TurtleMCU: A Platform for Teaching a Holistic View Into Embedded Computer Architecture and Security

Nathaniel Graff
College of Engineering
California Polytechnic State University San Luis Obispo
Email: nathaniel.graff@gmail.com

***Abstract*—This Innovative Practice Full Paper introduces an educational tool that combines a simple HDL processor design with a suite of simple, exploitable programs to teach hardware security concepts to students in introductory computer architecture. Recent high profile cyber-attacks that rely on vulnerabilities in computing hardware (including Spectre and Meltdown) have highlighted the need for hardware and software security principles to be taught at all levels of computer architecture and hardware design. Unfortunately, there are currently few tools available that make hardware security concepts accessible to introductory computer architecture students. To address this gap, We propose a new platform called TurtleMCU.**

TurtleMCU provides introductory students with a holistic view into the security of embedded microcontrollers, allowing them to combine reverse engineering, vulnerability discovery and exploitation, and vulnerability response and mitigation techniques at both the hardware and software level. Notably, where other environments for teaching CPU architecture or software security provide only software simulations, TurtleMCU incorporates a fully synthesizeable HDL view of a basic microprocessor, allowing students to implement the design on FPGA development platforms and incorporate their designs into real hardware and peripherals. This paper discusses the design and proposed educational use cases of TurtleMCU, and provides the entire TurtleMCU platform as an open-source project for inclusion in readers' curricula.

1. Introduction

The security of modern computing systems depends on the ability of the underlying hardware to make guarantees of memory access protection, inter-process isolation, and other forms of malicious execution prevention. Where these facilities fail, we see the rise of critical security vulnerabilities like Meltdown and Spectre, in which CPU execution pipelines failed to enforce inter-process isolation guarantees [1], [2]. What's more, the trend of increasing virtualization and multi-tenancy in our computing applications makes the impact of these isolation-breaking vulnerabilities destabilizing to an ever-growing fraction of the world's computing infrastructure.

As these attacks grow in prevalence and we see the massive cost associated with their mitigation, it has become more and more crucial that engineers understand and engage with security principles from the beginning of their education in the design of computer systems. Unfortunately, computing students are often not exposed to security concepts until well after they have started their education. Educational tools are available for teaching many of these concepts, but they fall short at providing insight into the security concerns of computer hardware design.

TurtleMCU provides students with hands-on engagement with the security of embedded microcontrollers and allows them to combine reverse engineering, vulnerability discovery and exploitation, and vulnerability response and mitigation techniques at both the hardware and software level. This hands-on approach, which allows students to design and build custom solutions to security problems, is a crucial part of engineering education [3]. The implementation of secure processor platforms is no exception. TurtleMCU provides an environment for students to discover these principles as well as a platform for students to design their own solutions to these problems.

The TurtleMCU project consists of a specification of a custom instruction set architecture and assembly language, a Verilog implementation of the microprocessor, an assembler for the processor written in Python, and an NCurses-based simulated debugger for the processor. The project, including all source code and pre-written exercises, is published under the GPLv3 Open Source License, enabling free development and redistribution of the project for all academic purposes. Section 3 contains a description of the architecture and assembly language of TurtleMCU, section 4 describes the simulation tools developed for the platform, section 5 provides an overview of the educational applications for teaching security concepts with TurtleMCU, and section 6 discusses the pedagogical application and future work of the TurtleMCU platform.

2. Prior Work

TurtleMCU is certainly not the first educational processor platform, but builds on a long history of work on architectures like MIPS (Patterson and Hennessy, Harris and Harris), and RAT [4], [5], [6]. These solutions largely exist

978-1-5386-1174-6/18/$31.00 ©2018 IEEE

to teach students how a processor can work, but don't expose any of the security considerations. In fact, many processor extensions that exist in commercial designs to improve security are stripped from these academic implementations for the sake of simplicity.

In contrast, tools for education in software security are available in relative abundance. Software and network security education has greatly benefited from a plethora of educational tools and platforms including board games, capture-the-flag challenges, puzzle-based learning, and alternate reality games [7], [8]. These educational security games have been found to be effective at reaching a broad audience of undergraduate computer science students and engaging them in discussions and critical thinking related adversarial thinking, professional ethics, and more [8]. TurtleMCU is modeled after these examples of gamified learning to enhance architecture security education. To do so, it provides a processor architecture model that is simple enough for an introductory architecture student to understand and includes the necessary assemblers and simulation tools to allow students to explore the security ramifications of their design decisions.

TurtleMCU provides a microprocessor architecture and platform designed specifically to facilitate education in processor security extensions and the security implications of processor architecture design decisions. We seek to show in this paper how TurtleMCU is designed with these concerns in mind and can be immediately adopted into computer engineering curricula to teach security principles.

3. Architecture

TurtleMCU is a 16-bit big-endian microprocessor architecture designed to be simple enough to be understood and implemented by an introductory computer architecture student while still providing the flexibility necessary to explore the security implications of an embedded instruction set architecture. A heavily reduced instruction set was chosen to allow for ease of implementation and student comprehension. There are 32 instructions, supporting integer and bitwise arithmetic, branching, function calls, and stack manipulation. The instruction word length is fixed at 16 bits, which allows for each instruction to be fetched and decoded in a single cycle.

The microprocessor implements a Von Neumann architecture, which is distinguished by a single coherent memory space containing both program instructions and data. This is in contrast to Harvard Architecture, which stores instructions and data in separate memory spaces. The initial RAM state is stored within a ROM HDL module, generated by the assembler. The ROM data is copied into the RAM at processor initialization and execution starts at address 0x000.

3.1. Security Implications of Architecture Decisions

The choice of Von Neumann architecture is critical for teaching computer architecture security. By co-locating the data and instruction memory, as most general-purpose consumer processors do, the Von Neumann architecture also enables arbitrary code execution exploits which leverage untrusted data being treated as code. TurtleMCU enables students to explore both the exploitation of these vulnerabilities in the microprocessor, and the implementation of software and hardware techniques to mitigate these attacks.

TurtleMCU supplies a general purpose input/output (GPIO) bus to provide peripheral extensibility through an I/O port model. Custom peripheral modules providing additional processor features can be connected through the GPIO bus and interfaced with internal processor signals, allowing students to add custom CPU extensions like firmware validation, memory protection, and more. Historically, creating these modules has been made difficult by restrictive and uncustomizable assemblers. To remedy this, the TurtleMCU assembler can be extended to support custom macros and directives.

3.2. Instruction Set and Assembly Language

TurtleMCU supports 32 instructions, including integer and bitwise arithmetic, branching, function calls, and stack manipulation. To develop software using this instruction set, We also present an assembly language, TurtleASM, to allow for the rapid development of programs for TurtleMCU. In addition to the 32 instructions provided by TurtleMCU, TurtleASM natively supports comments, tags, hard-coded strings, and directives for specifying the location of code within the ROM.

The assembler is written in Python 3 using a recursive descent parser from the PyParsing library [9]. This allows for students to not only develop programs for TurtleMCU, but to adapt and extend the assembler as they see fit.

The output of the assembler is a System Verilog module implementing a ROM containing the program data. The ROM makes heavy use of System Verilog macros to display instruction data, allowing students to read the machine code generated by the assembler to see the effect the assembly process has on their code. Code samples 1 and 2 display example assembly and the resulting ROM.

Code Sample 1. *Sample Assembly*

```
.org 0x000

    .ldtag r0, mystringtag
    call puts
    .ldtag r0, myheaptag
    call gets
    call puts

done: jmp done
```

Code Sample 2. *Sample ROM Contents*

```
10'h000: dout = {`OPCODE_MVH, `R0, 8'h01};
10'h001: dout = {`OPCODE_MVL, `R0, 8'h00};
10'h002: dout = {`OPCODE_CALL, 10'h325, 1'b0};
10'h003: dout = {`OPCODE_MVH, `R0, 8'h02};
10'h004: dout = {`OPCODE_MVL, `R0, 8'h00};
10'h005: dout = {`OPCODE_CALL, 10'h311, 1'b0};
10'h006: dout = {`OPCODE_CALL, 10'h325, 1'b0};
10'h007: dout = {`OPCODE_JMP, 10'h007, 1'b0};
```

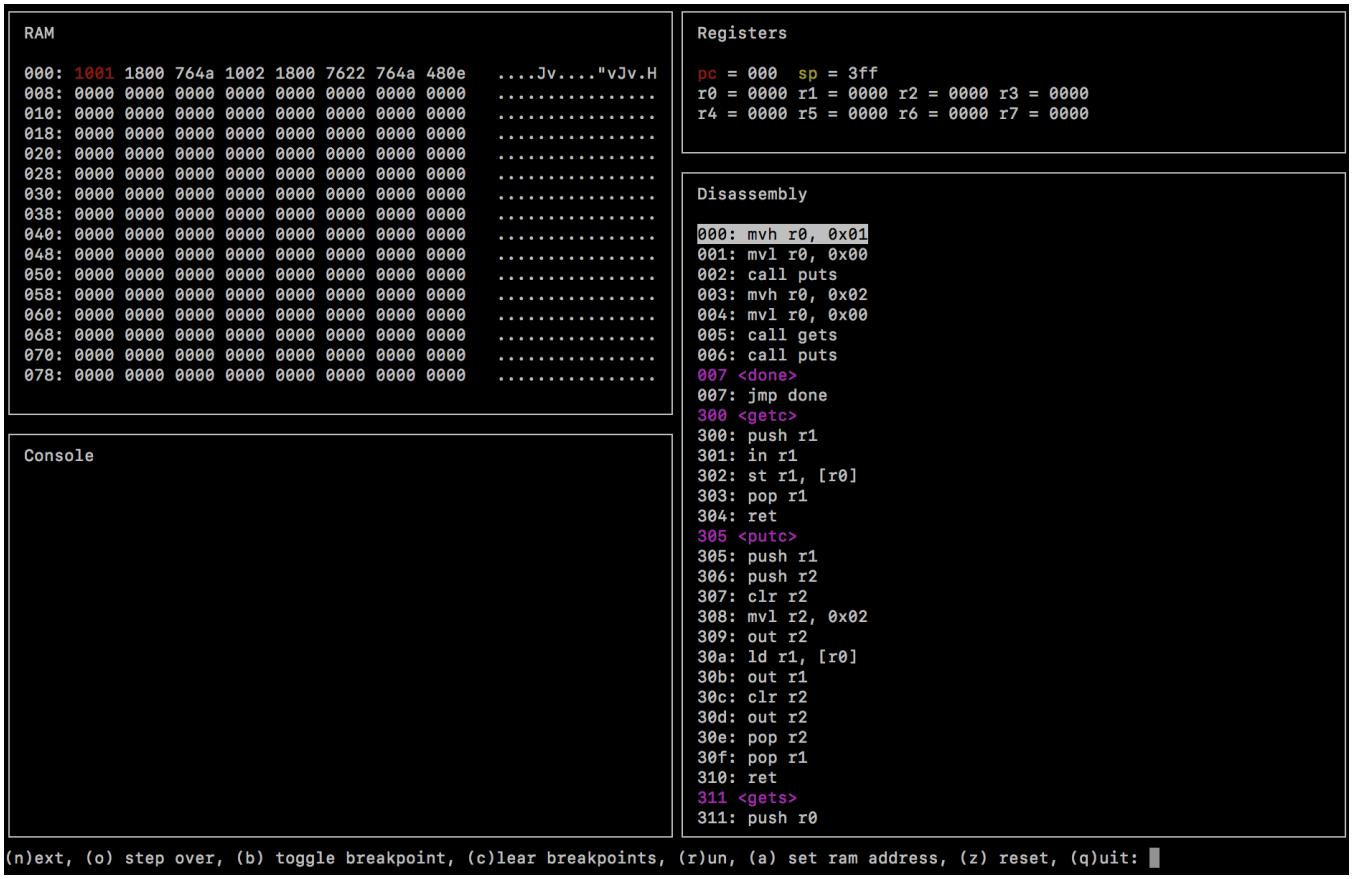

Figure 1. View of debugger interface.

4. Simulation Environment

TurtleMCU was developed to support both simulation and synthesis, and accomplishes the former by leveraging the open source Verilator simulator [10]. A major advantage of Verilator is that it is cross-platform, enabling TurtleMCU to be incorporated into any environment with a C++ build tool chain.

The output of the Verilator simulator is a C++ class which implements the logical behavior of the processor and provides the top block interface as member variables. This allows for a highly portable simulation of the CPU which can be embedded in a large variety of environments.

4.1. Debugger

TurtleMCU embeds the Verilator C++ class to provide a simulated debug environment in an NCurses GUI [11]. The debugger presents a hexadecimal memory dump of the whole RAM, the disassembly of the program being executed, register state, and a console for interacting with the microprocessor over standard I/O. Standard debug features are supported, including breakpoints, step-into, and step-over.

Not only can students simulate and debug their programs in the debugger, but the debug environment provides an interface to a number of pre-written security challenges written for TurtleMCU which teach buffer overflow attacks and reverse engineering of disassembly.

Figure 1 shows a screen-shot of the running debug environment.

5. Applications for Security Education

TurtleMCU is presented with a suite of pre-written security challenges, modeled after the online Capture-the-Flag (CTF) competition “Microcorruption”, designed to teach students reverse engineering concepts and buffer overflow attacks [12]. In addition to these pre-written challenges, TurtleMCU presents a perfect opportunity to direct students to discover and implement their own vulnerabilities and attack mitigations in both software and hardware by experimenting with the processor itself as well as expanding it with their own designs. Each of the following subsections describes a security education application of TurtleMCU as part of a hands-on, laboratory-style computer architecture curriculum.

5.1. Reverse Engineering

An oft-overlooked security concern is the possibility that an adversary may be able to reverse engineer machine code

to find passwords or possible exploits. TurtleMCU allows students to explore these types of attacks by providing a number of ways to engage them with simple reverse engineering challenges. First, the debug environment shows disassembly of the machine code executing on the microprocessor. This automatically shows students the effect of TurtleASM directives on the binary layout.

Furthermore, the availability of System Verilog macros in specifying the ROM allows educators to customize the difficulty of reverse-engineering assignments. At its easiest, a ROM might be identical to the disassembly shown in the debug environment. However, the ROM could display the machine code in hexadecimal, requiring students to learn how TurtleMCU decodes instructions and immediately apply what they learn to interpret the ROM.

5.2. Buffer Overflow Exploitation and Prevention

As previously stated, TurtleMCU comes with a suite of security challenges inspired by Microcorruption CTF, designed to teach students about buffer overflow exploits. For example, Code Sample 3 demonstrates a program written for TurtleMCU which is vulnerable to buffer overflow. Ostensibly, the program asks for input and then echoes it to the terminal. Students would endeavor to cause the program to print "This should never happen!" to the terminal by overflowing the allotted input buffer to overwrite the return address on the stack.

Code Sample 3. *Buffer Overflow Vulnerability*

```
.org 0x000

start:  lsp  0x80
    .ldtag r0, s1
    call puts
    rsp  r0
    mvl  r1, 0x20
    sub  r0, r1
    call gets
    call puts
done: jmp done

.org 0x100
.string s1 "Please type something: "
.string s2 "This should never happen!"

.org 0x200
    .ldtag r0, s2
    call puts
    jmp done
```

The program begins by setting the stack pointer to address 0x80 and then printing string s1 to the terminal. Next, 20 words of memory are reserved on top of the stack, and a pointer to that region of memory is passed to the gets function. If the user enters more than the allotted buffer, they overflow the buffer and start overwriting the stack frame, which contains the return address of the gets function. If they overwrite the return address with 0x200, then gets returns to code which prints out "This should never happen!".

This challenge derives heavily from the example of the Microcorruption CTF challenges and extends the challenge into the hardware domain. Whereas the Microcorruption CTF limits the scope of student interaction to a debugger and input/output interface, TurtleMCU provides students with the ability to observe all internal signals and registers within the CPU and the opportunity to modify any part of the environment to explore vulnerability mitigations. Students have the option to implement fixed-length string input functions or incorporate their own hardware memory protection units (section 5.3) to explore the process of combating these vulnerabilities. By doing both, students can compare and contrast the design decisions made by embedded platform designers to implement buffer overflow protections in their designs.

5.3. Memory Protection Unit

TurtleMCU's core can be extended to facilitate student development of advanced processor security techniques. Building on the TurtleMCU platform, students can develop their own Memory Protection Unit (MPU). Control unit signals can be broken out and connected to a custom module to allow programs to mark sections of RAM as read-only or non-executable. Subsequently, students can modify the TurtleASM assembler to support directives to interact with the MPU.

5.4. Hardware-Backed Program Signing

Embedded microprocessors can be found in any number of environments where they have access to privileged or private information and control interfaces [13], [14]. In these environments, it's critical that the device firmware be protected against substitution or modification by an attacker. To prevent firmware modification, many embedded platforms support firmware signing, and these techniques can be taught in a lab-style hands-on environment by building on top of the TurtleMCU platform. Students can extend TurtleMCU to enable initialization-time validation of firmware. Students can incorporate their own HDL to explore code signing techniques and extend both the processor itself and the assembler to enable ROM authenticity checks.

6. Pedagogical Application

TurtleMCU was developed in-part as a greenfield reimagining of the RAT Microcontroller platform currently in use at California Polytechnic State University San Luis Obispo (Cal Poly) [6]. As of writing, aspects of the TurtleMCU platform, such as the assembler, are being incorporated into the existing digital design curriculum at Cal Poly.

7. Conclusion

TurtleMCU combines reverse engineering, vulnerability discovery and exploitation, and vulnerability response and

mitigation techniques at both the hardware and software level to provide a platform which allows introductory computer architecture students with a view into the security of embedded microcontrollers. Where other platforms focus on software security or computer architecture, TurtleMCU combines both perspectives in a way which allows for student engagement even at an introductory level.

The repository with the GPL Version 3-licensed source can be found at
https://www.github.com/nategraff/TurtleMCU.

Acknowledgments

The author would like to thank Dr. Andrew Danowitz, Dr. Bruce DeBruhl, and Max Zinkus for their help reviewing and editing this paper.

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, Jan. 2018.

[2] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, Jan. 2018.

[3] L. D. Feisel and A. J. Rosa, "The role of the laboratory in undergraduate engineering education," *Journal of Engineering Education*, vol. 94, no. 1, pp. 121–130, 2005.

[4] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, 2013.

[5] D. Harris and S. Harris, *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[6] B. Mealy and F. Tappero, "Free range vhdl," *URL: http://www. freerangefactory. org/dl/free_range_vhdl. pdf*, 2013.

[7] M. Gondree and Z. N. Peterson, "Valuing security by getting [d0x3d!]: Experiences with a network security board game," in *6th Workshop on Cyber Security Experimentation and Test*. USENIX Association.

[8] T. Flushman, M. Gondree, and Z. N. Peterson, "This is not a game: Early observations on using alternate reality games for teaching security concepts to first-year undergraduates," in *8th Workshop on Cyber Security Experimentation and Test*. USENIX Association.

[9] Python package index: Pyparsing. [Online]. Available: https://pypi.org/project/pyparsing/

[10] Introduction to verilator. [Online]. Available: https://www.veripool.org/wiki/verilator

[11] Announcing ncurses 6.1. [Online]. Available: https://www.gnu.org/software/ncurses/

[12] Embedded security ctf. [Online]. Available: https://microcorruption.com/

[13] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Moderator-Ravi, "Security as a new dimension in embedded system design," in *Proceedings of the 41st annual Design Automation Conference*. ACM, 2004, pp. 753–760.

[14] J. Liu and W. Sun, "Smart attacks against intelligent wearables in people-centric internet of things," *IEEE Communications Magazine*, vol. 54, no. 12, pp. 44–49, 2016.