Towards Binary Diversified Challenges For A Hands-On Reverse Engineering Course

Christopher Stricklan, TJ OConnor
{cstricklan,toconnor}@fit.edu
Florida Institute of Technology

ABSTRACT

The balance of a practical hands-on and theoretical approach for reverse engineering coursework offers a strong approach for cybersecurity education. This balance is key to helping students build the skills necessary to contribute to the industry upon graduation. However, the remote learning demands of the current pandemic present a challenge to this approach. Inappropriate collaboration between students poses a threat to the educational benefits of practice-based learning. Specifically, inappropriate collaboration can threaten the development of critical problem skills gained during individual work. Further, relying on instructors to create unique challenges for each student fails to scale. To overcome these challenges, we have implemented a binary diversification system that produces unique reverse engineering challenges per student. In this paper, we present the technical details and lessons learned implementing this approach. We believe that sharing our approach will benefit cybersecurity education instructors looking to overcome the challenges of remote-learning cybersecurity coursework.

CCS CONCEPTS

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

ACM Reference Format:
Christopher Stricklan, TJ OConnor. 2021. Towards Binary Diversified Challenges For A Hands-On Reverse Engineering Course. In *26th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2021), June 26-July 1, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3430665.3456358

1 INTRODUCTION

In this paper, we share a description of the design of a binary diversification build system for use in reverse engineering (RE) course education. Our goal is to demonstrate a balanced approach to mitigate student collaboration while not being cumbersome to implement. The RE course is hands-on and emphasizes the building of a workflow for analyzing binaries that scale in difficulty. During this course, we strive to present the values, practices, and approaches of a hacker focused curriculum [4].

ITiCSE 2021, June 26-July 1, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8214-4/21/06…$15.00
https://doi.org/10.1145/3430665.3456358

Our approach leverages a jeopardy-based capture-the-flag (CTF) competition. This approach provides a valuable tie between security concepts while presenting an exemplar environment for their application [8]. This teaching approach of a hands-on RE course becomes challenging due to the necessity and demand for novel challenges each semester. The presented CMake build system facilitates creating a set of diverse binaries to alleviate this demand. In addition we have employed this as a framework to elicit student created challenges for a class head-to-head CTF. This provides the students with another view of the RE process; requiring them to evaluate their challenges to prove they apply a valid and appropriate RE workflow.

Our university implemented a hybrid learning environment during the recent novel coronavirus, with students participating remotely and in-class. Recent anecdotes of remote-learning cheating scandals at the US Military and Naval Academies demonstrate the problem of inappropriate collaboration[5, 10]. Unintended and inappropriate student collaboration eliminates the necessary struggle to develop the critical problem solving skills in reverse engineering. Instructor-produced challenges cannot be distributed to multiple students or reused in different semesters without the risk of a student inappropriately using the solution of a classmate. Once challenges and solutions are given out as soft copies we have little control over where they end up and how they are used. Our challenges and their solutions are dumped into the wild. As a smaller university, we do not have the faculty available to create a corpus of challenges for one-time use.

Using the solution presented in this paper offers control over the type of RE challenges we wish to create while being randomly modified for an added anti-cheat effect. Our binary diversity approach allows students to study past binaries, but not copy solutions requiring them to still follow through using the skill sets presented during the RE course. In the end we can manage the level of difficulty of each binary, provide an enjoyable learning experience, and minimize the time to generate the supporting meta-files. This paper makes the following contributions:

(1) We share our experiences, lessons learned, and materials for embracing a take home RE examination with a level of confidence of limiting student collaboration.
(2) We present a means to create binary diversity and supported meta-files for reverse engineering education.

Organization: Section 2 investigates previous work related to binary diversification in learning and CTF focused environments. Section 3 gives an overview of our build system. In Section 4, we provide a detailed description of our approach of using CMake as methodology for binary diversification. Section 5 offers insight and examines future challenges. Section 6 summarizes our conclusions.

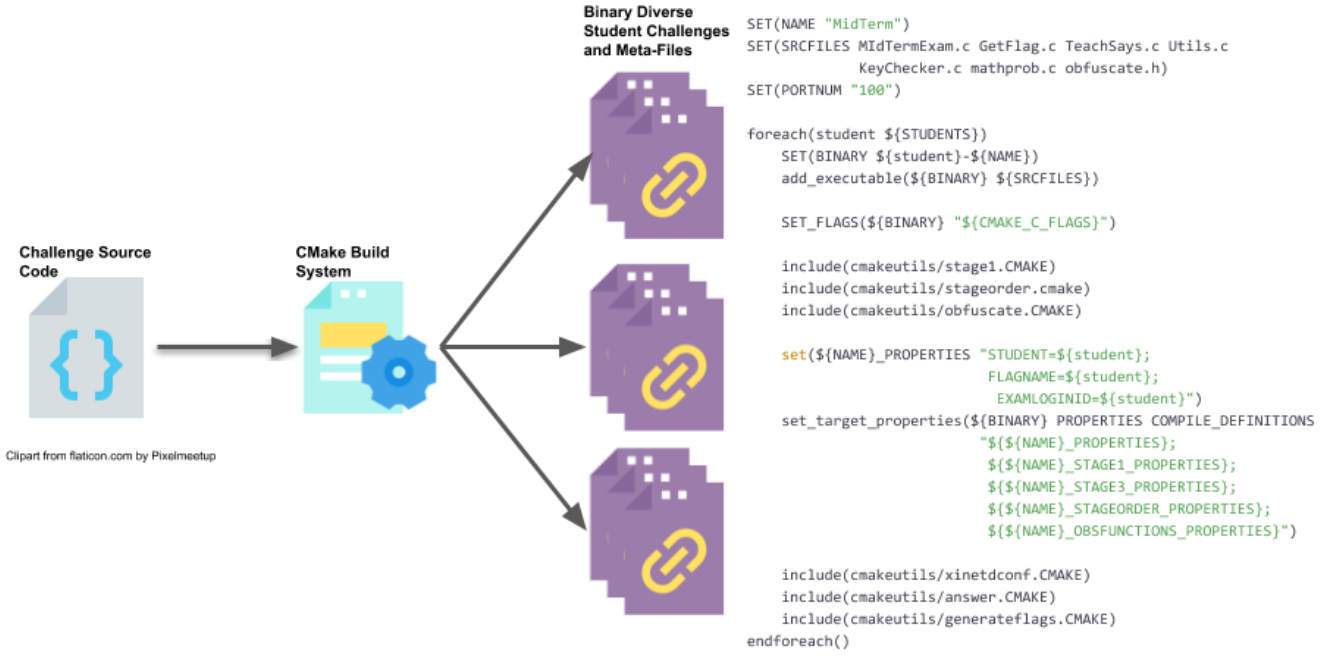

Figure 1: CMake Build System Flow for Binary Diversification (left) top-level CMake script generates diverse binaries (right)

2 PRIOR WORK

Binary analysis courses (e.g. binary RE and vulnerability research) typically require a corpus of compiled binaries for analysis. A simple solution is to present crackeme challenges [2]. Crackmes with their popularity in the hacker domain have an encyclopedia of published solutions available for students to find and use. However, readily available solutions make such challenges an invalid resource for grading opportunities like examinations. Automated tooling can lessen educators' burden from the time-consuming process of creating unique challenges. An initial step forward is the Tigress system [15] that obfuscates the source code through an automated means. This technique doesn't change the overall set of available solutions but does obfuscate the program's inner meaning similar to the international obfuscated C code contest [9]. In trying to create an anti-cheat environment this only presents a minimal barrier to student collaboration. In addition, we need to make sure that we provide a ramp in difficulty when presenting challenges to students [3] to allow time for the students to build a workflow in performing analysis. To go a bit further metamorphic code injection [6] has been used as an anti-cheating technique that produces binary diversification. This technique is the closest research to our goal but falls short in providing unique functionality changes to the challenges required to be solved.

PicoCTF [7] has shown great promise creating learning engagement in the classroom by leveraging a CTF approach. Typical CTF competitions consist of static binaries that are the same between competitors. As time and resource constraints limit the ability to both create and distribute unique binaries fairly among competitions. Our paper presents a dynamic method to develop underlying content for course based CTFs.

3 METHODOLOGY OVERVIEW

Our approach leverages the CMake build system version 3.0 [1] to construct unique challenges. *Figure 1* illustrates the creation of our unique challenges. We first generate the challenge source code. This source code contains the compiler definitions that the build system uses to specify the specific cases for binary diversification. The CMake build system processes these definitions to create diverse binaries, startup configuration files, answer keys, and flags (i.e., challenge solutions) for each student. The code listed in the figure is the top-level loop that performs each task to create unique data sets.

The following subsection discuss the key aspects of our approach including the creation of the binary compilation, answer files, flag files and configuration files. While the answer files and xinetd configuration use similar techniques for file generation, they each perform a specific goal in the overall build system. In addition, we do use the build system to generate flags for each challenge binary. While multiple methods exist to generate these flags, we use this system for simplicity. In the following sections, we limit our discussion of our tooling to use just a single source base for generating a single challenge binary and meta-files per student. We have used this approach to generate multiple unique challenge binaries for each student using the template presented here.

4 METHODOLOGY IMPLEMENTATION

4.1 Binary Compilation

Our compilation approach supports creating binaries for unique CPU architectures. Currently, we use this approach to present x86, x86_64, and ARM binaries to our students. On deployment to our

xinetd server we use qemu-user to emulate the ARM binaries. For reverse engineering challenges this provides a host of different combinations to evaluate our students learned skill set. The builds *CMakeLists.txt Figure 1* contains the top-level loop that iterates over the **STUDENTS** environment variable. We create a new binary name using the aggregation of **student** and **NAME** passed to the ***add_executable*** CMake function. This will create a separate compiled binary for each student. In our initial system we create a diverse binary per student using the following three techniques: function ordering, source code modifications, and obfuscation.

Function Ordering: The order of functions allow us to create a layer of randomness. We can include multiple challenges in the same binary and then perform a function call on a subset of those or a single challenge (e.g. Key Validator) could randomly choose an order given no order of operations dependencies. We demonstrate the result in Figure 2 that randomizes three different stage function (Stage 1, Stage 2, Stage 3) into different call graph orders. This is done by creating multiple environment variables **FUNCPTRx** that get string substituted during compilation. This example is pulled from our MidTerm exam at which point we haven't rigorously introduced the concept of CTFs to the students. As a result we typically compile multiple challenges into a single binary. We can see in Figure 3 where the compiler definitions are used to make the randomly ordered function calls. The other way we use this technique is in a key validator binary where we will evaluate user input as a key. In that case we would evaluate the key for correctness in random order.

Source Code Modifications: We also include source code modifications as apart of the compiler definitions. As a simple example we randomly choose between subtraction and addition mathematical operations Figure 4. We also choose how many times the student will have to successfully execute their responses to our input requests. While this is not an in depth example; it illustrates the technique. Source code modification is not limited to mathematical operations, other examples include using #define sections of code to include, change the operands of a function call, and set conditional expression values to be evaluated.

Function Obfuscation: Finally, obfuscating functions can be incorporated into source files. Though not as valuable as the prior techniques it still adds to the binary diversification. These functions are inline declared functions designed to lead the student away from the answer they should be solving. This noise requires the students to quickly figure out that this data is not valuable in solving the overall challenge. Specially it requires them to evaluate the disassembly graph view to find dead code paths. To implement we randomly choose an available function and set a compiler definition to be called during execution. These compiler definitions are string substituted during compile time. We can place them any where within our challenge to delay the students from getting to the core challenge code. We could interleave the functions amongst the challenge itself to make it a more difficult reversing task or simply place it at the start and end of the challenge.

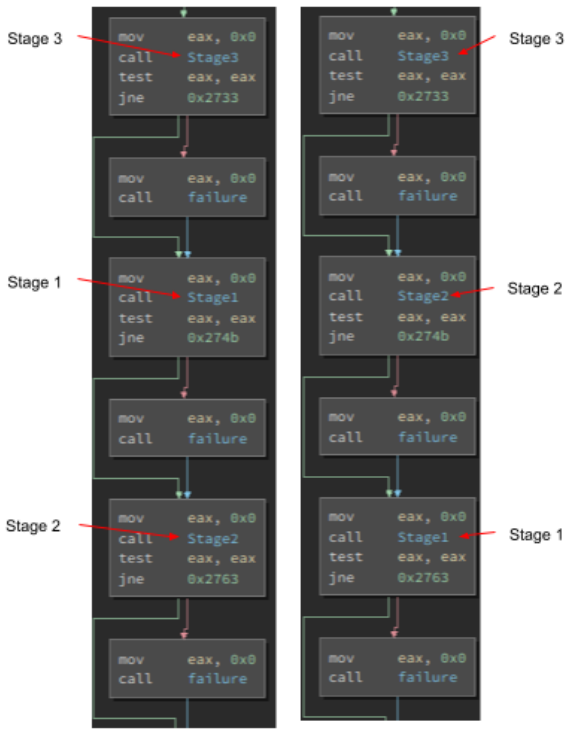

Figure 2: Disassembly view of Function Ordering results between two different binaries

Figure 3: Function ordering source

```
#define FP1 FUNCPTR1
#define FP2 FUNCPTR2
#define FP3 FUNCPTR3
FP1()
FP2()
FP3()
```

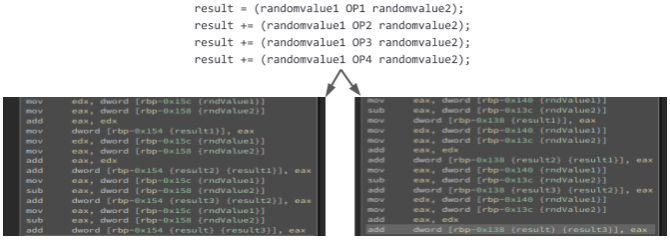

Figure 4: Source code modification of mathmatical operations. (left) Performs +,+,-,- (right) Performs -,+,-,+

4.2 Answer Files

To validate that our challenge is able to obtain a flag we generate a set of answer key files. Each answer file that is generated contains the customized settings used for generating the challenge binary. In *Listing 5* we demonstrate our template for the answer key for the challenge. To facilitate features such as debugging, local, and remote connection we use the Python library PwnTools [13]. This allows us to perform debugging locally as we are building our challenges, but also use the same script to connect to our deployed binary to obtain this students flags. This file acts as the template with the following environment variables.

- **${BINARY}**: The name of our current binary.
- **${student}**: The student we are current compiling for.
- **${STUDENT_ID}**: Internal ID we generated for this student.
- **${CHALLENGE_PORTNUM}**: Port number to use for this challenge.
- **${STAGE1OP(1,2,3,4)}**: The randomly generated operation used for this line of math

Using the ***configure_file*** function in the CMake script *Listing 5* all of the required environmental variables to generate a custom answer key are populated to a final student answer key fie installed into the students release directory.

4.3 xinetd Configuration

The challenge binaries presented are hosted on a cloud environment using the xinetd service. This provides a significant advantage since the binaries being generated aren't required to manually open listening sockets to be hosted. Xinetd performs the heavy lifting here to create a network service. Since our goal is to create a unique challenge binary per student we also need to generate the corresponding service file based on our template in *Listing 7*. At the top level *CMakeLists.txt* we set the environment variable **CHALLENGE_PORTNUM** to the lower hundred digits (e.g. 110, 506) of our 65535 port space. Aggregating the **ID_${student}** and **CHALLENGE_PORTNUM** we can generate a full port number to be used in our xinetd configuration file. Assuming using port numbers above 10000; we can have 55 students and 1000 challenges for each. This is more than sufficient for our purposes, but we could use any combination of decimal number places to add more students or challenges if needed. Using this information we can fill in the xinetd configuration template using a similar CMake script as our answer files to generate the students unique binary configuration file. Using the ***configure_file*** function all of the required environmental variables below will be filled in and generate a new configuration file to be installed in the release directory for the student.

- **${BINARY}**: The name of our current binary.
- **${student}**: The student we are current compiling for.
- **${STUDENT_ID}**: Internal ID we generated for this student.
- **${CHALLENGE_PORTNUM}**: Port number to use for this challenge.

4.4 Flags

In our final step of the process we generate a unique flag per student per binary. We use a universally unique identifier (UUID) as the flag.

Figure 5: Python answer script template

```
from pwn import *

DEBUGMODE = 0
LOCAL = 0
BINARY="./${student}-${BINARY}"
QUIETMODE = 1

HOST=<redacted>
PORT=${STUDENT_ID}${CHALLENGE_PORTNUM}

# Open our Connection
conn = remote(HOST, PORT)
conn.recvline();
conn.sendline("${student}");

loopcount = ${STAGE1STEPS}

    for i in range(0, loopcount ):
        values = conn.recvline();
        arrvalues = values.decode("utf-8")[:-1].split(",")

        value1 = int(arrvalues[0])
        value2 = int(arrvalues[1])

        if not QUIETMODE:
            print("%d␣,␣%d" % (value1, value2))

        result = value1 ${STAGE1OP1} value2
        result = result + (value1 ${STAGE1OP2} value2)
        result = result + (value1 ${STAGE1OP3} value2)
        result = result + (value1 ${STAGE1OP4} value2)

        conn.sendline(str(result))

    conn.recvline()
    print(conn.recvline().decode("utf-8")[:-1])
```

This allows us to randomly generate the flag using the CMake function ***string(UUID flag NAMESPACE ${UUID_DNS_NAMESPACE} NAME "${name}-${student}-${stage}" TYPE MD5)*** which will generate a random UUID. We can then save this string into a unique flag file with a name formatted by *${student}-${BINARY}*.

5 LESSONS LEARNED

Successes: Though not a rigorous study; we have observed that the students have taken different approaches in solving their challenge binaries. Each student turns in a detailed workflow analysis write up and answer files. From the course the students each have been taught the same workflow process, using binary diversity the students demonstrate unique results in how they have obtained their flags. In addition their answer python scripts that exercise the

Figure 6: CMake script for python answer file generation

```
configure_file(answer/answer.py ${student}_answer.py)

LIST(APPEND ANSWERFILES
     "${CMAKE_CURRENT_BINARY_DIR}/${student}_answer.py")

set(data "python3_./${student}_answer.py\n")
file(APPEND ${CMAKE_CURRENT_BINARY_DIR}/allflags.sh
     "${data}")
```

Figure 7: xinetd configuration file template

```
service ${BINARY}
{
    id          = ${BINARY}
    user        = ${student}
    server      = /home/${student}/${BINARY}
    disable     = no
    port        = ${STUDENT_ID}${CHALLENGE_PORTNUM}
    socket_type = stream
    protocol    = tcp
    wait        = no
    type        = UNLISTED
}
```

challenge contain unique approaches to the solutions despite given a template script file during the course. Our approach has provided a practical approach for anti-cheating.

We have found that our students enjoy the format of our RE course and specifically the challenges presented to them. Even after the course the students reach out to provide their experiences and overall just want to discuss the challenges further. We recently hosted a CTF where two students tied for the top spot. In order to break the tie a real-time head to head tie-breaker was performed. We gave the students two challenges to solve and the first to solve them both was declared the winner. This was well received by the entire class. The tie-breaker was hosted on discord and everyone in the class logged in and participated in the discussion as they watch their fellow classmates solve the challenges.

Challenges and Next Steps: One challenge found with using CMake is how the meta-files are created. These files are only created when the build system is generated; not when a binary is built. So during binary testing if you make source code changes and update your answer template files these files will not always be in sync. The build system doesn't update on source modifications. In contrast if you make a change to the *CMakeLists.txt* file your build system might get updated which in turns updates the meta-files. This can cause an out of sync issue when you have released your binaries to the students and your service server. During development the integrated development environment you choose may or may not present these challenges.

We need to extend the binary diversification by incorporating a code template system to modify the individual code blocks. The use of flow-based programming [11] could provide an approach. This has been demonstrated [14] with using it to generate challenges for SQL injection and cross-site scripting vulnerabilities. In addition, for our vulnerability research course we will evaluate the use of automated bug insertion [12] to enable further binary diversification for vulnerable challenges.

Finally, we are interested in extending the CTFd framework to provide a more dynamic game system. The CTFd framework presents a static game to the students. Being able to incorporate a dynamic game based on the binary diversification will be a paradigm shift in CTF competitions. Our focus would be to couple binaries and flags to a specific user allowing them to only view their unique set of challenges.

6 CONCLUSION

In this paper, we presented our technique to provide binary diversity for reverse engineering challenges. The goal was to provide a level of anti-cheat protections without having to manually create a unique set of examination binaries for each student. We have used this technique over three terms of classes thus far. It has been clear in the challenge analysis write ups, answer files, and post discussions with students that this provides a reasonable expectation to minimizing collaboration amongst the students. The use of the CMake build system demonstrates a practical means to incorporate randomness to generate an environment for binary diversification.

REFERENCES

[1] Website. CMake build system. http://www.cmake.org
[2] Website. crackmes.one. http://crackmes.one
[3] John Aycock, Andrew Groeneveldt, Hayden Kroepfl, and Tara Copplestone. 2018. Exercises for teaching reverse engineering. In *23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ACM, Larnaca Cyprus, 188–193.
[4] Sergey Bratus. 2015. What Hackers Learn that the Rest of Us Doní. (2015).
[5] Tom Vanden Brook. Dec 21, 2020. West Point accuses more than 70 cadets of cheating in worst academic scandal in nearly 45 years. *USA Today* (Dec 21, 2020). https://www.usatoday.com/story/news/politics/2020/12/21/west-point-catches-70-cadets-worst-cheating-scandal-50-years/5856130002/
[6] Wu chang Feng. 2015. A Scaffolded, Metamorphic CTF for Reverse Engineering. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/3gse15/summit-program/presentation/feng
[7] 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 Association, San Diego, CA. http://www.usenix.org/conference/3gse14/summit-program/presentation/chapman
[8] R. Fanelli and TJ OConnor. 2010. Experiences with Practice-Focused Undergraduate Security Education. Workshop on Cyber Security Experimentation and Test. In *2010 USENIX 3rd Workshop on Cyber Security Experimentation and Test*. USENIX, Washington, DC.
[9] S Cooper L Broukhis and LC Noll. Website. International Obfuscation C Code Contents. http://www.ioccc.org
[10] Heater Mongilio. Dec 22, 2020. Naval Academy reviewing final physics exam after 'inconsistencies'. *Capital Gazette* (Dec 22, 2020). https://www.capitalgazette.com/education/naval-academy/ac-cn-naval-academy-test-inconsistencies-20201222-llyhwrq5a5grxnkp4utupnwjla-story.html
[11] J. Paul Morrison. 2010. . CreateSpace.
[12] Jannik Pewny and Thorsten Holz. 2020. EvilCoder: Automated Bug Insertion. arXiv:2007.02326 [cs.CR]
[13] pwntools Website. PwnTools CTF Toolkit. https://github.com/Gallopsled/pwntools

[14] Marina Ribaudo and Andrea Valenza. 2019. Semi-Automatic Generation of Cybersecurity Exercises: A Preliminary Proposal. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on Ensemble-Based Software Engineering for Modern Computing Platforms* (Tallinn, Estonia) *(EnSEmble 2019)*. Association for Computing Machinery, New York, NY, USA, 16–21. https://doi.org/10.1145/3340436.3342728

[15] Clark Taylor and Christian Collberg. 2016. A Tool for Teaching Reverse Engineering. In *2016 USENIX Workshop on Advances in Security Education*. USENIX, Austin,TX.