Reconstructing Dalvik applications

Marc Schönefeld

University of Bamberg

Confidence 09, Krakow, PL

Agenda

1. Introduction
2. Dalvik development from a RE perspective
3. Parsing Strategy
4. Processing the results
5. Finalizing

The Speaker

Marc Schönefeld

- since 2002 Talks on Java-Security on intl. conferences (Blackhat, RSA, DIMVA,PacSec, CanSecWest, HackInTheBox)
- day time busy for Red Hat (since 2007)
- PhD Student at University of Bamberg (since 2005)

Motivation

- As a reverse engineer I have the tendency to look in the code that is running on my mobile device
- Coming from a JVM background I wanted to know what Dalvik is really about
- Wanted to learn some yet another bytecode language
- I prefer coding to doing boring stuff, like filling out tax forms

What is Dalvik

- Dalvik is the runtime that runs userspace Android applications
- invented by Dan Bornstein (Google)
- named after a village in Iceland
- register-based
- runs own bytecode dialect (not java bytecode)

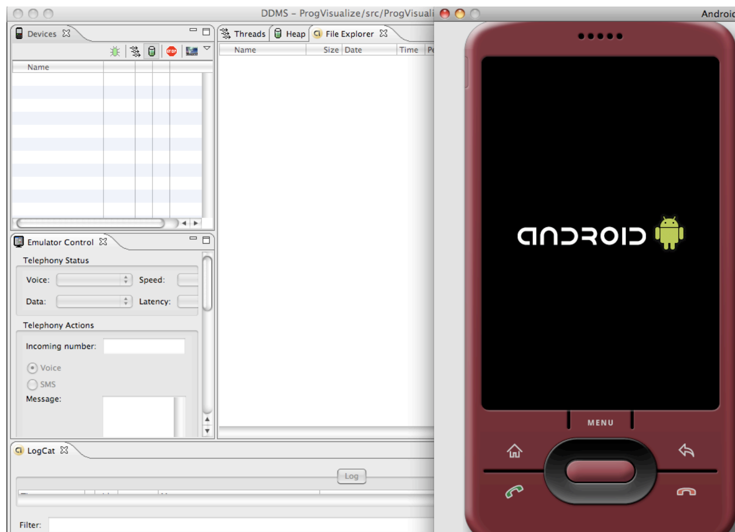
Dalvik vs. JVM

	Dalvik	JVM
Architecture	Register	Stack
OS-Support	Android	Multiple
RE-Tools	few	many
Executables	APK	JAR
Constant-Pool	per Application	per Class

Dalvik Development process

- Dalvik apps are developed using java developer tools on a standard desktop system (like eclipse),
- compiled to java classes (javac)
- transformed to DX with the dx tool (classes.dex)
- classes.dex plus meta data and resources go into a dalvik application ’apk’ container
- this is transferred to the device or an emulator (adb, or download from android market)

Dalvik Development process



Dalvik runtime libraries

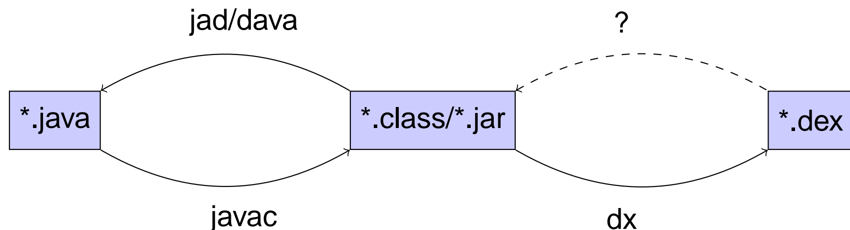
	Dalvik	JVM
java.io	Y	Y
java.net	Y	Y
android.*	Y	N
com.google.*	Y	N
javax.swing.*	N	Y
...	...	...

Dalvik development from a RE perspective

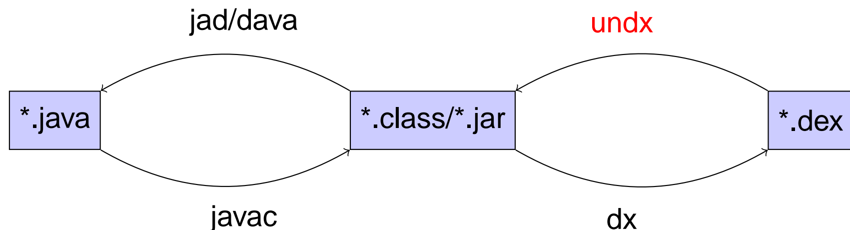
Dalvik applications are available as apk files, no source included, so you buy/download a cat in the bag. How can you find out, whether

- the application contains malicious code, ad/spyware, or phones home to the vendor ?
- has unpatched security holes (dex generated from vulnerable java code) ?
- contains copied code, which may violate GPL or other license agreements ?

Dalvik development from a RE perspective



Filling the gap



Tool design choices

- How to parse dex files?
  - write a complicated DEX parser
  - or utilize something existing
- How to translate to class files (bytecode library)?
  - ASM
  - BCEL

Parsing DEX files

- The dexdump tool of the android sdk can perform a complete dump of dex files, it is used by undx

	dexdump	parsing directly
Speed	Time advantage, do not have to write everything from	Direct access to binary structures (arrays, jump tables)
Control	dexdump has a number of nasty bugs	Immediate fix possible
Available info	Filters a lot	All you can parse
...	...	...

- The decision was to use as much of useable information from dexdump, for the rest we parse the dex file directly

Parsing DEX files

- This is useful dexdump output, good to parse

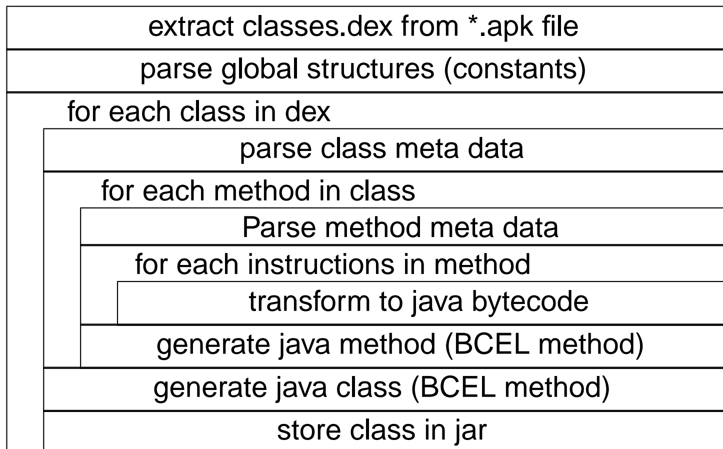
```
    #1              : (in LUnDxTest;)
      name          : 'main'
      type          : '([Ljava/lang/String;)V'
      access        : 0x0009 (PUBLIC STATIC)
      code          -
      registers     : 3
      ins           : 1
      outs          : 2
      insns size    : 20 16-bit code units
00024c:                          |[00024c] UnDxTest.main:([Ljava/lang/String;)V
00025c: 2200 0200                |0000: new-instance v0, LObj; // class@0002
000260: 7010 0000 0000           |0002: invoke-direct {v0}, LObj;.<init>:()V // method@0000
000266: 1251                     |0005: const/4 v1, #int 5 // #5
000268: 6e20 0200 1000           |0006: invoke-virtual {v0, v1}, LObj;.doit:(I)V // method@0002
00026e: 1301 ff00                |0009: const/16 v1, #int 255 // #ff
000272: 6e20 0200 1000           |000b: invoke-virtual {v0, v1}, LObj;.doit:(I)V // method@0002
000278: 1301 f000                |000e: const/16 v1, #int 240 // #f0
00027c: 6e20 0200 1000           |0010: invoke-virtual {v0, v1}, LObj;.doit:(I)V // method@0002
000282: 0e00                     |0013: return-void
```

Parsing DEX files

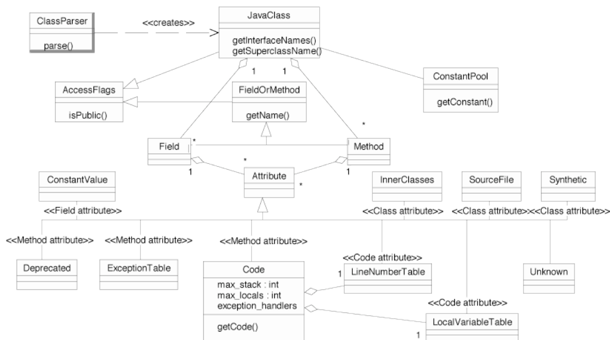
- This is useful dexdump output, omitting important data

```
name          : '<clinit>'
type          : '()V'
access        : 0x10008 (STATIC CONSTRUCTOR)
code          -
registers     : 1
ins           : 0
outs          : 0
insns size    : 34 16-bit code units
000310:                                        |[000310] MD5.<clinit>:()V
000320: 1200                                   |0000: const/4 v0, #int 0 // #0
000322: 6900 0200                              |0001: sput-object v0, LMD5;.md5:LMD5; // field@0002
000326: 1300 1000                              |0003: const/16 v0, #int 16 // #10
00032a: 2300 0f00                              |0005: new-array v0, v0, [C // class@000f
00032e: 2600 0700 0000                         |0007: fill-array-data v0, 0000000e // +00000007
000334: 6900 0000                              |000a: sput-object v0, LMD5;.hexChars:[C // field@0000
000338: 0e00                                   |000c: return-void
00033a: 0000                                   |000d: nop // spacer
00033c: 0003 0200 1000 0000 3000 3100 3200 ... |000e: array-data (20 units)
catches       : (none)
positions     :
0x0000 line=7
0x0003 line=8
```

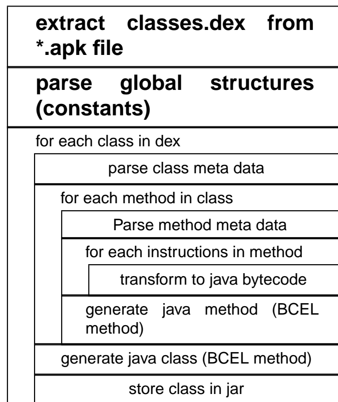
Strategy



- http://jakarta.apache.org/bcel/
- We chose the BCEL library from Apache as it has a very broad functionality (compared to alternatives like are ASM and javassist)

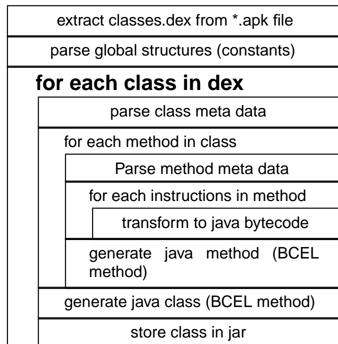


Structure



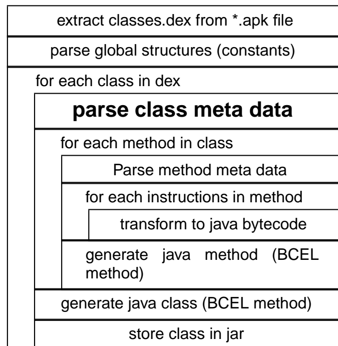
- Extract global meta information
- Transform into relevant BCEL constant structures
- Retrieve the string table to prepare the Java constant pool

Process classes



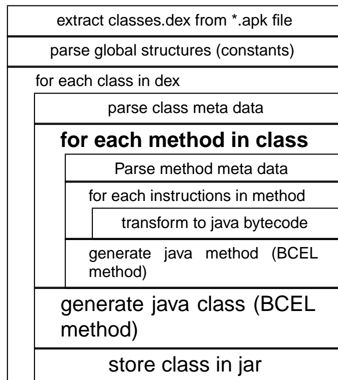
- Transform each class
- Parse Meta Data
- Process methods
- Generate BCEL class
- Dump class file

Process class Meta Data



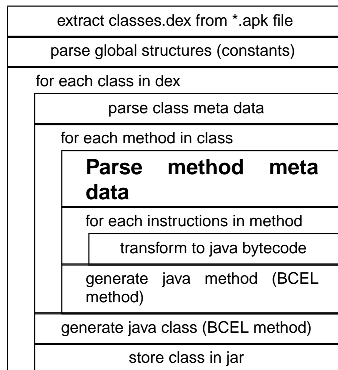
- Extract Class Meta Data
- Visibility, class/interface, classname, subclass
- Transfer static and instance fields

Process the individual methods



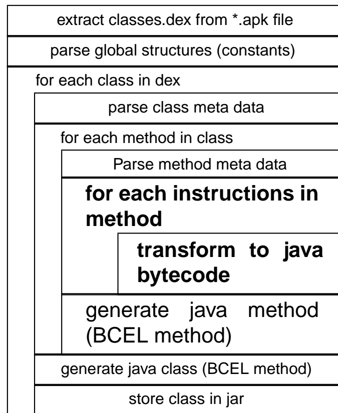
- Extract Method Meta Data
- Parse Instructions
- Generate JAVA method

Parse Method Meta Data



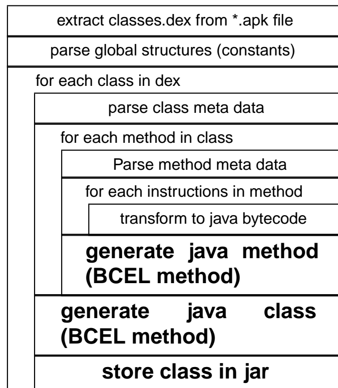
- transform method meta data to BCEL method structures
- extract method signatures,
- set up local variable tables,
- map Dalvik registers to JVM registers

Generate the instructions



- first create BCEL InstructionList
- create NOP proxies for every Dalvik instruction to prepare jump targets (satisfy forward jumps)
- For every Dalvik instruction add an equivalent JVM bytecode block to the JVM InstructionList

Store generated data in BCEL structures



- generate the BCEL structures
- store to current context
- in the end we have a class file for each defined class in the dex

Store generated data in BCEL structures

Dalvik

```
type          : '()LMD5;'
access        : 0x0009 (PUBLIC STATIC)
code          -
registers     : 1
ins           : 0
outs          : 1
insns size    : 14 16-bit code units
0003c0:                                        |[0003c0] MD5.getInstance:()LMD5;
0003d0: 6200 0200                              |0000: sget-object v0, LMD5;.md5:LMD5; // field@0002
0003d4: 3900 0900                              |0002: if-nez v0, 000b // +0009
0003d8: 2200 0200                              |0004: new-instance v0, LMD5; // class@0002
0003dc: 7010 0100 0000                         |0006: invoke-direct {v0}, LMD5;.<init>:()V // method@0001
0003e2: 6900 0200                              |0009: sput-object v0, LMD5;.md5:LMD5; // field@0002
0003e6: 6200 0200                              |000b: sget-object v0, LMD5;.md5:LMD5; // field@0002
0003ea: 1100                                   |000d: return-object v0
catches       : (none)
positions     :
0x0000 line=24
0x0004 line=26
0x000b line=29
```

JVM code

```
public static MD5 getInstance();
Code:
 0:   getstatic       #14; //Field md5:LMD5;
 3:   astore_0
 4:   aload_0
 5:   ifnonnull       20
 8:   new     #4; //class MD5
 11:  astore_0
 12:  aload_0
 13:  invokespecial   #73; //Method "<init>":()V
 16:  aload_0
 17:  putstatic       #14; //Field md5:LMD5;
 20:  getstatic       #14; //Field md5:LMD5;
 23:  astore_0
 24:  aload_0
 25:  areturn
```

Challenges

- Assign Dalvik regs to jvm regs
- obey stack balance rule (when processing opcodes)
- type inference (reconstruct flow of data assignment opcodes)

Store generated data in BCEL structures

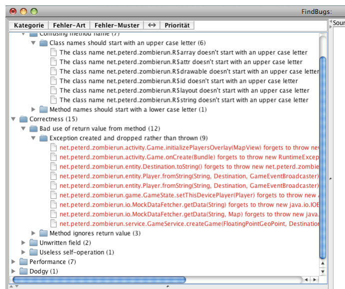
Dalvik

```
type          : '()LMD5;'
access        : 0x0009 (PUBLIC STATIC)
code          -
registers     : 1
ins           : 0
outs          : 1
insns size    : 14 16-bit code units
0003c0:                                  |[0003c0] MD5.getInstance:()LMD5;
0003d0: 6200 0200                        |0000: sget-object v0, LMD5;.md5:LMD5; // field@0002
0003d4: 3900 0900                        |0002: if-nez v0, 000b // +0009
0003d8: 2200 0200                        |0004: new-instance v0, LMD5; // class@0002
0003dc: 7010 0100 0000                   |0006: invoke-direct {v0}, LMD5;.<init>:()V // method@0001
0003e2: 6900 0200                        |0009: sput-object v0, LMD5;.md5:LMD5; // field@0002
0003e6: 6200 0200                        |000b: sget-object v0, LMD5;.md5:LMD5; // field@0002
0003ea: 1100                             |000d: return-object v0
catches       : (none)
positions     :
0x0000 line=24
0x0004 line=26
```

JVM code

```
 public static MD5 getInstance();
Code:
 0:   getstatic       #14; //Field md5:LMD5;
 3:   astore_0
 4:   aload_0
 5:   ifnonnull       20
 8:   new     #4; //class MD5
 11:  astore_0
 12:  aload_0
 13:  invokespecial   #73; //Method "<init>":()V
 16:  aload_0
 17:  putstatic       #14; //Field md5:LMD5;
 20:  getstatic       #14; //Field md5:LMD5;
 23:  astore_0
 24:  aload_0
 25:  areturn
```

Now we have bytecode, what to do with it?



Statically analyze it!

- Analyze the code with static checking tools (findbugs)
- Programming bugs, vulnerabilities, license violations

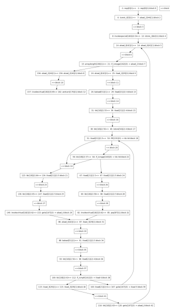
Now we have bytecode, what to do with it?

```
public class WebDialog extends Dialog
{

    public WebDialog(Context arg0)
    {
        super(arg0);
        Object obj = JVM INSTR new #14  <Class WebView>;
        ((WebView) (obj)).WebView(arg0);
        webView = ((WebView) (obj));
        obj = webView;
        obj = ((WebView) (obj)).getSettings();
        boolean flag = true;
        ((WebSettings) (obj)).setJavaScriptEnabled(flag);
        obj = webView;
        setContentView(((android.view.View) (obj)));
        obj = "Welcome";
        setTitle(((CharSequence) (obj)));
    }

    public void loadUrl(String arg0)
    {
        WebView webview = webView;
        webview.loadUrl(arg0);
```

Decompile it!

- Feed the generated jar into a decompiler
- It will spit out JAVA-like code
- Structural equal to the original source (but some differences due to heavy reuse of stack variables)

Now we have bytecode, what to do with it?



Graph it!

- Findbugs comes with a control-flow-graph analyzer
- Generate nodes and arrays (50 locs)
- Write DIA file
- enjoy that you can reuse tools from the java world

Some smaller facts

Hard Facts and Trivia

- 4000 lines of code
- written in JAVA, only external dependency is BCEL
- command line only
- licensing is **GPL** (look out for undx on fedorahosted soon)
- will be published after having tested successfully with recent cupcake binaries and optimized dex files
- **undx** name suggested by Dan Brownstein

finally{}

- Thank you for your attention
- Time for Q & A
- or send me a mail

marc.schoenefeld -at- gmx DOT org

This presentation was build with open source tools:

- Fedora 10
- Latex
- Beamer
- OpenJDK