Protecting Programs Against Memory Violation In Hardware

A. L. M. Neto, L. T. C. Melo, O. P. V. Neto, F. M. Q. Pereira and L. B. Oliveira

***Abstract*— The C and C++ programming languages do not prevent ouf-of-bounds memory access, consequently leaving room to attacks such as buffer overflow and buffer overread. There are several techniques to make C programs safe. However these methods are usually implemented via software and tend to cause performance degradation. Our work aims at a hardware solution which is able to check bounds efficiently, by providing novel instructions that are aware of a buffer's valid memory range. Whenever a violation is found the program will terminate, a typical hardware exception behavior.**

***Keywords*— Code Security, Memory Violation, Buffer Overflow, Buffer Overread.**

I. INTRODUÇÃO

A LINGUAGEM C é uma das mais empregadas em meio à comunidade de programadores. A linguagem foi concebida com o foco na eficiência e permitiu que navegadores, sistemas operacionais e diversas outras aplicações essenciais para o progresso da Internet executassem numa velocidade compatível com os recursos computacionais disponíveis à época em que foram desenvolvidos. No entanto, um preço alto é pago nesta busca por eficiência. C, por exemplo, não realiza a Verificação de Limites de Arranjo (*Array-Bounds Check* – ABC) automaticamente.

O resultado dessa estratégia é que uma grande gama de programas escritos em C estão sujeitos a ataques que acessam memória para além de limites, tais como Estouro de Arranjo (*Buffer Overflow* – BOF) e Leitura pós Arranjo (*Buffer Overread* – BOR). Para ilustrar a capacidade destrutiva desses ataques, pode-se se citar o *worm* Morris [18], que nos idos de 80 abalou a Internet ao explorar uma vulnerabilidade de BOF causando um ataque de DoS sem precedentes; e, recentemente, o Heartbleed [19], que no ano passado deixou a comunidade de segurança digital em polvorosa, ao explorar a vulnerabilidades de BOR e, assim, furtar dados sigilosos de programas que utilizavam a biblioteca OpenSSL.

Evidentemente, ao longo dos anos diversas propostas surgiram com o intuito de proteger programas escritos em C ([1], [2], por exemplo). Grosso modo, os trabalhos existentes analisam programas, identificando os locais das vulnerabilidades e, posteriormente, inserindo ABCs nesses trechos de código. Em tese, tais propostas são capazes de contornar a questão da verificação de limites. Todavia, na prática, elas acabam sendo ineficientes. O problema está no fato de que a ABC feita em software acarreta alta sobrecarga (*overhead*).

Objetivo. O objetivo deste artigo é proteger sistemas computacionais contra ataques de Violação de Memória de forma eficiente. Mais precisamente, objetivamos conceber instruções seguras de linguagem de máquina para realizar o transporte de dados da memória para registradores e vice-versa. Desta forma, deixamos a cargo do hardware checar os limites de arranjos. Nossa proposta substitui as versões de leitura (*load*) e escrita (*store*) tradicionais por duas instruções análogas, mas seguras. Em ambas, os limites inferior e superior do arranjo manipulado são verificados antes dos acessos à memória.

Contribuição. Em particular, nossas principais contribuições são as seguintes.

1. Uma solução de ABC eficiente; já que a verificação é realizada em hardware.
2. Concepção, implementação e avaliação de uma instrução de leitura segura.
3. Concepção, implementação e avaliação de uma instrução de escrita segura.

Os resultados indicam que a nossa solução é, em média, 59% mais rápida que a estratégia em software

Organização. A Seção II aborda conceitos importantes para compreensão do trabalho. Na Seção III discutimos os trabalhos relacionados. A nossa solução é descrita na Seção IVIV. . A Seção V apresenta a metodologia de avaliação, bem como os resultados obtidos. Por fim, concluímos na Seção VI.

II. CONCEITOS

Esta seção apresenta brevemente conceitos para a compreensão do restante deste artigo. Em particular, discorremos sobre a arquitetura Y86 (Seção A.), a qual utilizamos para implementar nossa solução; e, subsequentemente, versamos sobre ABCs (Seção B.), apontando seu custo quando realizado via software.

A. Arquitetura Y86

A Y86 é uma arquitetura de 32 *bits* cujo conjunto de instruções (*Instruction Set Architecture* -- ISA) é baseado no ISA da arquitetura IA32. A Y86 suporta um conjunto reduzido de operações aritméticas e registradores.

A. L. M. Neto, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brasil, lemosmaia@dcc.ufmg.br

L. T. C. Melo, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brasil, ltcmelo@gmail.com

O. P. V. Neto, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brasil, omar@dcc.ufmg.br

F. M. Q. Pereira, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brasil, fernando@dcc.ufmg.br

L. B. Oliveira, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, Brasil, leob@dcc.ufmg.br

Estágio	rmmovl rA,valC(rB)	mrmovl rA,valC(rB)	subl rA,rB
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valC ← M_4[PC + 2] valP ← PC + 6 PC ← valP	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valC ← M_4[PC + 2] valP ← PC + 6 PC ← valP	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valP ← PC + 2 PC ← valP
Decode	valA ← R[rA] valB ← R[rB]	valB ← R[rB]	valA ← R[rA] valB ← R[rB]
Execute	valE ← valB + valC	valE ← valB + valC	valE ← valB – valA Set CC
Memory	M_4[valE] ← valA	valM ← M_4[valE]	
Write Back		R[rA] ← valM	R[rB] ← valE

Figura 1. Estágio das instruções rmmovl, mrmovl e subl ao longo do pipeline da arquitetura y86.

Ainda assim, Y86 é uma arquitetura com cinco estágios de *pipeline* em que programas relativamente complexos podem ser executados. Das instruções da arquitetura Y86, aquelas mais concernentes a este projeto são as de leitura e escrita à memória e comparação, a saber:

- `mrmovl`: lê da memória *(memory → register)*;
- `rmmovl`: escreve na memória *(register → memory)*;
- `subl`: subtrai dois operandos para posterior comparação;

Os estágios de execução dessas instruções através do *pipeline* da arquitetura são descritas a seguir:

1. *Fetch:* o Contador de Programa (*Program Counter* – PC) determina a instrução que deve ser executada. Os *bytes* da instrução são lidos e separados convenientemente em registradores de *pipeline*, que auxiliam na execução do próximo estágio. Além disso, o PC é incrementado de acordo com o tamanho da instrução lida (*valP*). Os campos de cada instrução são:
   - *icode* e *ifun*: os *bits* que determinam, respectivamente, o código (*opcode*) da instrução e a operação aritmética a ser executada no estágio *Execute*;
   - `rA` e `rB`: registradores utilizados pela instrução, fornecendo os valores dos cálculos e/ou recebendo o resultado computado ou o dado lido da memória;
   - *valC*: valor constante, com significado dependente da instrução.
2. *Decode:* os registradores envolvidos na instrução (no máximo dois – `rA` e `rB`) são lidos do Banco de Registradores (*Register file*) e armazenados em registradores de *pipeline*;
3. *Execute:* uma operação é computada pela Unidade Lógica Aritmética (*Arithmetic Logic Unit* – ALU). No caso das instruções de acesso à memória, a ALU é utilizada para calcular o endereço efetivo do acesso (soma do deslocamento *valC* com o valor do registrador base). Se a instrução é de subtração, por outro lado, a diferença entre os valores dos registradores é computada. Posteriormente, a operação *setCC* atribui valores a sinais de controle, os quais podem ser avaliados pela instrução seguinte para, por exemplo, realizar um desvio condicional;
4. *Memory:* são realizados os acessos de leitura ou escrita à memória;
5. *Write back:* o resultado de uma operação aritmética ou um dado de memória lido é escrito no registrador destino.

A Fig. 1 ilustra de maneira sintética os estágios descritos acima. Nela, $M_n[PC] \leftarrow x$ denota a escrita e $M_n[PC] \rightarrow x$ a leitura de *n bytes* da posição de memória PC. A notação $\$x{:}y \leftarrow M_1[PC]$, por sua vez, separa os dois componentes de um *byte*, ou seja, *x* recebe os 4 *bits* menos significativos de $M_1[PC]$ e *y* os *bits* restantes. Por fim, *R[rA]* simboliza a leitura ou escrita do registrador `rA` no banco de registradores.

B. ABCs e seu custo computacional

Uma forma simples de se proteger um programa em C contra ataques de Violação de Memória é testar a variável utilizada como índice do acesso ao arranjo contra os limites inferior e superior da memória base, a partir da qual aquele índice é construído. Por exemplo, observe as versões vulnerável (Fig. 2, lado esquerdo) e protegida (Fig. 2, lado direito) de um mesmo programa. (Note-se que na versão protegida há verificação de limites antes da atribuição `buffer[i] = a`). Agora suponha que elas sejam executadas com o índice `i` menor que zero, ou igual ou maior que `BUFSIZE`. Na versão vulnerável ocorrerá uma violação de memória, ao passo que na versão protegida não.

```
#define BUFSIZE 512

int main(int argc,
         char **argv) {
  int buffer[BUFSIZE];
  int a;
  int  i,j;
  ...
  for(i;i<j;i++){
    ...
    buffer[i] = a;
    ...
  }
  ...
}
```

```
#define BUFSIZE 512

int main(int argc,
         char **argv) {
  int buffer[BUFSIZE];
  int a;
  int  i,j;
  ...
  for(i;i<j;i++){
    ...
if((i >= 0)&&(i < BUFSIZE))
      buffer[i] = a;
    ...
  }
  ...
}
```

Figura 2. Versões vulnerável (esquerda) e protegida (direita) de um programa em C.

Agora, observe (Fig. 3) essas mesmas versões do programa traduzidas para a linguagem de montagem (*assembly*) da

arquitetura Y86. Note-se que a instrução efetiva, ou seja, que conclui a atribuição `buffer[i] = a`, corresponde à instrução `rmmovl %edx, -2060(%edi)`, em que o conteúdo do registrador `%edx` é copiado para o endereço de memória formado pela soma do registrador `%edi` e do valor de deslocamento, no caso `-2060`. Note-se ainda que quatro instruções são adicionadas para a verificação de cada limite, a saber:

1. `mrmovl`: carrega o valor do índice `i` em registrador;
2. `irmovl`: carrega o valor constante do limite (superior ou inferior) em registrador;
3. `subl`: subtrai o limite do índice;
4. `js` ou `jg`: desvio condicionado pelo resultado da operação de subtração.

Para ilustrar, considere a seguinte verificação de limite superior (Fig. 3, lado direito, `&& (i<BUFSIZE)`). O índice `i` é carregado no registrador `%edi` pela instrução `mrmovl -12(%ebp), %edi`; o registrador `%ebx` recebe o limite superior do arranjo em `irmovl $511, %ebx`; a instrução `subl %ebx, %edi` subtrai o limite superior do índice `i`; finalmente, o desvio condicional `jg L3` evita a execução do trecho da atribuição (`buffer[i] = a`) caso o resultado da subtração indique que o índice é maior (`jg` – jump if greater) que o limite superior.

```
L3:
 mrmovl  -12(%ebp),%eax
 mrmovl  -4(%ebp), %edx
 rrmovl  %eax, %edi
 sall    $2, %edi
 addl    %ebp, %edi
 rmmovl %edx,-2060(%edi)
 mrmovl  -12(%ebp),%edi
 iaddl   $1, %edi
 rmmovl  %edi, -12(%ebp)
L2:
 mrmovl  -12(%ebp),%eax
 mrmovl  -8(%ebp), %edi
 subl    %edi, %eax
 jl L3
 irmovl  $0, %eax
 leave
```

```
L4:
 mrmovl -12(%ebp), %edi
 irmovl $0, %ebx
 subl %ebx, %edi                 } if((i>=0)
 js L3
 mrmovl -12(%ebp),%edi
 irmovl $511, %ebx
 subl %ebx, %edi                 } &&(i<BUFSIZE))
 jg L3

 mrmovl -12(%ebp), %eax
 mrmovl -4(%ebp), %edx
 rrmovl %eax, %edi               } buffer[i] = a;
 sall $2, %edi
 addl %ebp, %edi
 rmmovl %edx, -2060(%edi)
L3:
    mrmovl  -12(%ebp), %edi
    iaddl   $1, %edi
    rmmovl  %edi, -12(%ebp)
L2:
    mrmovl  -12(%ebp), %eax
    mrmovl  -8(%ebp), %edi
    subl    %edi, %eax
    jl L4
    irmovl  $0, %eax
    leave
```

Figura 3. Versões vulnerável (esquerda) e protegida (direita) de um programa em linguagem de montagem.

Como se pode perceber, há um custo para se verificar limites. Isso, por sua vez, indica que ABCs são computacionalmente caros num contexto cujas aplicações possuem certo grau de complexidade e, por conseguinte, demandam um grande número de verificações para se protegerem. De fato, há trabalhos que evidenciam que o emprego de ABCs – quando implementado em software – pode acarretar sobrecarga (overhead) de até 70% no tempo de execução dos programas alvo [3].

III. Trabalhos Relacionados

Há na literatura uma infinidade de técnicas para proteger sistemas computacionais ([3], [4], [5]), abaixo, vamos nos concentrar àquelas relativas a ataques de Violação de Memória. A maioria das propostas de defesa, nesse caso, subdividem-se naquelas baseadas na Análise Estática e na Análise Dinâmica [6], [7].

Dentre as estratégias que abordam a Análise Estática de código [8], [9], é possível identificar diferentes técnicas que tentam sanar as falhas de Violação de Memória. Por exemplo, a ferramenta *ITS4* ([9]) faz análise léxica do código. O analisador identifica as funções do programa e, para cada uma delas, faz uma busca em uma lista de funções conhecidamente vulneráveis. O objetivo da ferramenta é dar suporte ao desenvolvimento durante a fase de codificação, destacando potenciais problemas de segurança à medida que o código é concebido. Já a ferramenta *Array Checker* ([4]), consiste de uma análise simbólica *bottom-up*, *inter-procedural* e sensível ao fluxo. Ela faz uma série de transformações no código, até chegar a uma representação simbólica do programa. Nesse estágio, a ferramenta aplica algoritmos de análise de fluxo e identifica as funções que devem ser revisadas pelo desenvolvedor.

A Análise Estática é uma fase importante do processo de desenvolvimento de software seguro. Acreditamos que tais técnicas aliadas às instruções de ABCs em hardware, propostas nesse trabalho, podem compor uma solução elegante e eficiente para as falhas de Violação de Memória.

A outra forma de se mitigar o problema de Violação de Memória é através da Análise Dinâmica [1], [10]. Nessa abordagem, diferentes técnicas de instrumentação de código são utilizadas. Entre elas, aquelas que mais se assemelham ao nosso trabalho fazem uso de modificações em hardware para melhorar os níveis de segurança e desempenho das aplicações ([11], [12], [13]).Uma dessa soluções, por exemplo, tem como objetivo proteger dados de controle, Endereços de Retorno (ERs) e ponteiros de funções, de ataques de desvio de fluxo [12]. Nesse esquema é mantida uma memória paralela onde há um *bit* de segurança correspondente a cada palavra da memória principal. Quando dados cruzam os limites entre os domínios da aplicação (dados de entrada), o sistema operacional muda seu modo de operação para o modo de escrita segura. Enquanto ativo, qualquer dado escrito em uma palavra da memória faz com que o *bit* de segurança correspondente seja marcado. Após a escrita, o sistema operacional volta ao modo de operação normal e a aplicação continua executando. Se, no futuro, o programa tenta usar como dado de controle uma palavra que teve o *bit* de segurança marcado, o processador levanta uma exceção. Finalmente, a aplicação será abortada com uma falha de segmentação ao invés de mudar o fluxo de programa para uma região de memória potencialmente perigosa. Para alcançar seus objetivos, os autores modificaram a semântica de instruções em um emulador de arquitetura IA32. Por exemplo, as instruções `return`, `call` e `jump` passaram a validar o *bit* seguro antes de completar suas operações. Já as instruções de acesso à memória foram modificadas para propagar o *bit* de segurança quando necessário.

Outro exemplo de solução em hardware é adotado em [13]. Nesse caso, a proposta é a proteção específica de Sistemas Embarcados contra ataques de mudança de fluxo de controle. Sua técnica consiste na utilização de uma pilha

especial para armazenamento de ERs. A nova pilha é mantida na memória de dados em uma posição diferente da pilha tradicional, além disso, ela é protegida pelo hardware. O mecanismo de segurança em hardware consiste em permitir que apenas instruções que manipulam ERs possam escrever na pilha de retorno. Para tal, os autores adicionaram um *bit* de identificação das instruções que podem modificar ERs na arquitetura AVR. Dessa forma, o acesso aos ERs é garantido apenas se o *bit* de identificação está ativo.

A principal diferença entre os trabalhos citados e a solução apresentada nesse artigo diz respeito ao alcance das soluções. Os trabalhos apresentados focam no problema de BOF, especificamente para desvio de fluxo de controle e ponteiros de funções. Isso significa que, mesmo utilizando esses mecanismos, um sistema computacional pode estar vulnerável a ataques de BOR. A nossa solução se propõe a resolver todos os diferentes tipos de vulnerabilidades de Violação de Memória.

Assim como a nossa solução, existem trabalhos que objetivam mitigar os ataques de Violação de Memória em sua totalidade e que, além disso, propõe instruções que aceleram, por serem executadas em hardware, o processo de verificação de limites. É o caso do conjunto de Extensões de Proteção de Memória (*Memory Protection Extensions* – MPX) que estará disponível na próxima geração dos processadores Intel [14], com entrada ao mercado prevista para meados de 2015. O MPX introduz uma família de instruções relacionadas à verificação de limites de arranjos, assim como registradores especiais e uma estrutura de diretórios de tabelas, em hardware, para armazenamento dos mesmos.

Há algumas diferenças entre nossa solução e o ferramental fornecido pela Intel com o MPX. A primeira delas é que o MPX introduz registradores especiais para armazenamento dos limites de arranjos (e respectivo diretório de tabelas para armazená-los), enquanto que a nossa solução considera o uso dos registradores de propósito geral, já existentes no processador. Essa decisão deve ser analisada frente a dois aspectos: i) implementação do hardware e ii) eficiência do software. É inevitável que os componentes "extras" existentes no MPX gerem um circuito maior, incorram em custos adicionais e deixem o projeto do hardware significativamente mais complexo. Por outro lado, a utilização dos registradores de propósito geral pode incorrer em *overhead* de software devido ao frequente número de *spills/fills*. Nosso projeto opta por simplicidade, generalidade e por um hardware enxuto, permitindo adoção sem necessidade de mudanças disruptivas, mas atingindo nosso objetivo primordial que é a verificação eficiente (quando comparado à literatura atual) de limites de arranjos.

A segunda diferença é quanto ao número de passos para conclusão do acesso seguro à memória, ou seja, utilizando as instrução que provêem ABCs em hardware. No MPX o processo envolve uma instrução para criação dos limites do arranjo nos registradores especiais (`bndmk`), uma instrução de verificação do limite inferior do arranjo (`bndcl`), uma instrução de verificação do limite superior (`bndcu`) e, finalmente, o acesso por uma instrução `mov` tradicional. A nossa proposta conclui o acesso à memória em uma única instrução, através de um aproveitamento inteligente dos recursos do estágio *Execute* do *pipeline*, reduzindo o número total de ciclos em relação ao MPX.

IV. INSTRUÇÕES SEGURAS

Nesta seção apresentamos uma solução de segurança que emprega instruções em hardware para proteger sistemas computacionais contra ataques de Violação de Memória. A seguir, apresentamos o projeto (Seção A.) e implementação (Seção B.) da nossa solução. Nossa proposta substitui as instruções de leitura (*load*) e escrita (*store*) tradicionais por duas instruções análogas. No entanto, essas novas instruções verificam limites de arranjos e são, portanto, seguras.

A. Projeto

Abaixo, apresentamos as decisões do projeto das instruções seguras.

Arquitetura. Optamos por projetar nossa solução sobre a arquitetura Y86, a qual foi inspirada na arquitetura IA32 e é atualmente uma das mais empregadas pela academia. Embora menor que o da IA32, o ISA da Y86 permite a execução de programas suficientemente complexos para se avaliar desempenhos. Ademais, informações detalhadas sobre o projeto da arquitetura estão públicas [15].

Registradores. Nossas instruções requerem quatro registradores. Como são instruções de acesso à memória, dois registradores são necessários, isto é, um para armazenar o endereço de acesso à memória e outro para armazenar o dado a ser escrito/lido, dependendo se a operação é de escrita ou leitura. Além disso, o acesso à memória é garantido por meio das verificações (comparações) de limites inferior e superior do arranjo. Portanto, dois registradores adicionais são necessários para armazenar esses limites. Com isso, chegamos à seguinte sintaxe de instrução: `s-movl rA, rB, rX, rY`.

Comparações. Em nosso projeto, há duas comparações. A primeira verifica o limite superior do arranjo frente ao endereço efetivo de acesso à memória. Tal operação foi projetada de maneira similar à instrução `subl`. Note-se que a subtração é computada como `rB - rA` (Fig. 4). Assim, podemos utilizar o registrador `rB` para armazenar o endereço do limite superior do arranjo e o registrador `rA` para armazenar o endereço efetivo de acesso à memória. A segunda comparação, verificação do limite inferior, também deve ser realizada por uma subtração. Contudo, a ALU do estágio *Execute* é capaz de computar somente uma operação por ciclo. Portanto, com o intuito de não gerar uma sobrecarga de tempo à nova instrução, adicionamos um componente de hardware (destacado em vermelho na Fig. 4, estágio *Execute*) a esta etapa que, em paralelo, calcula a diferença entre o endereço efetivo de acesso e o limite inferior do arranjo. Assim, o registrador `rX` armazena o limite inferior do arranjo e o novo componente computa a operação `rA - rX`.

Estágio	srmmovl rA,rB,rX,rY	smrmovl rA,rB,rX,rY
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] rX:rY ← M_1[PC + 2] valP ← PC + 3 PC ← valP	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] rX:rY ← M_1[PC + 2] valP ← PC + 3 PC ← valP
Decode	valA ← R[rA] valB ← R[rB] valX ← R[rX] valY ← R[rY]	valB ← R[rB] valX ← R[rX]
Execute	sec_{upper} ← (valB – valA > 0) sec_{lower} ← (valA – valX ≥ 0)	sec_{upper} ← (valB – valA > 0) sec_{lower} ← (valA – valX ≥ 0)
Memory	if (sec_{upper} && sec_{lower}) M_4[valA] ← valY else INTERRUPTION	if (sec_{upper} && sec_{lower}) valM ← M_4[valA] else INTERRUPTION
Write Back		R[rY] ← valM

Figura 4. Estágios das instruções seguras com destaque aos componentes adicionados/alterados.

Sinais de Controle. Consideramos o sinal de controle que valida o acesso à memória quanto ao limite superior do arranjo é verdadeiro se o resultado da subtração for estritamente maior que zero. Já o sinal de controle que valida o acesso à memória acerca do limite inferior do arranjo (isto é, primeira posição de memória do arranjo) é verdadeiro se o resultado da subtração for maior ou igual a zero. Ainda acerca dos sinais de controle, modificamos a arquitetura para que os mesmos possam ser avaliadas ainda durante a instrução corrente; dado que na arquitetura Y86 original sinais de controle gerados por uma subtração só podem ser avaliados posteriormente, durante a instrução seguinte.

Tratamento. Diante uma tentativa de acesso fora dos limites do arranjo, detectada por uma das comparações descritas acima, o processador deve sinalizar uma interrupção de hardware.

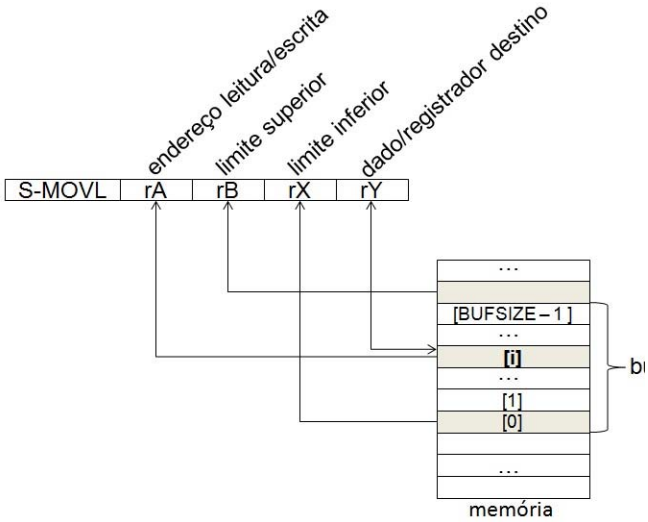

Figura 6. Representação da semântica das instruções seguras.

Terminologia. A denominação das novas instruções seguiram o padrão existente e foram, então, chamadas de `srmmovl`(*secure register memory movl*) e `smrmovl` (*secure memory register movl*).

Os estágios de execução ao longo do *pipeline* da arquitetura Y86 modificada são exibidos na Fig. 4. Nela, as operações adicionadas/modificadas foram destacadas. Já na Fig. 5 é apresentada a semântica das instruções. Por fim, na Fig. 6 é apresentado o caminho de dados (*datapath*) modificado da arquitetura Y86. Novamente, destacamos os componentes adicionados/alterados.

B. *Implementação*

Para a implementação das instruções seguras foi utilizado o simulador do processador Y86 descrito em [15] e disponível online [16]. Um aspecto importante na adoção desse simulador é a possibilidade de reportar o número de instruções e de ciclos necessários para a conclusão dos programas, o que possibilita a comparação entre ABCs realizados em software e hardware. Além disso, juntamente com o simulador, há um montador (assembler) Y86. O código deste simulador é aberto, o que nos permitiu adaptá-lo para reconhecer as novas instruções.

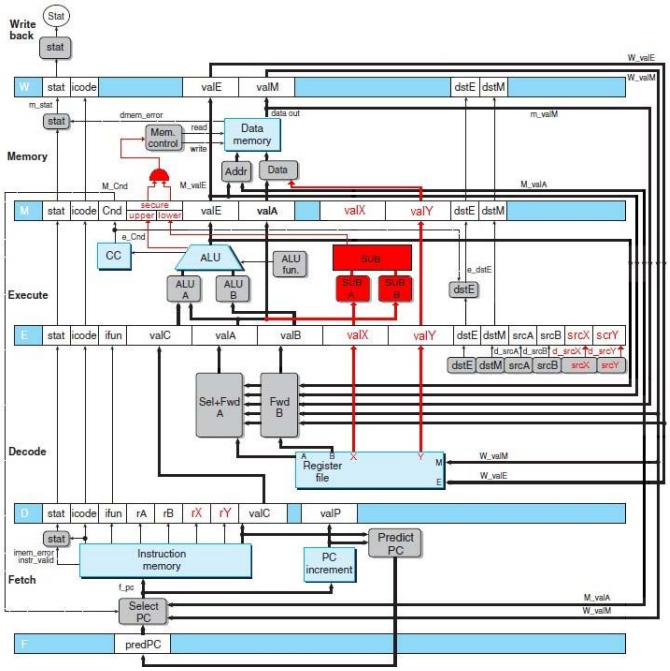

Figura 5. Caminho de dados da arquitetura Y86 modificada.

V. AVALIAÇÃO

Nesta seção descrevemos a metodologia (Seção A.) utilizada na avaliação da nossa solução e os resultados obtidos (Seção B.).

A. Metodologia

Nossa solução foi avaliada sobre versões ligeiramente modificadas dos programas: *bubblesort* e *quicksort,* pertencentes ao *benchmark* Stanford [17] . Em cada um deles foi considerado um arranjo de tamanho igual a 400 inteiros. O arranjo é inicializado em ordem decrescente e a rotina de ordenação é, então, chamada.

Para efeitos de comparação são criadas as seguintes versões de cada programa:

1. **baseline**: os arranjos são acessados sem quaisquer ABCs.
2. **ABC software**: a cada acesso a arranjos, tanto em atribuições quanto leituras, é utilizado um ABC via software.
3. **ABC hardware**: o código em linguagem de montagem da versão original é analisado de forma a identificar as instruções de *load* e *store* que acessam arranjos. Essa instruções são, então, trocadas por uma sequência equivalente de instruções concluída por uma instrução segura.

Cada um dos programas é executado no simulador Y86, fornecendo os resultados de número de ciclos e instruções necessários para suas conclusões. Na próxima seção esses resultados são reportados e analisados.

B. Resultados

Aqui serão apresentados e discutidos os resultados, sintetizados na Tabela I, para os dois estudos de caso considerados neste trabalho (*quicksort* e *bubblesort*).

TABELA I. RESULTADOS DAS SIMULAÇÕES DAS 3 VERSÕES DE CADA UM DOS PROGRAMAS DE AVALIAÇÃO

bubblesort	**Números**			**Sobrecarga**	
	Instr.	**Ciclos**	**CPI**	**Instr.**	**Ciclos**
baseline	3293834	4337245	1.32		
ABC software	7373623	10816629	1.47	123.86%	149.39%
ABC hardware	5213834	6257245	1.20	58.29%	44.27%
quicksort	**Números**			**Sobrecarga**	
	Instr.	**Ciclos**	**CPI**	**Instr.**	**Ciclos**
baseline	20307	26505	1.31		
ABC software	36206	50552	1.40	78.29%	90.73%
ABC hardware	28439	34637	1.22	40.05%	30.68%

Na tabela, para cada estudo de caso, a primeira coluna mostra o método utilizado. Nas próximas três colunas são apresentados o número total de instruções, o número total de ciclos e o CPI (Ciclos Por Instrução), respectivamente. Finalmente, as duas últimas colunas mostram a sobrecarga na quantidade de instruções e ciclos, respectivamente, quando comparados com o *baseline*. Nos dois casos, a aplicação das instruções seguras apresentou sobrecarga menor que as verificações via software, tanto em número de instruções quanto em número de ciclos.

No programa *quicksort*, os ABCs em software causaram uma sobrecarga de 78,29% no total de instruções e 90,73% em números de ciclos. Já a versão em hardware reduziu as sobrecargas para 40,05% e 30,68%, respectivamente. Esse resultado mostra que nossa solução precisou de 31,5% ciclos a menos que a versão via software para concluir a execução do programa.

Já para o programa *bubblesort*, os resultados da proposta apresentada neste trabalho foram ainda melhores, alcançando uma redução de 42,2% do número de ciclos em relação aos ABCs via software. A sobrecarga desse parâmetro caiu de 149,39% para 44,27% e no número de instruções a queda foi de 123,86% para 58,29%.

A maior redução percentual de número de ciclos em relação à versão via software observada no programa *bubblesort* é reflexo da complexidade dos algoritmos avaliados. Como o *bubblesort* é mais ineficiente que o *quicksort*, o número de comparações e trocas entre os elementos do arranjo é maior, o que demanda mais instruções de acesso à memória.

Note-se que em todos os programas analisados os ABCs em software causaram sobrecarga de ciclos superior à sobrecarga de instruções, o que resultou em um aumento do parâmetro CPI. Essa característica é justificada pois ABCs em software são traduzidos, entre outras, em instruções de desvio condicional que, por sua vez, podem levar a uma situação de *branch misprediction*, onde alguns ciclos extras são necessários para ajustar a execução do programa. Isso não é observado em nossa solução pois as instruções adicionais necessárias para concluir um acesso seguro são completamente paralelizáveis, ou seja, não incorrem na interrupção de nenhum estágio do *pipeline*, gerando uma redução no CPI.

A Fig. 7 apresenta o gráfico de *speedup* da nossa solução em relação à verificação de limites via software. Esse parâmetro indica o quão mais rápido os programas que verificam limites usando a nossa proposta podem executar. Os resultados mostram que nossa solução em hardware possibilita acelerar a execução dos programas *bubblesort* e *quicksort* em 1,73 e 1,46, respectivamente, quando comparados com a abordagem em software.

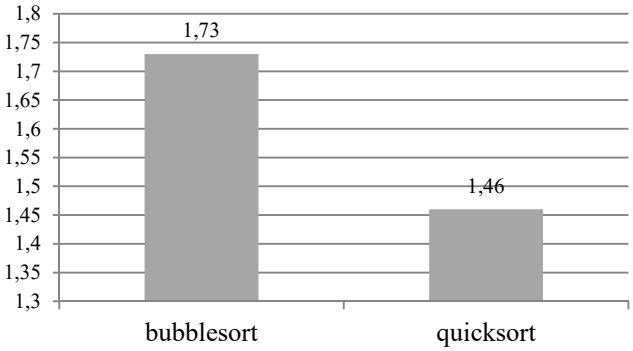

Figura 7. Gráfico de speedup das instruções seguras em relação a ABCs em software.

A análise média dos resultados apontam que a utilização da nossa estratégia em hardware diminui a sobrecarga causada or ABCs via software em cerca de 36% de número de ciclos e em 25% de número de instruções. Já o *speedup* indica que nossa solução executa os programas 1,59 vezes mais rápido que a solução por software.

VI. CONCLUSÕES

Programas escritos na linguagem C estão sujeitos à vulnerabilidades de Violação de Memória, tais como BOF e BOR, por deixar a cargo dos desenvolvedores a inserção de ABCs. As propostas de automatização de ABCs são, em sua maioria, técnicas de instrumentação via software, que tendem a prejudicar o desempenho dos sistemas já que aumentam significativamente a quantidade de código. O objetivo deste trabalho foi propor uma solução eficiente de verificação de limites em hardware. Nossa solução disponibiliza instruções seguras de leitura e escrita na arquitetura Y86. O resultado das comparações da nossa solução frente à estratégia em software apontaram melhoras de cerca de 59% no desempenho dos programas analisados.

AGRADECIMENTOS

Agradecemos à Intel, ao CNPq e à FAPEMIG pelo financiamento desde projeto.

REFERÊNCIAS

[1] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang et al., "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in the 7th USENIX Security Symposium, 1998, pp. 346–355.

[2] D. Dhurjati, S. Kowshik, and V. Adve, "SAFECode: enforcing alias analysis for weakly typed languages," in ACM SIGPLAN conference on Programming language design and implementation - (PLDI '06), 2006, pp. 144–157.

[3] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: a fast address sanity checker," in Usenix Annual Technical Conference (ATC'12), 2012, pp. 28–28.

[4] Y. Xie, A. Chou, and D. Engler, "Archer: using symbolic, path-sensitive analysis to detect memory access errors," in ACM SIGSOFT Software Engineering Notes, 2003, pp. 327–336.

[5] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention." in the 10th Annual Network and Distributed System Security Symposium (NDSS'03), 2003.

[6] M. F. T. Ferreira, T. de Souza Rocha, G. Breves, E. F. Martins, and E. Souto, "Análise de vulnerabilidades em sistemas computacionais modernos: Conceitos, exploits e proteções," Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, 2012, Curitiba. (SBSeg'12), 2012.

[7] B. Silva, D. C. da Silva Jr, E. M. Souza, F. Pereira, F. A. Teixeira, H. C. Wong, H. Nazaré, I. Maffra, J. Freire, W. F. Santos et al., "Segurança de software em sistemas embarcados: Ataques & defesas," Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, 2013, Manaus. (SBSeg'13), 2013.

[8] B. R. Silva, F. M. Q. Pereira, and L. B. Oliveira, "Uma representação intermediária para a detecção de vazamentos implícitos de informação," Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, 2013, Manaus. (SBSeg'13), 2013.

[9] J. Viega, J.-T. Bloch, Y. Kohno, and G. McGraw, "ITS4: A static vulnerability scanner for c and c++ code," in the 16th Computer Security Applications (ACSAC'00), 2000, pp. 257–267.

[10] T. Bell, "The concept of dynamic analysis," ACM SIGSOFT Software Engineering Notes, vol. 24, no. 6, pp. 216–234, Oct. 1999. [Online]. Available: http://doi.acm.org/10.1145/318774.318944

[11] H. Ozdoganoglu, T. Vijaykumar, C. E. Brodley, B. A. Kuperman, and A. Jalote, "Smashguard: A hardware solution to prevent security attacks on the function return address," IEEE Transactions on Computers, vol. 55, no. 10, pp. 1271–1285, 2006.

[12] K. Piromsopa and R. J. Enbody, "Secure bit: Transparent, hardware buffer-overflow protection," IEEE Transactions on Dependable and Secure Computing, vol. 3, no. 4, pp. 365–376, 2006.

[13] A. Francillon, D. Perito, and C. Castelluccia, "Defending embedded systems against control flow attacks," in the first ACM workshop on Secure execution of untrusted code, 2009, pp. 19–26.

[14] Intel Corporation, "Intel Architecture Instruction Set Extensions Programing Reference," http://download-software.intel.com/sites/default/files/ 319433-015.pdf., July 2013.

[15] R. Bryant and O. David Richard, Computer systems: a programmer's perspective. Prentice Hall, 2003.

[16] http://csapp.cs.cmu.edu/public/labs.html

[17] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in Code Generation and Optimization (CGO'04), 2004, pp. 75–88.

[18] http://en.wikipedia.org/wiki/Morris_worm

[19] http://en.wikipedia.org/wiki/Heartbleed

Antonio L. Maia Neto possui graduação em Engenharia de Computação pela FEEC/Unicamp. Atualmente é Mestrando em Ciência da Computação pelo DCC/UFMG. Possui vasta experiência em projeto e desenvolvimento de sistemas de segurança para sistemas embarcados e redes de computadores. Durante a graduação participou do Grupo de Tratamento de Imagens da Universidade Politécnica de Madrid. Suas áreas de interesse são segurança da informação, criptografia aplicada e arquitetura de computadores.

Leandro T. C. Melo é engenheiro, mestre em Engenheria Elétrica e candidato a Doutor em Ciência da Computação pela Universidade Federal de Minas Gerais. Possui ampla experiência com desenvolvimento de software e é especialista em C++, tendo contribuído com uma IDE conhecida na comunidade open-source. Trabalhou por anos na Alemanha em renomadas empresas de software, com temas ligados a compiladores e bancos de dados em-memória.

Omar P. Vilela Neto possui graduação em Engenharia de Computação, mestrado e doutorado em Engenharia Elétrica pela Pontifícia Universidade Católica do Rio de Janeiro. Atualmente é professor adjunto do Departamento de Ciência da Computação (DCC) da Universidade Federal de Minas Gerais (UFMG). Suas áreas de interesse de pesquisa são nanotecnologia computacional, nanocomputação e arquitetura de computadores.

Fernando M. Q. Pereira recebeu o título de doutor em Ciência da Computação pela Universidade da Califórnia, Los Angeles, em 2008. Atualmente ele é professor adjunto no Departamento de Ciência da Computação (DCC) da Universidade Federal de Minas Gerais (UFMG), onde ensina linguagens de programação e análise estática de programas. Fernando tem pesquisado maneiras de construir compiladores que gerem código mais seguro. Desse esforço resultaram diversas melhorias em compiladores de grande popularidade. Um exemplo é a análise que previne a injeção de código SQL usada por PHC, o compilador de PHP.

Leonardo B. Oliveira é professor do Programa de Pós-Graduação em Ciência da Computação da UFMG, onde atua na área de Segurança Digital e Computação Ubíqua. Doutorou-se no Laboratório de Criptografia Aplicada da Unicamp com período sanduíche no Grupo de Criptografia da Escola de Computação da Universidade de Dublin. Também trabalhou em parceria com a Microsoft Research, Intel Labs, Information Security Group - Royal Holloway e Palo Alto Research Center e foi professor da Universidade Estadual de Campinas. Publicou artigos em veículos de impacto, como EWSN, ACM/IEEE IPSN, JPDC, PLoS ONE e Signal Processing. Seu trabalho foi agraciado com as seguintes distinções: Microsoft Research PhD Fellowship Award, Prêmio do Concurso de Teses de Doutorado da Sociedade Brasileira de Computação, IEEE Young Professional Award, Intel Strategic Research Alliance Award e Intel Security Curriculum Program Award. Trabalhou na indústria, coordenando projetos para agências governamentais e/ou grupos bancários na área de Segurança Digital. Possui aproximadamente 90 trabalhos publicados, cerca de 1500 citações e h-index igual a 20. Dentre os trabalhos, destacam-se o NanoECC, SecLEACH, SecureTWS e TinyPBC. Leonardo coordena o Grupo de Pesquisa Segurança Digital, Criptografia e Privacidade certificado pelo CNPq e é coordenador geral da edição de 2014 do Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg'14). É membro do comitê consultivo da Comissão Especial de Segurança da Informação e Sistemas Computacionais (CESeg). Ademais, coordenou/coordena projetos Universal CNPq, Universal FAPEMIG e projetos da Intel labs, Microsoft Research e LGE.