

**SMOV: PROTECTING PROGRAMS AGAINST BUFFER
OVERFLOW IN HARDWARE**

ANTONIO LEMOS MAIA NETO

**SMOV: PROTECTING PROGRAMS AGAINST BUFFER
OVERFLOW IN HARDWARE**

Dissertação apresentada ao Programa de Pós-
-Graduação em Ciência da Computação do
Instituto de Ciências Exatas da Universidade
Federal de Minas Gerais como requisito par-
cial para a obtenção do grau de Mestre em
Ciência da Computação.

ORIENTADOR: LEONARDO BARBOSA E OLIVEIRA
COORIENTADOR: OMAR PARANAIBA VILELA NETO

Belo Horizonte
Fevereiro de 2015

ANTONIO LEMOS MAIA NETO

**SMOV: PROTECTING PROGRAMS AGAINST BUFFER
OVERFLOW IN HARDWARE**

Master's thesis presented to the Graduate Program in Ciência da Computação of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Master in Ciência da Computação.

ADVISOR: LEONARDO BARBOSA E OLIVEIRA
CO-ADVISOR: OMAR PARANAIBA VILELA NETO

Belo Horizonte

February 2015

© 2015, Antonio Lemos Maia Neto.
Todos os direitos reservados.

Maia Neto, Antonio Lemos

M217s SMOV: Protecting Programs against Buffer Overflow
in Hardware / Antonio Lemos Maia Neto. — Belo
Horizonte, 2015
xxii, 52 p. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de
Minas Gerais

Orientador: Leonardo Barbosa e Oliveira
Coorientador: Omar Paranaíba Vilela Neto

1. Computação - Teses. 2. Computadores - Medidas de
segurança. 3. Redes de computadores - Medidas de
segurança - Teses. I. Orientador.II. Coorientador.III. Título.

CDU 519.6*22(043)



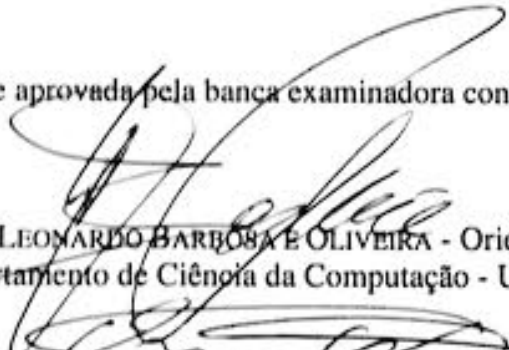
UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO


SMOV: protecting programs against buffer overflow in hardware

ANTONIO LEMOS MAIA NETO


Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:



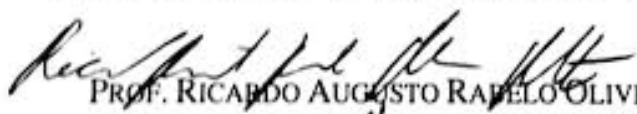
PROF. LEONARDO BARBOSA E OLIVEIRA - Orientador
Departamento de Ciência da Computação - UFMG



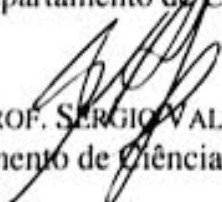
PROF. OMAR PARANAÍBA VILELA NETO - Coorientador
Departamento de Ciência da Computação - UFMG



PROF. ANTÔNIO OTÁVIO FERNANDES
Departamento de Ciência da Computação - UFMG



PROF. RICARDO AUGUSTO RABELO OLIVEIRA
Departamento de Computação - UFOP



PROF. SÉRGIO VALE AGUIAR CAMPOS
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 24 de fevereiro de 2015.

A Rui e Toninha

Acknowledgments

I apologize to English readers, but I must express my gratitude in Portuguese.

Agradeço primeiramente a Deus, pelo dom da vida e os caminhos aos quais me conduz. "Senhor que a Tua vontade leve a minha de cabresto, pra todo sempre até a querência do céu" (Dom Luis Felipe de Nadal). Aos meus pais Rui e Toninha e à minha irmã Lella que, como sempre, superaram-se nas demonstrações de amor e carinho, me dando forças para seguir com meus objetivos. Ao meu *anjo* Marina que alegra meus dias e me faz tão feliz.

Ao meu orientador e amigo Leo, por tamanha generosidade na condução desse trabalho e, principalmente, pelos conselhos valiosos. Ao meu co-orientador Omar, pelas dúvidas sanadas, textos revisados e provocações futebolísticas. Àqueles que ajudaram diretamente na execução deste trabalho: Leandro e professores Fernando e Wong.

À minha família que amo de todo o coração. De maneira especial à minha prima Tali, serei eternamente grato a tudo o que fez por mim. À família Lima/Acorinti que me acolheu como um filho. Aos amigos de Passos, BH e Campinas, pela grande parceria.

Aos amigos do wisemap e a todos os funcionários do DCC/UFMG, principalmente as secretárias do departamento que souberam responder com gentileza às minhas inúmeras requisições e dúvidas.

Às agências de fomento à pesquisa brasileiras, CNPq e CAPES, e à Intel pelo apoio financeiro.

Enfim, agradeço a todos que me ajudaram a concluir esse trabalho.

Resumo

Estouro de Arranjos (*Buffer Overflow* – BOF) continua a ser uma das principais vulnerabilidades encontradas em software. Ano passado, a comunidade de Segurança da Informação foi surpreendida quando pesquisadores revelaram uma vulnerabilidade de BOF no OpenSSL. Linguagens de programação como C e C++, amplamente usadas para desenvolvimento de sistemas e em uma grande variedade de aplicações, não proveem Verificação de Limites de Arranjos (*Array Bound Checks* – ABCs) nativamente. Existem inúmeras propostas que visam a proteção de memória para essas linguagens, através de soluções baseadas em software ou hardware. Ainda assim, entretanto, essas técnicas acabam por comprometer o desempenho dos programas, o que não é uma solução ideal para o problema. Este trabalho apresenta uma nova abordagem para alcançar verificação de limites de arranjos e acesso à memória (quando permitido) através de uma única instrução. Nós discutimos como ela pode ser implementada em arquiteturas com tamanho variável de instruções e disponibilizamos uma implementação de referência.

Abstract

A Buffer Overflow (BOF) continues to be among the top causes of software vulnerabilities. Last year the security world was taken by surprise when researches unveiled a BOF in OpenSSL. Languages like C and C++, widely used for system's development and for a large variety of applications, do not provide native Array-Bound Checks (ABC). A myriad of proposals endeavor memory protection for such languages by employing both software- and hardware-based solutions. Due to numerous reasons, none of them have yet reached the mainstream. In this work, we propose a novel approach to achieve an array's bound-check and a memory access (when allowed) within a single instruction. We discuss how it can be implemented on variable-length ISAs and provide a reference implementation.

List of Figures

2.1	Push parameter and call function <code>foo</code>	7
2.2	Beginning of function <code>foo</code>	8
2.3	Calling function <code>fread</code>	9
2.4	Execution of function <code>fread</code> - Buffer Overflow.	10
2.5	Returning from the function <code>fread</code> - Execution flow changed.	11
2.6	Heartbleed - A BOF vulnerability found in OpenSSL.	12
2.7	Fetch stage in the Y86 pipeline.	14
2.8	Decode stage in the Y86 pipeline.	15
2.9	Execute stage in the Y86 pipeline.	15
2.10	Memory stage in the Y86 pipeline.	16
2.11	WriteBack stage in the Y86 pipeline.	16
2.12	Y86 pipeline.	17
2.13	Complete MPX-based pipeline and associated components with our additions in red.	20
4.1	Fetch stage in the modified Y86 pipeline.	26
4.2	Decode stage in the modified Y86 pipeline.	27
4.3	Execute stage in the modified Y86 pipeline.	28
4.4	Memory stage in the modified Y86 pipeline.	28
4.5	Encoding and meaning of each field of a SMOV instruction.	29
4.6	Complete pipeline and associated components with our additions in red.	30
5.1	Runtime overhead caused by different security approaches.	42
5.2	Speed up of the two hardware strategies on top of the software-based approach.	43
5.3	Logical elements overhead to implement the two hardware strategies on top of the original Y86 project.	45

List of Tables

1.1	Unprotected (left) and protected (right) versions of a program in C.	2
2.1	Unprotected (left) and protected (right) versions of a program in assembly. . . .	13
2.2	<code>rmmovl</code> , <code>mrmovl</code> and <code>subl</code> instructions pipeline stages.	18
3.1	Comparison of hardware-related approaches that protect against BOFs.	24
4.1	Processing required for <code>srmovl</code> and <code>srmovl</code> instructions.	29
5.1	Identification of global array accesses (left) and their replacement to use <code>SMOV</code> (right).	40
5.2	Evaluation result of different ways to guard memory accesses.	41
5.3	Comparison of different implementations of the Y86 processor.	44

Contents

Acknowledgments	xi
Resumo	xiii
Abstract	xv
List of Figures	xvii
List of Tables	xix
1 Introduction	1
1.1 Contribution	3
1.2 Publications and Submissions	3
1.3 Organization	4
2 Background	5
2.1 Buffer Overflow	5
2.2 ABCs and its computational cost	12
2.3 Y86 Architecture	13
2.4 Memory Protection Extension - MPX	18
3 Related Work	21
3.1 Static and Dynamic memory protection techniques	21
3.2 Hardware-based memory protection techniques	22
4 SMOV	25
4.1 Architecture Choice	25
4.1.1 Design	26
4.1.2 Programming Interface and Capabilities	30
4.2 Implementation Efforts	31

5 Evaluation	37
5.1 Methodology	37
5.2 Results	41
5.3 Hardware Cost	44
6 Conclusion and Future Work	47
Bibliography	49

Chapter 1

Introduction

A Buffer Overflow (BOF) takes place whenever a system allows data to be accessed out of the bounds of an array [Cowan et al., 2000; Wagner et al., 2000; Lhee and Chapin, 2003; Bishop et al., 2012]. An adversary can leverage that to overwrite memory space that guides program's execution flow, divert it towards a malicious code, and thus take system control. BOFs are considered one of the most challenging sources of vulnerabilities in computing systems [Cowan et al., 2000].

The Morris worm [Moore et al., 2002] is a good case in point of how devastating BOF attacks might be. Back in 1988, the worm made use of the then-novel technique of buffer over-write, a sort of BOF, and compromised approximately 10% of the computers connected to the Internet.

Earlier in 2014, Internet users were taken by surprise when the security community found a new BOF vulnerability named Heartbleed¹, which allowed attackers to over-read an array. Half a million web servers were affected by it. The worsening factor was because the hole was found in OpenSSL, a widely used security library. Heartbleed is deemed by some as the worst security flaw that has been ever discovered on the Internet.

BOF attacks are still frequent because languages like C and C++, commonly used for system programming, do not prevent out-of-bounds memory accesses [Haugh and Bishop, 2003; Nagarakatte et al., 2010]. Those languages are inherently unsafe, since their semantics legitimately allow this sort of "illegal" memory access: an *array*[*i*] access is considered safe if (a) the variable *i* is greater than or equal to zero; and (b) the variable *i* is less than the defined length of *array*.

Operating systems and compilers writers have developed over time a series of defense mechanisms to protect against a few memory-violation issues. The most notable ones are Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) - also

¹<http://heartbleed.com/>

Table 1.1: Unprotected (left) and protected (right) versions of a program in C.

<pre> #define BUFSIZE 512 int main() { int buffer[BUFSIZE]; int a, i, j; ... for(i; i < j; i++) { buffer[i] = a; } } </pre>	<pre> #define BUFSIZE 512 int main() { int buffer[BUFSIZE]; int a, i, j; ... for(i; i < j; i++) { if(i >= 0 && i < BUFSIZE) buffer[i] = a; } } </pre>
---	---

known as the No-eXecute bit (NX). That eventually led to more advanced attacks such as the Return-to-libC attack [Tran et al., 2011] and its variations like Return Oriented Programming [Buchanan et al., 2008]. Current defenses against those are not totally efficient [Carlini and Wagner, 2014].

A myriad of proposals endeavor to mitigate BOF vulnerabilities by resorting to the so-called Array-Bound Checks (ABCs), which are tests performed at runtime to ensure that a particular array access is safe. An ABC check is demonstrated on the right side of Table 1.1.

ABCs can either be implemented in software, by instrumenting code with assertion statements, or in hardware, via a combination of multiple general-purpose instructions. However, both approaches tend to degrade program’s performance and, consequently, do not provide a satisfying solution against BOFs.

Software-based approaches usually consist of two passes. Initially, a program’s assembly is scanned to find code snippets containing potential vulnerabilities. Afterwards, in a second pass, ABCs are inserted to the select places. While effective in preventing BOFs from happening, such approaches typically slow down the resulting program by a significant amount of time. For instance, AddressSanitizer [Serebryany et al., 2012], a popular tool maintained by Google, is known to cause 70% time and 200% memory consumption overhead.

Hardware-based approaches (e.g., Devietti et al. [2008]; Nagarakatte et al. [2012, 2014]; Intel Corporation [2013]) offer new specific machine instructions for bound-checking purposes. They differ from each other in a variety of factors: the overall format and con-

tent of an instruction, the number of instructions necessary to complete a safe array access, the number of cycles a particular check takes, whether both the upper and lower limits are checked at once, and the required supporting hardware components.

Evaluation of hardware-based ABC solutions can be guided by distinct fronts: energy consumption, hardware size, hardware design and implementation complexity, Instruction Set Architecture (ISA) and compiler related friendliness. Ultimately, performance of a bound-checking enabled program is evaluated. Nevertheless, results are usually difficult to reproduce and subjected to characteristics and limitations of a particular simulation/emulation environment [Binkert et al., 2011; Bellard, 2005; Binkert et al., 2006; Yourst, 2007] or tightly coupled with a particular hardware architecture implementation.

1.1 Contribution

In this work, we present SMOV, a hardware-based solution for BOFs that consists of a pair of secure load and secure store instructions. The novelty in our instructions is the fact they perform an ABC and a memory access, when allowed, altogether. Our key observation is that a subtraction is an operation fast enough to be computed in sequence to an addition of the Arithmetic Logical Unit (ALU) operation without disturbing the normal pipeline.

We present our solution from a practical but still general perspective, accompanied by the formalism necessary to hardware design. Although our instructions have a x86-like format, they can easily be extended to any architecture that allows variable-length instructions. Given the subtleness involved in hardware performance evaluation, along with details that are highly simulation/emulation environment dependent, we prefer to focus our discussion on the overall modeling of SMOV and to describe how it can be implemented, since this is an important aspect not covered in previous work. On the other hand, in order to keep our ideas tangible, we offer a synthesizable Verilog for an academical processor which can be used for real evaluations.

1.2 Publications and Submissions

Below there is a list of publications related to this work:

- MAIA NETO, A. L. ; MELO, L. T. C. ; O. P. VILELA NETO, O. P. ; PEREIRA, F. M. Q. ; OLIVEIRA, L. B. Protecting Programs Against Memory Violation In Hardware. In: IEEE Latin America Transactions, 2015.

- MAIA NETO, A. L. ; PARAIBA, O ; PEREIRA, F. ; OLIVEIRA, L. B. . S-MOVL: Protegendo Sistemas Computacionais contra Ataques de Violação de Memória por meio de Instruções em Hardware. In: Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg' 14), 2014, Belo Horizonte.
- TORRES, E. ; MAIA NETO, A. L. ; PARAIBA, O ; OLIVEIRA, L. B. . Implementação em Hardware de Instrução Segura de Acesso à Memória - Caso MIPS 16 bit. In: Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg' 14), 2014, Belo Horizonte.

Our work was also submitted to:

- IEEE/IFIP International Conference on Dependable System and Networks (DSN' 15);
- IEEE Global Communications Conference, Exhibition & Industry Forum (GLOBE-COM' 15).

1.3 Organization

The remainder of this work is structured as follows. In Chapter 2 we present definitions and background concepts that are key for understanding our work. Chapter 3 discusses related work. Chapter 4 presents our solution. Chapter 5 evaluates SMOV in terms of runtime and hardware overhead. We conclude and comment about future work in Chapter 6.

Chapter 2

Background

In this chapter, we briefly present important background information for the understanding of our work. We begin with a description of BOF vulnerability. Section 2.2 explains what ABCs are and their cost when implemented in software. In Section 2.3, we present the Y86 architecture, then Section 2.4 analyses a hardware-based strategy to protect against BOF.

2.1 Buffer Overflow

A BOF occurs when we want to access data inside an array but end up accessing memory out of its bounds. This vulnerability is still possible in some programming languages that do not protect out-of-bounds memory accesses, such as C and C++. It can appear in different ways, for instance:

- direct memory manipulation without checking the access index: `array[i]` – where `i` is less than 0 or greater than or equal to the size of `array`;
- array copy without checking the size of the destination array: `memcpy(dest, src, num)` – where the size of the destination array `dest` is less than `num`;
- file reading without checking the size of the destination array: `fread(buffer, sizeof(char), 517, file)` – where the size of the array `buffer` is less than 517 bytes.

An adversary can use a buffer overflow to overwrite memory space that guides program's execution flow, divert it towards a malicious code, and thus take system control. As an example, consider the Code 2.1. In the function `foo`, the array `buffer` receives 517 bytes read from the file `badfile`, however the array's size is 12 bytes.

```
#include <stdio.h>
void foo(FILE *badfile) {
    char buffer[12];
    ...
    fread(buffer, sizeof(char), 517, badfile);
    ...
    return 1;
}
int main() {
    FILE *badfile;
    badfile = fopen("file", "r");
    foo(badfile);
    fclose(badfile);
    return 1;
}
```

Code 2.1: C program with a highlighted buffer overflow vulnerability.

Figures 2.1 to 2.5 describe step-by-step how the program's execution flow can be changed due to this BOF vulnerability. We are going to see the result of each instruction in the stack during the program execution. Each figure show on the left side the current instruction of the Code 2.1 in its assembly instructions representation and, on the right side, the stack's current layout:

1. in the first step we consider that the file `badfile` was already open and its file descriptor is in register `%eax`. This value is push onto the stack by the instruction `movl %eax, (%esp)` to serve as parameter to the function `foo`. The function is called by the instruction `call foo`. It pushes the return address onto the stack and changes the execution flow to `foo`'s address;

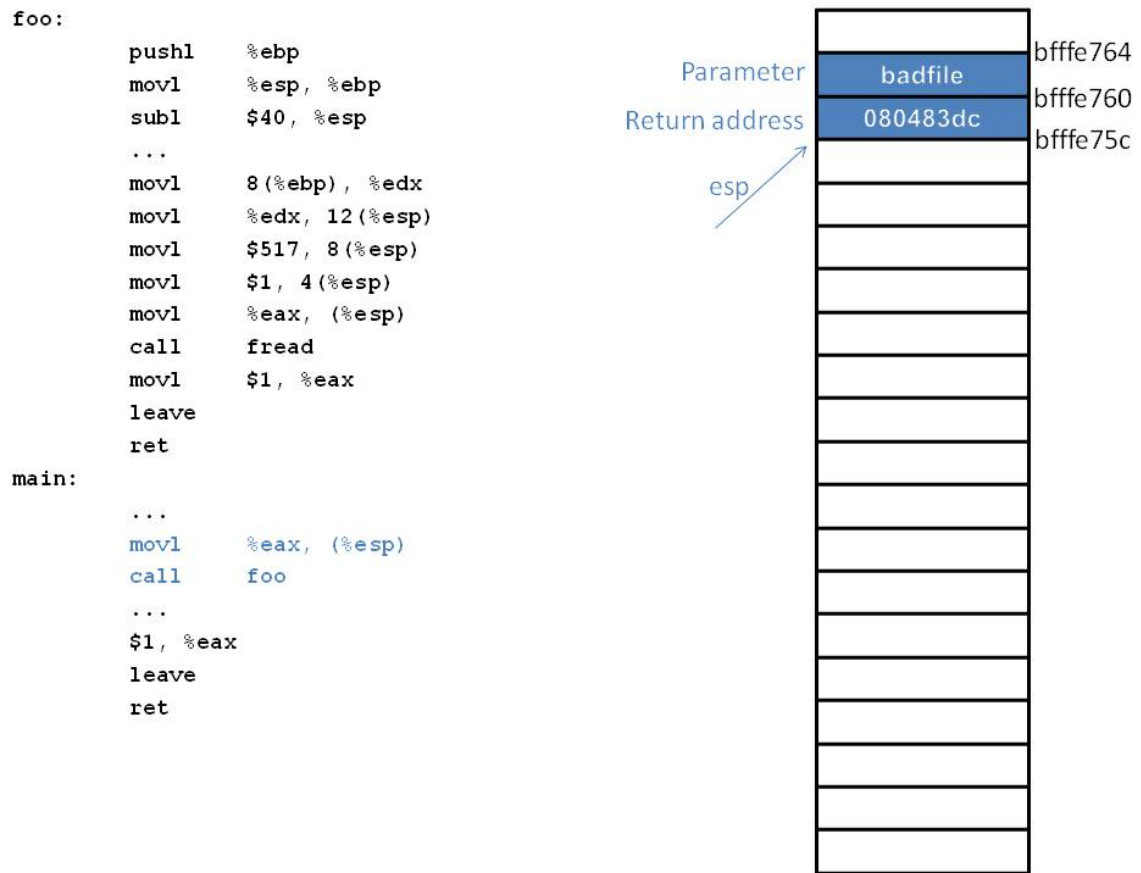
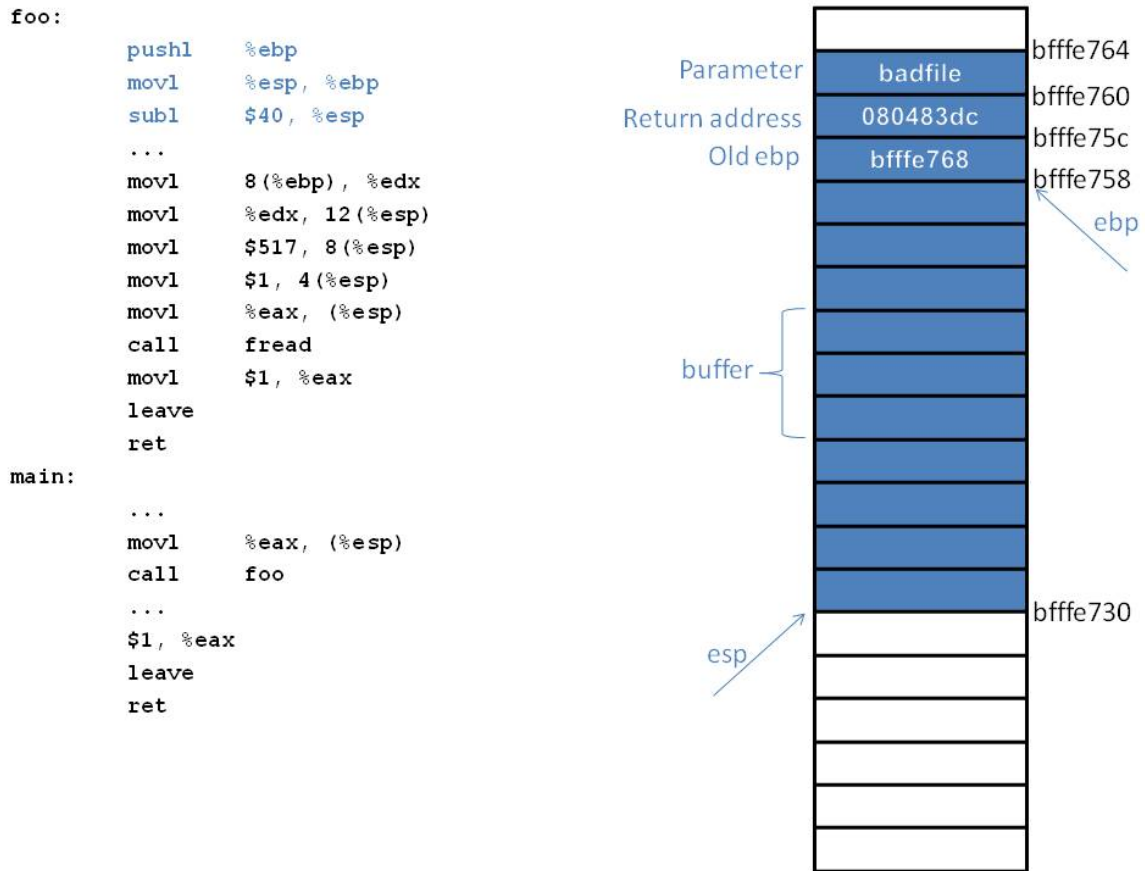


Figure 2.1: Push parameter and call funtion `foo`.

- function `foo` begins by saving the current base pointer (called old in the figure), setting a new one and reserving stack space for the current function. This reserved space includes the 12 bytes for the local array `buffer`;

Figure 2.2: Beginning of function `foo`.

- the following instructions push the parameters of the function `fread` onto the stack and call the function;

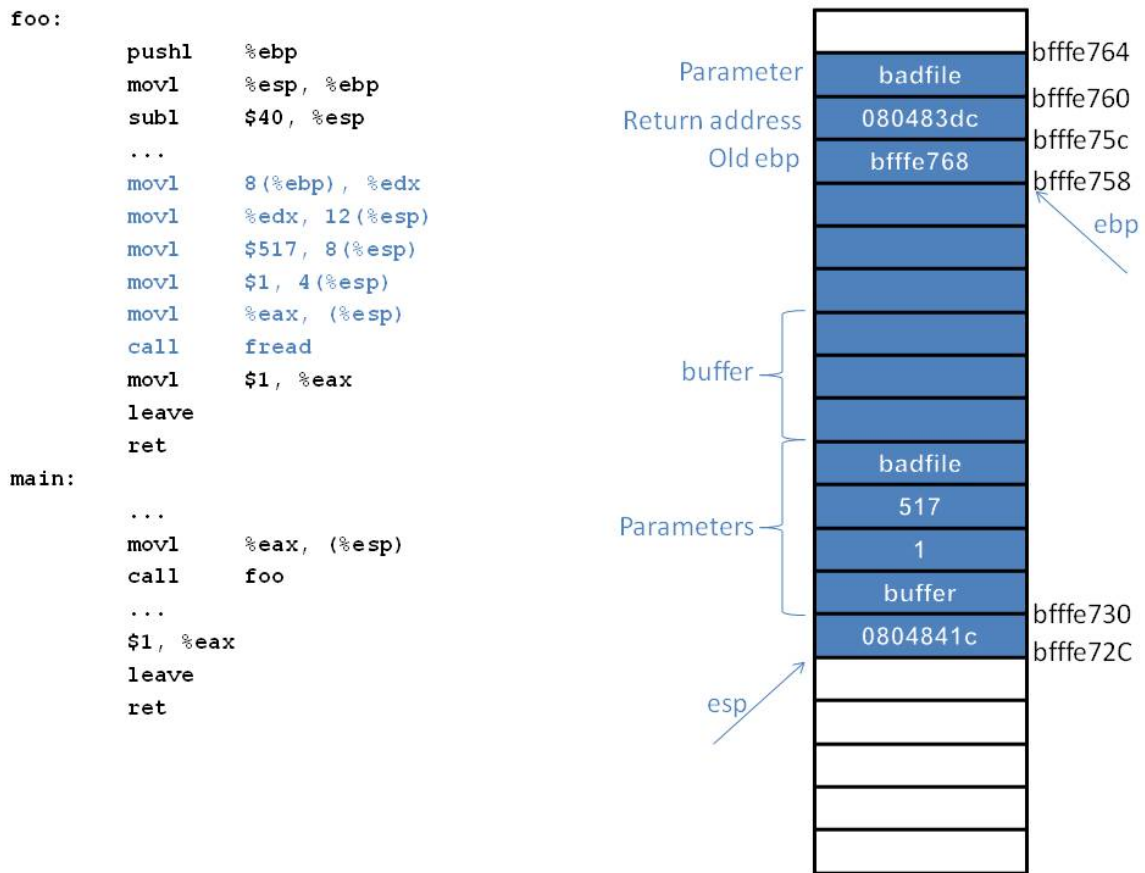
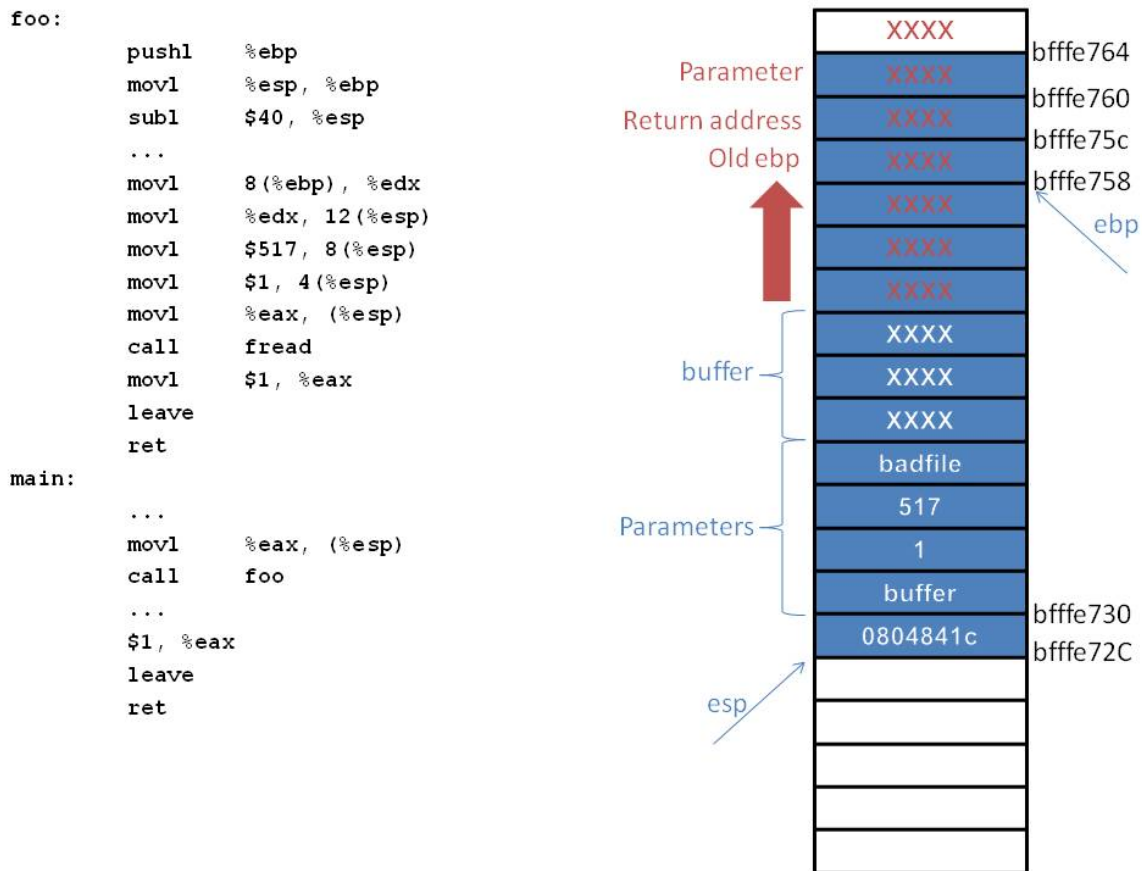


Figure 2.3: Calling function fread.

- the function `fread` fills the `buffer`'s 12 bytes, however, it was required to read 517 bytes, so the portion of memory located over the `buffer` is also filled causing a BOF. Notice that the return address described in step 2 is changed;

Figure 2.4: Execution of function `fread` - Buffer Overflow.

- when `foo` terminates, it must change the execution flow to the return address saved onto the stack. However, this value was changed by `fread`, therefore the execution flow will be deviated to somewhere else than it was supposed to be.

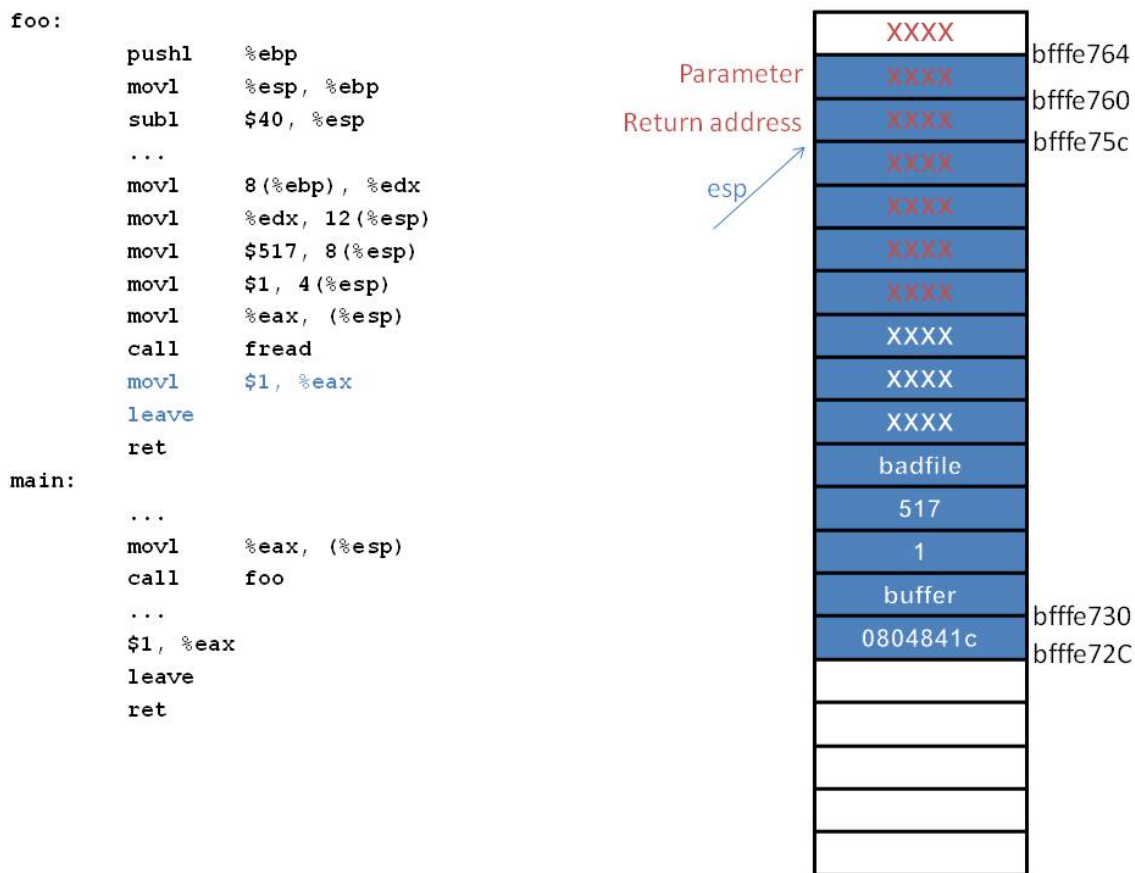


Figure 2.5: Returning from the function fread - Execution flow changed.

Another type of BOF is related to reading adjacent memory of an array. This is the kind of problem that have been found in OpenSSL, a library written in C that is used to protect computing systems and their communications. The problem's behavior is shown in Figure 2.6. OpenSSL has a feature – called Heartbeat – where the client can ask the server to respond a specific challenge. However, the client could misinform the challenge's size and get a response sized greater than the challenge, revealing sensitive information. This problem, called HeartBleed, compromised 17% of SSL web servers which used certificates issued by trusted certificate authorities, accounting for around half a million certificates. Heartbleed is deemed by some as the worst security flaw that has been ever discovered on the Internet.

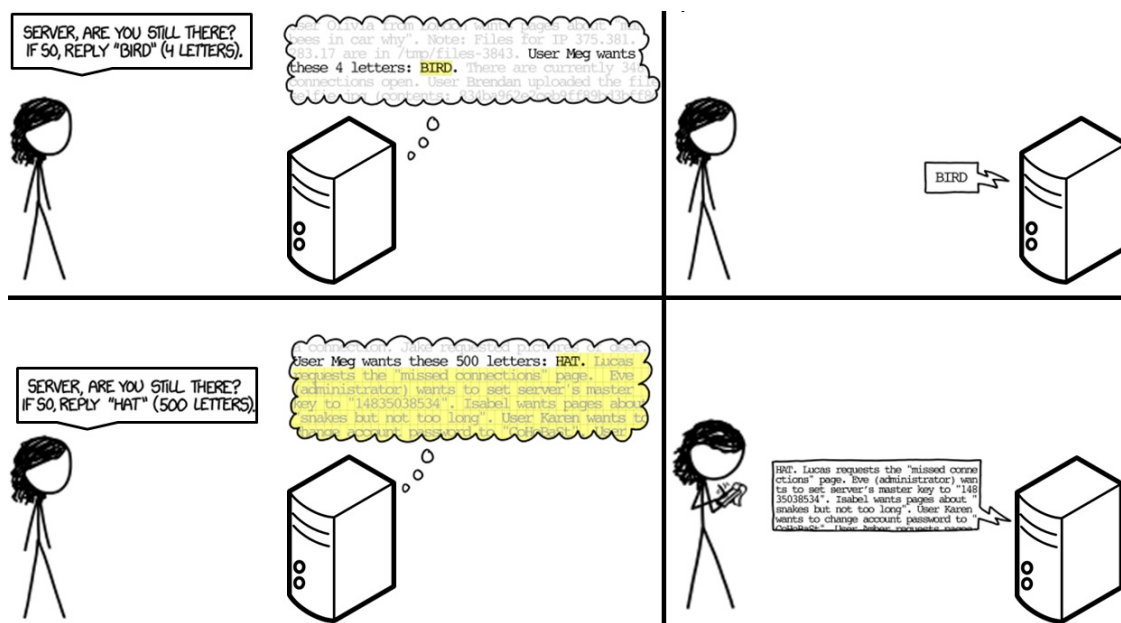


Figure 2.6: Heartbleed - A BOF vulnerability found in OpenSSL.

2.2 ABCs and its computational cost

A simple way to protect a C program against a BOF is testing the particular index variable against the upper and lower bounds of the allocated memory that corresponds to the array. For example, Table 1.1 shows an unprotected (to the left) and a protected (to the right) versions of the same program. In the latter, there is a bound limit verification of the index i before the assignment `buffer[i] = a`. Thus, if someone tries to execute the program with $i < 0$ or $i \geq \text{BUFSIZE}$ in the unprotected version, a BOF occurs. Whereas in the protected version the assignment is prevented.

Table 2.1 shows the same programs in their Y86 assembly code. Eight additional instructions are required to check the bounds of the array (four instructions for each bound), as highlighted in the right side of the table. The instruction `mrmovl` loads the index i in a register; instruction `irmovl` loads the bound (lower or higher) in another register; instruction `subl` subtracts the values; finally, instructions `js` or `jg` are conditional branches that, depending on the result of the subtraction, deviate the program flow.

As it can be observed, bound-checking comes at a high cost. This becomes too expensive in complex applications that require a large number of verifications. Indeed, there are works showing that the application of ABCs implemented in software can cause an overhead up to 70% in execution time [Serebryany et al., 2012].

<pre> 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 </pre>	<pre> L4: mrmovl -12(%ebp), %edi irmovl \$0, %ebx subl %ebx, %edi js L3 mrmovl -12(%ebp), %edi irmovl \$511, %ebx subl %ebx, %edi jg L3 </pre> <div style="display: flex; justify-content: flex-end; align-items: center; margin-left: 10px;"> <div style="font-size: 3em; margin-right: 5px;">}</div> <div style="font-size: 0.8em; margin-right: 5px;">if((i>=0))</div> </div> <div style="display: flex; justify-content: flex-end; align-items: center; margin-left: 10px; margin-top: 10px;"> <div style="font-size: 3em; margin-right: 5px;">}</div> <div style="font-size: 0.8em; margin-right: 5px;">&&(i<BUFSIZE)</div> </div> <div style="display: flex; justify-content: flex-end; align-items: center; margin-left: 10px; margin-top: 10px;"> <div style="font-size: 3em; margin-right: 5px;">}</div> <div style="font-size: 0.8em; margin-right: 5px;">buffer[i] = a;</div> </div> <pre> L3: mrmovl -12(%ebp), %eax mrmovl -4(%ebp), %edx rrmovl %eax, %edi sall \$2, %edi addl %ebp, %edi rmmovl %edx, -2060(%edi) L2: mrmovl -12(%ebp), %edi iaddl \$1, %edi rmmovl %edi, -12(%ebp) L1: mrmovl -12(%ebp), %eax mrmovl -8(%ebp), %edi subl %edi, %eax jl L4 irmovl \$0, %eax leave </pre>
--	--

Table 2.1: Unprotected (left) and protected (right) versions of a program in assembly.

2.3 Y86 Architecture

Y86 is a 32-bit ISA inspired by x86/IA32. The Y86 ISA has fewer data types, instructions, and addressing modes when compared to x86. Still, it is sufficiently complete to allow us to write and execute complex programs in a five-step pipelined processor. Within the scope of this work, the most relevant instructions are those related to data manipulation in memory, and data comparison, as summarized below:

- `mrmovl`: memory read (*memory* → *register*);
- `rmmovl`: memory write (*register* → *memory*);
- `subl`: subtract two operands for comparison.

Next, we present a description of what is done in each one of the five stages of the pipelined Y86 processor when the aforementioned instructions are executed:

1. *Fetch*: the Program Counter (PC) is used as the memory address and the fetch stage reads the bytes of an instruction from memory. The instructions bytes are sorted out and stored in special pipeline registers so they can be used in the upcoming stages. The value $valP$, computed as the current PC plus the length of the fetched instruction, will be the address of the next instruction - in the case of jumps or function returns it must be adjusted, but we will not enter the details here. The fields of the instruction along with their meaning are listed below:

- *icode* and *ifun*: instruction opcode, and function that must be executed in the execute stage, respectively;
- *rA* and *rB*: possible register specifier byte;
- *valC*: possible constant word whose meaning depends on the instruction (it is an offset in case of a memory access instruction).

In Figure 2.7 we have a visual representation of each field of an instructions in the Fetch stage:

		icode	ifun	valC	rA	rB
Fetch	rmmovl rA, valC(rB)	rmmovl	+	valC	rA	rB
	mrmovl valC(rB), rA	mrmovl	+	valC	rA	rB
	subl rA, rB	subl	-	0	rA	rB

Figure 2.7: Fetch stage in the Y86 pipeline.

2. *Decode*: the operands used in the instruction (at most 2 – *rA* and *rB*) are retrieved from the register file in the form of $valA$ and $valB$. The constant $valC$ is passed through the stages, as can be seen in Figure 2.8:

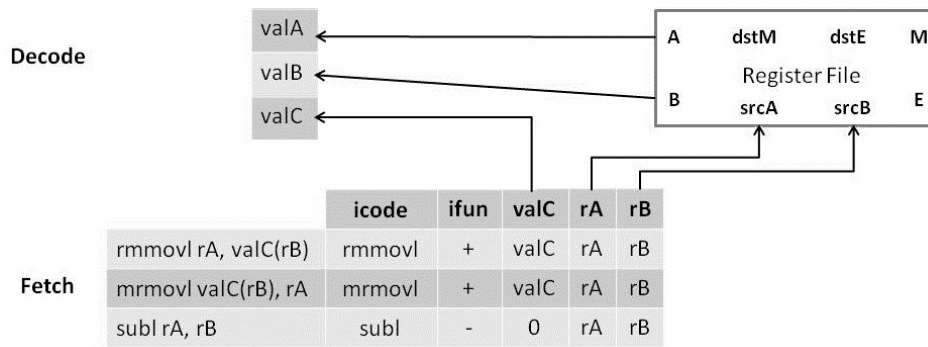


Figure 2.8: Decode stage in the Y86 pipeline.

3. *Execute*: the Arithmetic Logic Unit (ALU) performs the operation specified by the instruction (*ifun*). It sums the constant *valC* to the content *valB* of the base register *rB*. This result forms the effective memory address in case of a memory access instruction. In the case of a subtraction, the ALU subtracts the value *valA* from the value *valB*, and the condition codes used for comparisons are set. The result of the subtraction is named as *valE*. Figure 2.9 presents this behaviour;

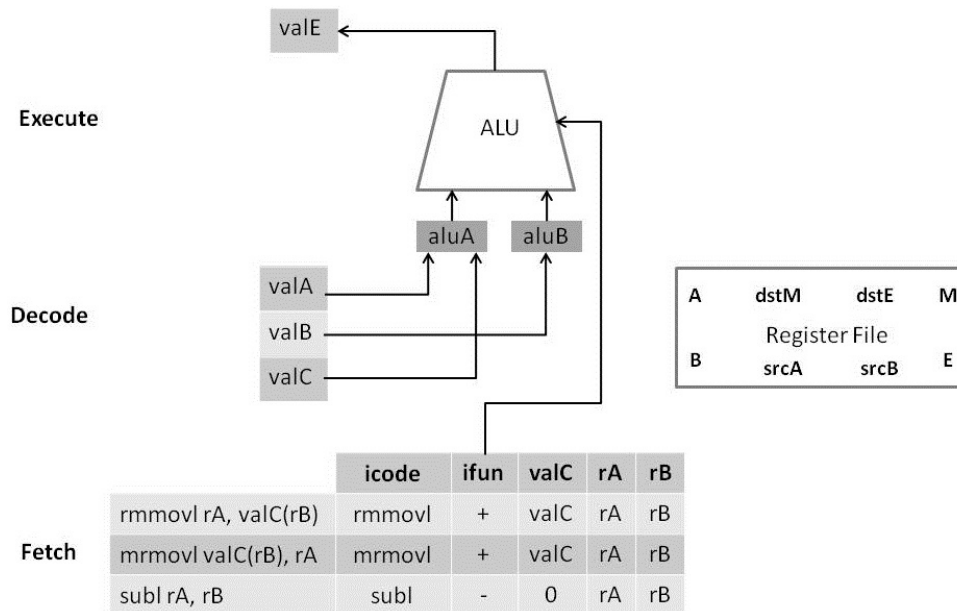


Figure 2.9: Execute stage in the Y86 pipeline.

4. *Memory*: the effective address calculated in the previous stage is used to access the memory. The memory control determines if the operation is a load or a store.

Data is read from the memory address to $valM$ in case of a load operation, or the value $valA$ is written to the memory address in case of a store, as shown in Firuge 2.10;

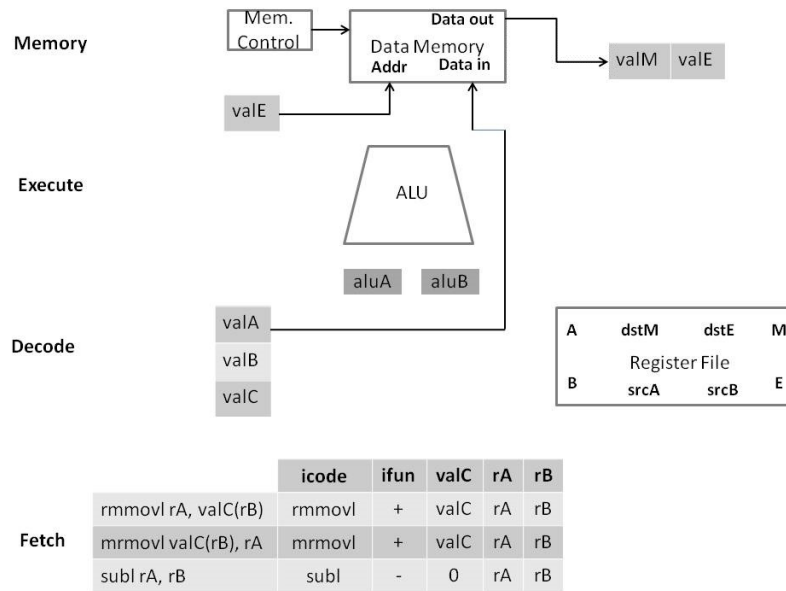


Figure 2.10: Memory stage in the Y86 pipeline.

5. *Write back*: this stage writes the ALU's output ($valE$) or data read from memory ($valM$) to the register file.

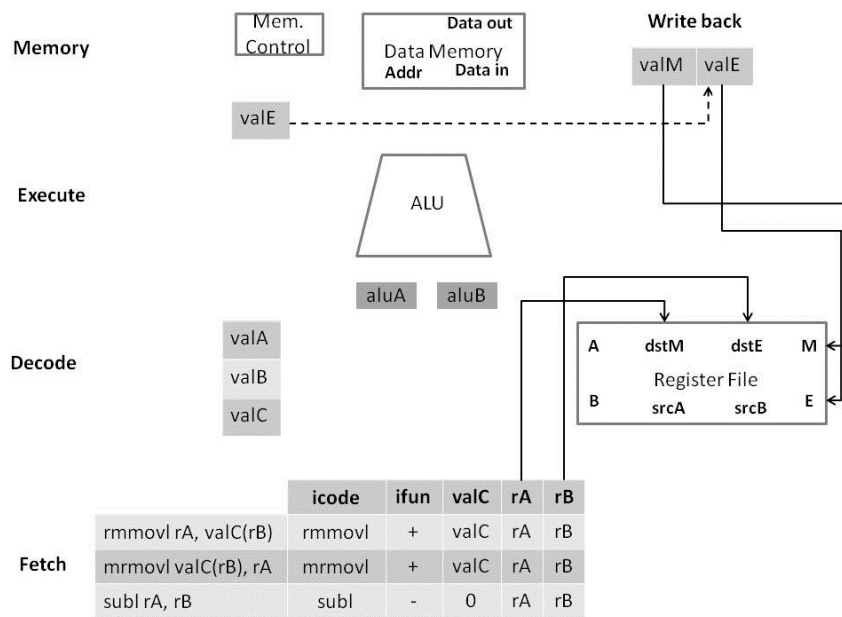


Figure 2.11: WriteBack stage in the Y86 pipeline.

Figure 2.12 shows the full pipeline of Y86 architecture.

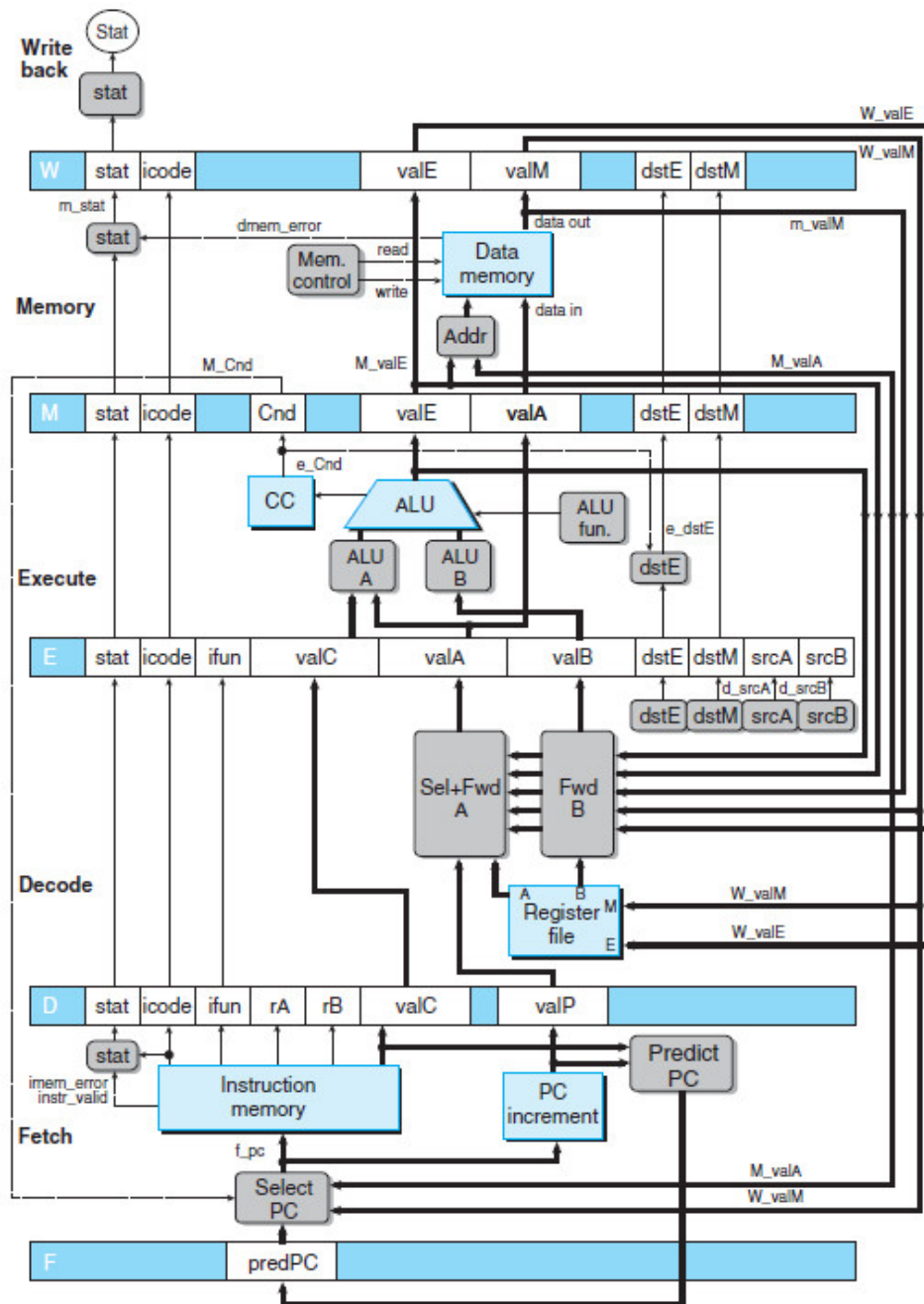


Figure 2.12: Y86 pipeline.

Table 2.2 summarizes how the three previously mentioned instructions proceed through the stages of the pipeline. These should be read from top to bottom, as they are evaluated in a sequence. Each line in the table describes an assignment (represented by \leftarrow) to some signal or stored data. For instance, $x \leftarrow M_n[Y]$ and $M_n[Y] \leftarrow x$ indicates, respectively, a read from memory and a write of n bytes to memory. If the source or destination of

an operation is a register, $R[y]$ is used instead of $M_n[Y]$. Lastly, the notation $x : y \leftarrow M_1[Y]$ means that the components of a byte are split between two variables: x receives the least significant 4 bits of $M_1[Y]$ while y receives the most significant 4 bits.

Stage	rmmovl rA, valC(rB)	mrmovl valC(rB), rA	subl rA, rB
Fetch	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC + 1]$ $valC \leftarrow M_4[PC + 2]$ $valP \leftarrow PC + 6$ $PC \leftarrow valP$	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC + 1]$ $valC \leftarrow M_4[PC + 2]$ $valP \leftarrow PC + 6$ $PC \leftarrow valP$	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC + 1]$ $valP \leftarrow PC + 2$ $PC \leftarrow valP$
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	$valB \leftarrow R[rB]$	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$
Execute	$valE \leftarrow valB + valC$	$valE \leftarrow valB + valC$	$valE \leftarrow valB - valA$ $SetCC$
Memory	$M_4[valE] \leftarrow valA$	$valM \leftarrow M_4[valE]$	
Write back		$R[rA] \leftarrow valM$	$R[rB] \leftarrow valE$

Table 2.2: `rmmovl`, `mrmovl` and `subl` instructions pipeline stages.

2.4 Memory Protection Extension - MPX

This section describes the operation of an approach similar to ours, the Memory Protection Extension (MPX). MPX is a set of processor features documented by Intel on July 2013 [Intel Corporation, 2013] that associates bounds with pointers, and checks, via hardware, if the pointer based accesses are suitably constrained. These operations are performed by using a new set of instructions added to the processor ISA. The general flow of checking a memory access using this approach is described below.

1. Create the pointer bounds information: the first task is to load in special registers the pointer's bounds. It is done through the new instruction `bndmk`. The instruction `bndmk %edi, %ebx, %bnd0` sets the lower 32 bits of the register `bnd0` with the content of register `%edi`, and the higher 32 bits of `bnd0` with the sum of the contents of registers `%edi` and `%ebx`. It means that `%edi` should keep the lower bound information and `%ebx` the size of the array.
2. Check the bounds: there are two instructions to perform the array bounds check before a memory access. To check the lower bound is used the instruction `bndcl %eax, %bnd0`, that verifies if the content of register `%eax` is lower than the content of the lower 32 bits of the register `bnd0`, if so an interruption is raised. Similarly, the upper bound is verified by the instruction `bndcu`. The statement `bndcu`

`%eax, %bnd0` checks if the content of `%eax` is greater than the higher 32 bits of special register `bnd0`, if so an interruption is raised.

3. Conclude memory access: if MPX does not raise any interruption during the bounds check, the memory access can be concluded. This task is done through the regular memory access instructions `load` and `store`.

Figure 2.13 presents a MPX-based project over the original Y86 pipeline, containing the basic operation of MPX (instructions `bndmk`, `bndcl` and `bndcu`).

As we could see in the previous definition, MPX adds special registers to keep the pointers' bounds. In fact, there are four new special registers to keep the bounds. Even that it is not in the scope of this work, it is worth mention that to deal with more than four pointers, MPX provides instructions that read or write this kind of information to/from memory. It has an internal organization in the form of directories.

Another interesting feature found in MPX is backwards compatibility. The MPX instructions when executed in legacy Intel hardware are interpreted as NOP (No Operation) instructions. We speculate that is the main reason MPX does not optimize even more its new set of instructions, like performing the bound checks and memory access in a single instruction.

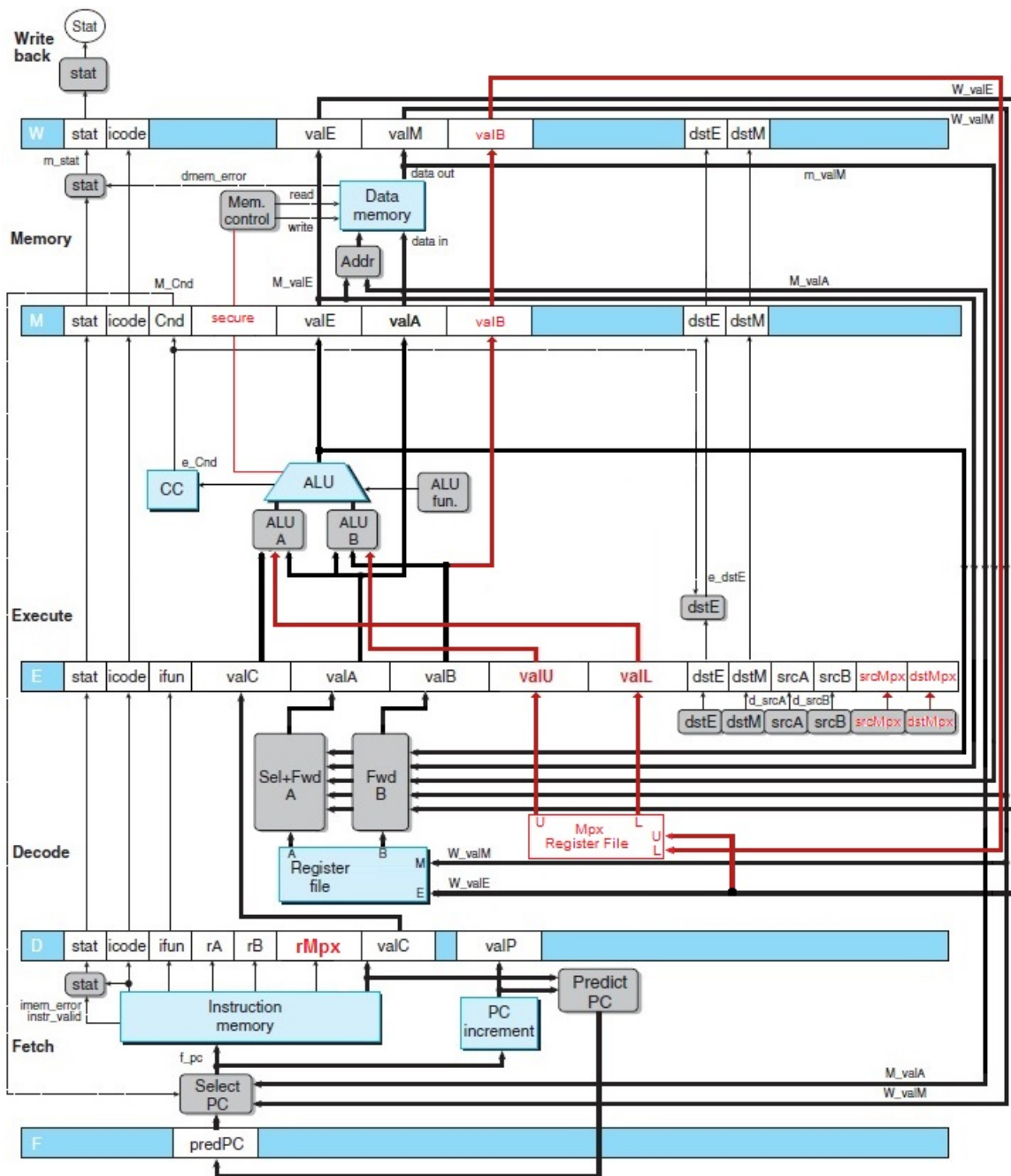


Figure 2.13: Complete MPX-based pipeline and associated components with our additions in red.

Chapter 3

Related Work

The literature contains a vast number of techniques to protect computational systems(e.g. Jones and Kelly [1997]; Haugh and Bishop [2003]; Wagner et al. [2000]; Xie et al. [2003]; Dhurjati et al. [2006]; Serebryany et al. [2012]; Misra [1987]; Wagner and Dean [2001]; Evans and Larochelle [2002]; Chess and West [2007]; Holzmann [2002]; Cousot et al. [2005]; Viega et al. [2000]; Cowan et al. [1998]; Bell [1999]; Wilander and Kamkar [2003]; Newsome and Song [2005]; Piromsopa and Enbody [2006]; Francillon et al. [2009]; McGregor et al. [2003]; Devietti et al. [2008]; Nagarakatte et al. [2014]; Intel Corporation [2013]). In this chapter, we shall focus on the most prominent ones and those that directly relate to our work, hardware-based protection against BOFs. Before diving into the hardware-related approaches, we shall mention the methods that can be implemented at the compiler/run-time level. To make our review more systematic, we shall divide these strategies into static and dynamic approaches.

3.1 Static and Dynamic memory protection techniques

Among the static techniques [Misra, 1987; Wagner and Dean, 2001; Evans and Larochelle, 2002; Chess and West, 2007; Holzmann, 2002; Cousot et al., 2005; Chess and West, 2007; Chipounov and Candea, 2011], there exist works that are specially tailored to identify memory-related vulnerabilities, such as those that we seek to prevent with our new instructions. For instance, Array Checker [Xie et al., 2003] implements a symbolic, flow-sensitive analysis to find out which memory accesses can fall out-of-bounds. This tool points back to programmers which functions are vulnerable. The programmers must fix the vulnerable code until no more warnings are reported. Similar behavior is implemented by ITS4, another

static analysis tool [Viega et al., 2000]. Static analyses, such as these two, can be used to replace ordinary load/store operations with our guarded instructions.

Differently from static analyses, dynamic analyses are implemented while the program runs. A common way to implement this kind of analysis is through the dynamic instrumentation of binary code [Cowan et al., 1998; Bell, 1999; Wilander and Kamkar, 2003; Newsome and Song, 2005; Ozdoganoglu et al., 2006; Piromsopa and Enbody, 2006; Dhurjati et al., 2006; Francillon et al., 2009; Serebryany et al., 2012]. For instance, Piromsopa *et al.* have designed and implemented a technique to protect the return address of functions. They maintain a shadow memory in such a way that every word in the original memory space is shadowed with a bit, which indicates if that word has a high or a low security level. Any attempt to move information from a higher to a lower level may awake a resident system that handles such exceptions. To implement their idea, the authors have modified data and control flow instructions used in an IA32 emulator. For instance, branches such as `return`, `call` and `jump`, check the shadow bit of the addresses where they are pointing. There are techniques, implemented at the hardware level, whose goal is to make dependable systems safer. For instance, Francillon *et al.* Francillon et al. [2009] have proposed new machinery to protect the return address of functions against illegal control flow changes. Their approach also uses a verification bit to prevent unwanted writes on the return address of functions. Similar to Piromsopa *et al.*'s, Francillon *et al.*'s also require modifications in the instruction set architecture. We have also modified instructions in a x86-like architecture, but our idea is very different: we do not create a shadow area to determine if memory is valid or invalid. Our instructions use compiler support to guard array accesses whose symbolic size we can infer statically.

3.2 Hardware-based memory protection techniques

As mentioned before, hardware-based approaches (e.g., Devietti et al. [2008]; Nagarakatte et al. [2012, 2014]; Intel Corporation [2013]) typically offer new machine instructions for specific bound-checking purposes. They differ from each other in a variety of factors. We analyze them over the following aspects: whether the checking method is explicit or implicit; whether both the upper and lower limits are checked together, whether the memory access can be combined into the check; the required hardware support; and backwards compatibility.

Scheduled to emerge in the market this year is the Intel's Memory Protection Extension [Intel Corporation, 2013] (MPX). MPX introduces a family of instruction to store, recover and verify bounds of arrays. The supporting hardware comes with four special bound

registers, machinery to index bound tables efficiently, and other integration features. MPX's programming interface to perform bound-checking is composed by two instructions: `bndcu` to check the upper bound, and `bndcl` to check the lower bound. A violation triggers an exception and the program is terminated. Otherwise, execution flow continues, and a standard `mov` instruction may be performed. We believe that one of reasons for such independent checks is due to backwards compatibility: if a program is running on unsupported hardware, those instructions are converted to `NOPs`. We imagine that a similar mechanism could be employed with the `SMOV` instructions, since we carefully designed them in a way that the data necessary for a non-secure `mov` instruction is embedded into the secure version. In addition, x86 has powerful addressing modes and variable-length encoding, which makes this idea feasible. Having that in mind, `SMOV` is the only work that allows an architecture to combine security (through bound-checking), memory access, and backwards compatibility altogether.

WatchdogLite [Nagarakatte et al., 2014] is another work, similar in principle to MPX, where the bound-checking method is explicit. However, instead of extending the hardware with special registers, it relies on general purpose registers for bound-checking. The programming interface adds a single instruction, `SChk`, that checks both lower and upper bounds of an array. WatchdogLite also requires a standard `mov` instruction to access memory. It does not address backwards compatibility aspects.

The other category of hardware-based ABC is through an implicit method. This is adopted by HardBound [Deviatti et al., 2008] and Watchdog [Nagarakatte et al., 2012]. Under this method, the program must inform, through special instructions (`setbounds` in HardBound and `setident` in Watchdog) the lower and upper portions of memory that represent arrays. With this information, the hardware takes care of checking the bounds before any access to this marked memory is performed. To support the implicit method, these proposals have to augment the pointer data type with metadata to keep track of the size of the allocated regions. This requires deep changes in a typical computer architecture, like the addition of a cache of meta data and a new, or augmented, register file, resulting in an expensive hardware. Once the bounds are implicitly checked by the hardware, one can say that the memory access is also performed. It is not totally clear whether this is a kind of two-phase microcode operation. These works do not mention backwards compatibility either.

Table 3.1 summarizes the features found on each discussed solution, including ours. The table also shows runtime information and hardware overhead assumptions. `SMOV` is not as fast as Hardbound, at the price of more flexibility and simpler hardware.

Table 3.1: Comparison of hardware-related approaches that protect against BOFs.

	MPX	WatchdogLite	HardBound	Watchdog	SMOV
checking method	explicit	explicit	implicit	implicit	explicit
register file changes	✓	no	✓	✓	no
bound check + access	no	no	no	no	✓
addressed backwards compatibility	✓	no	no	no	✓
runtime overhead	N/A	29%	9%	25%	18%
hardware overhead	N/A	N/A	N/A	N/A	3.6%

Chapter 4

SMOV

In this chapter, we present SMOV, a hardware-based solution against BOF attacks. To the best of our knowledge, this is the first work that demonstrates the idea and feasibility of performing an array bound-check and a memory access as part of a single instruction.

4.1 Architecture Choice

Hardware performance benchmarks are often difficult to reproduce due to hardware implementation variations. While cycle-accurate simulators and emulators [Yourst, 2007; Bellard, 2005] do exist, results become more subjective on how we extend the underlying platform to consider new hardware components and datapath changes. In-house simulators and tools have also been used for hardware-based ABC proposals [Nagarakatte et al., 2014], but those are even less accessible in the sense of validation and adoption. An interesting alternative would be open-source hardware [Parulkar et al., 2008], but unfortunately there does not seem to exist a full-blown x86 implementation available ¹.

The strategy we have taken to realize SMOV is to implement it on top of the publicly available Y86 research architecture [Bryant and David Richard, 2003]. The Y86 is a 32-bit ISA inspired by x86. It has fewer data types, instructions, and addressing modes, but it is complete enough to allow us to execute real programs in a five-step pipelined processor. The reader familiar with Intel’s x86 should find it easy to bridge the two ISAs. For those who need a thorough introduction, please refer to Bryant and David Richard [2003].

¹http://zet.aluzina.org/index.php/Zet_processor

4.1.1 Design

SMOV relies on two new instructions, one is a secure load and the other is a secure store. They are derived from the following standard move operations from Y86:

- `mrmovl`: Load (*memory* \rightarrow *register*);
- `rmmovl`: Store (*register* \rightarrow *memory*);

Our central idea is to embed two extra registers, one for the upper bound and another for the lower bound, in the standard move operations so they can be used for bound-checking. The memory access is only allowed if the particular address being indexed is within the allocated memory region for the array in question. Otherwise, program execution is terminated.

The impact of adding those registers to the instruction, performing the valid memory region computation, and eventually allowing the memory access, is observed under different perspectives. We individually analyze all of them.

1. **Instruction Fetch and Encoding:** we use two extra registers, `rU` and `rL`, to store the upper and lower bounds of an array, respectively. This means one byte larger than the largest standard Y86 instruction. As long as the architecture allows us to read this extra byte without an increased cycle, there should exist no collateral effect, since the Y86 uses variable-length encoding. When compared to the standard Y86 load and store instructions, `rmmovl rA, D(rB)` and `mrmovl D(rB), rA`, our corresponding secure versions look like `srmmovl rA, D(rB), rU, rL` and `smrmovl D(rB), rA, rU, rL`. Figure 4.1 shows each field of the instructions in the modified Y86 Fetch stage:

		icode	ifun	valC	rA	rB	rU	rL
Fetch	<code>srmmovl rA, valC(rB), rU, rL</code>	<code>srmmovl</code>	+	valC	rA	rB	rU	rL
	<code>smrmovl valC(rB), rA, rU, rL</code>	<code>smrmovl</code>	+	valC	rA	rB	rU	rL

Figure 4.1: Fetch stage in the modified Y86 pipeline.

2. **Register File:** in order to have the secure registers decoded together with the other two registers that are part of a standard move operation, the register file must support four simultaneous read ports. In the worst case, a convention could require that bound registers are always read in the second half of the clock cycle and impose a constraint that reading and writing to bound registers within a single cycle is illegal. Figure 4.2

presents the Decode stage of the modified Y86 pipeline, where the four read port register file can be seen:

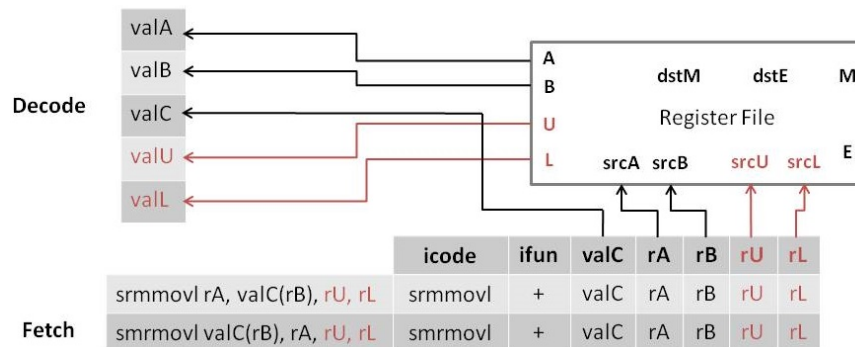


Figure 4.2: Decode stage in the modified Y86 pipeline.

- ALU and Bound-Checking:** bound-checking requires that two subtractions are made: $\mathcal{M}_I - \mathcal{M}_L$ and that $\mathcal{M}_U - \mathcal{M}_I$, or in Y86 notation, $rU - D(rB)$ and $D(rB) - rL$. At this moment, the ALU is already busy computing an addition of a register-indirect memory address. Therefore, we insert two specific-purpose subtractors to check the bounds. We also need a flagging mechanism that immediately points out whether the subtractions' result is zero, greater than zero, or negative - this can be achieved with a combination logic circuit. Addition and subtractions are relative fast operations and can normally be cascaded without any overhead. In particular, it is worth to recall that a Translation Lookaside Buffer (TLB) miss leads to a penalty of several cycles and main memory access is a well-known limiting factor of a pipeline. Figures 4.3 and 4.4 show the extra subtractors, and the logic circuit added to the Execute and Memory stages of the Y86 pipeline:

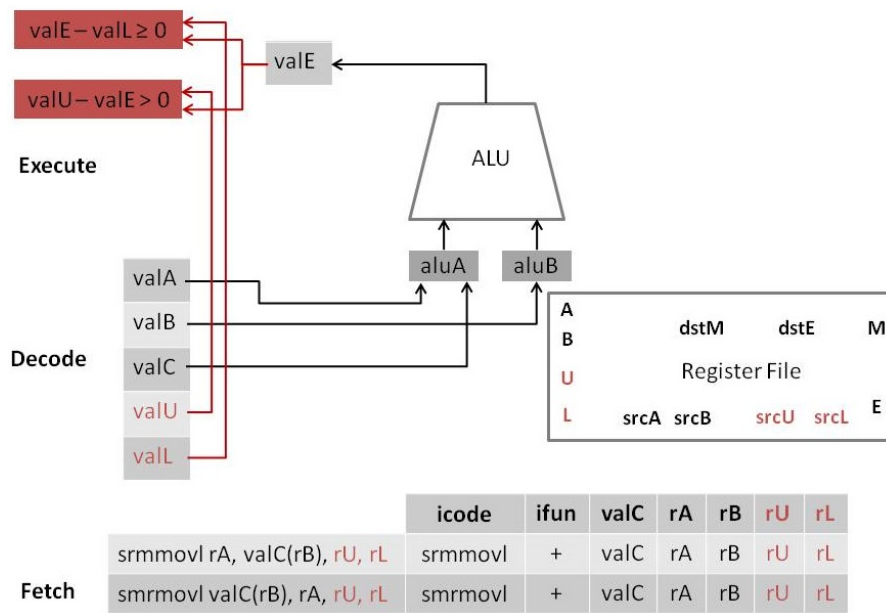


Figure 4.3: Execute stage in the modified Y86 pipeline.

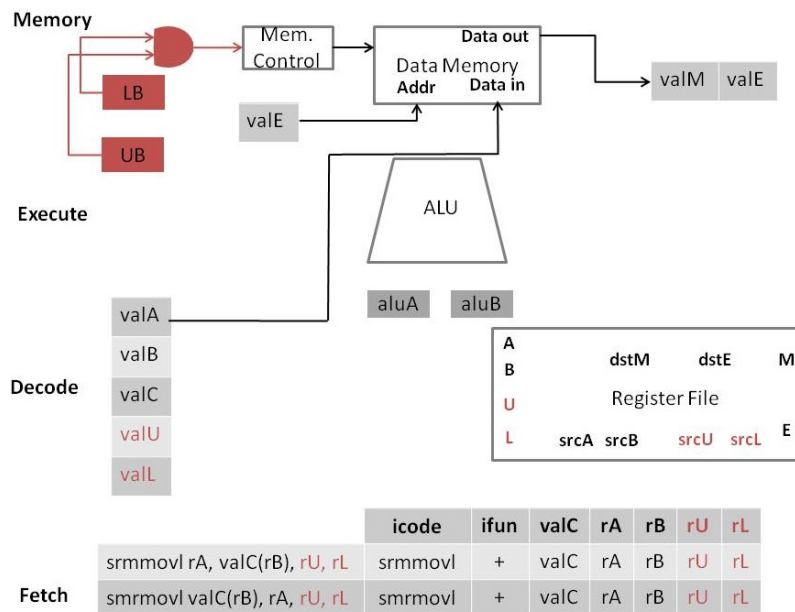


Figure 4.4: Memory stage in the modified Y86 pipeline.

To summarize, Table 4.1 shows how our secure instructions evolve through the pipeline stages. A few of the operations performed are just the same as those from the original Y86 `rmmovl` and `mrmovl` instructions. The portions highlighted in red are the ones we introduced. A visual representation of the `srmovl` and `smrmovl` encodings, along with

the meaning of each field, is illustrated by Figure 4.5. Finally, the entire pipeline, registers, and major components are presented by Figure 4.6. The parts in red are again our own additions.

Stage	srmovl $rA, D(rB), rU, rL$	smrmovl $D(rB), rA, rU, rL$
Fetch	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC + 1]$ $valC \leftarrow M_4[PC + 2]$ $rU : rL \leftarrow M_1[PC + 6]$ $valP \leftarrow PC + 7$ $PC \leftarrow valP$	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC + 1]$ $valC \leftarrow M_4[PC + 2]$ $rU : rL \leftarrow M_1[PC + 6]$ $valP \leftarrow PC + 7$ $PC \leftarrow valP$
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$ $valU \leftarrow R[rU]$ $valL \leftarrow R[rL]$	$valB \leftarrow R[rB]$ $valU \leftarrow R[rU]$ $valL \leftarrow R[rL]$
Execute	$valE \leftarrow valB + valC$ $UB \leftarrow (valU - valE > 0)$ $LB \leftarrow (valE - valL \geq 0)$	$valE \leftarrow valB + valC$ $UB \leftarrow (valU - valE > 0)$ $LB \leftarrow (valE - valL \geq 0)$
Memory	$UB \ \&\& \ LB ? M_4[valE] \leftarrow valA : Except$	$UB \ \&\& \ LB ? valM \leftarrow M_4[valE] : Except$
Write back		$R[rA] \leftarrow valM$

Table 4.1: Processing required for `srmovl` and `smrmovl` instructions.

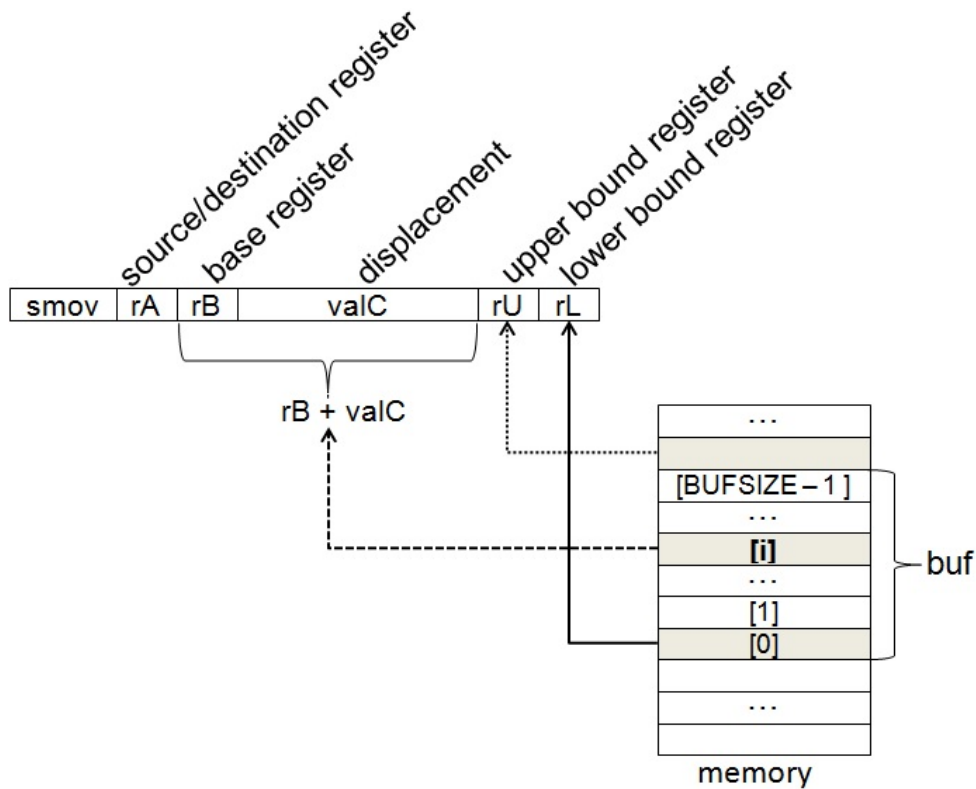


Figure 4.5: Encoding and meaning of each field of a SMOV instruction.

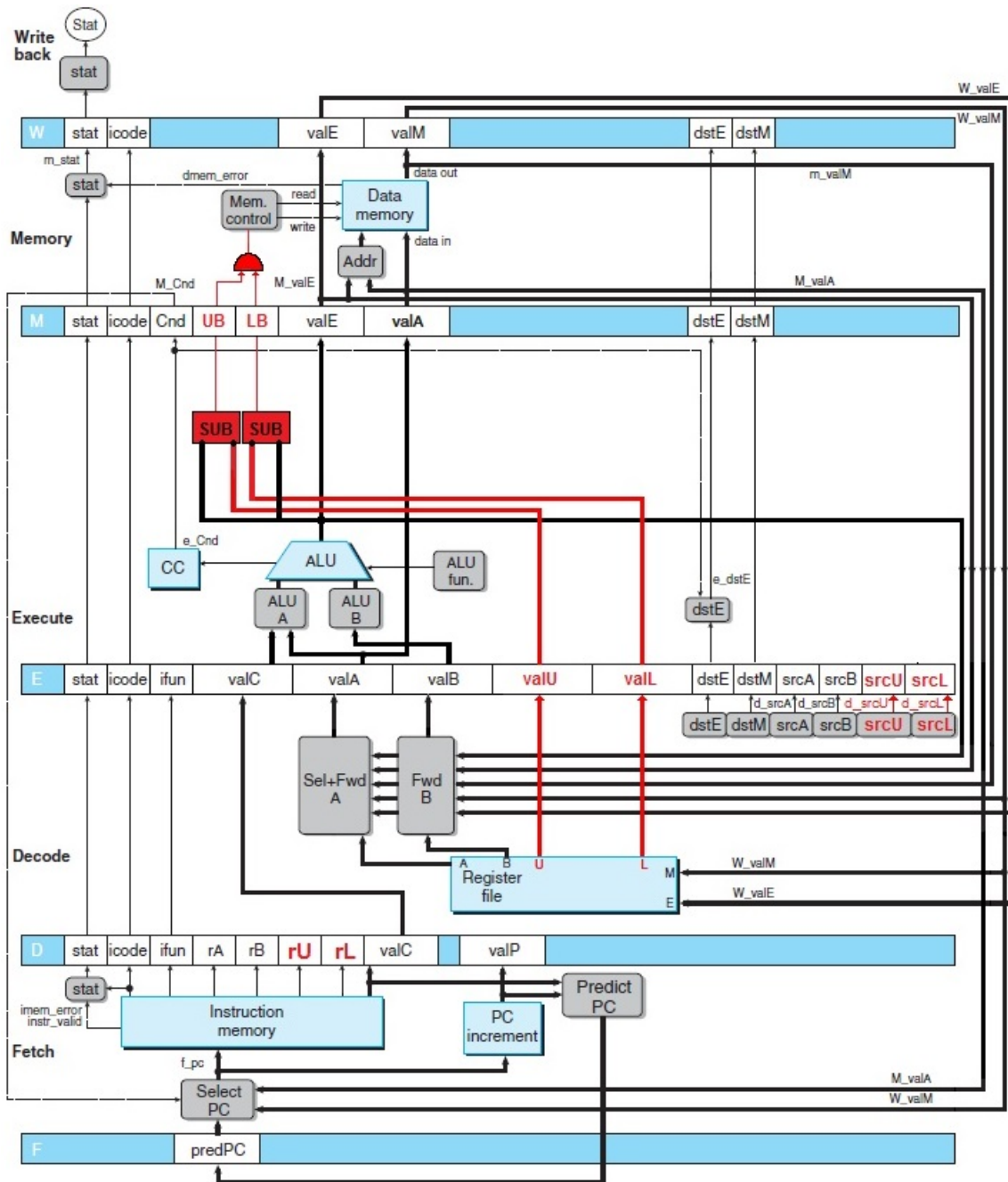


Figure 4.6: Complete pipeline and associated components with our additions in red.

4.1.2 Programming Interface and Capabilities

We identify a few prominent aspects that, although not intrinsically tight to our ABC proposal, are relevant to be discussed. In this regard, we raise attention to the fact that SMOV is a high-level proposal to achieve bound-checking and memory access within a single instruction. Specifically, it consists of a secure version of load and store operations, combined with

a technical reference on how to implement it on a variable-length ISA pipeline, accompanied by a deep analysis on the involved aspects.

SMOV is not to be thought of as a complete memory protection platform such as Intel's MPX [Intel Corporation, 2013]. In fact, given the extent and flexibility of x86, we claim that our combined bound-checking and memory access secure instructions could augment MPX's programming interface that is currently composed of only two bound-checking instructions, namely `bndcu` and `bndcl`, that independently check the upper and lower bound of an array, and without completing the memory access.

As an orthogonal aspect, MPX provides a way to store and load array bounds that are kept in a Bound-Register Table. In fact, this is reflected at the Application Binary Interface (ABI) level that comes with bound-preserving calling conventions. Such a concept could be equally implemented on an architecture that implements SMOV's secure instructions. A similar observation concerns the bound-specific registers provided by MPX, to which SMOV would impose no restrictions.

A particularly useful characteristic of our secure instructions is that they carry underneath all the data necessary to perform a non-secure load or store. Without knowing how exactly an Intel processor transforms (through microcode) an MPX instruction into a `NOB`, when running on unsupported hardware, we imagine that a similar transformation could be employed to convert a secure load/store instruction into a standard load/store.

4.2 Implementation Efforts

The foundation of our implementation is both the simulator and the synthesizable Verilog from Bryant and David Richard [2003] available online². We extended the Y86 processor with the hardware modifications that we propose and, of course, with the handling of `smrmovl` and `srmovl` instructions. Following, we have the C (software simulator) and Verilog (hardware project) codes showing the changes in each pipeline stage of the processor:

- Fetch: in the Codes 4.1 and 4.2 we can verify the fetch of the registers that keep the bounds information `rL` and `rU`;

```
regids = HPACK(REG_NONE, REG_NONE);
if (gen_need_plus_regids()) {
    get_byte_val(mem, valp, &regids);
    valp ++;
```

²<http://csapp.cs.cmu.edu/public/students.html>

```

}
if_id_next->ru = HI4(regids);
if_id_next->rl = LO4(regids);

```

Code 4.1: Part of the C code needed to implement `smrmovl` and `srmovl` in the Fetch Stage of the Y86 simulator.

```

assign rA = ibytes[7:4];
assign rB = ibytes[3:0];
assign valC = need_regids ? ibytes[39:8] : ibytes[31:0];
assign rL = need_plus_regids ? ibytes[43:40] : 4'hf;
assign rU = need_plus_regids ? ibytes[47:44] : 4'hf;

```

Code 4.2: Part of the Verilog code needed to implement `smrmovl` and `srmovl` in the Fetch Stage of the Y86 hardware project.

- Decode: in this stage it's the implementation of the two new read ports added to the register file. The values read are kept in variables named `valL` and `valU`, as we can see in Codes 4.3 and 4.4;

```

d_regvala = get_reg_val(reg, id_ex_next->srca);
d_regvalb = get_reg_val(reg, id_ex_next->srcb);
d_regvalu = get_reg_val(reg, id_ex_next->srcu);
d_regvall = get_reg_val(reg, id_ex_next->srcl);

```

Code 4.3: Part of the C code needed to implement the new two read ports in the register file - Decode Stage of the Y86 simulator.

```

module regfile(dstE, valE, dstM, valM,
              srcA, valA, srcB, valB,
              srcL, valL, srcU, valU,
              reset, clock,
              eax, ecx, edx, ebx, esp, ebp, esi, edi);
...
input [3:0] srcL;
output [31:0] valL;
input [3:0] srcU;
output [31:0] valU;
...
assign valL =

```

```

    srcL == REAX ? eax :
    srcL == RECX ? ecx :
    srcL == REDX ? edx :
    srcL == REBX ? ebx :
    srcL == RESP ? esp :
    srcL == REBP ? ebp :
    srcL == RESI ? esi :
    srcL == REDI ? edi :
    0;

assign    valU =
    srcU == REAX ? eax :
    srcU == RECX ? ecx :
    srcU == REDX ? edx :
    srcU == REBX ? ebx :
    srcU == RESP ? esp :
    srcU == REBP ? ebp :
    srcU == RESI ? esi :
    srcU == REDI ? edi :
    0;

```

Code 4.4: Part of the Verilog code needed to implement the new two read ports in the register file - Decode Stage of the Y86 hardware project.

- Execute: the Arithmetic Logic Unit have to be extended to calculate two more subtractions, setting the values of the secure. These operations are shown in Codes 4.5 and 4.6;

```

word_t compute_alu(alu_t op,
                  word_t argA, word_t argB,
                  bool_t *lower,
                  bool_t *upper,
                  word_t argL, word_t argU) {
    ...
    word_t subVal, subA, subB;
    bool_t zero, sign, ovf;
    subA = argL;
    subB = val;

```

```

subVal = subB-subA;

zero = (subVal == 0);
sign = ((int)subVal < 0);
ovf = (((int) subA > 0) == ((int) subB < 0)) &&
      (((int) subVal < 0) != ((int) subB < 0));
*lower = ( (sign^ovf^1)|(zero) );

subA = val;
subB = argU;
subVal = subB-subA;
zero = (subVal == 0);
sign = ((int)subVal < 0);
ovf = (((int) subA > 0) == ((int) subB < 0)) &&
      (((int) subVal < 0) != ((int) subB < 0));
*upper = ( (sign^ovf^1)&(zero^1) );

return val;
}

```

Code 4.5: Part of the C code that implements the two extra subtractions in the Execute state of the Y86 simulator.

```

module alu(aluA, aluB, alufun, valE, new_cc,
          lower_bound, upper_bound, aluL, aluU);
...
input [31:0] aluL, aluU;
output      lower_bound;
output      upper_bound;
...
assign lower_bound = ((aluB + aluA) >= aluL)? 1'b1:1'b0;
assign upper_bound = ((aluB + aluA) < aluU)? 1'b1:1'b0;

```

Code 4.6: Part of the Verilog code that implements the two extra subtractions in the Execute state of the Y86 hardware project.

- **Memory:** the control signal related to the secure access must be evaluated to decide if the flow continues or is interrupted. In the Codes 4.7 and 4.8 we can see these signals being checked/generated.


```

mem_addr = gen_mem_addr();
mem_data = gen_mem_data();
sec_write = gen_mem_sec_write();
mem_write = gen_mem_write();
mem_write = (mem_write) &&
             (!sec_write ||
              (sec_write &&
               sec_upper_bound &&
               sec_lower_bound));

sec_read = gen_mem_sec_read();
mem_read = gen_mem_read();
mem_read = (mem_read) &&
            (!sec_read ||
             (sec_read &&
              sec_upper_bound &&
              sec_lower_bound));

...
if( (sec_write || sec_read) &&
    (!sec_upper_bound || !sec_lower_bound) ){
    sim_log("\tRaise Interruption\n");
}

```

Code 4.7: Part of the C code where the secure signals are evaluated and the interruption is set - Memory Stage of the Y86 simulator.

```

assign interruption = ((mem_sec_read & !top_mem_read) |
                      (mem_sec_write & !top_mem_write));

```

Code 4.8: Part of the Verilog code where the interruption signals is set - Memory Stage of the Y86 hardware project.

From that on, we were able to have concrete measurements of how many clock cycles a program takes to execute.

Based on Intel's reference documentation [Intel Corporation, 2013], we also developed an alternative ABC scheme inspired by MPX on top of Y86, so we could make some comparisons in our benchmark. In this scheme, we implemented the MPX basic instructions:

1. `bndmk` to create the bound registers;

2. `bndcu` to verify the upper bound;
3. `bndcl` to check the lower bound.

Figure 2.13 presents the modified pipeline considering MPX-based instructions.

Naturally, we also needed to extend the author’s assembler, which is now also able to generate Y86 object code for our instructions. Nevertheless, a quite useful practical contribution we have is a synthesizable Verilog implementation for the well-known Altera DE2-115 FPGA³ using a real SRAM. Although it might not be clear at first, the original Y86 Verilog code synthesizes a memory component within the FPGA itself, assuming a somewhat peculiar memory specification which allows a more convenient implementation of the architecture without having to worry about alignment, concurrent reads during *Fetch* and *Memory* stages, and the ability to entirely retrieve a maximum-length instruction of 6 bytes in one go, which might not be possible in a 32-bit system.

We also have an on-going effort that, to our knowledge, will be the first Y86 compiler. This is an LLVM-based backend publicly available⁴. Once the fundamental parts are complete we plan to integrate our safe instructions combined with an smart front-end that would help us improve code generation by eliminating ABCs [Nazaré et al., 2014] whenever we detect it is safe to do so.

³<https://github.com/ltmlcmelo/Y86Proc>

⁴<https://github.com/ltmlcmelo/llvm>

Chapter 5

Evaluation

In this chapter, we present our experiments to evaluate SMOV. We describe our methodology in Section 5.1, and analyze the results in Section 5.2. In Section 5.3, we discuss a Verilog synthesis of SMOV and estimate the hardware overhead when compared to the original Y86 architecture.

5.1 Methodology

To evaluate our work we use three programs from the Stanford [Lattner and Adve, 2004] benchmark: Bubblesort, Quicksort and Perm. They were slightly modified to make the Y86 assembly code generation more convenient. The programs were compiled using gcc version 5.0.0 - revision 214719, this version of the compiler allows us to generate code with the MPX extensions enabled. We currently have an on-going effort to create what would be the first Y86 compiler, however it is not ready yet. Therefore, we converted the 32-bit x86 assembly code into Y86 using an in-house "transliterate" script. We then obtained a binary compatible with our simulator by using modified assemblers that understand SMOV or MPX instructions. For each program, we considered four variations:

1. **Unprotected:** each array access is done without bound-checks. Code 5.1 presents the C code of the sort function in the Bubblesort program;

```
void Bubble() {
    int i, j;
    top=n-1;
    while ( top>0 ) {
        i=0;
        while ( i<top ) {
```

```

    if ( list[i] > list[i+1] ) {
        j = list[i];
        list[i] = list[i+1];
        list[i+1] = j;
    }
    i=i+1;
}
top=top-1;
}
}

```

Code 5.1: Unprotected version of the C code of the sort function in Bubblesort.

2. **Software-based:** a safe version of the program, but with additional software ABCs - implemented with conditional branches inserted before all load and store instructions that involve array accesses. In Code 5.2 we have all the ABCs highlighted;

```

void Bubble() {
    int i, j;
    top=n-1;
    while ( top>0 ) {
        i=0;
        while ( i<top ) {
            if((i>=0) && (i<n) && (i+1>=0) && (i+1<n))
            if ( list[i] > list[i+1] ) {
                if ((i>=0) && (i<n) )
                j = list[i];
                if((i>=0) && (i<n) && (i+1>=0) && (i+1<n))
                list[i] = list[i+1];
                if ((i+1>=0) && (i+1<n))
                list[i+1] = j;
            }
            i=i+1;
        }
        top=top-1;
    }
}

```

```
}

```

Code 5.2: C code of the sort function in Bubblesort with array bound checks implemented directly by the programmer.

3. **MPX**: using the appropriate gcc flags¹ we generate MPX-conformant code. The MPX instructions used to load the limits of an array to special registers (bndmk), to verify the upper bound of an array (bndcu), and to verify the lower bound of an array (bndcl) can be seen highlighted in Code 5.3. This code shows part of the Bubblesort algorithm implemented in the Y86 assembly language;

```
Bubble:
...
    irmovl 1604,%eax
    irmovl sortlist,%ebx
    bndmk %ebx,%eax,%bnd0
bndmk    %ebx,%eax,%bnd0
...
    mrmovl -4(%ebp), %eax
    rrmovl %edx, %edi
    sall  $2, %eax
    iaddl list, %edi
bndcl   %edi,%bnd0
bndcu   3(%edi),%bnd0
    rmmovl %eax,(%edi)
...

```

Code 5.3: Part of the code of the Bubblesort function protected with MPX instructions in Y86 assembly.

4. **SMOV**: our hardware-based solution. In the assembly code, all unprotected versions of load and store instructions that manipulate arrays were replaced by a SMOV instruction. The identification of such loads and stores is simplified by the fact that the arrays from the evaluation programs are global. This makes gcc use the variable's name as a displacement, simulating a compiler intelligence for the array-bounds tracking. Table 5.1 illustrates the identification of unprotected global array accesses (left) and their SMOV replacement (right).

¹-fcheck-pointer-bounds -mmpx

Bubble:	Bubble:
...	...
<code>mrmovl -4(%ebp), %eax</code>	<code>mrmovl -4(%ebp), %eax</code>
<code>iaddl \$1, %eax</code>	<code>iaddl \$1, %eax</code>
<code>rrmovl %eax, %edi</code>	<code>rrmovl %eax, %edi</code>
<code>sall \$2, %edi</code>	<code>sall \$2, %edi</code>
<code>mrmovl list(%edi), %edx</code>	<code>irmovl list, %ebx</code>
<code>mrmovl -4(%ebp), %eax</code>	<code>irmovl list, %ecx</code>
<code>rrmovl %eax, %edi</code>	<code>iaddl \$1597, %ecx</code>
<code>sall \$2, %edi</code>	<code>smrmovl list(%edi), %edx, %ecx, %ebx</code>
<code>mrmovl -4(%ebp), %eax</code>	<code>mrmovl -4(%ebp), %eax</code>
<code>rrmovl %eax, %edi</code>	<code>rrmovl %eax, %edi</code>
<code>sall \$2, %edi</code>	<code>sall \$2, %edi</code>
<code>rrmmovl %edx, list(%edi)</code>	<code>irmovl list, %ebx</code>
 	<code>irmovl list, %ecx</code>
 	<code>iaddl \$1597, %ecx</code>
 	<code>srmovl %edx, list(%edi), %ecx, %ebx</code>
...	...

Table 5.1: Identification of global array accesses (left) and their replacement to use SMOV (right).

A point that can be raised by the analysis of the Code 5.3 is about the different access addresses used by the MPX bound-checking instructions and those from SMOV. Notice that the memory address used to check the upper bound (`bndc1`) is the sum of the address used by the lower bound-check (`bndcu`) plus three bytes (instruction `bndcu 3(%edi), %bnd0`). This is due to the fact the operation involves a 4-byte integer, which means that if the memory access starts at, let's say, $addr$, it will affect the bytes $M_1[addr]$, $M_1[addr + 1]$, $M_1[addr + 2]$ and $M_1[addr + 3]$. Therefore, the lower bound must be compared with the lowest memory position affected, and the upper bound compared with the highest memory address affected.

On the other hand, our solution uses the same address to check both bounds in the same instruction. SMOV keeps the bounds in general purpose registers demanding the load of this information before each use of the secure instructions, so the compiler, which knows the data-type in question, can properly set the upper bound letting the last possible address in accordance with the size of the data being manipulated. This behavior is shown on the right side of the Table 5.1 by the instructions `iaddl $1597, %ecx`. The real size of the array is 1600 (400 integer elements) but the value 1597 ($400 \times 4 - 3$) is being used as upper bound.

We ran each program with one of our simulators (MPX or SMOV)², and collected the reported number of instructions and cycles. Results are discussed in the next section.

²The unprotected and software-secured versions can be run in any Y86 simulator. The results in terms of cycles and number of instructions must be the same.

5.2 Results

In this section, we present and discuss our results (Table 5.2).

Bubblesort	Instructions	Cycles	CPI	Exec. time (s)	Runtime Overhead
Unprotected	328622114	432722716	1.32	54.090	-
Software-based	735922114	1079582716	1.47	134.948	149%
MPX	472382714	576483316	1.22	72.060	33%
SMOV	424462114	528562716	1.25	66.070	22%
Quicksort	Instructions	Cycles	CPI	Exec. time (s)	Runtime Overhead
Unprotected	2024315	2650117	1.31	0.331	-
Software-based	3654915	5096217	1.39	0.637	92%
MPX	2637215	3263017	1.24	0.408	23%
SMOV	2430915	3056717	1.26	0.382	15%
Perm	Instructions	Cycles	CPI	Exec. time (s)	Runtime Overhead
Unprotected	205149315	258538417	1.26	32.317	-
Software-based	366425315	500452417	1.37	62.557	94%
MPX	278619615	332008717	1.19	41.501	28%
SMOV	245468315	298857417	1.22	37.357	16%

Table 5.2: Evaluation result of different ways to guard memory accesses.

The first column of Table 5.2 shows the program and its version. The next two columns show the total number of instructions and cycles needed to run the program. CPI gives us the number of cycles per instruction. To compute the execution time, we consider a hypothetical 8 Mhz processor. Finally, the last column reports the runtime overhead when we compare each safe version against the unprotected – original – code.

If we consider Quicksort, then we see that the software-based ABCs have caused a 92% runtime overhead. All the hardware approaches have decreased this amount, albeit by different factors. The overhead imposed by MPX is 23%. SMOV yields an overhead of 15%. This result show us that SMOV yields 40% less cycles than the software strategy and 6% less cycles than the MPX approach when employed in our benchmark. Still considering the hypothetical 8 Mhz processor, we have that the code with ABCs implemented in software, and the code with MPX extensions run in 0.637s and 0.408s. Our solution, on the other hand, takes only 0.382s to execute.

The results observed in the Bubblesort case study, SMOV has given us a 22% runtime overhead while the software-based and MPX-based approaches have yielded overheads of

149% and 33%, respectively. These numbers mean that SMOV reduced the runtime overhead caused by the software version by 83% and the runtime overhead caused by the MPX version by 33%. In the 8 Mhz processor these numbers represent 134.948s and 72.060s for the last strategies and 66.070s for our solution.

For the last the program, Perm, SMOV reduced the runtime overhead caused by the software-based ABC, and MPX variations by 83%, and 45%. The runtime overhead found in software-based version was 94%, in MPX it was 28%, and in SMOV 16%. In terms of execution time, the SMOV variation runs in 37.357s, the software-based approach takes 62.557s, and the MPX version 41,501s, considering the hypothetical 8 Mhz processor.

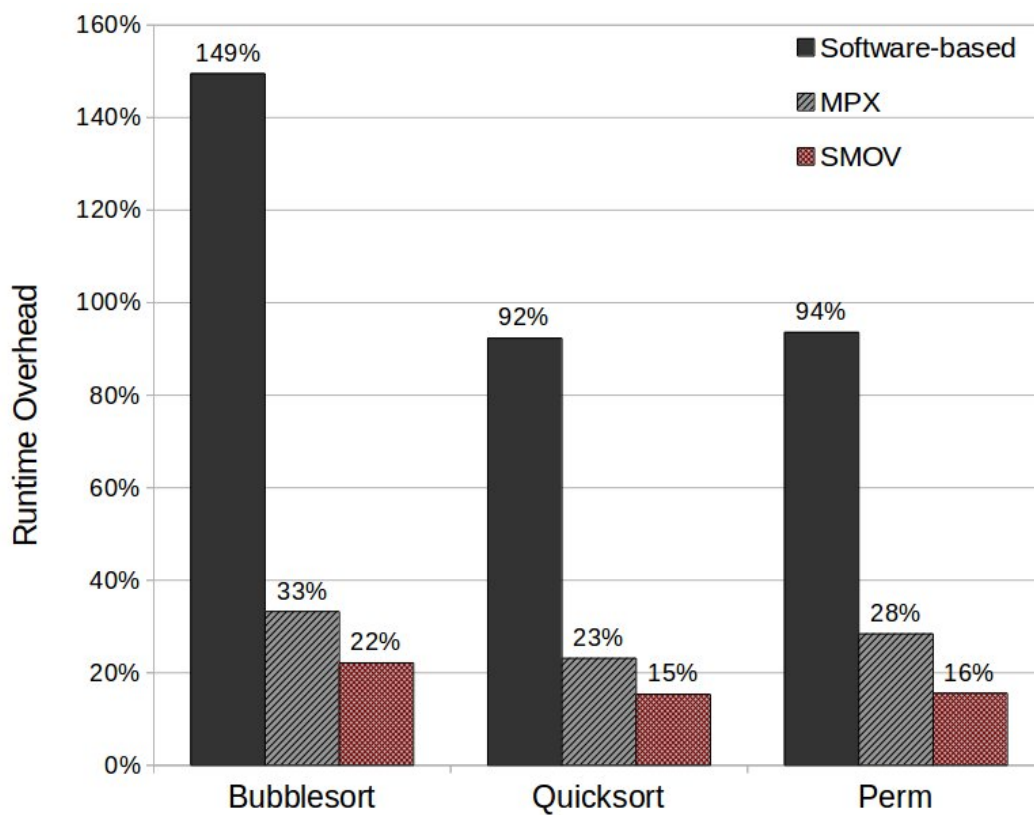


Figure 5.1: Runtime overhead caused by different security approaches.

Figure 5.1 provides a visual comparison between the different runtime overheads imposed by the three safe approaches. SMOV had its best relative results in the Perm case study. This happened because Perm performs all possible combinations of elements in an array, which represents a high number of array accesses. Thus, the implementation of this algorithm exercises more array bound checks.

In all case studies, we had similar behavior in the Cycles per Instructions relation when comparing with the unprotected version. The software-based variation has increased

CPI. This increase comes from the fact that software ABCs are implemented through conditional branches instructions, which can, sometimes, lead to branch mispredictions. In face of mispredictions, some extra cycles are required to adjust the program's execution. Another possible cause of the increase is data hazards – when the next instruction depends on the result of the current one. Data hazards also can lead to a stall in the processor, requiring extra cycles to manipulate the correct data. Our solution, and also MPX, do not increase the CPI, on the contrary, they reduced this relation. The reduction happens because SMOV and MPX increase the number of executed instructions, however the majority of these instructions do not incur in stalls in the processor. The extra registers in MPX reduce even more the CPI relation. That is because having more registers, the probability of data hazards evolving general propose registers is reduced.

Figure 5.2 shows the speed up obtained by each hardware solution, when compared to the software-based approach. These data gives us an idea about how faster is the runtime of a program when we change from a software-based ABC to one of the hardware-based approaches. The results show that MPX can accelerate the runtime for Bubblesort, Quicksort, and Perm by 87%, 56%, and 50%, respectively, while SMOV gives accelerations of 104%, 66%, and 67%.

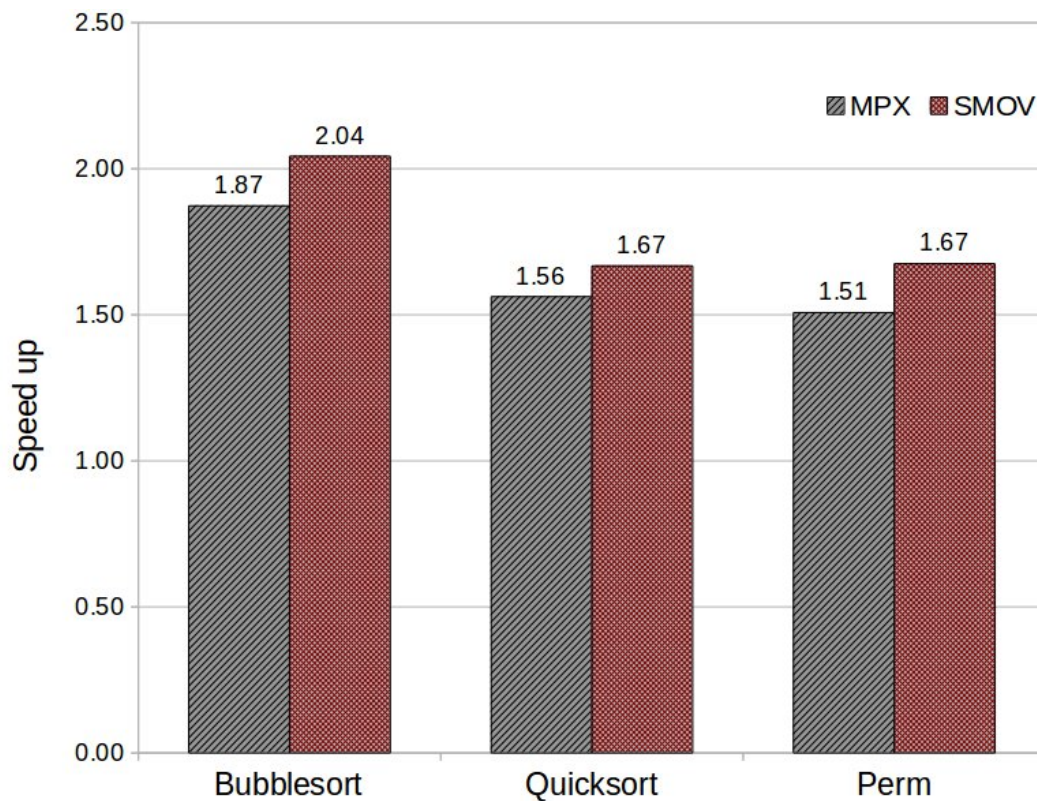


Figure 5.2: Speed up of the two hardware strategies on top of the software-based approach.

The results found in our experiments tell us that, on average, SMOV reduces the runtime overhead caused by the software approach by 83% and the runtime overhead caused by MPX by 37%. The average speedup found indicates that our solution can run these programs 1,79x faster than when the ABCs are implemented in software, and 1,09x faster than when they are implemented using the MPX extensions.

5.3 Hardware Cost

It is a hard task to estimate the hardware cost without having the actual hardware. However, the literature indicates us that a good estimative is the number of logic elements reported by the design tool after the HDL synthesis. The table 5.3 presents this result besides the number of C code lines needed to implement each version of the simulator, the size of each executable, and the number of lines of Verilog code needed to describe each hardware project.

	Y86 original project	MPX	SMOV
Number of lines - C code	2459	3002	2881
Simulator - executable size(KBytes)	52	254	62
Number of lines - Verilog code	1004	1417	1341
Number of logical elements	279	320	289

Table 5.3: Comparison of different implementations of the Y86 processor.

The number of logical elements confirm us that SMOV can be implemented with few hardware changes. In fact, the overhead gotten when comparing with the original version (Figure 5.3) is only 3.58%. On the other hand, the MPX-like project took 14.70% more logical elements than the original version. The bigger overhead reached by the MPX version is explained by the new register file that needs to be added to the project.

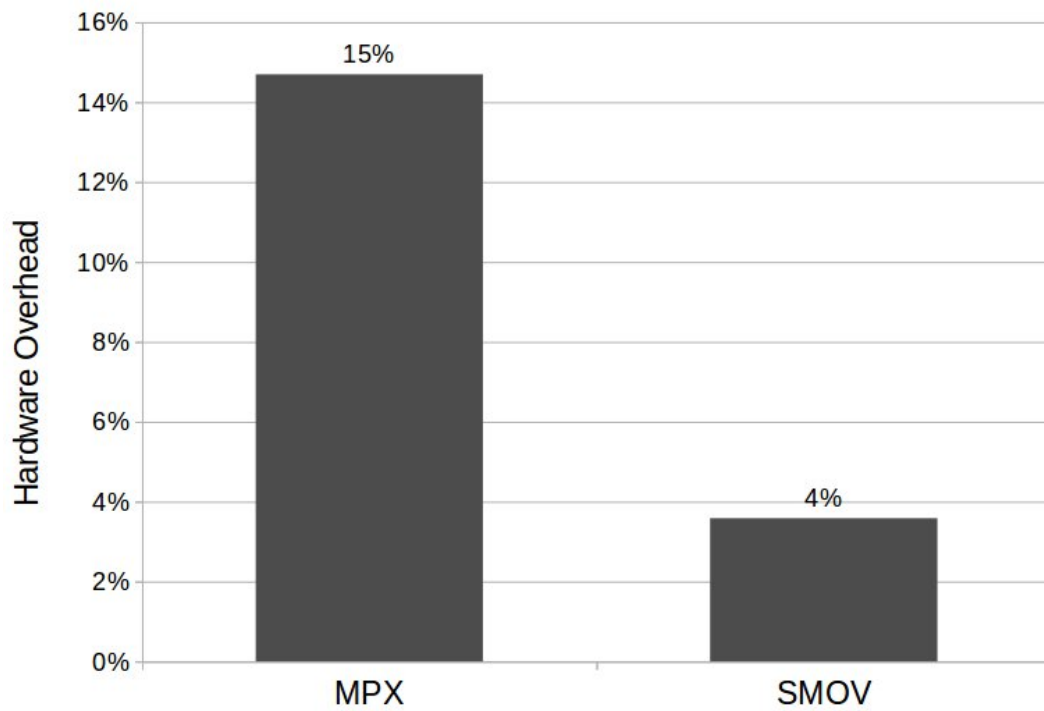


Figure 5.3: Logical elements overhead to implement the two hardware strategies on top of the original Y86 project.

Chapter 6

Conclusion and Future Work

Programming languages that do not natively provide ABCs, such as C or C++, are susceptible to BOFs. There are both software- and hardware-based approaches that aim at providing memory protection to programs written in those languages. This is normally done by instrumenting the source with assertion or in hardware, via a combination of dedicated instructions. However, those proposals end up causing a high runtime overhead, compromising programs' performance, or incurring on a significant hardware size/cost penalty. This work proposes SMOV, a novel hardware-based approach able to perform ABCs and memory access within a single instruction. SMOV consists of a pair of secure load and secure store instructions. We present our solution from a practical but still general perspective, accompanied by the formalism necessary to hardware design. We focus on demonstrating how it can be implemented on a typical variable-length ISA architecture and discuss all relevant hardware details, an aspect that is not covered by previous work. Our proof of concept, implemented on top of an academical architecture, presented an overhead of 3.6% in terms of hardware size compared to the original processor. In our experimental results, SMOV could secure the considered programs causing a runtime overhead of 18%, running, on average, 1.79 times faster than a corresponding program with software-based ABC. The two results combined show that our solution reaches good level of security a simple and cheap hardware project.

One of the challenges we faced during our experiments was the absence of an Y86 compiler. We had to adjust the programs almost by hand. A script was developed to translate instructions from x86 assembly to Y86, however we still had to check the correctness of the translation and locate the memory access instructions to change them by the secure ones. Some workarounds were used - global arrays to identify array access, for instance - even so, the process could be automated by a tool working in one of the compilation phases if such compiler existed. This is a natural extension of this work. There is already an on-going effort to develop the Y86 compiler and tools that uses a hardware apparatus like ours.

We believe that SMOV is a step towards delivering users secure, reliable, and ultimately systems with improved efficiency. Besides, we are also making contributions towards the improvement of the Y86 ecosystem, by extending the experimentation platform and the simulator, by providing synthesizable Verilog for a well-known FPGA, and by the development of the aforementioned compiler.

Bibliography

- Bell, T. (1999). The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes*.
- Bellard, F. (2005). Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*.
- Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., et al. (2011). The gem5 simulator. *ACM SIGARCH Computer Architecture News*.
- Binkert, N. L., Dreslinski, R. G., Hsu, L. R., Lim, K. T., Saidi, A. G., and Reinhardt, S. K. (2006). The m5 simulator: Modeling networked systems. *IEEE Micro*.
- Bishop, M., Engle, S., Howard, D., and Whalen, S. (2012). A taxonomy of buffer overflow characteristics. *IEEE Transactions on Dependable and Secure Computing*.
- Bryant, R. and David Richard, O. (2003). *Computer systems: a programmer's perspective*. Prentice Hall.
- Buchanan, E., Roemer, R., Shacham, H., and Savage, S. (2008). When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the Conference on Computer and Communications Security (CCS'08)*.
- Carlini, N. and Wagner, D. (2014). Rop is still dangerous: Breaking modern defenses. In *Proceedings of the USENIX Security Symposium (USENIX Security'14)*.
- Chess, B. and West, J. (2007). *Secure programming with static analysis*. Addison-Wesley Professional.
- Chipounov, V. and Candea, G. (2011). Enabling sophisticated analyses of x86 binaries with revgen. In *Proceedings of the International Conference on Dependable Systems and Networks Workshops (DSN-W'11)*.

- Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2005). The ASTRÉE analyzer. In *Proceedings of the European Symposium on Programming (ESOP'05)*.
- Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., et al. (1998). Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium (USENIX Security'98)*.
- Cowan, C., Wagle, F., Pu, C., Beattie, S., and Walpole, J. (2000). Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the Information Survivability Conference and Exposition (DISCEX'00)*.
- Devietti, J., Blundell, C., Martin, M. M., and Zdancewic, S. (2008). Hardbound: architectural support for spatial safety of the c programming language. *ACM SIGOPS Operating Systems Review*.
- Dhurjati, D., Kowshik, S., and Adve, V. (2006). SAFECODE: enforcing alias analysis for weakly typed languages. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'06)*.
- Evans, D. and Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *IEEE software*.
- Francillon, A., Perito, D., and Castelluccia, C. (2009). Defending embedded systems against control flow attacks. In *Proceedings of the Workshop on Secure Execution of Untrusted Code (SecuCode'09)*.
- Haugh, E. and Bishop, M. (2003). Testing c programs for buffer overflow vulnerabilities. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS'03)*.
- Holzmann, G. J. (2002). Static source code checking for user-defined properties. In *Proceedings of the Integrated Design and Process Technology (IDPT'02)*.
- Intel Corporation (2013). Intel Architecture Instruction Set Extensions Programming Reference. <https://software.intel.com/en-us/isa-extensions>.
- Jones, R. W. and Kelly, P. H. (1997). Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of the Symposium on Automated and Analysis-Driven Debugging (AADEBUG'97)*.

- Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the Code Generation and Optimization (CGO'04)*.
- Lhee, K.-S. and Chapin, S. J. (2003). Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience*.
- McGregor, J. P., Karig, D. K., Shi, Z., and Lee, R. B. (2003). A processor architecture defense against buffer overflow attacks. In *Proceedings of the International Conference on Information Technology: Research and Education (ITRE'03)*.
- Misra, D. K. (1987). A quasi-static analysis of open-ended coaxial lines. *IEEE Transactions on Microwave Theory and Techniques*.
- Moore, D., Shannon, C., and claffy, k. (2002). Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of the Workshop on Internet Measurement (IMW'02)*.
- Nagarakatte, S., Martin, M. M., and Zdancewic, S. (2012). Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the International Symposium on Computer Architecture (ISCA'12)*.
- Nagarakatte, S., Martin, M. M., and Zdancewic, S. (2014). Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'14)*.
- Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. (2010). CETS: compiler enforced temporal safety for c. In *Proceedings of the International Symposium on Memory Management (ISMM'10)*.
- Nazaré, H., Maffra, I., Santos, W., Oliveira, L. B., Gonnord, L., and Quintão Pereira, F. M. (2014). Validation of memory accesses through symbolic analyses. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*.
- Newsome, J. and Song, D. (2005). Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Computer Security Applications (ACSAC'00)*.
- Ozdoganoglu, H., Vijaykumar, T., Brodley, C. E., Kuperman, B. A., and Jalote, A. (2006). Smashguard: A hardware solution to prevent security attacks on the function return address. *IEEE Transactions on Computers*.

- Parulkar, I., Wood, A., Hoe, J. C., Falsafi, B., Adve, S. V., Torrellas, J., and Mitra, S. (2008). Opensparc: An open platform for hardware reliability experimentation. In *Proceedings of the Workshop on Silicon Errors in Logic-System Effects (SELSE'08)*.
- Piromsopa, K. and Enbody, R. J. (2006). Secure bit: Transparent, hardware buffer-overflow protection. *IEEE Transactions on Dependable and Secure Computing*.
- Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. (2012). Addresssanitizer: a fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference*.
- Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., and Ning, P. (2011). On the expressiveness of return-into-libc attacks. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID'11)*.
- Viega, J., Bloch, J.-T., Kohno, Y., and McGraw, G. (2000). ITS4: A static vulnerability scanner for c and c++ code. In *Proceedings of the Computer Security Applications (ACSAC'00)*.
- Wagner, D. and Dean, R. (2001). Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'01)*.
- Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A. (2000). A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS'00)*.
- Wilander, J. and Kamkar, M. (2003). A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS'03)*.
- Xie, Y., Chou, A., and Engler, D. (2003). Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the ACM SIGSOFT Software Engineering Notes (SEN'03)*.
- Yourst, M. T. (2007). PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS'07)*.