StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Apps

Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, Fabio Massacci

yury.zhauniarovich, maqsood.ahmad, bruno.crispo, fabio.massacci@unitn.it
olga.gadyatskaya@uni.lu

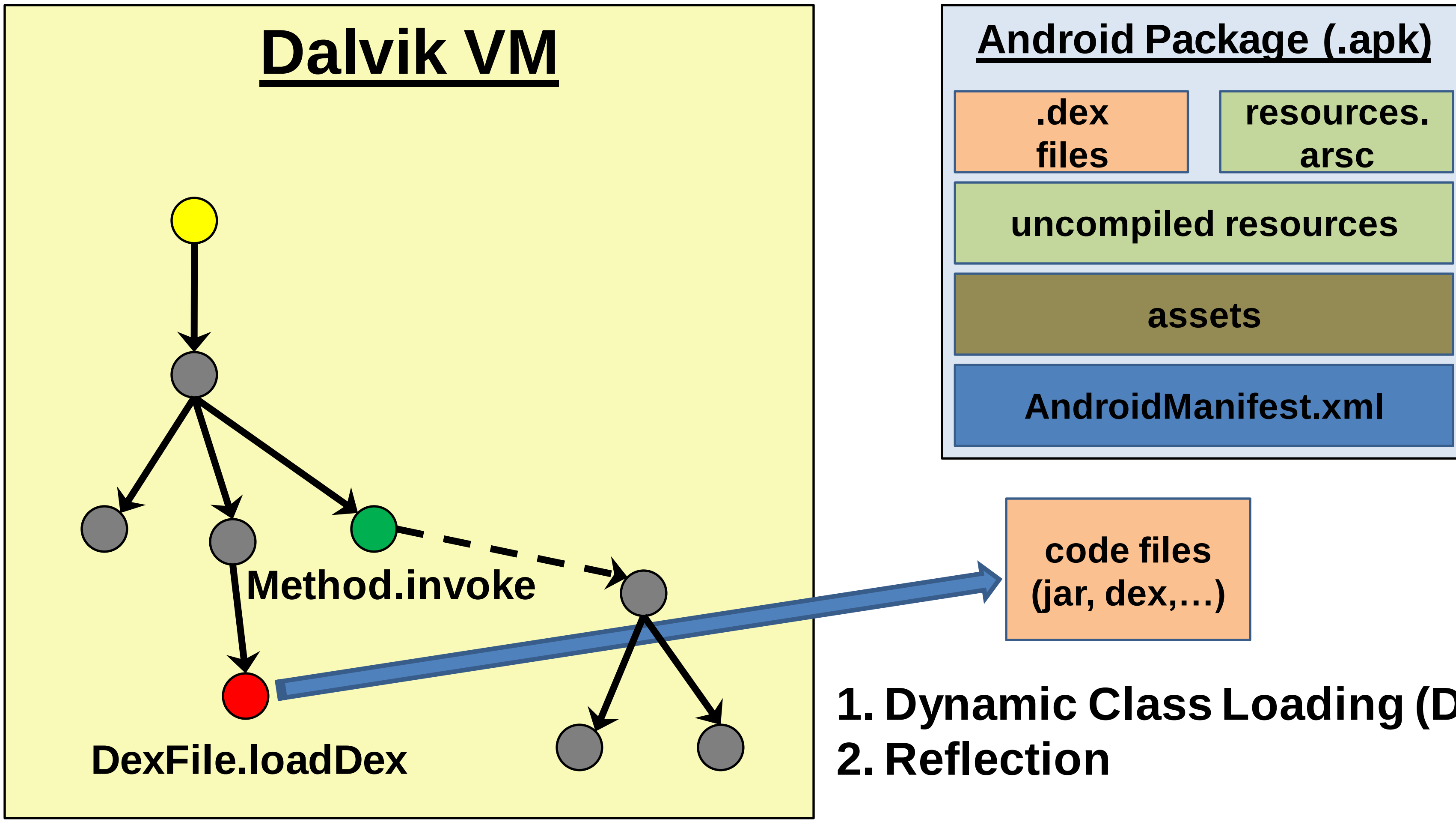
University of Trento
SnT, University Of Luxembourg

Analysis Types

- **Static analysis** – is the analysis of applications which is performed without the actual execution of an application

- **Dynamic analysis** – is the analysis which is performed by executing an application in real or emulated environments

Dynamic Code Updates*



1. Dynamic Class Loading (DCL)
2. Reflection

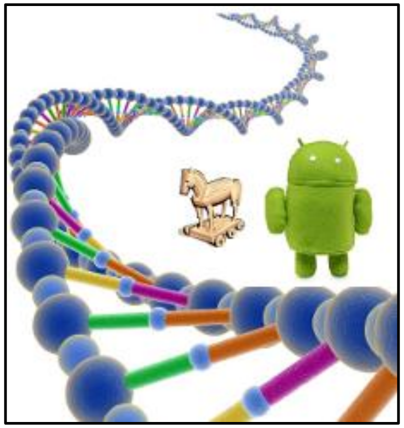
* S. Poeplau et al. “Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications”. *In Proc. Of NDSS’14*

Motivation

- In Android, code loaded dynamically has the same privileges as original
- Static analyzers cannot fully inspect an app in the presence of dynamic code update features (AndroGuard, FlowDroid, etc.)
- Heavily used by malware to conceal malicious behavior
- Used in real applications to bypass Android limitations

Reflection and DCL Usage

- Google Play:
  - analyzed 13863 apps
  - 19% contain DCL calls
  - 88% use reflection
- Third-party markets:
  - analyzed 14283 apps from 6 markets
  - 6% contain DCL calls (F-Droid: 1%)
  - 74% use reflection (F-Droid: 57%)
- Malware dataset:
  - 1260 samples analyzed
  - 20% contain DCL calls
  - 81% use reflection



Representative Example

```java
[com.sec.android.providers.drm.Doctype]
public static Object b(File paramFile, String paramString1, String paramString2, Object[]
paramArrayOfObject)
{
  String str3;
  if (paramFile == null) {
    String str1 = a.getFilesDir().getAbsolutePath();
    //get the name of the file to be loaded
    //9CkOrC32uI327WBD7n__ -> /anserverb.db
    String str2 = Xmlns.d("9CkOrC32uI327WBD7n__");
    str3 = str1.concat(str2);
  }
  for (File localFile = new File(str3); ;localFile = paramFile) {
    String str4 = localFile.getAbsolutePath();
    String str5 = a.getFilesDir().getAbsolutePath();
    ClassLoader localClassLoader = a.getClassLoader().getParent();
    //get the class specified by "paramString1" from anserverb.db
    Class localClass = new DexClassLoader(str4, str5, null, localClassLoader).loadClass(
    paramString1);
    Class[] arrayOfClass = new Class[5];
    arrayOfClass[0] = Context.class;
    arrayOfClass[1] = Intent.class;
    arrayOfClass[2] = BroadcastReceiver.class;
    arrayOfClass[3] = FileDescriptor.class;
    arrayOfClass[4] = String.class;
    //get the method specified by "paramString2"
    Method localMethod = localClass.getMethod(paramString2, arrayOfClass);
    //create new instance of the class
    Object localObject = localClass.newInstance();
    //call the corresponding method with arguments in array "paramArrayOfObject"
    return localMethod.invoke(localObject, paramArrayOfObject);
  }
}
```

Problem: Dynamic Code Updates

Issue: *How to analyze Android apps in the presence of*

- reflection calls,
  - detect the name of the called function/class
- dynamic class loading?
  - download and analyze the loaded code

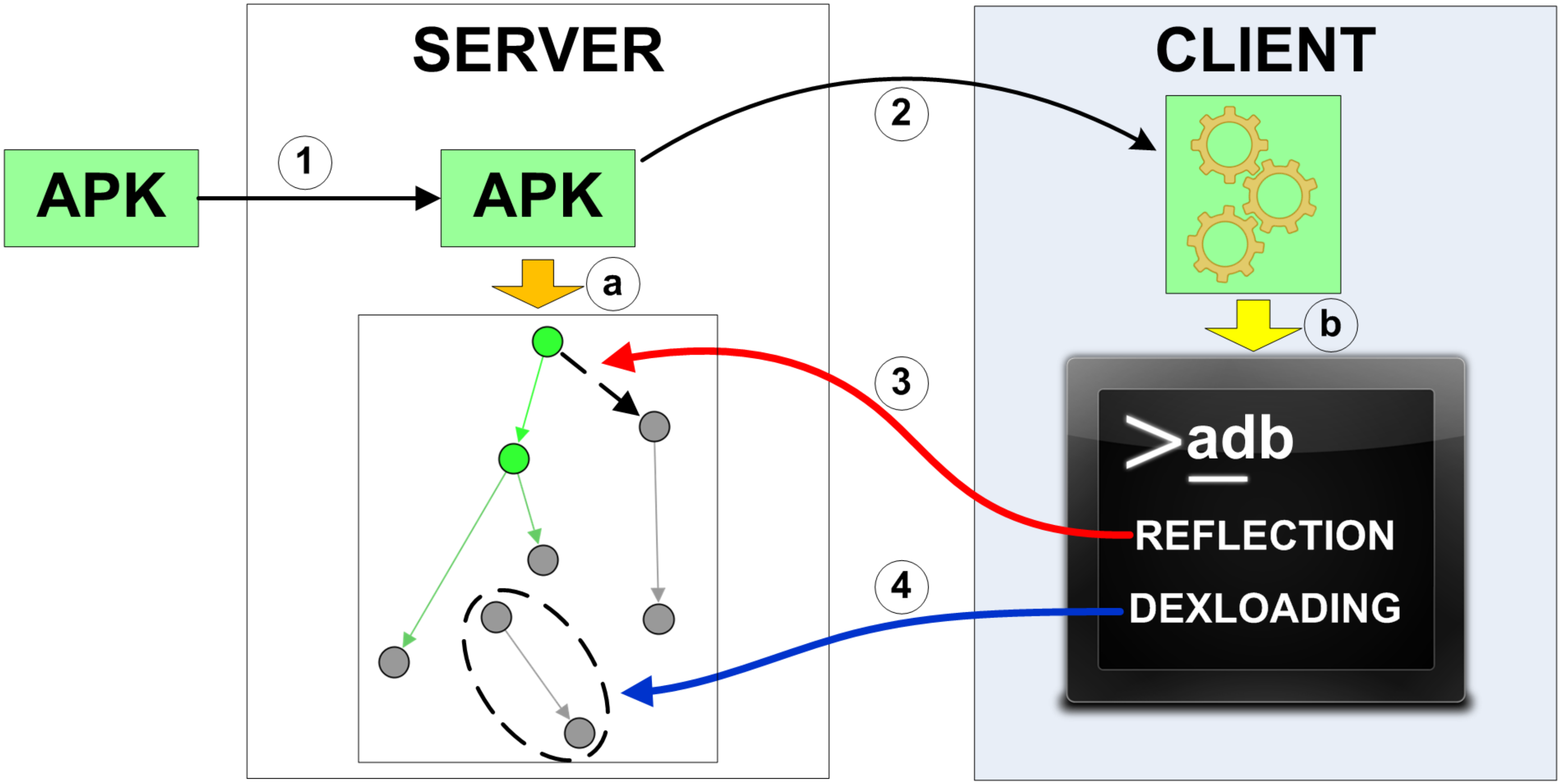
- **Method Call Graph (MCG)** is a directed graph showing the calling relationships between methods in a computer program

StaDynA: Idea

- Apps with Dynamic Code Update features expose their dynamic behavior **at runtime**

- **IDEA:** combine static and dynamic analysis techniques to detect and explore Dynamic Code Update features

StaDynA: Overview



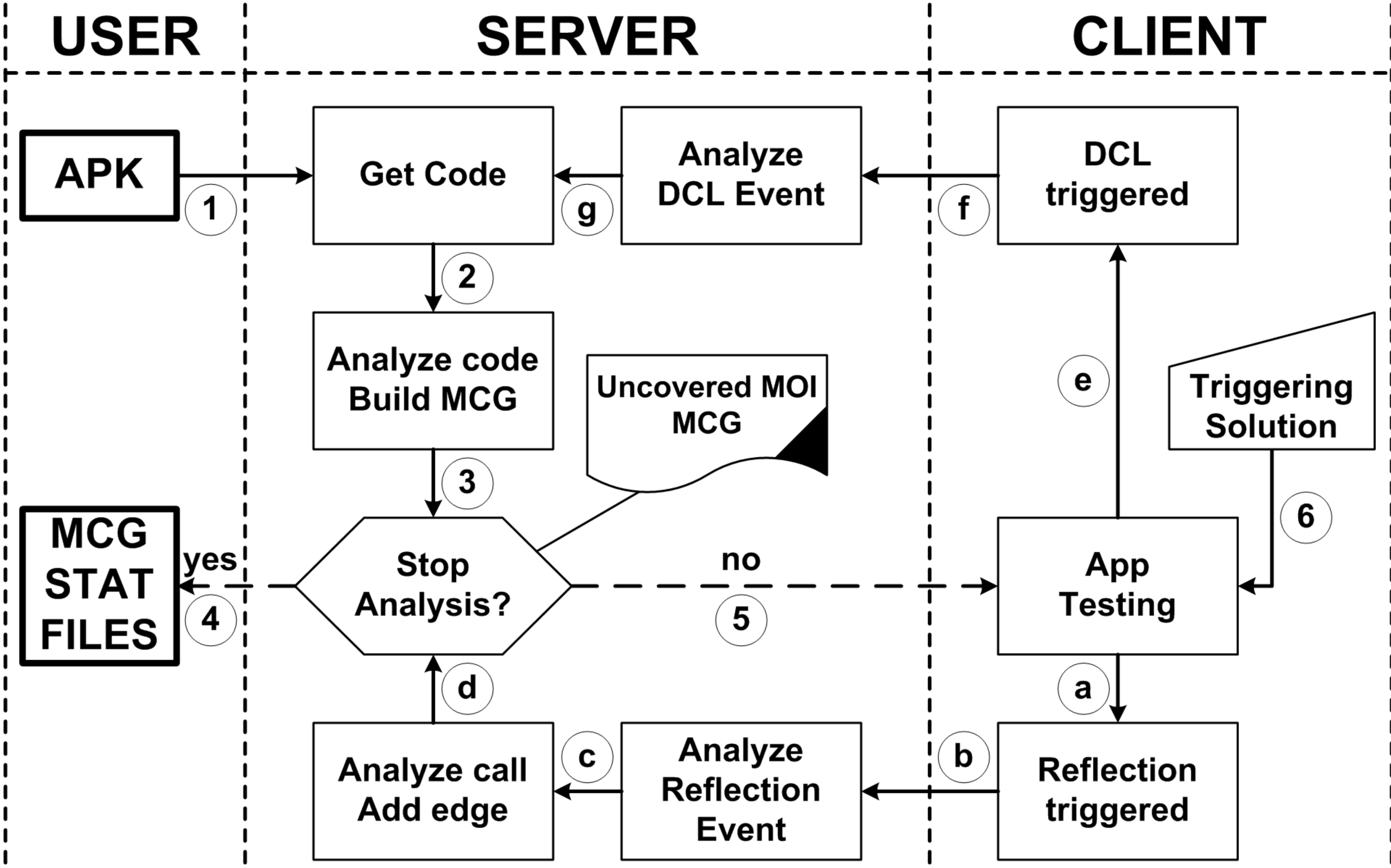
StaDynA: Approach

- Find API calls responsible for reflection and DCL at static time (we name the methods calling these API functions as **Methods of Interest (MOI)**)

Class	Method	Prot.
Dynamic class loading		
Ldalvik/system/PathClassLoader;	$< init >$	.
Ldalvik/system/DexClassLoader;	$< init >$	.
Ldalvik/system/DexFile;	$< init >$	.
Ldalvik/system/DexFile;	*loadDex*	.
Class instance creation through reflection		
Ljava/lang/Class;	*newInstance*	.
Ljava/lang/reflect/Constructor;	*newInstance*	.
Method invocation through reflection		
Ljava/lang/reflect/Method;	*invoke*	.

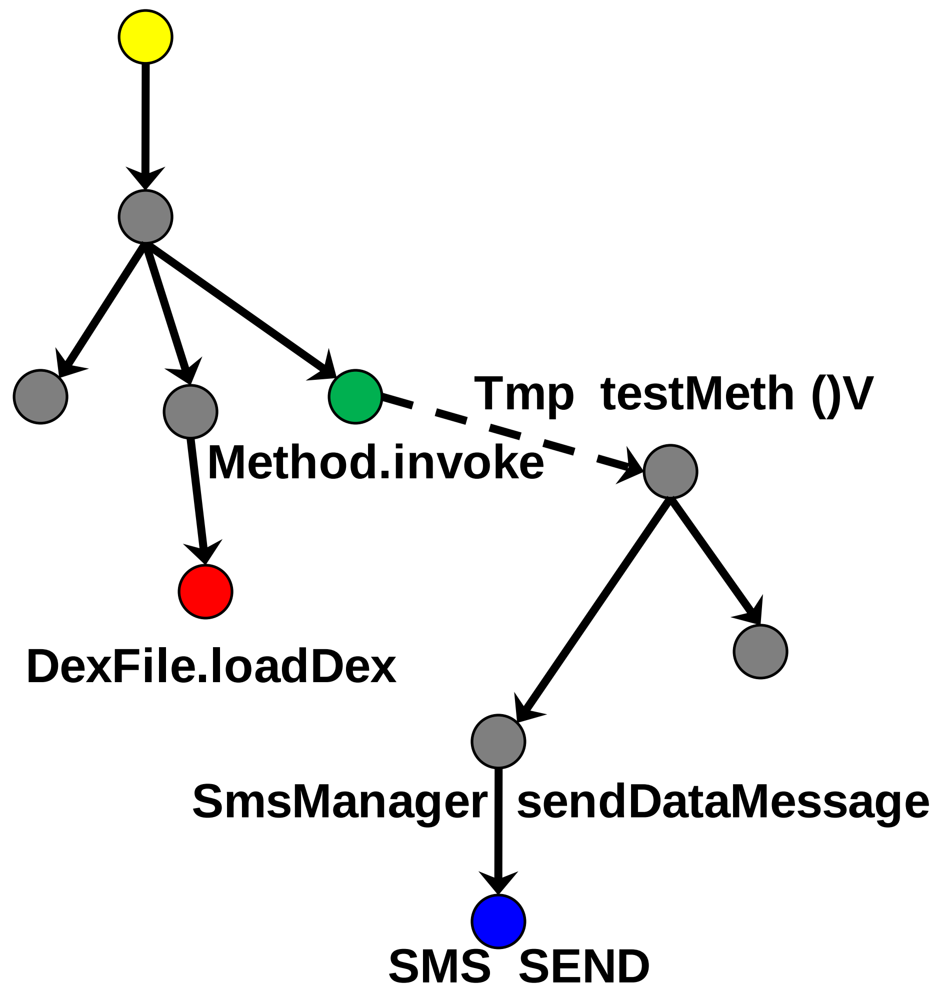
- Analyze their behavior at runtime

StaDynA: Workflow



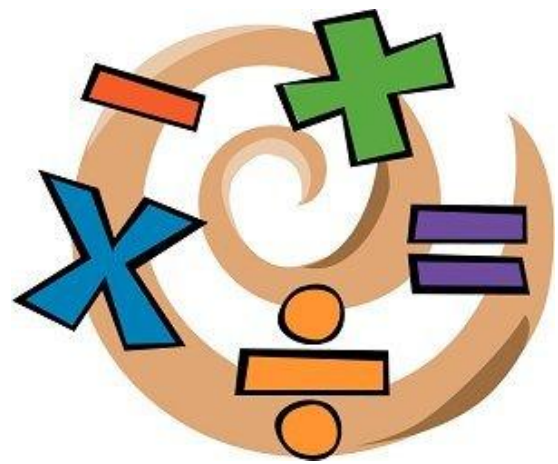
StaDynA: Features

- Stores and analyzes the code loaded dynamically
- Builds MCG of the app including the information obtained at runtime
- Discovers at runtime the qualifiers of the methods/constructors called through reflection
- Discovers suspicious behavior patterns



StaDynA: Evaluation

- Dataset:
  - 5 benign (FlappyBird, Norton AV, Avast AV, Viber, Floating Image)
  - 5 malicious (FakeNotify.B, AnserverBot, BaseBridge, DroidKungFu4, SMSSend)
- The dataset is small:
  - StaDynA requires manual triggering



- Evaluation parameters:
  - the increase of the MCG
  - coverage of the MOI detected in the application
  - discovered suspicious patterns

Evaluation: MCG Increase

	Nodes		Edges	
Apps	Initial	Final	Initial	Final
Benign Applications				
FlappyBird	8592	8614	11014	11031
Norton AV	42886	55372	65960	85665
Avast AV	31317	32363	43554	44956
Viber	42536	46312	60078	65627
ImageView	5708	5713	6488	6496
Malicious Applications				
FakeNotify.B	148	171	137	191
AnserverBot	1006	1614	1138	2093
BaseBridge	1172	1780	1364	2333
DroidKungFu4	1550	21168	1779	23589
SMSSend	431	537	826	951

Evaluation: Coverage

	Refl. Invoke		Refl. NewInstance		DCL	
Apps	Found (Init.)	Triggered	Found (Init.)	Triggered	Found	Triggered
Benign Applications						
FlappyBird	11 (10)	6	6 (6)	0	1 (1)	1
Norton AV	137 (18)	5	12 (8)	2	4 (4)	2
Avast AV	42 (42)	6	19 (19)	5	1 (1)	1
Viber	107 (101)	26	47 (21)	14	2 (2)	1
ImageView	6 (6)	5	2 (2)	2	0 (0)	0
Malicious Applications						
FakeNotify.B	68 (68)	68	9 (9)	9	0 (0)	0
AnserverBot	4 (4)	1	5 (4)	2	6 (5)	3
BaseBridge	5 (5)	1	3 (2)	2	3 (2)	3
DroidKungFu4	13 (9)	1	6 (4)	0	1 (1)	1
SMSSend	193 (193)	128	1 (1)	1	0 (0)	0

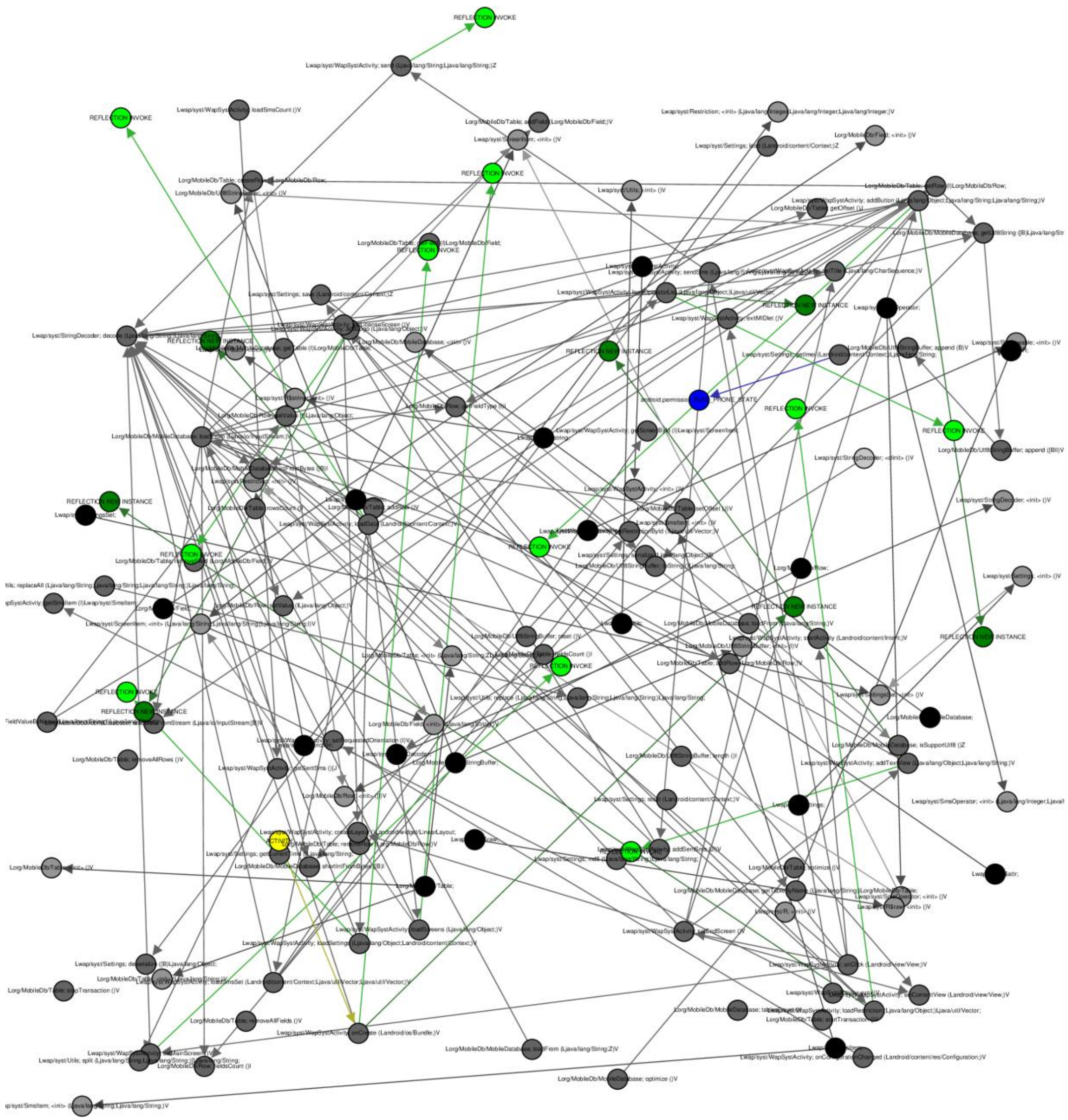
Evaluation: Suspicious Patterns

Benign Applications		
Norton AV	WRITE_SETTINGS	
	READ_PHONE_STATE	
	INTERNET	
	WRITE_SYNC_SETTINGS	v
	GET_TASKS	
Avast AV	INTERNET	
Viber	READ_PHONE_STATE	
	BLUETOOTH	
	INTERNET	

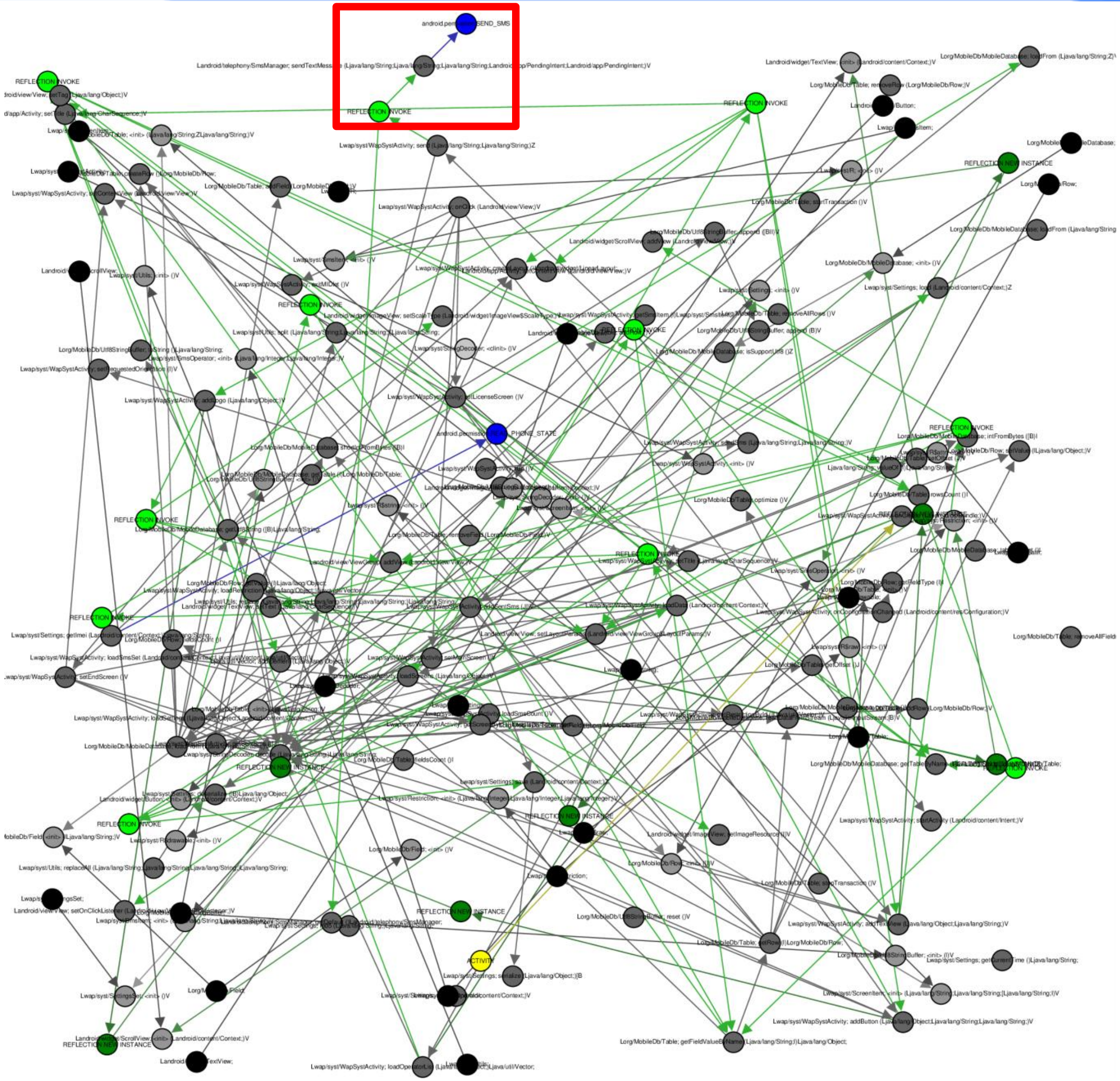
Malware		
FakeNotify.B	SEND_SMS	v
AnserverBot	INTERNET	
	READ_PHONE_STATE	
BaseBridge	INTERNET	
	READ_PHONE_STATE	
DroidKungFu4	CHANGE_NETWORK_STATE	v
	ACCESS_COARSE_LOCATION	
	BLUETOOTH	v
	INTERNET	
	BLUETOOTH_ADMIN	v
	WRITE_SETTINGS	v
	SET_TIME_ZONE	v
	WRITE_SYNC_SETTINGS	v
	READ_PHONE_STATE	
	CHANGE_WIFI_STATE	v
	MODIFY_AUDIO_SETTINGS	v
	MOUNT_UNMOUNT_FILESYSTEMS	v
SMSSend	READ_PHONE_STATE	v
	SEND_SMS	v

- Access to the functionality protected with dangerous permissions from the loaded code
- Ticks show that the usage of the corresponding permission has not been found in the initial app file (over-privileged apps)

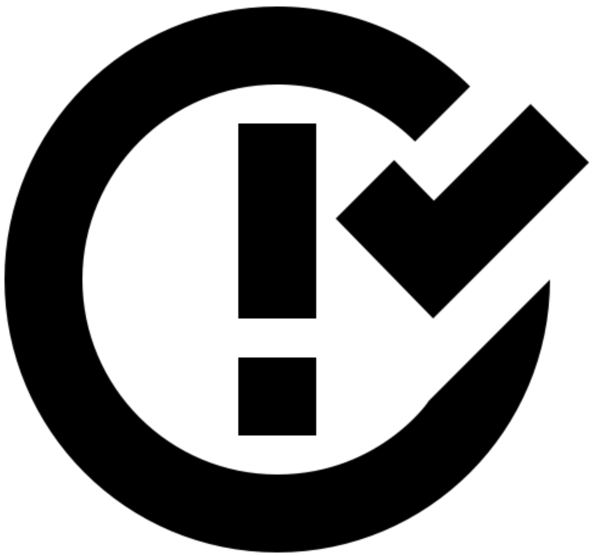
FakeNotify.B before StaDynA



FakeNotify.B after StaDynA



StaDynA: Issues

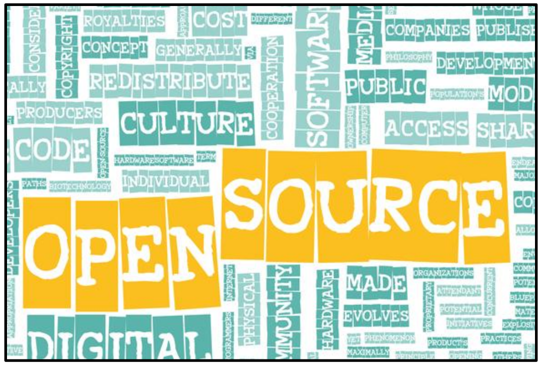


- Manual triggering
- Resolution of all reflection targets is done at runtime
- The information obtained during different runs is not merged
- No separation according to the name of the package (UID is used instead)
- Not all types of dynamic code updates have been covered

StaDynA: Summary

- Dynamic code updates is a serious problem for Android
  - the code loaded dynamically has the same privileges as the original application
- We proposed an approach that facilitates the analysis of apps in the presence of reflection and DCL
  - discovers at runtime the qualifiers of the methods/constructors called through reflection
  - stores and analyzes code loaded dynamically
  - builds MCG of the app including the information obtained at runtime
  - discovers suspicious behavior patterns
- **Open-source:**

  https://github.com/zyrikby/StaDynA



BACKGROUND SLIDES

StaDynA: Main Function

Protocol 4 App analysis main function algorithm

```
function PERFORM_ANALYSIS(inputApkPath, resultsDirPath)
    makeAnalysis(inputApkPath)
    if !containsMethodsToAnalyze() then
        performInfoSave(resultsDirPath)
        return
    end if
    dev ← getDeviceForAnalysis()
    package_name ← get_package_name(inputApkPath)
    dev.install_package(inputApkPath)
    uid ← dev.get_package_uid(package_name)
    messages ← dev.getLogcatMessages(uid)
    loop
        msg ← dequeue(messages)
        analyseStadynaMsg(msg)
        if finishAnalysis then
            performInfoSave(resultsDirPath)
            return
        end if
    end loop
end function
```

Analysis of Invoke Event

Protocol 6 The algorithm for analysis of the reflection invoke message

```
function PROCESSREFLINVOKEMSG(message)
    cls ← message.get(JSON_CLASS)
    method ← message.get(JSON_METHOD)
    prototype ← message.get(JSON_PROTO)
    stack ← message.get(JSON_STACK)
    invDstFrCl ← (class, method, prototype)
    invPosInStack ← findFirstInvokePos(stack)
    thrMtd ← stack[invPosInStack]
    invSrcFrStack ← stack[invPosInStack + 1]
    for all invPathFrSrcs ∈ sources_invoke do
        invSrcFrSrcs ← invPathFrSrcs[0]
        if invSrcFrSrcs ≠ invSrcFrStack then
            continue
        end if
        addInvPathToMCG(invSrcFrSrcs, thrMtd, invDstFrCl)
        if invPathFrSrcs ∈ uncovered_invoke then
            uncovered_invoke.remove(invPathFrSrcs)
        end if
        return
    end for
    addSuspiciousInvoke(thrMtd, invDstFrCl, stack)
end function
```

Analysis of DCL Event

Protocol 7 The algorithm for analysis of the DCL message

```
function processDexLoadMsg(message)
    source ← message.get(JSON_DEX_SOURCE)
    stack ← message.get(JSON_STACK)
    newFile ← dev.get_file(source)
    newFilePath ← processNewFile(newFile)
    dlPathFrStack = getDLPathFrStack(stack)
    if dlPathFrStack then
        srcFromStack ← dlPathFrStack[0]
        thrMtd ← dlPathFrStack[1]
        if dlPathFrStack ∈ uncovered_dexload then
            uncovered_dexload.remove(dlPathFrStack)
        end if
        addDLPathToMCG(srcFromStack, thrMtd, newFilePath)
        if !fileAnalysed(newFilePath) then
            makeAnalysis(newFilePath)
        end if
        return
    end ifaddSuspiciousDL(newFilePath, stack)
end function
```