Running Soot and Flowdroid

Johnathan Lemon

Introduction:

Call graphs are a powerful tool that can be used to not only visualize, but also analyze data. There are many tools that utilize them such as Flowdroid and CAT. These tools can be used to analyze applications by having the call graph used to analyze the flows of the application and then build upon it. For instance, FlowDroid uses the call graph to find flows released in an update and then test said flow in the code for bugs. Having never worked with them, I was curious how easy it would be to set up the tools, and then also see how well the visual portion of it was as well. I used the tool called Soot which is the backbone of many call graph generations and then also ran Flowdroid which utilizes soot to analyze code.

Motivation:

I have always been interested in testing. Be it for work or school projects something about running a test and seeing green while analyzing aspects like coverage makes me marvel at the possibilities. I started by trying to run QAdroid and recreating and analyzing; however, it did not work very well, and I was unable to get a valid run and the information needed due to errors with the android environment. I then wanted to analyze and recreate a tool known as Flowdroid to see some of the underlying technologies that QAdroid utilizes. However, the tool did not output the expected behavior and I was unable to see a visual representation of the call graph that was being made. This led me to recreating the tool known as soot by following a tutorial to see the output and visualization of the call graph that Flowdroid would be using. This project was mainly for me to gain insight and experience with the tool.

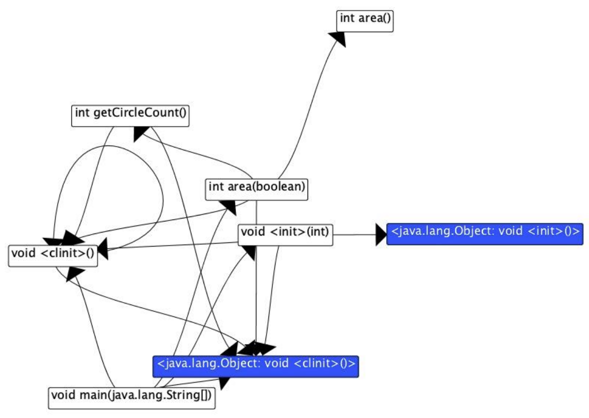
Methodology:

I originally ran the tool Flowdroid. It was very simple to run using their documentation at https://github.com/secure-software-engineering/FlowDroid. It was very simple and just needed an APK to be passed in via the command line. This did not provide meaningful output. I then ran a Soot tutorial that can be found at https://github.com/noidsirius/SootTutorial. I originally planned to run this tool on windows. This did not work however as acquiring the java 8 jdk was not convenient and running the tool VIA docker ran into issues of not having an X11 image. This may be due to me not configuring the systems. I then went to run the tool via mac and it worked as expected. I simply needed to homebrew the java 8 jdk, clone the repo, build the tool, and then run the run command with certain args to get the desired result.

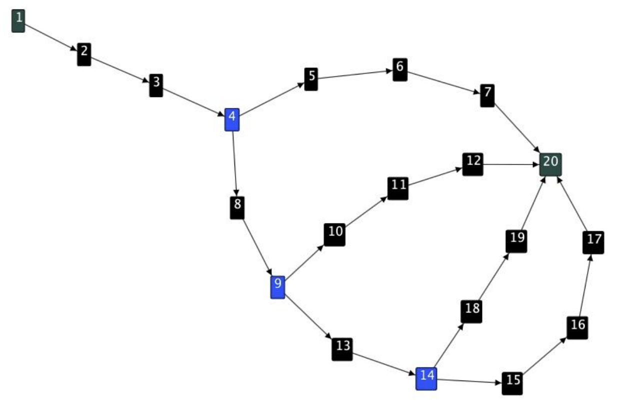
Findings:

I set out to see if the tools would be 1) easy to use 2) produce useful results. The tool provided great visual representations I can say that the tools were easy to use; however, there are some issues. The first is that the tools was not simple to run on windows at all. It was difficult to get the right environments and analzye them for the soot tutorial; however, for Flowdroid it was very straightforward. Running Soot on mac was incredibly simple and well documented. I can say that flowdroid did not make useful information, but did generate call graphs but could not be visualized. Recretaing the Soot flow made very useful maps of the program and also generated very helpful visual representations of call graphs and control flow graphs.

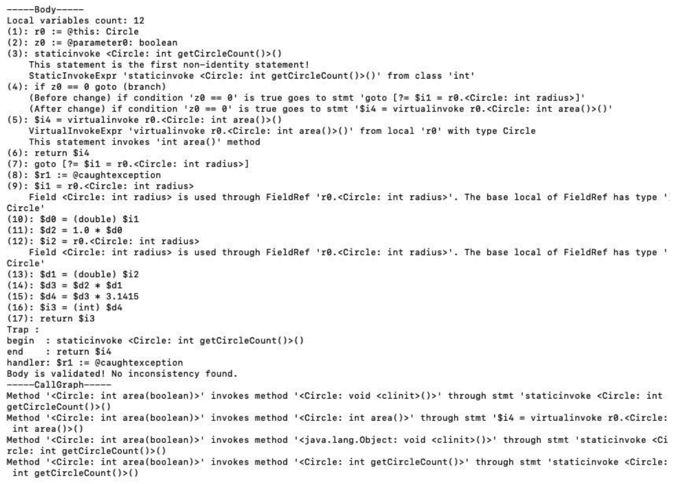
Example outputs:


int area()
int getCircleCount()
int area(boolean)
void <init>(int)
<java.lang.Object: void <init>()>
void <clinit>()
<java.lang.Object: void <clinit>()>
void main(java.lang.String[])

The left is an example of a call-graph generated. In my opinion, the graph provides an insightful visualization of the flows of the program.

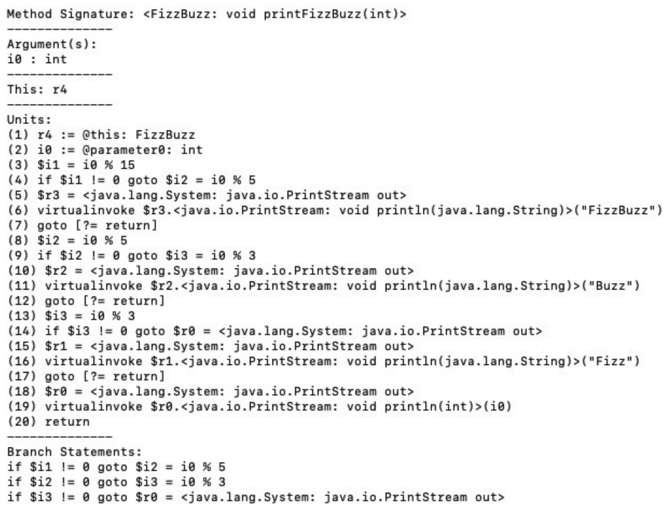


The left is an example of a control flow graph. Again, the tool provides a useful graph as well as a visualization.



```
-----Body-----
Local variables count: 12
(1): r0 := @this: Circle
(2): z0 := @parameter0: boolean
(3): staticinvoke <Circle: int getCircleCount()>()
    This statement is the first non-identity statement!
    StaticInvokeExpr 'staticinvoke <Circle: int getCircleCount()>()' from class 'int'
(4): if z0 == 0 goto (branch)
    (Before change) if condition 'z0 == 0' is true goes to stmt 'goto [?= $i1 = r0.<Circle: int radius>]'
    (After change) if condition 'z0 == 0' is true goes to stmt '$i4 = virtualinvoke r0.<Circle: int area()>()'
(5): $i4 = virtualinvoke r0.<Circle: int area()>()
    VirtualInvokeExpr 'virtualinvoke r0.<Circle: int area()>()' from local 'r0' with type Circle
    This statement invokes 'int area()' method
(6): return $i4
(7): goto [?= $i1 = r0.<Circle: int radius>]
(8): $r1 := @caughtexception
(9): $i1 = r0.<Circle: int radius>
    Field <Circle: int radius> is used through FieldRef 'r0.<Circle: int radius>'. The base local of FieldRef has type '
Circle'
(10): $d0 = (double) $i1
(11): $d2 = 1.0 * $d0
(12): $i2 = r0.<Circle: int radius>
    Field <Circle: int radius> is used through FieldRef 'r0.<Circle: int radius>'. The base local of FieldRef has type '
Circle'
(13): $d1 = (double) $i2
(14): $d3 = $d2 * $d1
(15): $d4 = $d3 * 3.1415
(16): $i3 = (int) $d4
(17): return $i3
Trap :
begin  : staticinvoke <Circle: int getCircleCount()>()
end    : return $i4
handler: $r1 := @caughtexception
Body is validated! No inconsistency found.
-----CallGraph-----
Method '<Circle: int area(boolean)>' invokes method '<Circle: void <clinit>()>' through stmt 'staticinvoke <Circle: int
getCircleCount()>()
Method '<Circle: int area(boolean)>' invokes method '<Circle: int area()>' through stmt '$i4 = virtualinvoke r0.<Circle:
 int area()>()
Method '<Circle: int area(boolean)>' invokes method '<java.lang.Object: void <clinit>()>' through stmt 'staticinvoke <Ci
rcle: int getCircleCount()>()
Method '<Circle: int area(boolean)>' invokes method '<Circle: int getCircleCount()>' through stmt 'staticinvoke <Circle:
 int getCircleCount()>()
```

The left are examples of the units of jimple code which is essentially a better visualization of java code. I think this is once again of very useful output to help visualize the code and how it works.



```
Method Signature: <FizzBuzz: void printFizzBuzz(int)>
--------------
Argument(s):
i0 : int
--------------
This: r4
--------------
Units:
(1) r4 := @this: FizzBuzz
(2) i0 := @parameter0: int
(3) $i1 = i0 % 15
(4) if $i1 != 0 goto $i2 = i0 % 5
(5) $r3 = <java.lang.System: java.io.PrintStream out>
(6) virtualinvoke $r3.<java.io.PrintStream: void println(java.lang.String)>("FizzBuzz")
(7) goto [?= return]
(8) $i2 = i0 % 5
(9) if $i2 != 0 goto $i3 = i0 % 3
(10) $r2 = <java.lang.System: java.io.PrintStream out>
(11) virtualinvoke $r2.<java.io.PrintStream: void println(java.lang.String)>("Buzz")
(12) goto [?= return]
(13) $i3 = i0 % 3
(14) if $i3 != 0 goto $r0 = <java.lang.System: java.io.PrintStream out>
(15) $r1 = <java.lang.System: java.io.PrintStream out>
(16) virtualinvoke $r1.<java.io.PrintStream: void println(java.lang.String)>("Fizz")
(17) goto [?= return]
(18) $r0 = <java.lang.System: java.io.PrintStream out>
(19) virtualinvoke $r0.<java.io.PrintStream: void println(int)>(i0)
(20) return
--------------
Branch Statements:
if $i1 != 0 goto $i2 = i0 % 5
if $i2 != 0 goto $i3 = i0 % 3
if $i3 != 0 goto $r0 = <java.lang.System: java.io.PrintStream out>
```

Possible Improvements:

Since this was more of an exploration due to other ideas having unexpected issues, my main and only improvement would be documentation. All of these tools can be vague and often have errors when trying to recreate and often have weird solutions that they don’t talk about at all. They are powerful tools, but they definitely have more to be desired in the department of documentation and ease of use. Overall, the tools were very useful and easy to use after initial bumps and the tools could provide more documentation for debugging.

Threats to validity:

The main threat to validity is that this was my first experience these technologies. I may have messed up running the tools (for example running soot on windows and docker) and this lack of experience may lead to bad findings that are my own personal fault and not the tools.