Capability Safe Reflection for the Wyvern Language

Esther Wang
Carnegie Mellon University
estherw@alumni.cmu.edu

Jonathan Aldrich
Carnegie Mellon University
aldrich@cs.cmu.edu

Abstract

Reflection allows a program to examine and even modify itself, but its power can lead to violations of encapsulation and even security vulnerabilities. The Wyvern language leverages static types for encapsulation and provides security through an object capability model. We present a design for reflection in Wyvern which respects capability safety and type-based encapsulation. This is accomplished through a mirror-based design, with the addition of a mechanism to constrain the visible type of a reflected object. In this way, we ensure that the programmer cannot use reflection to violate basic encapsulation and security guarantees.

Keywords reflection, capability safety, mirrors, Wyvern

1. Introduction

In a system with multiple components, it is often the case that some components are less trustworthy than others. For example, a user-facing component is more likely to be subverted for a malicious cause. A desirable security guarantee for such cases is the *principle of least authority*, where a module is given only the resources it needs to perform its task. Then, if the module is subverted, it will have limited power to damage other areas of the system [2]. This principle is difficult to enforce in practice because many modern languages provide maximum permissions by default and rely on the programmer to restrict the authority of a module.

The object-capability model aims to address this difficulty. In the object-capability model, access privileges are managed through the use of capabilities, which are unforgeable keys to controlled resources. Rather than maximum privilege by default, a module only has the capabilities it is explicitly granted [3]. The advantages of object capabilities for security led to their integration in languages such as E [6] and Joe-E [5], and also in Wyvern.

A major focus of this paper is how to design a reflection system that is compatible with capability-based reasoning about security properties—i.e. one that is *capability safe*. We will demonstrate how unconstrained reflection would violate capability safety, and how our design provides a more secure approach. We will also consider how our design interacts with some of the other features of Wyvern, in particular, type-based encapsulation. Finally, we will describe the extent to which we provide reflective functionality.

Meta'16 October 30, 2016, Amsterdam, Netherlands

2. Wyvern

Wyvern is a statically typed and pure object oriented language which is designed for engineering web and mobile applications. In accordance with this aim, it is intended to be simple and safe by default. Wyvern also explores a combination of other properties, including support for type specific languages, capability safety, and delegation.

2.1 Type-based Encapsulation

Wyvern programs are structured as modules, with type, class, method, and value declarations within modules. Modules provide convenient features for namespace management (e.g. via importing other modules) but at their core are simply a way to define top-level objects. Modules include `type` declarations that define a structural type. Objects can be created with the keyword `new`. Information hiding is provided by type ascription; for example, if an object or module is ascribed a type, only the members in that type will be visible to clients of the object. Consider the following example:

```
type List =
    def append[T](object:T) : Unit
    def get[T](index:Integer) : T

type ImmutableList =
    def get[T](index:Integer) : T

def make[T]() : List =
    new
    /* ... Method body ... */
```

This simple snippet of Wyvern code demonstrates how types are defined in Wyvern. Since Wyvern is structurally typed, objects match any type that declares a subset of the object's methods and fields with appropriate signatures. `List` objects can be used as `ImmutableList` objects. However, the data and functions in an object are only visible if explicitly included in the corresponding type's signature. If a `List` object is ascribed to the `ImmutableList` type, the

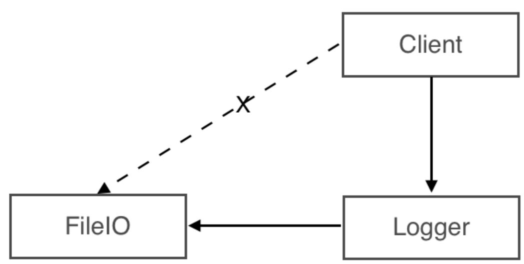

Figure 1: A logging system which uses capability safety.

`append` method ceases to be visible. This approach of using type ascription to hide members of an object that should not be visible provides more flexibility than explicit keywords such as `private`, `protected`, and `public`, since an object can be progressively ascribed increasingly general types as needed.

Now suppose the programmer had access to unrestricted reflection. Our encapsulation guarantees would no longer hold, because reflection allows the program to examine an object at run time. Type-based encapsulation is enforced during the typechecking phase, so if an access of a "private" field is constructed and executed at run time, the violation would not be detected.

2.2 Capability Safety

Wyvern enforces capability safety as part of the module system. Under the hood, capabilities are unforgeable keys to a resource.

Pure modules are declared with

```
module path/to/module
```

Modules with state or system privilege require capabilities and are declared with

```
resource module path/to/module
```

Types can also be declared with `resource type` to indicate that the object or module described by that type encapsulates a resource, such as access to a file or network connection, that we might want to track and protect.

Wyvern also considers any object with mutable state to be a resource to be tracked, because mutable state can be used to store a capability and then retrieve it later in some other part of the program; thus reasoning about mutable objects is important to reasoning about other capabilities [6]. Capabilites to system resources such as files are given to the `Main` module from the operating system, and other modules cannot use these capabilities unless `Main` passes them to the other modules [2].

Figure 1 illustrates a Client module which is permitted to use Logger to write to a specific set of log files. The main module (not shown) has initialized Logger with the capability for FileIO, allowing Logger to write to arbitrary files in the file system. Logger and FileIO are both resource modules, since both have permission to use a system resource. If the Client could use an unconstrained reflection library, it could access Logger's instance of FileIO and leverage that reference to perform filesystem operations that it would be unable to perform otherwise.

Capability safety is a major goal in Wyvern's design. Our design for reflection will preserve capability safety while interacting well with the other language features of Wyvern.

3. Prior Work

Prior work in statically typed and capability-safe languages includes Joe-E, a subset of Java. Joe-E excludes the reflective facilities that allow a program to bypass the access controls of object internals and break encapsulation. It also ensures that reflection does not provide any permission which was not already granted by the program code, and restricts propagation of capabilities by leveraging the static type system [5]. However, because Joe-E builds on Java core reflection, the reflective API does not satisfy the design principle of structural correspondence. Additionally, Joe-E leverages Java's relatively simple private declarations to distinguish fields that should not be accessible by clients. In contrast, Wyvern implements the more expressive approach of using type ascription to hide members of an object. As we will see, this expressivity makes it more challenging to design a reflection system for Wyvern that is compatible with capabilities.

We will be describing a mirror-based reflective architecture for Wyvern. Prior work with mirror-based reflective includes reflection in AmbientTalk [7] and work done in JavaScript to enforce language invariants with proxies [8]. Both of these approaches combined a mirror-based architecture with intercession, which our current design in Wyvern does not yet support. AmbientTalk and JavaScript are both dynamically typed, which permits greater flexibility in reflection as there are fewer invariants to maintain. However, this prior work suggests that in the future Wyvern may be able to support intercession in the future, once the basic library has been designed.

4. Design

4.1 Reflection

Our goal with regard to reflection was to create a library to perform computational reflection in Wyvern. *Computational reflection* is the ability of a program to perform computation on its internal structures through a casual connection between a system and its self-representation. A robotic arm, for instance, has a casual connection to the internal representation of its position: a change in one produces an analogous change in the other [4].

Features for computational reflection can be further categorized according to their purpose: introspection, self-modification, and intercession. *Introspection* is the ability of a program to examine itself, *self-modification* is the ability of program to modify itself, and *intercession* is the ability of a program to modify the programming language semantics [1]. When designing our API, we prioritized supporting introspection and self-modification over intercession. Introspection and self-modification will enable many common applications of reflection, such as dynamic patching of code, plugin support, and debuggers.

4.2 Mirrors

Bracha and Ungar made a survey of reflection, investigating specific uses of reflection and languages which support reflection. Their resulting paper presents three design principles and their justifications. These principles are encapsulation, stratification, and ontological correspondence. *Encapsulation* refers to the property that clients of a reflective API do not rely on any particular implementation of that API. For example, the Java Debug Interface (JDI) satisfies the encapsulation principle because the reflection interfaces it defines can have multiple implementations, and clients cannot distinguish which one is used. *Stratification* is the property of being separable, meaning that reflection does not impose costs when it is not being used. By this property, the meta level and the base level should have a clear boundary, and crossing that boundary should only be permitted through a limited set of operations. *Ontological correspondence* is the dual property of structural correspondence, a connection between meta level structures and the structure of the language being manipulated, and temporal correspondence, the association of a meta level API with either compile-time or runtime reflection [1]. In our design, we aimed to adhere to these three principles by constructing our API as a mirror-based architecture similar to the JDI.

4.3 Reflection in Wyvern

Reflection in Wyvern is organized as a collection of four modules:

```
resource module wyvern/reflection/full
resource module wyvern/reflection/limited
module wyvern/reflection/static
module wyvern/reflection/dynamic
```

The `full` and `limited` modules provide methods to perform reification, or the initialization of values from program structures. These two modules have the same signature, and contain the methods `reflectType[T]()` and `reflect[T](obj)` (see Fig. 2).

`reflectType[T]()` reflects on a type `T` and produces an object of type `Type`, which has methods for introspection on the given type `T`. `reflect[T](obj)` produces an object mirror of type `Object`, which has methods for introspection and self-modification on the object `obj`. The difference between `limited` and `full` is that `full.reflect` produces a mirror which reveals an object's full structure, including elements of the object which are not visible in the object's type at the time the mirror was obtained. `limited.reflect` enforces the guarantee that the object mirror returned will respect the signature of type parameter `T`.

```
resource module wyvern/reflection/full
import wyvern/reflection/static
import wyvern/reflection/dynamic

def reflect(object:T) : Object
def reflectType[T]() : Type
```

Figure 2: Signature of `full`

```
module wyvern/reflection/static

type Type =
    def equals(type:Type) : Boolean
    def fields() : List[Field]
    def fieldByName(name:String) : Field
    def methodByName(name:String) : Method
    def methods() : List[Method]
    def name() : String

type Method =
    def arguments() : List[Variable]
    def equals(method:Method) : Boolean
    def name() : String
    def returnType() : Type

type Variable =
    def equals(variable:Variable) : Boolean
    def name() : String
    def typeOf() : Type

type Field =
    def equals(field:Field) : Boolean
    def name() : String
    def typeOf() : Type
```

Figure 3: Signature of `static`

The `static` module contains the types `Type`, `Method`, `Variable`, and `Field`, which represent their corresponding program structures (see Fig. 3). These types are useful for introspecting on structures in code, in other words, static elements of the program. A `Type` mirror obtained from `reflectType` is a purely functional, as are the mirrors for methods, variables and fields; thus they need not be capabilities and can be used freely in the program.

The `dynamic` module contains the type `Object`. This type includes methods for introspecting and modifying the state of the object (see Fig. 4). Since the underlying object may be a capability that contains mutable state or enables access to some system resource, the `Object` mirror must also be treated as a capability. This makes sense because the mir-

```
module wyvern/reflection/dynamic
import wyvern/reflection/static

resource type Object =
    def equals(object:Object) : Boolean
    def get(f:Field) : Object
    def invoke (m:Method, \
        args:List[Object]) : Object
    def set(f:Field, value:Object) : Unit
    def typeOf() : Type
    def viewAtType(t:Type) : Object
```

Figure 4: Signature of `dynamic`

ror type `Object` contains the `set` method for mutating the reflected object's state and the `invoke` method for invoking methods on the reflected object.

By creating distinct types for each program structure, we satisfy structural correspondence. Temporal correspondence is satisfied through the separation of the types into different modules: `static` for for reflection on compile-time structure of the program and `dynamic` for reflection on run-time objects.

Mirror architectures such as this one are commonly adopted because they help satisfy the principle of encapsulation. In our case, the types such as `Object`, `Type`, `Method`, etc. defined in the reflection modules are structural types that can be implemented by different library providers, while hiding the specific details of the reflective implementation. Our modules provide one implementation via the `reflect` and `reflectType` methods, but others, such as proxies representing objects in remote virtual machines, are possible.

Lastly, stratification is satisfied because meta-level program elements are contained in separate but parallel objects to the base-level elements. Base-level objects cannot directly reference the corresponding meta-level object. Only by using the generator methods `reflect` and `reflectType` can the metal level be reached from the base level, so the meta-level functionality is clearly separated from the base-level.

5. Examples

Reflection produces a *mirror object* that ascribes to a mirror interface and represents an object at the meta level. An object mirror is casually connected to the object it reflects, and can be used to indirectly affect the original object. For example, an object's method can be invoked by acquiring the mirror of that object and calling the `invoke` method of the object mirror. Type mirrors are similar, but provide only functionality for observing the type that was reflected.

```
val listObj:Object = reflect[List]( \
    List.make().append(1))
val listType:Type = reflectType[List]()
listType.name() // ``List''
val getMethod:Method = listType.methodByName( \
    ``get'')
listObj.invoke(getMethod, \
    List.make().add(0)) // 1
```

This code is equivalent to creating an instance of a list containing 1, and invoking `get(0)` on that instance. The type mirror is being used to view the name of the type, and to get a method which permits access to the object mirror. It is easy to see from the API above which features can be examined.

`reflectType` produces a type mirror and permits only introspection on the static characteristics of a type. `reflect` produces an object mirror and allows some self-modification in addition to introspection on the dynamic characteristics of a specific object. A type mirror can be produced from an object mirror, but that mirror might not be equivalent to the type mirror produced from `reflectType`. The following example illustrates this relationship.

```
val list:ImmutableList = \
    List.make().append(1)
val listType:Type = \
    reflectType[ImmutableList]()

val listObj:Object = \
    reflect[ImmutableList](list)
val listType2:Type = listObj.typeOf()
listType2.equals(listType) // false
```

The last line evaluates to false because even though `myList` was ascribed to the `ImmutableList` type, its dynamic type is still `List`, so the type mirror produced will not be the same as the type mirror which only reflects `ImmutableList`.

6. Safety

The `Object` type includes a method called `viewAtType`. This method hides elements of the dynamic type which are not visible in the type argument given. In the following example, `viewAtType` is used to generate a mirror of an immutable list from the mirror of a mutable list. In the immutable list mirror, the `append` method which was accessible in the list mirror is no longer visible or invokable.

```
val list:List = List.make().append(1)
val listObj:Object = reflect[List](list)
val immutableListType:Type = reflectType \
    [ImmutableList]()
val immutableListObj:Object = listObj \
    .viewAtType(immutableListType)
```

Note that the argument of `object.viewAtType` must be an instance of `Type` representing a base level type which is a subtype of the type of the base level object for `object`. In the example given,

```
resource module wyvern/reflection/limited
require wyvern/reflection/full

def reflect[T](object:T): Object =
    val objMirror = full.reflect[T]( \
        object)
    val typeMirror = full.reflectType[T]()
    objMirror.viewAtType(typeMirror)

def reflectType[T](): Type =
    full.reflectType[T]()
```

Figure 5: Implementation of `limited`

```
immutableListObj.viewAtType( \
    reflectType[List]())
```

would be invalid, since `List` is not a subtype of `ImmutableList`. This is significant because it ensures that once information in an object mirror is hidden, it can be used by an untrusted module. The untrusted module cannot discover the hidden information even if it gains access to an instance of `Type` which reveals full information about the base level object.

Since `viewAtType` is used to hide information, it can be used to maintain type-based encapsulation and capability safety in reflection. `limited` uses `full` reflection, but automatically calls `viewAtType` using the type and object provided to `reflect`. Because of this, the mirrors returned will only represent the static type provided, so a module cannot use `limited` to discover type information that is not already visible. Figure 5 shows the implementation of `limited`. An untrusted module would be given the capability for limited rather than full reflection.

6.1 Encapsulation

To verify that reflection satisfies encapsulation, a programmer would typically need to reason through the reflective code and observe whether any particular command violates this property. In our design, full reflection might still be used to violate encapsulation. However, the `limited` module provides a simple option for maintaining type-based encapsulation guarantees in reflective code: if the `limited` module is used, encapsulation is preserved because the type system will not allow access to any member which is not in the reflected object's type at the time the mirror was created. Even if a program were to apply limited reflection on the mirror, no further information would be exposed than what was already in the type definition of the mirror. This prevents the program from gaining access to the base level structure being reflected.

Full reflection provides reflective power that cannot be given while maintaining the encapsulation guarantees we provide in Wyvern. But because a capability is required to access full reflection, it is straightforward to determine the areas of code at risk for violations of encapsulation.

6.2 Capability Safety

Limited reflection is as central to preserving capability safety as to preserving type-based encapsulation in Wyvern. Consider the example given earlier involving the Logger and FileIO modules (see Figure 1). If Client is only given the ability to perform limited reflection, it will be able to use reflection without being able to access the field in which Logger stores its instance of FileIO.

Without the `limited` module, the only way for Client to use reflective facilities would be for a mirror to be passed in to Client from some module which does have the capability for full reflection. In this example, an `Object` reflecting the Logger instance might be passed to Client from another module. However, because the environment in which the `Object` was produced would be one with higher privilege than the destination of the mirror objects produced, there would be greater risk. When the `Object` for Logger is passed to Client, it must have had the `viewAtType` method called on it with the Logger signature. Otherwise, Client would receive information that was previously hidden. But if the more trusted module requires the full mirror of Logger (one which has not had `viewAtType` called), it would need to manage two versions of the Logger mirror and use the appropriate one as needed.

With limited reflection, Client is able to use reflection directly and no longer needs a more trusted module to pass mirrors to it. This reduces the need for passing mirrors and potentially leaking capabilities from one module to another. It also migrates the responsibility of managing mirrors to the module which uses them, which is more intuitive than having a separate, though more trusted, module be responsible for the task. Limited reflection is a safe capability to grant to Client because `Object` mirrors produced from using `limited` reflection in the Client will only reveal information in the Logger type signature, since `viewAtType` is automatically called from within `limited`. If the Logger type signature exposes a capability, then Client will be able to access that capability regardless of whether limited reflection is granted. If the Logger type signature is safe, then we know that limited reflection is safe to use on instances of Logger in Client. We can see that limited reflection does not affect whether a program is safe.

The risk introduced by passing mirrors remains, however, and programs might still be written which follow this pattern. Passing `Object` mirrors between modules can be dangerous for capability safety, since these mirrors give direct access to the internals of an object. For this reason, `Object` is a resource type which requires a capability – not every module should be given permission to receive and manipulate an `Object`.

`Type` mirrors can be passed without a capability because they are significantly less risky; a client that has a `Type` mirror can look at what methods are available, but it cannot invoke any of them without having an object mirror in which those methods are visible. Therefore, to verify that Client can safely use reflection, it is sufficient for the programmer to verify that the Logger type does not expose any dangerous capabilities, either as base-level objects or as mirrors.

7. Reflective Ability

In this section we will discuss the functionality of our API in terms of its ability to perform introspection and self-modification. The third reflective ability, intercession, was seen as less necessary for common applications of reflection, and is not supported by our API.

7.1 Introspection

Our API for reflection is intended to have good support for introspection. Because Wyvern is object oriented, supporting introspection on objects provides enough functionality for the majority of use cases. This allowed us to keep the API simple.

Introspection is supported to the extent that structural correspondence is satisfied. The API includes methods to examine all structures that are represented, but the set of representations is not complete. Currently, only objects, methods, and their components are supported. A `Variable` type is defined to represent method arguments, but method bodies cannot be examined because there is no representation for individual lines of code.

Introspection is supported more completely with regard to the compile-time state of a structure. For each of the available structures, static features such as name and type are readily attainable. However, the values of variables are only defined at run time, and cannot be observed with this design of reflection because supporting this would detract from the performance of the language. On the other hand, the presence of the `full` module in addition to the `limited` module provides greater visibility of run-time state such as hidden fields. This information is useful in applications of reflection to debugging and logging.

7.2 Self-modification

With the current API design, a program is able to modify its own data, but not its procedures. The program's behavior can be altered by setting values for object fields which produce the desired execution.

Though simple, this degree of self-modification is comparable to mainstream reflective architectures. Java core reflection allows modification of fields, but not methods or other procedures. The Java Debug Interface, because it is used mainly for debugging purposes, has extensive support for introspection but similarly limited support for self-modification.

8. Future Work

Future work is available in several areas, including verification of safety, functionality, and evaluation. A formal verification of the safety of our design, particularly its ability to maintain encapsulation and thus capability safety, would provide insight into how this protection might be implemented in other reflective architectures. There is also a great deal of functionality that we can consider adding. Being able to reflect on expressions or declarations at run time will be necessary to implement a debugger, and the ability to convert an object mirror into a standard object (absorption) would be useful for constructing objects from scratch. Lastly, we must evaluate our library's utility in real applications. Eventually, it will be feasible to use Wyvern for implementing very realistic applications, including ones which are most easily implemented using reflection. For natural use cases, reflective code should be written and the performance analyzed. This will also permit us to observe how user-friendly this design is.

9. Conclusion

The combination of reflection and static typing is uncommon, but growing more popular as the advantages of reflection become increasingly known. However, to our knowledge, no prior language has attempted a combination of reflection, static typing, and capability safety. Our design takes the novel approach of providing `viewAtType` for customizing the visibility of type members. This allowed us to implement the `limited` module, which respects type-based encapsulation and capability safety. From this point, there is room for reflection in Wyvern to become a powerful and less dangerous tool than reflection has been in the past.

Acknowledgments

This work was supported by the U.S. National Security Agency lablet contract #H98230-14-C-0140.

References

[1] G. Bracha and D. Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA*, 2004.

[2] D. Kurilova, A. Potanin, and J. Aldrich. Modules in Wyvern: Advanced control over security and privacy. In *Symposium and Bootcamp on the Science of Security (HotSoS)*, 2016.

[3] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, 1984.

[4] P. Maes. Concepts and experiments in computational reflection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1987.

[5] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *Network and Distributed System Security Symposium*, 2010.

[6] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, May

[7] S. Mostinckx, T. Van Cutsem, S. Timbermont, E. Gonzalez Boix, E. Tanter, and W. De Meuter. Mirror-based reflection in AmbientTalk. *Softw. Pract. Exper.*, 39(7):661–699, May 2009.

[8] T. Van Cutsem and M. S. Miller. Trustworthy proxies: Virtualizing objects with invariants. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 154–178, 2013.