A Run-Time Reconfigurable Architecture for Embedded Program Flow Verification

Joseph ZAMBRENO, Tanathil ANISH and Alok CHOUDHARY [1]
Department of Electrical and Computer Engineering, Northwestern University

Abstract. Poorly written software can pose a serious security risk. Applications designed for embedded processors are especially vulnerable, as they tend to be written in lower-level languages for which security features such as runtime array bounds checking are typically not included. The problem is exacerbated by the fact that these potentially insecure embedded applications are widely deployed in a variety of high-risk systems such as medical devices, military equipment, and aerospace systems. These observations motivate additional research into embedded software security. In this paper, we present a compiler module and reconfigurable architecture for verifying the integrity of embedded programs. Our architecture prevents several classes of program flow attacks, as opposed to many current approaches which tend to address very specific software vulnerabilities. We demonstrate the correctness and feasibility of our approach with an FPGA-based prototype implementation that is effective in protecting applications with minimal performance overhead.

Keywords. Security, software protection, buffer overflows, reconfigurable architectures

Introduction

Embedded applications are typically not the main focus of secure solutions providers, as the personal and business computing world has been the traditional target for wide-scale attacks. This is a cause for concern for two reasons. Firstly, embedded software tends to be written in low-level languages, especially when meeting real-time constraints is a concern. These languages tend to not have facilities for runtime maintenance which can often cover programming errors leading to security holes. Secondly, embedded systems are used in a variety of high-risk situations, for which a malicious attack could have devastating consequences.

Many current approaches merely apply stopgap measures - either by patching specific vulnerabilities or disallowing behavior that is representative of the attacks considered. While effective at preventing the most popular exploits, these approaches are not able to handle an unexpected weakness or an unknown class of attacks. Clearly, additional research is needed to find a more general solution.

[1]Correspondence to: Alok Choudhary, Department of Electrical and Computer Engineering, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208 USA; Tel.: +1 847 467 4129; Fax: +1 847 467 4144; E-mail: choudhar@ece.northwestern.edu.

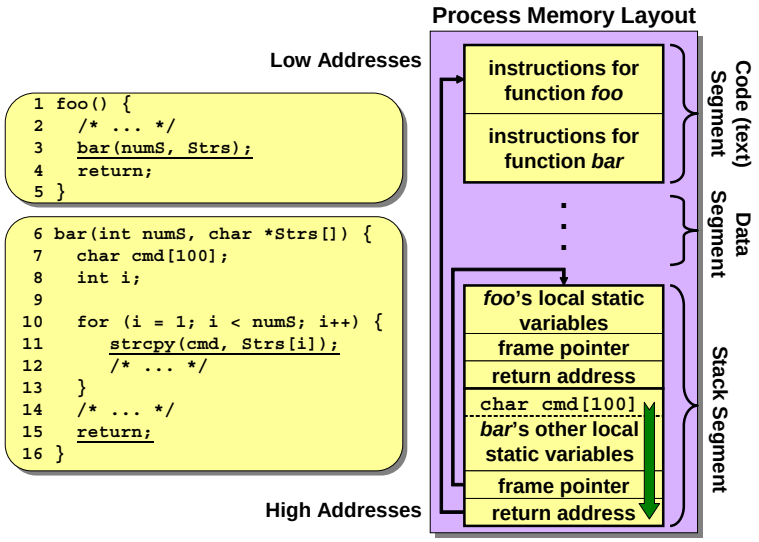

Figure 1. Process memory layout of an application with a potentially exploitable buffer overflow.

In this paper, we present an architecture and compiler module that can prevent a more general class of attacks. The input program is first analyzed by our compiler to generate program flow metadata. When the application is loaded onto the processor, this data is then stored in the custom memories on our architecture. Since this approach requires only a passive monitoring of the instruction fetch path, we are able to continually verify the program's execution with a minimal performance penalty.

To demonstrate the correctness of our architecture, we implemented a prototype on a Xilinx ML310 FPGA development board. The results presented here detail the performance of the protected applications given various architectural configurations. The feasibility of the total approach is also considered, with experimental results on both the area consumption of our architecture and the increase in application size due to the inclusion of the program flow metadata.

The remainder of this paper is organized as follows. We start with an explanation of unchecked buffer vulnerabilities alongside their commonly-found exploits. In Section 2, we discuss several of the current approaches to addressing this problem. We then present our proposed solution in Section 3, followed by a description of the prototype we developed on the FPGA board. Finally, we conclude the paper in Section 5 with a brief overview of future efforts that are planned for this project.

1. Vulnerabilities and Attacks

Consider the simple application depicted in Figure 1. In this C-style pseudo-code, function `bar` is called with an array of pointers to character arrays as its input. These input arrays are copied to a local array using the standard C library `strcpy` call before further processing is applied.

In this code example, since the `strcpy` call does not check to ensure that there is sufficient space in the destination array to fit the contents of the source string, it is

possible to write enough data to "overflow" the local array. In this example's process memory layout, the local variables for function `bar` are placed in the stack segment, just on top of run-time variables that are used to ensure proper program flow. The common convention is for the stack to grow backwards in memory, with the top of the stack being placed at lower physical addresses and local arrays growing upwards in memory.

When user input is placed into overflowable buffers, the application is vulnerable to the *stack smashing* attack [1]. Widespread exploits including the Code Red and SQL-Slammer worms have convincingly demonstrated that it is possible for an attacker to insert arbitrary executable code on the stack. The stack smashing attack works as follows:

- The attacker first fills the buffer with machine instructions, often called the *shellcode* since typical exploit demonstrations attempt to open a command-shell on the target machine.
- The remainder of the stack frame is filled until the return address is overwritten with a pointer back to the start of the shellcode region.
- When the vulnerable function attempts to return to its calling parent, it uses the return address that has been stored on the stack. Since this value has been overwritten to point to the start of the attacker-inserted code, this is where the program will continue.

The widespread notoriety of stack smashing exploits have led to a considerable amount of focus on their detection and prevention. In the future, it is likely that more complex attacks involving buffer overflows (see [2] for a description of *arc injection* and *pointer subterfuge*) will gain in popularity. Note that any arbitrary program flow modification can be potentially malicious, not just those involving unprotected buffers.

2. Current Approaches

Several compiler-based solutions currently exist for buffer overflow vulnerabilities. StackGuard [3] is a compiler modification that inserts a unique data value above the return address on the stack. The code is then instrumented such that this value is then checked before returning to the caller function. This check will fail if an overflowing buffer modifies this value. StackShield is a similar compiler extension that complicates the attack by copying the return addresses to a separate stack placed in a different and presumably safer location in memory. While they are effective, it should be noted that these protections can be bypassed in certain situations [4].

Other similar approaches include obfuscation-driven compiler transformations [5], where the goal is to limit code understanding through the deliberate mangling of program structure. In [6] a tamper-proofing approach is proposed where application integrity is asserted through the insertion instructions that perform code checksums during program execution. The modified applications are vulnerable to discovery using tools that can be built to automatically look for obfuscations or checksum instructions. Accordingly, these protections are only able to delay an eventual attack.

Designating memory locations as non-executable using special hardware tags is becoming a popular method for deterring buffer overflow attacks. Although available on a variety of older processors, most recently AMD has released hardware with their NX (No eXecute) technology and Intel has followed suit with a differently named yet functionally

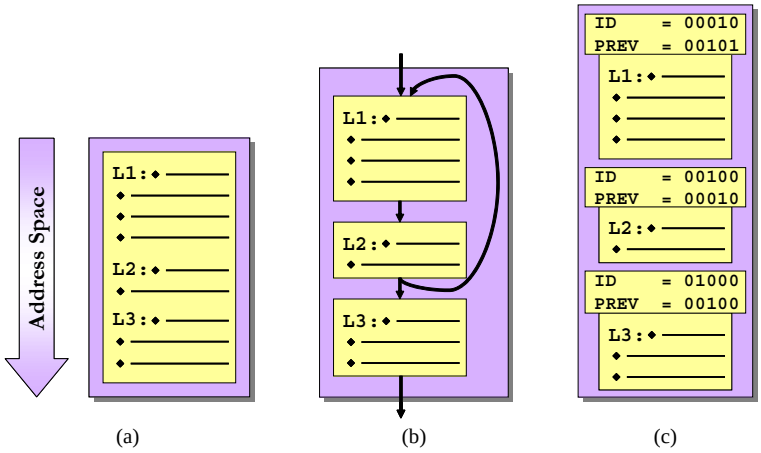

Figure 2. (a) Static view of an application's instructions. (b) Program-flow graph. (c) Runtime execution information added to the application via the PFencode compiler module.

equivalent XD (eXecute Disable) bit. Software emulation of non-executable memory is a less secure option for processors that do not include functionality similar to the NX bit. These approaches are being widely adopted for general-purpose processors, however they do not address any type of program flow attack that doesn't involve instructions being written to and later fetched and executed from data memory.

There have been several hardware-based approaches to protecting software. The authors in [7] utilize their DISE architecture to implement a concept similar to StackShield in hardware. The SPEF framework proposed in [8] provides options for both obfuscating and introducing integrity checks to the input application, which are verified at run-time by custom hardware. The XOM architecture proposed in [9] attempts to ensure that instructions stored in memory cannot be modified, with specialized hardware being used to accelerate cryptographic functionality. The latency overhead introduced by encrypting the instruction fetch path can be considerable and may not be acceptable in real-time embedded systems; this problem is currently being examined by the computer architecture community [10,11]. Industry support for secure processors include the companies in the Trusted Computing Group (TCG) [12], which define the Trusted Platform Module, a hardware component that provides digital signature and key management functionality for software protection and Digital Rights Management (DRM).

3. Our Approach

When considering a static view of an application (Figure 2), a *basic block* is defined as a subsequence of instructions for which there is only one entry point and one exit point. Basic block boundaries are typically found at jumps and branch targets, function entry and exit points, and conditionally executed instructions. The various types of program flow attacks can all be generalized as invalid edges in a flow graph at the basic block granularity:

- Flow passing between non-consecutive instructions in the same basic block.

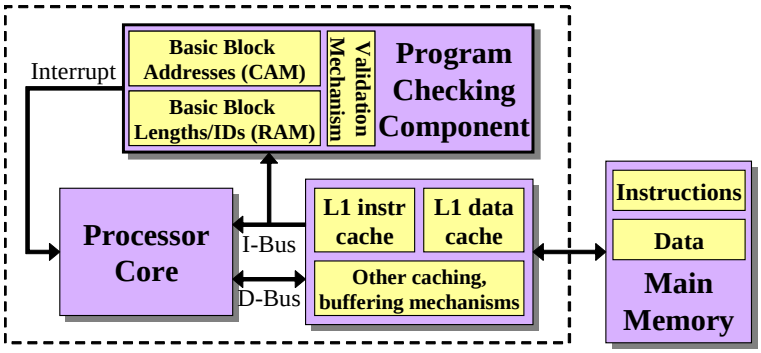

Figure 3. Insertion of the PFcheck component into a standard CPU architecture.

- Flow passing between between two blocks with no originally intended relation.
- Flow passing between any two instructions that are not both at basic block boundaries.
- Flow passing from an instruction back to that same instruction, when that instruction is not both the entry and exit point of a single basic block.

Our approach operates at the basic block level. In the example of Figure 2, flow passes unconditionally from block `L1` into `L2`, where it may either loop back into `L1` or pass conditionally into `L3`. This information is statically knowable by the compiler, which is typically able to break the input program into basic blocks for analysis. It is also possible to include a profiling pass to determine more complex program flows. Our compiler module, which we call PFencode, assigns identification labels to the basic blocks. These "IDs" are given a one-hot encoding, for reasons that will become apparent. For this example, block `L1` is encoded as **00010**, `L2` is encoded as **00100**, and `L3` is encoded as **01000**.

Next, the compiler generates a table of predecessors for each basic block. Since the blocks are already encoded with a unique bit, we can create this "prevID" table by just ORing the IDs or all the blocks that are valid predecessors for each block. As an example, since since block `L2` and the prefix code segment (with an ID of **00001**) are both valid predecessors of `L1`, the prevID value for `L1` is defined as **00101**. The compiler also generates a value representing the length of each basic block.

Figure 3 shows a high level view of how our program flow checking architecture can be inserted into a standard CPU architecture. Our hardware component, which we call PFcheck, uses a snooping mechanism to analyze instructions being fetched directly between the processor core and the lowest level of instruction cache. Since the PFcheck architecture does not delay the instruction fetch path, it is not expected to incur the same negative performance impact as other approaches.

The architecture contains several customized memories which it uses to hold the program flow metadata values. The basic block base addresses are stored in a Content-Addressable Memory (CAM). Typically on a CAM lookup operation, the output is a unencoded value of the lines in the memory that match the target address. When an instruction is being fetched, the PFcheck component sends this address to the CAM. If there is a match, this means that the instruction is an entry point for a basic block - the ID of that block is given as the output of the CAM.

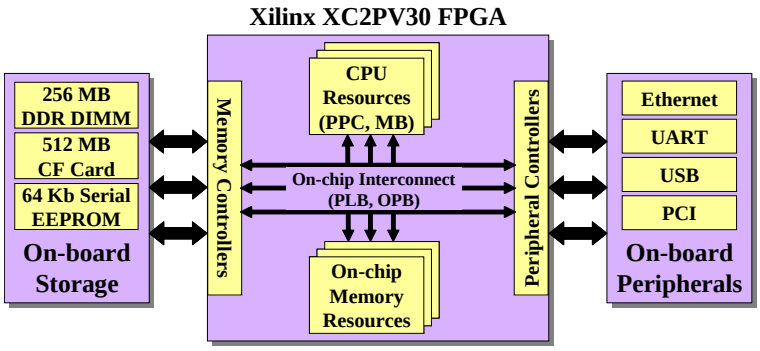

Figure 4. Architectural view of the Xilinx ML310 development platform.

Table 1. PFencode results for a variety of benchmarks

Benchmark	App Size	Num Basic Block	Metadata Size	Percentage Increase
`adpcm`	15.8 kB	75	1.30 kB	8.2%
`dijkstra`	23.3 kB	257	10.3 kB	44.2%
`laplace`	15.6 kB	32	0.38 kB	2.4%
`fir`	22.8 kB	240	9.12 kB	40.0%
`susan`	90.8 kB	1253	206 kB	226.9%

The basic block lengths and prevID table are stored in a standard RAM. The only output required by the component is an interrupt signal, which is used to notify the system that an invalid flow has been detected. A more detailed description of a PFcheck implementation is given in the following section.

4. Implementation and Results

We implemented our prototype system on a Xilinx ML310 FPGA development board (Figure 4), using version 7.1 of their Embedded Development Kit (EDK) for the design entry, simulation, and synthesis. The ML310 contains a XC2VP30 FPGA that has two PPC 405 processors, 13696 reconfigurable CLB slices, and 136 Block SelectRAM modules. The board and FPGA can act as a fully-fledged PC, with 256 MB on-board memory, some solid-state storage, and several standard peripherals.

We implemented our PFencode compiler as a post-pass of the GCC compiler targeting the Xilinx MicroBlaze soft processor [13]. We chose the MicroBlaze over the PPC for its simplicity and flexibility. Table 1 shows the results when we ran PFencode on a number of benchmarks customized to support MicroBlaze software libraries. For these relatively small applications, we found that the size of the metadata increased exponentially with the number of basic blocks. This is due to the fact that for an application with N basic blocks, our one-hot encoding approach requires an $N \times N$ memory to hold the entire prevID table. For the benchmarks that have relatively short basic blocks, this increase in application size can be drastic (see `susan` which is 5.8 times the size of `laplace` but requires 542 times the amount of metadata).

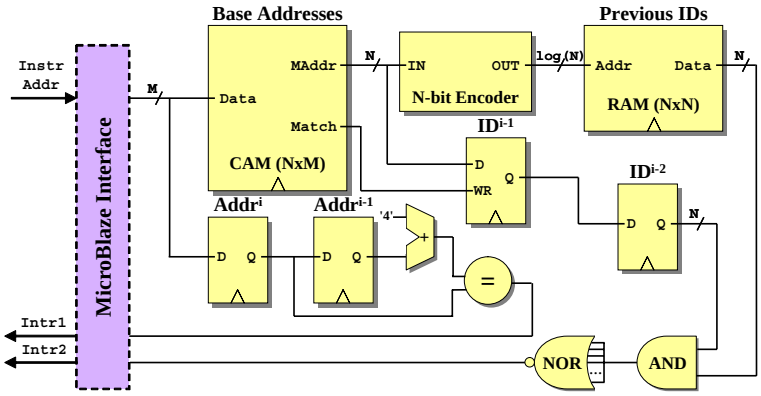

Figure 5. Internals of an implementation of the PFcheck architecture.

Table 2. PFcheck implementation results for a Xilinx XC2VP30 FPGA

Config ($N \times M$)	64x16	64x32	128x32	256x32	512x32
Num Slices	450 (3.2%)	510 (3.7%)	840 (6.1%)	1615 (11.8%)	3031 (22.1%)
Num BRAMs	5 (3.7%)	9 (6.6%)	18 (13.2%)	36 (26.5%)	80 (58.9%)
Clock Freq	71.3 MHz	71.3 MHz	64.9 MHz	56.0 MHz	59.32 MHz

Figure 5 shows the internals of our implementation of the PFcheck architecture. The instruction addresses (M bits wide) is sent into the CAM module which has M-bit wide entries for each of the N basic blocks. If there is a match, that means that this instruction address is a basic block base address - the one-hot ID value for that basic block is the output of the CAM on a match. Otherwise, the output of the CAM is all zeros indicating that the current instruction is inside a basic block. The output of the CAM is sent to an encoder that creates a $log(N)$ wide signal that drives the address of the RAM module to get the prevID value for the current block. The entry in the prevID table is checked with the previously fetched ID that is stored in a register. If the bits do not match up, an interrupt signal is sent to the processor. In the case where the current instruction is not a basic block entry point, an interrupt will be also be generated if the current instruction does not directly proceed the previous instruction. This is equivalent to checking that PC does not equal $PC + 4$.

Table 2 shows how the area consumption and performance of the PFcheck architecture is depends on the configuration. For our experiments, we synthesized designs with varying amounts of basic block support and address bus width. Each design is labeled $N \times M$. Several trends are readily apparent. The BlockRAM and slice usage increase linearly with N. Decreasing the address bus width has less of an impact on area usage, and no impact on clock frequency. The limiting factor for this medium-sized FPGA was the number of BlockRAMs; we were not able to fit the `1024x32` on the device.

To test for correctness, we used PFencode to compile an application that contained a buffer overflow vulnerability. We attempted to write attack code on the stack and to overwrite the return address to point back to this buffer. While the architecture did not protect values on the stack from being overwritten, when the program attempted to return to this malicious region an interrupt was triggered.

5. Conclusions and Future Work

In this paper we proposed and evaluated a reconfigurable architecture for verifying program flow integrity. We demonstrated the effectiveness of our approach with a implementation protecting a MicroBlaze processor running on a Xilinx ML310 FPGA board.

Further improvements are being made to the PFcheck architecture. We are currently analyzing the performance overhead of writing the program flow metadata values to the CAM and RAM. Also, we are investigating a paging architecture that will be able to protect larger applications (in terms of number of basic blocks) without a loss in the amount of security provided.

Acknowledgements

This work was supported in part by the National Science Foundation (NSF) under grant CCR-0325207 and also by an NSF graduate research fellowship.

References

[1] 'Aleph One'. Smashing the stack for fun and profit. In *Phrack*, vol. 7, no. 49, Nov. 1996.

[2] J. Pincus and B. Baker. Beyond stack smashing: recent advances in exploiting buffer overruns. In *IEEE Security and Privacy*, vol. 2, no. 4, Jul. 2004, pp. 20–27.

[3] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, Jan. 1998, pp. 63–78.

[4] 'Bulba' and 'Kil3r'. Bypassing StackGuard and StackShield. In *Phrack*, vol. 10, no. 56, May 2000.

[5] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Proceedings of the International Conference on Computer Languages (ICCL)*, May 1998, pp. 28–38.

[6] H. Chang and M. Atallah. Protecting software code by guards. In *Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management*, Nov. 2001, pp. 160–175.

[7] M. Corliss, E. Lewis, and A. Roth. Using DISE to protect return addresses from attack. In *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, Oct. 2004.

[8] D. Kirovski, M. Drinic, and M. Potkonjak. Enabling trusted software integrity. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, Oct. 2002, pp. 108–120.

[9] D. Lie, C. Thekkath, M. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Nov. 2000, pp. 168–177.

[10] W. Shi, H-H. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High efficiency counter mode security architecture via prediction and precomputation. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, Jun. 2005, pp. 14–24.

[11] J. Yang, Y. Zhang, and L. Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, Dec. 2003, pp. 351–360.

[12] Trusted Computing Group, http://www.trustedcomputing.org, 2005.

[13] Xilinx. *MicroBlaze Processor Reference Guide*, available at http://www.xilinx.com, 2005.