Toward an Automatic Exploit Generation Competition for an Undergraduate Binary Reverse Engineering Course

TJ OConnor
toconnor@fit.edu
Florida Institute of Technology
Melbourne, FL, USA

Carl Mann
cmann2013@my.fit.edu
Florida Institute of Technology
Melbourne, FL, USA

Tiffanie Petersen
tpetersen2018@my.fit.edu
Florida Institute of Technology
Melbourne, FL, USA

Isaiah Thomas
ithomas2018@my.fit.edu
Florida Institute of Technology
Melbourne, FL, USA

Chris Stricklan
cstricklan@fit.edu
Florida Institute of Technology
Melbourne, FL, USA

ABSTRACT

Analyzing binary programs without source code is critical for cybersecurity professionals. This paper presents an undergraduate binary reverse engineering course design that culminates with a comprehensive binary exploitation competition. Our approach challenges students to develop tools that automatically detect and exploit program vulnerabilities. We hypothesize that this competition presents a unique opportunity to exercise the core competencies of binary reverse engineering. We share our detailed design, labs, experiences, and lessons learned from this course for others to build on our initial success.

CCS CONCEPTS

• **Social and professional topics** → **Model curricula**; • **Software and its engineering** → **Software reverse engineering**.

ACM Reference Format:
TJ OConnor, Carl Mann, Tiffanie Petersen, Isaiah Thomas, and Chris Stricklan. 2022. Toward an Automatic Exploit Generation Competition for an Undergraduate Binary Reverse Engineering Course. In *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol 1 (ITiCSE 2022), July 8–13, 2022, Dublin, Ireland*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3502718.3524744

1 INTRODUCTION

We exist in an era where attackers can rapidly reverse engineer binary code, identify vulnerabilities, and weaponize exploits in days. Such rapid approaches rely heavily on automated reverse engineering tools that detect bugs and suggest possible exploitation approaches. For example, on June 8th, 2021, Microsoft released a routine patch that mitigated several low-risk vulnerabilities [34]. The patch included a fix for a vulnerability in the Windows Print Spooler that enabled a local low-privilege attacker to gain administrator permissions [36]. Within a day of Microsoft's patch, researchers reverse-engineered the code to discover a similar vulnerability. The new vulnerability existed in a nearly identical bug in a different function in the Print Spooler service. By July 1st, cybersecurity researchers published a proof-of-concept that enabled attackers to exploit the bug remotely [37]. Due to the high risk of exploitation in the wild, Microsoft pushed an out-of-band patch within a week [35]. The new vulnerability was given the *PrintNightmare* moniker due to the high risk and exposure. On September 29th, The Cybersecurity and Infrastructure Security Agency (CISA) and the Federal Bureau of Investigation (FBI) alerted that the Conti Ransomware Group was actively exploiting PrintNightmare against victims [14]. Further, on November 3rd, CISA added PrintNightmare to the catalog of vulnerabilities in active exploitation against the federal government [15].

As this previous example shows, analyzing binary code without source code is a critical skill for cybersecurity professionals [4]. The NSA Cyber Operations outcomes and NICE Workforce Competencies both capture Reverse Engineering as a unique knowledge unit [38, 41]. However, only limited works have examined how to teach this skill to students in the classroom [4, 48, 51, 52]. Aycock et al. argue that this is a direct result of the growing de-emphasis on assembly programming in computing disciplines [4]. The 2020 ACM Curriculum Recommendations Computing Discipline further strengthens their argument, failing to introduce any of the core tasks of reverse engineering, including static analysis, dynamic analysis, or symbolic execution. We agree with previous works that argue that we must further examine effective and practical approaches for teaching reverse engineering [4, 48, 51].

The following paper shares our experiences with an undergraduate course and culminating competition that challenged students to develop tools that automatically detect and exploit software vulnerabilities. We hypothesize that developing automatic exploitation tools exercises the core competencies of binary reverse engineering. We acknowledge that developing these tools initially presents a daunting task to undergraduate students. Therefore, we methodically design our course materials to inspire student confidence before approaching this culminating task. This paper makes the following contributions:

(1) We share our course design, lab materials, and lessons learned for an undergraduate reverse engineering course that culminates in an automatic exploit generation competition.

ITiCSE 2022, July 8–13, 2022, Dublin, Ireland

ACM ISBN 978-1-4503-9201-3/22/07…$15.00
https://doi.org/10.1145/3502718.3524744

(2) To allow other instructors to build on our initial success, we publish all code, binaries, and virtualization containers for our labs and competition at https://research.fit.edu/cyber.

Organization: Section 2 investigates previous work on reverse engineering education and motivates the case for auto-exploit generation competitions. In Section 3, we provide an overview of our course design and objectives. Section 4 examines our decisions behind the course infrastructure and labs. Section 5 presents the design, implementation, and student results from the competition. Section 6 offers insight and examines future challenges. Section 7 summarizes our conclusions.

2 BACKGROUND

Our paper advocates for integrating an automatic exploitation competition into a reverse engineering course. As such, we first review previous approaches for teaching reverse engineering. Further, we discuss the DARPA Cyber Grand Challenge results, the first entirely autonomous cyber warfare exercise.

2.1 Reverse Engineering Education Approaches

Previous works [4, 48, 51, 52] have explored approaches for reverse engineering education. These approaches predominately rely on the *Capture-The-Flag* (CTF) challenges [12, 13]. In standard CTF competitions, students race to solve computer security challenges which reveal success in the form of a flag [12]. Burns et al. analyzed 3,600 capture-the-flag challenges over five years (2011-2016) to observe trends [11]. They documented that reverse engineering CTF challenges required students to exercise static and dynamic analysis. The majority of static analysis challenges required students to disassemble Linux ELF binary programs into x86 or AMD64 assembly code and analyze the program's intent. Further, dynamic analysis challenges required the students to step through program execution, observing and setting variables at various breakpoints to alter execution. Aycock et al. presented a series of these challenges to teach reverse engineering that advances students from tool familiarity to analyzing malicious code in sandbox environments [4]. However, Stricklan and O'Connor argued that leveraging CTF challenges in the classroom can lead to unintended cooperation in the era of virtual education and presented an approach for dynamically creating unique binary challenges [51]. To this end, Taylor and Collberg presented an online tool for obfuscating binaries to present unique CTF challenges [17, 52]. However, these approaches have predominately focused exclusively on static or dynamic analysis methods for reverse engineering. In contrast, symbolic execution, as demonstrated by Springer and Feng, offers great promise as a tertiary method for reverse engineering [48]. The next subsection reviews the Darpa Cyber Grand Challenge and its results to understand the benefit of symbolic execution.

2.2 Darpa Cyber Grand Challenge

In August 2016, Darpa held the Cyber Grand Challenge competition at the DEF CON security conference [47]. The contest pitted seven teams against each other to build a cyber reasoning system capable of automatically finding bugs, exploiting the bugs on other teams, and formulating patches on their systems. Over 95 rounds, the competition scored each team on the availability of services, security of services (whether the bug had been exploited on their systems), and evaluation (whether the team successfully exploited the bug on other teams' systems). The winning prize of USD 2,000,000 attracted top academic and industry researchers to develop novel approaches for exploit development.

The winning team ForAllSecure, closely aligned with Carnegie Melon University, demonstrated the practicality of concolic execution (the combination of symbolic execution with concrete execution) by generating fuzzing test cases to detect bugs [3]. Their success also largely depended on their patching strategy, which leveraged a Bayesian classifier to make reactive decisions based on the probability of another team's ability to develop an exploit [3]. Even in the crash of their exploit-throwing framework halfway through the competition, this strategy succeeded. Shellphish, a highly-competitive capture-the-flag team associated with UC Santa Barbara, earned the third-place victory with the development of several binary analysis tools [44]. Their work, which they open-sourced as the *angr* Framework [46, 50, 58] successfully detected and exploited the most vulnerabilities against other teams. The success of both teams demonstrated the promise of symbolic execution as a new analysis method for reverse engineers. Open-sourcing angr has produced numerous derivative works that showcase the power of symbolic and concolic execution [18, 19, 29, 59, 60]. However, it has seen limited adoption by cybersecurity curriculum. Both the NSA Cyber Center of Excellence Knowledge Units and NICE Cybersecurity Workforce Skills only focus on traditional static and dynamic reverse engineering methods for analysis [38, 41]. However, Springer and Feng [48] provided an initial approach for introducing symbolic execution into the curriculum, challenging students to solve capture-the-flag problems with angr.

3 COURSE OVERVIEW

This section describes the course model for our undergraduate binary reverse engineering course. The course meets twice weekly for 75 minutes each class over 16 weeks. It is the fourth course in a six-course sequence for our cyber operations concentration. Our students are pursuing a variety of computing discipline degrees. The course prerequisites include an introductory cybersecurity course and an assembly programming course. We designed the course to satisfy the mandatory NSA CAE Cyber Operations Knowledge Units for reverse engineering. Based on this standard, the student outcomes include the following:

(1) Use industry tools to safely perform static and dynamic analysis of software of unknown origin, including obfuscation, to fully understand the software's functionality.
(2) Analyze binaries using reverse engineering tools including disassembling, debugging and virtualizing software in sandbox environments.
(3) Describe the techniques specifying program behavior, the classes of well-known defects, and how they manifest themselves in various languages.

The course topics, listed in Table 1, graduate students from the basics of binary analysis to exploitation. We follow a *lecture with labs* model where we balance theoretical lessons with practical labs.

Table 1: The course balances theory-based lessons with practical labs and contemporary research.

Lesson	References
AMD64 Assembly Refresher	[30, 32]
Dynamic Analysis Approaches & Tools	[20, 22, 49]
Static Analysis Approaches & Tools	[21, 43]
ELF Executable File Format	[32, 54]
Automating Binary Analysis	[6–8]
Binary Obfuscation	[16, 17, 27, 28, 52]
Anti Reverse Engineering Techniques	[56]
Reverse Engineering Network Functions	[55]
Symbolic Execution	[23, 44, 46, 50, 58]
ARM64 Architecture	[2, 31]
Reverse Engineering Windows Binaries	[25, 33, 42]
Binary Exploitation	[1, 26, 53]

In the following section, we expand on the design of these experiential learning labs. We provide students with reference materials from a broad array of sources. As mentioned in [4], the topic of reverse engineering lacks the traditional computing discipline resources. As recommended in [9], our materials span across technical references, industry-standard documents, academic publications, contemporary reverse engineer videos, blog posts, and security conference presentations. We carefully chose these nontraditional materials. In particular, we observed students enjoyed the technical content from *LiveOverflow*, a Google-sponsored researcher who produces reverse-engineering videos on YouTube [20–22].

4 COURSE IMPLEMENTATION

The following section describes the infrastructure and labs for the course. Specifically, we examine how the labs exercised the core competencies of reverse engineering and prepared the students for the culminating competition.

4.1 Course Infrastructure

We hosted our labs and associated services on the CTFd [13] capture-the-flag platform. CTFd provides an accessible user interface to host binaries, challenges and provide immediate feedback to students and instructors [13]. This approach also benefited remote students as our university operated in a hybrid model due to the current pandemic. To reduce the impact on our IT staff, we leased a CTFd server that hosted our challenges and services for USD 20 per month. Further, we purchased personal (academic) licenses of the Binary Ninja [57] reverse engineering platform for students at the cost of USD 79 per student. Although we prefer the user experience of Binary Ninja, we concede the freely available Ghidra [43] offers similar debugging, plugin, and intermediate language support and can achieve the same outcomes. Additionally, Hex-Rays offers free educational licenses for the IDA decompiler for x86, AMD64, ARM and ARM64 architectures [24]. We also provided students with a Docker image to standardize student environments and analytical tools. As we discuss in Section 6, standardizing environments reduces the troubleshooting surface area for instructors. The following paragraphs expand on the specific labs.

4.2 Course Labs

The practice-based experiential labs in the following subsections graduate students from the basics of binary analysis to exploitation. We gradually increased lab difficulty during the course. Further, we isolate the outcomes for each lab to emphasize a unique competency of reverse engineering, including dynamic analysis, static analysis, analysis automation, symbolic execution, and reversing different architectures.

4.3 Dynamic Analysis Lab: Bomb-Lab

Our first lab, *the bomb-lab*, introduced students to dynamic analysis. We based the bomb-lab on a traditional computer science program, which familiarizes students with the basics of debugging [10]. Our lab consisted of five levels, each that requested specific user input to successfully *disarm the bomb*. We carefully crafted each level to introduce students to the concepts of calling convention, memory management, recognizing programming constructs, and tracing the program execution. We primed the lab in the prior lecture, reviewing these concepts. We found that the distribution mechanism of the lab proved critical to producing student outcomes. Modern binary analysis frameworks (Binary Ninja, Ghidra, Ida) all de-compile binaries to high-level intermediate language (HLIL). This representation, which represents pseudocode of the source code, defeats the lab outcome of recognizing program constructs from assembly code. As such, we changed the distribution to a web-based interface and did not distribute the binary. This web interface, which provided students with a command-line gdb session, protected students from bypassing the outcomes. Further, we challenged students to solve all the levels to earn their *Binary Ninja* license. With this additional motivation, we observed that all the students solved the lab within the class-allotted time.

4.4 Static Analysis Lab: Math Problems

Our next lab introduced students to static analysis. Modern decompilers present a challenge for reverse engineering educators. Reversing a compiled program to a high-level representation was previously a rite-of-passage for reverse engineering students. This task requires a painstaking but necessary student effort to trace the assembly code and maintain the state of the program stack and memory registers (when possible). However, we must approach this problem differently in an era of modern decompilers. Previous works have addressed this challenge by adding substantial complexity to the problem with obfuscated code [16]. Such approaches rely on camouflaging the code's intent with jump tables, opaque predicates, and unreachable blocks. We shared concerns about prematurely introducing complexity and obfuscation too early to undergraduate students. Ultimately, we settled on a partial approach by developing problems that would force students to understand the underlying instructions. This approach required students to trace the program memory through a combination of *add, shl, sub, imul, pop*, and *mov* instructions. However, the decompiler resolves the binary code to a representation similar to the source.

*return zx.q((((arg1 + 0x65) « 8) - (arg1 + 0x65)) * 0x909090)*

While the decompiler's representation offers some clarity, we added minor complexity to the problem. Specifically, we made the problem only solvable via an integer overflow. This approach forced students to trace through the instructions, identify data types, and understand how the specific math operations handle results that exceed the data type.

4.5 Automation Lab: Networking Annotation

Next, we introduced students to automating the analysis of assembly code. Specifically, we asked the students to develop a plugin that annotated the networking calls from a disassembled binary. We directed students to use the Binary Ninja reverse engineering platform to add comments when a binary program made a networking call via a wrapper function or system call. Listing 1 shows a successful student solution that labels the Linux system call (0x31) as a socket *bind()* call and parses the IP, Port, and address length from memory registers. The Binary Ninja API provides a powerful function that resolves memory register values at a program address. Students leveraged this powerful feature to determine the parameters for each networking call. To verify student solutions, we generated twelve Linux *Metasploit Meterpreter* binaries. This dataset of binaries proved ideal as it included both wrapper and system calls for networking functions across TCP, HTTP, and ICMP channels. We found that this lab served as an excellent bridge between the analysis and automation aspects of reverse engineering. Further, we challenged students to use documentation, specifically the Linux man-pages, to determine how to resolve internet addresses from their stored reverse network byte ordering. We observed that this lab introduced greater anxiety and student apprehension than previous labs. Despite these observations, we argue that this lab proved an essential element of our design as the culminating exploitation competition requires substantial automation from students. In future course offerings, we intend to introduce the Binary Ninja API earlier in the course materials to reduce student anxiety about interfacing with a new API.

4.6 Symbolic Execution Lab: AngrY Challenges

Halfway through the course, we introduced symbolic execution as an analysis method. We hypothesize that introducing this concept too early in a reverse engineering class presents challenges. Like providing a calculator while initially learning addition, the tool could prevent student growth. Waiting until the midway point allowed us to ensure students could understand the *how* instead of just applying the tool. Our *AngrY Challenges* lab introduced students to symbolic execution as a tertiary analysis method for reverse engineering. In a similar approach to [48], we presented students with complex capture-the-flag binary programs. Each program prompted the user for an input and then ran that input through a series of permutations. The program then checked the final permutation against a pre-calculated value. When the user entered the correct input, the program outputted a flag. Instead of tracing or debugging the complex series of permutations, we encouraged students to leverage symbolic execution as a mechanism for solving the problem. Our previous in-class lectures demonstrated the theory behind symbolic execution and introduced the *angr* binary analysis framework [58]. While students generally succeeded at this lab, we believe we missed an opportunity to explore complex use-cases for symbolic execution. Leveraging minimal knowledge of the input constraints and conditional branches, students developed angr solution scripts that symbolically solved valid inputs. We intend to have students explore how to model memory registers and identify unconstrained program execution paths in future class offerings. We believe this misstep partially contributed to the limited adoption of symbolic execution in the culminating competition.

Listing 1: Example student results that annotated the networking calls of a de-compiled binary.

```
00400084  4897              xchg    rdi, rax
00400086  52                push    rdx
00400087  c7042402007a69    mov     dword [rsp], 0x697a0002
0040008e  4889e6            mov     rsi, rsp
00400091  6a10              push    0x10
00400093  5a                pop     rdx
00400094  6a31              push    0x31
00400096  58                pop     rax
00400097  0f05              syscall
// bind(fd=<RV>, sockaddr=[IP=0.0.0.0, Port=31337], addrlen=16)
```

4.7 Other Architectures Lab: ARM Challenges

We presented the students with similar CTF problems to the previous labs in the final lab but compiled the programs as ARM64 binaries. ARM64, also known as ARMv8, is a 64-bit reduced instruction set architecture. Smartphones and Internet-of-Things (IoT) devices commonly leverage ARM64 processors due to the lower power utilization. We lectured students on the instruction set in the corresponding lesson, highlighting the unique differences between AMD64 and ARM64 architectures. We illustrated the differences between the calling conventions, memory registers, page tables, and system calls. However, we purposely spent minimal time demonstrating *how to use tools* to statically, dynamically, or symbolically analyze ARM64 binaries. Instead, we leveraged this opportunity to challenge students to synthesize their understanding of reverse engineering. By presenting students a different architecture, we challenged them to demonstrate their ability to develop architecture-independent reverse engineering analysis approaches. This approach challenged the students to compose solutions rather than walk through choreographed debugging steps. We observed that student challenges revolved around creating debugging environments. While Docker can develop containers for different architectures (ARM, MIPS), it does not fully implement the *ptrace* capability required to run a debugger. After some significant student struggles, we provided our students with a Docker image that allowed *Qemu user-mode* to execute and debug ARMv8 binaries.

5 EXPLOIT GENERATION COMPETITION

5.1 Overview

The automatic exploit competition, which served as the final exam, allowed students to synthesize reverse engineering concepts by exploring binary exploitation. We challenged students to write a tool that automatically detected and exploited program vulnerabilities.

Listing 2: Example of a student solution that leveraged dynamic analysis, static analysis, and symbolic execution to develop an exploit.

```
# dynamic analysis: detect buffer overflow from coredump
padding = cyclic_find(core.read(core.rsp, 8), n=8)

# static analysis: resolve execve() call from symbol table
execve = p64(self.elf.sym['execve'])

# symbolic execution: construct gadgets to populate parameters
angr_p = angr.Project(self.file)
angr_rop = angr_p.analyses.ROP()
angr_rop.find_gadgets_single_threaded()
chain = angr_rop.write_to_mem(data_section, b"/bin/sh\x00")
chain += angr_rop.set_regs(rdi=data_section, rsi=0, rdx=0)

# exploit is constructed from all analysis methods
self.exploit = b'a'*padding + chain.payload_str() + execve
```

To balance the potential negative impacts of competition [5, 45], we divided the students into three balanced teams for the competition. We provided students with an initial sample of ten vulnerable binaries. The competition tested the teams' ability to automatically analyze and exploit the original binaries and an additional 85 vulnerable binaries. As proof of vulnerability (PoV), students had to redirect program execution to display the contents of a flag file. The following section discusses the classes of vulnerabilities in the programs.

5.2 Bug Classes and Exploit Techniques

We limited the scope of vulnerabilities to stack overflows and format string specifier vulnerabilities, most commonly found in beginner CTF competitions [11]. We avoided more complex vulnerability types, including heap and type confusion bugs, introduced later in our concentration. We compiled the 64-bit Linux ELF binaries with non-executable (NX) stack protection to require students to leverage multiple exploitation techniques. However, we did not enable stack canaries, address space layout randomization (ASLR), or full relocation read-only (RELRO), which would require more complex exploitation techniques. Additionally, we left the symbol tables in the binary and provided students with a list of standardized variable and function names. As an example, twelve binaries contained a *win()* function that would display a flag if called. We crafted the program vulnerabilities such that individual solutions might require multiple exploitation techniques to succeed. For example, a sample program could contain a stack-based buffer overflow and a format string specifier vulnerability. The format string specifier vulnerability might allow leaking the randomized base address of the libc library. Subsequently, the stack buffer overflow can be leveraged into overwriting the return pointer to redirect execution into the one-gadget return-oriented programming (ROP) chain in the libc library. The following list contains the scope of vulnerabilities and exploitation techniques required for students to succeed.

Stack-Based Buffer Overflows:

(1) ret2win
(2) ret2execve
(3) ret2syscall
(4) ret2libc/one-gadget
(5) rop parameter-passing gadgets
(6) rop write-what-where gadgets

Format String Specifier Vulnerabilities :

(1) arbitrary read of variable
(2) arbitrary read of libc address
(3) arbitrary write variable primtive
(4) arbitrary write of global offset table

5.3 Competition Environment

We provided students with a Docker container that standardized the competition environment. The environment contained a limited set of additional tools, including the pwntools exploit development library, one_gadget, ROPGadget, ropper, unicorn, capstone, z3-solver, qiling, gdb, pwndbg, and angr. We also allowed students to submit patches to the environment if their solution required additional tooling. Initially, we implemented this environment on an Ubuntu 20.20 container. However, the default GNU Libc Library (GLIBC) in Ubuntu 20.20 implements protection that requires a 16-bit aligned stack before a call instruction. After listening to initial student concerns about the complexity introduced by this protection, we changed to a Kali container with a Libc version that did not implement this protection. While we installed angr, we did not explicitly install the additional *angrop* tool for using symbolic execution to construct ROP chains. We believe this misstep partially contributed to the limited adoption of static analysis in student solutions. As the winning team identified, *angrop* provides a powerful mechanism to identify ROP gadgets for setting registers and write-primitives.

5.4 Student Results

Table 2 reports the student results of the competition. Student solutions far exceeded our expectations for the initial course offering. While students had initial reservations about the complexity of the problem, they showed great enthusiasm and excitement in class. Notably, students enjoyed running the competition live, watching their solutions exploit binaries and display flags. The winning team leveraged a combination of static analysis, dynamic analysis, and symbolic execution to identify and exploit over 80% of the vulnerable programs. Listing 2 depicts a snippet of their solution that demonstrates this approach for solving a stack-based buffer overflow via a ret2execve exploit technique. First, the students dynamically sent incrementally sized input to detect buffer overflows until a segmentation fault produced a core dump file that overwrites the stack. Next, they statically analyzed the symbol table to see if it contained an address for the *execve* call to execute a program. Finally, they leveraged symbolic execution to identify a write primitive and set memory registers for the call to *execve("/bin/sh",0,0)*. The two other teams followed similar approaches but failed to leverage symbolic execution to identify gadgets. Rather than leveraging symbolic execution, they used the *pwnlib.rop.rop* library from pwntools to statically identify ROP chains for setting memory registers

Table 2: The competition results, indicating that students adopted a variety of techniques for exploiting the binaries.

Team	Static Analysis	Dynamic Analysis	Symbolic Execution	Successfully Exploited
SADD3R	✓	✓	-	40.0 % (38)
Oh Man!	✓	✓	-	54.7 % (52)
Solar Panth3r	✓	✓	✓	82.1 % (78)

and write primitives. This limited approach yielded success for the initial challenges. However, our competition binaries required more complex ROP chains, requiring a symbolic execution approach. The winning team's only limiting factor was identifying functions to overwrite in the global offset table (GOT). After the competition (during the holiday break), the students enthusiastically submitted a patch that successfully exploited all the vulnerable binaries.

6 LESSONS LEARNED

6.1 Successes

Students Enthusiasm & Confidence: A key insight of our work is that binary exploitation offers an exciting framework to engage reverse engineering students. Before the final exam, students expressed informal comments suggesting their uncertainty and anxiety about writing an automatic exploit generation tool. Only limited students had previous exposure to beginner CTF binary exploitation challenges, favoring traditional reverse engineering or forensics CTF challenges. While we extensively primed the students for success, we also feared the competition complexity would challenge students. However, binary exploitation excited students to learn new tools and explore different analysis methods. We watched as students gained enthusiasm for preparing their tools. We observed that the cooperative approach yielded success as students felt comfortable scoping a solution to a particular element. Individual students focused on different aspects of the problem, including developing format string specifier harnesses or ROP chain constructors. Further, we observed that students took pride in their solutions, publishing them to GitHub and developing them for future CTF competitions. Notably, the winning team expanded their tool to exploit 100% of the competition binaries.

Standardizing with Docker: We observed throughout the course that standardizing our lab and competition environments on Docker virtualization platforms offered substantial benefits to both instructors and students. Installing reverse engineering tools and frameworks proves an often frustrating task of chasing library dependencies while managing program version conflicts. Further, differences between the instructor workstation and students' layouts can substantially increase student anxiety about an already complex topic. To this end, we discovered that baselining students on the same Docker image for lessons and labs significantly reduced this frustration and stress. In previous offerings, we distributed virtual course machines in OVF format. These images often exceed several GiB of memory and occasionally require an update. In contrast, we found that we could update and redistribute Dockerfiles as small text-based files. These Dockerfiles allowed students to rapidly update their environments to support a new tool, as in the ARM64 lab. Further, we could also quickly identify an issue in a student's solution when their workstations mirrored our environments.

6.2 Challenges

Greater Adoption of Symbolic Execution: We will emphasize symbolic execution as a tertiary analysis method in future course offerings. Reviewing the competition results, we realized the AngrY Lab challenges did not provide students with enough context to leverage symbolic execution. We believe that a more substantial introduction to symbolic execution would result in greater adoption in the competition. To this end, we identified opportunities for holistic integration into the existing course materials. For example, we could present angr's *claripy* as an alternate approach to the Math Challenges Lab. Further, we could use *angr* to identify opaque predicates during our code-obfuscation topic. Additionally, we must expand the AngrY Lab challenges to introduce modeling unconstrained paths and memory write primitives. Further, we intend to provide students with an updated Docker image containing more symbolic execution tools, including *angrop* and usage examples.

Ethical Concerns: As we have discussed in previous works [39, 40, 51], we incorporate a significant examination of professional behaviors, legal ramifications of abuse, and ethics into our concentration. Initial coursework discusses the Computer and Fraud Abuse Act, Electronic Communications Privacy Act, the Digital Millennium Copyright Act, and our university's acceptable use policy. While we require students to resign an ethics contract every semester, we believe our most powerful method to prevent abuse is frequent deliberate dialogues with students.

Formal Evaluation: Our work examines the initial experience report of the first iteration of our course. We did not capture user surveys before or during the course. We acknowledge that a formal evaluation would benefit a comprehensive analysis of the proposed methods. We plan to conduct evaluations in future works, specifically examining the balance between confidence and anxiety in the automatic exploit generation competition.

7 CONCLUSION

In this paper, we presented our undergraduate course on binary reverse engineering. Our approach balances theory-based instruction with experiential learning. The course culminates with a binary exploitation competition, which challenges students to develop tools that automatically detect and exploit program vulnerabilities. We argue that a binary exploitation competition provides a unique opportunity to exercise modern reverse engineering techniques. We share our detailed design, labs, experiences, and lessons learned from this course for others to build on our initial success.

ACKNOWLEDGEMENTS

This material is based upon work supported in whole or in part with funding from the Office of Naval Research (ONR) contract #N00014-21-1-2732. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the ONR and/or any agency or entity of the United States Government.

REFERENCES

[1] Aleph One. 1996. Phrack: Smashing the stack for fun and profit. http://www.phrack.com/issues/49/14.html

[2] Arm Limited. 2020. Arm Architecture Reference Manual Armv8. https://developer.arm.com/documentation/ddi0487/gb/

[3] Thanassis Avgerinos, David Brumley, John Davis, Ryan Goulden, Tyler Nighswander, Alex Rebert, and Ned Williamson. 2018. The mayhem cyber reasoning system. *IEEE Security & Privacy* 16, 2 (2018), 52–60.

[4] John Aycock, Andrew Groeneveldt, Hayden Kroepfl, and Tara Copplestone. 2018. Exercises for teaching reverse engineering. In *Conference on Innovation and Technology in Computer Science Education*. ACM, Larnaca Cyprus, 188–193.

[5] César Morillas Barrio, Mario Muñoz-Organero, and Joaquín Sánchez Soriano. 2015. Can gamification improve the benefits of student response systems in learning? An experimental study. In *IEEE Transactions on Emerging Topics in Computing*, Vol. 4.3. IEEE, Piscataway, NJ, 429–438.

[6] Binary Ninja. 2021. Using Plugins. https://docs.binary.ninja/guide/plugins.html

[7] Binary Ninja. 2021. Binary Ninja Documentation: Working with Types, Structures, and Symbols. https://docs.binary.ninja/guide/type.html

[8] Tim Blazytko. 2021. Automation in Reverse Engineering: String Decryption. https://synthesis.to/2021/06/30/automating_string_decryption.html

[9] Sergey Bratus. 2007. What hackers learn that the rest of us don't: notes on hacker curriculum. *IEEE Security & Privacy* 5, 4 (2007), 72–75.

[10] Logan Brown, Gavin Hayes, and Tejas Rao. 2017. Reinventing Bomblab. Reinventing Bomblab. https://digital.wpi.edu/downloads/s7526g02j

[11] Tanner J Burns, Samuel C Rios, Thomas K Jordan, Qijun Gu, and Trevor Underwood. 2017. Analysis and Exercises for Engaging Beginners in Online CTF Competitions for Security Education. In *2017 USENIX Workshop on Advances in Security Education (ASE 17)*. USENIX, Vancouver, BC, Canada, 9 pages.

[12] Peter Chapman, Jonathan Burket, and David Brumley. 2014. PicoCTF: A game-based computer security competition for high school students. In *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*. USENIX, San Diego, CA, 10 pages.

[13] Kevin Chung. 2017. Live Lesson: Lowering the Barriers to Capture The Flag Administration and Participation. In *2017 USENIX Workshop on Advances in Security Education (ASE 17)*. USENIX, Vancouver, BC, Canada, 6 pages.

[14] CISA. 2021. Joint Cybersecurity Advisory: Conti Ransomware.

[15] CISA. 2021. Known Exploited Vulnerabilities Catalog.. https://www.cisa.gov/known-exploited-vulnerabilities-catalog

[16] Christian Collberg. 2018. Code obfuscation: Why is this still a thing?. In *Conference on Data and Application Security and Privacy*. ACM, Tempe,AZ, 173–174.

[17] Christian Collberg. 2021. The Tigress C Obfuscator. https://tigress.wtf/

[18] Shruti Dixit, TK Geethna, Swaminathan Jayaraman, and Vipin Pavithran. 2021. AngErza: Automated Exploit Generation. In *Conference on Computing Communication and Networking Technologies (ICCCNT)*. IEEE, West Bengal, India, 1–6.

[19] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. 2019. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries.. In *NDSS*. Internet Society, San Diego, CA, 15 pages.

[20] Fabian Faessler. 2015. LiveOverflow: Reversing and Cracking First Simple Program - Bin 0x05. https://www.youtube.com/watch?v=VroEiMOJPm8

[21] Fabian Faessler. 2016. LiveOverflow: Simple Tools and Techniques for Reversing a Binary - Bin 0x06. https://www.youtube.com/watch?v=3NTXFUxcKPc

[22] Fabian Faessler. 2019. LiveOverflow: Patching Binaries. https://www.youtube.com/watch?v=LyNyf3UM9Yc

[23] John Hammond. 2020. Google CTF - Beginner Reverse Engineering with Angr. https://www.youtube.com/watch?v=RCgEIBfnTEI

[24] Hex Rays. 2022. IDA Educational Licenses. https://hex-rays.com/educational/

[25] Mateusz Jurczyk. 2020. Windows System Call Tables. https://github.com/j00ru/windows-syscalls

[26] Max Kamper. 2021. ROP Emporium. https://ropemporium.com

[27] Peter LaFosse. 2017. Automating Opaque Predicate Removal. https://binary.ninja/2017/10/01/automated-opaque-predicate-removal.html

[28] Xusheng Li. 2021. Winning The Grand Reverse Engineering Challenge. https://binary.ninja/2021/09/02/winning-the-grand-re-challenge.html

[29] Danjun Liu, Jingyuan Wang, Zelin Rong, Xianya Mi, Fangyu Gai, Yong Tang, and Baosheng Wang. 2018. Pangr: A Behavior-Based Automatic Vulnerability Detection and Exploitation Framework. In *Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering*. IEEE, New York, NY, 705–712.

[30] Chris Lomont. 2009. Introduction to x64 Assembly - Intel. https://www.intel.com/content/dam/develop/external/us/en/documents/introduction-to-x64-assembly-181178.pdf

[31] Maria Markstedter. 2020. Introduction to ARM Assembly Basics. https://azeria-labs.com/writing-arm-assembly-part-1/

[32] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2013. System V ABI. https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf

[33] Microsoft. 2021. Visual Studio 2019: x64 Calling Convention. https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-160

[34] Microsoft. 2021. MSRC Customer Guidance Security Update Gude: Vulnerability CVE-2021-1675. Windows Print Spooler Remote Code Execution Vulnerability.. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-1675

[35] Microsoft. 2021. MSRC Customer Guidance Security Update Gude: Vulnerability CVE-2021-34527. Windows Print Spooler Remote Code Execution Vulnerability.. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-34527

[36] MITRE. 2020. CVE-2021-1675. Available from MITRE, CVE-ID CVE-2021-1675.. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-1675

[37] MITRE. 2021. CVE-2021-34527. Available from MITRE, CVE-ID CVE-2021-34527.. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-34527

[38] NSA. 2022. Academic Requirements for Designation as a CAE in Cyber Operations Fundamental. https://www.nsa.gov/Resources/Students-Educators/centers-academic-excellence/cae-co-fundamental/requirements/

[39] TJ OConnor. 2022. HELO DarkSide: Breaking Free From Katas and Embracing the Adversarial Mindset in Cybersecurity Education. In *Special Interest Group on Cyber Security Education (SIGCSE)*. ACM, Virtual Event.

[40] TJ OConnor and Chris Stricklan. 2021. Teaching a Hands-On Mobile and Wireless Cybersecurity Course. In *Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, Virtual Event, 296–302.

[41] Rodney Petersen, Danielle Santos, Matthew Smith, and Gregory Witte. 2020. Workforce Framework for Cybersecurity (NICE Framework).

[42] Matt Pietrek. 2002. An In-Depth Look into the Win32 Portable Executable File Format. https://docs.microsoft.com/en-us/archive/msdn-magazine/2002/february/inside-windows-win32-portable-executable-file-format-in-detail

[43] Roman Rohleder. 2019. Hands-on ghidra-a tutorial about the software reverse engineering framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*. ACM, London,UK, 77–78.

[44] Team Shellphish. 2017. Phrack: Cyber Grand Shellphish. http://www.phrack.org/issues/70/4.html#article

[45] Wei-Cheng Milton Shen, De Liu, Radhika Santhanam, and Dorla A Evans. 2016. Gamified technology-mediated learning: The role of individual differences. In *Pacific Asia Conference on Information Systems (PACIS)*. Association For Information System, Chiayi, Taiwan, 18 pages.

[46] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, 138–157.

[47] Jia Song and Jim Alves-Foss. 2015. The darpa cyber grand challenge: A competitor's perspective. *IEEE Security & Privacy* 13, 6 (2015), 72–76.

[48] Jacob Springer and Wu-chang Feng. 2018. Teaching with angr: A Symbolic Execution Curriculum and {CTF}. In *2018 Workshop on Advances in Security Education (ASE 18)*. USENIX, Baltimore, MD, 8 pages.

[49] Richard Stallman, Roland Pesch, and Stan Shebs. 2021. Debugging with GDB. https://sourceware.org/gdb/current/onlinedocs/gdb/index.html

[50] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*. Internet Society, San Diego, CA, 16 pages.

[51] Chris Stricklan and TJ OConnor. 2021. Towards Binary Diversified Challenges For A Hands-On Reverse Engineering Course. In *Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, Virtual Event, 102–107.

[52] Clark Taylor and Christian Colberg. 2016. A tool for teaching reverse engineering. In *Workshop on Advances in Security Education*. USENIX, Vancouver, BC, 8 pages.

[53] Team Teso. 2001. Exploiting Format String Vulnerabilities. http://www.madchat.fr/coding/c/c.seku/format_string/formatstring.pdf

[54] TIS Committee. 1993. Tool Interface Standard (TIS): Portable Formats Specification Version 1.1. http://refspecs.linux-foundation.org/elf/TIS1.1.pdf

[55] Linus Torvalds. 2021. The Linux Kernel: Linux Networking. https://linux-kernel-labs.github.io/refs/heads/master/labs/networking.html

[56] Yu-Jye Tung. 2021. Analysis of Anti-Analysis. https://github.com/yellowbyte/analysis-of-anti-analysis/

[57] Vector35. 2022. Binary Ninja. https://binary.ninja

[58] Fish Wang and Yan Shoshitaishvili. 2017. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development*. IEEE, Cambridge, MA, 8–9.

[59] Shenglin Xu, Peidai Xie, and Yongjun Wang. 2020. AT-ROP: Using static analysis and binary patch technology to defend against ROP attacks based on return instruction. In *International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, Shanghai, CN, 209–216.

[60] Yao Yao, Wei Zhou, Yan Jia, Lipeng Zhu, Peng Liu, and Yuqing Zhang. 2019. Identifying privilege separation vulnerabilities in IoT firmware with symbolic execution. In *European Symposium on Research in Computer Security*. Springer, Luxembourg, 638–657.