SMOV: Array Bound-Check and Access in a Single Instruction

Antonio Maia, Leandro Melo, Fernando Magno Quintão Pereira,
Omar P. Vilela Neto, and Leonardo B. Oliveira

Universidade Federal de Minas Gerais (UFMG) - Belo Horizonte, MG - Brazil
Email: {lemosmaia,ltcmelo,fernando,omar,leob}@dcc.ufmg.br

***Abstract*—A Buffer Overflow (BOF) continues to be among the top open doors to worms and malware. Earlier in 2014, the security world was taken by surprise when researches unveiled a BOF in OpenSSL. Languages like C and C++, widely used for system 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 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. Our results indicate that our solution can run programs 1,79x faster than the software-based approach.**

I. Introduction

A Buffer Overflow (BOF) takes place whenever a system allows data to be accessed out of the bounds of an array [1]–[4]. 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 [1].

The Morris worm [5] 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 Heartbleed [6], 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 [7], [8]. 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 $array$.

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

```
#define BUFSIZE 512

int main() {
  int buffer[BUFSIZE];
  int a,i,j;
  ...
  for(i;i<j;i++) {

      buffer[i] = a;
  }
}
```

```
#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;
  }
}
```

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 known as the No-eXecute bit (NX). That eventually led to more advanced attacks such as the Return-to-libC attack [9] and its variations like Return Oriented Programming [10]. Current defenses against those are not totally efficient [11].

A myriad of proposals endeavour 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-hand side of Table I.

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 programs 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 selected 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 [12], a popular tool maintained by Google, is known to cause 70% time and 200% memory consumption overhead.

Hardware-based approaches (e.g., [13]–[16]) offer new specific machine instructions for bound-checking purposes. They differ from each other in a variety of factors: the overall format and content 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 [17]–[20] or tightly coupled with a particular hardware architecture implementation.

Our contribution. In this work, we present Secure Move – SMOV, a hardware-based solution for BOFs which 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 an x86-like format, they can easily be extended to any architecture which 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 implemented a synthesizable Verilog for an academical processor which could be used for real evaluations.

Organization. The remainder of this paper is organized as follows. Section II introduces the necessary computer architecture background for understanding our work. As a follow up, Section III presents the technical details about SMOV and describe our reference implementation. In Section IV, evaluation results are discussed. Related work is available in Section V and we conclude in Section VI.

II. Background

In this section we discuss the relevant background for understanding our work. We first describe in more details what an ABC is and, afterwards, we recall the fundamentals of processor pipelining.

A. Array-Bound Checks (ABCs)

The naive way to protect a C program against a BOF is to always check that a particular index variable is greater than zero and less than the length of a array. Due to the performance impact of extra execution branches and, occasionally, to code styling conventions, this test is frequently avoided by programmers in contexts where that index is supposedly known to be valid. Of course, there are also situations in which the test is simply forgotten.

From a memory point of view, this test can be expressed in the following way. There is a memory location $\mathcal{M}_L$ which corresponds to the array's base address, i.e. `array[0]`. We will refer to this location as the *lower bound* of the valid memory region. Analogously, we refer to the *upper bound* as the memory location $\mathcal{M}_U$, which corresponds to the address of one past-the-end of the array's allocated memory. Finally, under this notation, `array[i]`, where `i` is the desired index to be accessed, is designated by $\mathcal{M}_I$.

What an ABC must ensure is that $\mathcal{M}_I - \mathcal{M}_L >= 0$ and that $\mathcal{M}_U - \mathcal{M}_I > 0$. If this condition is not met, execution of the program must be terminated.

When the above check is written in software, the performance cost might be prohibitive. In fact, there are works showing that such software-based ABCs can cause an overhead of up to 70% in execution time [12]. Therefore, most of the research in this area has led toward a solution directly in the hardware level. As presented in Section V, such proposals typically consist of extending the ISA with bound-checking purpose instructions. None of those, however, introduce a single instruction that performs both bound-checking and memory access, along with a description on how to implement it on a particular processor.

B. Processor Pipelining

There are numerous variations in processor design and implementation. It is unfeasible to investigate every particular aspect of them here. Having that in mind, we briefly review the fundamentals of processor pipelining as it is typically studied [21], [22]. Sustained by this content, our technical presentation of SMOV in Section III should apply without any loss of generality.

A classic pipeline is composed by the following five stages:

1) *Fetch:* The value of the Program Counter (PC) is loaded and the instruction to be executed is fetched. Afterwards, PC is updated to the next sequential PC. In architectures with fixed-length instructions this value is known in advance, while in the case of variable-length instructions this will be computed based on the length of the fetched instruction. Later on, the PC might be updated again in the case of a branch.
2) *Decode:* Registers specified in the instruction are read. Depending on the design, it might already be possible to perform an equality test and even complete a branch.
3) *Execute:* The Arithmetic Logic Unit (ALU) performs the operation specified in the instruction. In a load-store architecture, this is normally one of the following operations, as specified in the ALU's *function code* of the instruction's *opcode*: a register-register operation; a register-immediate operation; or a memory-reference operation. The later, an operation that adds a base register and a supplied offset to form an effective memory address, is the one we are particularly interested in this work.
4) *Memory:* A load instruction causes data from memory to be placed into a register. A store instruction causes a register value to placed back into memory.

5) *Write back:* Final computed values are finally saved into registers.

Within a pipeline a new instruction is initiated at every clock cycle. Consequently, a few issues need to be taken care of, the so-called *hazards*. There are three kinds of them: structural hazards, data hazards, and control hazards. Data and control hazards are not influenced anyhow by our changes. But we make an observation in the scope of structural hazards.

The register file of an architecture is commonly accessed at two distinct pipeline stages. In the fetch stage, the values of the registers specified in the instruction are read, and in the write-back stage, computed values are saved. For that reason, it is a reasonable assumption that a register file contains at least two read ports and two write ports, otherwise the pipeline would frequently be in *stall*. In additional, it may also be the case that the register being read and the register being written is actually the same register. A convention such as writing in the first half of the clock cycle and reading in the second half is therefore typically established [21]. As it will become clear in Section III, our approach requires a register file that allows four simultaneous reads.

III. SMOV

In this section 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.

Hardware performance benchmarks are often difficult to reproduce due to hardware implementation variations. While cycle-accurate simulators and emulators [18], [20] 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 [15], but those are even less accessible in the sense of validation and adoption. An interesting alternative would be open-source hardware [23], but unfortunately there does not seem to exist a full-blown x86 implementation available [24].

The strategy we have taken to realize SMOV is to implement it on top of the publicly available Y86 research architecture [22]. 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. Due to space limitations we do not make a fully-comprehensible introduction to the Y86, but only to aspects which are relevant within the context of our work. Nevertheless, 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 [22].

A. 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 → register)*;
- `rmmovl`: Store *(register → 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.

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`.

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.

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.

Table II. PROCESSING REQUIRED FOR `smrmovl` AND `srmmovl` INSTRUCTIONS.

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

To summarize, Table II 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 `srmmovl` and `smrmovl` encodings, along with the meaning of each field, is illustrated by Figure 1. Finally, the entire pipeline, registers, and major components are presented by Figure 2. The parts in red are again our own additions.

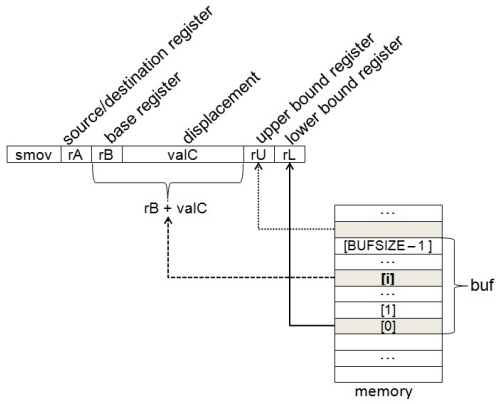

Figure 1. Encoding and meaning of each field of an SMOV instruction.

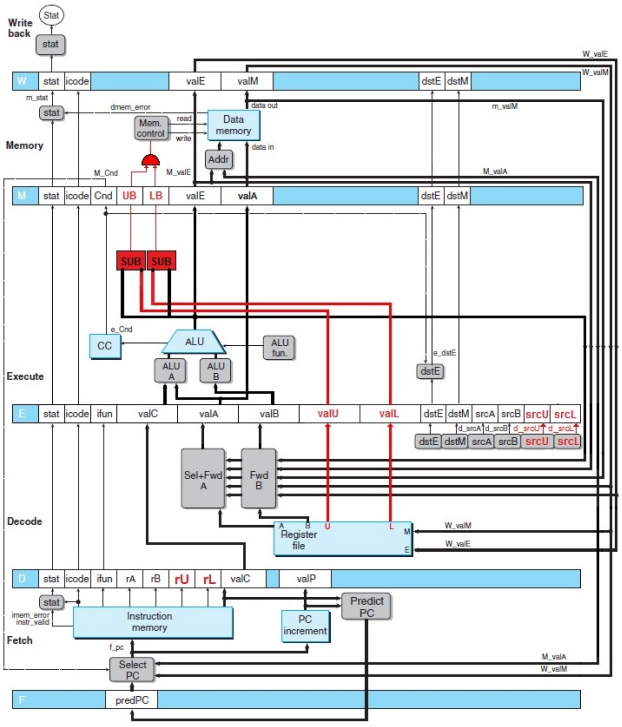

Figure 2. Complete pipeline and associated components with our additions in red.

B. 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 load and a secure 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 [16]. 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 till 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 which 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 `NOP`, 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.

IV. Evaluation

In this section we present our experiments with SMOV. We describe our methodology in Section IV-A and analyze the results in Section IV-B. In Section IV-C we discuss a Verilog synthesis of SMOV and estimate the hardware overhead when compared to the original Y86 architecture.

A. Methodology

To evaluate our work we use three programs from the Stanford [25] 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 4.8.2. We currently have an on-going effort to create what would be the first Y86 compiler, but 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 a modified assembler that understands SMOV instructions. For each program, we considered three variations:

Unprotected, where each array access is done without bound-checks.

Software-based, corresponding to the safe version of the program, but with additional software ABCs - implemented with conditional branches inserted before all load and store instructions that involved array accesses.

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 III illustrates the identification of unprotected global array accesses (left) and their SMOV replacement (right).

Table III. IDENTIFICATION OF GLOBAL ARRAY ACCESSES (LEFT) AND THEIR SMOV REPLACEMENT (RIGHT).

```
Bubble:
...
 mrmovl -4(%ebp), %eax
 iaddl   $1, %eax
 rrmovl %eax, %edi
 sall   $2, %edi


 mrmovl list(%edi), %edx
 mrmovl -4(%ebp), %eax
 rrmovl %eax, %edi
 sall   $2, %edi


 rmmovl %edx, list(%edi)
...
```

```
Bubble:
...
 mrmovl -4(%ebp), %eax
 iaddl   $1, %eax
 rrmovl %eax, %edi
 sall   $2, %edi
 irmovl lowerBound, %ebx
 irmovl upperBound, %ecx
 smrmovl list(%edi),%edx,%ecx,%ebx
 mrmovl -4(%ebp), %eax
 rrmovl %eax, %edi
 sall   $2, %edi
 irmovl lowerBound, %ebx
 irmovl upperBound, %ecx
 srmmovl %edx,list(%edi),%ecx,%ebx
...
```

We ran each program with our simulator and collected the reported number of instructions and cycles. Results are discussed in the next section.

B. Results

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

The first column of Table IV shows the variation of the evaluated program. In the following columns we have the total number of cycles needed to run the program and the number of cycles per instruction (CPI). To compute the execution time, showed in the next column, we consider a hypothetical 8Mhz processor. Finally, we have the runtime overhead caused by the insertion of ABCs when we compare each safe version against the unprotected (original) code. The results assume the architecture implements our design accordingly, without increasing the pipeline cycle duration.

Table IV. EVALUATION RESULTS FOR DIFFERENT WAYS TO GUARD MEMORY ACCESSES.

Bubblesort	**Cycles**	**CPI**	**Runtime (s)**	**Overhead**
unprotected	432722716	1.32	54.09	-
software	1079582716	1.47	134.94	149%
SMOV	528562716	1.25	66.07	22%
Quicksort	**Cycles**	**CPI**	**Runtime (ms)**	**Overhead**
unprotected	2650117	1.31	331.27	-
software	5096217	1.39	637.03	92%
SMOV	3056717	1.26	382.09	15%
Perm	**Cycles**	**CPI**	**Runtime (s)**	**Overhead**
unprotected	258538417	1.26	32.32	-
software	500452417	1.37	62.56	94%
SMOV	298857417	1.22	37.36	16%

In Bubblesort, SMOV reached a 22% runtime overhead, while the software-based yielded an overhead of 149%. This represents a runtime overhead improvement of 85% against the software-based variation. In terms of execution time, considering the 8Mhz processor, the software-based ABC variation would run in 2.45min and the SMOV variation would be executed in 1.1min.

Looking at Quicksort, we can see that the software-based ABC variation caused a 92% runtime overhead, while the overhead imposed by SMOV was only 15%. It means that SMOV consumes 40% less cycles, representing 382.090ms of execution time, against 637.027ms of the software-based approach.

For the last program, Perm, SMOV reduced the runtime overhead caused by the software-based ABC variation by 83%. The runtime overhead of the later was 94%, against 16% of SMOV. In terms of execution time, the SMOV variation runs in 37.357s and the software-based approach in 1.04min, considering the hypothetical 8Mhz processor.

Figure 3 provides a visual comparison between the different runtime overheads imposed by all secure approaches. SMOV had its best relative results in the Bubblesort program. This happened because Bubblesort contains a higher number of array accesses. Thus, its implementation requires more bound checks, which is exactly where SMOV thrives.

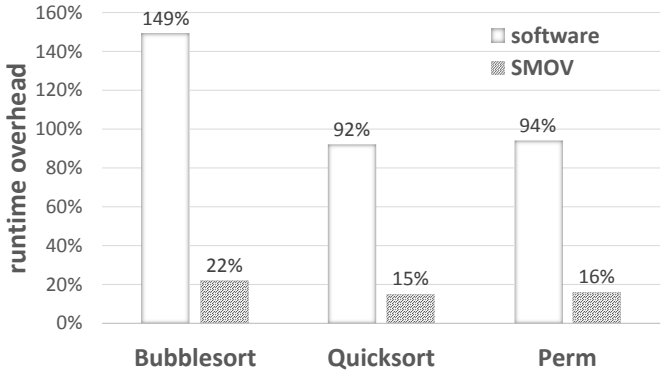

Figure 3. Runtime overhead caused by different security approaches.

In all case studies, the software-based solution has caused an increase in Cycles per Instructions (CPI). This is due to the "extra" conditional branches which can, sometimes, lead to mispredictions or inherently delays in the pipeline. Our solution, in the contrary, reduced the CPI because memory access instructions are completely parallelizable within the pipeline. Consequently, they do not cause interruptions.

Another comparison parameter that can be inferred from the results is the speed-up obtained by our solution. This gives an idea about how faster the runtime of a program is, when we change from a software-based ABC to SMOV. In fact, SMOV gives us an acceleration of 104%, 66%, and 67% on the runtime for Bubblesort, Quicksort and Perm programs, respectively.

Our results tell us that, on average, SMOV has a runtime overhead of 18%, reducing this factor from the software-based approach by 83%. The average speedup found indicates that our solution can run programs 1,79x faster than when the ABCs are implemented in software.

C. Hardware Cost

To estimate the hardware cost of our solution we have synthesized both architectures, the original Y86 and SMOV enhanced version. The Verilog code, which can be deployed to an Altera FPGA, implements our hardware project using only 3.59% more logical elements than the original version, which demonstrates the simplicity of SMOV and its low design impact.

V. Related Work

The literature contains a vast number of techniques to protect computational systems(e.g. [2], [7], [12], [13], [15], [16], [26]–[39]). In this section we shall focus on the most prominent ones and those which directly relate to our work, hardware-based protection against BOFs.

As mentioned before, hardware-based approaches (e.g., [13]–[16]) 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 [16] (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 `NOP`s. 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 [15] 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 propose registers for bounds-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 [13] and Watchdog [14]. 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 V 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 V. 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%

VI. Conclusion

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 which 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. We discuss how Intel's MPX could incorporate SMOV into its bound-checking suite of instructions. We believe that SMOV is a step towards delivering users secure, reliable, and, ultimately, efficient systems.

Acknowledgment. We thank the anonymous reviewers for their valuable comments and suggestions. This project was financially supported by Intel Corporation, CNPq-Brazil, and FAPEMIG-Brazil.

REFERENCES

[1] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *Proceedings of the Information Survivability Conference and Exposition (DISCEX'00)*, 2000.

[2] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *Proceedings of the Symposium on Network and Distributed System Security (NDSS'00)*, 2000.

[3] K.-S. Lhee and S. J. Chapin, "Buffer overflow and format string overflow vulnerabilities," *Software: Practice and Experience*, 2003.

[4] M. Bishop, S. Engle, D. Howard, and S. Whalen, "A taxonomy of buffer overflow characteristics," *IEEE Transactions on Dependable and Secure Computing*, 2012.

[5] D. Moore, C. Shannon, and k. claffy, "Code-red: a case study on the spread and victims of an internet worm," in *Proceedings of the Workshop on Internet Measurment (IMW'02)*, 2002.

[6] The Heartbleed Bug, 2014. [Online]. Available: http://heartbleed.com/

[7] E. Haugh and M. Bishop, "Testing c programs for buffer overflow vulnerabilities," in *Proceedings of the Symposium on Network and Distributed System Security (NDSS'03)*, 2003.

[8] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: compiler enforced temporal safety for c," in *Proceedings of the International Symposium on Memory Management (ISMM'10)*, 2010.

[9] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID'11)*, 2011.

[10] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to risc," in *Proceedings of the Conference on Computer and Communications Security (CCS'08)*, 2008.

[11] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *Proceedings of the USENIX Security Symposium (USENIX Security'14)*, 2014.

[12] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: a fast address sanity checker," in *Proceedings of the USENIX Annual Technical Conference*, 2012.

[13] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the c programming language," *ACM SIGOPS Operating Systems Review*, 2008.

[14] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *Proceedings of the International Symposium on Computer Architecture (ISCA'12)*, 2012.

[15] ——, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'14)*, 2014.

[16] Intel Corporation, "Intel Architecture Instruction Set Extensions Programming Reference," https://software.intel.com/en-us/isa-extensions, 2013.

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, 2011.

[18] F. Bellard, "Qemu, a fast and portable dynamic translator." in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005.

[19] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The m5 simulator: Modeling networked systems," *IEEE Micro*, 2006.

[20] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS'07)*, 2007.

[21] D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2008.

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

[23] I. Parulkar, A. Wood, J. C. Hoe, B. Falsafi, S. V. Adve, J. Torrellas, and S. Mitra, "Opensparc: An open platform for hardware reliability experimentation," in *Proceedings of the Workshop on Silicon Errors in Logic-System Effects (SELSE'08)*, 2008.

[24] Zet Processor, 2014. [Online]. Available: http://zet.aluzina.org/index.php/Zet_processor

[25] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the Code Generation and Optimization (CGO'04)*, 2004.

[26] R. W. Jones and P. H. Kelly, "Backwards-compatible bounds checking for arrays and pointers in c programs," in *Proceedings of the Symposium on Automated and Analysis-Driven Debugging (AADEBUG'97)*, 1997.

[27] D. Dhurjati, S. Kowshik, and V. Adve, "SAFECode: enforcing alias analysis for weakly typed languages," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'06)*, 2006.

[28] D. Wagner and R. Dean, "Intrusion detection via static analysis," in *Proceedings of the Symposium on Security and Privacy (S&P'01)*, 2001.

[29] G. J. Holzmann, "Static source code checking for user-defined properties," in *Proceedings of the Integrated Design and Process Technology (IDPT'02)*, 2002.

[30] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The ASTRÉE analyzer," in *Proceedings of the European Symposium on Programming (ESOP'05)*, 2005.

[31] J. Viega, J.-T. Bloch, Y. Kohno, and G. McGraw, "ITS4: A static vulnerability scanner for c and c++ code," in *Proceedings of the Computer Security Applications (ACSAC'00)*, 2000.

[32] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang *et al.*, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the USENIX Security Symposium (USENIX Security'98)*, 1998.

[33] T. Bell, "The concept of dynamic analysis," *ACM SIGSOFT Software Engineering Notes*, 1999.

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

[35] K. Piromsopa and R. J. Enbody, "Secure bit: Transparent, hardware buffer-overflow protection," *IEEE Transactions on Dependable and Secure Computing*, 2006.

[36] A. Francillon, D. Perito, and C. Castelluccia, "Defending embedded systems against control flow attacks," in *Proceedings of the Workshop on Secure Execution of Untrusted Code (SecuCode'09)*, 2009.

[37] J. P. McGregor, D. K. Karig, Z. Shi, and R. B. Lee, "A processor architecture defense against buffer overflow attacks," in *Proceedings of the International Conference on Information Technology: Research and Education (ITRE'03)*, 2003.

[38] H. Nazaré, I. Maffra, W. Santos, L. B. Oliveira, L. Gonnord, and F. M. Quintão Pereira, "Validation of memory accesses through symbolic analyses," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*, 2014.

[39] F. A. Teixeira, G. V. Machado, F. M. Pereira, H. C. Wong, J. Nogueira, and L. B. Oliveira, "Siot: securing the internet of things through distributed system analysis," in *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN'15)*, 2015.