Mobile Code Security by Java Bytecode Instrumentation

Ajay Chander, Stanford University
John C. Mitchell, Stanford University
Insik Shin, University of Pennsylvania

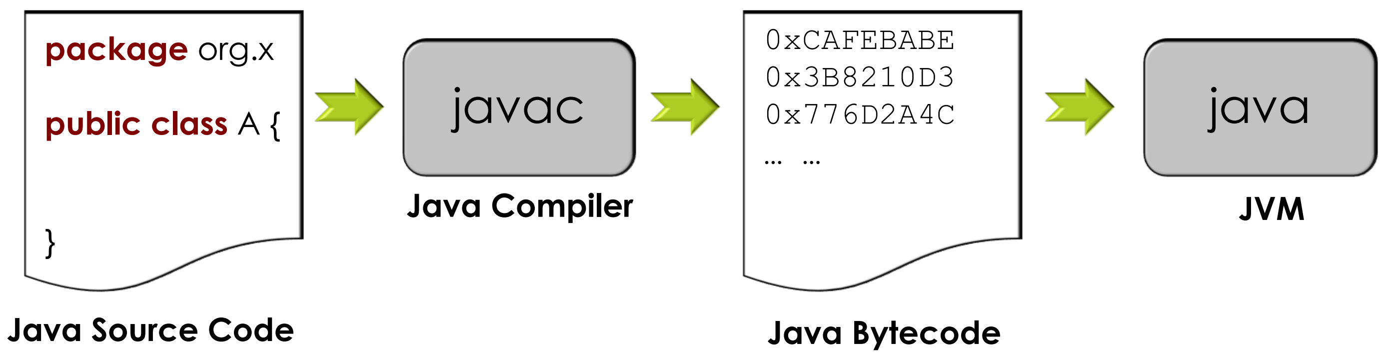
Slides and presentation by Ming Zhou

Binary-rewriting-based SFI

- Transform a program to meet safety properties.
- Several aspects
  - The form of input: compiled code (binary code on native machine, ELF)
  - The goal of transformation
    - Fine-grained: micromanaging behavior of program in hosted environment (CFI)
    - Coarse-grained: preventing program from abusing system resources (REINS)
  - Timing for transforming
    - Compile time
    - Loading time
    - Runtime

Java and bytecode

- What is bytecode?
  - The target code to be run on Java Virtual Machine (JVM)
  - Compiled from Java code



  - In recent years, new compilers emerged to compile various source code into bytecode

Applying SFI on bytecode

- Three aspects revisited
  - The form of input: bytecode (class)
  - The goal of transformation
    - Finer-grained goal is totally handled by JVM, which is a sandbox itself. The bytecode itself is not able to get access to memory area not managed by JVM.
    - Coarse-grained: preventing program from abusing system resources. This is partially handled by JVM through security manager though.
  - Timing for transforming
    - Loading time or download time
      - The bytecode contains voluminous and well-formatted information
      - we need to cater to portable code
- We will talk about these 3 aspects in more detail later

JVM overview: Class File

- A class file
  - is the basic unit of binary code, result of compiling a Java class from the source file
  - has a well-defined format
- Example

```
package pkg;

public class A extends B
implements I
{
    private int i = 19;
    public int increment(){
        return ++i;
    }
}
```

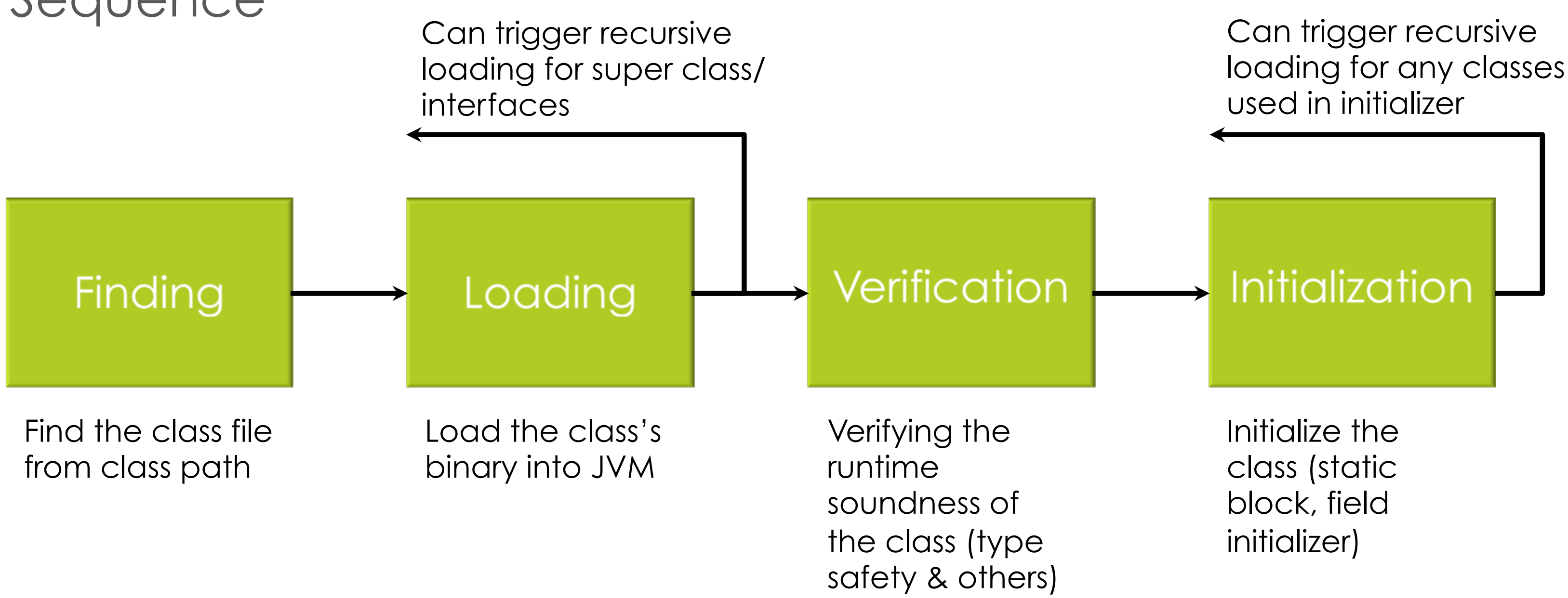
Field	Length	Description
Magic	Fixed	CAFEBABE
Version	Fixed	
Constants Pool (CP)	Varied	All the constants used
Access Flags	Fixed	**public**
This Class	Fixed	CP index (“**pkg/A**”)
Super Class	Fixed	CP index (“**pkg/B**”)
Interfaces	Varied	CP indices (“**pkg/I**”)
Fields	Varied	Field’s name, type, access
Methods	Varied	Method’s name, type, access, code, exceptions

JVM overview: Memory layout

- Standard stack-and-heap model
  - Stack
    Each thread has its own stack, which is composed of frames during runtime, with the topmost frame corresponding to the current running method.
    - Frame: consists of operand stack and local variables, the size of both predetermined by Java compiler and allocated in runtime by JVM when a method is called. Unlike register machines such as x86 and MIPS, JVM is a *stack machine*.
  - Heap
    - *Data Area*: variable throughout runtime, such as instances of classes.
    - *Method Area*: invariants throughout runtime, such as class information and bytecode in methods

JVM overview: Class loading

- Timing
  - The system/runtime classes are pre-loaded during startup
  - The class with entrance method (**main**) is always first loaded out of all the classes from the application
  - Later, when a class is first used in bytecode, it's loaded
- Sequence



Security in JVM: class verification

- Purpose of verification
  - Prevent JVM from running illegal bytecode or winding up an undefined state, and ensuring type/generic safety during runtime.
  - A class coming from standard-compliant compiler should be always legal. The verification is targeted at:
    - Class file with wrong format due to compiler/generator bugs
    - Class file tampered intentionally
- Example: verifying the compatibility of operands on the operand stack at any moment
  - Build the control flow of method based on basic block (BB)
  - At the entrance of each BB, calculate the number and type of operands for each connecting edge
  - Check if all the edges share the compatible operands

Security in JVM: Security Manager

- Portability of classes
  The machine-independent nature of Java class guarantees its great portability.
- Frameworks that leverage portability
  - Applet: browser-hosted rich client platform
  - Apache River: dynamic service and lookup
- Security concerns
  - Classes coming from network is untrusted
  - Verification is only concerned with class runnability
  - We want to prevent environment from being abused by malicious classes
  - Thus Java introduced *Security Manager*

Security Manager

- A runtime manager that applies permission check on various "system calls" invoked by application.
- The manager reads policy settings from a local protected file, or constructs policy settings during runtime.
- Example: System.exit(int)

```
package java.lang;

public final class System
{
    public static void exit(int status){
        SecurityManager manager =
        System.getSecurityManager();
        if(manager != null){
            manager.checkExit(status);
        }
        exitInternal(status);
    }
}
```

```
grant codeBase
"www.abc.com/"
{
  permission
  RuntimePermission
  exitVM;
}
```

A policy file that allows system exit.

Security Manager (cont.)

- Default setting
  - For local application, disabled by default
  - For network application (Applet), enabled by default
- Limitations
  - Grant permission based on principal of Applet. The user has to trust the party who provides the application at the first
  - Security issue of high-level semantic is not handled
    - Granting network permission for an app also enables a channel for information leakage
    - Granting AWT permission for an app also enables it to take control of the entire browser(or, tab) display
- Solution: the approach talked in this paper

New Threat Model to JVM

- High-level semantic threats
  - Denial of Service
    By opening large number of windows in AWT,
    running out of underlying resources (note AWT window is a thin wrapper of system-based GUI component)
  - Information Leak
    Given the privilege of socket communication, sending out sensitive information to a remote server
    (The other example in the paper of forging mail is unlikely since the policy file supports setting range of ports to be used)
  - Spoofing
    Displaying a URL that seems safe, but link to another hostile site under the hood

The solution to threats of these kinds

- Add another layer of protection using a combination of
  - Safer classes instead of original foundation classes
  - Bytecode instrumentation at loading

Layer	Mechanism	Supported by	Concerned with
0	Class verification	JVM	Type and state safety
1	Security Manager	JVM	Hosting environment
2	Preloading instrumentation	External filter	Hosting environment

* *(Bytecode) instrumentation* is binary rewriting by another name, which is widely used in Java community.

Background knowledge for bytecode instrumentation: Constant Pool

- A structured collection of various constants that are used in the class
  Note here the word *constant* means not only the literal value found in the class, such as a string or (big) integer, but also the name, type descriptor, generic signatures of class, interface, fields and methods. In some sense, CP is like a combination of (read only) data section and symbol table in ELF file.

- Entries of CP

	type									
Entry Type	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	***byte***
CONSTANT_Utf8	1	length		UTF-8 encoded String						
CONSTANT_Integer	3	value								
CONSTANT_Class	7	CP[1]								
CONSTANT_String	8	CP[1]								
CONSTANT_Fieldref	9	CP[7]		CP[12]						
CONSTANT_Methodref	10	CP[7]		CP[12]						
CONSTANT_NameAndType	12	CP[1]		CP[1]						

Not used

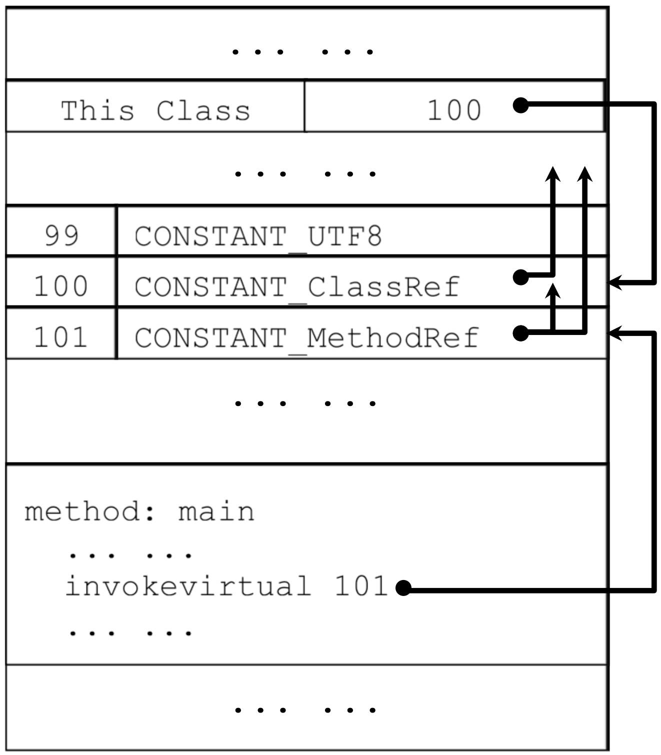
An index to CP entry of type 1 (UTF8)

The string is a type descriptor

Background knowledge for bytecode instrumentation: Constant Pool (cont.)

- Referring to CP entries in class file
  - The name, descriptor and signature of class, super class, interfaces, fields and methods (including the class initializer)
  - To refer to any class, field and method in bytecode, use the corresponding types of reference entry in CP
  - Example:

```java
package pkg;

public class A extends B {
    private int i = 19;
    public int increment(){
        return ++i;
    }

    public static void
    main(String[] args){
        increment();
    }
}
```



Class-level modification

- Supporting classes

  The safer version of the original extensible class. Implements semantic-level check and constraints and is a subclass of the original.

  Example:

  `java.awt.Window` → `Safe$Window (extends java.awt.Window)`

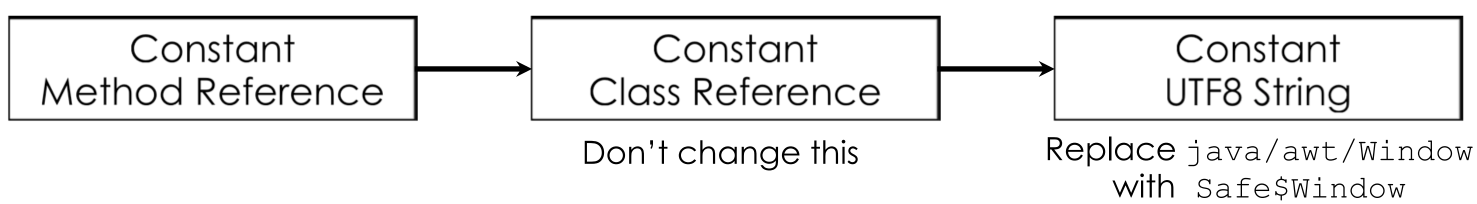
  Notes:

  (1) `$` is a legal symbol to be used in Java identifier, like "_";
  Conventionally, it's reserved for synthetic/generated name

  (2) For all the original class `C`, replace with another named `Safe$Window` with default package (no package)

- Strategy

  Keep all the class references unchanged, only modify the string which is referred to by class references.



NOTE: `java/awt/Window` is the internal notation of `java.awt.Window`

Background knowledge for bytecode instrumentation: Descriptor

- Descriptor is the internal notation of type information

  This corresponds to what we call the method signature in Java language; however, in bytecode, the term *signature* has different meaning (used to describe generic declaration).

- Notation
  - Basic type: a letter in upper case (8+1 in total)
    byte (B), boolean (Z), int (I), …, void (V)
  - Class type: `L<classname>;`, where `<classname>` is the full class name where"." is replaced with "/"
  - Array type: one additional "[" for each dimension
  - Method: `(<Type>)Type`

- Example

```
void setPriority(Thread t, int i)
→
(Ljava/lang/Thread;I)V
```

Background knowledge for bytecode instrumentation: Method invocation

- Bytecode sequence
  - Instance method
    1. Load reference to current object into operand stack
    2. Load arguments into the operand stack
    3. Invoke the method with given type
  - Class method
    1. Load arguments into the operand stack
    2. Invoke the method statically

- Invocation type
  - *Invoke virtual*: invoke the method declare in class or parent class virtually
  - *Invoke interface*: invoke the method declared in interface virtually
  - *Invoke special*: invoke the method concretely
  - *Invoke static*: invoke class method

Background knowledge for bytecode instrumentation: Operand Stack

- Stack-based machine
  Instead of registers, JVM uses a single operand stack as intermediate storage of operands.

- Operations on operand stack
  - load: load a variable into stack from *local variable table* (the collection of temporary variables used in a frame) or constant pool.
  - store: pop an operand from stack and save it to local variable table at certain location.
  - arithmetic (add, mul, and): pop a fixed number of operands and do the math, then push the result back to stack
  - invokexxx: pop a number of operands, where the number is decided by the descriptor of method, call the method and push the result back to stack.

Method-level modification

- Supporting classes
  The safer version of original class. Implements semantic-level check and constraints. It is NOT a subclass of the original, but it dispatches the call to the original eventually.

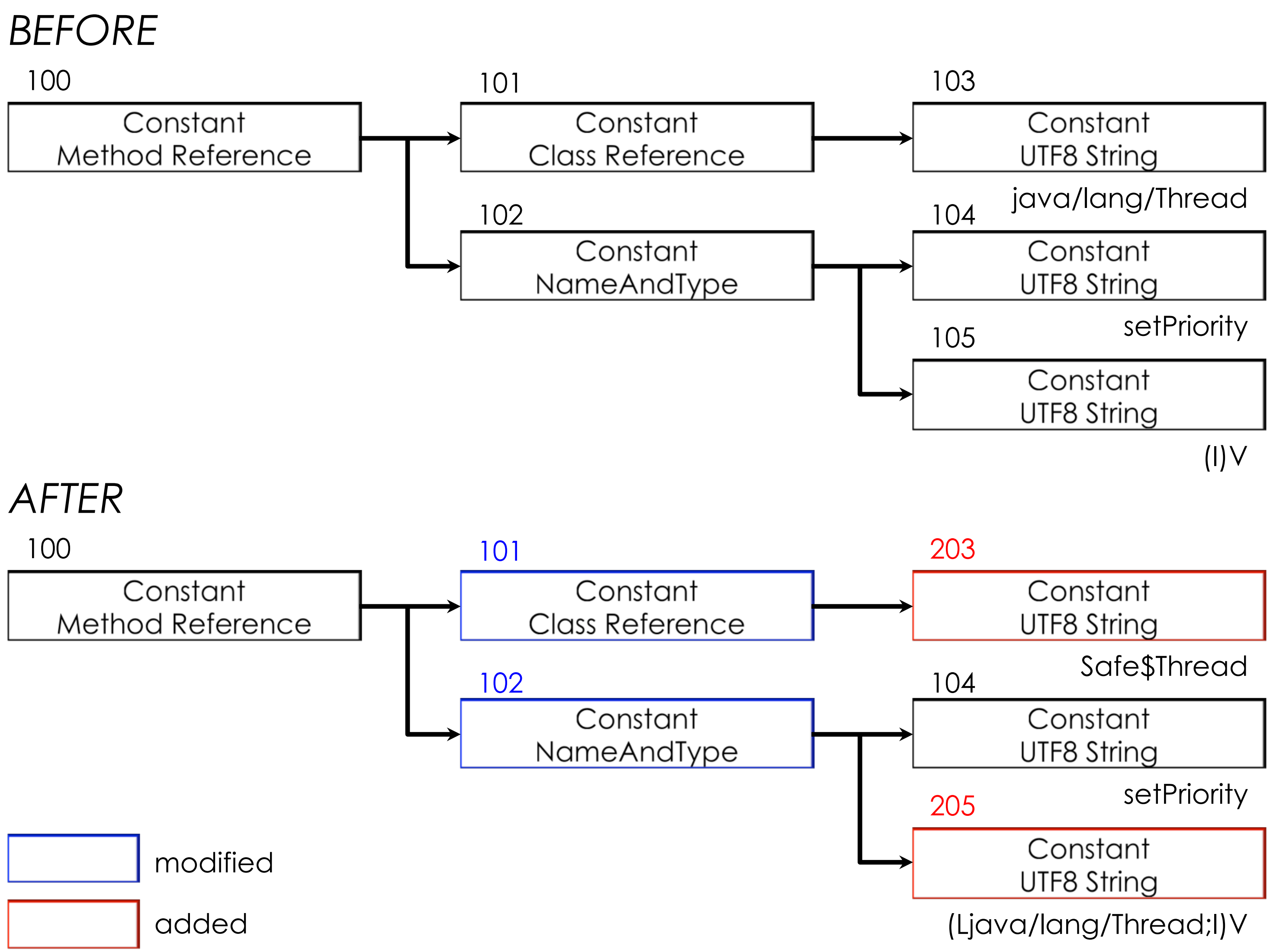
  Example:

  `java.lang.Thread` → `Safe$Thread`

- Why use method-level modification?
  - The original class is not extensible (decorated with **final**)
  - The method concerning us is not virtual
  - The safer method needs to have a different argument list

- Strategy
  - Add new CP entry for the safer class and safer method's descriptor
  - In CP entry of method reference, modify the references to class and descriptor.
  - May need to change bytecode leading up to invocation (but try to not change the max depth of operand stack)

Method-level modification: Constant Pool



Method-level modification: Bytecode

Bytecode	Comments	Java code
(BEFORE)		
`aload_1`	`Push reference to t as an implicit arg`	`Thread t = new Thread();`
`iload_2`	`Push local variable i (1st declared arg)`	`... ...`
`invokevirtual #100`		`t.setPriority(i);`
(AFTER)		*(Hypothetical)*
`aload_1`	`Push reference to t (1st declared arg)`	`Thread t = new Thread();`
`iload_2`	`Push local variable i (2nd declared arg)`	`... ...`
`invokestatic #100`		`Safe$Thread. setPriority(t, i);`

NOTE:

invokevirtual pops operands from stack equal to argument number + 1;
invokestatic pops operands from stack equal to argument number.
This modification doesn't change the maximum depth of operand stack.

When to instrument?

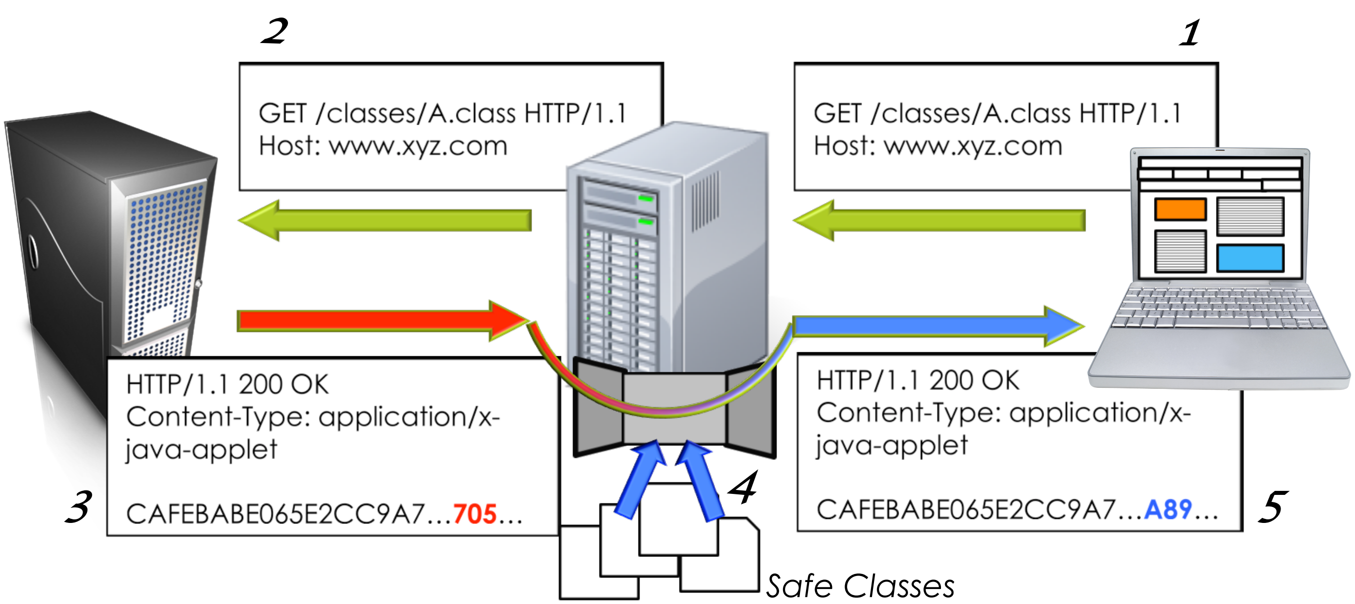
- Class loading
  - Java class ClassLoader uses method `defineClass(String name, byte[] bytecode, int offset, int length)` to load a class into JVM.
  - ClassLoader also allows user to override its core method `findClass(String name)`
  - Therefore we can create a new ClassLoader with following logic added into findClass:

```
public class FilteredClassLoader extends ClassLoader {

    protected Class findClass(String name){
        byte[] bytecode = load byte code from remote server:
        bytecode = instrument(bytecode);
        defineClass(name, bytecode, 0, bytecode.length);
    }

}
```

- Not used in this paper
  - Additional java code to be installed in browser
  - Since working as a customized part of class loading procedure in JVM, may lack flexibility

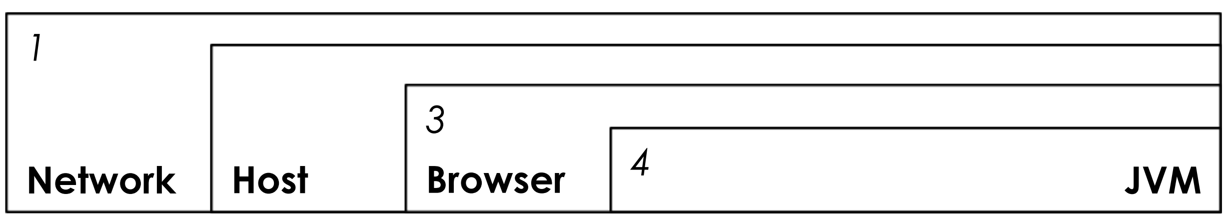
When to instrument? (cont.)

- Network proxy
  - Browser sends out HTTP request with MIME type = "application/x-java-applet"
  - We can always set up a proxy server at the front of protected network
  - Thus the proxy server can detect Applet transmission and interfere accordingly



A comparison

Type	Location	Timing	Pros	Cons
✔ 1	Proxy Server	Transmission	• Easy to implement • Quick prototyping	• Cannot be adopted by users
3	Browser	Pre-rendering	• Adoptable by users • Easier to configure (disable)	• Redundant development of multiple browsers
4	JVM	Class loading	• Adoptable by users • Hard to bypass	• Complex implementation • Need modify standard platform (JVM)



Thank You

QUESTIONS?