Static Analysis of Java Dynamic Proxies

George Fourtounis
University of Athens
Dept. of Informatics
Greece
gfour@di.uoa.gr

George Kastrinis
University of Athens
Dept. of Informatics
Greece
gkastrinis@di.uoa.gr

Yannis Smaragdakis
University of Athens
Dept. of Informatics
Greece
yannis@smaragd.org

ABSTRACT

The dynamic proxy API is one of Java's most widely-used dynamic features, permitting principled run-time code generation and linking. Dynamic proxies can implement any set of interfaces and forward method calls to a special object that handles them reflectively. The flexibility of dynamic proxies, however, comes at the cost of having a dynamically generated layer of bytecode that cannot be penetrated by current static analyses.

In this paper, we observe that the dynamic proxy API is stylized enough to permit static analysis. We show how the semantics of dynamic proxies can be modeled in a straightforward manner as logical rules in the Doop static analysis framework. This concise set of rules enables Doop's standard analyses to process code behind dynamic proxies. We evaluate our approach by analyzing XCorpus, a corpus of real-world Java programs: we fully handle 95% of its reported proxy creation sites. Our handling results in the analysis of significant portions of previously unreachable or incompletely-modeled code.

CCS CONCEPTS

• **Software and its engineering → Compilers**; *General programming languages*; • **Theory of computation → Program analysis**;

KEYWORDS

dynamic proxy, reflection, static analysis

ACM Reference Format:
George Fourtounis, George Kastrinis, and Yannis Smaragdakis. 2018. Static Analysis of Java Dynamic Proxies. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3213846.3213864

1 INTRODUCTION

Modern mainstream languages, such as Java and C#, are statically typed, yet with extensive dynamic features. *Reflection* and *dynamic loading* are the chief features permitting access to data and code of types unknown at program compilation time. These features are responsible for much of the real-world value of the host languages. Java, for instance, has dominated server-side computing, with a large part of its appeal deriving from dynamic features: Reflection allows flexible frameworks that apply to unknown classes, yet interface with them seamlessly. Dynamic loading enables libraries that generate on-the-fly glue code for interfacing with distributed entities, data stores, and other environmental resources. These dynamic features, as well as others (such as the policy of lazy loading of interfaces) permit the enhancement of running code without needing to restart it—an invaluable feature in enterprise environments.

Reflection and dynamic loading add flexibility to a language but make it hard to analyze statically, since the semantics of operations depend on the dynamic flow of values. This is the Achilles' heel of modern static analysis frameworks. Most research tools ignore dynamic features, resulting in significant unsoundness of their modeling [32]. In particular, static analysis of the reflection API has attracted significant research effort [13, 27, 28, 31, 33, 48]. However, dynamic loading, although prevalent in practice, has enjoyed far less attention.

The majority of practical uses of dynamic loading occur through the *dynamic proxy* API [41]: a standard Java library facility that allows defining *invocation handler* code (effectively, a dynamically-typed interpreter of calls, through a generic `invoke` method) for a set of methods. The library generates a proxy class supporting statically-typed calls to these methods, yet forwarding the calls to the (dynamically typed) handler method.

Since its introduction in Java 1.3, more than 17 years ago, the dynamic proxy API has become ubiquitous. A large spectrum of vanilla applications and libraries now routinely employ dynamic loading, hidden behind the library API and often unbeknownst to the application programmers. In a recent survey of 461 open-source Java projects [26], Landman *et al.* find that 21% of them use dynamic proxies. The dynamic proxy API is so powerful that it has found a myriad of uses: it can efficiently implement its namesake design pattern [57], aspect-oriented programming features [4, 7, 46], object algebras [37], and meta-object protocols [19]; it can support design by contract [10], futures [44] and behavioral types [34]; it underlies the dynamic deployment of application components [14], mobile or distributed Java objects [3, 20, 53, 55], and typed publish/subscribe facilities [12]; it can be used to build modular interpreters [22] and in general to refactor for modularity [8]; it is leveraged for interoperability between languages [21]. Dynamic proxying is also present in Android [2], where the execution environment is locked down and applications have fewer tools at their disposal to do dynamic code generation. Android apps frequently have to resort to the dynamic proxy API to create new classes or implement dynamic features [1, 52]. The dynamic proxy capability can also be a security threat as recent Android malware uses such proxies [35, 42].

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213864

The dynamic proxy API owes its power to its dynamism. Thus, uses of the API present a problem for static analyses: the API essentially creates at runtime additional bytecode that does reflection, thus negating many of the assumptions that static analysis tools are based on. In their survey, Landman *et al.* find proxies to be *"very harmful for static analysis"*, recommending to programmers to *"avoid the use of dynamic proxies at any cost"* (for statically-analyzable software), and assess that *"no clear solution seems to be on the horizon"* [26]. Indeed, there is little past work on static analysis of proxies, and only in a domain-specific context [44] (analysis of transparency violations for proxies used to implement futures). Concretely, the dynamically-created layer of reflective indirection can make code appear unreachable and casts appear impossible, generally inhibiting analysis.

In this paper, we tackle the static modeling of the Java dynamic proxy API. We model the behavior of dynamic proxies in the general setting of points-to analysis, at the level of Java bytecode. Points-to analysis forms the substrate of other program analyses, determining how object values flow inter-procedurally, via the stack (method calls/returns) and the heap (field loads/stores). Therefore, our techniques can be leveraged by virtually any practical static analysis. This is the first general-purpose static analysis modeling of the dynamic proxy API, i.e., a full capture of the semantic effect of dynamic proxies on object flow and methods called.

Our analysis is built on the Doop [6] framework, which expresses analysis algorithms as Datalog logical rules. Modeling of dynamic proxies is done in mutual recursion with the underlying points-to analysis logic. Thus, the semantics of dynamic proxy operations appeal to inferences made by points-to analysis (e.g., values of variables, methods called at a virtual call site) while they also result in fresh points-to analysis inferences (abstract proxy objects propagating to program variables, invocation handlers being called). We handle semantic complexities such as implicit boxing and unboxing for primitive values and appeal to existing algorithms for the handling of reflection.

We evaluate the analysis on XCorpus, a corpus of modern real Java programs with an explicit goal of being a target for analyzing dynamic proxies [9]. XCorpus offers several benefits: it is a sizeable corpus and it supplies ground-truth information, which is invaluable for a rigorous evaluation.

We find the approach to be effective in capturing the semantic implications of dynamic proxies. We capture most uses of dynamic proxies (originating from 95% of proxy creation sites), i.e. hidden call-graph edges to the "invocation handler" objects associated with each proxy. We only miss the invocations on one dynamic proxy creation site due to missing XCorpus entry points. We also examine two case studies of programs that show the analysis of dynamic proxies in more detail.

2 JAVA DYNAMIC PROXIES

This section presents the basics of Java dynamic proxies and shows why a static analysis can encounter problems in code that uses proxies.

2.1 Use of Proxies

The Java Dynamic Proxy API [41] captures the idiom of generating a new class, with a dynamically selected set of methods and dynamically determined method implementations. The class can conform to any static API it chooses, expressed as a set of interfaces. These interfaces provide the linkage to the statically compiled code: the rest of the code can call the methods of the generated class through these interfaces. (Typically each interface is statically known, but which interfaces the dynamically generated class implements may vary per execution.) The actual implementation of the methods is provided via a generic handler, which accepts a method identifier (a.k.a., a *reified method*) and packaged arguments, and determines how to proceed.

In detail, a *dynamic proxy class* for a set of interfaces, `ins`, is a dynamically generated Java class that is guaranteed to implement all interfaces in `ins`, i.e. it supports all methods declared in the interfaces. This class can then be instantiated to create a *dynamic proxy instance* by passing it an invocation handler, `h`. The proxy (produced by the library) forwards all calls to the interface methods to the `invoke` method of the handler, with appropriate packaging of arguments [40].

There are two equivalent ways to create a dynamic proxy instance for some interface `I`:

```
ClassLoader c = ...
Class<?>[] ins = new Class<?> { I.class };
InvocationHandler h = ...

// (1) Create dynamic proxy class and
//     instance at the same time.
I obj = (I)Proxy.newProxyInstance(c, ins, h);

// (2) Create dynamic proxy class and
//     instantiate it via reflection.
I obj = (I)Proxy.getProxyClass(c, ins)
                .getConstructor(InvocationHandler.class)
                .newInstance(h);
```

The extra class loader passed above is used for security checks and to load the generated proxy class.

In current Java platforms, both ways to create proxies perform the same two operations:

(1) They internally call `getProxyClass0(c, ins)` which creates the *dynamic proxy class* (or finds it cached if it has already been generated). The actual low-level dynamic code generation is done in `sun.misc.ProxyGenerator.generateClassFile()` (OpenJDK) or in `java.lang.reflect.Proxy.generateProxy()` (Android).
(2) They call the constructor of this new class and pass it the invocation handler, to complete the instantiation of the *dynamic proxy instance*.

2.2 Static Analysis Problems

Insufficient modeling of proxies in a static analysis produces concrete problems of two kinds.

Problem 1: Impossible casts. Without a static analysis, we only know that both `newInstance()` and `newProxyInstance()` return `Object` values. Such a value at some point has to be cast to one of the expected interfaces, such as `I` in our example. Here lies the first problem when performing static analysis on this code: how do we

know that this cast can succeed? The semantics of the proxy has to appeal to the handling of values in array `ins`, which contains the reified class value `I.class`.

Problem 2: Reified methods and unreachable code. Assuming that we have solved the problem above, we can proceed to treat `obj` as if it implemented interface `I`. Now a second problem appears: calling one of the interface methods on the dynamic proxy instance leads nowhere. An analysis unaware of dynamic proxies has no sources for the dynamically generated proxy classes, and thus sees a proxy instance as a black box that cannot be further analyzed. The semantics of dynamic proxies should help us here and inform the analysis that such an instance would simply be a stylized wrapper on top of an invocation handler. Thus, for an analysis to process methods of proxy objects, we must tell it about either the dynamically generated code or the intended semantics of proxies.

2.3 Realistic Example

We will next examine a real-world example of the use of dynamic proxies. In addition to illustrating the API, the example serves to emphasize that practical uses of dynamic proxies are often convoluted and in code that also heavily employs reflection. Therefore any modeling of dynamic proxies has to be performed in conjunction with an analysis that has a good handling of the flow of values through a program, as well as a modeling of reflection operations.

Consider the following code fragment, heavily simplified but capturing a pattern in OkHttp [50]: a popular HTTP client library. (The original code spans 3 different methods, contains extra arguments to the reflective and proxy calls, handles exceptions, and more.) To achieve multi-platform compatibility, the library uses the external library ALPN[1] without making it a compile-time dependency. Proxies address this need by enabling the handling of methods in expected interfaces via dynamic code, in an invocation handler.

```
// Handle the methods of ALPN's ClientProvider and
// ServerProvider without a compile-time dependency
// on those interfaces.
class JettyNegoProvider implements InvocationHandler {
  public Object invoke(Object proxy, Method method,
       Object[] args) {
    String methodName = method.getName(); ...
    if (methodName.equals("supports")) {...}
    else if (methodName.equals("unsupported")) {...}
    else if (methodName.equals("protocols")) {...}
    else if ...
    else { return method.invoke(this, args); }
  }
}

String name = "org.eclipse.jetty.alpn.ALPN";
Class<?> clientClass =
  Class.forName(name + "$ClientProvider");
Class<?> serverClass =
  Class.forName(name + "$ServerProvider");
Object provider =
  Proxy.newProxyInstance(Platform.class.getClassLoader(),
    new Class[] {clientClass, serverClass},
    new JettyNegoProvider());

Class<?> negoClass = Class.forName(name);
Class<?> providerClass =
  Class.forName(name + "$Provider");
Method putMethod =
  negoClass.getMethod("put", providerClass);
putMethod.invoke(null, provider);  // regular reflection
```

[1] http://www.eclipse.org/jetty/documentation/9.4.x/alpn-chapter.html

The top part of the code is the invocation handler class, `JettyNegoProvider`. As can be seen, the `invoke` method, to which all proxied methods will dispatch, is *effectively an interpreter*, dynamically looking up the identities of called methods and handling them. The handling often involves reflection (e.g., `method.invoke`). The class handles methods of both interfaces `org.eclipse.jetty.alpn.ALPN$ClientProvider` and `org.eclipse.jetty.alpn.ALPN$ServerProvider`. The second part of the code establishes the connection, via dynamic proxies. It first *reifies* the two interfaces and asks the system for a proxy that forwards to the invocation handler all calls to the interfaces' methods. The proxy class is generated dynamically, loaded, and instantiated to yield an object that is the value of `provider`.

Finally, this object (of a dynamic type that never appears in the program's source) is registered with class `jetty.alpn.ALPN` using regular reflection operations: the reified class object is looked up (and stored in variable `negoClass`), its static `put` method is also looked up (after retrieving in `providerClass` the static type of its argument), and finally that method is invoked to register the instance of the proxy class. Note that the `invoke` call of the last line results in a (reflective) call to method `put` and has no relationship with the `invoke` call in the invocation handler. In fact, the last four statements of the example are equivalent to straightforward Java code:

```
org.eclipse.jetty.alpn.ALPN.put(provider);
```

However, the latter would introduce the precise compile-time dependency that the library is trying to avoid.

We, thus, see that dynamic proxies are not a feature in isolation. Their mere presence in the code indicates significantly dynamic behavior, which is likely to also be associated with reflection. Concretely, for a points-to analysis to reason about this code, it has to (a) understand reflection objects (e.g., `Method`, `Class`), (b) reason about such objects produced by operations involving substrings (such as `"$ClientProvider"`), and finally (c) reason about the semantics of dynamic proxies.

3 MODELING DYNAMIC PROXIES

We next present a high-level model of our analysis of dynamic proxies.

3.1 Model

Our support for dynamic proxies is implemented in Doop [6]: a static analysis framework, built around points-to analysis algorithms, that encodes its analyses declaratively, in the Datalog language. The use of Datalog for practical static analysis has a long history (e.g., [5, 18, 23–25, 29, 36, 45, 49, 58, 59]). The literature on static analysis for reflection, in particular, is almost entirely based on Datalog [27, 28, 31, 33, 48]. Datalog is also a good vehicle for exposition of analysis logic, since it offers a much higher-level semantic view than imperative algorithms. Accordingly, the latest edition of the Java Virtual Machine specification [30, p.170-320]

offers a Prolog/Datalog specification of the JVM verifier, using *"English language text [...] to describe the type rules in an informal way, while the Prolog clauses provide a formal specification."*

Our analysis logic is mutually recursive with a baseline analysis of the flow of values through a program—i.e., a points-to analysis (for complex objects) and a substring analysis (for strings). That is, the analysis for dynamic proxies appeals to inferences about the values that program variables can take, and at the same time produces new inferences about the values of program variables. Datalog is a good fit for such recursive logic: the computational backbone of the language is the definition of recursive relations. Computation in Datalog consists of monotonic logical inferences that apply to produce more facts until fixpoint. A Datalog rule "$\mathbf{C}(z, x) \leftarrow \mathbf{A}(x, y), \mathbf{B}(y, z)$." means that if $\mathbf{A}(x, y)$ and $\mathbf{B}(y, z)$ are both true, then $\mathbf{C}(z, x)$ can be inferred.

We next present a model of the dynamic proxy analysis, directly appealing to the relations defined in an underlying analysis, just as in our full implementation framework. (The Doop framework has mature support for substring-flow, points-to, and reflection analyses [48].)

The domains of the analysis comprise variables, V; abstract objects/allocation sites, O; invocation sites, I; method signatures, S; types, T; methods, M; and natural numbers, N. The underlying analysis (not shown, but standard in the literature—see e.g., [47]) provides several rules for computing relations **VarPointsTo**($v : V, o : O$), **CallGraphEdge**($i : I, m : M$), and **ArrayContentsPointTo**($o_{arr} : O, o : O$). These relations encode the usual inferences of an analysis that tracks the flow of values: which objects a variable may point to, which methods may be called at a call instruction, and which objects (any index of) an array may point to. Notably, the **VarPointsTo** and **ArrayContentsPointTo** relations also capture substring flow: a variable is considered to point to a substring if it may point to a string derived from that substring (through any sequence of concatenation operations). Our own model of dynamic proxies adds to the underlying rules, exploiting their inferences and adding new ones.

The entire analysis takes as input an intermediate representation of the program text, the relevant parts of which are encoded in the relations/tables shown in Figure 1. These relations are part of the standard treatment of an object-flow analysis with reflection (e.g., [47, 48]).

Preliminaries. The first step of modeling dynamic proxies is to provide new abstract objects. These objects represent proxy instances as well as objects that get implicitly allocated in order to hold the arguments to the `invoke` method of the proxy's invocation handler. The abstract objects are encoded in two new input relations, shown in Figure 2. (In our actual implementation, these relations are not part of the input, but computed before the main value-flow analysis runs. However, this is a mere engineering concern and the result is functionally equivalent to supplying the relations a priori.)

Logic Rules. Using the above abstract objects, our model of dynamic proxies captures the semantics of standard proxy operations: `newProxyInstance` calls, which connect a proxy object to its handler, and subsequent virtual calls (via an interface) to proxy methods, which get delegated to the invocation handler. The analysis derives two new relations, useful as intermediate concepts—shown in Figure 3.

Over this schema, the analysis is captured in four logical rules. We list and explain each of them individually. Note that rules can have multiple predicates in the head, as a syntactic convenience—this is equivalent to replicating the rule body for each of the head predicates.

VarPointsTo(v_{ret}, o_{proxy}), **ProxyObjectHandler**($o_{proxy}, o_{handler}$)
$\leftarrow$
  Call(i, "`Proxy.newProxyInstance`"), **AssignRetValue**(i, v_{ret}),
  ActualArg($i, 1, v_{ifaces}$), **ActualArg**($i, 2, v_{handler}$),
  VarPointsTo(v_{ifaces}, o_{ifaces}), **VarPointsTo**($v_{handler}, o_{handler}$),
  ArrayContentsPointTo(o_{ifaces}, o_i),
  ReifiedType(t_i, o_i), **ReifiedProxyInstance**(t_i, i, o_{proxy}).

In words, the rule says: if at a call site, i, of `newProxyInstance` the analysis establishes that the interface argument points to an array value, o_{ifaces}, that contains the meta object, o_i, of interface type t_i, and if the handler argument points to object $o_{handler}$, **then** a) the return value of the call, stored in variable v_{ret}, can take as value the abstract object representing the dynamic proxy, o_{proxy}, corresponding to the pair t_i and i; b) the handler object, $o_{handler}$ is registered for this proxy object for use by later rules.

As can be seen, the rule both uses regular analysis inferences (appealing to **VarPointsTo** twice and to **ArrayContentsPointTo**) and establishes new inferences for the **VarPointsTo** predicate.

The above rule handles the semantics of `newProxyInstance` calls. The next part of the model deals with interface calls over proxies.

CallGraphEdge(i, m_{invoke}),
VarPointsTo($v_{this}, o_{handler}$),
VarPointsTo(v_{proxy}, o_{proxy}),
VarPointsTo(v_{meth}, o_{meth}),
VarPointsTo(v_{args}, o_{arr}),
ProxyCallInfo(i, m_{invoke}, o_{arr})
$\leftarrow$
  Call(i, m), **ActualArg**($i, 0, v_{proxy}$),
  VarPointsTo(v_{proxy}, o_{proxy}),
  ReifiedProxyInstance(_, i_{nPI}, o_{proxy}),
  ProxyObjectHandler($o_{proxy}, o_{handler}$),
  ObjType($o_{handler}, t_{handler}$),
  Lookup("`InvocationHandler.invoke`", $t_{handler}, m_{invoke}$),
  FormalParam($m_{invoke}, 0, v_{this}$), **FormalParam**($m_{invoke}, 1, v_{proxy}$),
  FormalParam($m_{invoke}, 2, v_{meth}$), **FormalParam**($m_{invoke}, 3, v_{args}$),
  ReifiedMethod(m, o_{meth}),
  ReifiedHandlerArgArray(m, i_{nPI}, o_{arr}).

Despite the daunting form of the rule, its meaning is straightforward, given the previously established predicates. It states that when a call site, i, of a method is found to have a proxy receiver object (0-th argument), o_{proxy}, that was created at invocation site i_{nPI} (a call to `newProxyInstance`), the analysis looks up the handler for the proxy (using the previously-established predicate **ProxyObjectHandler**) and finds the `invoke` method in that handler, as well as all its parameter variables. As a result, all parameter variables are made to point to the appropriate objects (handler, proxy object,

Call$(i : I, s : S)$	instruction i is an invocation to a method with signature s
ActualArg$(i : I, n : N, v : V)$	at invocation i, the n-th parameter is local var v. For virtual calls, variable `this` is the 0-th parameter
FormalParam$(m : M, n : N, v : V)$	method m has v as its n-th formal parameter. The receiver is the 0-th parameter for virtual calls
AssignRetValue$(i : I, v : V)$	at invocation i, the value returned is assigned to local variable v
ReturnVar$(m : M, v : V)$	v is the return variable (assumed single) of method m
ObjType$(o : O, t : T)$	object o has type t
Lookup$(s : S, t : T, m : M)$	in type t there exists method m with signature s
ReifiedType$(t : T, o : O)$	special object o represents the class meta-object of type t. Such special objects are created up-front and are part of the input
ReifiedMethod$(s : S, o : O)$	special object o represents the reflection object for method signature s

Figure 1: Relations representing the input program and their informal meaning.

ReifiedProxyInstance$(t_{iface} : T, i : I, o : O)$	o is the abstract object representing dynamic proxies allocated at instruction i (a `newProxyInstance` call), for interface type t_{iface}
ReifiedHandlerArgArray$(m : M, i : I, o_{arr} : O)$	array object o_{arr} represents the argument array supplied to the invocation handler's `invoke` method, for calls to method m (of a proxied interface) over proxy objects created at instruction i (a `newProxyInstance` call)

Figure 2: Extra input relations for proxy instances and their argument arrays.

ProxyObjectHandler$(o_{proxy} : O, o_{handler} : O)$	abstract proxy object o_{proxy} has its method calls handled by the `invoke` method of object $o_{handler}$
ProxyCallInfo$(i : I, m_{invoke} : M, o_{arr} : O)$	call instruction i invokes a proxy method, whose implementation is provided dynamically by calling (the handler's) method m_{invoke} and passing it as argument array o_{arr}

Figure 3: Intermediate relations for proxy reasoning.

reified method, argument array), per the dynamic proxy semantics, and a call-graph edge is inferred from the invocation site to the `invoke` method of the handler. Auxiliary predicate **ProxyCallInfo** is also populated, to be used in later propagation of argument values.

Again, the dynamic proxy semantics appeal to predicates of the underlying points-to/string-flow analysis (**VarPointsTo**) and produce more inferences for such predicates (**VarPointsTo**, **CallGraphEdge**).

Finally, we use the established **ProxyCallInfo** inferences of the above rule to also propagate values for arguments and returns, when such information is available. This is done via the two rules below:

ArrayContentsPointTo$(o_{arr}, o_{arg}) \leftarrow$
  ProxyCallInfo$(i, \_, o_{arr})$, **ActualArg**$(i, \_, v_{arg})$,
  VarPointsTo(v_{arg}, o_{arg}).

VarPointsTo$(v_{ret}, o_{ret}) \leftarrow$
  ProxyCallInfo$(i, m_{invoke}, \_)$,
  ReturnVar(m_{invoke}, v_{hRet}), **VarPointsTo**(v_{hRet}, o_{ret}),
  AssignRetValue(i, v_{ret}).

The first of these rules states that, at a proxy call site, i, any value, o_{arg}, that an actual argument points to also flows to the contents of the argument array of the `invoke` call on the handler. The second rule does the inverse: any value held by the return variable, v_{hRet}, inside the handler, also becomes a value of the return variable, v_{ret} at the call-site.

3.2 Discussion

Compared to a realistic treatment (as in our full implementation) the model of the previous section is simplified in several ways:

- The model only captures the first pattern of dynamic proxy instantiation from Section 2.1: via `Proxy.newProxyInstance()`, and not via `Proxy.getProxyClass()` and reflection. This is a straightforward addition.
- The model shows a context-insensitive version of the rules. It is standard to add context-sensitivity through extra variables (representing a calling context or a heap context) in all relations [47].
- We use simplified method signatures and pre-allocated abstract objects for proxies, for ease of exposition.
- We postpone to Section 4 the discussion of special cases, such as boxing and methods (e.g., `hashCode`) with special semantics, per the dynamic proxy specification.

Additionally, our model integrates an important design choice, which is a good fit for our target analysis, but may need to be revisited in other settings: According to the definition of our **ReifiedProxyInstance** predicate, we generate an abstract proxy object per-interface and not per-combination-of-interfaces. At every proxy generation site, there is a separate abstract object for each interface, but the concrete code can generate a single proxy object that implements multiple interfaces—e.g.:

```
void meth(I1 x, I2 y) { ... }

void meth2() {
  ...
  InvocationHandler h = ...;
  Class<?>[] ins = new Class<?>[] { I1.class, I2.class };
```

```
  Object p = Proxy.newProxyInstance(..., ins, h);
  meth((I1)p, (I2)p);
}
```

This is not a limitation in our setting. Although there are two abstract proxy objects—one for interface `I1` and another for `I2`—they both have the same handler object: the value of `h`. The identity of proxy objects does not matter, nor do the proxy objects have user-visible state other than their handler. Since our target analysis is a path-insensitive analysis of where objects flow, proxy object identity has no semantic bearing on the analysis. (E.g., the analysis will disregard object equality checks.) Therefore, it is fine to use two abstract proxy objects that correspond to the same concrete proxy object: the only analysis semantics of the proxy objects are those of implementing methods by delegating to their handler, and both abstract objects will have the same handler. In the above example, the two arguments to `meth` will receive different abstract objects (due to the casts) but these will be functionally equivalent.

This simplifying design choice is advantageous in engineering terms because it avoids the need for pre-generating a large number of abstract types (an infeasible 2^N dynamic proxy classes for a program with N interfaces) or running the analysis multiple times: first to over-approximate what values can reach the interfaces argument (`ins` in our example) of `newProxyInstance` and then with the appropriate types for all useful interface combinations created.

4 SEMANTIC ADD-ONS AND SPECIAL FEATURES

The previous section showed the core of our handling of dynamic proxies. This section shows how our analysis handles details of the dynamic proxy API.

4.1 Proxy Argument Boxing

Java does implicit conversions between primitive types and their wrapper classes (e.g. `int`-to-`Integer` and vice versa) [17]. This conversion of a primitive value to an object ("boxing") is important for a points-to analysis, since a new object may have to be allocated and thus a new allocation site for that object exists behind the scenes. (Although this new object is an encoding of a mere integer, it needs to be represented in the analysis, since it may have other semantic implications, such as getting methods called on it—e.g., `wait` or other methods with synchronization semantics.)

Boxing in Java is normally done by the compiler. All necessary conversions are explicit at the bytecode level and should pose no problems for static analyses working at that level. For example, Doop, which uses the Soot framework [54] to derive an initial representation of the input bytecode, is mostly oblivious to boxing, only seeing its results: the analysis just encounters allocations via `new` as if the programmer had written the conversions by hand.

Dynamic proxying, however, performs boxing that is not reflected in the statically-available bytecode. A caller of a proxy method can supply primitive arguments, which is what the proxied interface expects, yet these will be transformed into object references when passed to the handler's `invoke` method. This boxing transformation is performed in the dynamically generated code. As an example of the problems when ignoring boxing, assume we have the Java code in the listing below. If we do not handle boxing, then variables `arg0` and `arg1` will have no values flowing into them and our analysis loses information.

```
interface G { float mult(float x, float y); }

G g = (G)Proxy.newProxyInstance(..., G.class,
                                new AHandler());

class AHandler implements InvocationHandler {
  Object invoke(Object proxy, Method m, Object[] args)
  throws Throwable {
    String mn = m.getName();
    if (mn.equals("mult") &&
        args != null && args.length == 2) {
      Float arg0 = args[0];
      Float arg1 = args[1];
      // Do something with these two objects.
    }
  }
```

We deal with boxing in our modeling of proxies by generating extra abstract objects per-boxed-type-and-parameter and storing them in predicate $\textsc{ReifiedBoxedPrimitive}(v : V, o_{boxed} : O)$, only for variables v that are formal arguments with a primitive type. A new argument propagation rule is employed (in addition to the earlier-shown rule that covers the case when actual arguments have abstract objects flow to them).

$$\textsc{ArrayContentsPointTo}(o_{arr}, o_{boxed}) \leftarrow \textsc{ProxyCallInfo}(\_, m_{invoke}, o_{arr}), \textsc{FormalParam}(m_{invoke}, \_, v), \textsc{ReifiedBoxedPrimitive}(v, o_{boxed}).$$

Notably, although the dynamic proxy semantics also involve unboxing of returned objects, our analysis does not care about unboxing. Unboxing is irrelevant for a points-to or string-flow analysis: it creates primitive values, not heap allocations. For other analyses of dynamic proxies (e.g., related to constant propagation), unboxing should also be taken into account.

4.2 Special Semantics

According to the specification of dynamic proxies, the invocation handler must have a special treatment for methods that have the same signature and name as `java.lang.Object`'s methods `hashCode()`, `equals()`, and `toString()`. For these methods, the reified `Method` object should not come from an interface (even if it exists for that too) but from `Object`. This semantic complexity means that the second of our rules in Section 3.1 needs to be replicated twice, with each version employing the appropriate condition (whether the called method is special or not), and with the rule's **ReifiedMethod** clause appropriately adapted. Since the rule is long and the change is straightforward but tedious, we do not show it here.

The rest of the methods of class `Object` do not pass through the invocation handler, as they are declared `final` and there is no dynamic dispatch for them.

Finally, an interesting observation is that the **CallGraphEdge** inferences in our approach have to interact correctly with the exception-handling logic of the underlying analysis. The implementation of exception handling in the Doop framework [5, 23] satisfies this property: the rules handling exception propagation appeal to the **CallGraphEdge** relation without concern of what this encodes. Recall that our **CallGraphEdge** derivations connect the

handler's `invoke` method to the proxy's call site. This is effectively equivalent to re-throwing (in the proxy) all exceptions that the handler's `invoke` method throws, as also mandated by the dynamic proxy specification.

4.3 Discussion

Our approach makes several realistic design choices and compromises. It also offers significant opportunities for extensions.

- **Class loader handling.** Currently, we omit modeling the class loader argument, since its uses are orthogonal to our points-to analysis. For other analyses that might need it (e.g., to check security domains across programs that use multiple class loaders), this argument should be easy to add in the model, as its value is not modified by proxy class generation.
- **Packages and visibility.** Having single-interface proxy classes means we cannot follow the dynamic proxy specification for what the package of the proxy will be. The specification states that if all interfaces are `public`, then the proxy's package is `com.sun.proxy`; however, if all non-public interfaces come from some package *P*, then the proxy's package is *P*. The modeling of packages is unimportant for our analysis.

  We also ignore subtleties such as the visibility of proxy classes: a proxy class must be `public` if its interface is `public`, otherwise it is non-public.
- **Implicit Exceptions.** While our approach handles exceptions thrown by the invocation handler, the dynamic proxy specification also mandates the implicit throwing of exceptions in cases of type-incompatibility: when the value returned (or thrown) by `invoke` is not convertible to the type declared by the proxied interface, a `ClassCastException` or `UndeclaredThrowableException` or `NullPointerException` should be thrown. Supporting these cases required simple changes to Doop's exception analysis logic and is orthogonal to the basic analysis that we have already presented.
- **Array argument granularity.** Our analysis relies on arrays and their analysis since (a) the interfaces argument is an array and (b) method-based reflection is based on Java's varargs, which Soot translates into arrays. The Doop points-to analyses (as well as most points-to analyses in languages without extensive use of constant indices) are *array-insensitive*: they do not reason about specific indices of arrays. If an object flows into an array, we only know afterwards that some array location may point to this object. For proxies, this means that we do not capture the ordering semantics of proxies or the constraint that the interfaces argument must be a list of unique `Class` values [39, 41]. The ordering of interfaces is significant when a dynamic proxy implements many interfaces having duplicate methods. Our modeling with single-interface proxy classes means that in an ambiguous call to a method existing in multiple interfaces implemented by a proxy, a call graph edge is created for all of these methods, thus capturing all possible interface orderings. Not having ordering information is thus not a threat to correctness but to precision: this information might be useful for additional filtering of the possible outcomes of a proxy generation call.
- **Java 8 support.** Java 8 default interface methods can also be accessed via proxy. This requires two elements inside an invocation handler: a `meth.isDefault()` check and some complex code [16]. We currently do not offer support for default interface methods in proxies, since the underlying Doop reflection analysis also does not fully model the necessary reflection API (e.g., `java.lang.invoke.MethodHandles`). This support is part of future work.
- **Other Dynamic Proxy Patterns.** Our analysis intends to cover common uses of proxies, in conjunction with other analyzable language features. Clearly, some dynamic patterns are missed. For instance, if an interface is loaded from a file and then it is used to generate a proxy, we will miss this case. We also may miss reflective uses of reflection (this also happens for ordinary reflection in Doop). More generally, dynamic proxies are just one instance (albeit, the most popular) of dynamic loading. Less disciplined uses of dynamic loading are harder to model.

  There are, however, generalizations of the standard dynamic proxy pattern that could be modeled with an approach similar to ours. Uniform proxies [11] are similar to Java's dynamic proxies, but work for classes instead of interfaces, i.e., support the dynamic generation of new subclasses of given classes. Such functionality is provided for example by Android's ProxyBuilder[2] and the Enhancer class of the bytecode manipulation library cglib.[3] While dynamic proxy classes in Java are subclasses of `java.lang.reflect.Proxy`, and thus cannot admit a second superclass, our technique does not depend on having `Proxy` as a superclass.

  Notably, a common non-standard-library dynamic proxy pattern is Guava's dynamic proxy, which is a particularly good fit for our approach. Guava's dynamic proxies can only support a single interface. This means that the implementation can leverage generic methods, therefore eschewing the need for the cast that appears in standard Java proxies.[4] (Due to erasure, the cast must still appear in the bytecode.) Note that in Section 5.2 we evaluate our analysis on Guava as the latter is part of XCorpus.
- **Dynamic proxies in other languages.** Dynamic proxies also arise in other languages, in closely related form. The .NET platform offers a similar concept, both in the standard library and in third-party libraries [43]. Scala has Dynamic Proxy functionality but its use does not seem widespread.[5]

5 EVALUATION

We evaluate our analysis on the programs of XCorpus, a corpus of real Java programs, which explicitly mentions dynamic proxies as one of the features it captures [9]. We also examine two case studies (the `okhttp` library and the `guice` dependency injection framework) to discuss the analysis of patterns of dynamic proxies in detail.

5.1 Methodology

All analyses are run on a 64-bit machine with an Intel Xeon E5-2687W v4 3.00GHz with 512 GB of RAM. We use the Soufflé compiler (v.1.2.0), which compiles Datalog specifications into binaries via C++ and run the resulting binaries in parallel mode using four

[2] https://android.googlesource.com/platform/dalvik/+/26f9572/dx/src/com/android/dx/gen/ProxyBuilder.java
[3] http://cglib.sourceforge.net/apidocs/net/sf/cglib/Enhancer.html
[4] https://github.com/google/guava/wiki/ReflectionExplained#dynamic-proxies
[5] http://lampwww.epfl.ch/~hmiller/scaladoc/library/scala/reflect/DynamicProxy.html

jobs. Doop uses the Java 8 platform as implemented in Oracle JDK v1.8.0_121.

For all programs, we know beforehand (from XCorpus reports or manual inspection for the two case studies) which are the target `newProxyInstance()` sites that should be analyzed and run a context-insensitive points-to analysis without dynamic proxy support to test if the static analysis also finds them reachable. We then run the static analysis augmented with the proxy suport rules, measure the runtime, and examine the results.

Our context-insensitive analysis uses the "classic reflection" mode of Doop. This provides a reflection analysis with the following features:

- Forward reasoning of reflection operations based on string constants passed as arguments [31, 33].
- Reasoning about the flow of substrings through string buffers and string builders and generalization of the above reflection reasoning [48].
- Limited backward reasoning about reflective operations by leveraging how reflective objects are being used in later code [27]. The combination of forward and backward reasoning is optimized for precision, not soundness: the reasoning has to be with very high confidence (e.g., input strings or use of the reflective object in the same method) otherwise forward and backward inferences need to agree in order for a reflective call to be resolved.

Running times are to be seen only as indicative measurements: they represent a single run and (empirically) variations of around 5%-10% are not uncommon. However, our experimental assessment is about capturing the semantics of dynamic proxies—timings are only assessed in rough, qualitative terms, i.e., to indicate either order-of-magnitude slowdowns or minimal overhead of the proxy analysis.

5.2 XCorpus

XCorpus consists of executable Java programs accompanied by test suites to ensure code coverage. XCorpus provides a mechanism that records the features used in its programs, reporting 13 programs containing calls to `newProxyInstance()`, i.e. programs possibly creating dynamic proxy objects.

XCorpus stops at reporting call sites to `newProxyInstance()`, as reasoning about the possible calls involving invocation handlers needs a full-blown static analysis, similar to the one we present in this paper. Our analysis should be able to go one step further than the XCorpus result: it should show where calls on dynamic proxies create call-graph edges to the `invoke()` method of invocation handlers. Thus, we must answer the following research question:

> RQ: *For proxies created by a* `newProxyInstance()` *site reported by XCorpus, do we find call-graph edges to the proxy's invocation handlers?*

Table 1 shows the results of our analysis on XCorpus. For each program, the second column shows whether Doop actually finds reachable calls to `newProxyInstance()` reported by XCorpus. If yes, the third and fourth columns show if our static analysis resolves calls on the proxy: *Def-handled* means calls are resolved by our default reflection-based technique (*Def-Reflective*), while *Opt-handled* means calls are resolved by our optimized, limited-reflection technique (*Opt-Reflective*, more on the difference between these two variants of the analysis follow in Section 5.2.2).

The analysis uses the XCorpus tests as entry points for every program, except for `picocontainer`, for which there is a freely available example program[6] that we use to exercise dynamic dependencies. In `jhotdraw`, the XCorpus tests exercising one proxy creation site were also fixed to actually create proxies (since they would only test failed creation of dynamic proxies); the fix was based on the intended use of these proxies as documented in `jhotdraw` comments.[7]

5.2.1 Call Sites That Create Dynamic Proxies. In total, the static analysis finds (i.e., reports as reachable) 19 call sites that create dynamic proxies out of 20 possible call sites in the XCorpus code. This check is done in a preliminary context-insensitive analysis supporting reflection but not dynamic proxies. The original 20 call sites are those reported by running the "feature analysis" of XCorpus. The call site in `squirrel-sql` is not found as the XCorpus suite does not offer appropriate entry points that reach it.

Thus we fully handle 95% (19/20) of the sites reported by XCorpus when using our *Opt-Reflective* analysis vs. 55% (11/20) when using *Def-Reflective* (see below for the differences between the two analyses). For all these sites, our analysis finds the associated invocation handler and resolves method calls on the proxy by creating call-graph edges that point to the handler's `invoke()` method. If we only examine the sites analyzable in the given time/memory constraints, then the success rate is 100%: in all experiments, *whenever the analysis terminates, it has successfully analyzed every proxy creation site it found.* It should be noted that running out of resources is not caused by a looping analysis (we use Datalog for its terminating behavior) but due to input programs being too big or containing a lot of reflective code.

5.2.2 Opt-Reflective *vs.* Def-Reflective *Analysis.* We try two variants of our dynamic proxy analysis:

- The *Def-Reflective* analysis is the complete analysis shown in this paper, which is mutually recursive with Doop's reflection analysis [24]. It thus adds dynamic proxy support on top of the default reflection analysis of Doop.
- The *Opt-Reflective* analysis is the analysis shown in this paper, coupled with a focused part of Doop's reflection analysis that only reasons about expressions of the forms `X.class` and `Class.forName("constant string")`.[8] This analysis is often enough to reason about the *Class* objects representing interfaces that flow into the array parameter of `newProxyInstance()`. This analysis thus does not track the flow of reified method values and misses the rule of Section 3.1 that can reason about calls via such objects in the `invoke()` method of an invocation handler. However, this analysis can still track simple intraprocedural uses of reflective method calls, which allows it to penetrate the proxy-based OSX adapters used in `aoi`, `batik`, `jedit`, and `jhotdraw`.

We observe that *Opt-Reflective* and *Def-Reflective* are both useful but have different performance: if we only want to find call-graph

[6] https://github.com/avh4/picocontainer-example
[7] https://sourceforge.net/p/jhotdraw/svn/728/tree/branches/ConnectorStrategy/jhotdraw7/src/main/java/org/jhotdraw/gui/event/GenericListener.java
[8] In practice, `javac` compiles the former to the latter.

Table 1: XCorpus evaluation.

Benchmark	Proxy creation sites				Invocation handler edges		Analysis time
	XCorpus reported	**Doop reachable**	*Opt-handled*	*Def-handled*	*Opt-Reflective*	*Def-Reflective*	
aoi-2.8.1	1	1	1	–	6,911	–	206min, timeout (4hr)
batik-1.7	1	1	1	1	411	4,459	8min, 87min
castor-1.3.1	3	3	3	–	9,384	–	26min, timeout (4hr)
drools-7.0.0.Beta6	1	1	1	–	15,205	–	143min, timeout (4hr)
guava-21.0	2	2	2	2	5,350	10,214	11min, 42min
jedit-4.3.2	2	2	2	2	4,516	23,653	29min, 213min
jhotdraw-7.5.1	2	2	2	2	880	3,628	11 min, 183 min
jrat-0.6	1	1	1	1	10	10	3 min, 8 min
mockito-core-2.7.17	1	1	1	1	13	16	4 min, 8 min
picocontainer-2.10.2	1	1	1	1	881	6,943	2 min, 213 min
pmd-4.2.5	3	3	3	–	9	-	19 min, timeout (4hr)
quartz-1.8.3	1	1	1	1	12	4,828	7 min, 25 min
squirrel_sql-3.1.2	1	0	0	–	0	–	10min, timeout (4hr)
Total	20	19	19	11			

edges to invocation handlers, *Opt-Reflective* is a good choice, as it usually works as well as *Def-Reflective*, but terminates in less time. This is particularly telling for `aoi`, `castor`, `drools`, and `pmd`, where *Def-Reflective* times out, yet *Opt-Reflective* works.

However, the *Def-Reflective* analysis is necessary when analyzing reflective code, such as the `picocontainer` library. Although both analyses find uses of the same invocation handler object, *Def-Reflective* finds 6,943 call-graph edges, while *Opt-Reflective* only finds 881 edges. In this and other benchmarks (`batik`, `guava`, `jedit`, and `quartz`), we see significantly more call-graph edges under *Def-Reflective*, as full reflection support is needed to uncover extra code to analyze.

In general, the *Def-Reflective* analysis is necessary if we want to analyze `invoke()` methods that do reflection, interacting with the `java.lang.reflect.Method` parameter: for such code, the *Opt-Reflective* analysis will not be able to analyze method calls on these method objects.

5.3 Case Studies

We next discuss in more detail two additional applications and their dynamic proxy patterns. One corresponds to our earlier motivating example, while the other exhibits very extensive proxy-related behavior.

OkHttp. As we saw in Section 2.3, OkHttp makes use of dynamic proxies. The library uses dynamic proxies for setting up TLS, which is needed in practice [51] for HTTP/2 (one of the library's main features). Not having information about the proxy means the TLS setup of the library is not analyzed and there is incomplete information about the calls made by the fundamental "platform" object.

The total size of the OkHttp library is 50,919 lines of Java code.[9] For our evaluation, we analyze `mockwebserver`, the scriptable web server of OkHttp that exercises the full HTTP stack and thus makes realistic use of the library.

Our analysis discovers that the `provider` variable can point to objects of the two dynamic proxy classes that implement the ALPN interfaces `org.eclipse.jetty.alpn.ALPN$ClientProvider` and `org.eclipse.jetty.alpn.ALPN$ServerProvider`. Our analysis also finds that the `invoke()` method of the invocation handler is reachable and creates 2,866 new call graph edges. As we described in Section 2.3, this analysis is only possible because we have support for reasoning about substrings in the reflection analysis.

In practice, the analysis of this medium-size codebase takes more time, 178 vs 294 secs (no proxy support vs. proxy support).

Guice JNDI Client. Google Guice[10] is a lightweight framework for the dependency injection design pattern (DI) [15] that uses dynamic proxies.

The dependency injection pattern addresses (among other issues) the automated construction of object graphs in programs. This automation is orchestrated by the programmer, who gives a description of the parts making up a program and their dependencies. The DI framework will then compose the graph of objects by constructing appropriate ones, parameterized by their dependencies, according to a user-provided configuration.

Dependency injection is commonly used with resource locators, such as the Java Naming and Directory Interface (JNDI) [38], to find the objects that correspond to a given configuration. The Guice framework contains JNDI functionality [56] and includes an example of how to implement a JNDI provider client, `guice-jndi`. For this case study, we examined version 4.1 of Guice, containing 62,093 lines of Java code in total and ran our analysis on `guice-jndi`.

Compared to other dependency injection frameworks that use explicit code or external configuration resources (e.g., XML files) for dependency declarations, Guice uses Java annotations in the sources of the original program. However, the Java annotations parser uses dynamic proxies and thus a critical path in the Guice code is not followed by analyses that are not aware of proxies.

[9] Commit version `de8699b62d24e3b31205229a259be5b26b040957` in the library's Github repository [50].

[10] https://github.com/google/guice

We confirm this by checking what are the reachable methods containing calls to `newProxyInstance()` when running a simple analysis without dynamic proxy support. (This analysis is the standard context-insensitive analysis of Doop, but with dynamic proxy support turned off, so that the analysis finds calls to `newProxyInstance()` in already reachable code but cannot reason about them.) The following classes and reachable methods are found (to avoid ambiguity we show method descriptors, omitting package prefixes for clarity):

(1) In `AnnotationParser`:
```
Annotation annotationForMap(Class, Map)
```
(2) In `ConstructionContext`:
```
Object createProxy(Errors, InjectorImpl$InjectorOptions,
                   Class)
```
(3) In `Annotations`:
```
Annotation generateAnnotationImpl(Class)
```

Without support in the analysis for dynamic proxies, the set of reachable invocation handler methods contains mostly class and instance initializers (constructors):

(1) In `DelegatingInvocationHandler`:
```
void <init>()
void setDelegate(Object)
```
(2) In `AnnotationInvocationHandler`:
```
void <init>(Class,Map)
void <clinit>()
```
(3) In `Annotations$2`:
```
void <init>(Class,Map)
```

The existence of the initializers means the current analysis can already see that some handlers are created—thus, their classes have to be initialized and their constructors become reachable. However, the incomplete modeling of dynamic proxies means that the results of proxy instantiations can never be cast to the interface type: for Doop, these casts can only fail and, thus, no method can be called on these returned values.

When we turn on support for dynamic proxies, the set of reachable methods now contains the methods in the invocation handlers:

(1) In `AnnotationInvocationHandler`:
```
void <init>(Class,Map)
Object invoke(Object,Method,Object[])
Object cloneArray(Object)
String toStringImpl()
Boolean equalsImpl(Object)
AnnotationInvocationHandler asOneOfUs(Object)
Method[] getMemberMethods()
int hashCodeImpl()
String memberValueToString(Object)
boolean memberValueEquals(Object,Object)
int memberValueHashCode(Object)
Class access$000(AnnotationInvocationHandler)
void <clinit>()
void access$100(AnnotationInvocationHandler,Method[])
void validateAnnotationMethods(Method[])
```
(2) In `DelegatingInvocationHandler`:
```
void <init>()
Object invoke(Object,Method,Object[])
void setDelegate(Object)
```
(3) In `Annotations$2`:
```
void <init>(Class,Map)
Object invoke(Object,Method,Object[])
```

In this case, the analysis is successful in fully tracking the uses of the annotation parser code.

Our analysis also creates 13,172 call-graph edges due to dynamic proxies, a high number compared to the previous case study and most programs in XCorpus. It also shows that 524 different methods are called on proxy objects (and are intercepted by the invocation handler's `invoke()`). Thus Guice makes extensive use of dynamic proxies, instead of using them locally in a confined piece of code. Similarly, many call-graph edges were measured for the XCorpus program `picocontainer`, another DI framework. These observations indicate that dynamic proxies are pervasive in DI designs.

The running time of the reflection analysis without the proxy rules is 342 secs, compared to 1,298 secs for the same analysis supporting dynamic proxies. Clearly, this is a substantial difference, reflecting the extra call-graph edges and overall inferences of the analysis. (For instance, the main analysis relation, VarPointsTo, grows from a size of 104M tuples to 366M.)

5.4 Summary

Our experiments demonstrate the effectiveness of our dynamic proxy support for modeling the behavior of proxies and their clients. Over realistic programs with complex dynamic proxy patterns, our approach captures the semantics of dynamic proxies and models the relevant subsystems that they implement.

6 CONCLUSION

We have discussed a static modeling of the Java dynamic proxy pattern—the most widespread use of dynamic loading in Java programs. The modeling reveals several insights. Chief among them is the need for mutually recursive handling of dynamic proxies and other object flow through the program, both via regular operations (calls and heap loads/stores) and via reflective actions. Static modeling of proxies can rarely be effective without a full treatment of other program semantics (e.g., flow of string constants, or of reified interface objects) and needs to be integrated in a more general analysis model. We performed this integration both in our simplified formal model, consisting of a handful of Datalog rules that appeal to standard points-to and reflection analysis relations, and in our full implementation in the Doop framework.

Our model can be seen as a validation of the approach of expressing static analyses in terms of mutually recursive relations. A complex semantic task is made easily manageable as part of a declarative analysis that simultaneously tackles several semantic concerns.

Arguably, improving the soundness of a static analysis by covering complex language features also has great intangible value. Static analysis is rarely about covering the easy cases of program behavior—it excels, relative to shallower techniques, at covering hard-to-reveal cases. Additionally, static analysis is about achieving certainty, and the lack of modeling of a language feature has a high cost in loss of perceived certainty. Under this light, we consider the modeling of dynamic proxies to be a very-high-value proposition for static analysis techniques.

ACKNOWLEDGMENTS

We gratefully acknowledge funding by the European Research Council, grant 307334 (SPADE), a Facebook Research and Academic Relations award, and an Oracle Labs collaborative research grant.

REFERENCES

[1] 2016. FAQ - androidannotations Wiki. https://github.com/androidannotations/androidannotations/wiki/FAQ. Accessed: Jan. 28, 2018.

[2] 2016. Proxy | Android Developers. https://developer.android.com/reference/java/lang/reflect/Proxy.html Accessed: Jan. 28, 2018.

[3] G. Biegel, V. Cahill, and M. Haahr. 2002. *A Dynamic Proxy Based Architecture to Support Distributed Java Objects in a Mobile Environment.* Springer Berlin Heidelberg, Berlin, Heidelberg, 809–826. https://doi.org/10.1007/3-540-36124-3_54

[4] Jonas Bonér. 2008. Real-World Scala: Managing Cross-Cutting Concerns using Mixin Composition and AOP. http://jonasboner.com/real-world-scala-managing-cross-cutting-concerns-using-mixin-composition-and-aop.

[5] Martin Bravenboer and Yannis Smaragdakis. 2009. Exception Analysis and Points-to Analysis: Better Together. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA '09)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1572272.1572274

[6] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[7] Ruzanna Chitchyan and Ian Sommerville. 2004. Comparing Dynamic AO Systems. In *Proceedings of the 2004 Dynamic Aspects Workshop (DAW04)*.

[8] Jens Dietrich, Catherine McCartin, Ewan Tempero, and Syed M. Ali Shah. 2010. *Barriers to Modularity - An Empirical Study to Assess the Potential for Modularisation of Java Programs.* Springer Berlin Heidelberg, Berlin, Heidelberg, 135–150. https://doi.org/10.1007/978-3-642-13821-8_11

[9] Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. 2017. XCorpus – An executable Corpus of Java Programs. http://www.jot.fm/contents/issue_2017_04/article1.html. *Journal of Object Technology* 16, 4 (Aug. 2017), 1:1–24. https://doi.org/10.5381/jot.2017.16.4.a1

[10] A. Eliasson. 2002. Implement Design by Contract for Java using Dynamic Proxies. http://www.javaworld.com/javaworld/jw-02-2002/jw-0215-dbcproxy.html. Accessed: Jan. 28, 2018.

[11] Patrick Eugster. 2006. Uniform Proxies for Java. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 139–152. https://doi.org/10.1145/1167473.1167485

[12] Patrick Eugster. 2007. Type-based Publish/Subscribe: Concepts and Experiences. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 1, Article 6 (Jan. 2007). https://doi.org/10.1145/1180475.1180481

[13] Stephen J. Fink et al. [n. d.]. WALA UserGuide: PointerAnalysis. http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis.

[14] Marc Fleury and Francisco Reverbel. 2003. The JBoss Extensible Server. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware (Middleware '03)*. Springer-Verlag New York, Inc., New York, NY, USA, 344–373.

[15] Martin Fowler. 2004. Inversion of Control Containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html#ServiceLocatorVsDependencyInjection

[16] Dominic Fox. 2015. New Tricks with Dynamic Proxies in Java 8 (part 2). https://opencredo.com/dynamic-proxies-java-part-2/. Accessed: Jan. 28, 2018.

[17] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

[18] Salvatore Guarnieri and Benjamin Livshits. 2009. GateKeeper: mostly static enforcement of security and reliability policies for Javascript code. In *Proceedings of the 18th USENIX Security Symposium (SSYM' 09)*. USENIX Association, Berkeley, CA, USA, 151–168.

[19] Youssef Hassoun, Roger Johnson, and Steve Counsell. 2003. Reusability, Open Implementation and Java's Dynamic Proxies. In *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java (PPPJ '03)*. Computer Science Press, Inc., New York, NY, USA, 3–6.

[20] Youssef Hassoun, Roger Johnson, and Steve Counsell. 2005. Applications of dynamic proxies in distributed environments. *Software: Practice and Experience* 35, 1 (2005), 75–99. https://doi.org/10.1002/spe.629

[21] Rich Hickey. [n. d.]. Clojure - Java Interop: Implementing Interfaces and Extending Classes. https://clojure.org/reference/java_interop#_implementing_interfaces_and_extending_classes Accessed: Jan. 28, 2018.

[22] Pablo Inostroza and Tijs van der Storm. 2017. Modular interpreters with implicit context propagation. *Computer Languages, Systems & Structures* 48 (2017), 39 – 67. https://doi.org/10.1016/j.cl.2016.08.001 Special Issue on the 14th International Conference on Generative Programming: Concepts & Experiences (GPCE'15).

[23] George Kastrinis and Yannis Smaragdakis. 2013. Efficient and Effective Handling of Exceptions in Java Points-to Analysis. In *Proceedings of the 22nd International Conference on Compiler Construction (CC '13)*. Springer, 41–60. https://doi.org/10.1007/978-3-642-37051-9_3

[24] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-Sensitivity for Points-To Analysis. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA.

[25] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. 2005. Context-sensitive program analysis as database queries. In *Proceedings of the 24th Symposium on Principles of Database Systems (PODS '05)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1065167.1065169

[26] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection – Literature Review and Empirical Study. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017.*

[27] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-Inferencing Reflection Resolution for Java. In *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP '14)*. Springer, 27–53.

[28] Yue Li, Tian Tan, and Jingling Xue. 2015. Effective Soundness-Guided Reflection Analysis. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings (Lecture Notes in Computer Science)*, Sandrine Blazy and Thomas Jensen (Eds.), Vol. 9291. Springer, 162–180. https://doi.org/10.1007/978-3-662-48288-9_10

[29] Percy Liang and Mayur Naik. 2011. Scaling abstraction refinement via pruning. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 590–601. https://doi.org/10.1145/1993498.1993567

[30] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2015. *The Java Virtual Machine Specification, Java SE 8 Edition.*

[31] Benjamin Livshits. 2006. *Improving Software Security with Precise Static and Runtime Analysis.* Ph.D. Dissertation. Stanford University.

[32] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. https://doi.org/10.1145/2644805

[33] Benjamin Livshits, John Whaley, and Monica S. Lam. 2005. Reflection Analysis for Java. In *Proceedings of the 3rd Asian Symp. on Programming Languages and Systems.* Springer, 139–160. https://doi.org/10.1007/11575467_11

[34] Iain McGinniss and Simon Gay. 2011. Hanoi: A Typestate DSL for Java. (April 2011).

[35] Alvaro Muñoz and Christian Schneider. 2016. Serial Killer: Silently Pwning Your Java Endpoints. https://www.rsaconference.com/writable/presentations/file_upload/asd-f03-serial-killer-silently-pwning-your-java-endpoints.pdf. In *RSA Conference.* Accessed: Jan. 28, 2018.

[36] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 308–319. https://doi.org/10.1145/1133981.1134018

[37] Bruno C. d. S. Oliveira, Tijs van der Storm, Alex Loh, and William R. Cook. 2013. Feature-Oriented Programming with Object Algebras. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 27–51. https://doi.org/10.1007/978-3-642-39038-8_2

[38] Oracle. [n. d.]. Java Naming and Directory Interface (JNDI). http://www.oracle.com/technetwork/java/jndi/index.html Accessed: Jan. 28, 2018.

[39] Oracle. 2016. Dynamic Proxy Classes. http://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html Accessed: Jan. 28, 2018.

[40] Oracle. 2016. InvocationHandler (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/InvocationHandler.html Accessed: Jan. 28, 2018.

[41] Oracle. 2016. Proxy (Java Platform SE 8). http://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html Accessed: Jan. 28, 2018.

[42] Palo Alto Networks, Inc. 2016. PluginPhantom: New Android Trojan Abuses "DroidPlugin" Framework. http://researchcenter.paloaltonetworks.com/2016/11/unit42-pluginphantom-new-android-trojan-abuses-droidplugin-framework/. Accessed: Jan. 28, 2018.

[43] Mariano Martinez Peck, Noury Bouraqadi, Stéphane Ducasse, Luc Fabresse, and Marcus Denker. 2013. Ghost: A Uniform and General-Purpose Proxy Implementation. *CoRR* abs/1310.7774 (2013). http://arxiv.org/abs/1310.7774

[44] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. 2004. Transparent Proxies for Java Futures. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. ACM, New York, NY, USA, 206–223. https://doi.org/10.1145/1028976.1028994

[45] Thomas W. Reps. 1994. Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*, R. Ramakrishnan (Ed.). Kluwer Academic Publishers, 163–196.

[46] Chris Schaefer, Clarence Ho, and Rob Harrop. 2014. *Introducing Spring AOP.* Apress, Berkeley, CA, 161–239. https://doi.org/10.1007/978-1-4302-6152-0_5

[47] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Foundations and Trends® in Programming Languages* 2, 1 (2015), 1–69. https://doi.org/10.1561/2500000014

[48] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More Sound Static Handling of Java Reflection. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS '15)*. Springer.

[49] Yannis Smaragdakis and Martin Bravenboer. 2011. *Using Datalog for Fast and Easy Program Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 245–251. https://doi.org/10.1007/978-3-642-24206-9_14

[50] Square. [n. d.]. OkHttp – An HTTP & HTTP/2 client for Android and Java applications. http://square.github.io/okhttp/ Accessed: Jan. 28, 2018.

[51] Daniel Stenberg. [n. d.]. TLS in HTTP/2. https://daniel.haxx.se/blog/2015/03/06/tls-in-http2 Accessed: Jan. 28, 2018.

[52] DroidPlugin Team. 2016. DroidPluginTeam/DroidPlugin: A plugin framework on android, Run any third-party apk without installation, modification or repackage. https://github.com/DroidPluginTeam/DroidPlugin. Accessed: Jan. 28, 2018.

[53] E. Tilevich, W. R. Cook, and Y. Jiao. 2009. Explicit Batching for Distributed Objects. In *2009 29th IEEE International Conference on Distributed Computing Systems*. 543–552. https://doi.org/10.1109/ICDCS.2009.39

[54] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative research (CASCON '99)*. IBM Press, 125–135. http://dl.acm.org/citation.cfm?id=781995.782008

[55] Willem van Heiningen, Tim Brecht, and Steve MacDonald. 2006. Exploiting Dynamic Proxies in Middleware for Distributed, Parallel, and Mobile Java Applications. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06)*. IEEE Computer Society, Washington, DC, USA, 231–231.

[56] Robbie Vanbrabant. 2008. *Google Guice: Agile Lightweight Dependency Injection Framework*. APress.

[57] John Vlissides. 2001. GoF à la Java. *Java Report* (March 2001).

[58] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*. Springer, 97–118. https://doi.org/10.1007/11575467_8

[59] John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, New York, NY, USA, 131–144. https://doi.org/10.1145/996841.996859