

Understanding smali

Lecture 2

CSCI 4450 & CSCI 6670

Dr. Xiaolu Zhang & Dr. Frank Breitingner

Objectives

- Memorize and assess smali code.

Overview

- Smali is an intermediate representation which follows Jasmin syntax
 - Byte code → Smali → Java Code
- Smali code is similar to **Dalvik instructions (opcodes)**
 - http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html
- Smali code is generated by reorganizing the distributed information in DEX file
- Smali code is class-based; each of the **class is stored in a smali file** (inner class is stored independently)

smali code example [1]

```
.class public LHelloWorld;
```

```
.super Ljava/lang/Object;
```


```
.method public static main([Ljava/lang/String;)V  
    .registers 2
```

```
    sget-object v0, Ljava/lang/System;.>out:Ljava/io/PrintStream;
```

```
    const-string v1, "hello,world!"
```

```
    invoke-virtual {v0, v1}, Ljava/io/PrintStream;.>println(Ljava/lang/String;)V
```

```
    return-void  
.end method
```



Let us discuss it line
by line.

smali code example [2]

.class public LHelloWorld;

.super Ljava/lang/Object;

.method public static main([Ljava/lang/String;)V

.registers 2

sget-object v0, Ljava/lang/System; -> out:Ljava/io/PrintStream;

const-string v1, "hello,world!"

invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/String;)V

return-void

.end method

L1: is the class declaration
like we know it from Java
- "L" is explained later

L2: every class that does not
extend another class
automatically extends the
„object“ class

smali code example [3]

```
.class public LHelloWorld;

.super Ljava/lang/Object;

.method public static main([Ljava/lang/String;)V
    .registers 2

    sget-object v0, Ljava/lang/System;:->out:Ljava/io/PrintStream;

    const-string v1, "hello,world!"

    invoke-virtual {v0, v1}, Ljava/io/PrintStream;:->println(Ljava/lang/String;)V

    return-void
.end method
```

L3: a public method “main”
which has a parameter:

- parameter is a string
 - [indicates that it is an array
 - L indicates it is a class (not a native type like int)
- parameter is a string array!

V at the end indicates the
type of the return value
which is here void.

Data types in Dalvik

Syntax	Meaning
V	<code>void</code> ; only valid for return types
Z	<code>boolean</code>
B	<code>byte</code>
S	<code>short</code>
C	<code>char</code>
I	<code>int</code>
J	<code>long</code>
F	<code>float</code>
D	<code>double</code>
<code>Lfully/qualified/Name;</code>	the class <code>fully.qualified.Name</code>
<code>[descriptor</code>	array of <code>descriptor</code> , usable recursively for arrays-of-arrays, though it is invalid to have more than 255 dimensions.

Let's test your knowledge

- B
- C
- L/java/lang/string
- [I
- [L/java/lang/Object
- [V

smali code example [4]

```
.class public LHelloWorld;

.super Ljava/lang/Object;

.method public static main([Ljava/lang/String;)V
    .registers 2

    sget-object v0, Ljava/lang/System;-.>out:Ljava/io/PrintStream;

    const-string v1, "hello,world!"

    invoke-virtual {v0, v1}, Ljava/io/PrintStream;-.>println(Ljava/lang/String;)V

    return-void
.end method
```

L4: registers 2 means that this functions needs 2 registers (think of it like variables for now). Depending on the settings of the compiler, this might not always be shown.

Register are indicate by vX.

smali code example [5]

```
.class public LHelloWorld;

.super Ljava/lang/Object;

.method public static main([Ljava/lang/String;)V
    .registers 2

    sget-object v0, Ljava/lang/System;:->out:Ljava/io/PrintStream;

    const-string v1, "hello world!"

    invoke-virtual {v0, v1}, Ljava/io/PrintStream;:->println(Ljava/lang/String;)V

    return-void
.end method
```

Remember "L" is the data type
and requires a
fully.qualified.classname

L5: in short, it stores a
printstream object in the
value register 0 (v0).

- *sget-object* is an opcode name, it reads the object reference into the value register v0.
- *Ljava/lang/System* is the class
- *out* is a (static) field of the class
- *Ljava/io/PrintStream* is the return type / type of *out*

In short, we will print something later

smali code example [6]

```
.class public LHelloWorld;

.super Ljava/lang/Object;

.method public static main([Ljava/lang/String;)V
    .registers 2

    sget-object v0, Ljava/lang/System;:->out:Ljava/io/PrintStream;

    const-string v1, "hello,world!"

    invoke-virtual {v0, v1}, Ljava/io/PrintStream;:->println(Ljava/lang/String;)V

    return-void
.end method
```

L6: saves hello world in v1.

- const-string is an opcode name again
- it puts a reference to a string constant into v1.

smali code example [7]

```
.class public LHelloWorld;

.super Ljava/lang/Object;

.method public static main([Ljava/lang/String;)V
    .registers 2

    sget-object v0, Ljava/lang/System;.->out:Ljava/io/PrintStream;

    const-string v1, "hello,world!"

    invoke-virtual {v0, v1}, Ljava/io/PrintStream;.->println(Ljava/lang/String;)V

    return-void
.end method
```

What is "L" again?

What is "V" again?

L7: finally prints the string

- *invoke-virtual* invokes a virtual method with parameters
- v0 in the brackets is like "this" (printstream) from a java class, thus we are calling "out" object.
- v1 is then the parameter (our string) that we pass.
- Ljava/io/PrintStream is the class we call
- println is the function we call which accepts 1 argument (a string)

Field vs. Function

- Let's compare the lines 5 and 7:
 - L5: [...] Ljava/lang/System;->out:Ljava/io/PrintStream;
 - L7: [...] Ljava/io/PrintStream;->println(Ljava/lang/String;)V
- *out* does not have brackets and there is no return type at the end of the line
- in contrast *println* is followed by (<parameter>) and as the return type void at the end.

smali code example [8]

```
.class public LHelloWorld;

.super Ljava/lang/Object;

.method public static main([Ljava/lang/String;)V
    .registers 2

    sget-object v0, Ljava/lang/System;.->out:Ljava/io/PrintStream;

    const-string v1, "hello,world!"

    invoke-virtual {v0, v1}, Ljava/io/PrintStream;.->println(Ljava/lang/String;)V

    return-void
.end method
```

L8: indicates the return value which is void.

L9: indicates the end of the method.

To assemble / run it on your phone/ emulator

- `java -jar smali.jar -o classes.dex HelloWorld.smali`
- `zip HelloWorld.zip classes.dex`
- `adb push HelloWorld.zip /data/local`
- `adb shell dalvikvm -cp /data/local/HelloWorld.zip HelloWorld`
- if you get out of memory type errors when running `smali.jar`
 - give java more memory with `-Xmx512m`, like this:
 - `java -Xmx512m -jar smali.jar HelloWorld.smali`

Exercise

- Generate the java code for the given example.

Dalvik Registers [1]

- Registers are always 32 bits and can hold any type of value
 - 2 registers are used to hold 64 bit types (Long and Double)
- Two ways to specify how many registers are available in a method:
 1. number of registers in a method can be specified using `.registers` which specifies the total number of registers.
 - E.g., `registers 2` in line 2 of the example
 2. or `.locals` directive which specifies the number of non-parameter registers in the method
 - The total number of registers would therefore include the registers needed to hold the method parameters.

Dalvik Registers [2]

- How method parameters are passed into a method?
 - Parameters of the method are placed into the last n registers.
 - I.e. given a method with 2 args and 5 registers (v0-v4), the arguments are placed in registers v3 and v4
 - The first parameter to a non-static methods is always the object that the method is being invoked on
 - in Java this means “this”
 - For static methods it's the same thing, except there isn't an implicit this argument.

Dalvik Registers [3]

Register names

- 2 naming schemes for registers:
 - the normal v naming scheme
 - the p naming scheme for parameter registers.
 - first register in the p naming scheme is the first parameter register in the method

Dalvik Registers - Example

- For example, a non-static method `LMyObject;->callMe(II)V` that specifies 5 registers in the method (`v0-v4`):
 - `.registers 5` or `.locals 2`
 - `v0` and `v1` are method internally; `v2-v4` are the parameters
 - `v3` = first integer parameter; `v4` = second integer parameter
 - `v2` = is an implicit `LMyObject` (“this” in java)
- Giving the name scheme, that means:
 - `v2=p0`; `v3=p1`; `v4=p2`

Local	Param	
v0		the first local register
v1		the second local register
v2	p0	the first parameter register
v3	p1	the second parameter register
v4	p2	the third parameter register

Dalvik Registers – Long/Double

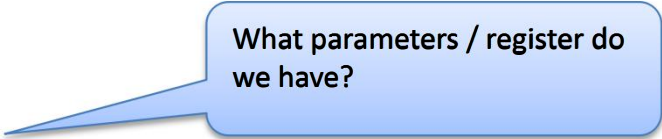
- long (J) and double (D) primitives are 64 bit values, and require 2 registers.
 - This is important to keep in mind when you are referencing method arguments.
 - For example, let's say you have a (non-static) method `LMyObject;->MyMethod(IJZ)V`.
 - The parameters to the method are `LMyObject;`, `int`, `long`, `bool`. So this method would require 5 registers for all of its parameters.

Register	Type
p0	this
p1	I
p2, p3	J
p4	Z

Examples

- Ex1:

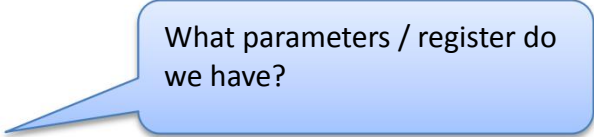
.method public getTokens(I)I
.locals 2



What parameters / register do we have?

- Ex2:

.method static getGoodGrade(BD)I
.registers 5



What parameters / register do we have?

Opcodes in Dalvik

- Opcodes determine the format of the instruction
- Opcodes can be classified into 13 groups
 - discussed over the next slides
- Opcodes can have 1-3 parameters
- Note, the slides will only provide an overview, we cannot discuss all in detail (see references).

Opcodes [1]

- **Const**, e.g., `const vAA, #+BBBBBBBBBB`
 - Moves a constant `#+BBBBBBBBBB` (or string, int, class) into `vAA`
- **Move**, e.g., `move vA, vB`
 - Moves `vB` into `vB`
 - Note, there are also move opcodes with only one operand
 - e.g., `move-exception vAA` saves a just-caught exception into the given register.

Opcodes [2]

- ***Instance***, e.g., new-instance vx,type
 - Instantiates an object type and puts the reference of the newly created instance into vx.
- ***Array***, e.g., new-array vx,vy,type_id
 - Generates a new array of type_id type and vy element size and puts the reference to the array into vx.

Opcodes [3]

- ***Calculations***, e.g., add-int vx,vy,vz
 - Calculates $vy + vz$ and puts the result into vx.
- ***Compare***, e.g., cmpg-float vx, vy, vz
 - Compares the float values in vy and vz and sets the integer value in vx accordingly.
 - if $vy > vz$ then vx is positive; if $vy == vz$ then vx = 0; else negative

Opcodes [4]

- ***Field***, e.g., `iget-boolean vx,vy,field_id`
 - Reads a boolean instance field into vx. The instance is referenced by vy.
- ***Invoke call***, e.g.,
`invoke-virtual { parameters }, methodToCall`
 - Invokes a virtual method with parameters.

Opcodes [5]

- ***Return***, e.g., return-object vx
 - Return with vx object reference value.
- ***Synchronization***, e.g., monitor-enter vx
 - Obtains the monitor of the object referenced by vx.
- ***Exception***, e.g., throw vx
 - Throws an exception object. The reference of the exception object is in vx.

Opcodes [6]

- ***Jump***, e.g., if-eq vx,vy,target
 - Jumps to target if $vx == vy$. vx and vy are integer values.
- ***Cast***, e.g., int-to-long vx, vy
 - Converts the integer in vy into a long in vx,vx+1.

Structure of smali files

- As previously mentioned, every smali file indicates one **class / inner class**.
- In the following we will have a brief look at how smali files are structure

Terminology

- #<access modifiers>:
 - public, private, protected or default.
- #[non-access modifiers]:
 - static, final, abstract, synchronized or volatile.

Methods structure

see next slide

#direct methods / #virtual methods

.method <Access Modifiers> [Non-Access Modifiers] **Methodname (parameters) return value**

<.locals>

#how many local variables in a method; could be .registers

[.parameter]

#indicates the name of parameter(s) in a method

[.prologue]

#the code start from here

[.line]

#line number in source code

<Instructions (opcodes)>

.end

Tags in brackets are optional;
the compiler might not include
them!

Example was discussed
in the beginning

Examples

Methodname (parameter)return value

java code: `void doVoodoo(int puppets){...}`

#we call a method inside the same class

smali code: `doVoodoo(I)V`

java code: `char charAt(int index){...}`

#calling a method outside the class

smali code: `Ljava/lang/String;->charAt(I)C`

#if outside, class path is needed!

java code: `void getChars(int srcBegin,int srcEnd,char dst[],int dstBegin)`

smali code: `Ljava/lang/String;->getChars(II[CI)V`

note [C indicates the char array

java code: `boolean equals(Object anObject){...}`

smali code: `Ljava/lang/String;->equals(Ljava/lang/Object)Z`

#objects as parameters also need path

Class structure

.class <access modifiers> [non-access modifiers] <class name>

.super <name of super class>

.source <name of the source file>

.implements <name of the interface>

.annotation system <annotation class>

value = {...}

.end annotation

.field <access modifiers> [non-access modifiers] <field name>:<type>

Class structure - example

```
.class public Lcom/social_touch/demo/MainActivity;  
.super Landroid/app/Activity;  
.source "MainActivity.java"
```

interfaces

```
.implements Landroid/view/View$OnClickListener;
```

static fields

```
.field private static final pi:F = 3.14f
```

Comparing smali and Java

smali code

```
# virtual methods
.method public add(II)V
    .locals 1
    .parameter "x"
    .parameter "y"
    .prologue
    .line 34
    add-int v0, p1, p2
    return v0
.end method
```

Java code

```
public int add(int x, int y) {
    return x + y;
}
```

The more tricky parts

- Loops
 - For loop, While loop, Iterator loop
- Switch
 - if else, Switch case
- Try/catch

For loop

smali

```
const/4 v5, 0x5
.local v5, size:l
const/4 v1, 0x0
.local v1, i:l
:goto_0
    if-lt v5,v1, :cond_0
    ...
    add-int/lit8 v1,v1,0x1
    goto    :goto_0
:cond_0
```

java

```
int size = 5;
for(int i=0; i<size; i++){
    ...
}
```

While loop

smali

:goto_0

```
invoke-interface {v4}, Ljava/util/Iterator;-  
>hasNext ()Z
```

...

```
invoke-interface {v4}, Ljava/util/Iterator;-  
>next()Ljava/lang/Object;
```

goto **:goto_0**

Java

```
Iterator iterator= data.iterator();  
while (iterator.hasNext() && (String)  
        iterator.next().startsWith("f"))  
{  
    ...  
}
```

Switch

smali

```
.method public getMsgString(I)Ljava/lang/String;
    .locals 1
    const-string v0, "Hello,switch"
    packed-switch p1, :pswitch_data_0
        :goto_0
            return-object v0
        :pswitch_0
            const-string v0, "Hello,switch 0"
            goto :goto_0
        :pswitch_1
            const-string v0, "Hello,switch 1"
            goto :goto_0
        :pswitch_data_0
        .packed-switch 0x0
        :pswitch_0 #0x0
        :pswitch_1 #0x1
        .end packed-switch
    .end method
```

Java

```
String getMsgString(int i){
    String str = null;
    switch(i){
        case 0:
            str = "Hello,switch 0";
        case 1:
            str = "Hello,switch 1";
        default:
            str = "Hello,switch";
    }
    return str;
}
```


Try / catch

```
:try_start
    invoke-static {v1}, Ljava/lang/Integer;->parseInt(Ljava/lang/String;)I
    move-result v1
:try_end
.catch Ljava/lang/Exception;{:try_start .. :try_end} :catch_0
    :goto_0
    return-void

:catch_0
    move-exception v0
    invoke-virtual {v0}, Ljava/lang/Exception;->toString()Ljava/lang/String;
    move-result-object v0
    invoke-virtual {p0, v0}, Lcom/alipay/helloworld/MainActivity;->showToastMessage(Ljava/lang/String;)V
    goto :goto_0
```

Comprehensive example

- Let's see what we understand so far:
- <http://androidcracking.blogspot.com/2010/09/examplesmali.html>

Hands-on example

- Converting smali code to DEX file by using the tool “Smali”, then execute the DEX on an Android device / emulator.
 - Smali tool can be downloaded from <https://bitbucket.org/JesusFreke/smali/downloads>
 - “java -jar smali.jar assemble Main.smali” for generating the DEX file.
 - Push DEX to device.
 - To run: `dalvikvm -cp <DEX file> <Main method>`

Hands-on example (cont'd)

Smali code injection

- Inject the helloWorld function into the MethodOverloading class
- Thus “hello world” can be printed out before “42”.
- Some tips:
 - Write the injection function in java code in order to get the correct smali code.
 - Copy the function to the target smali file and call the function by “invoke”.
 - Do not forget to change the register number.

LAB 02

Practice (1): read smali code

- Each team will get one page of smali code.
- Discuss with your teammate and translate the smali code to Java (or other programming language you are familiar with).

Lab 02 – Task 01

- Convert the smali code (task 01.txt) to JAVA.
 - Copy the Java code into your report; please enable source code highlighting.
 - We recommend to write in a Java environment to avoid (syntax) errors.
 - Tools are not allowed.

Lab 02 – Task2 – Overview

- Hijacking the user name and password in “Instagram” application.
 - The Instagram APK file can be downloaded from <https://apkpure.com/>
- Description:
 - Repack the APK file and Inject code in proper smali file(s) for hijacking the user name and password.
 - Once the user who has installed the repacked “Instagram” clicks the “log in” button on the login panel (launch the app, click “already have an account”) , the username and password will be hijacked and sent to a private server.

Lab 02 – Task2 – Requirements

- Provide the repacked Instagram APK file.
- Write the lab report and document the entire procedure with screen shots. The report should answer the following questions:
 - Where is the code injected?
 - How did you find the appropriate location for injection?
 - How is your injected code structured / what does it do?
 - Where do you store the hijacked username and password?
- Provide the link of the web page that displays the hijacked username and password (whenever the user types in their credential on the fake Instagram, it will be showed on this page).
 - The Server can be shared between teams.