ESSE Mobile Security – VO 5: Code Obfuscation und Code Encryption

Raphael Kiefmann, Paul Kalauner, Andreas Ehringfeld

INSO – Industrial Software
Institut für Information Systems Engineering | Fakultät für Informatik | Technische Universität Wien

Agenda

Charakteristiken & Motivation

Code Obfuscation

- Identifier Renaming
- Control Flow Obfuscation (CFO)
- Data Obfuscation
- Reflection-based Obfuscation

Code Encryption

Spezialfälle

Demo – PreEmptive DashO

Charakteristiken von Code Obfuscation/Encryption

- **Code Obfuscation:** Veränderung von Code ohne Beeinflussung des vorgesehenen Verhaltens
- **Code Encryption:** Verschlüsselung von Code, welcher während Laufzeit entschlüsselt und nachgeladen wird
- Lesbarkeit des Codes (nach Dekompilierung) wird dadurch deutlich verringert
- **Kein** 100%-iger Schutz gegen Dekompilierung, erhöht lediglich die dafür notwendigen Ressourcen

Motivation

- Erschwerung von Reverse Engineering
- Schutz intellektuellem Eigentums
- Schutz vor Piraterie und Repackaging
- Insbesondere unter Android relevant, da Bytecode verhältnismäßig einfach dekompilierbar ist
- Allerdings: häufig auch von Malware-Entwickler:innen eingesetzt, um Erkennung und Analyse von schadhaften Programmen zu erschweren

Code Obfuscation

Übersicht an Techniken

- Identifier Renaming
- Control Flow Obfuscation
  - Dead Code Injection
  - Code Reordering
  - Erweiterungen durch Schleifen
  - Method Inlining
  - Control Flow Flattening
- Data Obfuscation
- Reflection-based Obfuscation

Identifier Renaming

- Entwickler verwenden im Normalfall sinnvolle Bezeichnungen für Code-Elemente

  $\Rightarrow$ auch für Reverse Engineering hilfreich

- Zuweisung bedeutungsloser Namen an Variablen, Klassen, Methoden
- Meist lexikalische Sequenzen oder Permutationen
- Standardmäßig von APK Optimizern wie ProGuard/R8 durchgeführt
  - Verkleinerung
  - Optimisierung
  - Grobe Obfuskierung

Identifier Renaming – Beispiel

```java
public class ExampleClass {
    private String someString = "abc";
    private int someInt = 42;

    public void doSomething() {
        // ...
    }
}
```

```java
public class A {
    private String aaa = "abc";
    private int aab = 42;

    public void a() {
        // ...
    }
}
```

Identifier Renaming in Binärdateien

- Auch in Binärdateien sind oftmals Zusatzinformationen enthalten
  - Funktionsnamen
  - Variablennamen
  - Debug Informationen
- Das Programm **strip** entfernt für den Ablauf irrelevante Code Stellen
- Primär zur Verkleinerung der Dateien aber mit dem Nebeneffekt der Code Obfuscation

Control Flow Obfuscation (CFO)

- Veränderung des Control Flows
- Veränderte Ausführungspfade bei unverändertem Verhalten
- Vielzahl an Möglichkeiten

CFO: Dead Code Injection

- Einfügen von Code, welcher niemals ausgeführt wird oder keine Auswirkungen hat

```
1 for (int i = 0; i < 10; i++) {
2     System.out.println(i);
3 }
```

```
1 for (int i = 0; i < 10; i++) {
2     if (i % 1337 > 10) {
3         System.exit(0);
4     }
5     System.out.println(i);
6 }
```

CFO: Code Reordering

- Veränderung der Ausführungsreihenfolge von Anweisungen

```
1 x = 0;
2 while (x < maxNum) {
3     i[x] += j[x];
4     x++;
5 }
```

```
1 x = maxNum;
2 while (x > 0) {
3     x--;
4     i[x] += j[x];
5 }
```

(Vergleiche Faruki et al. (2016))

CFO: Erweiterungen durch Schleifen

- Ersetzung von if-Anweisungen durch komplexere Schleifen
- Einführung zusätzlicher Konditionen mit keinerlei Auswirkungen

```
1 for (int i = 0; i < 15; i++) {
2     if (x < 10) {
3         x += 10;
4     }
5     System.out.println(x);
6 }
```

```
1 for (int i = 0; i < 15 i++) {
2     while (x < 10 || x % 20 == 0) {
3         x += 10;
4     }
5     System.out.println(x);
6 }
```

CFO: Method Inlining

- Ersetzung von Methodenaufrufen durch Methodenkörper
- Entfernt prozedurale Abstraktionen von Programmen
- Auch von Optimizern angewandt

```java
int y = doSomething(42);
private int doSomething(int x) {
    // ...
    x += 1;
    return x * 1337;
}
```

```java
int x = 42;
// ...
x += 1;
int y = x * 1337;
```

CFO: Control Flow Flattening

- Verschiebung von Funktionskörpern, Schleifen und if-Anweisungen in einzige Schleife
- Programmablauf durch `switch` oder mehrere if-Anweisungen geregelt
- Blöcke des Control Flow Graphs werden damit auf selbe Ebene gebracht

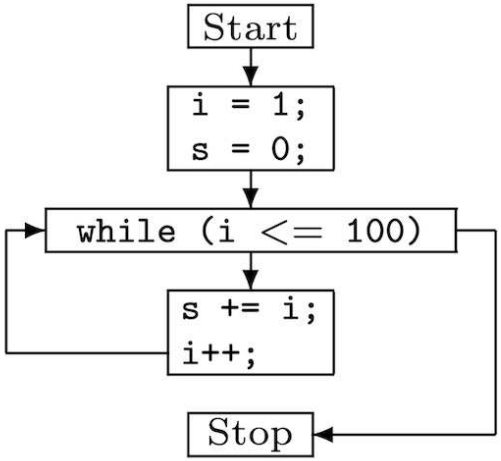
CFO: Control Flow Flattening – Beispiel

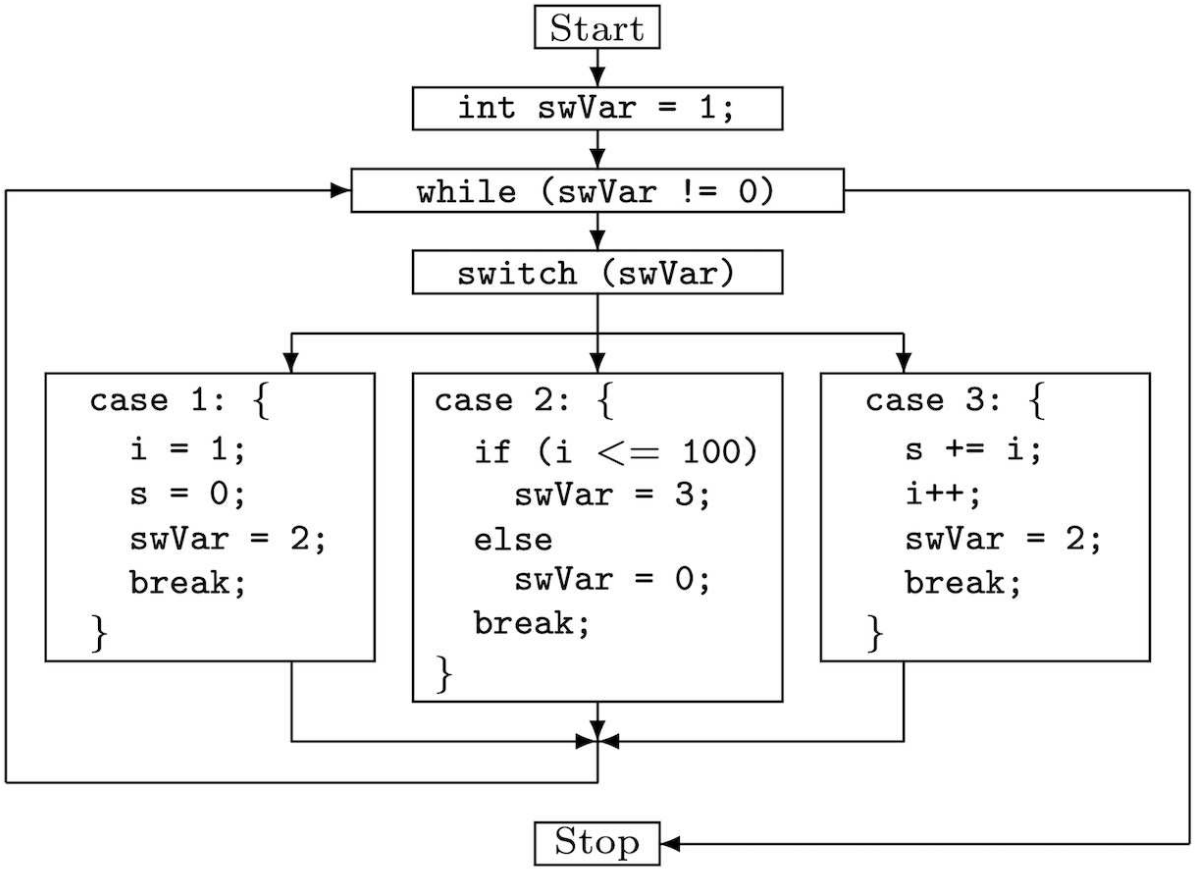
```
1  int i = 1;
2  int s = 0;
3  while (i <= 100) {
4      s += i;
5      i++;
6  }
```

```
1  int swVar = 1;
2  while (swVar != 0) {
3      switch (swVar) {
4          case 1:
5            i = 1; s = 0; swVar = 2;
6            break;
7          case 2:
8            if (i <= 100) swVar = 3;
9            else swVar = 0;
10           break;
11         case 3:
12           s += i; i++; swVar = 2;
13           break;
14     }
15 }
```

(Vergleiche László und Kiss (2009))

CFO: Control Flow Flattening – Beispiel Control Flow Graph





(Vergleiche László und Kiss (2009))

Data Obfuscation

- Verstecken von Daten/Informationen in Applikationen
- Resource Encryption
  - Verschlüsselung von Ressourcen, die in App mitgepackt sind
  - Injektion von Entschlüsselungslogik in Methoden zum Öffnen von Ressourcen
- String Encryption
  - Ersetzung von Strings durch verschlüsselte Gegenstücke
  - Injektion von Methode(n), die den zugehörigen ursprünglichen String für den gegebenen verschlüsselten String zurückliefern
  - Entschlüsselung bei Zugriff auf String während Laufzeit
  - Invertierbare Ver- und Entschlüsselungsalgorithmen notwendig

String Encryption – Beispiel

```java
public class ExampleClass {
    private String someSecretString = "mobsec{0123456789abcdef}";
    private String password = "t0p_s3cr3t";

    // ...
}
```

```java
public class ExampleClass {
    private String someSecretString = InjectedClass.decrypt("vz325b6ft76jas/E7ft76j$!");
    private String password = InjectedClass.decrypt("T(/$SWksgfzg$%u");

    // ...
}
```

Reflection-based Obfuscation

- Java API
- Erlaubt es, Methoden dynamisch aufzurufen
- Verschleierung von Methodenaufrufen durch Entfernung direkter Referenzen
- Referenzen werden dynamisch während Laufzeit über API abgerufen
- Insbesondere in Kombination mit anderen Obfuskierungs-Techniken (z.B. String Encryption) hilfreich

```
1 // ...
2 SomeClass clazz = new SomeClass();
3 clazz.doSomething();
```

```
1 // ...
2 Object c = Class.forName("com.example.SomeClass").newInstance();
3 Method m = c.getClass().getMethod("doSomething");
4 m.invoke(c);
5
```

Code Encryption

Allgemeines zur Code Encryption

- Code ist verschlüsselt und wird erst zur Laufzeit entschlüsselt und "lauffähig" gemacht
- Funktioniert nicht mit allen Programmiersprachen, da oft die Mechanismen dafür fehlen
  - Viele Sprachen können neuen Code starten, aber nicht anschließend damit interagieren
- Erfordert einen `ClassLoader`-Mechanismus (JVM-basierte Sprachen) bzw. die Möglichkeit den Speicher zu manipulieren (C, C++)

JVM-basierte Code Encryption

- JVM-basierte Sprachen können in der Regel auf den `ClassLoader` zugreifen
  - Ermöglicht es, Code während Laufzeit nachzuladen
- Ein eigens-implementierter `ClassLoader` ermöglicht, dass Objekte entschlüsselt werden bevor sie nachgeladen werden

JVM-basierte Code Encryption – Beispiel

- Als Grundlage hierfür dient ein Projekt von Sefik Serengil

```
1   ...
2   // decrypt the class
3   Cipher decryption = Cipher.getInstance(algorithm);
4   decryption.init(Cipher.DECRYPT_MODE, new SecretKeySpec(key, 0, key.length, algorithm
     ));
5   byte[] decryptedContent = decryption.doFinal(encryptedContent);
6
7   // convert the byte array into a class
8   Class clazz = defineClass(name, decryptedContent, 0, decryptedContent.length);
9
10  // search for the wanted method and call it
11  Method m = clazz.getMethod("main", String[].class);
12  m.invoke(null, new Object[] {null});
13  ...
```

(Vergleiche `https://github.com/serengil/encrypted-class-loader`)

Maschinencode-basierte Code Encryption

- Maschinencode-basierte Sprachen können oftmals den Speicher direkt manipulieren
- Auch Funktionen sind nur Daten im Speicher
  - Oftmals enstehen aber Einschränkungen durch das Betriebssystem
  - Schutzmechanismen wie **W^X** verhindern, dass Code direkt geladen und ausgeführt wird
- Erfordert meist ein eigenes Setup

Maschinencode-basierte Code Encryption – Beispiel 1/2

- Als Grundlage hierfür dient ein Projekt von Adam Rosenfield

```
1  #include <string.h>
2  #include <sys/mman.h>
3
4  char shellcode[] = "\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x05
       \xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x68\x65\x6c\x6c\x6f";
5
6  // error checking omitted for expository purposes
7  int main(int argc, char **argv)
8  {
9    // allocate some read-write memory
10   void *mem = mmap(0, sizeof(shellcode), PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS
         , -1, 0);
11
12   // copy the shellcode into the new memory
13   memcpy(mem, shellcode, sizeof(shellcode));
14
15   // make the memory read-execute
16   mprotect(mem, sizeof(shellcode), PROT_READ|PROT_EXEC);
```

Maschinencode-basierte Code Encryption – Beispiel 2/2

```
18    // call the shellcode
19    int (*func)();
20    func = (int (*)())mem;
21    (int)(*func)();
22
23    // now, if we managed to return here, it would be prudent to clean up the memory:
24    munmap(mem, sizeof(shellcode));
25
26    return 0;
27  }
```

(Vergleiche https://stackoverflow.com/a/19749664)

Spezialfälle

JNI Obfuscation

- Das Java Native Interface (JNI) ermöglicht Code in geteilten Bibliotheken aufzurufen
- Durch die Keywords ist relativ schnell ersichtlich, welche Methoden aufgerufen werden
- Neben dem automatischen Mapping kann die Verbindung zwischen JVM-Code und nativem Code auch manuell umgesetzt werden
  - Die JVM-API sieht hierfür die native Methode `RegisterNatives` vor
- Wird auch in Android eingesetzt

Manuelles JNI Handling – Beispiel 1/2

```
1  ...
2  static jint
3  add(JNIEnv* /*env*/, jobject /*thiz*/, jint a, jint b) {
4  int result = a + b;
5    ALOGI("%d + %d = %d", a, b, result);
6    return result;
7  }
8  static const char *classPathName = "com/example/android/simplejni/Native";
9  static JNINativeMethod methods[] = {
10   {"add", "(II)I", (void*)add },
11 };
12 /*
13  * Register several native methods for one class.
14  */
15 static int registerNativeMethods(JNIEnv* env, const char* className,
16   JNINativeMethod* gMethods, int numMethods)
17 {
18   jclass clazz;
19   clazz = env->FindClass(className);
20   if (clazz == NULL) {
21     ALOGE("Native registration unable to find class '%s'", className);
22     return JNI_FALSE;
23   }
```

Manuelles JNI Handling – Beispiel 2/2

```
24    if (env->RegisterNatives(clazz, gMethods, numMethods) < 0) {
25      ALOGE("RegisterNatives failed for '%s'", className);
26      return JNI_FALSE;
27    }
28    return JNI_TRUE;
29  }
30  /*
31   * Register native methods for all classes we know about.
32   *
33   * returns JNI_TRUE on success.
34   */
35  static int registerNatives(JNIEnv* env)
36  {
37    if (!registerNativeMethods(env, classPathName,
38                 methods, sizeof(methods) / sizeof(methods[0]))) {
39    return JNI_FALSE;
40    }
41    return JNI_TRUE;
42  }
43  ...
```

(Vergleiche https://android.googlesource.com/platform/development/+/master/samples/SimpleJNI/jni/native.cpp)

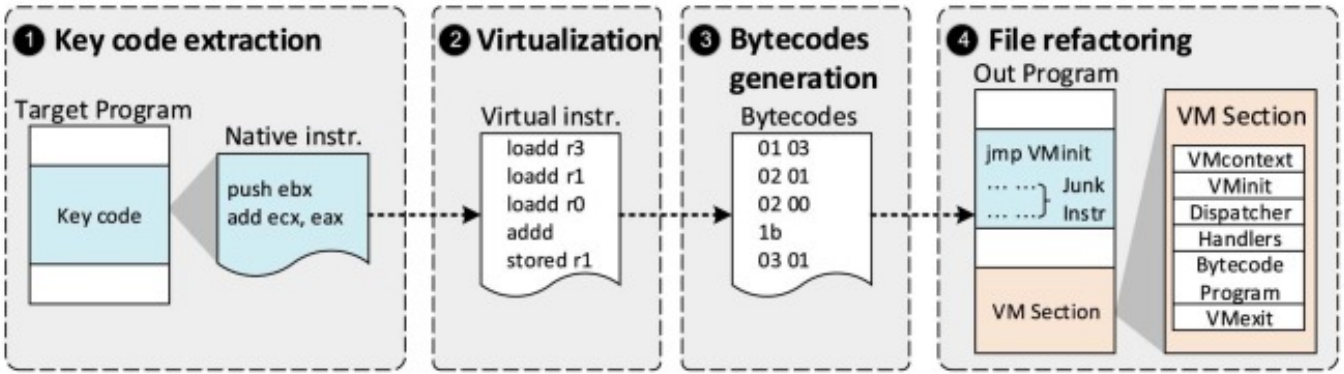
Obfuscation von Shared Libraries

- Spezielle Form des Code-nachladens
- Ein im Code verstecker BLOB kann zur Laufzeit entpackt und nachgeladen werden
- Greift auf Standardverhalten von, z.B., UNIX-basierten System zurück

```
1  typedef int (*secret_function)(char *param);
2
3  // save the blob somewhere in the filesystem
4  ...
5  void shared_lib = dlopen("/path/to/my.so", RTLD_NOW);
6
7  // ensure that dlopen was a success
8  ...
9
10 // find the secret function and make it accessible
11 secret_function *func = dlsym(shared_lib, "super_secret_function");
12
13 func("[*] Initiate new stuff ...");
14 ...
```

VM-basierte Obfuscation

- Der "heilige Gral" der Code Obfuscation
- Zwingt den Reverse Engineer dazu, dass zusätzlich zur, z.B. ARM-Architektur, noch eine zusätzliche Prozessor-Architektur dekompiliert werden muss
- Der eigentliche Binärcode wird in eine andere Sprache umgewandelt, die von einer VM interpretiert und ausgeführt wird
- Sehr aufwändig zu realisieren



(Vergleiche Kuang et al. (2018))

Beispiele für Tools

- Frei verfügbar/Open-Source
  - ProGuard
  - R8
- Kommerziell
  - DexGuard
  - DexProtector
  - Promon
  - PreEmptive DashO

Demo – PreEmptive DashO

Zusammenfassung

- Obfuscation: Veränderung von Code ohne Beeinflussung des Verhaltens
- Encryption: Verschlüsseln des Codes, Entschlüsselung während Laufzeit
- Erschwert Reverse Engineering
- Vielzahl an Varianten, beispielsweise
  - Identifier Renaming
  - Control Flow Obfuscation
  - Data Obfuscation

Literaturverzeichnis 1/3

- Parvez Faruki, Hossein Fereidooni, Vijay Laxmi, Mauro Conti, und Manoj Gaur. Android Code Protection via Obfuscation Techniques: Past, Present and Future Directions. 2016. doi: 10.48550/arXiv.1611.10231

- Pierre Graux, Jean-Francois Lalande, und Valérie Viet Triem Tong. Obfuscated Android Application Development. In *Proceedings of the Third Central European Cybersecurity Conference*, CECC 2019. Association for Computing Machinery, 2019. doi: 10.1145/3360664.3361144

Literaturverzeichnis 2/3

- Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, und Kehuan Zhang. Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. In *International Conference on Security and Privacy in Communication Systems*, Seiten 172–192. Springer, 2018. doi: 10.1007/978-3-030-01701-9_10
- Tımea László und Ákos Kiss. Obfuscating C++ Programs via Control Flow Flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30(1):3–19, 2009

Literaturverzeichnis 3/3

- Kaiyuan Kuang, Zhanyong Tang, Xiaoqing Gong, Dingyi Fang, Xiaojiang Chen, und Zheng Wang. Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling. *Computers & Security*, 74:202–220, 2018
- PreEmptive DashO – `https://www.preemptive.com/products/dasho/` – abgerufen 29. 11. 2022

Vielen Dank!

https://establishing-security.at/