S-MOVL: Protegendo Sistemas Computacionais contra Ataques de Violação de Memória por meio de Instruções em Hardware

Antonio L. Maia Neto, Omar P. Vilela Neto, Fernando M. Q. Pereira, Leonardo B. Oliveira

[1] Departamento de Ciência da Computação (DCC)
Universidade Federal de Minas Gerais (UFMG) - Belo Horizonte, MG - Brazil

{lemosmaia,omar,fernando,leob}@dcc.ufmg.br

Abstract. *The C programming language does not prevent ouf-of-bounds memory access and thus gives room to attacks such as buffer overflow and buffer overread. There exist several techniques to secure C programs. Nevertheless, these methods are usually implemented via software and therefore tend to slow down programs and frequently compromise performance of applications. This work aims at presenting a hardware solution able to check bounds efficiently.*

Resumo. *A linguagem C não verifica limites de arranjo e abre brechas para ataques de violação de memória, tais como buffer overflow e o buffer overread. A maioria das propostas para adicionar essa funcionalidade à linguagem são implementadas em software, estratégia que prejudica o desempenho de aplicações. Neste trabalho apresentamos uma solução em hardware capaz de realizar a verificação de limites eficientemente.*

1. Introdução

A linguagem C é uma das mais empregadas em meio à comunidade de programadores e foi concebida com o foco na eficiência, permitindo que as aplicações executem numa velocidade compatível com os recursos computacionais muitas vezes escassos. 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. Ao invés disso, a linguagem deixa a cargo do programador inserir ABCs quando este achar necessário.

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[1], 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[2], que no ano passado deixou a comunidade de segurança digital em polvorosa, ao explorar a vulnerabilidade 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 ([Dhurjati et al. 2006], por exemplo). Grosso modo,

[1] http://en.wikipedia.org/wiki/Morris_worm
[2] http://en.wikipedia.org/wiki/Heartbleed

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*).

O objetivo deste artigo é conceber instruções em linguagem de máquina que realizem, de forma segura, o transporte de dados da memória para registradores e vice-versa. Desta forma, deixamos a cargo do hardware – mais eficiente que o software – checar os limites de arranjos. Nossa proposta, chamada de S-MOVL, cria versões seguras das instruções de leitura (*load*) e escrita (*store*) em memória, onde os limites inferior e superior do arranjo manipulado são verificados antes da conclusão do acesso à memória.

2. Trabalhos Relacionados

Há na literatura uma infinidade de técnicas para proteger sistemas computacionais. Por motivos de restrição de espaço, vamos nos concentrar àquelas relativas a ataques de Violação de Memória e que, além disso, lancem mão de modificações em hardware para melhorar os níveis de segurança e de desempenho das aplicações.

Grande parte das propostas de modificações em hardware para conter ataques de Violação de Memória tem como principal objetivo proteger dados de controle, endereços de retorno e ponteiros de funções, de ataques de desvio de fluxo [Piromsopa and Enbody 2006]. Esse tipo de abordagem, ao atacar especificamente problemas de BOF, não impede que um sistema computacional esteja vulnerável a ataques de BOR, por exemplo. Nossa solução visa prover uma forma de defesa que abrange todos os tipos de ataques de Violação de Memória.

Assim como a nossa solução, existem trabalhos que, através de modificações em hardware, objetivam mitigar os ataques de Violação de Memória em sua totalidade. Dentre essas propostas está o conjunto de Extensões de Proteção de Memória (*Memory Protection Extensions* – MPX) adotadas na nova geração dos processadores Intel [Intel Corporation 2013]. Esse conjunto de extensões consiste em uma série de novas (oito) instruções de gerenciamento e verificação de limites de arranjo. Considerando o uso da extensão MPX, o processo de acesso seguro a uma posição de memória é: **i)** carregar os limites superior e inferior de um arranjo em registradores especiais – instrução `BNDMK`; **ii)** verificar os limites superior e inferior – instruções `BNDCL` e `BNDCU`, respectivamente; e, finalmente, **iii)** concluir o acesso (escrita ou leitura) à memória – instruções tradicionais de `mov`.

A principal diferença entre nossa solução e o ferramental fornecido em MPX é a possibilidade de, no nosso caso, concluir a verificação de limites e o acesso à memória por meio de uma única instrução – `srmmovl` para escrita e `smrmovl` para leitura. Outro ponto que difere as abordagens é com relação às mudanças em hardware. Enquanto MPX cria novos registradores internos no processador, nós nos atemos aos registradores de propósito geral já existentes.

3. S-MOVL

Optamos por projetar nossa solução sobre a arquitetura Y86. A arquitetura Y86 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 [Bryant and David Richard 2003].

As nossas instruções – da forma **`s-movl rA, rB, rX, rY`** – consistem em versões seguras das instruções de leitura e escrita na memória que, antes de concluir um acesso, validam os limites do arranjo. A verificação dos limites é feita por comparações, que, no caso do Y86, são resultado da avaliação de subtrações. Portanto, a primeira comparação foi baseada na instrução já existente `subl rA, rB`, onde a operação `rB - rA` é computada no estágio *Execute* do *pipeline*. Logo, podemos utilizar o registrador `rB` para armazenar o endereço do limite superior do arranjo e o registrador `rA` para armazenar o endereço de acesso. Para a segunda comparação, verificação do limite inferior, é importante notar que o estágio *Execute* é incapaz de computar mais de uma operação por ciclo. Assim, com o intuito de não gerar uma sobrecarga de tempo à nova instrução, adicionamos um novo componente de hardware a esta etapa que, em paralelo, calcula a diferença entre o endereço de acesso e o limite inferior do arranjo. E então, o registrador `rX` armazena o limite inferior do arranjo e o novo componente computa a operação `rA - rX`.

Optamos por seguir a convenção da linguagem C considerando que o sinal de controle que valida o acesso à memória quanto ao limite superior do arranjo é verdadeiro se o resultado da primeira subtração for estritamente maior que zero. Analogamente, em relação ao limite inferior do arranjo, o acesso à memória é garantido se o resultado da segunda subtração for maior ou igual a zero. A tentativa de acesso fora dos limites do arranjo, detectada por uma das comparações descritas acima, deve, obrigatoriamente, ser sinalizada por uma exceção de hardware.

Os estágios de execução ao longo do *pipeline* da arquitetura Y86 modificada são exibidos na Tabela 1. Nela, as operações adicionadas/modificadas foram destacadas.

Stage	**`srmmovl rA,rB,rX,rY`**	**`smrmovl rA,rB,rX,rY`**
Fetch	$icode : ifun \leftarrow M_1[PC]$	$icode : ifun \leftarrow M_1[PC]$
	$rA : rB \leftarrow M_1[PC+1]$	$rA : rB \leftarrow M_1[PC+1]$
	$\mathbf{rX : rY \leftarrow M_1[PC+2]}$	$\mathbf{rX : rY \leftarrow M_1[PC+2]}$
	$valP \leftarrow PC+3$	$valP \leftarrow PC+3$
	$PC \leftarrow valP$	$PC \leftarrow valP$
Decode	$valA \leftarrow R[rA]$	$valA \leftarrow R[rA]$
	$valB \leftarrow R[rB]$	$valB \leftarrow R[rB]$
	$\mathbf{valX \leftarrow R[rX]}$	$\mathbf{valX \leftarrow R[rX]}$
	$\mathbf{valY \leftarrow R[rY]}$	
Execute	$\mathbf{if(valB - valA > 0) \rightarrow exception}$	$\mathbf{if(valB - valA > 0) \rightarrow exception}$
	$\mathbf{if(valA - valX \geq 0) \rightarrow exception}$	$\mathbf{if(valA - valX \geq 0) \rightarrow exception}$
Memory	$\mathbf{M_4[valA] \leftarrow valY}$	$\mathbf{valM \leftarrow M_4[valA]}$
Write back		$\mathbf{R[rY] \leftarrow valM}$

Tabela 1. Estágios do *pipeline* do Y86 para das instruções seguras.

4. Resultados

Nossa solução foi avaliada sobre uma versão ligeiramente modificada do programa *bubblesort* pertencente ao *benchmark* Stanford. Foi considerado apenas um arranjo de tamanho igual a 400 inicializado em ordem decrescente para, então, ser ordenado. Foram criadas as seguintes versões do programa: **original** – os arranjos são acessados sem quaisquer ABCs; **baseline** – a cada acesso a arranjos, tanto atribuições quanto leituras, é utilizado um ABC via software; **S-MOVL** – o código em linguagem de montagem da versão

original é analisado de forma a identificar as instruções de *load* ou *store* que acessam arranjos. Essa instruções são, então, trocadas por uma sequência equivalente de instruções concluída por uma instrução S-MOVL.

Cada uma das versões foi executada no simulador Y86 descrito em [Bryant and David Richard 2003] e disponível online[3], fornecendo o total de número de ciclos e instruções durante cada execução. Os resultados, sintetizados na Tabela 2, mostram que os ABCs em software causaram uma sobrecarga de 123,86% no total de instruções e 149,39% em número de ciclos. Já a versão em hardware reduziu as sobrecargas para 58.29% e 44,27%, respectivamente. Esse resultado mostra que S-MOVL precisou de 57,85% ciclos a menos que o *baseline* para executar.

bubblesort	**Números**			**Sobrecarga**	
	Instruções	**Ciclos**	**CPI**	**Instruções**	**Ciclos**
original	3293834	4337245	1.32		
baseline	7373623	10816629	1.47	123.86%	149.39%
S-MOVL	5213834	6257245	1.20	58.29%	44.27%

Tabela 2. Resultados das simulações das 3 versões do programa *bubbesort*

5. Conclusão

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 dos programas. O objetivo deste trabalho foi propor uma solução eficiente de verificação de limites em hardware, através de instruções seguras de leitura e escrita na memória para a arquitetura Y86. O resultado das comparações da nossa solução frente à estratégia em software apontaram melhoras de cerca de 58% no desempenho dos programas analisados.

Como continuidade do projeto pretendemos avaliar nossa solução em relação a outras estratégias em hardware, principalmente a extensão MPX.

Referências

Bryant, R. and David Richard, O. (2003). *Computer systems: a programmer's perspective*. Prentice Hall.

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

Intel Corporation (2013). Intel Architecture Instruction Set Extensions Programing Reference. `http://download-software.intel.com/sites/default/files/319433-015.pdf`.

Piromsopa, K. and Enbody, R. J. (2006). Secure bit: Transparent, hardware buffer-overflow protection. *IEEE Transactions on Dependable and Secure Computing*, 3(4):365–376.

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

