Deep Static Modeling of invokedynamic

George Fourtounis
Yannis Smaragdakis

University of Athens

ECOOP 2019
The invokedynamic Framework

- JVM-friendly programmable dynamic (re)linking
- Ultra-powerful, crucial to analyze. Its core method handles API:
  - “poses a risk to the secure implementation of the Java platform.” (Holzinger et al. 2016)
  - “seems to provide less security by design than the Core Reflection API.” (Security Explorations 2011)
- Too dynamic for static analysis to tackle!
- Features using invokedynamic cause unsoundness in call-graph construction in current analysis tools (Reif et al. 2018)
- “there is a significant difference between supporting invokedynamic as a general feature, and invokedynamic as it is used by the Java 8 compiler for lambdas” (Sui et al. 2018)
invokedynamic Use Is Growing!

- Lambdas and method references are a pervasive feature of Java 8+ code
  - “… an increasing trend in the adoption rate of lambdas.” (Mazinanian et al. 2017)
- String concatenation in Java 9+ code
- Libraries and language runtimes using invokedynamic for its expressive power
- Dynamic JVM languages (e.g., Groovy)
- INVOKEDYNAMIC intrinsic proposal for Java source code (JEP 303)
Technology Background

• One instruction: `invokedynamic` (Java 7)
• Delegates linking of call sites to user-defined “bootstrap” code
  – Reifies call sites as Java objects
  – Call sites contain method handles
• Method handles + method types = type-safe method pointers
• The method handles API contains a dynamic code generator ("lambda forms")
Example: Late Linking

class C {
    A obj = new A();
    void run() {
        A.print(obj);
    }
}

class A {
    static void print(A a) {} } 

(new C()).run();
Example: Late Linking

class C {
    A obj = new A();
    void run() {
        INVOKEDYNAMIC "print" "(A)V" [obj]
        <A : CallSite bootstrap(MethodHandles.Lookup,
            String, MethodType)>
        []
    }
}
class A {
    static void print (A a) {}
    static CallSite bootstrap(..., String name, ...
        ...) {
        MethodType mt = ...
        MethodHandles.Lookup lookup = ...
        MethodHandle handle =
            lookup.find(A.class, name, mt);
        return new ConstantCallSite(handle);
    }
}
(new C()).run();
Java 8 Functional Features

• Lambdas and method references
  – Support functional programming idioms such as streaming pipelines
  – Java generalization: every single-abstract-method type (“SAM type”) becomes a lambda, automatically!

• Lambdas and method references are implemented with invokedynamic
Our Solution

- Model the full `invokedynamic` framework (including method handles, method types, and related APIs)
  - Work alongside a points-to analysis to integrate handling of the reified call site objects
  - Simulate behavior of dynamically-generated/native code

- Give a fast variant of our model for the common case of lambdas and method references

- Declarative model (Doop analysis framework)
  - Rules written in Datalog
  - Automatic mutual recursion between a robust points-to analysis, call-graph construction, exception analysis, reflection analysis, ...
Main Design Elements (Overview)

- Lots of mock objects (with the key features our analysis infers, and nothing more!)
  - for method handles, lookup objects, varargs, boxed allocations, ...
- Mutual recursion of invokedynamic analysis with points-to analysis, reflection analysis
  - much in the spirit of Doop/declarative analysis
- Connection of API elements based on how mock objects are used
  - “a handle that looks like this method reached this invokedynamic instruction, hence...”
Method Handles API: Invocation

- Call-graph edges, parameter values, return values
- Method handles and method types: better together
- Core technique: **mock analysis objects**
  - Invocation may convert arguments: analysis mocks boxed allocations
  - Constructor method handles are special, they allocate objects

\[
i \xrightarrow{h} m \quad Constr(m) \quad val = mock_h(t, h) \quad Ret(i) = v
\]

\[
m/this \leftrightarrow val \quad v \leftrightarrow val
\]

**MHConstr**
Method Handles API: Look-up

- Method handle lookup API
- Caller sensitivity: we tag mock values to propagate caller information in the program
- Interplay with classic reflection
  - Understand 
  - Conversions from reflective values
Generic invokedynamic

- invokedynamic calls the bootstrap code to create call sites
  - “Boot” call-graph edges: maintain a separate call graph for bootstrap calls
  - Model argument shifting: special handling of bootstrap invocations
  - Model methods accepting varargs: mock values to the rescue again

- Call sites contain method handles
  - Again, special handling for constructors
Resolving invokedynamic

**bootstrapped invokedynamic**

= invoke the method handle of the returned call site

\[ CSite(c, i, t) \quad CSite_C(c, h, m) \quad h = \langle *, \{ t, * \} \rangle \quad \text{MHCGE}_{\text{DYN}} \]

\[ i \xrightarrow{h} m \]
Call Sites Need Precision

• Our static analysis mutually recurses with orthogonal points-to analysis
  – to reason about the contents of call sites (and thus target methods)
• But a bootstrap method may be used in many call sites!
• To avoid polluting all sites with all handles, we filter call site targets according to method signature
Special, Fast-Path Modeling of Lambdas and Method References

- Java lambdas use `invokedynamic`
  - For implementation independence
    - alternative: static transformation (Retrolambda)
- Others: ad hoc, partial modeling of lambdas
- Very common features
  - Modeling must not depend on (slow) reflection analysis
  - Reuse non-reflective part of previous rules
- Phases:
  - Linkage: create lambda factory via metafactory
  - Capture: capture values from environment at lambda creation
  - Invocation: invocation of lambda (possibly elsewhere)
- Main technique: mock objects (carrying metadata) that propagate in the program
Evaluation I

- Test suite 1 (our own): extensive coverage of features of method handles, lambdas, method references, `invokedynamic`
- Freely available, bundled with Doop
- Reflection expensive for full `invokedynamic`

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method References</td>
<td>27</td>
</tr>
<tr>
<td>Lambdas</td>
<td>23</td>
</tr>
<tr>
<td>Method Handles and <code>invokedynamic</code></td>
<td>378</td>
</tr>
</tbody>
</table>
Evaluation II

- Test suite 2 (Sui et al. 2018): tuned for dynamic language features, provides ground truth
- Dynamo is the generic invokedynamic benchmark
- Loss of one target scenario due to absence of flow sensitivity in Doop

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Reachable</th>
<th></th>
<th>Unreachable</th>
<th></th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>expected</td>
<td>analysis</td>
<td>expected</td>
<td>analysis</td>
<td></td>
</tr>
<tr>
<td>LambdaConsumer</td>
<td>1</td>
<td>✓</td>
<td>1</td>
<td>✓</td>
<td>21</td>
</tr>
<tr>
<td>LambdaFunction</td>
<td>1</td>
<td>✓</td>
<td>2</td>
<td>✓</td>
<td>21</td>
</tr>
<tr>
<td>LambdaSupplier</td>
<td>1</td>
<td>✓</td>
<td>1</td>
<td>✓</td>
<td>22</td>
</tr>
<tr>
<td>Dynamo</td>
<td>1</td>
<td>✓</td>
<td>1</td>
<td>–</td>
<td>242</td>
</tr>
</tbody>
</table>
Conclusion

• We can analyze code containing `invokedynamic`!
• Our technique:
  – models API behavior
  – uses mock analysis objects
  – connects metadata across the program
• Full case aided by reflection analysis
• Common cases (lambdas/method references) supported by custom mode
Thank you!