para espalhar-se pela Internet [Rochlis and Eichin 1989].

A verificação de acessos a arranjos em C e C++ é um problema de difícil solução. Existem ferramentas que, agregadas ao compilador, alteram o código binário produzido a partir de programas escritos naquelas linguagens, provendo garantias de acesso legítimo a arranjos. Exemplos incluem SAFECode [Criswell et al. 2007, Criswell et al. 2009] e AdressSanitizer [Serebryany et al. 2012]. Entretanto, essas ferramentas têm um grande impacto no desempenho do código protegido. Nossos experimentos revelam que SAFECode reduz a velocidade do programa instrumentado em cerca de 30%, e AddressSanitizer em cerca de 73%. Além disso, AddressSanitizer aumenta o consumo de memória do programa em até 340%. É nossa intenção diminuir esse impacto, preservando, contudo, as garantias que essas ferramentas provêem. O instrumento que usaremos para alcançar tal objetivo será a *análise estática de código*.

Esse artigo apresenta uma coleção de análises estáticas de programas, implementadas a nível do compilador, que, combinadas, mostram que algumas indexações de arranjos são seguras. Por indexação segura entende-se um acesso dentro de limites legitimamente alocados ao arranjo. Acessos seguros não precisam ser instrumentados. Ao reduzir a necessidade de guardas no programa protegido, nós consequentemente aumentamos o seu desempenho. Nossa técnica envolve três análises de código, a saber:

- Uma análise de largura de variáveis, que estima o menor e o maior valor que cada variável inteira no programa pode assumir.
- Uma análise de tamanhos relativos, que determina, para cada variável v no programa, o conjunto das variáveis menores que v.
- Uma análise de inferência de tamanho de arranjos, que associa a cada arranjo um conjunto de variáveis que representa o seu tamanho.

Um acesso a arranjo como $v[i]$ é seguro se o valor da variável i for menor que o tamanho da área alocada para o arranjo v e for não-negativo. Essa observação, combinada com os resultados de nossas três análises, nos dá subsídios para levar adiante uma transformação de código: removemos todas as verificações realizadas sobre indexações sabidamente seguras.

As duas primeiras análises estáticas que utilizamos não são novas. O pro-

blema de inferir a largura de variáveis vem já sendo estudado desde que pesquisadores deram seus primeiros passos no campo da interpretação abstrata de programas [Cousot and Cousot 1977]. A análise de tamanho relativo é mais recente: ela foi proposta por Logozzo e Fahndrich em 2008 [Logozzo and Fahndrich 2008]. Acreditamos, contudo, que diversos detalhes de nossa implementação desses algoritmos são novas, conforme discutiremos na seção 3. Por outro lado, a análise de inferência de tamanho de arranjos é uma contribuição original deste trabalho. A eliminação de acessos guardados a arranjos é um problema antigo [Bodik et al. 2000]. Porém, os trabalhos anteriores não necessitavam da análise que propomos. Essas pesquisas anteriores voltam-se a linguagens que agregam junto aos arranjos meta-informação descrevendo o seu tamanho, como Java e SML. Essa informação não está disponível em C.

A fim de validar nossa técnica, nós a implementamos sobre AddressSanitizer. Essa ferramenta é distribuída como parte do compilador LLVM [Lattner and Adve 2004]. LLVM é um compilador de qualidade industrial, mantido por empresas como Apple e Cray. Nós testamos as nossas análises sobre um conjunto de benchmarks com mais de 2.3 milhões de linhas de código C, que inclui os programas presentes em SPEC CPU 2006. Fomos capazes de mostrar que cerca de 40% dos acessos a arranjos são seguros nesses programas. O resultado direto desse esforço é ganho de desempenho: nossa versão de AddressSanitizer é capaz de gerar código até 19% mais rápido que a edição original daquela ferramenta. Entre os programas disponíveis em SPEC CPU 2006, por exemplo, obtivemos um ganho médio de desempenho de 10%, sendo nosso melhor resultado obtido em `464.h264ref` – 19.85% – e nosso pior resultado obtido em `444.namd` – 2.74%.

2. Vulnerabilidades devido a Arranjos Inseguros

A figura 1 mostra dois programas de semântica similar. O primeiro deles, na parte (a) da figura, foi implementado em Java, e o segundo, na parte (b), foi escrito em C. Ambas as realizações da função `copyAndPrint` recebem um arranjo de entrada e o copiam para um buffer declarado localmente. O comportamento dinâmico desses programas, contudo, é muito diferente. A indexação na linha 7 da figura 1 (a) eventualmente leva a um erro em tempo de execução. A mensagem produzida por esse erro pode ser vista na figura 1 (c). A exceção foi levantada devido a uma tentativa de escrever à quinta célula do arranjo `buf`, declarado com somente quatro posições. O programa escrito em C, por outro lado, não gera um erro em tempo de execução. Sua saída é mostrada na figura 1 (d). Entretanto, ainda que o erro não tenha acontecido, o programa está em um estado *indefinido*, e seu comportamento já não pode mais ser previsto pela semântica de C.

O programa da figura 1 (b) realiza cinco escritas em memória na linha 10. Quatro dessas gravações acontecem em memória alocada para o arranjo `buf`. A quinta escrita acontece em área não explicitamente alocada. Esse é o princípio básico de um ataque de estouro de arranjos. Essas escritas ilegais, também chamadas *ilegítimas*, sobre-escrevem dados necessários ao correto funcionamento do programa. Um desses dados pode ser o endereço de retorno da função `copyAndPrint`. Um adversário pode alterar esse endereço para desviar o fluxo de execução do programa, por exemplo, para uma das funções de sistema, tais como `shell`. Ele poderia, assim, invocar essa função com os mesmos privilégios do programa comprometido.

Existem diversas ferramentas que conseguem proteger arranjos em C de


```
1  public class P {
2    final static int SIZE = 4;
3
4    static void copyAndPrint(byte[] v) {
5      byte[] buf = new byte[SIZE];
6      for (int i = 0; i < v.length; i++) {
7        buf[i] = v[i];
8        System.out.println("-> " + buf[i]);
9      }
10   }
11
12   public static void main(String args[]) {
13     byte[] v = {0, 1, 2, 3, 4};
14     copyAndPrint(v);
15   }
16
17 }
```

(a)

```
1  #define SIZE 4
2
3  void copyAndPrint(char v[], int n) {
4    char buf[SIZE];
5    int i;
6    for (i = 0; i < n; i++) {
7      buf[i] = v[i];
8      printf("-> %d\n", buf[i]);
9    }
10 }
11
12 int main() {
13   int VSIZE = SIZE + 1;
14   char a[VSIZE] = {0, 1, 2, 3, 4};
15   copyAndPrint(a, VSIZE);
16 }
```

(b)

```
-> 0
-> 1
-> 2
-> 3
Exception in thread "main"
java.lang.ArrayIndexOutOfBounds
Exception: 4
        at P.copyAndPrint(P.java:7)
        at P.main(P.java:14)
```

(c)

```
-> 0
-> 1
-> 2
-> 3
-> 4
```

(d)

Figura 1. (a) Exemplo de programa em Java (b) Programa equivalente em C. (c) Saída do programa Java. (d) Saída do programa C.

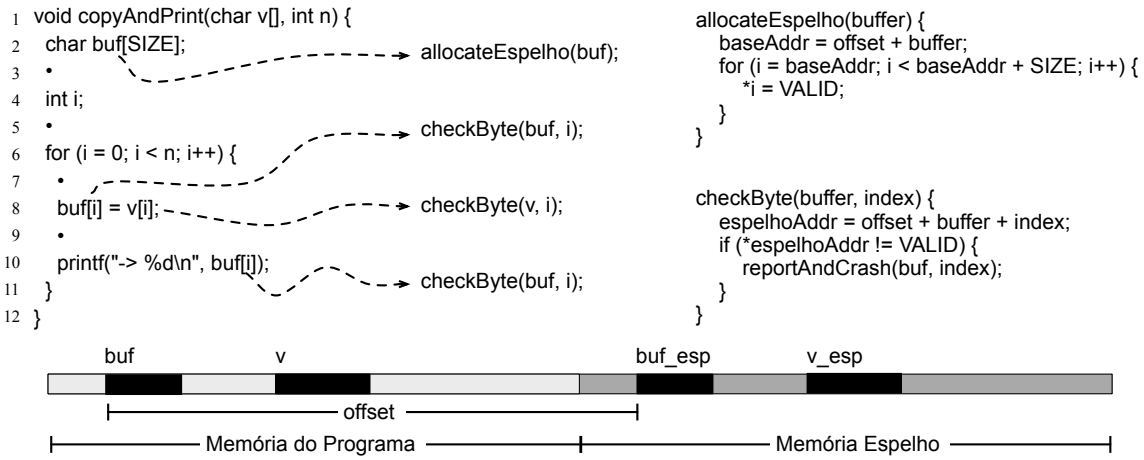

Figura 2. Exemplo de programa guardado por `AddressSanitizer`.

acessos indevidos. Os exemplos mais famosos são `Memcheck`, parte de Valgrind [Nethercote and Seward 2007], SAFECode [Criswell et al. 2007] e AddressSanitizer [Serebryany et al. 2012]. Essas ferramentas garantem a segurança de indexações em C via a combinação de duas técnicas: o *espelhamento* de memória e a *instrumentação* de código. O espelhamento [1] consiste na replicação da memória. Cada espaço v alocado pelo programa espelhado leva à alocação de um espaço adicional s, que guarda informações relativas a v. A instrumentação de código consiste em asserções que garantem a legitimidade de cada indexação. Essas garantias são obtidas verificando-se se a memória acessada possui um espelho válido.

O programa na figura 2 ilustra esses dois princípios, aplicados pela ferramenta AddressSanitizer. Ao compilar a função `copyAndPrint`, AddressSanitizer insere o código de instrumentação sobre os binários que produz. O nosso exemplo requer quatro chamadas a código de proteção, as quais podem ser vistas na figura 2. A primeira dessas chamadas, feita sobre `allocateEspelho`, replica a memória reservada para `buf`

[1] Tradução livre do termo *shadowing*.

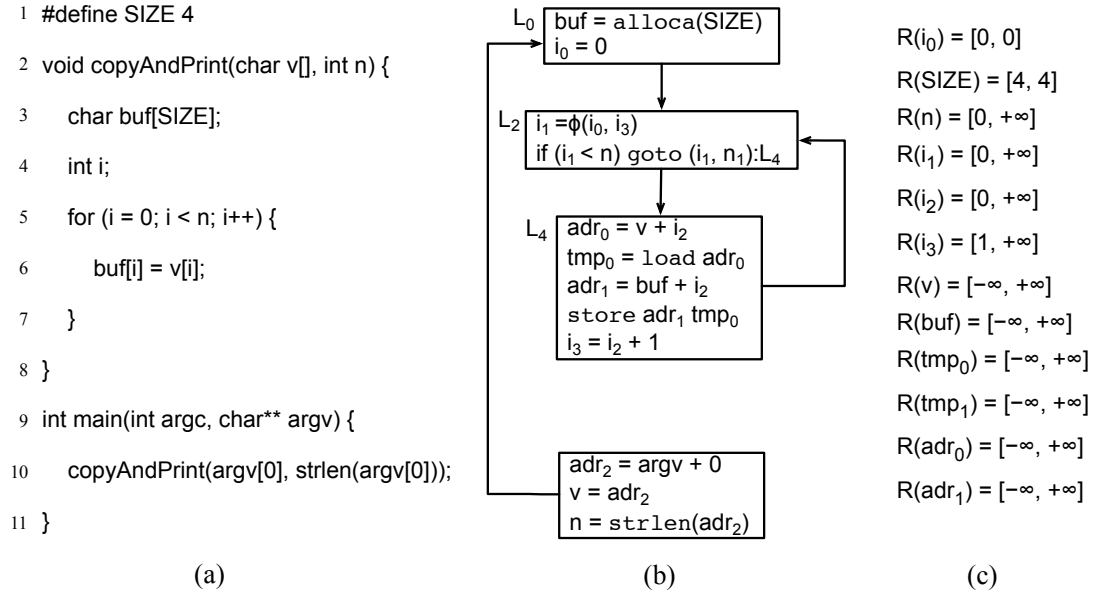

Figura 3. (a) O programa que usaremos como exemplo no restante desta seção. (b) O mesmo programa em formato e-SSA. (c) Resultado da análise de largura de variáveis no programa exemplo.

na memória de espelhamento. As outras três verificam se as indexações de arranjos são válidas. Cada uma dessas três verificações possuem um custo, em termos de tempo computacional. O objetivo desse trabalho é eliminar alguns desses testes, e por conseguinte, o seu custo.

3. Saneamento Estático de Acessos

Nesta seção, descreveremos as técnicas que utilizamos para mostrar que acessos a arranjos são seguros. Conforme explicamos na seção 1, usamos uma combinação de três análises estáticas de código para alcançar nosso objetivo: largura de variáveis, tamanhos relativos e inferência de tamanho de arranjos. No que se segue explicaremos cada um desses métodos.

Nossas análises dão-se sobre a representação intermediária dos programas, não sobre o seu código fonte. Em outras palavras, analisamos o código de três endereços obtido a partir de um programa escrito em linguagem de alto nível. A figura 3 (a) mostra o exemplo que usaremos no restante desta seção para explicar nossas técnicas. A representação intermediária desse exemplo pode ser vista na figura 3 (b). Escolhemos implementar nossas análises dessa forma por razões pragmáticas: trabalhando ao nível da representação intermediária de programas, podemos lidar com programas originalmente escritos em C, ou em C++, usando a mesma infraestrutura.

Nós adotamos uma representação intermediária conhecida como *formato de atribuição estática única estendida* (e-SSA) [2] [Bodik et al. 2000]. A representação e-SSA é uma extensão do formato SSA, inicialmente proposto por Cytron *et al.* [Cytron et al. 1991], e hoje usado em virtualmente qualquer compilador de importância. O formato e-SSA possui duas propriedades: (i) toda variável é definida por

[2]Tradução livre da expressão inglesa *Extended Single Assignment Form*.

somente uma instrução do programa fonte, e (ii) variáveis são renomeadas após serem usadas em testes condicionais. A grande vantagem dessa representação é que ela nos permite associar informações a cada variável. Por exemplo, podemos dizer que uma variável v é sempre menor que outra variável u, o que o tamanho de um arranjo é sempre s.

Uma vez que usamos o formato e-SSA, algumas variáveis presentes no programa original podem ter de ser renomeadas. Tal é o caso da variável i, que na figura 3 (b) foi renomeada para i_0, i_1, i_2 e i_3. Cada um desses novos nomes poder ser associado a diferentes propriedades da variável i. Note, por exemplo, que o teste condicional, logo no final do bloco L_2, cria dois novos nomes, i_2 e n_1. Uma vez que esses nomes existem somente nos caminhos de programa em que $i < n$, sabemos que $i_2 < n$ sempre. Funções ϕ, como aquela vista no bloco L_2, juntam alguns desses nomes em novas definições. A variável i_1, por exemplo, é a junção das variáveis i_0 e i_3.

3.1. Análise de Largura de Variáveis

O objetivo da análise de largura de variáveis é estimar o menor e o maior valores que cada variável inteira pode assumir no decurso da execução de um programa. Esse tipo de análise é já bem conhecida na literatura de linguagens de programação. A primeira versão de uma análise de largura de variáveis foi proposta por Cousot e Cousot, em 1977 [Cousot and Cousot 1977]. Desde então novos algoritmos surgiram, melhorando a descrição original tanto em termos de desempenho, quanto em termos de precisão.

Neste artigo nós adotamos um algoritmo proposto recentemente por Rodrigues *et al.* [Rodrigues et al. 2013]. Como esse algoritmo não é uma contribuição de nosso trabalho, omitiremos sua descrição, atendo-nos somente aos seus resultados. Esses resultados podem ser vistos na figura 3 (c). A análise de largura de variáveis foi capaz de inferir intervalos para sete das variáveis usadas em nosso programa. Alguns desses intervalos são constantes, como aqueles associados a i_0 e `SIZE`. Outros não. Por exemplo, a variável i_1 pode assumir qualquer valor não-negativo. Tais informações são inferidas em tempo linear no número de variáveis presentes no programa em formato e-SSA.

3.2. Análise de Tamanhos Relativos

A análise de tamanhos relativos encontra, para cada variável v no programa fonte, o conjunto das variáveis que são menores que v. Análises de propósito semelhante já foram utilizadas anteriormente, por exemplo, por Bodik *et al.* [Bodik et al. 2000], ou por Logozzo e Fahndrich [Logozzo and Fahndrich 2008]. Entretanto, acreditamos que nossa abordagem difere daqueles trabalhos quanto às restrições que analisamos. Como a maior parte dos algoritmos de análise estática de código, a resolução de tamanhos relativos dá-se via a interpretação abstrata de instruções. Para cada instrução presente em nossa representação intermediária nós definimos uma interpretação abstrata diferente. Essas interpretações definem um sistema de restrições, o qual é mostrado na figura 4. O objetivo da análise de tamanhos relativos é associar a cada variável v presente no programa um conjunto $L(v)$ que contém os nomes de variáveis comprovadamente menores ou iguais a v.

O sistema de restrições visto na figura 4 usa os resultados da análise de largura de variáveis para inicializar cada conjunto L. Uma vez concluída essa etapa inicial, resolvemos as restrições iterando-as até que um ponto fixo seja atingido. Durante uma iteração escolhemos uma instrução, como por exemplo $i_3 = i_2 + 1$, e interpretamos a restrição

Initialization

- For every variables v $\in$ P, L(v) $\supseteq$ {v}
- For every variables u and v $\in$ P, if up(u) $\leq$ inf(v) $\Rightarrow$ L(u) $\supseteq$ {v}

Constraints

[AsgLt] "u = v" $\in$ P $\Rightarrow$ L(u) = L(v)

[PhiLt] "v = phi(v_1,..., v_n)" $\in$ P $\Rightarrow$ L(v) = L(v_1) $\cap$ ... $\cap$ L(v_n)

[AddLt] "u = v + c" $\in$ P $\Rightarrow$ L(u) $\supseteq$ L(v) $\cup$ {v}, if nat(c) and c > 0; L(v) $\supseteq$ L(u) $\cup$ {u}, if nat(c) and c < 0

[MulLt] "u = v × c" $\in$ P $\Rightarrow$ L(u) $\supseteq$ L(v) $\cup$ {v}, if float(c) and c > 1; L(v) $\supseteq$ L(u) $\cup$ {u}, if float(c) $\in$ R and c > 0 and c < 1

[ModLt] "u = v % w" $\in$ P $\Rightarrow$ L(w) $\supseteq$ L(u) $\cup$ {u} and L(v) $\supseteq$ L(u) $\cup$ {u} if up(R(v)) > 0 and up(R(w)) > 0

[LthLt] "if u < v goto (u_0, v_0):L_0 else goto (u_1, v_1):L_1" $\in$ P $\Rightarrow$ L(v_0) $\supseteq$ L(u_0) $\cup$ {u_0} and L(u_1) $\supseteq$ L(v_1) $\cup$ {v_1}

[LeqLt] "if u $\leq$ v goto (u_0, v_0):L_0 else goto (u_1, v_1):L_1" $\in$ P $\Rightarrow$ L(v_0) $\supseteq$ L(u_0) and L(u_1) $\supseteq$ L(v_1)

Figura 4. Sistema de restrições usado para implementar a análise de tamanhos relativos.

L(i_0) = {i_0}	L(n) = {i_2, n}	L(i_3) = {i_2, i_3}
L(i_1) = {i_0, i_1}	L(adr_0) = {buf, i_2, adr_0}	L(adr_2) = {argv, adr_2}
L(i_2) = {i_0, i_2}	L(adr_1) = {buf, i_2, adr_1}	L(v) = {adr_2, v}

Figura 5. Resultado da análise de tamanhos relativos no programa visto na figura 3 (b).

que lhe corresponde. Para esse exemplo particular, temos a restrição dada pela regra [AddLt]. Essa regra nos leva a inserir todos os símbolos presentes em $L(i_2)$, além do próprio nome i_2 ao conjunto $L(i_3)$. Esse algoritmo iterativo sempre termina, pois cada conjunto $L(v)$ pode somente crescer durante cada interpretação abstrata. Uma rápida inspeção da figura 4 revela que variáveis são inseridas nos conjuntos, mas nunca retiradas deles.

Continuando com o nosso exemplo, a figura 5 mostra o resultado da análise de tamanhos relativos aplicada ao programa da figura 3 (b). Podemos concluir, por exemplo, que a variável i_2, usada para indexar o arranjo v é sempre menor que n. Para mostrarmos que essa indexação é segura, é necessário que sejamos capazes de associar a variável n ao tamanho do arranjo v. Essa etapa é assunto de nossa próxima seção.

3.3. Inferência de Tamanho de Arranjos

A terceira de nossas análises estáticas busca associar a cada ponteiro base de um arranjo o tamanho da região de memória apontada. Seu objetivo é criar um mapa S, tal que $S(v) = n$ indica que n é o tamanho da região alocada para v. A partir dos resultados dessa análise, mais as informações geradas nos passos anteriores, podemos mostrar que algumas indexações são seguras. Obtemos o tamanho de arranjos a partir de algumas chamadas da

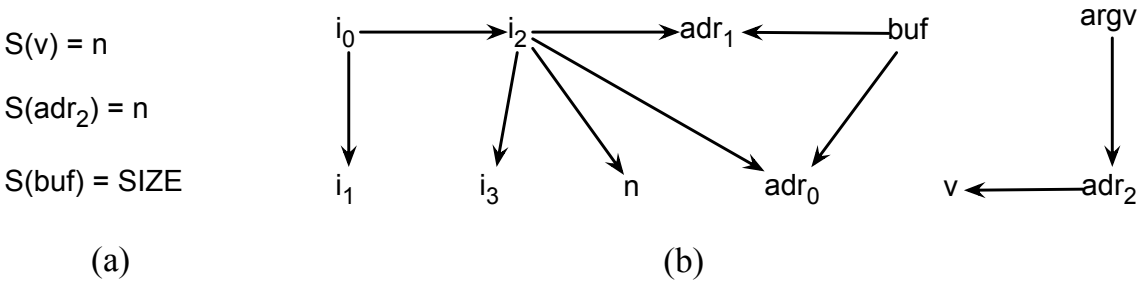

Figura 6. (a) Resultado da inferência de tamanhos de arranjos. (b) Grafo de tamanhos relativos.

API de alocação de memória, e a partir de funções como `strlen`. A presença de uma instrução $v = \texttt{alloca}(n)$ no texto do programa alvo nos dá $S(v) = n$. De modo similar, $n = \texttt{strlen}(v)$ dá-nos a mesma informação. O resultado da inferência de tamanhos de arranjos para o nosso exemplo corrente pode ser visto na figura 6 (a).

3.4. Eliminação de Guardas Redundantes

De posse das informações produzidas pelas análises vistas nas seções 3.1, 3.2 e 3.3, passamos à eliminação de guardas redundantes. Com tal propósito, construímos um grafo de tamanhos relativos. Esse grafo direcionado possui um vértice v para cada variável no programa, e uma aresta $(u \rightarrow v)$ se for o caso que $L(u) = v$. Nosso grafo é uma simplificação da estrutura usada por Bodik *et al.* para eliminar testes de acesso a arranjos em Java [Bodik et al. 2000].

A figura 6 (b) mostra o grafo que produzimos para nosso exemplo da figura 3. Uma indexação $v[i]$ é segura quando existe um caminho do nodo que representa a variável i até o nodo que representa o tamanho do arranjo v. Em nosso exemplo, existe um caminho de i_2 até n, o tamanho de v. Logo, a indexação de v na linha 6 da figura 3 (a) é segura. Por outro lado, não existe caminho de i_2 até SIZE, o tamanho do arranjo `buf`. Assim, não é possível mostrar que o acesso a esse arranjo, ainda na linha 6 da figura 3, é segura.

3.5. `Aranot` – Uma Ferramenta de Apoio à Programação Segura em C

Desta pesquisa resultaram duas implementações de *software*. A primeira delas é uma extensão de AddressSanitizer, que elimina testes desnecessários de limites de arranjos. O objetivo dessa primeira ferramenta é auxiliar o compilador a gerar código mais eficiente. A sua efetividade será discutida na seção 4. Essa extensão de AddressSanitizer é uma ferramenta útil para o compilador, não para o programador. Por outro lado, durante o desenvolvimento de nossas análises de código, pudemos perceber que elas também são úteis para o desenvolvedor de *software*.

A partir dessa observação, criamos `Aranot`, uma ferramenta que aponta erros de indexação de arranjos em programas escritos em C ou C++. `Aranot` foi implementada como uma extensão do compilador LLVM. Sua entrada é uma aplicação, escrita em C ou C++, possivelmente em vários arquivos separados. A sua saída é o mesmo programa, com anotações na forma de comentários. Apesar de a ferramenta emitir código nas mencionadas linguagens, as análises continuam sendo sobre a representação intermediária, utilizando-se a informação de arquivo fonte e linha disponível nos metadados de depuração. As anotações provêem dois tipos de informação. Elas podem indicar que

(a)
```
int main(int argc, char **argv) {
  int *a = malloc(argc * sizeof(int));
  for (int i = 0; i < argc; ++i)
    a[i] = i;
  free(a);
  return 0;
}
```

(b)
```
int main(int argc, char **argv) {
  int *a = malloc(argc * sizeof(int));
  for (int i = 0; i < argc; ++i)
    /* bytes(&a[i]) = sizeof(int) (4 bytes) */
    /* index(&a[i]) = 1          (4 bytes) */
    /* SAFE: Access is provably safe. */
    a[i] = i;
  free(a);
  return 0;
}
```

Figura 7. (a) Programa com acesso seguro. (b) Anotação produzida por `Aranot`.

(a)
```
int main() {
  const char s[] = "Simple string";
  const size_t len = strlen(s);
  char *a = malloc(len * sizeof(char));
  strcpy(a, s);
  free(a);
  return 0;
}
```

(b)
```
int main() {
  const char s[] = "Simple string";
  const size_t len = strlen(s);
  char *a = malloc(len * sizeof(char));
  /* bytes(a) = 13 * sizeof(char) (13 bytes) */
  /* index(a) = 13              (13 bytes) */
  /* bytes(s) = 14 * sizeof(char) (14 bytes) */
  /* index(a) = 14              (14 bytes) */
  /* WARNING: Possible unsafe memory access. */
  strcpy(a, s);
  free(a);
  return 0;
}
```

Figura 8. (a) Programa com acesso incorreto. (b) Anotação gerada por `Aranot`.

acessos a arranjos são seguros, ou podem apontar erros óbvios de indexação. A figura 7 contém um exemplo da primeira categoria de anotação.

A figura 8 ilustra um exemplo de *bug* que `Aranot` é capaz de encontrar. Nesse caso, nossa ferramenta pôde reportar que 14 bytes de dados estão sendo incorretamente copiados para um arranjo que comporta somente 13 bytes. Existem ferramentas dinâmicas, como Valgrind [Nethercote and Seward 2007] ou PIN [3], que conseguem capturar esse tipo de erro. Tais ferramentas, uma vez que analisam traços da execução do programa, são mais precisas que `Aranot`. Por outro lado, `Aranot` não requer que o programa sob análise seja executado, e portanto não impõe qualquer custo sobre ele. Além disso, `Aranot` produz diagnósticos definitivos: se nossa ferramenta reporta que um acesso é seguro, então o programador tem a garantia de que ele nunca será lido ou escrito fora de limites alocados.

4. Resultados Experimentais

A fim de validar nossa análise, nós a implementamos sobre LLVM versão 3.4. Todos os experimentos que apresentaremos nesta seção foram executados em um computador com 12 núcleos Intel Xeon a 2GHz, e com 16GB de memória RAM. O sistema operacional usado foi Ubuntu GNU/Linux 12.04. Utilizaremos como *benchmarks* os programas presentes na coleção SPEC CPU 2006. Todos os tempos de execução que reportamos foram obtidos em programas compilados com o nível de otimização `-O0`.

A efetividade do algoritmo de eliminação de guardas. A figura mostra a percentagem de testes de limite de arranjos que AddressSanitizer deixou de inserir devido à intervenção de nossa análise. Em média, pudemos economizar 25% das guardas usadas na indexação de arranjos. Esse número é a média geométrica entre nossos 17 *benchmarks*. A média aritmética é ainda melhor: 31%. Nosso pior resultado foi obtido sobre `gcc`, e nosso

[3] `http://www.pintool.org/`

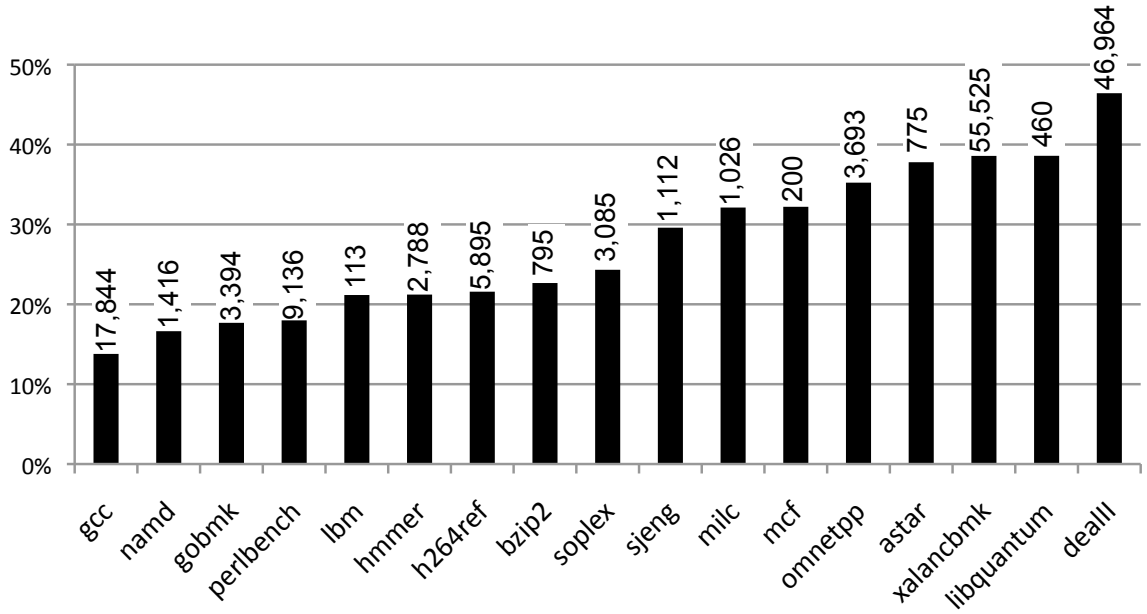

Figura 9. Percentagem de testes sobre arranjos eliminados. Valores sobre barras indicam o número absoluto de testes eliminados.

melhor resultado foi conseguido em `dealII`, uma aplicação que usa arranjos intensivamente. A partir dos dados vistos na figura 9, podemos concluir que nossa análise é efetiva. Em geral somos capazes de eliminar quase todas as guardas sobre arranjos unidimensionais alocados no escopo local de funções. Arranjos multidimensionais, e arranjos cujo tamanho pode mudar dinamicamente são ainda um desafio para nossos algoritmos.

Ganhos em tempo de execução. O principal objetivo de nosso trabalho é aumentar o tempo de execução dos programas guardados contra ataques de estouro de arranjos. Esse aumento de velocidade é consequência da eliminação dos testes de limites. A figura mostra quão efetivos nós estamos sendo no cumprimento desse objetivo. O primeiro gráfico naquela figura compara o tempo de execução absoluto dos programas produzidos de três formas diferentes: (i) sem qualquer teste de limites de arranjos, (ii) com a eliminação de alguns testes, devido à técnica que discutimos neste artigo e (iii) com todos os testes de limites de arranjos que AddressSanitizer insere. No modo (i), somente as guardas de verificação são removidas - operações de *bookkeeping* necessárias para o funcionamento da ferramenta ainda são executadas. Assim, os tempos em (i) são o limite de tempo inferior para (ii), e quanto mais precisa a nossa análise, mais (ii) tende a se aproximar de (i). Vemos que essa aproximação acontece em alguns *benchmarks*, como `xalancbmk` e `libquantum`. Coincidentemente, esses são alguns dos *benchmarks* em que somos capazes de eliminar mais guardas.

O segundo gráfico visto na figura 10 mostra a redução percentual do tempo de execução dos programas seguros devido à nossa análise. Note que essa figura compara somente programas invulneráveis a ataques de estouro de arranjos. Fomos capazes de obter um ganho de velocidade em todos os *benchmarks*. Em média, pudemos observar um aumento de velocidade de cerca de 10% ao remover alguns testes inseridos por AddressSanitizer. Esse número variou de 2.4% em `namd` a 20% em `h264ref`. A melhoria de tempo de execução é inferior ao percentual de testes eliminados porque esses testes

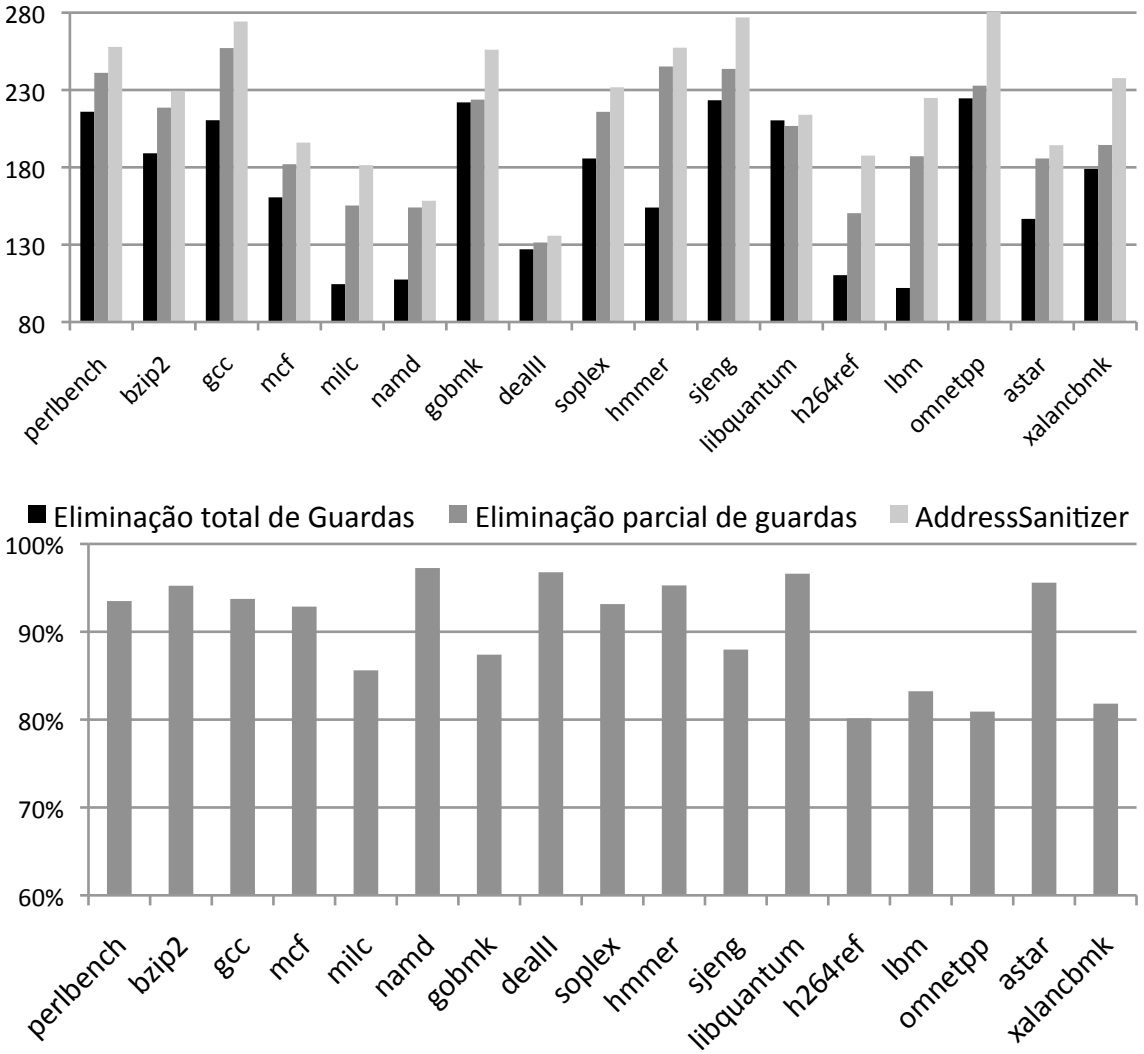

Figura 10. (Topo) Redução absoluta do tempo de execução dos programas devido à eliminação de testes de arranjos. Tempo é medido em segundos. (Base) Redução percentual do tempo de execução dos programas em relação aos programas totalmente guardados por AddressSanitizer.

representam somente uma pequena parcela de todas as instruções usadas nos programas que analisamos.

Tempo de execução de nossas análises. A figura 11 compara o tempo de execução das três análises descritas na seção 3. Todas os algoritmos que usamos neste trabalho são lineares no número de instruções dos programas analisados. Para evidenciar esse fato, o gráfico da figura 11 contém também o tamanho de cada *benchmark*, em número de instruções *assembly*. O coeficiente de determinação das três análises, obtido com relação ao tamanho dos programas, foi sempre maior que 0.9, indicando o comportamento linear em todos os casos. Dentre nossas três análises, a inferência de tamanhos de arranjos é a mais rápida, e a análise de largura de variáveis a mais lenta. A inferência de tamanhos de arranjos ocupou 0,04% do tempo total de compilação de nossos *benchmarks*, usando-se o nível de otimização `-O0`. A análise de tamanhos relativos foi responsável por 1,4% do tempo de compilação, e a análise de intervalos ocupou 2,2%. Note que a

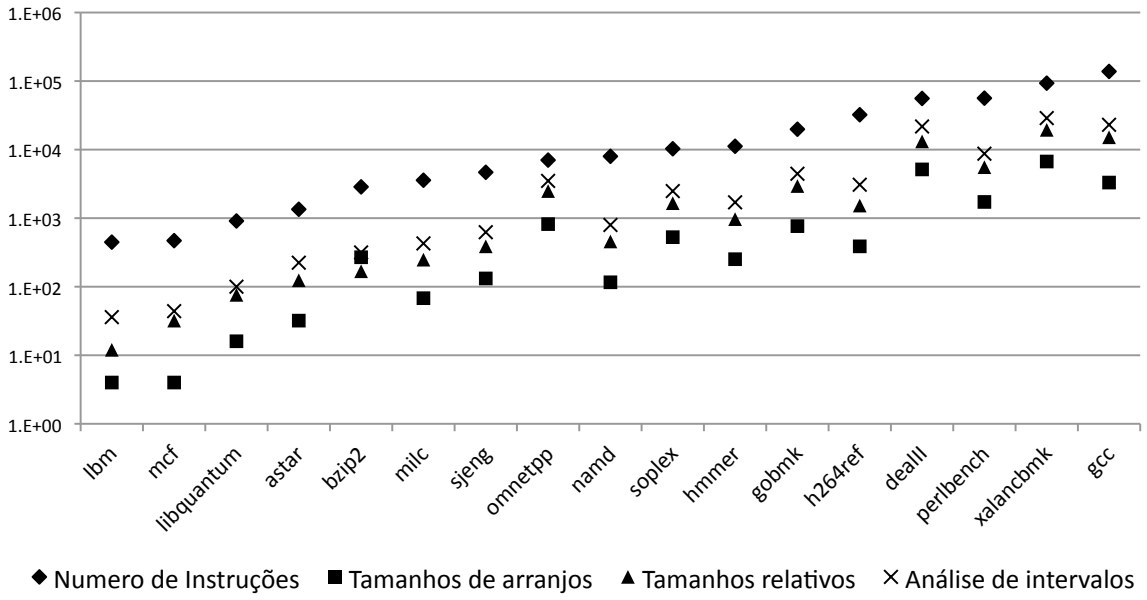

Figura 11. Tempo de execução de cada uma de nossas análises e o número de instruções *assembly* do programa analisado.

análise de intervalos é um algoritmo já bem estabelecido na literatura de compiladores. Em particular, utilizamos uma implementação popular, que vem sendo otimizada por diversos programadores em todo o mundo desde 2011 [4]. O fato que as outras duas análises que tivemos de implementar são mais rápidas que aquele algoritmo demonstra que nossas implementações são competitivas. Fica claro, dessa discussão, que todas as nossas análises possuem um tempo de execução modesto, quando comparadas com outras fases do processo de compilação.

5. Trabalhos Relacionados

A eliminação de testes de limites de arranjos é um problema já bem estudado na área de segurança computacional e linguagens de programação. Os primeiros trabalhos nesse campo valiam-se de otimizações clássicas de código para diminuir o efeito de testes de limites. Algumas dessas otimizações não tinham o objetivo direto de eliminar testes redundantes; por outro lado, o faziam como uma consequência natural das transformações de programa que implementavam [Rosen et al. 1988]. Entre os algoritmos especialmente projetados para remover testes de limites, merece destaque a pesquisa de Harrison [Harrison 1977] e Suzuki *et al.* [Suzuki and Ishihata 1977]. Esses dois trabalhos pioneiros utilizavam técnicas substancialmente diferentes para atingir a mesma meta. Harrison valia-se da análise de largura de variáveis, enquanto Suzuki utilizava programação linear inteira e indução via provadores de teoremas. Enquanto a abordagem de Harrison sofria o inconveniente da imprecisão, a técnica de Suzuki era muito custosa para ser aplicada a programas grandes.

Trabalhos posteriores enriqueceram esses dois artigos pioneiros, tanto em termos de precisão, quanto em termos de escalabilidade. No início da década de 90 vimos surgirem as primeiras análises de fluxo de dados especialmente desenvolvidas para a

[4] `http://code.google.com/p/range-analysis`

verificação de indexações em arranjos [Gupta 1993]. Naquela mesma época, a grande popularidade de Java, uma linguagem fortemente tipada, trouxe novo ânimo e incentivo para a pesquisa nessa área. Merece destaque o trabalho de Bodik *et al.* [Bodik et al. 2000], que propunha um método para eliminar guardas de acesso a arranjos em tempo de execução de programas. Essa técnica, conhecida como ABCD, influenciou diversos grupos de pesquisa subsequentes. O trabalho de Logozzo *et al.*, a partir do qual adaptamos a idéia da análise de tamanhos relativos, nasceu naquele contexto [Logozzo and Fahndrich 2008].

A principal diferença de nossa abordagem para aquelas mencionadas no parágrafo anterior refere-se à linguagem de programação alvo. Java, ao incluir junto aos arranjos seus tamanhos, facilita substancialmente a tarefa de eliminar testes de limites. C, por outro lado, é muito menos estruturada. Várias de nossas decisões, como a inferência de tamanhos de arranjos, foram motivadas por essa falta de estrutura. Existem, contudo, trabalhos que buscam validar acessos a arranjos em C. O mais importante dentre esses trabalhos foi o verificador estático proposto por Venet e Guilhaume [Venet and Brat 2004]. Aquela técnica difere da nossa porque ela se vale de uma análise estática baseada em poliedros [Cousot and Halbwachs 1978]. Poliedros são mais precisos que as análises que propomos neste trabalho. Por outro lado, eles são também muito mais caros computacionalmente. Embora Venet não informe seus tempos de análise, em suas palavras, a maior parte de seus testes "Could be analyzed in a few hours". Nossas análises, por outro lado, acontecem em poucos segundos. Além disso, a técnica de Venet e Guilhaume foi desenvolvida especificamente para um subconjunto de C usado na programação de sistemas embarcados. A nossa técnica pôde ser testada tanto em C quanto em C++.

6. Considerações Finais

Este artigo apresentou um conjunto de análises estáticas de programas que, combinadas, impedem a ocorrência de ataques de estouro de arranjos em linguagens como C e C++. Algumas dessas análises, como a inferência de largura de variáveis, ou a inferência de tamanhos relativos, são técnicas já conhecidas. A inferência de arranjos é uma contribuição deste trabalho. Além disso, a eliminação de testes de limites na linguagem C também é uma novidade. Essa linguagem, ao oferecer ao programador menos meta-informações, como o tamanho de arranjos, por exemplo, é mais difícil de analisar que Java, C#, Python e outras linguagens fortemente tipadas. O resultado final de nosso esforço é a eliminação de cerca de 40% de todos os testes realizados por AddressSanitizer para prevenir ataques de estouro de buffer. A partir dessa economia pudemos apurar um ganho de desempenho de até 20% em aplicações presentes em SPEC CPU 2006 quando compiladas por LLVM `-O0`. Salientamos que tanto LLVM quanto AddressSanitizer são ferramentas de uso industrial. Nossas análises são não somente úteis ao compilador, mas também ao programador. Para validar essa afirmação, nós as utilizamos para implementar uma ferramenta que aponta vulnerabilidades de ataques de estouro de arranjos em programas.

Software e Reproducibilidade: construímos uma ferramenta, `Aranot`, disponível via licença de código aberto, a partir das idéias discutidas neste artigo. Essa ferramenta pode ser obtida na página `http://code.google.com/p/ecosoc/`.

Referências

Bhatkar, E., Duvarney, D. C., and Sekar, R. (2003). Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *USENIX Security*,

pages 105–120.

Bodik, R., Gupta, R., and Sarkar, V. (2000). ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM.

Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM.

Cousot, P. and Halbwachs, N. (1978). Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM.

Criswell, J., Geoffray, N., and Adve, V. (2009). Memory safety for low-level software/hardware interactions. In *SSYM*, pages 83–100. USENIX Association.

Criswell, J., Lenharth, A., Dhurjati, D., and Adve, V. (2007). Secure virtual architecture: a safe execution environment for commodity operating systems. In *SOSP*, pages 351–366. ACM.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490.

Gupta, R. (1993). Optimizing array bound checks using flow analysis. *ACM Lett. Program. Lang. Syst.*, 2(1-4):135–150.

Harrison, W. H. (1977). Compiler analysis of the value ranges for variables. *IEEE Trans. Softw. Eng.*, 3(3):243–250.

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

Logozzo, F. and Fahndrich, M. (2008). Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *SAC*, pages 184–188. ACM.

Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM.

Pierce, B. C. (2004). *Types and Programming Languages*. MIT Press, 1st edition.

Rochlis, J. A. and Eichin, M. W. (1989). With microscope and tweezers: the worm from MIT's perspective. *Commun. ACM*, 32:689–698.

Rodrigues, R. E., Campos, V. H. S., and Pereira, F. M. Q. (2013). A fast and low overhead technique to secure programs against integer overflows. In *CGO*. ACM.

Rosen, B. K., Zadeck, F. K., and Wegman, M. N. (1988). Global value numbers and redundant computations. In *POPL*, pages 12–27. ACM Press.

Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. (2012). Addresssanitizer: a fast address sanity checker. In *USENIX*, pages 28–28. USENIX Association.

Suzuki, N. and Ishihata, K. (1977). Implementation of an array bound checker. In *POPL*, pages 132–143. ACM.

Venet, A. and Brat, G. (2004). Precise and efficient static array bound checking for large embedded c programs. In *PLDI*, pages 231–242. ACM.

