Victoria University of Wellington
School of Engineering and Computer Science

SWEN326: Safety-Critical Systems

Assignment 3

Due: Monday 20th May @ 23:59

Overview

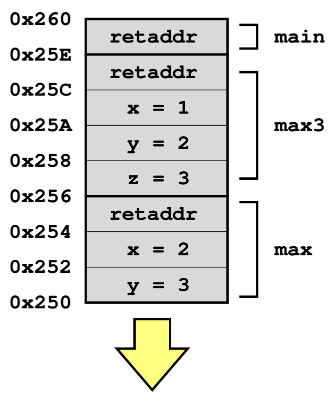
In this assignment, you will develop a *static analysis* for AVR programs, such as those which run on the Tiny-Boy emulator. This will determine whether or not dynamically allocated memory could exceed the available resources. Safety-critical standards (e.g. DO-178B or ISO-26262) mandate upper bounds be determined on the amount of storage space used. This is because, when memory resources are exhausted, the program will begin to behave in an unpredictable fashion as *memory corruption* occurs.

Stack Usage Analysis

The focus of this assignment is on determining the maximum amount of memory required for the *call stack* during any execution of the target program. This is more commonly known as *stack usage analysis* and has been studied extensively in the research literature [8, 6, 9, 4, 7, 3, 5, 2].

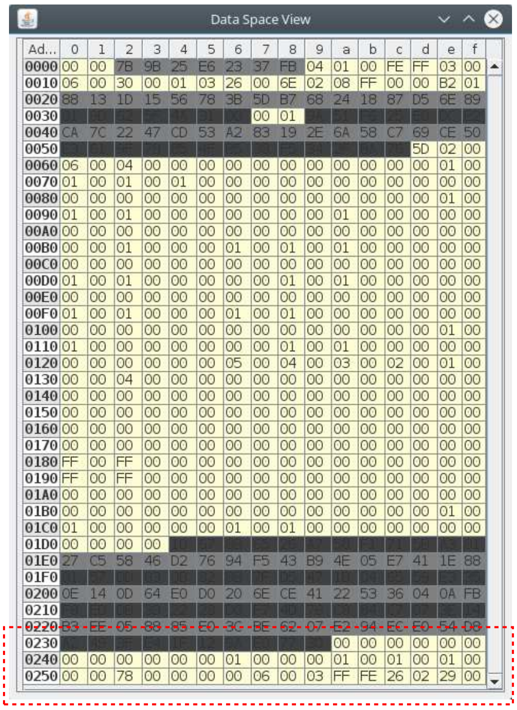
During the execution of a program a *stack frame* is created every time a method is executed. The stack frame typically contains (amongst other things) the current values of local variables and the *return address* (i.e. where execution should continue from when the method returns). As such, each stack frame occupies space on the stack. If too many stack frames are added, then the stack will *overflow* and lead to *memory corruption*. The following illustrates:

```
int max(int x, int y) {
  if(x > y) { return x; }
  else { return y; }
}

int max3(int x, int y, int z) {
  int xy = max(x,y); // max of x,y
  return max(xy,z);  // max of x,y,z
}

int main() { return max3(1,2,3); }
```



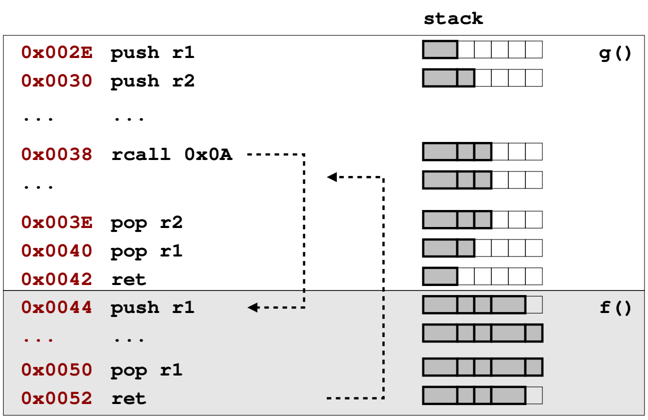
The diagram on the right illustrates the state of the stack during the second invocation of method `max(int,int)`. The stack grows downwards from the highest possible memory location (`0x25F`). In this case, each element on the stack occupies two bytes (i.e. is a 16bit value). The diagram also shows the actual value for each program variable stored on the stack for reference.

On an AVR microcontroller, the stack is placed at the end of available memory. On a small microcontroller, such as the ATtiny85, the amount of available memory is very small (i.e. 512 bytes) and the stack can easily overflow. The following illustrates the memory usage when executing a simple program on an ATtiny85:



Here, the yellow boxes indicate memory cells (i.e. bytes) which are currently in use by the program. Furthermore, the dashed red line indicates the region currently used for the stack. The dark cells in between the stack and the rest of memory represent the remaining space into which the stack can expand before memory corruption occurs.

The *stack analysis* you will develop in this assignment is a *control-flow based analysis*. This means it follows the flow of control through the program. The following illustrates:



The analysis follows the program instructions starting from address `0x0000`. In the above example, when it arrives at location `0x002E` there is only *one* item on the stack. This is the *return address* for the invocation which got us to this point. Furthermore, observe that a return address occupy's *two bytes* on the stack. In contrast, a `push` instruction adds only *one byte* to the stack.

In the above example, the analysis keeps track of the size of the stack as it traverses through each instruction. For example, after a `push r1` instruction we see another element has been added to the stack. When the analysis reaches the `rcall` instruction it must jump to the specified address (which is `0x003A + 0x0A = 0x0044`). Upon reaching the `ret` instruction, the analysis returns to the instruction following the `rcall` and continues from there.

Getting Started. To get started, download the accompanying software `avr-stackanalysis.tgz`, and import the Java code into Eclipse as appropriate. In addition, you will need to add the libraries `javr-1.X.jar` as external archives.

HINT: you will find the "AVR Instruction Set Manual" to be *invaluable* for this assignment [1]!

Part 1 — Simple Sequences (worth 10%)

The first part of the assignment is to implement simple instruction sequences. That is, instructions which execute in a linear fashion and do not fork control-flow. Of particular interest here are the `push` and `pop` instructions, as well as the unconditional branching instructions `jmp` and `rjmp`. **You should consult the AVR Instruction Set Manual to find out more about what these instructions do**.

HINT: the implementation for `rjmp` instructions has been given for you.

Part 2 — Method Invocations (worth 10%)

The second part of the assignment is to implement instructions (i.e. `call`, `rcall`, `ret`) which enable method invocation. Remember that, when executing a `call` or `rcall` instruction, the location of the *following instruction* is pushed on to the stack. Likewise, when executing a `ret` instruction, this location is popped off the stack. **Remember that the return address is always a 16bit value**! Also, note that we are ignoring the possibility of recursion for now.

The suggested approach for handling method invocation is as follows:

- **Split**. When a `call` or `rcall` instruction is encountered, start a traversal from that address. When the traversal is complete, continue the current traversal from the instruction following the `call` or `rcall` instruction.
- **Terminate**. When a `ret` instruction is encountered, simply terminate the current traversal. This works because above mechanism will ensure execution is picked after the `call` or `rcall` which lead here.

Part 3 — Conditional Branches (worth 20%)

The third part of the assignment is to implement instructions which enable conditional control flow (i.e. `breq`, `brge`, `sbrs`, `sbrc`). As with `call` instructions, these are challenging because they require your traversal to be split in two. **Care must also be taken with the skip instructions as these rely on the *width* of the following instructions to work properly.**

Part 4 — Recursion and Loops (worth 20%)

The fourth path of the assignment is to add support for both *recursion* and *loops*. This is challenging because a naive implementation of the stack analysis will cause a `StackOverflowException` when encountering a loop. There are two cases to account for when the current traversal encounters an instruction it has visited before:

- **Stable Stack Height**. On encountering an instruction previously seen, can terminate current traversal if stack height unchanged.
- **Unstable Stack Height**. On encountering an instruction previously seen, must make a *worst case assumption* if stack height differs (e.g. it is growing).

For example, recursion always results in a growing stack because every invocation puts more items on the stack. The worst case assumption is simply that an *infinite* amount of memory can be used. This is represented in the analysis by the constant `Integer.MAX_VALUE`.

Part 5 — TinyBoy Programs (worth 40%)

The final part of the assignment is to develop your analysis so that it can work on realistic programs. Specifically, we will be using the firmware programs (e.g. `snake.hex`, `tetris.hex`, etc) developed for the TinyBoy emulator. **The challenge here is to ensure as many AVR instructions are supported by your analysis as possible.**

Submission

Your assignment solution should be submitted electronically via the *online submission system*, linked from the course homepage. The following files are required for submission:

`avranalysis/core/StackAnalysis.java`

You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. *Note, the jar file does not need to be executable*. See the following Eclipse tutorials for more on this:

   `http://ecs.victoria.ac.nz/Support/TechNoteEclipseTutorials`

2. **The names of all classes, methods and packages remain unchanged**. That is, you may add new classes and/or new methods and you may modify the body of existing methods. However, you may not change the name of any existing class, method or package. *This is to ensure the automatic marking script can test your code*.
3. **The testing mechanism supplied with the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. However, this does not prohibit you from adding new tests. *This is to ensure the automatic marking script can test your code*.
4. **You have removed any debugging code that produces output, or otherwise affects the computation.** *This ensures the output seen by the automatic marking script does not include spurious information*.

Note: Failure to meet these requirements could result in your submission being reject by the submission system and/or zero marks being awarded.

Assessment

Marks for this assignment will be awarded based on the number of passing tests, as determined by the *automated marking system*. **NOTE:** the test cases used for marking will differ from those in the supplementary code provided for this assignment. In particular, they may constitute a more comprehensive set of test cases than given in the supplementary code.

Appendix — Making your Own Firmware Images

Firmware images for the TinyBoy are written in the C programming language and compiling using the `avr-gcc` tool chain. This is installed on the lab machines, though you will need to install yourself on your laptop in order to use it there.

The process of creating your own firmware is relatively easy and the accompanying software includes the source of all firmware images in the *ROMS* directory, along with an appropriate makefile for compiling them to `hex` files.

References

[1] AVR instruction set manual, 2016. `http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf`.

[2] avstack.pl, daniel beer, `https://www.dlbeer.co.nz/oss/avstack.html`, accessed 2017.

[3] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms. *MONET*, 10(4):563–579, 2005.

[4] D. Brylow, N. Damgaard, and J. Palsberg. Static checking of Interrupt-Driven software. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE)*, pages 47–56. IEEE Computer Society, 2001.

[5] Krishnendu Chatterjee, Di Ma, Rupak Majumdar, Tian Zhao, Thomas A. Henzinger, and Jens Palsberg. Stack size analysis for interrupt-driven programs. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 109–126. Springer, 2003.

[6] Daniel Kästner and Christian Ferdinand. Proving the absence of stack overflows. In *SAFECOMP*, volume 8666, pages 202–213. Springer, 2014.

[7] William P. McCartney and Nigamanth Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys 2006, Boulder, Colorado, USA, October 31 - November 3, 2006*, pages 167–180. ACM, 2006.

[8] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computing Systems*, 4(4):751–778, 2005.

[9] Bastian Schlich. Model checking of software for microcontrollers. *ACM Transactions on Embedded Computing Systems*, 9(4):36:1–36:27, 2010.