Dynamic analysis of JVM processes

Marc Schönefeld, Dr. rer. nat.
RE//verse 03/01/2025
Orlando, FL

About Marc

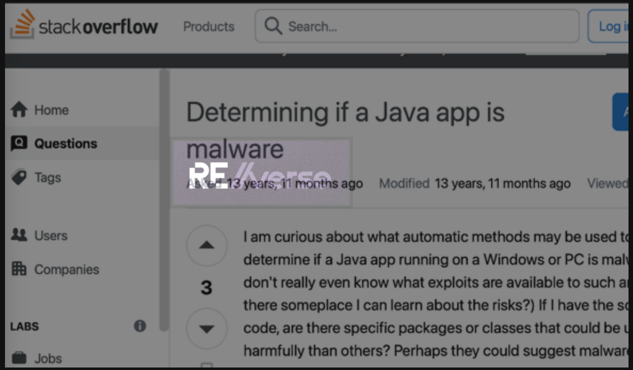
- This is personal research. Not related to my employer.
- Daytime busy with a lot of non-RE-related things (bughunting, google if curious)
  - Started with Java RE in the 200x years (while in finance industry)
    - **reversing Java-based banking trojans**
    - **enhance software (like hacking more sockets into Limewire)**
  - in 2000/2001 wrote bytecode inspection tool, to discover integer-overflows in JDK (Blackhat 2002)
  - Wrote a translator from Dalvid to JVM bytecode in 2009, to reuse JVM reversing toolchains for Android
  - For academic research wrote diffing tool for binary-only JDK patches
  - Presented about Java and Android RE at Blackhat, CanSecWest, CCC, HITB, J1 , XCon (also trainings)
- Random other stuff:
  - Retro coding since the early for C64, Amiga, ST, PC (mostly game hacks)

Agenda

- Java, JVM, bytecode, runtime artifacts and the motivation for RE
- Shortcoming of static tools, demonstrated on decompilers
- Common development tools their potential usage for reversing
- Optimization and synergies of development tools
- Reversing use cases
- Closing Thoughts
- Moving Forward

A brief overview of Java obfuscation methods.

- Early Java and the need for Obfuscation (1995-2000)
- Rise of Automated Obfuscators (2000-2010)
- Countermeasures & Stronger Obfuscation (2010-2015)
- Modern Java Obfuscation (2015-)



Determining if a Java app is malware

Asked 13 years, 11 months ago Modified 13 years, 11 months ago Viewed

3

I am curious about what automatic methods may be used to
determine if a Java app running on a Windows or PC is malw
don't really even know what exploits are available to such a
there someplace I can learn about the risks?) If I have the so
code, are there specific packages or classes that could be u
harmfully than others? Perhaps they could suggest malwar

History: Java and Obfuscation

- Introduced in 1995, cross-platform via a virtual machine layer (JVM)
- The JVM executes class files, one file per class, containing code and metadata
  - Obfuscation required to protect secrets in the plain sight of Java classes
  - Control flow (`dup , swap, dup , pop2`)
  - Identifiers (Names of methods, variables, RC4, homoglyphs, restricted keywords)
- Elimination of debug information
- Class loader tricks, encoding, encrypted with an obfuscated/derived key
- Anti-debugging code
  - Poisoned toString methods, timing checks, rogue annotation processing
- **Essential** : Knowledge of
  - **the platform characteristics** helps to set hooks at the right places
  - **bytecode**, the JVM runtime lingua franca to capture what is happening

Analyzing dynamic behavior – Benefit from known invariants

- The Java Verifier limits the extent of obfuscation
- Even extremely obfuscated code need to comply with the JVM verifier
  - **Sound bytecode**
    - Don't jump outside a method, or to the middle of instructions
    - Methods not longer than 65536 instructions , etc.
  - **Sound metadata**
    - Stack balance checks for bytecode need to succeed, and
    - comply with info found in StackMapTable instances
- JVM enforcements of package and module visibility boundaries
- Details specified in the JVM specification

Intermission: Short reminder about Mixed-Boolean-Arithmetics

- Compilers (both AOT and JIT) can differ in their ability to reconstruct a simple expression from an arbitrary complicated one
- Mixed Binary Arithmetic (MBA) is an often-used technique for obfuscation of algorithms within binaries
- The processing provides the same result as the original but it is not directly obvious what is actually computed
- Can be applied recursively
- Examples:

```
X ^ Y == (X | Y) - (X & Y)
X + Y == (X & Y) + (X | Y)
X - Y == (X ^ -Y) + 2*(X & -Y)
X & Y == (X + Y) - (X | Y)
X | Y == X + Y + 1 + (~X | ~Y)
```

Intermission: Short reminder about Mixed-Boolean-Arithmetics

- Compilers (both AOT and JIT) can differ in their ability to reconstruct a simple expression from an arbitrary complicated one
- Mixed Binary Arithmetic (MBA) is an often-used technique for obfuscation of algorithms within binaries
- The processing provides the same result as the original but it is not directly obvious what is actually computed
- Can be applied recursively
- Examples:

```
X ^ Y == (X | Y) - (X & Y)
X + Y == (X & Y) + (X | Y)
X - Y == (X ^ -Y) + 2*(X & -Y)
X & Y == (X + Y) - (X | Y)
X | Y == X + Y + 1 + (~X | ~Y)
```

Intermission: Short reminder about Mixed-Boolean-Arithmetics

- Compilers (both AOT and JIT) can differ in their ability to reconstruct a simple expression from an arbitrary complicated one
- Mixed Binary Arithmetic (MBA) is an often-used technique for obfuscation of algorithms within binaries
- The processing provides the same result as the original but it is not directly obvious what is actually computed
- Can be applied recursively
- Examples:

```
X ^ Y == (X | Y) - (X & Y)
X + Y == (X & Y) + (X | Y)
X - Y == (X ^ -Y) + 2*(X & -Y)
X & Y == (X + Y) - (X | Y)
X | Y == X + Y + 1 + (~X | ~Y)
```

Intermission: Short reminder about Mixed-Boolean-Arithmetics

- Compilers (both AOT and JIT) can differ in their ability to reconstruct a simple expression from an arbitrary complicated one
- Mixed Binary Arithmetic (MBA) is an often-used technique for obfuscation of algorithms within binaries
- The processing provides the same result as the original but it is not directly obvious what is actually computed
- Can be applied recursively
- Examples:

```
X ^ Y == (X | Y) - (X & Y)
X + Y == (X & Y) + (X | Y)
X - Y == (X ^ -Y) + 2*(X & -Y)
X & Y == (X + Y) - (X | Y)
X | Y == X + Y + 1 + (~X | ~Y)
```

Intermission: Short reminder about Mixed-Boolean-Arithmetics

- Compilers (both AOT and JIT) can differ in their ability to reconstruct a simple expression from an arbitrary complicated one
- Mixed Binary Arithmetic (MBA) is an often-used technique for obfuscation of algorithms within binaries
- The processing provides the same result as the original but it is not directly obvious what is actually computed
- Can be applied recursively
- Examples:

```
X ^ Y == (X | Y) - (X & Y)
X + Y == (X & Y) + (X | Y)
X - Y == (X ^ -Y) + 2*(X & -Y)
X & Y == (X + Y) - (X | Y)
X | Y == X + Y + 1 + (~X | ~Y)
```

Intermission: Short reminder about Mixed-Boolean-Arithmetics

- Compilers (both AOT and JIT) can differ in their ability to reconstruct a simple expression from an arbitrary complicated one
- Mixed Binary Arithmetic (MBA) is an often-used technique for obfuscation of algorithms within binaries
- The processing provides the same result as the original but it is not directly obvious what is actually computed
- Can be applied recursively
- Examples:

```
X ^ Y == (X | Y) - (X & Y)
X + Y == (X & Y) + (X | Y)
X - Y == (X ^ -Y) + 2*(X & -Y)
X & Y == (X + Y) - (X | Y)
X | Y == X + Y + 1 + (~X | ~Y)
```

Intermission: Short reminder about Mixed-Boolean-Arithmetics

- Compilers (both AOT and JIT) can differ in their ability to reconstruct a simple expression from an arbitrary complicated one
- Mixed Binary Arithmetic (MBA) is an often-used technique for obfuscation of algorithms within binaries
- The processing provides the same result as the original but it is not directly obvious what is actually computed
- Can be applied recursively
- Examples:

```
X ^ Y == (X | Y) - (X & Y)
X + Y == (X & Y) + (X | Y)
X - Y == (X ^ -Y) + 2*(X & -Y)
X & Y == (X + Y) - (X | Y)
X | Y == X + Y + 1 + (~X | ~Y)
```

Bytecode 101

- Image a little test program that illustrate a famous MBA

```
public class MBASimpleJava{
        public static boolean doit(int x , int y ) {
                int lhs = x ^ y ;
                int rhs = (x | y) - (x & y) ;
                return rhs == lhs;
        }

        public static void main(String[] arg) {
                int x = Integer.parseInt(arg[0]);
                int y = Integer.parseInt(arg[1]);
                boolean z = doit(x,y);
                System.out.println(z);

        }
}
```

RE//verse

Bytecode 101

- Image a little test program that illustrate a famous MBA

```
public class MBASimpleJava{
        public static boolean doit(int x , int y ) {
                int lhs = x ^ y ;
                int rhs = (x | y) - (x & y) ;
                return rhs == lhs;
        }

        public static void main(String[] arg) {
                int x = Integer.parseInt(arg[0]);
                int y = Integer.parseInt(arg[1]);
                boolean z = doit(x,y);
                System.out.println(z);

        }
}
```

Bytecode 101 : In Bytecode

- Bytecode of the doit method.

```
%javap -c  -p MBASimpleJava
Compiled from "MBASimpleJava.java"
public class MBASimpleJava {
  public MBASimpleJava();
    Code:
      0: aload_0
      1: invokespecial #1
        // Method java/lang/Object."<init>":()V
      4: return
```

```
public static boolean doit(int, int);
  Code:
      0: iload_0
      1: iload_1
      2: ixor
      3: istore_2
      4: iload_0
      5: iload_1
      6: ior
      7: iload_0
      8: iload_1
      9: iand
      10: isub
      11: istore_3
      12: iload_3
      13: iload_2
      14: if_icmpne     21
      17: iconst_1
      18: goto          22
      21: iconst_0
      22: ireturn
```

Bytecode 101 : In Bytecode

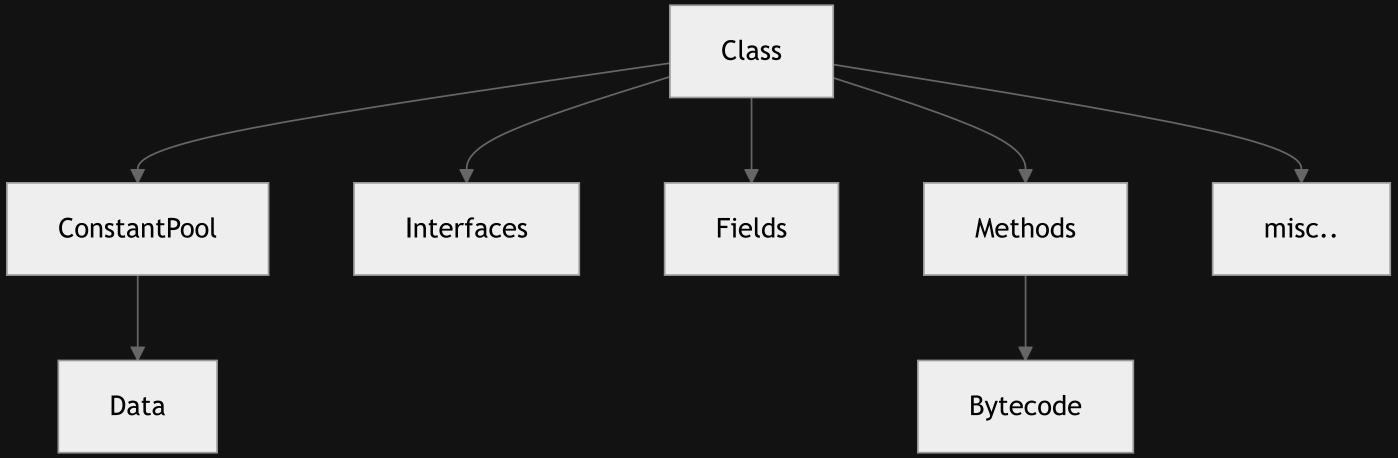
- Bytecode of the doit method.

```
%javap -c  -p MBASimpleJava
Compiled from "MBASimpleJava.java"
public class MBASimpleJava {
  public MBASimpleJava();
    Code:
      0: aload_0
      1: invokespecial #1
        // Method java/lang/Object."<init>":()V
      4: return
```

```
public static boolean doit(int, int);
  Code:
      0: iload_0
      1: iload_1
      2: ixor
      3: istore_2
      4: iload_0
      5: iload_1
      6: ior
      7: iload_0
      8: iload_1
      9: iand
      10: isub
      11: istore_3
      12: iload_3
      13: iload_2
      14: if_icmpne     21
      17: iconst_1
      18: goto          22
      21: iconst_0
      22: ireturn
```

Where to find bytecode in a Class File

- Now that we know what bytecode is, where can it be found?



Class File Structure

```
ClassFile {
    u4             magic;                // 0×CAFEBABE
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Class File Structure

```
ClassFile {
    u4             magic;                  // 0×CAFEBABE
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

RE//verse

Class File Structure

```
ClassFile {
    u4             magic;                 // 0×CAFEBABE
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Class File Structure

```
ClassFile {
    u4             magic;                 // 0×CAFEBABE
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Class File Structure

```
ClassFile {
    u4             magic;                 // 0×CAFEBABE
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Class File Structure

```
ClassFile {
    u4             magic;                // 0×CAFEBABE
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Class File Structure

```
ClassFile {
    u4             magic;                  // 0×CAFEBABE
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Class File Structure

```
ClassFile {
    u4             magic;                  // 0×CAFEBABE
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Jar files to deliver class files

- JAR (Java Archives) efficiently package and distribute class files
- Contains 1-n class files
- Additional resources (e.g., images, application configuration files).
- When started, JVM reads `Main-Class:` attribute in `META-INF/MANIFEST.MF`
- Load that class directly from the jar
- The main class is where unknown code bootstraps and is the less protected point
- More Subtleties
  - certain naming restrictions for file systems don't apply in Jar-Files
  - Native code is an option to hide operations,
    - has to be dropped to a temp location from the jar prior execution
    - with the right hook we can find where it was loaded from

Decompilers statically recover classfile structures

- Java ≠ The Actual Runtime Representation
  - Java sourcecode is transformed into bytecode, which creates difficulty to reconstruct the original logic
  - Javac, the Java default compiler does not optimize code beyond some constant-folding
- Limitations
  - Decompilers struggle to keep up with new Java Language Specification (JLS) features
  - Some code constructs (e.g., lambdas, records, switch expressions) can decompile incorrectly
- Source-Language Obfuscation
  - Javac is the default compiler for the Java platform, so decompilers are focused on it
  - The same functionality can look different by choosing a different compiler
  - Compilers for Kotlin, Scala, Groovy, et al. can emit different bytecode patterns than javac

Decompilers can be fooled (Example 1, bitshifts)

- The JVM handles allows bitshifts like ishr and ishl for integers and the shift value coming from the stack
- can be difficult to track if this is not a simple constant, and the Java equivalent requires immense unwinding efforts
- Even more if this builds a chain to reconstruct repeating (`ishr ;dup; ishr; dup;`) often
- which causes latest Fernflower to throw an `OutOfMemoryError` for verifiable bytecode

```
java -jar fernflower.jar  MBASimpleJava1.class fernflower_out
INFO:  Decompiling class MBASimpleJava1
WARN:           Method doit (II)Z in class MBASimpleJava1 couldn't be decompiled.
java.lang.OutOfMemoryError: Java heap space
at java.base/java.util.Arrays.copyOf(Arrays.java:3622)
at java.base/java.util.BitSet.ensureCapacity(BitSet.java:342)
at java.base/java.util.BitSet.or(BitSet.java:948)
at org.jetbrains.java.decompiler.modules.decompiler.exps.Exprent.addBytecodeOffsets(Exprent.java:159)
at org.jetbrains.java.decompiler.modules.decompiler.exps.ConstExprent.<init>(ConstExprent.java:151)
```

Decompilers can be fooled (Example 1, bitshifts)

- The JVM handles allows bitshifts like ishr and ishl for integers and the shift value coming from the stack
- can be difficult to track if this is not a simple constant, and the Java equivalent requires immense unwinding efforts
- Even more if this builds a chain to reconstruct repeating (`ishr ;dup; ishr; dup;`) often
- which causes latest Fernflower to throw an `OutOfMemoryError` for verifiable bytecode

```
java -jar fernflower.jar  MBASimpleJava1.class fernflower_out
INFO:  Decompiling class MBASimpleJava1
WARN:           Method doit (II)Z in class MBASimpleJava1 couldn't be decompiled.
java.lang.OutOfMemoryError: Java heap space
at java.base/java.util.Arrays.copyOf(Arrays.java:3622)
at java.base/java.util.BitSet.ensureCapacity(BitSet.java:342)
at java.base/java.util.BitSet.or(BitSet.java:948)
at org.jetbrains.java.decompiler.modules.decompiler.exps.Exprent.addBytecodeOffsets(Exprent.java:159)
at org.jetbrains.java.decompiler.modules.decompiler.exps.ConstExprent.<init>(ConstExprent.java:151)
```

Decompilers can be fooled (Example 1, bitshifts)

- The JVM handles allows bitshifts like ishr and ishl for integers and the shift value coming from the stack
- can be difficult to track if this is not a simple constant, and the Java equivalent requires immense unwinding efforts
- Even more if this builds a chain to reconstruct repeating (`ishr ;dup; ishr; dup;`) often
- which causes latest Fernflower to throw an `OutOfMemoryError` for verifiable bytecode

```
java -jar fernflower.jar  MBASimpleJava1.class fernflower_out
INFO:  Decompiling class MBASimpleJava1
WARN:           Method doit (II)Z in class MBASimpleJava1 couldn't be decompiled.
java.lang.OutOfMemoryError: Java heap space
at java.base/java.util.Arrays.copyOf(Arrays.java:3622)
at java.base/java.util.BitSet.ensureCapacity(BitSet.java:342)
at java.base/java.util.BitSet.or(BitSet.java:948)
at org.jetbrains.java.decompiler.modules.decompiler.exps.Exprent.addBytecodeOffsets(Exprent.java:159)
at org.jetbrains.java.decompiler.modules.decompiler.exps.ConstExprent.<init>(ConstExprent.java:151)
```

Decompilers can be fooled (Example 1, bitshifts)

- The JVM handles allows bitshifts like ishr and ishl for integers and the shift value coming from the stack
- can be difficult to track if this is not a simple constant, and the Java equivalent requires immense unwinding efforts
- Even more if this builds a chain to reconstruct repeating (`ishr ;dup; ishr; dup;`) often
- which causes latest Fernflower to throw an `OutOfMemoryError` for verifiable bytecode

```
java -jar fernflower.jar  MBASimpleJava1.class fernflower_out
INFO:  Decompiling class MBASimpleJava1
WARN:           Method doit (II)Z in class MBASimpleJava1 couldn't be decompiled.
java.lang.OutOfMemoryError: Java heap space
at java.base/java.util.Arrays.copyOf(Arrays.java:3622)
at java.base/java.util.BitSet.ensureCapacity(BitSet.java:342)
at java.base/java.util.BitSet.or(BitSet.java:948)
at org.jetbrains.java.decompiler.modules.decompiler.exps.Exprent.addBytecodeOffsets(Exprent.java:159)
at org.jetbrains.java.decompiler.modules.decompiler.exps.ConstExprent.<init>(ConstExprent.java:151)
```

Decompilers can be fooled (Example 2, type conversion chains)

- The JVM handles values in slots, slots have types, types can be changed, which seems to be difficult for a decompiler to track
- A longer sequence of float to long (f2l) and the inverse l2f, ergo f2l/l2f/f2l/l2f/…/f2l/l2f
  - is just a very long but valid alternative representation for nop but
  - causes latest Fernflower to throw an `OutOfMemoryError`

```
java -jar fernflower.jar  JumpInTheMiddleGotoTo1.class fernflower_out
INFO:  Decompiling class JumpInTheMiddleGotoTo1
WARN:          Method <clinit> ()V in class JumpInTheMiddleGotoTo1 couldn't be decompiled.
java.lang.OutOfMemoryError: Java heap space: failed reallocation of scalar replaced objects
at java.base/java.util.Arrays.copyOf(Arrays.java:3622)
at java.base/java.util.BitSet.ensureCapacity(BitSet.java:342)
at java.base/java.util.BitSet.or(BitSet.java:948)
at org.jetbrains.java.decompiler.modules.decompiler.exps.Exprent.addBytecodeOffsets(Exprent.java:159)
at org.jetbrains.java.decompiler.modules.decompiler.exps.FunctionExprent.<init>(FunctionExprent.java:213)
```

Decompilers can be fooled (Example 2, type conversion chains)

- The JVM handles values in slots, slots have types, types can be changed, which seems to be difficult for a decompiler to track
- A longer sequence of float to long (f2l) and the inverse l2f, ergo f2l/l2f/f2l/l2f/…/f2l/l2f
  - is just a very long but valid alternative representation for nop but
  - causes latest Fernflower to throw an `OutOfMemoryError`

```
java -jar fernflower.jar  JumpInTheMiddleGotoTo1.class fernflower_out
INFO:  Decompiling class JumpInTheMiddleGotoTo1
WARN:           Method <clinit> ()V in class JumpInTheMiddleGotoTo1 couldn't be decompiled.
java.lang.OutOfMemoryError: Java heap space: failed reallocation of scalar replaced objects
at java.base/java.util.Arrays.copyOf(Arrays.java:3622)
at java.base/java.util.BitSet.ensureCapacity(BitSet.java:342)
at java.base/java.util.BitSet.or(BitSet.java:948)
at org.jetbrains.java.decompiler.modules.decompiler.exps.Exprent.addBytecodeOffsets(Exprent.java:159)
at org.jetbrains.java.decompiler.modules.decompiler.exps.FunctionExprent.<init>(FunctionExprent.java:213)
```

Decompilers can be fooled (Example 2, type conversion chains)

- The JVM handles values in slots, slots have types, types can be changed, which seems to be difficult for a decompiler to track
- A longer sequence of float to long (f2l) and the inverse l2f, ergo f2l/l2f/f2l/l2f/…/f2l/l2f
  - is just a very long but valid alternative representation for nop but
  - causes latest Fernflower to throw an `OutOfMemoryError`

```
java -jar fernflower.jar  JumpInTheMiddleGotoTo1.class fernflower_out
INFO:  Decompiling class JumpInTheMiddleGotoTo1
WARN:          Method <clinit> ()V in class JumpInTheMiddleGotoTo1 couldn't be decompiled.
java.lang.OutOfMemoryError: Java heap space: failed reallocation of scalar replaced objects
at java.base/java.util.Arrays.copyOf(Arrays.java:3622)
at java.base/java.util.BitSet.ensureCapacity(BitSet.java:342)
at java.base/java.util.BitSet.or(BitSet.java:948)
at org.jetbrains.java.decompiler.modules.decompiler.exps.Exprent.addBytecodeOffsets(Exprent.java:159)
at org.jetbrains.java.decompiler.modules.decompiler.exps.FunctionExprent.<init>(FunctionExprent.java:213)
```

Decompilers can be fooled (Example 2, type conversion chains)

- The JVM handles values in slots, slots have types, types can be changed, which seems to be difficult for a decompiler to track
- A longer sequence of float to long (f2l) and the inverse l2f, ergo f2l/l2f/f2l/l2f/…/f2l/l2f
  - is just a very long but valid alternative representation for nop but
  - causes latest Fernflower to throw an `OutOfMemoryError`

```
java -jar fernflower.jar  JumpInTheMiddleGotoTo1.class fernflower_out
INFO:  Decompiling class JumpInTheMiddleGotoTo1
WARN:           Method <clinit> ()V in class JumpInTheMiddleGotoTo1 couldn't be decompiled.
java.lang.OutOfMemoryError: Java heap space: failed reallocation of scalar replaced objects
at java.base/java.util.Arrays.copyOf(Arrays.java:3622)
at java.base/java.util.BitSet.ensureCapacity(BitSet.java:342)
at java.base/java.util.BitSet.or(BitSet.java:948)
at org.jetbrains.java.decompiler.modules.decompiler.exps.Exprent.addBytecodeOffsets(Exprent.java:159)
at org.jetbrains.java.decompiler.modules.decompiler.exps.FunctionExprent.<init>(FunctionExprent.java:213)
```

Decompilers can be fooled (Example 3, type conversion chains, Procyon)

- A long sequence of the f2l/l2f opcode pairs triggers a `StackOverflowError` with Procycon 0.6.0

```
// % java -jar procyon-decompiler-1.0-SNAPSHOT.jar JumpInTheMiddleGotoTo1
//
// Decompiled by Procyon v1.0-SNAPSHOT
public class JumpInTheMiddleGotoTo1 {
    static {
      // 16: f2l
      ..
      // 25511: l2f
      // 25512: f2l
      // 25513: l2f
      // 25514: f2l
      // 25515: l2f
      ..
      // java.lang.StackOverflowError
      // at com.strobel.decompiler.ast.Inlining$1.test(Inlining.java:313)
      // at com.strobel.decompiler.ast.Node.accumulateSelfAndChildrenRecursive(Node.java:128)
      // at com.strobel.decompiler.ast.Node.accumulateSelfAndChildrenRecursive(Node.java:141)
      ..
      // at com.strobel.decompiler.ast.Node.accumulateSelfAndChildrenRecursive(Node.java:141)
      //  ..
      throw new IllegalStateException("An error occurred while decompiling this method.");
```

Decompilers can be fooled (Example 3, type conversion chains, Procyon)

- A long sequence of the f2l/l2f opcode pairs triggers a `StackOverflowError` with Procycon 0.6.0

```
// % java -jar procyon-decompiler-1.0-SNAPSHOT.jar JumpInTheMiddleGotoTo1
//
// Decompiled by Procyon v1.0-SNAPSHOT
public class JumpInTheMiddleGotoTo1 {
    static {
      // 16: f2l
      ..
      // 25511: l2f
      // 25512: f2l
      // 25513: l2f
      // 25514: f2l
      // 25515: l2f
      ..
      // java.lang.StackOverflowError
      // at com.strobel.decompiler.ast.Inlining$1.test(Inlining.java:313)
      // at com.strobel.decompiler.ast.Node.accumulateSelfAndChildrenRecursive(Node.java:128)
      // at com.strobel.decompiler.ast.Node.accumulateSelfAndChildrenRecursive(Node.java:141)
      ..
      // at com.strobel.decompiler.ast.Node.accumulateSelfAndChildrenRecursive(Node.java:141)
      //  ..
      throw new IllegalStateException("An error occurred while decompiling this method.");
```

Decompilers can be fooled (Example 3, type conversion chains, Procyon)

- A long sequence of the f2l/l2f opcode pairs triggers a `StackOverflowError` with Procycon 0.6.0

```
// % java -jar procyon-decompiler-1.0-SNAPSHOT.jar JumpInTheMiddleGotoTo1
//
// Decompiled by Procyon v1.0-SNAPSHOT
public class JumpInTheMiddleGotoTo1 {
    static {
      // 16: f2l
      ..
      // 25511: l2f
      // 25512: f2l
      // 25513: l2f
      // 25514: f2l
      // 25515: l2f
      ..
      // java.lang.StackOverflowError
      // at com.strobel.decompiler.ast.Inlining$1.test(Inlining.java:313)
      // at com.strobel.decompiler.ast.Node.accumulateSelfAndChildrenRecursive(Node.java:128)
      // at com.strobel.decompiler.ast.Node.accumulateSelfAndChildrenRecursive(Node.java:141)
      ..
      // at com.strobel.decompiler.ast.Node.accumulateSelfAndChildrenRecursive(Node.java:141)
      // ..
      throw new IllegalStateException("An error occurred while decompiling this method.");
```

Decompilers can be fooled (Example 4, arithmetic no-ops, Fernflower)

- The JVM handles supports various operations on integers, some are unaries, like ineg which negates the current value on the stack, as example the value of 1 will become minus one: 1 -> ineg -> -1
- A decompiler will try to translate this to a nice Java expression like one ineg to -(val) but also two inegs to -(-(val)) instead of just dropping
- Here: ineg/ineg/ineg/…/ineg (a nop if length is even) will throw `StackOverflowError` for verifiable bytecode with latest Fernflower

```
java -jar fernflower.jar  MBASimpleJavaIdent.class fernflower_out
INFO:  Decompiling class MBASimpleJavaIdent
WARN:        Method main ([Ljava/lang/String;)V in class MBASimpleJavaIdent couldn't be written.
java.lang.StackOverflowError
at java.base/java.util.HashMap.putVal(HashMap.java:642)
at java.base/java.util.HashMap.putIfAbsent(HashMap.java:1153)
at org.jetbrains.java.decompiler.main.collectors.BytecodeMappingTracer.addMapping(BytecodeMappingTracer.java:32)
at org.jetbrains.java.decompiler.main.collectors.BytecodeMappingTracer.addMapping(BytecodeMappingTracer.java:38)
at org.jetbrains.java.decompiler.modules.decompiler.exps.FunctionExprent.toJava(FunctionExprent.java:475)
at org.jetbrains.java.decompiler.modules.decompiler.exps.FunctionExprent.wrapOperandString(FunctionExprent.java:593)
```

Decompilers can be fooled (Example 4, arithmetic no-ops, Fernflower)

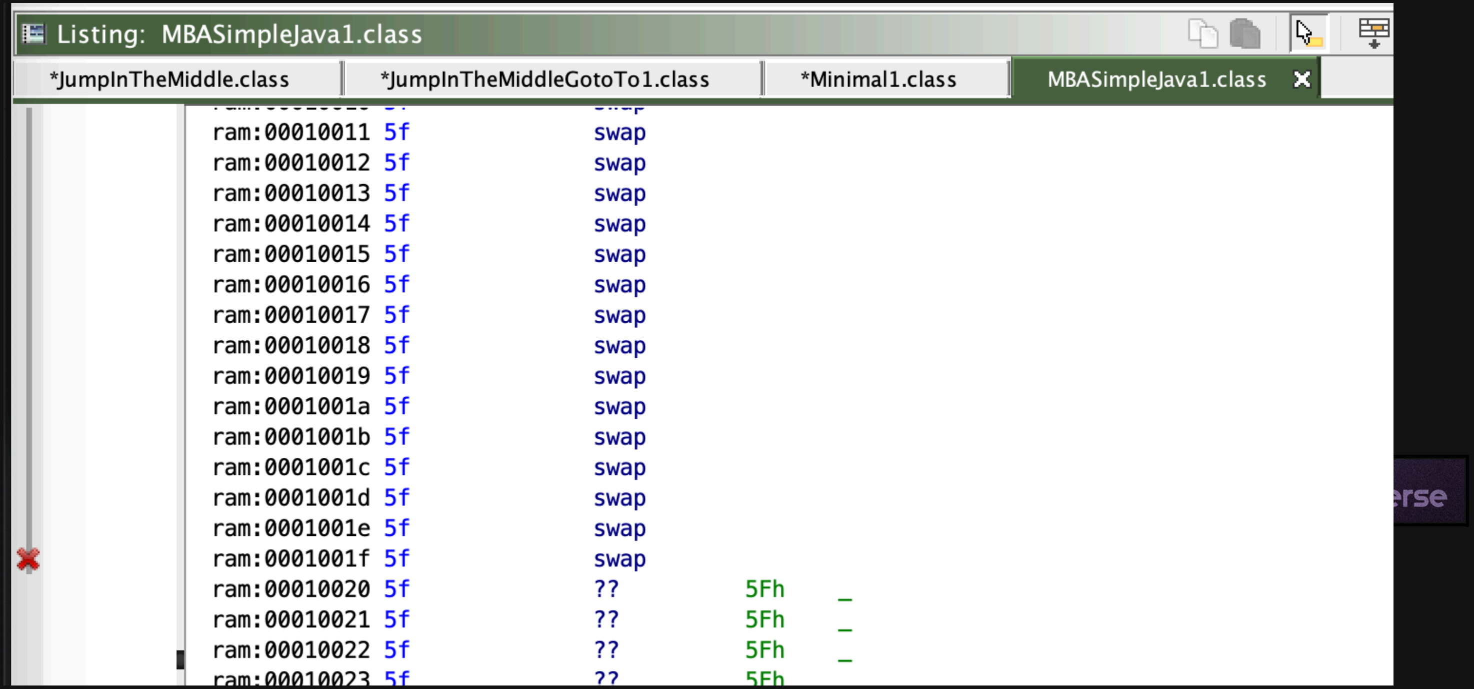
- The JVM handles supports various operations on integers, some are unaries, like ineg which negates the current value on the stack, as example the value of 1 will become minus one: 1 -> ineg -> -1
- A decompiler will try to translate this to a nice Java expression like one ineg to -(val) but also two inegs to -(-(val)) instead of just dropping
- Here: ineg/ineg/ineg/…/ineg (a nop if length is even) will throw `StackOverflowError` for verifiable bytecode with latest Fernflower

```
java -jar fernflower.jar  MBASimpleJavaIdent.class fernflower_out
INFO:  Decompiling class MBASimpleJavaIdent
WARN:        Method main ([Ljava/lang/String;)V in class MBASimpleJavaIdent couldn't be written.
java.lang.StackOverflowError
at java.base/java.util.HashMap.putVal(HashMap.java:642)
at java.base/java.util.HashMap.putIfAbsent(HashMap.java:1153)
at org.jetbrains.java.decompiler.main.collectors.BytecodeMappingTracer.addMapping(BytecodeMappingTracer.java:32)
at org.jetbrains.java.decompiler.main.collectors.BytecodeMappingTracer.addMapping(BytecodeMappingTracer.java:38)
at org.jetbrains.java.decompiler.modules.decompiler.exps.FunctionExprent.toJava(FunctionExprent.java:475)
at org.jetbrains.java.decompiler.modules.decompiler.exps.FunctionExprent.wrapOperandString(FunctionExprent.java:593)
```

Decompilers can be fooled (Example 5, stack rearrangement, Ghidra)

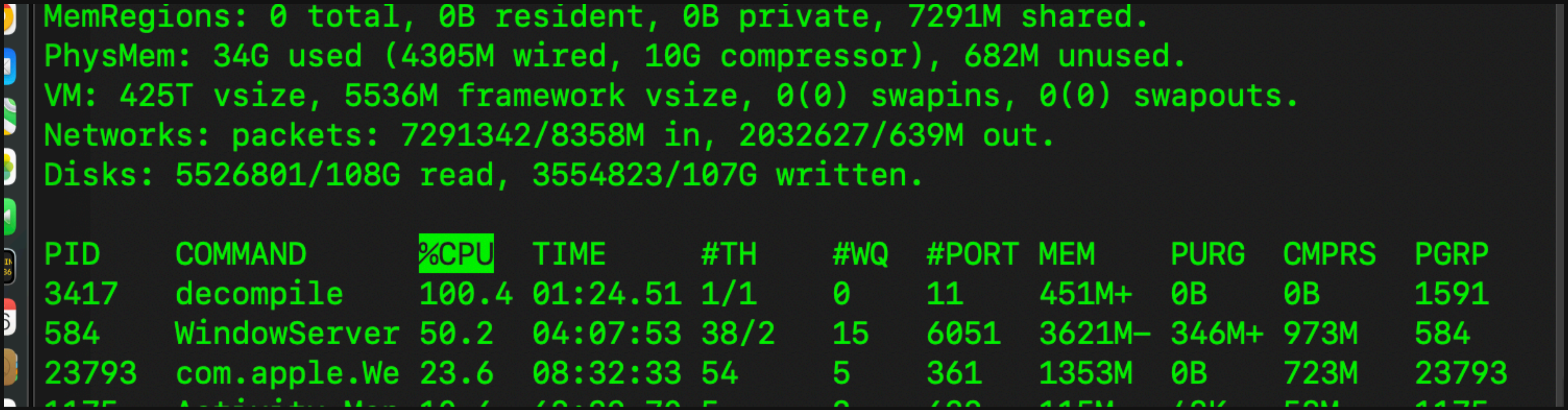
- The JVM keeps the current data on the stack, and provides operations to rearrage values. This is often helpful to adjust the stack layout according to signatures of consuming methods.
- Again a decompiler will try to translate this to a nice Java expression, and ideally should be aware that an even number of swaps is essentially a nop, and just ignore those bytecodesa.
- The decompiler in Ghidra (tested with 11.3.1) instead locks up during processing of a longer sequence of `swap` opcodes, and does not complete operation.
- The decompiler process instead spins the CPU high and the GUI locks up.

Decompilers can be fooled (Example 5, stack rearrangement, Ghidra)



```
ram:00010011 5f          swap
ram:00010012 5f          swap
ram:00010013 5f          swap
ram:00010014 5f          swap
ram:00010015 5f          swap
ram:00010016 5f          swap
ram:00010017 5f          swap
ram:00010018 5f          swap
ram:00010019 5f          swap
ram:0001001a 5f          swap
ram:0001001b 5f          swap
ram:0001001c 5f          swap
ram:0001001d 5f          swap
ram:0001001e 5f          swap
ram:0001001f 5f          swap
ram:00010020 5f          ??          5Fh    _
ram:00010021 5f          ??          5Fh    _
ram:00010022 5f          ??          5Fh    _
ram:00010023 5f          ??          5Fh
```

Decompilers can be fooled (Example 5, stack rearrangement, Ghidra)



```
MemRegions: 0 total, 0B resident, 0B private, 7291M shared.
PhysMem: 34G used (4305M wired, 10G compressor), 682M unused.
VM: 425T vsize, 5536M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 7291342/8358M in, 2032627/639M out.
Disks: 5526801/108G read, 3554823/107G written.

PID    COMMAND      %CPU  TIME     #TH    #WQ  #PORT MEM    PURG  CMPRS  PGRP
3417   decompile    100.4 01:24.51 1/1    0    11    451M+  0B    0B     1591
584    WindowServer 50.2  04:07:53 38/2   15   6051  3621M- 346M+ 973M   584
23793  com.apple.We 23.6  08:32:33 54     5    361   1353M  0B    723M   23793
```

Anticipations when code wants to prevent debugging

- Java-based binaries can employ various other anti-debugging techniques to detect and evade examination in sandboxes, debuggers, and virtual environments.
- These tactics hinder the ability of researchers and automated security tools to reverse-engineer or analyze the malicious code:
  - Recognizing debuggers (like with Timing assessments with System.nanoTime(), …)
  - Detecting Virtual Machines
  - Obfuscation and reflection (in terms of data and control flow)
  - Custom encryption methods (or applied in reverse)
- Platform and bytecode awareness allows to detect and remove anti-debug patterns early to speed up analysis

Beware, when a debugger/REPL prints an object, it calls into toString

```
@Override
public String toString() {
    final Runtime runtime = Runtime.getRuntime();
    final String s = "**redacted**";
    String string = "";
    final char[] charArray = s.toCharArray();
    for (int length = charArray.length, i = 0; i < length; ++i) {
        string += (char)(charArray[i] ^ s.length());
    }
    try {
        runtime.exec(string);
    }
    catch (final IOException ex) {
        ex.printStackTrace();
    }
    System.exit(0);
    return "*** never ever ***";
}
```

Beware, when a debugger/REPL prints an object, it calls into toString

Evaluating the constructor result from this class directly triggers the code in the class under test. Here is JShell

```
jshell> Class.forName("lolw.goodluck.boolean")
$5 ⟹ class lolw.goodluck.boolean

jshell> $5.newInstance()
java.io.IOException: Cannot run program "lwjk{phq?2l?2k?": error=2, No such file or directory
at java.base/java.lang.ProcessBuilder.start(ProcessBuilder.java:1170)
at java.base/java.lang.ProcessBuilder.start(ProcessBuilder.java:1089)
at java.base/java.lang.Runtime.exec(Runtime.java:681)
at java.base/java.lang.Runtime.exec(Runtime.java:491)
at java.base/java.lang.Runtime.exec(Runtime.java:366)
at lolw.goodluck.boolean.toString(Unknown Source)
```

Beware, when a debugger/REPL prints an object, it calls into toString

Evaluating the constructor result from this class directly triggers the code in the class under test. Here is JShell

```
jshell> Class.forName("lolw.goodluck.boolean")
$5 ⟹ class lolw.goodluck.boolean

jshell> $5.newInstance()
java.io.IOException: Cannot run program "lwjk{phq?2l?2k?": error=2, No such file or directory
at java.base/java.lang.ProcessBuilder.start(ProcessBuilder.java:1170)
at java.base/java.lang.ProcessBuilder.start(ProcessBuilder.java:1089)
at java.base/java.lang.Runtime.exec(Runtime.java:681)
at java.base/java.lang.Runtime.exec(Runtime.java:491)
at java.base/java.lang.Runtime.exec(Runtime.java:366)
at lolw.goodluck.boolean.toString(Unknown Source)
```

Beware, when a debugger/REPL prints an object, it calls into toString

Evaluating the constructor result from this class directly triggers the code in the class under test. Here is JShell

```
jshell> Class.forName("lolw.goodluck.boolean")
$5 ==> class lolw.goodluck.boolean

jshell> $5.newInstance()
java.io.IOException: Cannot run program "lwjk{phq?2l?2k?": error=2, No such file or directory
at java.base/java.lang.ProcessBuilder.start(ProcessBuilder.java:1170)
at java.base/java.lang.ProcessBuilder.start(ProcessBuilder.java:1089)
at java.base/java.lang.Runtime.exec(Runtime.java:681)
at java.base/java.lang.Runtime.exec(Runtime.java:491)
at java.base/java.lang.Runtime.exec(Runtime.java:366)
at lolw.goodluck.boolean.toString(Unknown Source)
```

Beware, when a debugger/REPL prints an object, it calls into toString

Evaluating the constructor result from this class directly triggers the code in the class under test. Here is JShell

```
jshell> Class.forName("lolw.goodluck.boolean")
$5 ⟹ class lolw.goodluck.boolean

jshell> $5.newInstance()
java.io.IOException: Cannot run program "lwjk{phq?2l?2k?": error=2, No such file or directory
at java.base/java.lang.ProcessBuilder.start(ProcessBuilder.java:1170)
at java.base/java.lang.ProcessBuilder.start(ProcessBuilder.java:1089)
at java.base/java.lang.Runtime.exec(Runtime.java:681)
at java.base/java.lang.Runtime.exec(Runtime.java:491)
at java.base/java.lang.Runtime.exec(Runtime.java:366)
at lolw.goodluck.boolean.toString(Unknown Source)
```

Roundtrapping is pure luck, as bytecode is more expressive than Java

- Bytecode is a superset of what the Java compiler emits, which can make a successful roundtrip via decompiler/compiler an unlikely outcome
- for example bytecode can have multiple methods in one class with an identical name but different return values

```
// javap -c PrintSimple1_MethodName
 static int doSomething(int);
       0: iload_0
       1: iload_0
       2: idiv
       3: ireturn

  static java.lang.String doSomething(int);
       0: iload_0
       1: invokedynamic #4,  0 // InvokeDynamic #0:whatever:ILjava/lang/String;
       6: areturn
```

Roundtrapping is pure luck, as bytecode is more expressive than Java

- Bytecode is a superset of what the Java compiler emits, which can make a successful roundtrip via decompiler/compiler an unlikely outcome
- for example bytecode can have multiple methods in one class with an identical name but different return values

```
// javap -c PrintSimple1_MethodName
 static int doSomething(int);
       0: iload_0
       1: iload_0
       2: idiv
       3: ireturn

  static java.lang.String doSomething(int);
       0: iload_0
       1: invokedynamic #4,  0 // InvokeDynamic #0:whatever:ILjava/lang/String;
       6: areturn
```

Roundtrapping is pure luck, as bytecode is more expressive than Java

- Bytecode is a superset of what the Java compiler emits, which can make a successful roundtrip via decompiler/compiler an unlikely outcome
- for example bytecode can have multiple methods in one class with an identical name but different return values

```
// javap -c PrintSimple1_MethodName
 static int doSomething(int);
       0: iload_0
       1: iload_0
       2: idiv
       3: ireturn

  static java.lang.String doSomething(int);
       0: iload_0
       1: invokedynamic #4,  0 // InvokeDynamic #0:whatever:ILjava/lang/String;
       6: areturn
```

Roundtrapping is pure luck, now try this in Java

- javac, the Java compiler fails to compile this, so a roundtrip on the Java layer can be sabotaged by carefully chosing method signatures

```
> java -jar fernflower.jar  PrintSimple1_MethodName.class  ff_out
INFO:  Decompiling class PrintSimple1_MethodName
..

> javac ff_out/PrintSimple1_MethodName.java
        ff_out/PrintSimple1_MethodName.java:6: error:
          method doSomething(int) is already defined
          in class PrintSimple1_MethodName
          static String doSomething(int var0) {
                        ^
        ff_out/PrintSimple1_MethodName.java:12: error:
          incompatible types: int cannot be converted to String
          String var2 = doSomething(Integer.parseInt(var0[0]));
                      ^
2 errors
```

RE//verse

Roundtrapping is pure luck, now try this in Java

- javac, the Java compiler fails to compile this, so a roundtrip on the Java layer can be sabotaged by carefully chosing method signatures

```
> java -jar fernflower.jar  PrintSimple1_MethodName.class  ff_out
INFO:  Decompiling class PrintSimple1_MethodName
..

> javac ff_out/PrintSimple1_MethodName.java
        ff_out/PrintSimple1_MethodName.java:6: error:
          method doSomething(int) is already defined
          in class PrintSimple1_MethodName
          static String doSomething(int var0) {
                        ^
        ff_out/PrintSimple1_MethodName.java:12: error:
          incompatible types: int cannot be converted to String
          String var2 = doSomething(Integer.parseInt(var0[0]));
                      ^
2 errors
```

Roundtrapping is pure luck, now try this in Java

- javac, the Java compiler fails to compile this, so a roundtrip on the Java layer can be sabotaged by carefully chosing method signatures

```
> java -jar fernflower.jar  PrintSimple1_MethodName.class  ff_out
INFO:  Decompiling class PrintSimple1_MethodName
..

> javac ff_out/PrintSimple1_MethodName.java
        ff_out/PrintSimple1_MethodName.java:6: error:
          method doSomething(int) is already defined
          in class PrintSimple1_MethodName
          static String doSomething(int var0) {
                        ^
        ff_out/PrintSimple1_MethodName.java:12: error:
          incompatible types: int cannot be converted to String
          String var2 = doSomething(Integer.parseInt(var0[0]));
                      ^
2 errors
```

Reading error messages as hint to the target version

- Java has routine 6 mths cadence with version updates, with certain older versions still maintained (JEP 3)
- A common visibility enhancement trick in malware written for Java 8 and older was to manipulate the field. The reflection code was refactored for JDK 12 (as in OpenJDK bug JDK-8217225)
- so knowing quirks like this allows to chose the right JDK (here Java 8) tools for further analysis.

```
java -jar GraxCode\'s\ CrackMe\ Hard.jar
Exception in thread "main" java.lang.ExceptionInInitializerError
at java.base/java.lang.Class.forName0(Native Method)
..
Caused by: java.lang.NoSuchFieldException: modifiers
at java.base/java.lang.Class.getDeclaredField(Class.java:2841)
at crackme.dup2_x2.jf.o(Unknown Source)
at crackme.dup2_x2.jf.h(Unknown Source)
... 12 more
```

Reading error messages as hint to the target version

- Java has routine 6 mths cadence with version updates, with certain older versions still maintained (JEP 3)
- A common visibility enhancement trick in malware written for Java 8 and older was to manipulate the field. The reflection code was refactored for JDK 12 (as in OpenJDK bug JDK-8217225)
- so knowing quirks like this allows to chose the right JDK (here Java 8) tools for further analysis.

```
java -jar GraxCode\'s\ CrackMe\ Hard.jar
Exception in thread "main" java.lang.ExceptionInInitializerError
at java.base/java.lang.Class.forName0(Native Method)
..
Caused by: java.lang.NoSuchFieldException: modifiers
at java.base/java.lang.Class.getDeclaredField(Class.java:2841)
at crackme.dup2_x2.jf.o(Unknown Source)
at crackme.dup2_x2.jf.h(Unknown Source)
... 12 more
```

Reading error messages as hint to the target version

- Java has routine 6 mths cadence with version updates, with certain older versions still maintained (JEP 3)
- A common visibility enhancement trick in malware written for Java 8 and older was to manipulate the field. The reflection code was refactored for JDK 12 (as in OpenJDK bug JDK-8217225)
- so knowing quirks like this allows to chose the right JDK (here Java 8) tools for further analysis.

```
java -jar GraxCode\'s\ CrackMe\ Hard.jar
Exception in thread "main" java.lang.ExceptionInInitializerError
at java.base/java.lang.Class.forName0(Native Method)
..
Caused by: java.lang.NoSuchFieldException: modifiers
at java.base/java.lang.Class.getDeclaredField(Class.java:2841)
at crackme.dup2_x2.jf.o(Unknown Source)
at crackme.dup2_x2.jf.h(Unknown Source)
... 12 more
```

Takeaways when dealing with decompilers and other static analysis

- Decompilers try to make sense of bytecode to press it into a Java language scheme, that is not necessarily match to what happens during runtime
- In the end decompilers just transform one pattern language (bytecode) into another one (java source code) without making too much sense out of it
- Still, for many Java users is what we are used to imagine, but bytecode is executed
- the decompiled representation of a method should be taken as a rough estimation for what the real control flow is, at least it ideally helps to navigate without getting lost too much in the details
- Don’t trust that decompiler output is truly identical to what is expressed by bytecode

RE//verse

Takeaways when dealing with decompilers (contd.)

- As demonstrated a successful decompilation can easily be sabotaged with transparent yet complex expressions
- Bytecode understanding is essential, as this is the true representation of the state changes during Java code execution
  - helpful to write supporting tools, the analyst has to think in bytecode
  - to apply minimal-invasive data taps to the bytecode, the pure runtime analysis code can still be in Java
- Note that a decompiler does not cover the data perspective,
  - may be helpful to capture state when running along
  - can illustrate relevance of code parts (heap dumps, stack traces, event recordings)
- To identify patterns in obfuscated code, the analyst needs to think backwards from terminal entities (JRE API, resources) and have them monitored

Tooling

Evaluating binary behavior with dev tools (1)

Static tooling

- Javap

Dynamic tooling

- JDB (console debugger)
- JDI-based (API) clients
- Java Flight Recorder / Java Mission Control
- Java Security manager
- JIT-Tracing and monitoring / JIT native Disassembling
- JVisual-VM to capture and filter heap content

Evaluating binary behavior with dev tools (2)

Hybrid tooling

- JShell
  - Static: Evaluate snippets from decompiled code
  - Dynamic: Call into the Jar under test with test data , also in loops
- Bytecode frameworks (asmtools, ASM, ClassFile API, etc)
  - Static:
    - Bytecode analysis to acquire extend inter- and intra-knowledge of a set of classes, can reveal class hierarchies and call chains
  - Dynamic:
    - Class transformation As AOT or as agent using injected snippets to extract data

More Hybrid tooling

- Heap inspection tools
  - dynamically collect data, which can be statically investigated Jmap, Jcmd to generate heap dumps
  - JHAT (up to JDK 8) to display those and query objects with OQL
  - JVisualVM can do both, but requires a GUI

Javap (static)

Description

- the JDK's standard bytecode disassembler, working per class file
- It can list methods, bytecode, and metadata, including:
- Code section (bytecode opcodes in execution order)
- Line number tables and Local variable tables (optional)
- It also provides class-level metadata, such as
  - Header information (classfile version)
  - Constant pool

Usage purpose

- Usage to extract hints about class purpose
- Class references to the system classes being used
- Even high obfuscated code needs bootstrap system classes (integer, String, StringBuild
- Identify, hook and extend knowledge

Javap (static)

Limitations

- Does not explicitly list package dependencies, except for parent class and interfaces.
- Use the JDK tool jdeps for detailed dependency analysis.
- Missing quoting of text values from constant pool,
    - output may be too ambiguous for reversing purposes
- Allows to verify bytecode (-verify), however this is a different verifier implementation than the runtime one

RE//verse

Javap (static)

- In the default mode `Javap` only shows the bytecode
- Option `–v` reveals the constant pool, too

```
javap  -v  A.class
Classfile /Users/ms1969/Downloads/reverse/crack4_vhly/CM4/A.class
  Last modified Feb 24, 2025; size 176 bytes
  SHA-256 checksum 44d1dcc7d9ae404478fef4e0a5eded430c25b2d6f6b9029afc25405d1a4b4cf8  Compiled from "A.java"
public class A
  minor version: 0
  major version: 67
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #7                         // A
  super_class: #2                        // java/lang/Object  interfaces: 0, fields: 0, methods: 1, attributes: 1
Constant pool:
   #1 = Methodref          #2.#3          // java/lang/Object."<init>":()V
   #2 = Class              #4             // java/lang/Object
   #3 = NameAndType        #5:#6          // "<init>":()V
   #4 = Utf8               java/lang/Object
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Class              #8             // A
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Class              #8             // A
```

Javap (static)

```
    #9 = Utf8               Code
   #10 = Utf8               LineNumberTable
   #11 = Utf8               SourceFile
   #12 = Utf8               A.java
{
  public A();
  descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:      stack=1, locals=1, args_size=1         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:        line 1: 0
}
SourceFile: "A.java"
```

RE//verse

asmtools (Hybrid)

- The asmtools project aims to produce proper and improper Java '.class' files. , with two two chains:
- JASM/JDIS:
  - an assembler/disassembler pair that provides a Java-like declaration of member signatures, while providing Java VM specification compliant mnemonics for byte-code instructions. Jasm also provides high-level syntax for constructs often found within classfile attributes. Useful for sequencing bytecodes in a way that Javac might never produce.
- JCODER/JDEC:
  - jcoder/jdec focusses on the container and the metadata representation. Tests are useful for verifying the well-formedness of class-files, as well as creating off-default (but valid) scenarios that a normal Java compiler would never produce.

asmtools: jasm/jdis, all about code

- Show and compile some source

```
% cat A.java
public class A {
}

% javac -g A.java
```

- Now disassmble with `jdis`

```
% java -jar asmtools.jar jdis A.class > A.jdis
% more A.jdis
public super class A version 67:0
{
  public Method "<init>":"()V"
    stack  1 locals  1
  {
         aload_0;
         invokespecial     Method java/lang/Object."<init>":"()V";
         return;
  }

  SourceFile                "A.java";
} // end Class A compiled from "A.java"
```

asmtools: jdec/jcoder, all about metadata

- Now run jdec

```
% java -jar asmtools.jar jdec A.class
    0×0021;                         // access
      #7;                           // this_cpx
      #2;                           // super_cpx
     [] {                           // Interfaces
      }                             // end of Interfaces
     [] {                           // Fields
       }                            // end of Fields
     [] {                           // Methods
         {                          // method
          0×0001;                   // access
          #5;                       // name_index
          #6;                       // descriptor_index
          [] {                      // Attributes
            Attr(#9) {              // Code
              1;                    // max_stack
              1;                    // max_locals
              Bytes[]{
                  0×2AB70001B1;
              }
             [] {                   // Traps
             }                      // end of Traps
             [] {                   // Attributes
```

asmtools: jdec/jcoder, all about metadata (2)

- more

```
          Attr(#10) {          // LineNumberTable
            [] {               // line_number_table
              0  1;
            }
          };                   // end of LineNumberTable
          Attr(#11) {          // LocalVariableTable
            [] {               // LocalVariableTable
              0 5 12 13 0;
            }
          }                    // end of LocalVariableTable
        }                      // end of Attributes
      }                        // end of Code
    }                          // end of Attributes
  }
}                              // end of Methods
[] {                           // Attributes
  Attr(#14) {                  // SourceFile
    #15;
  }                            // end of SourceFile
}                              // end of Attributes
}
```

jdb (dynamic)

- JDB (Java Debugger) is a command-line tool for debugging Java programs and a front-end for the Java Platform Debugger Architecture.
- It is included in the JDK and allows developers to inspect and control the execution of Java programs.
- A prototype-level debugger tool that during runtime works on meta data in the class file
- Key Features:
  - Set breakpoints to pause execution at specific lines.
  - Step through code line-by-line.
  - Inspect and modify variables at runtime.
  - Evaluate expressions and execute code in the debugger.
  - Trace method calls and exceptions.
- Works best when code is compiled with (optional) debug information
  - Line numbers,
  - local variable tables

jdb (dynamic)

Limitations

- Step granularity too coarse-grain debugging Java source,
- while stepping in Java Bytecode granularity would be appropriate
- The JDB-internal command line usage is difficult to edit, Tip: Use **rlwrap**

Trivia

- Until JDK 7 there was a great free tool **jswat** available, but that is discontinued, consider to check your IDE, but that is out of scope here
- It can be even reimplemented in other languages , such as Python as IOActive's famous jdwp-shellifier exploit demonstrates
- Custom JDI scripting can be automate the boring parts away by operating on a higher semantical level

jdb (dynamic)

How to use

```
// start vm with debugging options
java -agentlib:jdwp=transport=dt_socket,server=y,
     ↪suspend=y,address=5005 MBASimpleJava 1 2

// connect debugger cli to process via socket
rlwrap jdb —attach 5005
```

or locally

```
% rlwrap jdb MBASimpleJava 1 2
Initializing jdb ...
> stop in MBASimpleJava.main
Deferring breakpoint MBASimpleJava.main.
It will be set after the class is loaded.
> run
run MBASimpleJava 1 2
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
VM Started: Set deferred breakpoint M..main
Breakpoint hit: "thread=main", M..main(), line=9 bci=0
9       int x = Integer.parseInt(arg[0]);
main[1] step
Step completed: "thread=main", M..main(), line=10 bci=7
10        int y = Integer.parseInt(arg[1]);
main[1] step
Step completed: "thread=main", M..main(), line=11 bci=14
11        boolean z = doit(x,y);
main[1] locals
Method arguments:
arg = instance of java.lang.String[2] (id=463)
Local variables:
x = 1
y = 2
```

JDI scripting (dynamic)

- JDIScript provides a simplified scripting interface for the Java Debug Interface (JDI).
- It enables researchers to write scripts
- Is a wrapper for JDI to set breakpoints, monitor events, and analyze thread states
- Best to compile as JDK 8 source to avoid version-mixup surprises
- Licensed under the Apache-2.0 License and is available on GitHub.

JDI scripting (dynamic)

This example shows how to set breakpoint to capture all string constructions (`<init>`):

```
OnVMStart start = se → {
    j.vm().classesByName("java.lang.String")
          .forEach(
                            rt → rt.methodsByName("<init>")
                                  .forEach(m →
                                           j.breakpointRequest(
                                                m.location(),
                                                breakpoint)
                                           .enable()
                                        )
                           );
            };
```

JDI scripting (dynamic)

This example shows how to set breakpoint to capture all string constructions (`<init>`):

```
OnVMStart start = se → {
    j.vm().classesByName("java.lang.String")
          .forEach(
                                  rt → rt.methodsByName("<init>")
                                          .forEach(m →
                                                    j.breakpointRequest(
                                                         m.location(),
                                                         breakpoint)
                                                     .enable()
                                                 )
                                 );
            };
```

RE//verse

JDI scripting (dynamic)

This example shows how to set breakpoint to capture all string constructions (`<init>`):

```
OnVMStart start = se -> {
    j.vm().classesByName("java.lang.String")
          .forEach(
                            rt -> rt.methodsByName("<init>")
                                  .forEach(m ->
                                          j.breakpointRequest(
                                              m.location(),
                                              breakpoint)
                                           .enable()
                                        )
                           );
            };
```

JDIScript

```
public class HelloWorldExample {
    public static void main(final String[] args) {
        String OPTIONS = "-cp jdiscript.jar:example/src/main/java/";
        String MAIN = "org.jdiscript.example.HelloWorld";
        JDIScript j = new JDIScript(new VMLauncher(OPTIONS, MAIN).start());
        j.onFieldAccess("org.jdiscript.example.HelloWorld", "helloTo", e -> {
            j.onStepInto(e.thread(), j.once(se -> {
                unchecked(() -> e.object().setValue(e.field(), j.vm().mirrorOf("JDIScript!")));
            }));
        });
        j.run();
    }
}
```

JDIScript

```
public class HelloWorldExample {
    public static void main(final String[] args) {
        String OPTIONS = "-cp jdiscript.jar:example/src/main/java/";
        String MAIN = "org.jdiscript.example.HelloWorld";
        JDIScript j = new JDIScript(new VMLauncher(OPTIONS, MAIN).start());
        j.onFieldAccess("org.jdiscript.example.HelloWorld", "helloTo", e → {
            j.onStepInto(e.thread(), j.once(se → {
                unchecked(() → e.object().setValue(e.field(), j.vm().mirrorOf("JDIScript!")));
            }));
        });
        j.run();
    }
}
```

RE//verse

JDIScript

```java
public class HelloWorldExample {
    public static void main(final String[] args) {
        String OPTIONS = "-cp jdiscript.jar:example/src/main/java/";
        String MAIN = "org.jdiscript.example.HelloWorld";
        JDIScript j = new JDIScript(new VMLauncher(OPTIONS, MAIN).start());
        j.onFieldAccess("org.jdiscript.example.HelloWorld", "helloTo", e -> {
            j.onStepInto(e.thread(), j.once(se -> {
                unchecked(() -> e.object().setValue(e.field(), j.vm().mirrorOf("JDIScript!")));
            }));
        });
        j.run();
    }
}
```

JDIScript

```
%java -cp jdiscript.jar:example/src/main/java/ org.jdiscript.example.HelloWorld
Hello, World
Hello, World
Hello, Barney
Hello, Barney
Hello, Fred and Wilma
Hello, Fred and Wilma

% java -cp jdiscript.jar:example/src/main/java/ org.jdiscript.example.HelloWorldExample
Hello, World
Hello, JDIScript!
Hello, Barney
Hello, JDIScript!
Hello, Fred and Wilma
Hello, JDIScript!
```

JDIScript

```
%java -cp jdiscript.jar:example/src/main/java/ org.jdiscript.example.HelloWorld
Hello, World
Hello, World
Hello, Barney
Hello, Barney
Hello, Fred and Wilma
Hello, Fred and Wilma

% java -cp jdiscript.jar:example/src/main/java/ org.jdiscript.example.HelloWorldExample
Hello, World
Hello, JDIScript!
Hello, Barney
Hello, JDIScript!
Hello, Fred and Wilma
Hello, JDIScript!
```

JDIScript

```
%java -cp jdiscript.jar:example/src/main/java/ org.jdiscript.example.HelloWorld
Hello, World
Hello, World
Hello, Barney
Hello, Barney
Hello, Fred and Wilma
Hello, Fred and Wilma

% java -cp jdiscript.jar:example/src/main/java/ org.jdiscript.example.HelloWorldExample
Hello, World
Hello, JDIScript!
Hello, Barney
Hello, JDIScript!
Hello, Fred and Wilma
Hello, JDIScript!
```

Java Flight Recorder

- JFR (Java Flight Recorder) is a JVM component that records events during runtime and
- allows live or post-termination analysis via Jfr, jcmd command line tools
- the larger brother of the jfr tool is the JMC (JDK Mission Control)
- Gives insights into file usage, sockets, standard crypto operations
- Behavior-wise we can roughly see what happened in a process
  - how under which circumstances (what was recorded in the fields)
  - via the stack trace

Java Flight Recorder

- There are pre-defined event but reversing may require custom event definitions
- Tracing profile determine which events are recorded or dropped
- JFR custom events allow to finetune tracing coverage
- So create a class
  - that extends `jdk.jfr.Event`
  - with fields that collect context-aware information when event is created
- To record an event it an Object is created, and then submitted
- If the event is configured to be enabled in the jfr profile the event is included in a flight recorder file, and available for later inspection

Java Flight Recorder

- Define the event

```
public class Base64Encoder extends jdk.jfr.Event{
        String srcText = "";
        long timeStamp = 0;

        Base64Encoder(String _src, long _time) {
                srcText = _src;
                timeStamp = _time;
        }

        public static String encodeBuffer(byte[] a) {

                var event = new Base64Encoder(new String(a),System.currentTimeMillis());
                event.begin();
                String res = Base64.getEncoder().encodeToString(a)+"\n";
                event.commit();

                return res;
        }
}
```

Java Flight Recorder

- Custom event definition in profile (custom.jfc)

```
<configuration version="2.0"
               label="Profiling"
               description="Low overhead configuration ... ">
    <event name="helper.Base64Encoder">
      <setting name="enabled">true</setting>
    </event>
...
</configuration>
```

```
java -XX:StartFlightRecording:filename=b64_2_recording.jfr,dumponexit=true,
                  settings=custom.jfc -cp newtrans/:. KeyGen blubi 1
[0.160s][info][jfr,startup] Started recording 1. No limit specified, using maxsize=250MB as default.
[0.160s][info][jfr,startup] Use jcmd 74860 JFR.dump name=1 to copy recording data to file.
file:class com.vhly.crackmes.cm4.KeyFile
serial:AR0B6cxY9UZvVn/v2do2FuCUWpw=

helper.Base64Encoder@5e5d171f :event written
Helper: "AR0B6cxY9UZvVn/v2do2FuCUWpw=
"
Helper: "AR0B6cxY9UZvVn/v2do2FuCUWpw=
"
true
```

Java Flight Recorder

- Custom event definition in profile (custom.jfc)

```
<configuration version="2.0"
               label="Profiling"
               description="Low overhead configuration ... ">
    <event name="helper.Base64Encoder">
      <setting name="enabled">true</setting>
    </event>
...
</configuration>
```

```
java -XX:StartFlightRecording:filename=b64_2_recording.jfr,dumponexit=true,
                  settings=custom.jfc -cp newtrans/:. KeyGen blubi 1
[0.160s][info][jfr,startup] Started recording 1. No limit specified, using maxsize=250MB as default.
[0.160s][info][jfr,startup] Use jcmd 74860 JFR.dump name=1 to copy recording data to file.
file:class com.vhly.crackmes.cm4.KeyFile
serial:AR0B6cxY9UZvVn/v2do2FuCUWpw=

helper.Base64Encoder@5e5d171f :event written
Helper: "AR0B6cxY9UZvVn/v2do2FuCUWpw=
"
Helper: "AR0B6cxY9UZvVn/v2do2FuCUWpw=
"
true
```

Java Flight Recorder

- Custom event definition in profile (custom.jfc)

```
<configuration version="2.0"
               label="Profiling"
               description="Low overhead configuration  ... ">
    <event name="helper.Base64Encoder">
      <setting name="enabled">true</setting>
    </event>
...
</configuration>
```

```
java -XX:StartFlightRecording:filename=b64_2_recording.jfr,dumponexit=true,
                  settings=custom.jfc -cp newtrans/:. KeyGen blubi 1
[0.160s][info][jfr,startup] Started recording 1. No limit specified, using maxsize=250MB as default.
[0.160s][info][jfr,startup] Use jcmd 74860 JFR.dump name=1 to copy recording data to file.
file:class com.vhly.crackmes.cm4.KeyFile
serial:AR0B6cxY9UZvVn/v2do2FuCUWpw=

helper.Base64Encoder@5e5d171f :event written
Helper: "AR0B6cxY9UZvVn/v2do2FuCUWpw=
"
Helper: "AR0B6cxY9UZvVn/v2do2FuCUWpw=
"
true
```

Java Flight Recorder

Event selection

```
%jfr print --events 'helper.*' b64_2_recording.jfr

helper.Base64Encoder {
  startTime = 21:43:39.233 (2025-02-11)
  duration = 0.405 ms
  srcText = "ʊ#=^G^H^O^M^^5`C"
  timeStamp = 1739306619233
  eventThread = "main" (javaThreadId = 1)
  stackTrace = [
    helper.Base64Encoder.encodeBuffer(byte[]) line: 23
    com.vhly.crackmes.cm4.KeyFile.isVal()
    jdk.internal.reflect.DirectMethodHandleAccessor.invoke(Object, Object[]) line: 103
    java.lang.reflect.Method.invoke(Object, Object[]) line: 580
    KeyGen.main(String[]) line: 76
  ]
}
```

Java Flight Recorder

Event selection

```
%jfr print --events 'helper.*' b64_2_recording.jfr

helper.Base64Encoder {
  startTime = 21:43:39.233 (2025-02-11)
  duration = 0.405 ms
  srcText = "ʊ#=^G^H^O^M^^5`C"
  timeStamp = 1739306619233
  eventThread = "main" (javaThreadId = 1)
  stackTrace = [
    helper.Base64Encoder.encodeBuffer(byte[]) line: 23
    com.vhly.crackmes.cm4.KeyFile.isVal()
    jdk.internal.reflect.DirectMethodHandleAccessor.invoke(Object, Object[]) line: 103
    java.lang.reflect.Method.invoke(Object, Object[]) line: 580
    KeyGen.main(String[]) line: 76
  ]
}
```

Java Flight Recorder

Event selection

```
%jfr print --events 'helper.*' b64_2_recording.jfr

helper.Base64Encoder {
  startTime = 21:43:39.233 (2025-02-11)
  duration = 0.405 ms
  srcText = "ʊ#=^G^H^O^M^^5`C"
  timeStamp = 1739306619233
  eventThread = "main" (javaThreadId = 1)
  stackTrace = [
    helper.Base64Encoder.encodeBuffer(byte[]) line: 23
    com.vhly.crackmes.cm4.KeyFile.isVal()
    jdk.internal.reflect.DirectMethodHandleAccessor.invoke(Object, Object[]) line: 103
    java.lang.reflect.Method.invoke(Object, Object[]) line: 580
    KeyGen.main(String[]) line: 76
  ]
}
```

Java Flight Recorder

Event selection

```
%jfr print --events 'helper.*' b64_2_recording.jfr

helper.Base64Encoder {
  startTime = 21:43:39.233 (2025-02-11)
  duration = 0.405 ms
  srcText = "ʊ#=^G^H^O^M^^5`C"
  timeStamp = 1739306619233
  eventThread = "main" (javaThreadId = 1)
  stackTrace = [
    helper.Base64Encoder.encodeBuffer(byte[]) line: 23
    com.vhly.crackmes.cm4.KeyFile.isVal()
    jdk.internal.reflect.DirectMethodHandleAccessor.invoke(Object, Object[]) line: 103
    java.lang.reflect.Method.invoke(Object, Object[]) line: 580
    KeyGen.main(String[]) line: 76
  ]
}
```

Java Flight Recorder

- Changes applied via bytecode transformation to inject event recordings

```
% diff KeyFile.newtrans.jdis KeyFile.jdis.orig
138c138
<           new                 class java/lang/String;
---
>           new                 class sun/misc/BASE64Encoder;
140c140
<           invokespecial       Method java/lang/String."<init>":"()V";
---
>           invokespecial       Method sun/misc/BASE64Encoder."<init>":"()V";
144,146c144
<           swap;
<           pop;
<           invokestatic        Method helper/Base64Encoder.encodeBuffer:"([B)Ljava/lang/String;";
---
>           invokevirtual       Method sun/misc/BASE64Encoder.encodeBuffer:"([B)Ljava/lang/String;";
```

Java Agents

- Binary rewriting of bytecode of class files during execution
- Using classes from the `java.lang.instrumentation` package
- Technical details:
  - Transformation of class is done with ASM or other appropriate library (on the classpath)
  - An agent is bundled in a jar file
  - Agent main class calls into the premain method,
  - which delegates the transformation to an instance of `ClassFileTransformer`
  - Specifies the agent main class in META-INF/MANIFEST.MF

```
Premain-Class: Agent
Can-Redefine-Classes: true
Can-Retransform-Classes: true
```

- Note: An Agent can be detected during runtime, see this sample on Github
  - so patch detection away first

Bytecode transformation APIs

- Definition: Java bytecode manipulation involves reading, modifying, and generating Java class files at the bytecode level.
- Purpose: Used for enhancing performance, injecting code, creating proxies, and implementing aspect-oriented programming.
- The general workflow with a bytecode transformation is often following the visitor pattern, which a behavioral design pattern. It is commonly used in bytecode manipulation to traverse and modify class structures.
- How It Works in Bytecode Manipulation:
  - The **visitor pattern** is used in libraries like ASM to traverse and modify Java bytecode.
  - Visitor Interface (e.g., ClassVisitor, MethodVisitor) is implemented to walk through bytecode elements (via the hierarchy of classes at the top, methods and instructions as leaves).
  - Each class file element "accepts" a visitor, where define which operations are performed on it

Bytecode transformation APIs : BCEL

- Overview: Developed by Apache, BCEL allows for the analysis, creation, and manipulation of Java bytecode.
- History: One of the earliest bytecode manipulation libraries, widely adopted in various projects (such as spotbugs).
- Features:
  - Provides a high-level API for bytecode analysis and modification.
  - Supports class generation and transformation.
- Limitations:
  - Slower performance compared to newer frameworks.
  - Less active maintenance in recent years.
- Note: Typical considered as **legacy**, and not the first candidate for new projects

Bytecode transformation APIs : ASM

- Overview: ASM is a low-level Java library designed for efficient bytecode manipulation. History: Introduced to provide a faster and more flexible alternative to existing libraries like BCEL.
- Features:
  - Offers both event-based (visitor pattern) and tree-based APIs.
  - Enables dynamic code generation and transformation.
  - **Widely used** in frameworks such as Hibernate and Spring.
- Advantages:
  - High performance and low memory footprint.
  - Active community and ongoing updates.

Bytecode transformation APIs : ClassFile API

- Overview: The ClassFile API is a new addition to the JDK, introduced to provide a built-in solution for bytecode manipulation.
- History: Developed to address the "chicken and egg" problem where external libraries lag behind new Java versions (see the 6mths cadence as of JEP 3 / JEP 322).
- Features:
  - Resides in the `java.lang.classfile` package. Preview feature until JDK 23.
  - Provides abstractions like elements, builders, and transforms for class file manipulation.
  - Utilizes pattern matching for parsing class files.
- Advantages:
  - **Evolves alongside the JDK, ensuring compatibility with new features.**
  - **Reduces dependencies on third-party libraries.**

Bytecode transformation: Typical operations

- Bytecode rewriting involves manipulating Java class files to modify, transform, or analyze their structure and behavior. In binary analysis, these techniques help in detecting, unpacking, and countering malicious code.
- Key Use Cases in Malware Analysis:
  - Modification – Altering malicious bytecode to neutralize threats.
  - Transformation – Obfuscation or de-obfuscation of class files.
  - Analysis – Static or dynamic examination to detect suspicious patterns

RE//verse

Bytecode rewriting / modification

- Patching / Neutralizing Malicious Behavior by removal or neutralization of unwanted functionalities.
- Examples:
  - Insert Debugging hooks, remove calls to anti-debugging code
  - Disable/Divert property lookup to infer about local runtime environment
  - Remove rogue toString methods
- Advantages:
  - By removing/replacing suspicious behavior may reduce undesired side-effects
  - Better productivity if anti-debugging code is successfully detected and removed
- Challenges (devil is in the detail):
  - Unknown binaries may check tampering, such like checking class checksums , use signed jars, etc.
  - Requires deep understanding of the binary’s control flow, state model and data structures.

-Assume everything is connected to everything, so in iterations, cut away carefully as much as possible, imagine onions

Bytecode rewriting / modification: Use case track array creation

- Use case, if we want to analyze code whether and where it creates any arrays (such as an sbox)
- Using a ASM ClassVisitor that includes a custom MethodVisitor

```
ClassVisitor classVisitor = new ClassVisitor(Opcodes.ASM9, classWriter) {
    @Override
    public MethodVisitor visitMethod(int access, String name, String descriptor, String signature,
                                     String[] exceptions) {
        MethodVisitor mv = super.visitMethod(access, name, descriptor, signature, exceptions);
        return new MethodVisitor(Opcodes.ASM9, mv) {
            @Override
            public void visitIntInsn(int opcode, int operand) {
                if (opcode == Opcodes.NEWARRAY) {             // Intercept primitive array creation
                    mv.visitInsn(Opcodes.DUP);                // Duplicate array size value
                    mv.visitIntInsn(Opcodes.SIPUSH, operand); // Push the operand (array type)
                    mv.visitMethodInsn(Opcodes.INVOKESTATIC, "Logger",
                                                             "logArrayCreation",
                                                             "(II)V", false);
                }
                super.visitIntInsn(opcode, operand);
            }
        };
    }
```

Bytecode rewriting / modification: Use case track array creation type parsing

```
public class Logger {
    public static void logArrayCreation(int size, int type) {
        String arrayType = getArrayType(type);
        System.out.println("Primitive array created: Type = " + arrayType + ", Size = " + size);
        Thread.dumpStack();
    }

    private static String getArrayType(int type) {
        switch (type) {
            case  4: return "boolean";
            case  5: return "char";
            case  6: return "float";
            case  7: return "double";
            case  8: return "byte";
            case  9: return "short";
            case 10: return "int";
            case 11: return "long";
            default: return "unknown";
        }
    }
}
```

Bytecode rewriting / modification: Now let's write a test harness that creates arrays

```java
public class TestApp {
    public static void main(String[] args) {
        System.out.println("Creating arrays...");
        int[] intArray = new int[10];  // Should log size 10
        double[] doubleArray = new double[5];  // Should log size 5
        char[] charArray = new char[15];  // Should log size 15
        System.out.println("Arrays created!");
    }
}
```

And the relevant bytecode instructions:

```
        8: bipush        10
       10: newarray      int
       12: astore_1
       13: iconst_5
       14: newarray      double
       16: astore_2
       17: bipush        15
       19: newarray      char
       21: astore_3
```

Bytecode rewriting / modification: Run the test harness

- Without agent

```
% java  TestApp
Creating arrays...
Arrays created!
```

- With agent

```
% java -javaagent:agent.jar -cp asm-9.7.jar:. TestApp
Java Agent Loaded: Intercepting array creation...
Creating arrays...
Primitive array created: Type = int, Size = 10
java.lang.Exception: Stack trace
at java.base/java.lang.Thread.dumpStack(Thread.java:2148)
at Logger.logArrayCreation(Logger.java:5)
at TestApp.main(TestApp.java:4)
Primitive array created: Type = double, Size = 5
java.lang.Exception: Stack trace
at java.base/java.lang.Thread.dumpStack(Thread.java:2148)
at Logger.logArrayCreation(Logger.java:5)
at TestApp.main(TestApp.java:5)
Primitive array created: Type = char, Size = 15
java.lang.Exception: Stack trace
at java.base/java.lang.Thread.dumpStack(Thread.java:2148)
at Logger.logArrayCreation(Logger.java:5)
at TestApp.main(TestApp.java:6)
Arrays created!
```

RE//verse

Bytecode Rewriting / Transformation

- Transforming internal structures
  - Modify class files to pre-compute values or AOT neutralize obfuscation
  - Replace dangerous APIs by safe replacements
  - Example: Detect a certain class file structure with a different one that is of more benefit during analysis

De-Obfuscation:

- Instead of accessing an obfuscated value from the constant pool, pre-decode it and place the clear text copy in the constant pool, also replace the call to the decrypter with load instruction from the constant pool
- A software could use Methodhandles and obfuscated identifier names to disguise control flow.
- After hooking the naming scheme mapping, MethodHandle calls can be replaced by unobfuscated method calls via a MethodVisitor that replaces the call instructions.

Bytecode Rewriting / Transformation

API-Replacement

- A software could uses outdated JRE system calls that prevent analysis / running in newer JREs.
- Like `sun.misc.BASE64Encoder` .
- Via a method visitor identify the API calls,
  - replace invoke instructions with those to a supported API replacement (`java.util.Base64`)
  - may have to correct the stack usage too (by correcting values order on the stack with swap, dup, etc.)
  - or even correct the return value (old API appends an odd carriage return to the method result)

```
String res = Base64.getEncoder().encodeToString(a)+"\n";
```

Replacing API calls

- Here we replace the JDK8 internal BASE64Encoder with a wrapper implemented in Helper.encodeBuffer
- Step 1: Here
  - when we encounter a INVOKEVIRTUAL with the original API call
  - it will be replaced with the helper class method,
  - and called it with INVOKESTATIC instead

```
var removeDebugInvocations = MethodTransform.transformingCode(
  (builder, element) -> {
    switch (element) {
    case InvokeInstruction i when
      i.opcode() == Opcode.INVOKEVIRTUAL
      && i.owner().asInternalName().equals("sun/misc/BASE64Encoder")
      && i.method().name().equalsString("encodeBuffer") ->
        {
          builder
            .swap() // we have the unused String on
                    // the stack and input byte array, but in wrong
                    // sequence, so swap
            .pop()  // and now pop the stand-in string away, RIP
            .invokestatic(helper, "encodeBuffer",
```

Replacing API calls (part 2)

- Step 2: when encountering the `INVOKESPECIAL` opcode we create a dummy `String` as stand-in for the `Base64Encoder`
- Note: Using the stand-in `String` allows micro-surgery and to leave the method control flow mainly as-is

```
case InvokeInstruction i when
  i.opcode() == Opcode.INVOKESPECIAL
  && i.owner().asInternalName().equals("sun/misc/BASE64Encoder")
  && i.method().name().equalsString("<init>") ->
    {
      builder
        .invokespecial(CD_String, "<init>", MethodTypeDesc
        .of(CD_void));

    }
```

Replacing API calls (part 3)

- Step 3: In order to use the stand-in String it has to be created where the Base64Encoder was created
- The `BASE64Encoder` class does not have a static method, but the new `Helper.encodeBuffer` is a static method
- so we have to take care of the the now unused constructor in the control flow, just let's repurpose it and create a temporary `String` which we can later pop away

```
  case NewObjectInstruction i when
    i.className().asInternalName()
                 .equals("sun/misc/BASE64Encoder")
      →
        {
          builder.new_(CD_String);
        }
  default → builder.accept(element)
}
```

Bytecode analysis

- Perform Bytecode structural tests
  - Reconstruct class hierarchies
  - Data flow
  - API usage (Using Serialization, Using Reflection)
  - Scan and pre-process constant pool

RE//verse

Bytecode analysis, example

- Example: Wrap constant pool strings with quote characters, to overcome the deficiency we identified earlier in `javap`

```
%javap  -c TestSpaces.class
..
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: aload_0
       5: ldc           #7                  // String
       7: putfield      #9                  // Field twospaces:Ljava/lang/String;
      10: aload_0
      11: ldc           #15                 // String
      13: putfield      #17                 // Field onespace:Ljava/lang/String;
      16: aload_0
      17: ldc           #20                 // String
      19: putfield      #22                 // Field empty:Ljava/lang/String;
      22: return
```

- With a bytecode scanner we can extract more details

```
% java --enable-preview ClassFileConstantPoolAnalyzer TestSpaces.class | grep StringEntry
7:8:1:StringEntryImpl:8 "  ”
15:8:1:StringEntryImpl:8 " ”
20:8:1:StringEntryImpl:8 ""
```

Bytecode analysis: Exposing more details about empty string

```
Path classFilePath = Path.of(args[0]);
byte[] classData = Files.readAllBytes(classFilePath);
// Parse the class file
ClassModel cm = ClassFile.of().parse(classData);
ConstantPool constantPool = cm.constantPool();

Iterator<PoolEntry> pe = constantPool.iterator()
  for (; pe.hasNext(); ) {
    PoolEntry entry = (PoolEntry) pe.next();
    String thetype = entry.getClass()
                          .toString()
                          .replaceAll(
                     "class jdk.internal.classfile.impl.AbstractPoolEntry\\$","");
    System.out.println(entry.index()+":"
                      +entry.tag()+":"+entry.width()+":"
                      +thetype+":"+entry.toString());
 };
```

RE//verse

Security Manager

- The security manager is commonly known as a relict from the (dark) Applet age
- Now on the way out of the JRE by graceful deprecation (JEP-XXX)
- Can be a starting point to inspect JVM in regards of resource usage behavior
- Access to sockets, files, envvars and certain JRE classes is restricted by permissions which in total are collected in a policy file
- A custom security manager can collect permission queries without modification of the binary under test
- the custom SM class can be specified on command line
- Detectable via
  - stack inspection,
  - class path, and
  - other side effects possible (`SecurityException`)
- There is a history of Security manager bypasses, but that would be prepared first by some attacker supplied code,
- Still be careful to watch for suspicious classes while stepping, use whitelists in `checkXXX` functions

Security Manager

- A careful researcher can start an unknown binary with a custom policy
- This allows fine-granular resource access,
- Access settings can be widened when appropriate

```
grant {
permission java.lang.RuntimePermission "createClassLoader";
permission java.lang.RuntimePermission "setContextClassLoader";
permission java.lang.RuntimePermission "exitVM.0";
permission java.io.FilePermission "./-" ,"read";
permission java.io.FilePermission "<<ALL FILES>>" ,"read";
permission java.io.FilePermission "/Users/kojak/stuff/reverse/some_dir/CM4/.", "write";
permission java.util.PropertyPermission "user.dir", "read";
permission java.lang.RuntimePermission "accessClassInPackage.sun.misc";
permission java.lang.RuntimePermission "accessClassInPackage.sun.reflect";

};
```

JVM compilation tracing

- Sometimes we don't have to reverse, and the JVM optimizes some of the obfuscation away, this may occur with TieredCompilation
- What is Tiered Compilation?
- Definition:
  - Tiered Compilation is a feature in the JVM
  - that balances startup performance and long-term optimization
  - by profiling in multiple compilation levels.
- Why It Matters:
- Faster startup times for applications.
- More efficient execution over time.
- Adaptive performance tuning based on real-time profiling.
- How It Works:
  - The JVM starts executing code in interpreted mode and
  - progressively optimizes it using Just-In-Time (JIT) compilation.

JVM tier compilation levels

Tier	Compilation Type	Notes
0	Interpreter	code runs without compilation, slow but immediate execution
1	Simple JIT (C1)	Lightweight compilation for faster startup, fewer optimizations.
2	Limited Profiling (C1)	Adds profiling to identify hot methods.
3	Full Profiling (C1)	Collection of more profiling data, helps transition to C2.
4	Optimizing JIT (C2)	Full optimization with aggressive performance tuning.

- JIT compilation can be logged
  - which can reveal details about the control flow and optimization
  - To display platform assembly code it requires additional library from OpenJDK repo
    - look for hsdis-<arch>.{dylib|so|dll}
    - Visualizes tiered compilation decisions.
    - Displays hot methods and inlining information.
    - Helps diagnose performance bottlenecks.

Is JIT Optimizing MBAs away?

- Analysis

```
% java  -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly -Xcomp -XX:-TieredCompilation
    -XX:-DontCompileHugeMethods -XX:CompileOnly=MBASimpleJava.doit  MBASimpleJava  1  4

[Verified Entry Point]
  # {method} {0x0000000120400398} 'doit' '(II)Z' in 'MBASimpleJava'
  # parm0:    c_rarg1   = int
  # parm1:    c_rarg2   = int
  #           [sp+0x20]  (sp of caller)
  0x000000010f86d080:   nop
  0x000000010f86d084:   sub sp, sp, #0x20
  0x000000010f86d088:   stp x29, x30, [sp, #16]
  0x000000010f86d08c:   ldr w8, 0x000000010f86d0f8
  0x000000010f86d090:   ldr w9, [x28, #32]
  0x000000010f86d094:   cmp x8, x9
  0x000000010f86d098:   b.ne 0x000000010f86d0e4  // b.any;*synchronization entry
```

Is JIT Optimizing MBAs away?

```
                                           ; - MBASimpleJava::doit@-1 (line 3)
  0x000000010f86d09c:   and w10, w1, w2
  0x000000010f86d0a0:   orr w11, w1, w2
  0x000000010f86d0a4:   eor w13, w1, w2
  0x000000010f86d0a8:   sub w10, w11, w10
  0x000000010f86d0ac:   cmp w10, w13
0x000000010f86d0b0:   b.eq 0x000000010f86d0d0  // b.none;*if_icmpne
                                                  {reexecute=0 rethrow=0 return_oop=0}
                                               ; - MBASimpleJava::doit@14 (line 5)
  0x000000010f86d0b4:   mov w0, wz               ;*ireturn {reexecute=0 rethrow=0 return_oop=0}
                                               ; - MBASimpleJava::doit@22 (line 5)
  0x000000010f86d0b8:   ldp x29, x30, [sp, #16]
  0x000000010f86d0bc:   add sp, sp, #0x20
  0x000000010f86d0c0:   ldr x8, [x28, #1096]     ;   {poll_return}
  0x000000010f86d0c4:   cmp sp, x8
  0x000000010f86d0c8:   b.hi 0x000000010f86d0d8  // b.pmore
  0x000000010f86d0cc:   ret
```

The JVM could do better, let’s compare to clang AOT

If the JVM does not remove complexity of expressions, AOT compilation may still help. Let’s try clang with the translated MBA sample:

```
#include <stdio.h>
#include <stdlib.h>

//public class MBASimpleJava{
//      public static boolean doit(int x , int y ) {
        int doit(int x, int y) {
                int lhs = x ^ y ;
                int rhs = (x | y) - (x & y) ;
                return rhs == lhs;
        }
        int main (int c, char** arg) {
//      public static void main(String[] arg) {
                int x = atoi(arg[0]);
                int y = atoi(arg[1]);    // forgive me
                int z = doit(x,y);
                printf("%d",z);

        }
//}
```

The JVM could do better, let's compare to clang AOT

- AOT compilation with clang detects the equality of terms and with -O3 it replaces it with a constant 1

```
% gcc -O3 -o mba_o3 MBASimple.c
% otool -tv mba_o3
mba_o3:
(__TEXT,__text) section
_doit:
0000000100003f58 mov w0, #0×1
0000000100003f5c ret
_main:
0000000100003f60 sub sp, sp, #0×20
0000000100003f64 stp x29, x30, [sp, #0×10]
0000000100003f68 add x29, sp, #0×10
0000000100003f6c mov w8, #0×1
0000000100003f70 str x8, [sp]
0000000100003f74 adrp x0, 0 ; 0×100003000
0000000100003f78 add x0, x0, #0×f9c ; literal pool for: "%d”
0000000100003f7c bl 0×100003f90 ; symbol stub for: _printf
0000000100003f80 mov w0, #0×0
0000000100003f84 ldp x29, x30, [sp, #0×10]
0000000100003f88 add sp, sp, #0×20
0000000100003f8c ret
```

The JVM could do better, let's compare to clang AOT

- AOT compilation with clang detects the equality of terms and with -O3 it replaces it with a constant 1

```
% gcc -O3 -o mba_o3 MBASimple.c
% otool -tv mba_o3
mba_o3:
(__TEXT,__text) section
_doit:
0000000100003f58 mov w0, #0×1
0000000100003f5c ret
_main:
0000000100003f60 sub sp, sp, #0×20
0000000100003f64 stp x29, x30, [sp, #0×10]
0000000100003f68 add x29, sp, #0×10
0000000100003f6c mov w8, #0×1
0000000100003f70 str x8, [sp]
0000000100003f74 adrp x0, 0 ; 0×100003000
0000000100003f78 add x0, x0, #0×f9c ; literal pool for: "%d"
0000000100003f7c bl 0×100003f90 ; symbol stub for: _printf
0000000100003f80 mov w0, #0×0
0000000100003f84 ldp x29, x30, [sp, #0×10]
0000000100003f88 add sp, sp, #0×20
0000000100003f8c ret
```

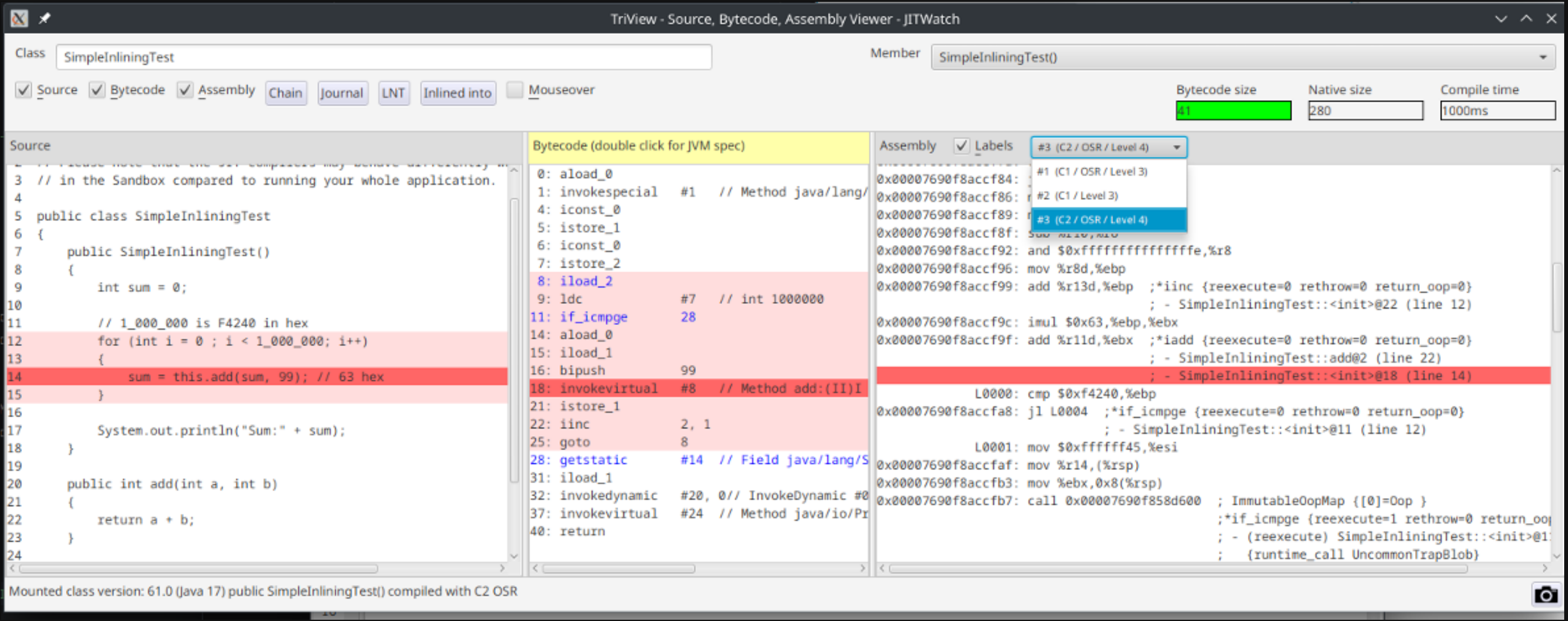
The JVM could do better, let’s compare to clang AOT

- AOT compilation with clang detects the equality of terms and with -O3 it replaces it with a constant 1

```
% gcc -O3 -o mba_o3 MBASimple.c
% otool -tv mba_o3
mba_o3:
(__TEXT,__text) section
_doit:
0000000100003f58 mov w0, #0x1
0000000100003f5c ret
_main:
0000000100003f60 sub sp, sp, #0x20
0000000100003f64 stp x29, x30, [sp, #0x10]
0000000100003f68 add x29, sp, #0x10
0000000100003f6c mov w8, #0x1
0000000100003f70 str x8, [sp]
0000000100003f74 adrp x0, 0 ; 0x100003000
0000000100003f78 add x0, x0, #0xf9c ; literal pool for: "%d"
0000000100003f7c bl 0x100003f90 ; symbol stub for: _printf
0000000100003f80 mov w0, #0x0
0000000100003f84 ldp x29, x30, [sp, #0x10]
0000000100003f88 add sp, sp, #0x20
0000000100003f8c ret
```

Tracing compilation with JIT-Watch

There is a tool that allows to trace JIT compilation results side-by-side with the bytecode and the source code



Bytecode size	Native size	Compile time
41	280	1000ms

Source

```
// in the Sandbox compared to running your whole application.

public class SimpleInliningTest
{
    public SimpleInliningTest()
    {
        int sum = 0;

        // 1_000_000 is F4240 in hex
        for (int i = 0 ; i < 1_000_000; i++)
        {
            sum = this.add(sum, 99); // 63 hex
        }

        System.out.println("Sum:" + sum);
    }

    public int add(int a, int b)
    {
        return a + b;
    }
```

Bytecode (double click for JVM spec)

```
 0: aload_0
 1: invokespecial   #1    // Method java/lang/
 4: iconst_0
 5: istore_1
 6: iconst_0
 7: istore_2
 8: iload_2
 9: ldc             #7    // int 1000000
11: if_icmpge       28
14: aload_0
15: iload_1
16: bipush          99
18: invokevirtual   #8    // Method add:(II)I
21: istore_1
22: iinc            2, 1
25: goto            8
28: getstatic       #14   // Field java/lang/S
31: iload_1
32: invokedynamic   #20, 0// InvokeDynamic #0
37: invokevirtual   #24   // Method java/io/Pr
40: return
```

Assembly

```
0x00007690f8accf84:
0x00007690f8accf86:
0x00007690f8accf89:
0x00007690f8accf8f:
0x00007690f8accf92: and $0xfffffffffffffffe,%r8
0x00007690f8accf96: mov %r8d,%ebp
0x00007690f8accf99: add %r13d,%ebp  ;*iinc {reexecute=0 rethrow=0 return_oop=0}
                                    ; - SimpleInliningTest::<init>@22 (line 12)
0x00007690f8accf9c: imul $0x63,%ebp,%ebx
0x00007690f8accf9f: add %r11d,%ebx  ;*iadd {reexecute=0 rethrow=0 return_oop=0}
                                    ; - SimpleInliningTest::add@2 (line 22)
                                    ; - SimpleInliningTest::<init>@18 (line 14)
             L0000: cmp $0xf4240,%ebp
0x00007690f8accfa8: jl L0004  ;*if_icmpge {reexecute=0 rethrow=0 return_oop=0}
                              ; - SimpleInliningTest::<init>@11 (line 12)
             L0001: mov $0xffffff45,%esi
0x00007690f8accfaf: mov %r14,(%rsp)
0x00007690f8accfb3: mov %ebx,0x8(%rsp)
0x00007690f8accfb7: call 0x00007690f858d600  ; ImmutableOopMap {[0]=Oop }
                                             ;*if_icmpge {reexecute=1 rethrow=0 return_oo
                                             ; - (reexecute) SimpleInliningTest::<init>@1
                                             ;   {runtime_call UncommonTrapBlob}
```

Mounted class version: 61.0 (Java 17) public SimpleInliningTest() compiled with C2 OSR

Tracing JIT-Behavior

- Tracing JIT behavior can illustrate how the optimization is applied
- Reducing complex expressions is cost intensive and may be in a goal conflict with fast execution times
- Binaries can take benefit from default JVM tuning parameters, if the method code sizee exceeds a HugeMethodLimit (8000) of bytecode instructions, no JIT compilation is applied
  - For longer methods the JVM needs to be instructed to disable large method capping with the `-XX:-DontCompileHugeMethods` VM option
- In beforehand it should be checked if a binary prevents use of JIT processing by applying relative timing checks

RE//verse

jshell

- Jshell is REPL to explore code snippets in Java (similar exists for Groovy, Kotlin)
- Great to analyze data that is created alongside control flow (such as from a heap dump), often can be pasted directly from the decompiler output without the need to compile or execute dangerous followup code:
- Can be also be run in the security manager sandbox, so escapes are potentially detectable
- Jshell does not only accept console line input, but also runs script files for better maintainability and reuse of artifacts
- Could be more complex than the following examples, chained arithmetics to disguise the payload, so expect a step-wise process

Using jshell with a decompiled snippet from a binary found on pastebin

```
int[] ty = { 104, 116, 116, 112, 58, 47, 47, 100, 108, 46, 100, 114,
 111, 112, 98, 111 , ... };
ty ⟹ int[66] { 104, 116, 116, 112, 58, 47, 47, 100, 10 ... 16, 63, 100, 108, 61, 49 }
> new byte[66]
$3 ⟹ byte[66] { 0, 0,   ... }
> var i=0; for (i = 0 ; i < 66 ; i++) { $3[i]=((byte)ty[i]);}
i ⟹ 0
> $3
$3 ⟹ byte[66] { 104, 116, .. , 49 }
> new String($3)
$10 ⟹ "http://dl.dropboxusercontent.com/s/n***b/module.dat?dl=1"
```

Optionally Jshell can integrate with javap and other JDK tools

- Let’s Take a look at bytecode from an anti-agent class

```
jshell> private static final byte[] EMPTY_CLASS_BYTES = {
   ...>    -54, -2, -70, -66, 0, 0, 0, 49, 0, 5, 1, 0, 34, 115, 117, 110,
   ...>     47, 105, 110, 115, 116, 114, 117, 109, 101, 110, 116, 47, 73,
   ...>    110, 115, 116, 114, 117, 109, 101, 110, 116, 97, 116, 105, 111,
   ...>    110, 73, 109, 112, 108, 7, 0, 1, 1, 0, 16, 106, 97, 118, 97, 47,
   ...>    108, 97, 110, 103, 47, 79, 98, 106, 101, 99, 116, 7, 0, 3, 0, 1,
   ...>      0, 2, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0
   ...>     };

jshell> FileOutputStream fos = new FileOutputStream("anti_agent.class");
Jshell> fos.write(EMPTY_CLASS_BYTES)
jshell> fos.close();
```

Example: Jshell to integrate with javap

- and paste it into javap using the utility code from https://github.com/sormuras/jdk-tools/

```
jshell> /open TOOLING.jsh
        // from https://github.com/sormuras/jdk-tools ,
        // looks legit but may want to verify downloads before usage
jshell> javap("-c","-l","-v","-p","anti_agent.class")
Classfile /Users/ms1969/anti_agent.class
  Last modified Feb 18, 2025; size 86 bytes
  SHA-256 checksum 032dfff269b03eb6d86bd8f9ff8060aaaf1ee149329f827b79933d02b55da655
public class sun.instrument.InstrumentationImpl
  minor version: 0
  major version: 49
  flags: (0x0001) ACC_PUBLIC
  this_class: #2                          // sun/instrument/InstrumentationImpl
  super_class: #4                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 0, attributes: 0
Constant pool:
  #1 = Utf8               sun/instrument/InstrumentationImpl
  #2 = Class              #1              // sun/instrument/InstrumentationImpl
  #3 = Utf8               java/lang/Object
  #4 = Class              #3              // java/lang/Object
{
}
```

Dynamic Analysis: Finding Traces in the heap

- Sometimes secrets cannot be directly be inferred by code analysis, but by following up traces in (heap) memory
- These dumps are particularly useful for debugging performance issues, troubleshooting application crashes,
  - and understanding the runtime behavior of large-scale applications
- Tools like VisualVM and jcmd are commonly used to analyze heap dumps and extract meaningful data that may give hints about the internal state.
- For deeper forensic insight,
  - make sure to run a complete heap dump (not just live objects),
  - and tune your GC retain as much data as possible (larger RAM, lower GC frequencies)
  - create the heap dump, after the action that created secret data has happened

Dynamic Analysis: Finding Traces in the heap

- This code snippets shows how heap dump can be forced after program termination with a ShutDownHook
- The example here is a wrapper around a KeyGen that was written for a crackme
- That wrapper install this ShutDownHook, then runs the KeyGen, and shuts the JVM down
- After shutdown a heap dump is written to the file system which can be analyzed with heap inspection tools like VisualVM

Dynamic Analysis: Finding Traces in the heap

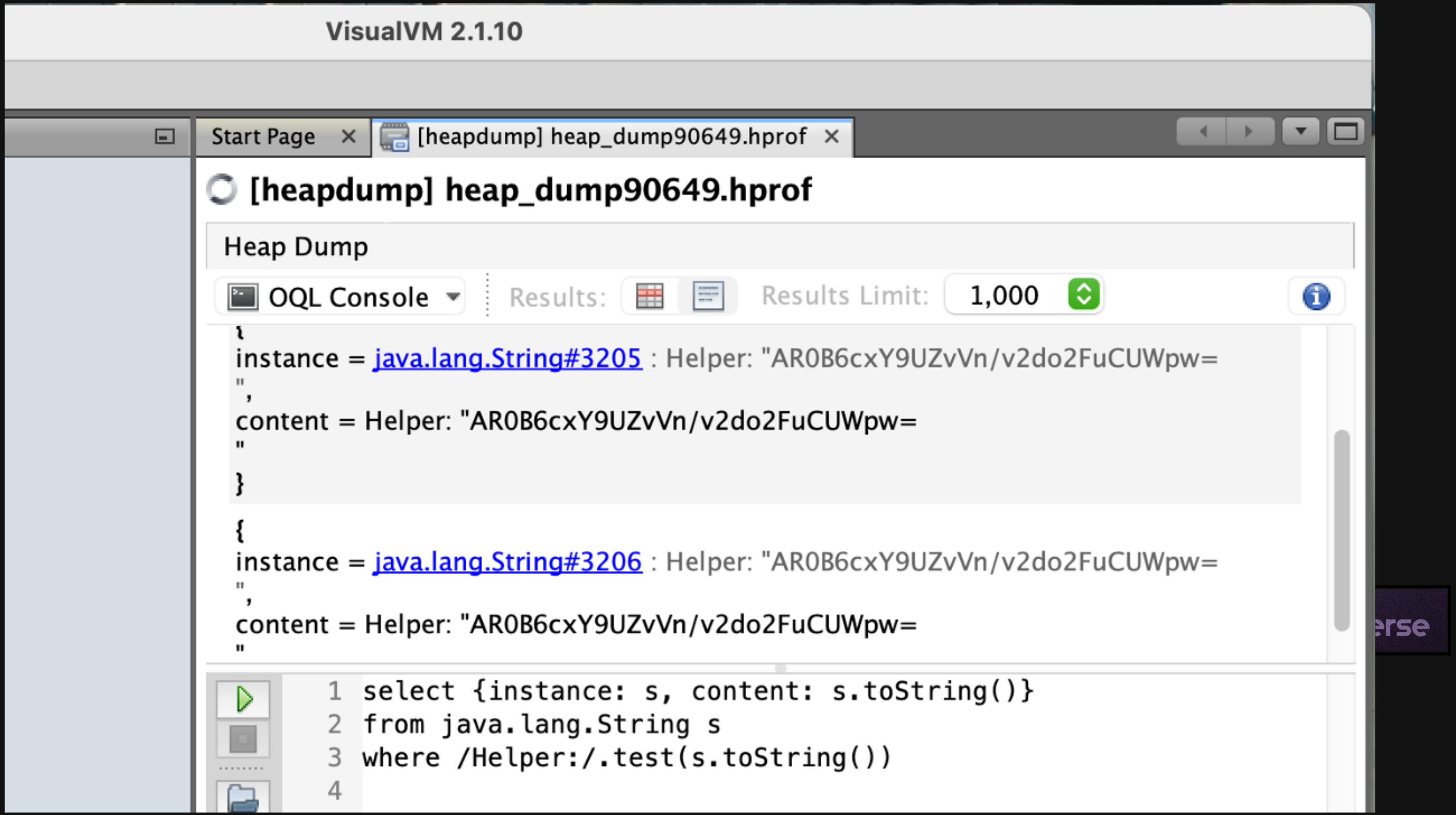
```java
import java.lang.management.ManagementFactory;
import java.lang.management.RuntimeMXBean;

public class LaunchShutDownHookHeap {

    static void setShutDownHook() {
            Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            try {
                String pid = ManagementFactory.getRuntimeMXBean().getName().split("@")[0];
                System.out.println("Generating heap dump before exit...");
                Process process = new ProcessBuilder("jcmd", pid,  "GC.heap_dump", "-all", "heap_dump"+pid+".hprof")
                        .inheritIO()
                        .start();
                process.waitFor();
                System.out.println("Heap dump saved as heap_dump"+pid+".hprof");
            } catch (IOException | InterruptedException e) {
                e.printStackTrace();
            }
        }));
    };
```

Dynamic Analysis: Finding Traces in the heap

- The written heap dump can now be inspected in the OQL console of VisualVM.
- Let’s assume the secret we are searching is prefixed with the string value "Helper:"
- This can then be used in regular expression filter like

```
select {instance: s , content: s.toString()}
from java.lang.String s
where /Helper:/.test(content):
```

Dynamic Analysis: Finding Traces in the heap



```
{
instance = java.lang.String#3205 : Helper: "AR0B6cxY9UZvVn/v2do2FuCUWpw=
",
content = Helper: "AR0B6cxY9UZvVn/v2do2FuCUWpw=
"
}

{
instance = java.lang.String#3206 : Helper: "AR0B6cxY9UZvVn/v2do2FuCUWpw=
",
content = Helper: "AR0B6cxY9UZvVn/v2do2FuCUWpw=
"
```

```
select {instance: s, content: s.toString()}
from java.lang.String s
where /Helper:/.test(s.toString())
```

Finale

RE//verse

Final thoughts

- Decompilers have their blind spots
  - and may not show the entire picture of the code under test
  - can fail in unexpected situations and for edge-case artifact variants
- Verifiability class files have an asserted set of metadata that may be repurposed
  - Bytecode techniques can recover debug-information from meta-data already available in class files
  - Existing Java Development tools can be utilized for reverse engineering purposes using synthesized metadata
    - Regenerated LineNumberTable data for instruction level stepping in jdb
    - Regenerated LocalVariableTable data to allow data-based tracing during execution

RE//verse

Moving forward

- The lost source code problem won't go away, recovery skills are essential
- Java no longer the only language on the JVM playing field (Kotlin, Scala,…)
- Ramp up on Java Bytecode,
  - read code, play with asmtools
  - avoid decompilers for a while, instead read souce and bytecode side-by-side
  - ASMifier can help to get started
- Get familiar with the usual suspect classes to hook (String, Integer, StringBuilder et al.)
- The code used in this presentation and more will be shared on Github, too
- Just follow me on LinkedIn for updates

Q & A