Binary Reverse Engineering for All

John Aycock
aycock@ucalgary.ca
University of Calgary
Calgary, Alberta, Canada

ABSTRACT

We report our experience with a novel course on binary reverse engineering, a university computer science course that was offered at the second-year level to both computer science majors as well as non-majors, with minimal prerequisites. While reverse engineering has known, important uses in computer security, this was pointedly not framed as a security course, because reverse engineering is a skill that has uses outside computer science and can be taught to a more diverse audience. The original course design intended students to perform hands-on exercises during an in-person class; we describe the systems we developed to support that, along with other online systems we used, which allowed a relatively easy pivot to online learning and back as necessitated by the pandemic. Importantly, we detail our application of "ungrading" within the course, an assessment philosophy that has gained some traction primarily in non-STEM disciplines but has seen little to no discussion in the context of computer science education. The combination of pedagogical methods we present has potential uses in other courses beyond reverse engineering.

CCS CONCEPTS

• **Security and privacy** → **Software reverse engineering**; • **Social and professional topics** → *Student assessment*; • **Applied computing** → Interactive learning environments.

KEYWORDS

binary reverse engineering, ungrading, assembly language

ACM Reference Format:
John Aycock. 2023. Binary Reverse Engineering for All. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023), July 8–12, 2023, Turku, Finland.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3587102.3588790

1 INTRODUCTION

In the world of computer security, reverse engineering binary code is a critical skill. Security analysts reverse engineer malicious software to understand its inner workings; legitimate software is studied to discover exploitable flaws; the inner workings of filesystems are probed for computer forensics purposes. Binary reverse engineering also has non-security-related applications in situations including where a legacy program's source code is lost, where necessary hardware and software are defunct (e.g., [19]), when an insufficiently documented API must be used, or when understanding a mysterious network protocol.

What may come as a surprise is that binary reverse engineering is not only useful in security, or even in computer science. Indeed, a recent case for it was made in the archaeology journal *Antiquity* as one way to analyze digital artifacts [2]. Casting the net more widely within the humanities, we can find work approaching reverse engineering in subareas such as platform studies (e.g., [1, 26]) and what Kirschenbaum called 'textual forensics' [22] but is arguably media archaeology, with other subareas such as game studies, software studies, and critical code studies also standing to benefit. And for history, well-accustomed to textual evidence, binary code and data can be viewed as a "text," albeit not one using natural language. Reverse engineering need not even be conducted *for* any specific application domain, and can simply be undertaken as a challenging intellectual puzzle – as the most neutral framing, we adopted this perspective for the course.

To teach reverse engineering to a diverse audience, we created a new "Introduction to Reverse Engineering" course, an optional computer science course at the second-year (sophomore) level open to both computer science majors as well as non-major students. The only course prerequisite was an introductory programming class – CS1 or equivalent – allowing the primary focus to be on reverse engineering rather than teaching programming. It was initially offered in January 2022 to two sections of 30 students each, and ran again with one section of 60 students in January 2023. The course can be roughly divided into four parts: binary data formats first, and the other parts introducing reverse engineering for three different assembly languages and platforms of increasing complexity.

This high-level description of the course content admittedly sounds ambitious, and it is worthwhile to examine why this plan is feasible. We can group the students into two categories: more senior computer science students taking the course, and students (major or non-major) who have little to no experience beyond CS1.

We consider the latter category of students first. In our own undergraduate computer science education some years ago, we had two computer science courses in the first year. The first course, equivalent to CS1 now, was learning to program in a high-level language, and *the second first-year course was programming in assembly language*. It is not that low-level assembly language is hard, or impossible for students new to programming to grasp; we have just chosen not to teach it at that point in the program any more. In other words, the low-level course material of a reverse engineering class, suitably targeted, is completely reasonable to tackle with only a CS1 background. This low bar to entry is why the course can be open to majors and non-majors, because the difference in their computer science background at that point is essentially nil.

ITiCSE 2023, July 8–12, 2023, Turku, Finland
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0138-2/23/07…$15.00
https://doi.org/10.1145/3587102.3588790

Senior students in computer science, the first category, may well have taken the only mandatory course remaining in our program that covers assembly language. However, a reverse engineering course still adds value for these students, due to a global historical trend in computer science education over a span of decades: there is increasingly less focus on low-level content. Looking at the ACM/IEEE curriculum guidelines from 2013 [20], nearly a decade old now, we already see low-level data representation and assembly language having second-class 'Tier-2' status, with only a single-digit number of hours dedicated to the topics in total. The learning outcomes listed for these topics are overwhelmingly 'Familiarity,' a far cry from the level of expertise a student would need for applied computer security, for example. Given that low-level topics are increasingly alien to computer science majors, a reverse engineering course that offers a gentle re-introduction and deeper dive into those areas is beneficial to students needing to specialize more.

In the remainder of this paper we explain our reverse engineering course in detail. Section 2 contrasts what we are doing with the limited examples of reverse engineering course content in the literature. Section 3 talks about the four parts of the course; Section 4 explains the role of "ungrading" in it. Our experience with the course and ungrading is in Section 5, followed by our conclusions.

2 RELATED WORK

Security and reverse engineering challenges are frequently presented in the guise of "capture-the-flag" (CTF) competitions, e.g., [10, 37, 38, 43]. Some, but not all, question the effects of the competitive aspect of CTFs; Taylor et al., for instance, write that 'competition can have a negative effect on educational outcomes' [38, p. 6] for novices. This is echoed elsewhere [40, p. 55]: 'It may be possible that competitions discourage those with little prior experience in cybersecurity.' Vykopal et al. try to measure the efficacy of CTF usage, although they concede that their post-CTF questionnaire asking about a preference for CTFs versus 'normal homework' had few takers [43]. Cole [11] reported a stronger result, but still questions remain. In addition to these questions only being raised from the security point of view, we would argue that the people who have signed up for a security course or a CTF competition are already self-selecting to a large extent and are not representative of the wider population. There may be a much greater untapped pool of people who would be interested in, and capable of, reverse engineering were it not for its presentation in a competitive setting. For that reason, we have deliberately opted for a noncompetitive approach to reverse engineering to avoid creating an offputting learning environment.

But can reverse engineering be taught? Mahoney and Gandhi dismissively opine that 'students either get it [reverse engineering] or they don't' [24, p. 56] but this could be a reflection of their teaching approach. By contrast, Richard claimed students could learn 'an impressive set of skills' [32, p. 1] in the time frame of his one-semester, malware-based reverse engineering course. One step further is the argument that reverse engineering is a 'thought process' to be acquired [3], which in a security setting might be conflated to some extent with adversarial thinking [31]. A recent study of reverse engineers performing static analysis tasks lends credence to the idea that there is indeed a process to be taught [25]. No courses seem to present reverse engineering outside the context of computer security, however, nor do they target as diverse an audience as we do. The archaeologist Gabriel Moshenka provides some validation of a non-security-exclusive position on reverse engineering, in fact, pointing out the linkages between reverse engineering and archaeology [27].

There is a fair bit of work on automatically generating a variety of challenges for capture-the-flag and reverse engineering exercises (e.g., [3, 10, 15, 30, 39]). We can see generating a range of challenges more generally in terms of randomizing problems for students [7, 8, 44], software diversity [17], a legitimate application of malware polymorphism and metamorphism [36], and even procedural content generation [41]. Regardless, it is no problem integrating any of these problem-generation techniques into our course, and with three different assembly languages and platforms, we have substantial latitude as to where and how they can be used. It is notable that automatic challenge generation is seen as an anti-cheating measure [10, 30], implying the primacy of getting the right answer over the reverse engineering thought process. This is the reverse of our course's emphasis on process, an emphasis we reinforce through ungrading.

Ungrading itself has seen only fleeting mentions within computer science education (e.g., [4, 9, 28]) and we are not aware of any experience reports with it like we provide here apart from Riesbeck's 'critique-driven' method [33]. This is not unexpected given Blum's observation about ungrading in STEM disciplines, that 'there tends to be more flexibility' in non-STEM fields [5, p. 222], making our account of its use within computer science a substantial contribution.

3 THE COURSE

As mentioned, the content in our reverse engineering course can be divided into four parts, presented here in the order in which they appear in the course. There were additionally two guest lectures, one by a colleague in the law faculty regarding legal aspects of reverse engineering, and another given by a colleague who demonstrated reverse engineering code for mobile (Android) devices.

3.1 Binary Data Formats

The course begins, once past the usual introductory material, with data formats and beginning to train students to look for patterns in numbers. We start with all numbers in base 10 for familiarity, where each (memory or file) location can hold a value between 0 and 999, with numbers formatted for analysis in the same way that eventual hex-dumping displays will show. This gives the basis for explaining endianness, string formats, character encodings, and sequences of items. It also allows problems to be highlighted early, like potential ambiguities in reverse-engineering data that cannot be resolved without recourse to code analysis or being able to craft specific inputs, and discussions on how complex a pattern a reverse engineer might consider looking for in data.

The course has no tutorials or labs, and was designed from the outset to give students enough time in class to try their hand at working through the reverse engineering exercises themselves. A corollary to this is that all online systems used for instruction had to function only using a modern web browser, regardless of the device

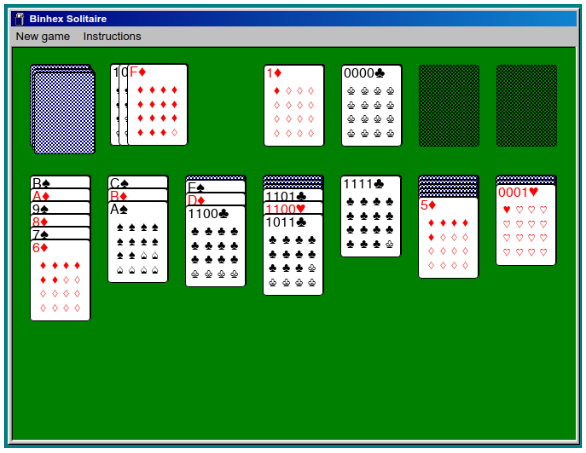


Figure 1: Binary/hexadecimal Solitaire variant

students brought with them, without software installations – even a Chromebook or tablet had to work. Leaning into the presentation of exercises as puzzles, rather than a security focus that might not resonate with all students, we gave in-class exercises difficulty ratings as one would see for other puzzles. For example, an exercise with a one-star rating might be determining a string's location, format, and encoding, knowing the input numbers and the output string; a three-star exercise might be the same task, but where a simple substitution cipher is involved.

Once these basic ideas are in place, we transition into the more usual number bases for reverse engineering: base 2 and base 16. That opens the door to exercises using a variety of data types (e.g., signed and unsigned integers, pointers) and our first web-based systems. For learning and practicing number bases, we created two adapted online versions of Solitaire using 16 cards per suit rather than the usual 13, where each card has a binary or hex number on it, and the player plays Solitaire as usual but must decide which cards can be played on one another based on their values in the respective bases.[1] One version (not shown) is binary-only and the other has suits of binary and hex (Figure 1). We also introduce a first reverse engineering tool with a web-based version of Kaitai Struct [21], which allows students to explore hex data as well as document their reverse engineering findings using Kaitai Struct's YAML-based data description language.

3.2 CLMC

On to assembly code. We introduce assembly language using a simplified architecture called CLMC. CLMC stands for Calgary Little Man Computer, and is based on the so-called "Little Man Computer" architecture used by Stuart Madnick at MIT starting in the mid-1960s [14, 23].[2] It uses base 10, putting other bases on hold temporarily, a small 100-location memory (each able to hold a value between 0...999 per the initial data format examples), and a single accumulator (A), program counter, and overflow status bit (V).

[1]Despite the game's apparent ubiquity, not everyone knew how to play Solitaire!

[2]The reference to "Man" in the name is unfortunate to a modern eye. It refers not to the programmer but to the simplified explanation of instruction execution as being performed by a '"little man" in a room' [23, p. 2].

Table 1: The CLMC instruction set

Instruction	*Description*
`halt`	Stops the CLMC
`add` *address*	Adds value at *address* to A, stores result in A
`sub` *address*	Subtracts value at *address* from A, stores result in A
`lda` *address*	Loads value at *address* into A
`sta` *address*	Stores value in A at *address*
`b` *address*	Branches to *address* unconditionally
`bz` *address*	Branches to *address* if A = 0
`bo` *address*	Branches to *address* if V = 1
`get`	Reads a number from input into A
`put`	Outputs the number in A as a number
`putc`	Outputs the number in A as an ASCII character

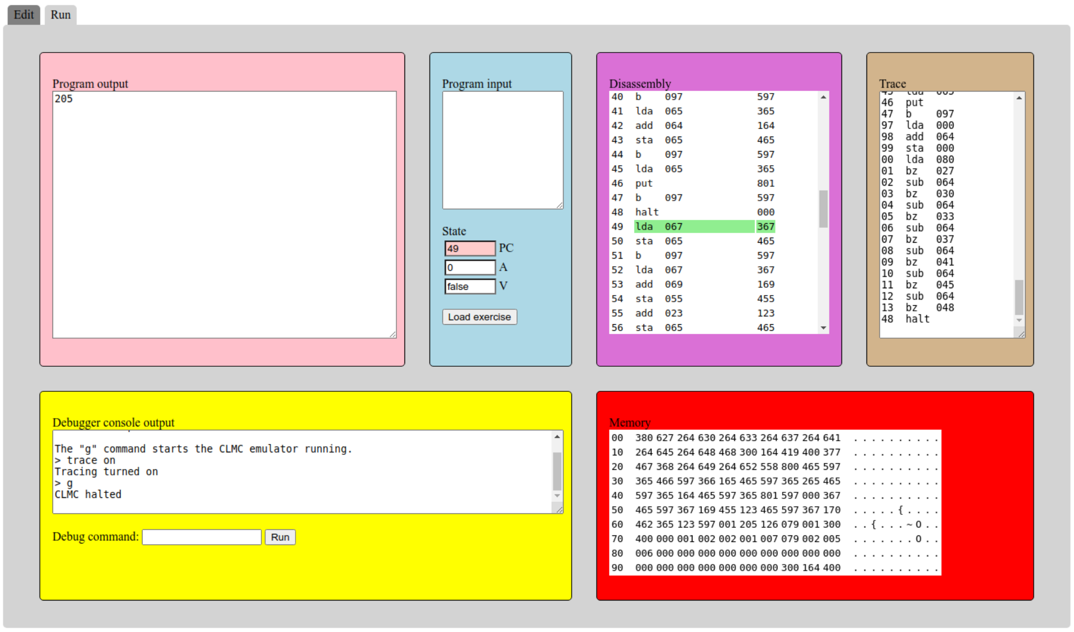
Importantly, from the point of view of learning the language and reverse engineering CLMC programs later, the entire instruction set (Table 1) is brief enough to fit on a single slide, with room to spare for assembler directives. Character-based output was added to permit more interesting exercises, and for advanced reverse engineering exercises later, there is an undocumented instruction hidden in CLMC.[3] For other instructors wanting to adopt our approach, we note that the use of CLMC specifically is not imperative; what is important is selecting a simple and approachable first assembly language, where the design of the instruction set and the interface acts as a stepping stone to later systems.

Students are first given a crash course in (CLMC) assembly programming by way of seven examples of increasing complexity followed by a selection of programming puzzles. The idea is not to make students into adept assembly language programmers, nor to focus on programming for long; students only need enough to understand the basics of CLMC, and how far programs need to be broken down when they are implemented in a low-level language. By way of analogy, when learning a foreign human language, it is possible to read and understand much more than one can write in the language, and we are leveraging that same principle here, but for reverse engineering assembly code instead.

The web-based interface we built for CLMC is split across two tabs. When programming, the one tab has a code editor, and an assembler to invoke as needed. When running and reverse engineering code, the other tab is their primary interface (Figure 2); it shows input, output, current machine status, a complete memory view, and a disassembly. The debugger console supports features for reverse engineering of code, including the ability to set a breakpoint, a watchpoint, single-step forwards and backwards, and capture an execution trace. The interface also highlights registers and memory values that have been changed when an instruction executes, for easier use by novices. Both tabs allow instructor-made assembly/preassembled exercises to be loaded, respectively.

Through preassembled CLMC examples and exercises, the basics of reverse engineering code are taught, both static and dynamic analysis. More straightforward CLMC programs are used to illustrate the thought process, and how to ask and answer appropriate questions using the available debugging tools; we then move on to

[3]The implementation of the undocumented instruction is obfuscated in the JavaScript code for CLMC. Something about reverse engineering will be learned either way.



Program output:

```
205
```

State: PC 49, A 0, V false

Disassembly:

```
40  b     097       597
41  lda   065       365
42  add   064       164
43  sta   065       465
44  b     097       597
45  lda   065       365
46  put             801
47  b     097       597
48  halt            000
49  lda   067       367
50  sta   065       465
51  b     097       597
52  lda   067       367
53  add   069       169
54  sta   055       455
55  add   023       123
56  sta   065       465
```

Trace:

```
46  put
47  b    097
97  lda  000
98  add  064
99  sta  000
00  lda  080
01  bz   027
02  sub  064
03  bz   030
04  sub  064
05  bz   033
06  sub  064
07  bz   037
08  sub  064
09  bz   041
10  sub  064
11  bz   045
12  sub  064
13  bz   048
48  halt
```

Debugger console output:

```
The "g" command starts the CLMC emulator running.
> trace on
Tracing turned on
> g
CLMC halted
```

Memory:

```
00  380 627 264 630 264 633 264 637 264 641  . . . . . . . . . .
10  264 645 264 648 468 300 164 419 400 377  . . . . . . . . . .
20  467 368 264 649 264 652 558 800 465 597  . . . . . . . . . .
30  365 466 597 366 165 465 597 365 265 465  . . . . . . . . . .
40  597 365 164 465 597 365 801 597 000 367  . . . . . . . . . .
50  465 597 367 169 455 123 465 597 367 170  . . . . . { . . . .
60  462 365 123 597 001 205 126 079 001 300  . . { . . . ~ O . .
70  400 000 001 002 002 001 007 079 002 005  . . . . . . . O . .
80  006 000 000 000 000 000 000 000 000 000  . . . . . . . . . .
90  000 000 000 000 000 000 000 300 164 400  . . . . . . . . . .
```

Figure 2: In-browser CLMC execution and debugging environment

advanced examples where deliberate obfuscation is used [12], including forms of custom virtual machines and "encrypted" code. The latter naturally leads into learning bitwise operations like exclusive-or, which is also supported by an online game [45].

3.3 6502

The next step is to transition to reverse engineering code on a real, but simple architecture on a simple platform. For this, we chose the Commodore VIC-20 from 1980, with a 6502 CPU; platform-wise, the machine has ROM code but no operating system *per se* to contend with. Reference materials for the widely used 6502 and its small set of 56 instructions are readily available, as is an excellent reference manual for the VIC-20 [16]. Because the difference from one assembly language to the next tends to be relatively minor, we present it by way of comparison to CLMC as much as possible, something we do in the later transition from 6502 to x86-64 too.

The reverse engineering interface is shown in Figure 3, with the VIC-20's screen contents at the top and debugger functionality below. The Internet Archive has the ability to run the MAME emulator (among others) in-browser [34]; we built on MAME and the Internet Archive code and created a version with a debugger that can be used for reverse engineering – the CLMC debugger's commands had actually been chosen to map into the MAME debugger's more featureful command set.

Reverse-engineering 6502 code naturally brings back base 16 from before, and lets new ideas be introduced that would have been impossible (or unwieldy) on CLMC. For example, the concept of a stack and subroutines can now be incorporated, as can interrupts and bitwise operations. Given the timing of this 6502 material, and the real but not overly complex platform, the students' term project (discussed later) is to reverse engineer a game on the VIC-20.

3.4 x86-64

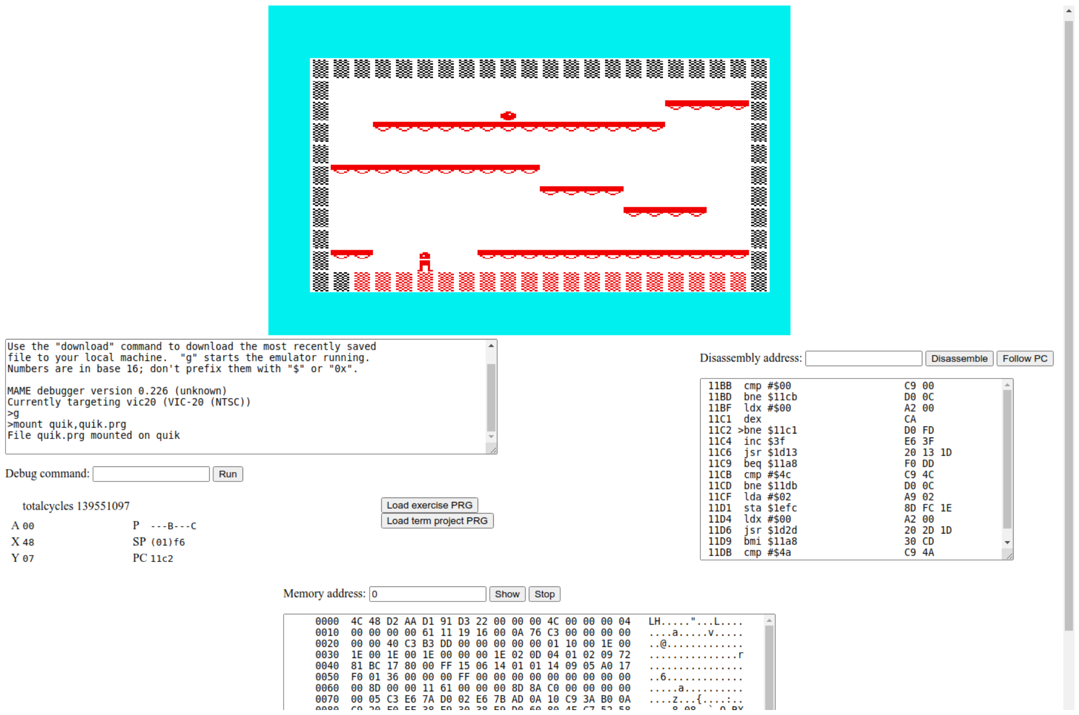
The final step brings students into the modern era, reverse engineering x86-64 code on Linux. For in-browser access, we made use of Ubuntu VMs running on our university's VMware Horizon system; in other words, each student had their own instance of a graphical Linux desktop available from their browser.

Learning the absurdly large x86-64 instruction set is not the main goal here. This was the transition into a much more complex operating system environment, and working with compiler-generated code. It is necessary to introduce systems content here to understand what is being seen during reverse engineering, like stack frames, static and dynamic linking, process address space contents, system vs. library calls, and executable file formats, along with a litany of command-line tools to tease out this information statically and dynamically.

Ultimately, this leads to using the most extensive reverse engineering tool the students see, Ghidra [29]. Navigating Ghidra's feature set, like its decompiler, control flow graph views, and annotation facilities easily consumes the remaining time in the course.

4 UNGRADING

A thorny problem for a reverse engineering course is that of assessment. The nature of reverse engineering puzzles is that, regardless of a reverse engineer's background and skill level, some puzzles need an appropriate epiphany to solve. An analogy we use is an autostereogram image, more commonly known by the trademarked name "Magic Eye": some people will see the illusory 3D image in the image immediately, others will see it after some time and numerous attempts, and others will need it expressly pointed out to them, and the same is true of solving reverse engineering puzzles. It therefore seems unnecessarily cruel to require students learning



```
Use the "download" command to download the most recently saved
file to your local machine.  "g" starts the emulator running.
Numbers are in base 16; don't prefix them with "$" or "0x".

MAME debugger version 0.226 (unknown)
Currently targeting vic20 (VIC-20 (NTSC))
>g
>mount quik,quik.prg
File quik.prg mounted on quik
```

```
 totalcycles 139551097
A 00          P  ---B---C
X 48          SP (01)f6
Y 07          PC 11c2
```

```
11BB  cmp #$00       C9 00
11BD  bne $11cb      D0 0C
11BF  ldx #$00       A2 00
11C1  dex            CA
11C2 >bne $11c1      D0 FD
11C4  inc $3f        E6 3F
11C6  jsr $1d13      20 13 1D
11C9  beq $11a8      F0 DD
11CB  cmp #$4c       C9 4C
11CD  bne $11db      D0 0C
11CF  lda #$02       A9 02
11D1  sta $1efc      8D FC 1E
11D4  ldx #$00       A2 00
11D6  jsr $1d2d      20 2D 1D
11D9  bmi $11a8      30 CD
11DB  cmp #$4a       C9 4A
```

Memory address: 0

```
0000  4C 48 D2 AA D1 91 D3 22 00 00 00 4C 00 00 00 04   LH....."...L....
0010  00 00 00 00 61 11 19 16 00 0A 76 C3 00 00 00 00   ....a.....v.....
0020  00 00 40 C3 B3 DD 00 00 00 00 00 01 10 00 1E 00   ..@.............
0030  1E 00 1E 00 1E 00 00 00 1E 02 0D 04 01 02 09 72   ...............r
0040  81 BC 17 80 00 FF 15 06 14 01 01 14 09 05 A0 17   ................
0050  F0 01 36 00 00 00 FF 00 00 00 00 00 00 00 00 00   ..6.............
0060  00 8D 00 00 11 61 00 00 00 8D 8A C0 00 00 00 00   .....a..........
0070  00 05 C3 E6 7A D0 02 E6 7B AD 0A 10 C9 3A B0 0A   ....z...{....:..
```

Figure 3: In-browser MAME VIC-20 emulator and debugger

reverse engineering to get the "right" answer in order to receive marks. But how can learning be decoupled from assessment?

A solution comes in the form of *ungrading* [6]. Ungrading does not have a single definition, but refers to practices that reduce or eliminate the primacy of instructor-assigned grades. The argument is that grades are deeply engrained into our education system and their existence there is largely unchallenged, even though they lead to a wide variety of attendant problems [35]. Of particular relevance to our reverse engineering course are three of Stommel's observations [35, p. 28]:

- 'Grades [. . .] incentivize the wrong stuff: the product over the process', yet it is the *process* of reverse engineering that students need to learn.
- 'Grades encourage competitiveness over cooperation', the polar opposite of the course's desired learning environment.
- 'Grades don't reflect the idiosyncratic [. . .] character of learning' and, as already observed, the insight to solve reverse engineering problems is highly individual.

Using ungrading as an underlying philosophy has two highly desirable ramifications in the context of our course. First, there is no penalty for trying and failing, as long as the process (e.g., what a student tried and what they learned) is documented – something that should be present in a science-based discipline anyway. Second, there is no need or benefit to google an answer for a reverse engineering puzzle, because the emphasis is on developing students' reverse engineering thought processes and their individual practice, not about getting the right answer. Ultimately, students are learning to solve a certain kind of puzzle, and as for other types of puzzle, there is no point, no enjoyment, and no sense of accomplishment in completing a puzzle by flipping to the answers in the back of the puzzle book. Ungrading also puts a different spin on getting help from others in the class; it becomes perfectly acceptable, but as with any puzzles, the key is to give hints and not spoilers.

Our course's assessment was in two parts: 50% was based on weekly homework exercises for which the best 10/12 marks were taken; 50% was based on a term project where students were asked to reverse engineer as much of a VIC-20 game as they could. The task for the term project can be easily imagined, with the nontraditional aspect being that the term project mark was based on a self-assessment. Students' ability to perform that judgment hinged in turn on the ungrading-based homework exercises.

The homework exercises were designed to take a limited amount of time to complete, and each was given 'minimal grading' [13, 35] out of two points, where 2/2 was satisfactory, 1/2 needed improvement, and 0/2 was no submission. Some exercise topics involved writing brief reflections on what the students had learned and hoped to learn, as part of mindfully developing their reverse engineering practice. Early reverse engineering puzzles asked students to document the process as opposed to getting the right, or even a complete, answer. For instance, they would be expected to document what they looked for in a puzzle and found, what they looked for and didn't find, and to do so well enough that someone else could follow and reproduce what they did. For assembly programming, in lieu of completing a programming puzzle, students could get full marks for keeping a log documenting their efforts on the program in ten-minute intervals. Later reverse engineering puzzles were structured to help students accurately self-assess their reverse engineering in preparation for their term project self-assessment, and those had a puzzle to do, but where the (two-point) marks were *not* for solving the puzzle, but *for the accuracy of their self-assessment.*[4] Students had three of these self-assessment calibration exercises before having to self-assess their term project.

5 EXPERIENCE

As it did with so many things, the pandemic threw some curve balls into course delivery. With relatively short notice, the term began online and changed to in-person delivery midway through the semester, although we continued streaming (and recording) lectures throughout. The online systems planned for in-class activities made this a relatively painless transition.

Especially considering the non-competitive affordances of ungrading, we hoped to cultivate an in-class culture where students would work together on exercises if they chose to, but unfortunately that did not manifest itself. For the fully online portion, we set up an optional Gather space [18] for students, which allowed students' avatars to "walk" around, and see the screen shares and take part in voice chats of nearby avatars, effectively as close to an in-person experience as could be managed; it was an initial novelty but usage died off rapidly. By the time we were in person, absent the socialization that typically takes place early in a class combined with hesitancy getting physically close to others, there seemed little interest in people working together in class. We also have two physical card games to help teach concepts that were not used for safety reasons, and online versions of them are not ready to deploy yet. Overall, this course seems to have potential for cooperation and physicality that is as yet unrealized. For the second (in-person) offering of the course, we made a concentrated effort to encourage cooperation and community within the classroom while working on puzzles, with better results that could be further improved.

Ungrading worked extremely well until the end of the course. The two-point homework allowed us the time to give meaningful feedback instead of splitting hairs over marks, making assessment enjoyable rather than a chore. We were initially uncomfortable with the idea of a 50% term project being self-assessed, to be honest. And yet, the student-assessed term project marks were tolerably close to what we might have assigned anyway, with few adjustments required (and those were almost always upward, in line with other ungrading experiences [35]). However, a university letter grade needed to be assigned in the end. All methods of calculating grades seemed like equally poor fits to the ungrading philosophy, and we had arbitrarily chosen using letter grades for course components. Unfortunately, much consternation ensued when people calculated their grades as percentages and wondered why they did not receive that grade instead. We have changed to percentages for future offerings to align better with student expectations.

A related consideration that would help in general, as well as the specific case of the term project, is providing further guidance for self-assessment. Recall that students had completed several reverse engineering exercises where they self-assessed their work and received feedback on their self-assessment. We now have complete reverse engineered solutions of past term projects for comparison – not something that was available for the first offering – and we supplied a more detailed rubric for self-assessment, one developed in discussion with the students, that seemed to help students calibrate their own work better and increased the level of transparency. The considerations that should be taken into account, and their relative weight, from the discussion were summarized in a self-assessment template; students would fill this in to gather evidence about their performance and to make a final determination of their grade along with an accompanying justification.

Additionally, the first time, the term project had an optional milestone where students could get early feedback that few took advantage of; this has now been changed to a non-optional milestone with a self-assessment, again to give students more practice with that, again based on the accuracy of their self-assessment. This non-optional milestone was worth 10% of the term project's 50% and, as a higher-stakes component, students could revise their self-assessment based on feedback if their self-assessment was substantially at odds with the supplied evidence. With this change, effectively most students had passed the course prior to the final term project being due, lessening the pressure to complete the challenging (but realistic) reverse engineering task.

Despite messaging for non-majors on the website advertising the course, targeted outreach to student groups and colleagues in non-major fields, and advertising on the university's subreddit, most students in the class were from computer science. This is not a bad thing and perhaps simply indicates not enough available options for computer science students, but also suggests that more might be done to attract non-majors, including more extensive outreach, or possibly having separate sections for majors and non-majors.

6 CONCLUSION

A seemingly technical topic like the reverse engineering of binary code can be taught to a diverse audience. While we have strived to make the course accessible to both computer science majors and non-majors, we have also taken pains not to exclude people in ways that are independent of any disciplinary divide. Stressing a non-competitive, puzzle-based view of reverse engineering as opposed to a competitive security focus is one aspect of this, which is backed up by ungrading. Another aspect is our emphasis on students' own individual practice: there are many ways to solve a reverse engineering problem, and students can select approaches that make the most sense at their different skill levels.

This course has broader implications. For computer security, where there is an enormous demand for trained people in the workforce [42], some of these positions might be filled by people outside traditional fields with appropriate training. Beyond computer security, a similar approach to other topics can help increase diversity in computer science by building bridges to other disciplinary areas. What *else* in computer science can be taught to non-majors?

ACKNOWLEDGMENTS

Thanks to Stuart Madnick for kindly sharing his LMC notes, to Greg Hagen and Joel Reardon for guest lectures, and to Chamila Jayasinghe for VMware Horizon support. A big thank you also to the students in both offerings of CPSC 399.05, and TJ O'Connor for feedback on a draft of this paper.

[4]The obvious way of gaming this, not attempting a puzzle and – accurately – self-assessing poorly to garner full points for the homework, did happen occasionally.

REFERENCES

[1] N. Altice. 2015. *I AM ERROR: The Nintendo Family Computer / Entertainment System Platform.* MIT Press.

[2] J. Aycock. 2021. The coming tsunami of digital artifacts. *Antiquity* 95, 384 (2021), 1584–1589.

[3] J. Aycock, A. Groeneveldt, H. Kroepfl, and T. Copplestone. 2018. Exercises for Teaching Reverse Engineering. In *23rd Annual ACM Conference on Innovation and Technology in Computer Science Education.* 188–193.

[4] A. Berns. 2020. Scored out of 10: Experiences with Binary Grading Across the Curriculum. In *51st ACM Technical Symposium on Computer Science Education.* 1152–1157.

[5] S. D. Blum. 2020. Not Simple but Essential. See [6], 219–228.

[6] S. D. Blum (Ed.). 2020. *Ungrading: Why Rating Students Undermines Learning (and What to Do Instead).* West Virginia University Press.

[7] S. Bradley. 2016. Managing Plagiarism in Programming Assignments with Blended Assessment and Randomisation. In *16th Koli Calling International Conference on Computing Education Research.* 21–30.

[8] S. Bridgeman, M. T. Goodrich, S. G. Kobourov, and R. Tamassia. 2000. SAIL: A System for Generating, Archiving, and Retrieving Specialized Assignments using LaTeX. In *31st SIGCSE Technical Symposium on Computer Science Education.* 300–304.

[9] S. Brown and V. Chávez. 2022. Communicating Alternative Grading Schemes: How to Shift Students' Attention to Their Learning from Grades. In *53rd ACM Technical Symposium on Computer Science Education (V.2).* 1189.

[10] J. Burket, P. Chapman, T. Becker, C. Ganas, and D. Brumley. 2015. Automatic Problem Generation for Capture-the-Flag Competitions. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education.* 8 pages. https://www.usenix.org/system/files/conference/3gse15/3gse15-burket.pdf

[11] S. Cole. 2022. Impact of Capture The Flag (CTF)-style vs. Traditional Exercises in an Introductory Computer Security Class. In *27th ACM Conference on Innovation and Technology in Computer Science Education (Vol. 1).* 470–476.

[12] C. Collberg and J. Nagra. 2010. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley.

[13] P. Elbow. 1997. Grading Student Writing: Making It Simpler, Fairer, Clearer. *New Directions for Teaching and Learning* 1997, 69 (1997), 127–140.

[14] I. Englander and W. Wong. 2021. The Little Man Computer. In *The Architecture of Computer Hardware, Systems Software, and Networking: An Information Technology Approach* (6th ed.). Wiley, Chapter 6, 152–165.

[15] W.-C. Feng. 2015. A Scaffolded, Metamorphic CTF for Reverse Engineering. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education.* 8 pages. https://www.usenix.org/system/files/conference/3gse15/3gse15-feng.pdf

[16] A. Finkel, N. Harris, P. Higginbottom, and M. Tomczyk. 1982. *VIC-20 Programmer's Reference Guide.* Commodore Business Machines and Howard W. Sams & Co.

[17] B. Foster and A. Somayaji. 2010. Object-Level Recombination of Commodity Applications. In *12th Annual Conference on Genetic and Evolutionary Computation.* 957–963.

[18] Gather Presence, Inc. [n. d.]. Gather. Retrieved 29 December 2022 from https://www.gather.town/

[19] C. Gorey. 1 Feb 2018. NASA satellite brought back from the dead will need major reverse engineering. Silicon Republic. https://www.siliconrepublic.com/innovation/nasa-image-satellite-reverse-engineering

[20] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science.* ACM. https://doi.org/10.1145/2534860

[21] Kaitai Project. [n. d.]. Kaitai Struct. Retrieved 29 December 2022 from https://kaitai.io/

[22] M. G. Kirschenbaum. 2008. *Mechanisms: New Media and the Forensic Imagination.* MIT Press.

[23] S. Madnick. 1993. Understanding the Computer (Little Man Computer). Unpublished article, based on 1979 version.

[24] W. Mahoney and R. A. Gandhi. 2012. Reverse Engineering – Is It Art? *ACM Inroads* 3, 1 (2012), 56–61. https://doi.org/10.1145/2077808.2077826

[25] A. Mantovani, S. Aonzo, Y. Fratantonio, and D. Balzarotti. 2022. RE-Mind: A First Look Inside the Mind of a Reverse Engineer. In *31st USENIX Security Symposium.* 2727–2745.

[26] N. Montfort and I. Bogost. 2009. *Racing the Beam: The Atari video computer system.* MIT Press.

[27] Gabriel Moshenska. 2016. Reverse engineering and the archaeology of the modern world. *Forum Kritische Archäologie* 5 (2016), 16–28.

[28] D. S. Myers. 2022. Designing specifications grading systems. *Journal of Computing Sciences in Colleges* 37, 5 (2022), 91–92.

[29] National Security Agency. [n. d.]. Ghidra. Retrieved 29 December 2022 from https://ghidra-sre.org/

[30] TJ OConnor, C. Mann, T. Petersen, I. Thomas, and C. Stricklan. 2022. Toward an Automatic Exploit Generation Competition for an Undergraduate Binary Reverse Engineering Course. In *27th ACM Conference on on Innovation and Technology in Computer Science Education (Vol. 1).* 442–448.

[31] TJ OConnor and C. Stricklan. 2021. Teaching a Hands-On Mobile and Wireless Cybersecurity Course. In *27th ACM Conference on on Innovation and Technology in Computer Science Education (Vol. 1).* 296–302.

[32] G. G. Richard III. 2009. A Highly Immersive Approach to Teaching Reverse Engineering. In *2nd Workshop on Cyber Security Experimentation and Test.* USENIX, 6 pages. http://usenix.org/event/cset09/tech/full_papers/richard.pdf

[33] C. Riesbeck. 2020. Critique-Driven Learning and Assessment. See [6], Chapter 8, 123–139.

[34] J. Scott. 14 April 2015. Behold the Emularity. Retrieved 4 January 2023 from http://ascii.textfiles.com/archives/4604

[35] J. Stommel. 2020. How to Ungrade. See [6], Chapter 1, 25–41.

[36] P. Szor. 2005. *The Art of Computer Virus Research and Defense.* Addison-Wesley.

[37] K. H. Tan and E. L. Ouh. 2021. Lessons Learnt Conducting Capture the Flag CyberSecurity Competition during COVID-19. In *2021 IEEE Frontiers in Education Conference.* 9 pages.

[38] C. Taylor, P. Arias, J. Klopchic, C. Matarazzo, and E. Dube. 2017. CTF: State-of-the-Art and Building the Next Generation. In *2017 USENIX Workshop on Advances in Security Education.* 11 pages.

[39] C. Taylor and C. Collberg. 2016. A Tool for Teaching Reverse Engineering. In *2016 USENIX Workshop on Advances in Security Education.* 8 pages. https://www.usenix.org/system/files/conference/ase16/ase16-paper-taylor.pdf

[40] D. H. Tobey, P. Pusey, and D. L. Burley. 2014. Engaging Learners in Cybersecurity Careers: Lessons from the Launch of the National Cyber League. *ACM Inroads* 5, 1 (2014), 53–56.

[41] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. 2011. Search-Based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 172–186.

[42] U.S. Bureau of Labor Statistics. [n. d.]. Occupational Outlook Handbook: Information Security Analysts. Retrieved 2 January 2023 from https://www.bls.gov/ooh/computer-and-information-technology/information-security-analysts.htm

[43] J. Vykopal, V. Švábenský, and E.-C. Chang. 2020. Benefits and Pitfalls of Using Capture the Flag Games in University Courses. In *51st ACM Technical Symposium on Computer Science Education.* 752–758.

[44] J. Vykopal, V. Švábenský, P. Seda, and P. Čeleda. 2022. Preventing Cheating in Hands-on Lab Assignments. In *53rd ACM Technical Symposium on Computer Science Education (V. 1).* 78–84.

[45] H. Wright and J. Aycock. 2020. 10 Binary Games for Computer Science Education. In *51st ACM Technical Symposium on Computer Science Education.* 1308. Poster.