USING CONCOLIC EXECUTION TO IDENTIFY IA32 PROGRAM ERRORS

By
Carl Zhang

Senior Thesis in Computer Engineering
University of Illinois at Urbana-Champaign
Advisor: Associate Professor Steven Lumetta

May 2020
Abstract

In Computer Science education, one of the most important tasks is to provide students with feedback that can help them discover errors in their assignment code. Traditionally, this check is achieved by executing a series of pre-defined test cases. But many bugs are not easily exposed by such test cases, which are thus insufficient for fair grading. Furthermore, failed test cases give students little feedback as to how to fix their code.

In the last decade, tools have been developed for code testing that aim at achieving high code coverage even in strict environments, such as interacting with the operating system. These tools can be helpful if applied in Computer Science education. Among these tools, KLEE [1] is particularly designed for improving control flow paths coverage by exploring different execution paths in the program using concolic execution.

In this thesis, we investigate the possibility of using concolic execution with KLEE to generate feedback for student assignments written in IA32 (32-bit version of x86) assembly, like the MP1 in our operating systems course (ECE391). By developing tools for lexical and control flow analysis to translate IA32 to C, we were able to take advantage of KLEE to explore the program’s execution path thoroughly to generate test cases and feedback that can be helpful for students to detect problems in their programs. The initial test shows that among the 180 student codes, our tool picked up 139 cases that contain errors compared to the 105 cases that got picked up by the normal grader, and that all student codes that have errors detected by the grader have been detected to contain errors by our tool.

Subject Keywords: Concolic Execution; Symbolic Execution; IA32; Lexical Analysis; Control Flow Analysis
Acknowledgments

I want to send my deepest gratitude to Professor Steven Lumetta, who offered me his earnest help on both my academic and life path. Your brilliance and integrity will forever inspire me.

I want to thank my parents and my grandma for the support, both physically and mentally, that guided me through the four years of my college life.
# Contents

1. Introduction ......................................................................................................................... 1

2. Literature Review .................................................................................................................. 3  
   2.1 Concolic Execution & KLEE ......................................................................................... 3  
   2.2 Use of Concolic Execution as Automated Grading Tool ............................................ 3  
   2.3 McSema & Remill ......................................................................................................... 4  

3. Software Architecture ......................................................................................................... 5  
   3.1 Lexical Analysis ............................................................................................................. 6  
   3.2 Simulation of Machine Architecture ............................................................................ 6  
   3.3 Operation of Instruction Restore .................................................................................. 7  
   3.4 Control Flow Restore ................................................................................................. 7  
   3.5 Wrapper File ............................................................................................................... 12  

4. Test on ECE 391 MP1: Life or Death ................................................................................... 13  
   4.1 Functions to be Implemented ...................................................................................... 13  
   4.2 Example on testing mp1_ioctl_startgame ................................................................. 14  
      4.2.1 Wrapper Design .................................................................................................... 15  
      4.2.2 Results ................................................................................................................ 18  

5. Conclusion .......................................................................................................................... 20  

References ............................................................................................................................ 21
1. Introduction

Programming assignments, especially for introductory-level courses, usually use high-level programming languages like C/C++, Python or Java. But advanced courses that introduce concepts like computer systems or hardware often require that students use assembly language to understand better how software is executed. For instance, we use LC-3 extensively in multiple courses (ECE120, ECE220, ECE 385) at UIUC, and IA32 (32-bit version of x86) is also used in the Computer System Engineering course (ECE 391). Programming in assembly can be quite confusing because it requires a good understanding of the computation model. Traditionally, students debug using a set of test cases, checking the result for every case. The problem with this approach is that it is hard to have a set of test inputs that can discover all bugs. In many cases, there are unexposed bugs that cannot be caught by the given test cases. Therefore, a tool that provides students with more relevant information that can help them discover problems is often desired.

Previously, Dr. Jianxiong Gao, a Ph.D student under the supervision of Professor Lumetta, developed a system that uses concolic execution to generate feedback for student assignments written in C [2]. His tool utilized the concolic execution engine KLEE [1] and made innovations that enable KLEE to give accessible feedback efficiently (usually within 5 minutes). Concolic execution tries to cover all paths in a program by assigning an initial value to the input and using a satisfiability modulo theories (SMT) solver on conditional expressions at a branch to determine values needed to take all paths through the program, thus has an advantage over the traditional test-case-based system of being able to customize feedback for each individual student program. Given the success of the project, we think that a similar approach can also be applied to assembly assignments.

In this thesis, we investigate using concolic execution to generate feedback for student assignments written in IA32 assembly. I developed a tool that tests the student program more thoroughly and generates information useful for debugging. We utilized KLEE as the core of checking the execution paths of the program. To take advantage of KLEE, which takes as input LLVM IR, a generic assembly language used by LLVM compilers, we built a tool to convert the assembly code to C, while retaining the operations and control flow of the original code.
Dr. Gao’s tool takes LLVM IR as input, but also leverages the block structure implied by use of the C programming language in order to reduce the computation needed. To transform IA32 into block-structured code, structural transformations and use of code replication that maintain the behavior and control flow paths of the original program are needed. In particular, I wrapped an emulation environment in C around the translated program, which implies both careful translation of individual instructions as well as accurate replication of emulated behavior of the code as a whole.

The generated code, including KLEE setup code and a set of reimplemented given functions of the assignment that include the testing logic for all functionalities, is compiled by Clang to generate LLVM IR, which is then consumed by KLEE. Finally, KLEE outputs test cases consisting of vectors of values for symbolic variables such that every execution path is represented, as well as information relevant to the errors detected.
2. Literature Review

2.1 Concolic Execution & KLEE

Symbolic execution tries to determine the output of a program in terms of the expressions and constraints of inputs that are marked as symbols. The problem with symbolic execution is that in an actual application, it is common that the expression for a program will take forever to generate. Concolic execution is a combination of normal execution and symbolic execution, in which some inputs are designated as symbolic and may be constrained to limited sets of values. For example, an unsigned integer symbolic variable \( v \), if constrained by the condition “\( v < 2 \)”, will have possible values 0 and 1. Concolic execution starts with a set of concrete values assigned to each input, and just execute it. Along the execution, constraints and expressions that might make the program fork will be collected. When the execution is finished, these constraints get evaluate by a satisfiability modulo theories (SMT) solver and a new set of values that would make the program go down a different path will be chosen efficiently, and a new execution is made. This process is repeated until there are no more paths to be explored in the program, and the algorithm terminates. As a result, concolic execution is able to take advantage of symbolic execution to achieve high path coverage, but keep the duration of the execution within a reasonable amount of time.

To take advantage of concolic execution in our project, we chose to use KLEE [1]. KLEE is a state-of-the-art concolic execution engine that takes in source code interpreted in LLVM IR and outputs a set of test cases containing input values that lead to different execution paths.

2.2 Use of Concolic Execution as Automated Grading Tool

Similar research has already been done by Dr. Jianxiong Gao [2] on using KLEE to detect errors in student assignments written in C. He built the automated tool that generates timely feedback to students that is easily understandable. The innovation in this research project lies in 3 ways. First, I/O functions extension is added to KLEE to reduce symbolic I/O overhead when handling I/O. By not having to compile the uClibc C code to LLVM IR, the extension achieves 10x speedup and 100x less memory usage. Second, to avoid having to explore the exponential number of paths when loops are present, an algorithm is developed so that only
necessary paths that contribute to code coverage are explored, achieving fast execution while maintaining the ability to maximize code coverage. Third, to reduce the amount of time spent on solving SMT queries, a cross-execution cache that stores solutions to the queries is implemented so that when similar path conditions are encountered, the cache can help speed up the entire execution process.

2.3 McSema & Remill

McSema [3] is an open-source project that can translate machine code from executable IA32 binary files to LLVM bitcode. It recovers the control flow of the binary by using licensed disassembler tools and output a control flow graph (CFG) file. Then, it uses a library called Remill [4] to convert the CFG file to LLVM bitcode. Although the aim of the project is mostly aligned with our purpose, several concerns eventually prevented us from using it. First, the use of a paid disassembler tool to generate the CFG file, which is actually not needed for us because McSema deals with machine code, while what we have is assembly code. Second, Remill keeps data structures that simulate the x86 ISA in its entirety, a big portion of which is not covered in the student assignments, like floating-point operations and registers. Such extra details might burden KLEE on the execution. Therefore, we decided to implement our own software to make the conversion.
3. Software Architecture

The core functionality of my tool is the conversion from IA32 to LLVM IR for KLEE, as shown in Figure 3.1. The conversion consists of three steps. First, a lexical analysis tool is needed to abstract information on each instruction. Second, a tool that determines the control flow structure for the program is used to convert the corresponding program into C. Finally, we need a file containing KLEE setup, functional correctness check, and the modified version of the given functions in the assignment which work as callbacks so that we capture the behavior of the program by replacing the original version to make the final source code to be checked by KLEE.

Figure 3.1: Data flow of the IA32 concolic testing system.
3.1 Lexical Analysis

A lexical analysis tool is needed to convert every IA32 instruction into an abstracted data structure for further control flow analysis. We used Flex [5] as the lexical analyzer generator. For every instruction, we need to abstract information including the opcode, the instruction address, its operand format, and its operands. Figure 3.2 gives an example of such an analysis.

(a). Example IA32 code.

| 1. leal (%esi),%ebx |
| 2. movw $0x1234, %ax |

(b). Instructional information extracted (stored in data structures).

| 1. // information of first instruction |
| 2. IR opcode: 23 (LEA) |
| 3. IR addr: 0 |
| 4. IR opform: 5 (source: memory location; destination: register) |
| 5. IR operand length: 32 |
| 6. IR dest reg: 1 (%ebx) |
| 7. IR src addr: esi |

| 8. // information of second instruction |
| 9. IR opcode: 20 (MOV) |
| 10. IR addr: 1 |
| 11. IR opform: 3 (source: immediate value; destination: register) |
| 12. IR operand length: 16 |
| 13. IR dest reg: 8 (%ax) |
| 14. IR src imm: 4660 (0x1234) |

Figure 3.2: Example instruction analysis

Lexical analysis produces an array of per-instruction data structures that is then used to perform control flow analysis.

3.2 Simulation of Machine Architecture

The generated C code must perform the exact same operations as the original program. To enable this behavior, we store IA32 processor state as variables. In the register file, eight general-purpose registers (eax, ebx, ecx, edx, esi, edi, ebp, esp) and the flag register (eflags) are
sufficient for the purpose of the subset of IA32 used by our class. For memory simulation, since students need to manipulate data on the runtime stack, the global memory region, and dynamically allocated memory, we used one array of 300 bytes combining the stack (200B) and the global region (100B), and a second array of 8192 bytes for the heap.

3.3 Operation of Instruction Restore

To reproduce the operation of every instruction, information from the lexical analysis is vital to determining the correct operation in C. The C representation of the operation of an instruction is determined by the opcode, the source and destination operands, and operand length. One other important operation that can affect the control flow is the update of the condition codes, which are stored in the eflags register. Many instructions in IA32 update one or more condition codes, which jump instructions use to determine whether to jump to the target address. Such behavior also needs to be reflected in the generated C code, as illustrated later in Figure 3.4 (c).

3.4 Control Flow Restore

Our tool first uses the instruction analysis described above to generate a representation of the control flow graph. The control flow graph (CFG) is a directed graph that shows the control flow of a program during execution. In a control flow graph, a basic block is a sequence of instructions with no internal entry points and a single exit, and an edge is a jump from one block to another to indicate the potential execution sequence. Blocks are typically terminated by unconditional jump (JMP), conditional jump (JNE, JE, etc.) or return (RET) instructions. These blocks jump to other blocks based on the target address or the address of the first instruction of the succeeding block. There are some blocks that end in other instructions and jump only to their succeeding blocks, such as those that are succeeded by a block with multiple incoming arcs.

The next step is to use the control flow graph to generate the equivalent C code. In this part, there are key elements in the control flow graph that need to be recognized, including branches and strongly connected components (SCC). Branch structures in the control flow are translated to if-else statements. SCCs represent loop structures in the program, which are
converted to loops in C. If we simply want to reproduce the CFG, then converting all jump instructions to goto statements in C would be straightforward, and this analysis would become unnecessary. But since we want to enable the use Dr. Gao’s work, which has optimizations on loops and requires that each loop has a single entry point, we need to have C style loops instead of goto statements. Therefore, we developed an algorithm that dissects the CFG into units of SCCs and recursively dissects and generates code for every SCC to restore the control flow of the original program, as shown in Figure 3.3. The generate_func_for_scc function generates a string that contains the function for an SCC. It calls the process_scc function to process the current SCC and recursively dissects the SCC into smaller SCCs. For every SCC, generate_func_for_scc breaks back arcs for the SCC. Once the SCC is processed, the back arcs need to be restored, because an SCC might have multiple entry points which would make the code structure different, so the SCC needs to retain its structure for different entry points. Our code produces a separate copy of an SCC for each entry point, ensuring that the code seen by Gao’s tool observes only an entry point per loop. The process_scc function recursively processes the current SCC by looking for smaller SCCs (sub SCCs) starting on the current block. If the sub SCC contains only a single block, all of the instructions in that block get converted to the corresponding C code. Otherwise, a new function for the sub SCC is generated. The current SCC uses the return value (ID unique for every block) for the function of the sub SCC to determine which path to take. If the sub SCC contains multiple outgoing arcs, a switch statement or if-else statement is generated. These outgoing arcs can produce different operations. If an outgoing arc leaves the current SCC, then it generates a return statement. If an outgoing arc points to the head block of the current SCC, it generates a continue statement. If the outgoing arc stays within the current SCC but does not point to the head block, it implies a sequential structure, and process_scc is called on the beginning of that subsequent block. This process continues until all blocks in the current SCC have been processed and the corresponding code has been generated.
1. `generate_func_for_scc(scc) {`
2.   /* break back arcs in the scc that points to head */
3.   break_back_arcs();
4.   generate_while_loop_statement(); // print while loop statement
5.   process_scc(scc.entry, scc);
6.   recover_back_arcs();
7. }
8. 
9. `process_scc(start_block, scc) {`
10.   sub_scc = detect_scc(start_block);
11.   /* find removed back arcs and back arcs pointing
12. * to current sub_scc's head
13. */
14.   curr_back_arcs = find_curr_back_arcs(sub_scc, scc);
15.   old_back_arcs = find_old_back_arcs(sub_scc, scc);
16.   /* case where the sub_scc contains only 1 block */
17.   if (sub_scc.size == 1) {
18.       generate_c_instruction(sub_scc.block); // print instructions in C
19.   } else {
20.     /* case where sub_scc contains multiple blocks,
21.   * proceed to generate code for the sub_scc first
22. */
23.       generate_func_for_scc(sub_scc);
24.   }
25. 
26.   /* For SCC with multiple outgoing arcs, generate switch or if-statement
27. generate_switch_or_if_statement(); // print switch statement
28. 
29.   */ generates code for each case for the switch/if statement */
30.   for (arc in curr_back_arcs) // back arcs for current SCC -> continue
31.       generate_continue_statement();
32.   for (arc in old_back_arcs) // back arcs for larger SCC -> return
33.       generate_return_statement();
34.   /* For arcs pointing out, if points to blocks in current SCC,
35.   * keep processing. Otherwise, generate return statement */
36.   for (arc in sub_scc.after_blocks) {
37.       if (scc.blocks.find(arc.block))
38.         process_scc(arc.block, scc);
39.       else
40.         generate_return_statement();
41.   }
42. }

Figure 3.3: Pseudo code of control flow restore algorithm.
(a). Example IA32 code (SCCs are block 3&4, block 2 and block 5)

(b). Corresponding control flow graph.

Figure 3.4: Example Conversion from IA32 to C code (continued on following page)
1. int block_4_3() {
   2.     while (1) {
   3.         // block 4
   4.         res = (uint64_t)ecx + 1;
   5.         ecx = (uint32_t)res;
   6.         update_cc(res, OP_INC); // update cc for INC
   7.         res = (uint64_t)ecx - 16;
   8.         update_cc(res, OP_CMP); // update cc for CMP
   9.     if (eflags & 0x0040) { // check zero flag for je instruction
   10.         return 5;
   11.     } else {
   12.         // block 3
   13.         *(uint32_t *)(M + ecx) = (uint32_t)0;
   14.         continue;
   15.     }
   16.   } // block 5
17. }
18.
19. int entry() {
20.   // block 2
21.   res = (uint64_t)ecx ^ ecx;
22.   ecx = (uint32_t)res;
23.   update_cc(res, OP_XOR); // update cc for XOR
24.   block_4_3();
25.   // block 5
26.   return eax;
27. }
28.}

(c). Converted C code

Figure 3.4 (cont.): Example Conversion from IA32 to C code

Figure 3.4 shows the C code generated for an example code. The CFG is shown as well, with block 3&4, block 2 and block 5 as SCCs. Block 2 and 5 are single-block SCCs, so their code gets directly translated into C. Block 3 and 4 form a loop that is entered at label “loop” and is terminated at the “je done” instruction. This loop is converted to a while loop, and a separate function block_4_3 is generated. Since the entry point for the loop is the beginning of block 4, block 3, which has an arc pointing at the head of the loop, is terminated by a continue statement. Block 4 contains a branch at the end, with one arc pointing outside of the block 3&4 SCC and one arc pointing to block 3. Therefore, an if-else statement is generated, with the outgoing arc converted to return and the other arc proceeding to execute code in block 3.
3.5 Wrapper File

The final step is to implement setup code for KLEE. First, certain variables need to be marked as symbolic so that KLEE can actively explore different paths in the code. The selection of symbolic variables depends on what students use to branch their code. For instance, if they are asked to use the return value of a given function to branch, then a state variable inside the given function should be made symbolic so that the given function will return different values to affect the student code’s execution. Such initializations are made by calling function `klee_make_symbolic`, which marks a variable symbolic, and `klee_assume`, which marks some constraints that the symbolic should meet. Second, some checking on the functional correctness of the program should also be setup. While KLEE actively explores the execution path, it doesn’t really know what our program does, thus unable to check the correctness of the program. Therefore, checks should be manually implemented to check if the program generates the intended output. We used `assert` to check such outputs. The wrapper file also provides the modified version of the given function in the assignment, which will be called to record some states on the student program that can be used to check. Finally, the generated C code combined with the wrapper will be converted into LLVM IR by Clang and inputs to KLEE to generate the test cases and feedback for students. The entire process will be illustrated with an example in Chapter 4.
4. Test on ECE 391 MP1: Life or Death

In the Machine Problem 1 in ECE 391 during Fall 2019, students need to implement a
text-mode game to save humans from a virus by finding a sequence of DNA bases that can lead
to vaccination against the virus. This program operates as an extension of the Linux real-time
clock (RTC) driver, with the intention of giving students experience on interacting with the
Linux kernel as well as techniques like double buffering, jump tables, and argument checking
using assembly.

The student program should implement some functions to manipulate data including two
game boards (current_board and next_board) and game state information: aggression,
population, and infection.

4.1 Functions to be Implemented

Student program must implement the following functions:

void mp1_rtc_tasklet (unsigned long arg)

This function is called whenever an RTC interrupt is generated. It must update the game
boards and draw on the screen, then notify the user-level code that the boards have been
updated.

int mp1_ioctl (unsigned long arg, unsigned long cmd)

This function jumps to one of the five following core functions based on the argument
cmd, and the target core function takes arg as the parameter and executes.

int mp1_ioctl_startgame (unsigned long seed)

This function needs to allocate memory for two game boards, current_board, and
next_board, and fill them with 0s. Next, the given init_virus function should be called to
place virus into the current_board and initialize the infection, aggression, and population
values.

int mp1_ioctl_endgame (unsigned long ignore)
This function needs to free the memory allocated for the two game boards and set board pointers to 0.

**int mp1_ioctl_keystroke (struct keystroke_args *keystroke_args)**

This function needs to copy the passed-in keystroke_args data structure into kernel memory, update the data structure, draw on the screen to reflect the effect of the pressed key and pass the data structure back to user-space memory.

**int mp1_ioctl_getstatus (unsigned long *user_status)**

This function needs to copy to user data structure that includes population and infection information of the game.

**int mp1_ioctl_vaccinate (unsigned long packed_args)**

This function needs to call the given generate function to determine whether to kill each cell on the board based on information in the packed_args and update the aggression variable.

### 4.2 Example on testing mp1_ioctl_startgame

Our approach to testing the functions is to test each function separately. We implemented a thorough testing on one of the core functions, mp1_ioctl_startgame.

For mp1_ioctl_startgame, the operations start by pointing current_board and next_board to two dynamically allocated memory regions, each of size 1,600 bytes, that is allocated by calls to the provided function mp1_malloc. If either one of the allocations fails, mp1_free should be called to free any allocated memory and leave the two pointers set to 0, and the function should return -1. If allocation succeeded, both boards should be filled with 0s, and init_virus should be called on the current_board and the return value should become the initial value for infection. The values aggression and population should be set to 8,000,000 and 80 respectively. The function should return 0 if both allocations succeed.
4.2.1 Wrapper Design

The wrapper file includes the reimplementation of the given function to facilitate testing, marking symbolic variables for KLEE and functionality checks.

Given functions including \textit{mp1_malloc}, \textit{mp1_free}, \textit{seed\_generate} and \textit{init\_virus} were rewritten in analysis versions that do checking on things like the validity of the arguments that get passed in. The analysis version of the given function also contains symbolic variables on which the return value of the function depends. The analysis version of \textit{init\_virus} is shown in Figure 4.1.

```c
1. int init_virus(unsigned char* board) {
2.     // get correct address
3.     board = (uint8_t*)((uint32_t)board + (uint32_t)M);
4.     // keep count of calls on init_virus
5.     init_virus_cnt++;
6.     // check argument validity
7.     if ((uint32_t)board != *(uint32_t*)(M + current_board) + (uint32_t)M) {
8.         invalid_init_virus_arg++;
9.     }
10.    // return using a symbolic variable
11.    return infection_init;
12. }
```

Figure 4.1: Example analysis version of given function: \textit{init\_virus}

We set up the following symbolic variables as shown in Figure 4.2:

1. \textit{reg\_offset}: A 1-bit value to be added to the callee-saved registers (ebx, esi, edi, ebp) to verify the preservation of these registers when the subroutine finishes.
2. \textit{check\_malloc}: This variable defines the number of calls to \textit{mp1\_malloc} that returns properly allocated memory rather than NULL. If calls to \textit{mp1\_malloc} exceed \textit{check\_malloc}, \textit{mp1\_malloc} will return NULL. By setting this variable as symbolic, KLEE is able to check execution paths that involve two successful allocations, one successful allocation followed by one failed allocation, and two failed allocation.
3. \textit{infection\_init}: This variable is used as the return value of the analysis version of \textit{init\_virus} to check if it is called correctly and its return value is used to initialize the \textit{infection} variable.

```c
1. klee_make_symbolic(&reg_offset, sizeof(reg_offset), "reg_offset");
```
2. klee_make_symbolic(&check_malloc, sizeof(check_malloc), "check_malloc");
3. klee_make_symbolic(&infection_init, sizeof(infection_init), "infection_init");
4. klee_assume(reg_offset < 2);
5. klee_assume(check_malloc <= 2);
6. ebx = reg_offset + EBX_MAGIC;
7. esi = reg_offset + ESI_MAGIC;
8. edi = reg_offset + EDI_MAGIC;
9. ebp = reg_offset + EBP_MAGIC;

Figure 4.2: Marking symbolic variables and initialization

The wrapper includes the following functional correctness check:

1. The correct return value of three different execution paths (1. 0 if both allocations succeed; 2. -1 if first allocation succeeds and second fails; 3. -1 if first allocation fails)
2. current_board and next_board set to 0 if either one of the allocations fails.
3. mp1_free should be called if first allocation succeeds and second fails.
4. In case both allocations succeed, both boards should be filled with 0s.
5. In case both allocations succeed, init_virus should be called on the current_board.
6. seed_generator should be called with the argument that is passed into mp1_ioctl_startgame.
7. In case both allocations succeed, variables containing game state information should be initialized correctly (aggression be set to 80, population be set to 8,000,000, infection be set to return value of init_virus).
8. The preservation of all callee-saved registers (ebx, esi, edi, ebp).

Figure 4.3 shows the code for functional correctness check for mp1_ioctl_startgame.
1. ret = mp1_ioctl_startgame();
2. // check callee saved regs
3. if (ebx != reg_offset + EBX_MAGIC)
4.    assert(0&&"Callee-saved reg ebx corrupted");
5. if (esi != reg_offset + ESI_MAGIC)
6.    assert(0&&"Callee-saved reg esi corrupted");
7. if (edi != reg_offset + EDI_MAGIC)
8.    assert(0&&"Callee-saved reg edi corrupted");
9. if (ebp != reg_offset + EBP_MAGIC)
10.   assert(0&&"Callee-saved reg ebp corrupted");

11. // check return value
12. if (ret)
13.    assert(0&&"return value is not 0 for successful malloc");
14. // check seed generator things
15. if (seed != SEED_MAGIC)
16.    assert(0&&"seed_generator not called properly");
17. // check boards
18. if (invalid_malloc_arg_cnt != 0)
19.    assert(0&&"malloc not called with size arg of 1600");
20. if (*(uint32_t*)(M + current_board) == 0)
21.    assert(0&&"current_board is null");
22. if (*(uint32_t*)(M + next_board) == 0)
23.    assert(0&&"next_board is null");
24. // Checking first and last cell
25. if (*(uint8_t*)((uint32_t)M + *(uint32_t*)(M + current_board)) != 0 &&
26.     *(uint8_t*)((uint32_t)M + *(uint32_t*)(M + current_board) + 1599) != 0)
27.    assert(0&&"some cell in current_board not set to 0");
28. if (*(uint8_t*)((uint32_t)M + *(uint32_t*)(M + next_board)) != 0 &&
29.     *(uint8_t*)((uint32_t)M + *(uint32_t*)(M + next_board) + 1599) != 0)
30.    assert(0&&"some cell in next_board not set to 0");
31. if (invalid_init_virus_arg)
32.    assert(0&&"init_virus not called on the current_board");
33. // check aggression
34. if (*(uint32_t*)(M + aggression) != 80)
35.    assert(0&&"Wrong aggression number");
36. // check population
37. if (*(uint32_t*)(M + population) != 8000000)
38.    assert(0&&"Wrong population number");
39. // check aggression
40. if (*(uint32_t*)(M + infection) != infection_init)
41.    assert(0&&"Wrong infection number");

Figure 4.3: Functionality check of one path (two successful allocations)
4.2.2 Results

To deploy the test, our concolic execution tool was run in Ubuntu 18.04, and a traditional grader was run on a 32-bit Linux system virtual machine with QEMU. In order to perform the test, we collected code from all students who took ECE391 during Fall 2019. Some of the student codes were unable to be tested due to lost data and code from the anonymization process, and the fact that our tool was unable to handle additional subroutines written by students. So we ended up performing the test on 180 samples.

Figure 4.4 shows the number of student code with or without errors that are detected by the grading program used by the class vs. the number of student code with or without errors detected by our tool. This grader is a fairly rigorous one because one of the faculty members was involved in the design. We show that all student codes containing errors that are detected by the grader have also been picked up by our tool, and that our tool detected more cases with errors than the grader did.

<table>
<thead>
<tr>
<th></th>
<th>Free of errors (Concolic Execution Tool)</th>
<th>With errors (Concolic Execution Tool)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free of errors (Grader)</td>
<td>41</td>
<td>34</td>
</tr>
<tr>
<td>With errors (Grader)</td>
<td>0</td>
<td>105</td>
</tr>
</tbody>
</table>

Figure 4.4: Student code with/without errors (Grader vs. Concolic Execution Tool)

The detailed test results are shown in Figure 4.5. The Errors column shows the errors that the tools try to find in the student code. The Grader column represents how many student codes were found to have the error by the grader, and the Concolic Execution Tool column represents how many student codes were found to have the error by our tool.

From the test results presented, we can tell that the concolic execution tool in general is able to detect more errors in most cases. In many cases, the traditional grader outputs a generic message for the error, which could indicate a crash. While it is hard to detect some problems if the program crashes, our concolic execution tool, with the simulated machine environment, is still able to detect a decent number of errors. In Figure 4.5, the results should not be compared
row-by-row because errors that were found by the two tools generate different error messages. For instance, if a student assumes that the second malloc always succeeds, then the test in the grader crashes and generates a generic failure messages for the second malloc failure, whereas our tool, depending on the error, generates a memory error message. There are also errors that the grader is not able to detect, such as the callee-save of registers, due to bugs in the grader.

Further tests revealed that making the register offset variable symbolic adds a significant amount of execution time in some cases. Without the symbolic variable, the average execution time is 0.331s. If the variable is made symbolic, the execution in some cases cannot terminate within an hour. This possibly has something to do with the SMT solver having a hard time solving for the value if the symbolic variable is used in memory operations.

<table>
<thead>
<tr>
<th>Errors</th>
<th>Grader</th>
<th>Concolic Execution Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return -1 if first malloc fails</td>
<td>45</td>
<td>18</td>
</tr>
<tr>
<td>Set current_board to null if first malloc fails</td>
<td>N/A</td>
<td>30</td>
</tr>
<tr>
<td>Set next_board to null if first malloc fails</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Return -1 if second malloc fails</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Set current_board to null if second malloc fails</td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>Set next_board to null if second malloc fails</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Generic fail when second malloc fails</td>
<td>87</td>
<td>N/A</td>
</tr>
<tr>
<td>Call mp1_free</td>
<td>23</td>
<td>29</td>
</tr>
<tr>
<td>Call mp1_free on current_board</td>
<td></td>
<td>38</td>
</tr>
<tr>
<td>Return 0 if both malloc succeed</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>Board size should be 1,600</td>
<td>14</td>
<td>N/A</td>
</tr>
<tr>
<td>Call seed_generator with given argument</td>
<td>20</td>
<td>29</td>
</tr>
<tr>
<td>Malloc is called with argument 1,600</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>current_board should not be null</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>next_board should not be null</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>Cells in current_board set to 0</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>Cells in next_board set to 0</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>init_virus called with current_board</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>Aggression number set to 80</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Population number set to 8,000,000</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Infection number set to return value of init_virus</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Memory error</td>
<td>N/A</td>
<td>47</td>
</tr>
<tr>
<td>Callee-saved registers</td>
<td>N/A</td>
<td>46</td>
</tr>
<tr>
<td>Free of bugs</td>
<td>75</td>
<td>41</td>
</tr>
</tbody>
</table>

Figure 4.4: Test result (normal grader vs. concolic execution tool)
5. Conclusion

Based on our initial experiment, the use of concolic execution on checking student programs written in IA32 assembly has been promising. Our tool enabled the correct translation of the control flow and operations of the original code to be checked by KLEE, and the testing script enabled KLEE to fully explore all the execution paths that cover the entirety of the input code. As a result, the output test cases by KLEE should be a reliable type of feedback that the students can use to debug their code.

The next step is to keep perfecting the tool. There are some cases where the control flow analysis tool is unable to handle, like self-written subroutines by students, which we expect to fix. We also expect to do more test to explore the impact of limitations of KLEE, such as the costly memory operation and the unexpected SMT solver cost. With the possible issues in mind, Dr. Jianxiong Gao [2] developed a series of techniques that can help alleviate the problem in his test on C code. Therefore, similar ideas can be applied to make KLEE more time-efficient in our tool. For instance, the loop reduction functionality should help reduce execution time significantly for memory operations that uses symbolic variables.
References


