Imperial College London

Telecom Paristech

Hardware-based Buffer Overflow Protection

Submitted for part completion of MEng Electronic and Information Engineering

Author: Jake Humphrey
Supervisor: Guillaume Duc

Abstract

Implementing security features in hardware can free the software programmer from worrying about implementing such security, leading in turn to simplified code. This can be especially useful when programming in low-level languages such as C, which provide minimal built-in protection.

The work detailed in this project report modifies the Memory Management Unit (MMU) of an existing processor to implement protection against buffer overflow attacks, then produces some assembly code showing how firmware or an operating system kernel would make use of the protection. Finally, it analyses the effect of the added protection on hardware area and clock frequency.

Contents

Chapter 1

Introduction

Today computers have access to large amounts of our personal data. It can often have disastrous effects on our personal and professional lives if the data is compromised. The problem with leaving security issues to software developers is that often multiple processes will be running on the same system, and a vulnerability in any one of them can potentially leak sensitive data, regardless of how secure the others are.

It is sometimes preferable to place security measures at the lower levels of execution, i.e. the hardware, kernel or operating system (OS). This restricts system security to a single point-of-failure, and can result in easier, faster, and more secure code.

1.1 Motivation

A **Buffer Overflow Attack** makes use of out-of-range array accesses to read or write memory locations intended to be inaccessible. They are typically associated with C and C++, which provide no built-in protection against array accesses falling outside the bounds of the array. Using a Buffer Overflow Attack, an attacker could, for example, corrupt the return address inside a function to point to and thus run arbitrary code.

The Heartbleed bug in OpenSSL[1] was an example of a vulnerability to this type of attack. An attacker was able to send a server a payload text string along with its length and have the server reply with the exact same payload. However, if the length supplied was longer than the actual length of the payload, the returned message would be appended with whatever followed the text string in memory.[1] This potentially allowed an attacker access to server private keys, user passwords, or really anything stored on the server's memory.

1.2 Existing Solutions

A potential solution would be to use other programming languages or the Standard Template Library container classes in C++, but for those who cannot afford the performance

[1] `http://xkcd.com/1354/`

cut, there exist other protection strategies. For example, the use of **canary values** involves placing a known value after the buffer. Since buffer overflows are often carried out with such C functions as `strcpy`, they affect contiguous memory areas. A write would therefore alter the canary value, and thus the program can infer that a buffer overflow has occurred and can, for example, invalidate the memory area as a countermeasure.

The above solution works for writes, which corrupt the canary value, but would offer no protection against out-of-bounds reads, such as those making use of the Heartbleed vulnerability. Fortunately, the same idea can be extended to protect against both types of attack, though at significant cost, as will be explained.

1.3 Project Aims

One could imagine a system making use of the Memory Management Unit, whereby a buffer would be placed immediately before a page boundary, and the following page marked as invalid in the MMU. A buffer overflow, whether a read or a write, would then trigger a page fault, which could potentially allow the OS to resolve the situation.

The biggest downside to this method is that it requires an entire virtual memory page be invalidated. On x86 processors, for example, this is 4KiB. It would be nice to have a smaller invalidatable section of memory in the MMU to use for this purpose.

This project investigates a method to allow the invalidation of smaller memory sections within the the MMU, specifically by adding hardware registers to allow for memory invalidation at the cache block level; in x86 the cache block size is 64B. This allows use of the above protection system without excess wasted memory, however it adds extra silicon in the form of the hardware registers. This tradeoff is also examined during the project.

Chapter 2

Research

To carry out the project aims, a **Soft Processor**, a processor which is to some extent configurable, is needed. Specifically we need to be able to directly modify the MMU code. This chapter provides a comparison of the various Soft Processors available.

2.1 Comparison of Soft Processors

The following metrics will be taken into account:

- Bus Used
- Architecture (MIPS/ ARM)
- Cache
- MMU (Memory Management Unit)
- Protection/User Levels
- License
- Documentation
- Toolchain/Debugger/Monitor/JTAG
- Maturity

2.1.1 OpenRISC

Developer OpenCores

Bus Used OpenRISC uses Wishbone, an open source specification by OpenCores.

Architecture 32/64-bit Mips-like.

Cache 1–64kB for Instructions and Data each.

MMU Virtual Memory support. TLB (Table Lookaside Buffer) with page size 8kB and 16–256 entries.

User Levels User/Supervisor Mode.

License There are two main open source implementations available. A Verilog implementation, OpenRISC 1200, is released under LGPL. Another Verilog implementation, mor1kx, is available under a weak copy-left licence created specifically for it, the OHDL.

Documentation The specification can be found here. Documentation for both implementations are also available at their respective Github pages: Mor1kx, OR1200.

Toolchain A Compiler (based on gcc) and Debugger (based on gdb) are available. See here.

Maturity The OpenRISC 1000 specification has been under development since 2001. Version 1.0 was released in 2012, and Version 1.1 in 2014. The OpenRISC 1200 implementation is stable but not in active development, and is the widely used version according to the OpenRISC project overview. Mor1kx is still in active development, but is stated to be more sophisticated.

2.1.2 LEON2/3/4

Developer Aeroflex Gaisler

Bus Used AMBA2

Architecture 32-bit. ISA based on SPARC-V8

Cache LEON2: Instruction and Data caches. Set-associative with 1–4 sets and 1–64kB/set. LRR or LRU replacement. LEON3/4: 1–4 sets, 1–256 kB/set. Random, LRR or LRU replacement. LEON4 includes an optional L2 cache. 256-bit internal, 1-4 sets, 16kB–8MB.

MMU SPARC Reference MMU (SRMMU) with configurable TLB

User Levels The User and Supervisor modes of SPARC.

License Gaisler state on their website that LEON2 is available from Atmel (AT697 and AT7913), but I found what appears to be the VHDL source code on Github. LEON3 is available under GPL as part of the GRLIB library. LEON4 is only available under a commercial license from Gaisler.

Documentation LEON3 and LEON4 have documentation.

Toolchain Being SPARC V8 conformant, compilers and kernels for SPARC V8 can be used with LEON processors. Cross compiler BCC. gdb can be used for debugging.

Maturity No information found.

2.1.3 Lm32

Developer Lattice

Bus Used Wishbone

Architecture 32-bit. ISA here.

Cache Configurable between no cache, 8kB I-cache, and 8kB I-cache and 8kB D-cache.

MMU None.

User Levels No separate modes.

License LatticeMico32 is licensed under a free (IP) core license.

Documentation Documentation is available here, but registration is required.

Toolchain GCC supports the LM32 since version 4.5.0, and Binutils since 2.19. gdb and Newlib are also supported.

Maturity Introduced in 2006.

2.1.4 Microblaze

Developer Xilinx

Bus Used Customisable between CoreConnect PLB, OPB, FSL, LMB, AXI4.

Architecture In terms of its instruction set architecture, MicroBlaze is very similar to the RISC-based DLX architecture, which in turn is based upon the MIPS architecture.

Cache 2kB–64kB Direct mapped write-through or write-back.

MMU Optional. 4GB virtual memory, supports page sizes of 1kB, 4kB, 16kB, 64kB, 256kB, 1MB, 4MB, and 16MB. Page-replacement strategy controlled by software.

User Levels User and privileged modes.

License Proprietary. Available from Xilinx along with the EDK (Embedded Development Kit). A smaller, less functional version with limited target devices is available for free, but the full version costs $3000–$6000, depending on the license type. The license also restricts the use of Microblaze to Xilinx FPGAs. LGPL clones also exist in Verilog, VHDL, or MyHDL. See here.

Documentation Rather comprehensive reference manual here.

Toolchain Xilinx provides the Vivado design suite, including an IDE, SDK, Simulator, Analyser, and Synthesiser. Support for Microblaze also exists in gcc as of version 4.6.

Maturity Currently in version 8.50b, under development since 2002.

2.1.5 Nios II

Developer Altera

Bus Used Altera's own Avalon bus.

Architecture 32-bit RISC architecture, with support for up to 256 custom instructions. 3 different configurations are available: fast (f), standard (s), and economy (e).

Cache Nios II/f has separate instruction and data caches of 512B to 64kB, while the Nios II/s has only an instruction cache, and the Nios II/e has none.

MMU There is an optional MMU and MPU for the Nios II/f. Default is 16-way set-associative cache, the default size depends on the target FPGA but is either 128 or 256 entries or 4 or 8kB. Replacement strategy is set by the software.

User Levels The Nios II supports user and supervisor modes only when the design includes an MMU or MPU.

License The License for the Nios II IP core is around $500. Currently the Altera website has a redirect loop and I am unable to find information on toolchain licenses.

Documentation Both Hardware and Software development documentation can be found here.

Toolchain Qsys and Quartus for hardware, Altera's Embedded Design Suite for software, based on Eclipse and the GNU toolchain.

Maturity In development since 2004, currently in version 13.1, released in 2013.

2.1.6 Cortex-M1

Developer ARM

Bus Used ARM Corelink Interconnect

Architecture Three-stage 32-bit RISC, ARMv6M ISA.

Cache No cache, TCM only, up to 1MB each.

MMU None.

User Levels I don't think it has different levels.

License Proprietary, must contact sales team for cost, I think.

Documentation No reference manual specifically for the M1. See the "resources" tab on this page for more information.

Toolchain An extension of Altera's Quartus II. Supported by the ARM microprocessor development kit for software development.

Maturity Couldn't find any information.

2.2 Choice and Justification

As well as having to contain an MMU, another requirement for the project is that the processor have open code for it. Just having a customisable size is not enough, since the aims include adding entirely new registers and logic to the MMU.

These restrictions eliminate all the above save OpenRISC and the LEON2/3 processors. The OpenRISC was chosen for the project.

Chapter 3

Implementation

This chapter gives an account of the work done during the project. Initially an Altera SoCKit was used as the target device. However, even after extensive modifications, a working build was not obtained. The details of the attempt will nonetheless be detailed both as a record of work done and to advise future attempts.

Following this attempt, an Altera DE2 board was used. This target device proved successful, and the hardware modifications and code written that are detailed in the following sections were run on this board (although in theory, the code is device-agnostic).

3.1 SoCKit

The initial project aim was to use an Altera SoCKit Field Programmable Gate Array (FPGA) development board[2] to instantiate and test the hardware. This section details the (ultimately unsuccessful) attempt to get it working.

3.1.1 Building

The first step was to build the OpenRISC processor and program it onto the board. For this a tool called **FuseSoC**[3] (formerly **OrpSoCv3**) was used. FuseSoC's readme describes it as a "package manager and a set of build tools for HDL [Hardware Description Language] code." Using this, it is simple[1] to build **mor1kx** (the OpenRISC implementation in question) for the SoCKit with the command `fusesoc build sockit`. This command invokes the relevant Quartus executables to build a netlist to be programmed.

However, the build did not work at first. A lot of searching revealed the problem to be a bug in FuseSoC's libraries. As shown below, the port names between two files did not agree. Patching this error allowed the build to continue.[2]

Figure 3.1 gives the declaration of the `avalon_to_wb_bridge` module. If this is compared to an instantiation of that module, such as the one given in Figure 3.2, it can be seen

[1] In theory.

[2] Unfortunately for me, by the time I thought to submit a bug report or a pull request upstream, the bug had alread been found and patched by the developers.

```
module avalon_to_wb_bridge \#(
        parameter DW = 32,      // Data width
        parameter AW = 32       // Address width
)(
        input                 wb_clk_i,
        input                 wb_rst_i,
        // Avalon Slave input
        input [AW-1:0]        s_av_address_i,
        input [DW/8-1:0]      s_av_byteenable_i,
        input                 s_av_read_i,
        output [DW-1:0]       s_av_readdata_o,
        input [7:0]           s_av_burstcount_i,
        input                 s_av_write_i,
        input [DW-1:0]        s_av_writedata_i,
        output                s_av_waitrequest_o,
        output                s_av_readdatavalid_o,
        // Wishbone Master Output
        output [AW-1:0]       wbm_adr_o,
        output [DW-1:0]       wbm_dat_o,
        output [DW/8-1:0]     wbm_sel_o,
        output                wbm_we_o,
        output                wbm_cyc_o,
        output                wbm_stb_o,
        output [2:0]          wbm_cti_o,
        output [1:0]          wbm_bte_o,
        input [DW-1:0]        wbm_dat_i,
        input                 wbm_ack_i,
        input                 wbm_err_i,
        input                 wbm_rty_i
);
```

Figure 3.1: The declaration of the avalon_to_wb_bridge module[4]

that several port names are incorrect. This is likely due to the former file being changed without proper care being taken to update port names throughout the code.

Once the build succeeded, it could be programmed onto the board using the command `fusesoc pgm sockit`.

3.1.2 Running

After the processor was built and the board programmed, the next step was to load an object file onto the processor. For this a linux image was used, compiled using the `or1k-elf-gcc` toolchain as detailed on the OpenCores website[6]. The resulting object file was simulated under the toolchain's simulator and found to run as expected.

To debug the running processor, including loading object files, the **Open On-chip Debugger** (**OpenOCD**) was used as detailed, once again, on the OpenCores website[7].

However, running OpenOCD for the SoCKit gave an error regarding mismatched clock speeds. Some searching revealed a discussion board[8] where the error was mentioned,

```
wb_to_avalon_bridge #(
        .DW                     (32),
        .AW                     (32),
        .BURST_SUPPORT          (1)
) hps_ddr3_wb2avl_bridge (
        .wb_clk_i               (wb_clk),
        .wb_rst_i               (wb_rst),
        // Wishbone Slave Input
        .wb_adr_i               (wb_m2s_hps_ddr3_adr),
        .wb_dat_i               (wb_m2s_hps_ddr3_dat),
        .wb_sel_i               (wb_m2s_hps_ddr3_sel),
        .wb_we_i                (wb_m2s_hps_ddr3_we),
        .wb_cyc_i               (wb_m2s_hps_ddr3_cyc),
        .wb_stb_i               (wb_m2s_hps_ddr3_stb),
        .wb_cti_i               (wb_m2s_hps_ddr3_cti),
        .wb_bte_i               (wb_m2s_hps_ddr3_bte),
        .wb_dat_o               (wb_s2m_hps_ddr3_dat),
        .wb_ack_o               (wb_s2m_hps_ddr3_ack),
        .wb_err_o               (wb_s2m_hps_ddr3_err),
        .wb_rty_o               (wb_s2m_hps_ddr3_rty),
        // Avalon Master Output
        .m_av_address_o         (avm_hps_ddr3_address),
        .m_av_byteenable_o      (hps_0_f2h_sdram0_data_byteenable),
        .m_av_read_o            (hps_0_f2h_sdram0_data_read),
        .m_av_readdata_i        (hps_0_f2h_sdram0_data_readdata),
        .m_av_burstcount_o      (hps_0_f2h_sdram0_data_burstcount),
        .m_av_write_o           (hps_0_f2h_sdram0_data_write),
        .m_av_writedata_o       (hps_0_f2h_sdram0_data_writedata),
        .m_av_waitrequest_i     (hps_0_f2h_sdram0_data_waitrequest),
        .m_av_readdatavalid_i   (hps_0_f2h_sdram0_data_readdatavalid)
);
```

Figure 3.2: An example attempt to instantiate the avalon_to_wb_bridge module[5]

which pointed to a patch for the issue[9].

Compiling a version of OpenOCD with this patch applied seemed to get around the issue, but unfortunately it would still not run.[3] After a meeting between the Author and Supervisor, it was decided to scrap the SoCKit route and work with the DE2 board, which had been used succesfully in a previous project.

[3] At time of writing, I've long since forgotten the exact issue, if indeed I ever worked out what it was. Let this be a lesson to write this kind of thing down somewhere.

3.2 DE2

3.2.1 Building

3.2.2 Debugging

3.2.3 Testing

3.3 Hardware

3.4 Software

Chapter 4

Results

4.1 Hardware Comparison

4.2 Conclusion

4.3 Future Work

Bibliography

[1] Heartbleed Website
`heartbleed.com`
Fetched 2015-06-15.

[2] SoCKit page on Distributor site
`parts.arrow.com/item/detail/arrow-development-tools/sockit`
Fetched 2015-06-16.

[3] FuseSoC repository on Github
`github.com/olofk/fusesoc`
Fetched 2015-06-16.

[4] OrpSoC-cores repository on Github
`github.com/openrisc/orpsoc-cores/blob/14fb5aa0d96aea7e2594d0bb42195c9477dd33f8/cores/wb_avalon_bridge/verilog/avalon_to_wb_bridge.v`
Fetched 2015-06-16.

[5] OrpSoC-cores repository on Github
`https://github.com/openrisc/orpsoc-cores/blob/70b775e2ec2e53d15e3f595336e05053ba81f139/systems/sockit/rtl/verilog/orpsoc_top.v#L587`
Fetched 2015-06-16.

[6] Linux info on the OpenCores website
`opencores.org/or1k/Linux`
Fetched 2015-06-16.

[7] Debugging info on the OpenCores website
`opencores.org/or1k/Debugging_physical_targets_(FPGA/ASIC)`
Fetched 2015-06-16.

[8] Discussion of bug with OpenOCD and the SoCKit
`comments.gmane.org/gmane.comp.debugging.openocd.devel/26474`
Fetched 2015-06-16.

[9] OpenOCD patch on Github
`github.com/fjullien/openOCD/commit/0c834568e020ed3b61a055086d9d142f03860e0f`
Fetched 2015-06-16.