**Proposed OOPSLA 2012 Paper**

The main contributions of this paper are:

* it examines the shortcomings of analyzing program executions via logged raw machine state such as heap dumps
  + **Unfamiliar:** The abstractions provided by application code and the programming language itself are not available, forcing the programmer to understand and work with low-level machine concepts
  + **Complex**: Analysis code must be written at the meta-programming level, which is bulky and time-consuming to write
  + **Brittle:** Analysis easy breaks, or worse produces incorrect results, if applied to a heap dump produced by an execution of newer code
  + **Type-unsafe:** a type mismatch is not detected until internal state mismatches, if at all
  + **Insecure:** Analysis code and tools can easily inspect private data, even if accidentally, since state encapsulation is lost
* it demonstrates that the use of object holograms improves the accuracy, readability, simplicity and maintainability of remote or post-hoc execution debugging and analysis
  + **Familiar:** Analysis code can use the same APIs and data structure as the original program code
  + **Simple:** Analysis code becomes an order of magnitude smaller
  + **Robust:** The same analysis code works correctly against multiple versions of the source code iff the relevant APIs do not change
  + **Type-safe**: Analysis is compiled against the original data structure definitions and hence mismatches are caught at compile time
  + **Secure:** Code-level access ensures proper encapsulation; the same mechanisms for overriding access control through reflection are available, but require explicit use and are hence much easier to control
* it demonstrates that mirror-based virtual objects, generally studied in the context of dynamic languages, can be efficiently implemented in the statically-typed, pre-compiled Java language on commodity JVMs
  + An implementation based on bytecode rewriting is presented
  + Pre-built binaries without source code, such as the JRE, are fully supported
  + The overhead of object holograms is comparable to enabling debugging tools
  + The use of a mirror-based architecture is shown to offer benefits for bytecode rewriting frameworks in terms of rigor and simplicity
* and it reports on our experience applying object holograms to heap dump analysis
  + A collection of Eclipse MAT plugins for inspecting common object state is compared with their equivalent implementations using object holograms w.r.t. code complexity and execution speed

**Implementation**

* Mirrors API
  + Unless otherwise indicated, package is edu.ubc.mirrors
  + Interface hierarchy (will certainly be a UML-ish diagram):
  + ObjectMirror – root interface for object references, equivalent of java.lang.Object, defines getClassMirror() : ClassMirror
    - InstanceMirror – represents instances of classes (as opposed to array instances), defines getMemberField(String) : FieldMirror
      * ClassMirror – represents java.lang.Class , defines:
        + getSuperclassMirror
        + isInterface
        + getStaticField(String) : FieldMirror
        + getBytecode() : byte[]
        + ...
      * ClassLoaderMirror – represents java.lang.ClassLoader, defines loadClassMirror(String) : ClassMirror
      * ThreadMirror– represents java.lang.Thread, special case that defines getStackTrace : ObjectArrayMirror (where each element is a StackTraceElement mirror)
    - ArrayMirror – defines getLength() : int
      * ObjectArrayMirror – represents any reference array, defines get(int) : ObjectMirror and set(int, ObjectMirror)
      * ByteArrayMirror – represents byte[], defines get(int) : byte and set(int, byte)
      * (similar \*\*\*ArrayMirror interfaces for the other seven primitive types)
  + FieldMirror – analogous to java.lang.reflect.Field, defines
    - get() : ObjectMirror
    - getByte() : byte
    - getInt() : int
    - ...
    - set(ObjectMirror)
    - setByte(byte)
    - setInt(int)
    - ...
* Built-in implementations
  + edu.ubc.mirrors.native package
    - Implementations backed by native object (i.e. class and array instances) through reflective calls
    - Read-only, since values must be lifted to mirages on reading, and native objects cannot store arbitrary mirage instances in their fields/array elements
  + edu.ubc.mirrors.simple package
    - FieldMapMirror – implements InstanceMirror backed by a map from field names to values
    - DirectArrayMirror – implements all ArrayMirror subclasses (by boxing values) backed by a Object array
    - Mutable
* Class transformation
  + To allow mirror-based objects, class files are transformed to add a layer of indirection so that every place the bytecode operates on an object reference, a mirror instance is used instead
    - Using the ASM framework
    - (**TODO**: other special cases)
  + Native methods
    - Many native methods have reasonable alternatives
      * E.g. System.arraycopy
    - ClassMirror interface includes mechanism for specifying stub methods that implement a subset of the native methods on the class
    - Other native methods are replaced with stubs that throw a runtime error
      * Only those native methods that are encountered by the code to be executed need to be implemented
  + Object construction
    - Need to run original constructor methods, but reading/writing object fields through mirrors as above
    - Mirror instance is added as extra parameter to “<init>” methods
      * When called on an uninitialized value (i.e. after a NEW call), the mirror is created as above and passed in
      * When called on an uninitialized “this” value (i.e. as a call to super(…) or this(…)), the extra argument is passed up the hierarchy
    - Note: can only tell the difference between the two cases via the same dataflow algorithm
    - JVM forbids reading fields from uninitialized objects, but allows assigning fields on uninitialized “this” value
      * Fields may be set before actually assigning mirror field
        + Common with anonymous inner classes in Java
      * If the target object is an uninitialized “this” value, the mirror is obtained from the extra constructor argument instead of from the object
  + Note: An alternative implementation could replace object references with mirror instance directly and re-implement dynamic dispatch manually instead
    - Future work could evaluate, but tradeoff between manual method dispatch and extra wrapping of mirrors as mirages is likely comparable
* Isolating mirages
  + Class transformation is managed by a MirageClassLoader
    - References to the mirrors API and implementation (e.g. the heap dump implementation) are handled via delegation to a parent ClassLoader as per usual
    - Also holds a reference to a ClassLoaderMirror instance, which provides ClassMirror instances by class name
    - A mirror is wrapped as a mirage with MirageClassLoader#makeMirage(mirror)
      * Retrieves the mirage class defined by *mirageclass(*mirror.getClassMirror(), true)
        + If not already defined, transforms the result of # getBytecode()on the class mirror
        + Bytecode for the classes in the mirror model do not have to be related to the surrounding class hierarchy in any way
        + In the heap dump case, ideally this would be read from the PermGen area of memory directly, but this is not normally dumped by the JVM
        + Instead it is the user’s responsibility to recreate the class context

Should be straight-forward assuming the use of source version control

* + - * Instantiates it reflectively, passing the mirror value
      * The result cannot be cast to any type outside the mirage class loader (aside from Object), but a reflective call can start executing code on the mirage object model
      * Object#toString works because of the above, however, which is convenient
    - Thus bytecode rewriting and the less efficient execution is not applied to the whole application using the MirageClassLoader