Defense Against the Dark Arts

Mark W. Bailey
Department of Computer Science
Hamilton College
Clinton, NY USA
mbailey@hamilton.edu

Clark L. Coleman
Department of Computer Science
University of Virginia
Charlottesville, VA USA
clc5q@cs.virginia.edu

Jack W. Davidson
Department of Computer Science
University of Virginia
Charlottesville, VA USA
jwd@virginia.edu

ABSTRACT

Computer science faculty must attract and retain students by offering innovative courses that spark student interest, yet still teach core, computer science concepts. These efforts have become particularly important as computer science enrollments have declined, thus increasing concerns for attracting and retaining students. We describe an innovative course that leverages students' interest in computer security issues to attract and retain technically-oriented students. Our initial vision was to offer a course covering computer viruses—a subject that even a novice computer user has some familiarity. To avoid the controversy associated with teaching students how to *write* malicious software, we focused the course on techniques for *defending against* viruses, so we named the course *Defense Against the Dark Arts*. In teaching the course, we have found the subject matter provides an engaging way to introduce and reinforce many important computer science concepts that other courses often cover, most particularly, the traditional compiler course. We have taught the course three times at two separate institutions, with a third school soon to follow. The course has been well received by students—completely filling each semester with enrollments that are four to five times greater than the compiler course. Furthermore, student surveys indicate that the course raises students' awareness of computer security while introducing students to important program translation and analysis concepts.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—Computer science education; D.4.6 [**Operating Systems**]: Security and Protection—Invasive software; D.3.4 [**Programming Languages**] Processors—Compilers.

General Terms

Security, Languages, Theory.

Keywords

Anti-virus software, compilers, computer viruses, computer science education.

SIGCSE'08, March 12–15, 2008, Portland, Oregon, USA.

1 INTRODUCTION

The recent decline in computer science enrollments demands that we develop core curricula that today's students consider relevant. Given that computer security breaches, including those committed by viruses and worms, continually make news headlines, many students' interests are naturally drawn to the field of computer security. Thus, security seems to be an appropriate topic to demonstrate the relevance of computer science while simultaneously providing a solid foundation in core computer science principles. In this paper, we describe an anti-virus course that achieves these objectives and our experience teaching the course at two different institutions.

In our course, students study how effective anti-virus software must automatically identify and defend against malicious software. However, in the course we cover more than just anti-virus techniques and vulnerability issues in software. By studying the ongoing battle between virus writers and anti-virus researchers, students learn that pattern-matching techniques used in early anti-virus software are no longer capable of detecting modern, evolving, and obfuscated viruses. We demonstrate that cutting-edge virus detection techniques use sophisticated program analyses that go far beyond simple pattern matching using virus signatures. Interestingly, modern compilation systems also commonly use such program analyses.

The development of virus detectors using compiler tools presents opportunities to accomplish the objectives of two courses in one. While teaching about the computer security topics related to viruses, worms, vulnerable software, etc., we also introduce core concepts of computer science by studying compiler program analyses. These concepts include applications of theoretical computer science from compiler-related domains such as formal languages and computability theory. We study topics including regular expressions, automata, pattern matching tools (e.g., *lex*), compiler intermediate representations, SSA (static single assignment) form [3], data flow analyses, and the Chomsky hierarchy in the applied subject area of computer security that students find interesting and exciting.

We designed the anti-virus course to be widely disseminated. To ensure the course's suitability for a broad spectrum of colleges and universities, we have developed the course at both a large, public, research university, and a small, private, liberal arts college. We have used feedback from both institutions to refine the course twice, and we soon will make the curriculum materials available through a curriculum repository.

2 COURSE OVERVIEW

Table I lists the course topics. Weeks 1–3 covers anti-virus terminology and the Intel IA-32 assembly language and architecture. Students use simple disassembly and machine code display tools, such as Microsoft Visual Studio's `dumpbin`, in the first programming assignment (assigned at the beginning of week three) to review IA-32 assembly language and examine the Windows program file format.

Week	Contents
1	Introduction; ethics; threat models
2	Terminology; Intel IA-32 architecture
3	Binary disassembly tools; Phoenix compiler
4	Boot, interrupt hooking, memory resident viruses
5	Virus infection of machine code program files
6	Exam 1; detecting viruses using patterns
7	Regular expressions and *lex*; obfuscation
8	SSA form; Phoenix SSA and IR
9	Anti-anti-virus schemes; tunneling, armored, and retro-viruses
10	Exam 2; Encrypted and oligomorphic viruses
11	Polymorphic and metamorphic viruses.
12	Dynamic tools and SDT: security applications
13	Vulnerabilities & exploits; secure coding standards
14	Exam 3; rootkits
15	Special topics and exam review
16	Final exam

Table I. Course topics

In developing the course, we wanted to use an existing infrastructure rather than develop tools from scratch. The ideal choice would enable examining of machine code programs (it is not realistic to tell the students: "pretend the virus writer has given you his source code to analyze"). The system should also enable students to write their virus analyzing code as an extension without requiring the students learn all the details of the infrastructure. The system should document and expose its intermediate representations and its control and data flow data structures, to help the students learn these topics.

We determined that the Phoenix compiler [11] from Microsoft Research satisfied all these criteria. Microsoft plans for this compiler suite to become the next generation of the Visual Studio product line. Unlike many other compilation systems, the Phoenix compiler can manipulate either source code or machine code. The compiler translates the machine code into a low-level intermediate representation (LIR) that numerous Phoenix program analyses can process. Students can take advantage of the modular plug-in architecture of Phoenix to insert analyses without understanding the intricacies of the supporting compiler. Phoenix provides Visual Studio wizards that generate code to plug in new analyses into Phoenix, so that students can focus on core topics rather than insignificant system details.

The lectures and documentation on Phoenix describe the overall structure of the compiler, the various intermediate forms used in Phoenix (e.g., high-, medium-, and low-level intermediate representations), and why the compiler requires different levels of IR for different analyses and transformations. In the second programming assignment, students learn to extract and analyze the LIR form of a program in preparation for more extensive analyses in subsequent assignments

Weeks 4–11 of the course integrate compiler concepts with anti-virus material. We teach the underlying mechanisms used by viruses, without teaching how to write a complete virus. We demonstrate the ongoing battle between developers of viruses and developers of anti-virus scanners through historical examples.

We use code examples from the DOS era. Modern virus writers still use the fundamental concepts these DOS era viruses exhibit. The fragments of DOS virus code we provide will not work in a modern operating system. By focusing on key fragments of DOS viruses, we teach the fundamental concepts without providing malicious code that could spread in today's computing environment.

Students gain experience in writing code to recognize viruses during week seven. After teaching how anti-virus scanners use pattern matching to find viruses in programs, we introduce regular expressions. We explain the use of tools such as *lex* [9] to recognize code using regular expressions and students apply *lex* skills in the third programming assignment. Viruses often use code sequences that a compiler would never generate. We ask the students to write a *lex* scanner specification to recognize such code as that used by a DOS interrupt-hooking virus:

```
mov  eax,4Ch
mov  dword ptr [eax],edx
```

This code fragment writes the contents of register `edx` into address 4C. In DOS, this address contains a pointer to code to handle disk interrupts. An interrupt-hooking virus would make register `edx` point to the address of its own replacement interrupt-handling code for disk accesses so that all disk interrupts would pass through the virus code. This enables the virus code to infect diskettes as they are loaded into the system, for example. In a modern Windows system, the code fragment code generates a system error message and terminates the program; we also construct the code such that register `edx` does not point to any malicious code. The code fragment still provides an opportunity for realistic pattern matching of code that would cause an anti-virus scanner to raise an alarm.

Students pass the machine code through `dumpbin` to disassemble it, then pass the disassembled code through the program produced by their *lex* pattern specification. A correct pattern specification will detect the malicious code.

In a compiler course, students would learn how to create a *lex* scanner to recognize keywords and identifiers of a programming language while studying lexical analysis. *Defense Against the Dark Arts* exposes students to the same material—*lex* and regular expressions—in a more interesting context.

In the first teaching of the course, we then discussed simple code obfuscation techniques used by virus writers. Students then created a *lex* pattern file to reverse code obfuscations. Students passed the resulting output through their program from the previous assignment to detect the (now obfuscation reversed) malicious code. Based on student feedback, we dropped this programming assignment in the subsequent versions of the course in favor of more programming assignments using Phoenix compiler technology, which the students found to be the most interesting aspect of the course assignments.

In recent versions of the course, we explain obfuscation techniques, and show how regular expression tools can identify and reverse only a small subset of possible code obfuscations. For other obfuscation techniques, more powerful program analyses are required. For example, if a virus writer inserts instructions that can be seen, without program context, to accomplish nothing (e.g., instructions with no effect, such as adding zero to, or subtracting zero from, a register) then a *lex* transformation could remove them. However, if a virus writer obfuscates a malicious code sequence by inserting the instruction: `add eax, ebx`, then the instruction can only be declared to have no effect if the value in the destination register is not subsequently used.

We teach many fundamental computer science concepts using the difficulty of reversing obfuscations. First, we use the Chomsky language hierarchy and the pumping lemma for regular languages to explain why pattern-matching tools can never reverse all possible obfuscations. Students usually encounter such topics only in formal languages courses, or sometimes within compiler courses. In *Defense Against the Dark Arts*, students learn the direct application of topics that otherwise would seem to be purely theoretical in the computer science curriculum. For example, assume that the following instructions are part of a virus:

```
    mov     ebx,f400h   ; get destination addr
    mov     cx,3f4h     ; get size of virus
    mov     edx,eip     ; get instruction pointer
    sub     edx,472h    ; get start of virus code
L1: movs    [ebx],[edx] ; copy virus
    inc     ebx
    inc     edx
    loopcx  L1
```

Here, the virus is copying itself to a new address, perhaps to propagate itself, or for some other malicious purpose. We could add a pattern of the code above to a virus signature database to enable detection of the virus by pattern matching. However, the virus writer could simply add some obfuscating instructions to change the sequence, causing it not to match the signature in the database:

```
    mov     ebx,f400h   ; get destination addr
    inc     ebx         ; obfuscate
    mov     cx,3f4h     ; get size of virus
    mov     edx,eax     ; junk instruction
    mov     edx,eip     ; get instruction pointer
    sub     edx,472h    ; get start of virus code
    dec     ebx         ; obfuscate
L1: movs    [ebx],[edx] ; copy virus
    inc     ebx
    inc     edx
    loopcx  L1
```

Incrementing register `ebx` and decrementing it before its next use preserves the semantics of the program, but changes the virus signature. To detect this by pattern matching, one would have to create patterns that match, pair-wise, increments with decrements. However, there is no limit to the number of increments, as long as they have matching decrements. The pumping lemma for regular languages proves that no regular language (or finite state automaton) can match such an unbounded, paired sequence.

In compiler courses, students learn the relative power of languages at different levels of the Chomsky hierarchy using the problems of lexical analysis, syntax analysis, and semantic analysis. Each of these analyses requires the power of a language at a different level of the Chomsky hierarchy. In the anti-virus assignments, students are impressed with the different analytical requirements of obfuscation reversal tasks. By having a virus obfuscation technique defeat a student's analysis, and then having the student defeat the obfuscation with a more powerful analysis, we make this point more memorably than we could in a traditional compiler class.

We teach other compiler techniques that recognize obfuscating "junk instructions" with an interesting, motivating example. In the previous obfuscated code, the instruction "`mov edx, eax`" is a junk instruction because register `edx` immediately acquires a new value in the next instruction, so the old value is never used. A data flow analysis can follow a definition-use chain and find all definitions that have no uses. The corresponding assignment instructions can be removed from the code. Students are taught SSA form, both in general and in the data structures built within Phoenix. SSA form is an intermediate representation that incorporates control flow and data flow information, including definition-use chains [3]. We then assign the fourth programming project, in which students perform simple SSA data flow analysis to detect and remove junk instructions. Note that SSA form and data flow analysis concepts are not usually taught in undergraduate compiler courses, because too much engineering work is required in a one semester compiler course to create a working compiler with such advanced features. Our anti-virus course presents the perfect opportunity to introduce important compiler concepts that are often not possible to include in an undergraduate compiler course.

After describing how evolving and mutating viruses obfuscate themselves, we give a fifth programming assignment at the end of week 11. Again, we use a Phoenix plug-in analysis to disable a realistic virus obfuscation technique. Evolving viruses often change their code signatures by rearranging a sequence of instructions so that a jump is required to get from each instruction to its successor. A regular expression pattern matcher cannot detect all such rearrangements, but a compiler with simple control flow analysis can rearrange such jumbled code into a simple sequence. After explanation of the control flow analysis and resulting data structures within Phoenix, students write a plug-in that finds code blocks that can only be reached by a jump (not by fall-through from a previous instruction) and that only have a single jump as a predecessor in the control flow graph. These code blocks are moved so that they directly follow their predecessors. After this process iterates to completion, the obfuscating jumps all transfer directly to the next instruction in the program, and existing Phoenix optimizations remove them. Students print the control flow graph and inspect it to confirm that all obfuscating jumps have been removed (and to learn about control flow graphs!)

Lectures emphasize that the battle against viruses can never be completely won. Any reverse obfuscation program that could verify that an obfuscated program is equivalent to another program could solve the halting problem, which is undecidable. We illustrate this fundamental concept with an compelling example.

Week 12 of the course looks at dynamic tools, in general, and Software Dynamic Translation (SDT) in particular. SDT fetches, translates, and executes instructions within a virtual execution environment. While fetching and translating the instructions, SDT can apply certain security policies with little overhead [14]. For example, SDT can ensure that a program does not fetch instructions from the address range containing the stack, which is indicative of a code injection attack. Other dynamic tools, such as emulators, can be important in analyzing newly discovered viruses. An SDT or emulator can observe an encrypted virus decrypt itself, and then analyze the virus; this is not possible with static analysis tools. We discuss the differing capabilities of static and dynamic compilation concurrently with discussion of different capabilities of static and dynamic anti-virus tools.

In weeks 13–15, we move beyond viruses to other important security issues. We examine code vulnerabilities, and exploits of those vulnerabilities, in detail. Students learn secure coding techniques that reduce or eliminate such vulnerabilities. In the sixth and final programming assignment, we give the students a machine code program that contains unused code to print a certain string (such as "You have passed this assignment!"), and tell them that the program contains a buffer overflow vulnerability using its command line arguments. Using `dumpbin` to find the address of the string, and then to find the function that prints the string, students must figure out how to pass an address in the command line to overwrite a return address and cause the function to be called and print the string. How to perform such an exploit is common textbook knowledge, but performing the exercise gives an intimate understanding of code exploits and how hackers can easily discover such vulnerabilities (and why the secure coding standards we have presented are the proper defense).

After covering vulnerabilities and exploits, we discuss rootkits. We use the Sony rootkit fiasco of 2005 [13] as an example of how their personal computers can be commandeered by malicious software that conceals its presence. Sony was attempting to prevent copying of its music CD's, a topic of great interest to college students! Special topics, especially those that are in current headlines, can be taught throughout the course.

3 EXPERIENCE

3.1 University of Virginia

We first taught this course in the fall of 2005. Prerequisites were the core courses of the first two years of the curriculum, from CS1 through data structures. The size of the classroom limited enrollment to 50 students. Prior to the beginning of the semester, enrollment reached this limit and we placed students on a waiting list. This student interest contrasts sharply with the upper division elective course in compilers, which typically draws 12–14 students.

A personal computer laboratory with 50 computers running Windows XP is available for course work, as well as other labs maintained by the engineering school and the university. We installed on all 50 machines a release of Cygwin [8] with *lex*, the Phoenix compiler suite, and Microsoft Virtual PC, a product similar to VMware [17]. We installed Virtual PC in order to have a protected "sandbox" for student work to ensure that no virus assignments could do any damage. By the end of the semester, it was obvious that no such damage was possible from any of these assignments and Virtual PC was not necessary for the course.

Course feedback from the students was positive. The Phoenix compiler assignments drew particular praise, and several students wanted more of these assignments. Thus, we dropped the second *lex* assignment (reversing obfuscations) in order to make room for another Phoenix assignment.

After refining the course materials and adapting assignments and documentation to a new release of Phoenix, we offered the course again in the spring of 2007. Again, the room limited enrollment, this time to 30 students. We used questionnaires at the beginning and end of the semester to measure student interests and knowledge in computer security. Interest in taking further computer security courses and working in computer security increased during the semester. We observed significant increases in awareness of virus and worm issues and software vulnerabilities in the student responses. Student enrollment included seven students from outside the computer science department but within the engineering school, and two others from the economics department in the college of liberal arts and sciences.

Microsoft released a new version of Phoenix in the fall of 2006. With this release, Microsoft made Phoenix available for free, so students were able to work from home and were not constrained by the lab schedules thereby freeing valuable laboratory resources.

3.2 Hamilton College

In the spring of 2007, we taught the course at Hamilton College. Prerequisites were CS1, data structures, computer organization, and formal languages. In contrast to the University of Virginia, Hamilton is a small, highly selective, liberal arts college. The department has four computer science faculty members and enrollments in the major average about 10 students per class year. While attracting and retaining majors is a major concern, we also face the challenging curriculum pressures that all departments our size face.

We use a dedicated laptop laboratory with computers configured with Windows XP, Visual Studio, and the latest release of Phoenix. We found the dedicated lab to be particularly useful in this course since the software base is particularly sophisticated and cutting-edge. Students were neither familiar with Phoenix nor Visual Studio prior to enrollment in the course.

A key difference between the Hamilton and University of Virginia offerings is the prerequisites. Because we require formal languages and computer organization, we can compress much of the introductory material covering these core concepts. As a result, we have more time to study compiler analyses and we included a research and presentation component at the end of the semester.

Our experience teaching *Defense Against the Dark Arts* at Hamilton is that the course is both flexible and appropriate for a department with very high curriculum pressures. Because of the liberal arts mission of the college, we must focus on computer science fundamentals. This course has enabled us teach these core concepts, as well as advanced topics we never cover in our compilers course, in the context of a topic of great interest to students.

4 PRIOR WORK

As computer security has become a topic of interest to the public, many have developed security courses to address the growing demand. Among others, these include virus courses, anti-viruses courses, and hacking courses. While all three types of courses share a common theme—code vulnerabilities and exploits—they differ in approach. Hacking and virus courses focus on the learning how to develop malicious code to gain an understanding of how to exploit code vulnerabilities [1, 10, 7]. Such courses have raised concerns in both the popular press and anti-virus communities [12]. In contrast, we are unaware of any other work that focuses on teaching *anti*-virus techniques. This difference is critical: virus and hacking courses teach how to write malicious code, while our anti-virus course teaches techniques of program analysis that are often applied to more general problems as well as virus detection.

For decades, courses that study compiler design have been a staple of core curricula. Because of the size and complexity of a compiler, there is a rich body of work documenting approaches to teaching the course. A selection of recent work includes attempts to make compiler construction more manageable by using tiny languages [2, 6], more relevant by broadening language translation [4, 16], more viable in a broadening computer science curriculum [15], or more applicable to other courses [5]. These approaches focus on the process of constructing a compiler and thereby teach the underlying fundamental concepts that enable translation. While understanding program analysis techniques is fundamental, we believe that developing a compiler to learn the techniques is not.

5 CONCLUSIONS

We have developed and refined an anti-virus course that explores many fundamental and advanced compiler concepts. By teaching important topics from two domains in one course, we resolve the curriculum competition between hot, new, topical courses and traditionally core, course offerings. The shift in focus from compilers to security enables us to reach many more students while simultaneously teaching the most interesting and important ideas from a traditional compilers course. In addition, the course's flexible content makes it amenable to adoption at a variety of institutions.

6 ACKNOWLEDGEMENTS

We thank Microsoft Research External Research and Programs for their support of this work. In particular, we thank Yan Xu and John Lefor for their commitment to the project, and Andy Ayers and the entire Phoenix team for developing Phoenix and their technical support for the Phoenix infrastructure.

7 REFERENCES

[1] AYCOCK, J., AND BARKER, K. Viruses 101. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (2005), pp. 152–156.

[2] BALDWIN, D. A compiler for teaching about compilers. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (2003), pp. 220–223.

[3] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13*, 4 (1991), 451–490.

[4] DEBRAY, S. Making compiler design relevant for students who will (most likely) never design a compiler. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (2002), pp. 341–345.

[5] DEMAILLE, A. Making compiler construction projects relevant to core curriculums. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (2005), pp. 266–270.

[6] FRENS, J. D., AND MENEELY, A. Fifteen compilers in fifteen days. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (2006), pp. 92–96.

[7] JENSEN, B. K., CLINE, M., AND GUYNES, C. S. Teaching the undergraduate CS information security course. *SIGCSE Bulletin 38*, 2 (2006), 61–63.

[8] LAZENBY, D. Cygwin: For Windows NT. *Linux Journal 2000*, 75es (2000), 14.

[9] LESK, M. Lex: a lexical analyzer generator. Tech. Rep. 39, AT&T Bell Laboratories Computing Science, Murray Hill, New Jersey, USA, 1975.

[10] LOGAN, P. Y., AND CLARKSON, A. Teaching students to hack: curriculum issues in information security. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (2005), pp. 157–161.

[11] MICROSOFT CORPORATION. Phoenix. http://research.microsoft.com/phoenix/, 2007.

[12] READ, B. How to write a computer virus, for college credit (cover story). *Chronicle of Higher Education 50*, 19 (Jan. 2004).

[13] RUSSINOVICH, M. Sony, rootkits and digital rights management gone too far. http://tinyurl.com/y94s8m, 2005.

[14] SCOTT, K., AND DAVIDSON, J. Safe virtual execution using software dynamic translation. In *Proceedings of the 18th Annual Computer Security Applications Conference* (2002), p. 209.

[15] WAITE, W. M. The compiler course in today's curriculum: three strategies. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (2006), pp. 87–91.

[16] WAITE, W. M., JARRAHIAN, A., JACKSON, M. H., AND DIWAN, A. Design and implementation of a modern compiler course. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (2006), pp. 18–22.

[17] WALTERS, B. VMware virtual platform. *Linux Journal 1999*, 63es (1999), 6.