A Capability-Based Module System for Authority Control*†

Darya Melicher[1], Yangqingwei Shi[2], Alex Potanin[3], and Jonathan Aldrich[4]

1 **Carnegie Mellon University, Pittsburgh, PA, USA**
2 **Carnegie Mellon University, Pittsburgh, PA, USA**
3 **Victoria University of Wellington, Wellington, New Zealand**
4 **Carnegie Mellon University, Pittsburgh, PA, USA**

Abstract

The principle of least authority states that each component of the system should be given authority to access only the information and resources that it needs for its operation. This principle is fundamental to the secure design of software systems, as it helps to limit an application's attack surface and to isolate vulnerabilities and faults. Unfortunately, current programming languages do not provide adequate help in controlling the authority of application modules, an issue that is particularly acute in the case of untrusted third-party extensions.

In this paper, we present a language design that facilitates controlling the authority granted to each application module. The key technical novelty of our approach is that modules are first-class, statically typed capabilities. First-class modules are essentially objects, and so we formalize our module system by translation into an object calculus and prove that the core calculus is type-safe and authority-safe. Unlike prior formalizations, our work defines authority non-transitively, allowing engineers to reason about software designs that use wrappers to provide an attenuated version of a more powerful capability.

Our approach allows developers to determine a module's authority by examining the capabilities passed as module arguments when the module is created, or delegated to the module later during execution. The type system facilitates this by identifying which objects provide capabilities to sensitive resources, and by enabling security architects to examine the capabilities passed into and out of a module based only on the module's interface, without needing to examine the module's implementation code. An implementation of the module system and illustrative examples in the Wyvern programming language suggest that our approach can be a practical way to control module authority.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases Language-based security, capabilities, authority, modules

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.20

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.3.2.2`

* This work was supported in part by NSA lablet contract #H98230-14-C-0140 and by Oracle Labs Australia.
† A technical report containing a complete version of the formalism is also available [21].

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 20; pp. 20:1–20:27

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


1 Introduction

The principle of least authority [34] is a fundamental technique for designing secure software systems. It states that each component of a system must be able to access only the information and resources that it needs for operation and nothing more. For example, if an application module needs to append an entry to an application log, the module should not also be able to access the whole file system. This is important for any software system that divides its code into a trusted code base [33] and untrusted peripheral code, as in it, trusted code could run directly alongside untrusted code. Common examples of such software systems are extensible applications, which allow enriching their functionality with third-party extensions (also called plug-ins, add-ins, and add-ons), and large software systems, in which some developers may lack the expertise to write secure- or privacy-compliant code and thus should have a limited ability to access system resources in their code. Enforcing the principle of least authority helps to limit the attack surface of a software system and to isolate vulnerabilities and faults. However, current programming languages do not provide adequate control over the authority of untrusted modules [3, 38], and non-linguistic approaches also fall short in controlling authority [4, 18, 35, 42].

Application security becomes even more challenging if an application uses code-loading facilities or advanced module systems, which allow modules to be dynamically loaded and manipulated at runtime. In such cases, an application has extra implementation flexibility and may decide what modules to use at runtime, e.g., responding to user configuration or the environment in which the application is run. On the other hand, untrusted modules may get access to crucial application modules that they do not explicitly import via global variables or method calls. For example, although a third-party extension may import only the logging module and not the file I/O module, the extension could receive an instance of the file I/O module via a method call as an argument or as a return value. Dynamic module loading can be modeled as first-class modules, i.e., modules that are treated like objects and can be instantiated, stored, passed as an argument, returned from a function, etc. However, in a conventional programming language featuring first-class modules (e.g., Newspeak [2], Scala [31], and Grace [15]), it is difficult to track and control modules accesses.

In this paper,[1] we present a module system that helps software developers to control the authority of code by treating modules as first-class, statically typed *capabilities* [5]—i.e., communicable but unforgeable references allowing to access a resource—and making access to security- and privacy-related modules capability-protected, in the style of the E programming language [25]. Specifically, if module A wants to access module B, A may do so only if A possesses an appropriate capability. Leveraging capabilities allows us to support first-class modules (e.g., representing dynamic module loading, linking, and instantiation) while still providing a strong model for reasoning about application security and module isolation.

The design of the module system and the accompanying type system of the language simplify reasoning about module authority. To determine the authority of a module via capability-based reasoning, a security expert or a system architect must understand what capabilities the module can access. Since our module system is statically typed (in contrast to Newspeak [2], which provides a capability-safe but dynamically typed module system), the architect needs to examine only the module's interface and the interfaces of its imports and does not need to examine the code of any module. For example, suppose an application

[1] A one-paragraph poster abstract for this work appeared elsewhere [16].

has a trusted logger module that legitimately imports a module for file I/O, and the logger module is the only module imported by an extension. To ensure that the extension does not have access to the file I/O module, except as mediated (i.e., attenuated [25]) by the logger module, it is sufficient to verify that the extension does not import the file I/O module directly and that the extension cannot get direct access to a file I/O capability by calling the logger's methods. The first condition is a syntactic check, and the second condition requires inspecting only the logger's interface, e.g., to ensure that none of the methods in the interface return a file object (or indeed the file I/O module itself, since modules are first-class). Our module system enjoys an *authority safety* property that statically guarantees that the above two possibilities are all a developer has to consider. This is in contrast to conventional languages and module systems, in which global variables, unrestricted reflection, arbitrary downcasts, and other "back doors" make capability-based reasoning infeasible.

Our work has four central contributions. The first contribution is the design of a module system that supports first-class modules (cf. Newspeak, Scala, and Grace) and is capability-safe [22, 25]. Our approach forbids global state, instead requiring each module to take the resources it needs as parameters, which ensures that modules do not carry ambient authority [40] (similar to Newspeak, but in contrast to Scala and Grace). For practical purposes, our module system supports module-local state and does not restrict the imports of non-state-bearing modules (in contrast to Newspeak).

The second contribution is a type system that distinguishes modules and objects that act as capabilities to access sensitive resources, from modules and objects that are purely functional computation or store immutable data. This design makes it easy for an architect to focus on the parts of an interface that are relevant to the authority of a module. Overall, the type system allows developers to determine the authority of a module at compile time by examining only the interfaces of the module and the modules it imports, without having to look at the implementation of the involved modules.

The third contribution of our work is the formalization of authority control in the designed module system, in which we introduce a novel, *non-transitive* definition of authority that explicitly accounts for attenuated authority (e.g., as in the logger example above). We also introduce a definition of authority safety and formally prove the designed system authority-safe. Our result contrasts prior, transitive definitions of authority safety that cannot account for authority attenuation [7, 20].

The final contribution is the implementation of the designed module system in Wyvern, a statically typed, capability-safe, object-oriented programming language [29], demonstrating the feasibility and practicality of the proposed approach.

We start the paper by describing the Wyvern module system from the perspective of a software developer in Section 2 and present the formalization of the designed module system in Section 3. We continue by introducing the definition of authority safety, state authority-related properties of Wyvern's module system, and prove Wyvern authority-safe in Section 4. Then, we report on the implementation of the Wyvern module system and on the limitations of our approach in Sections 5 and 6 respectively. Finally, we compare our approach to other language-based approaches in Section 7 and conclude in Section 8.

2 Wyvern Module System

In Wyvern, modules have several features distinguishing its module system from others:

- Modules are first-class, i.e., they are treated as objects and can be instantiated, stored, passed as arguments into methods, and returned from methods.

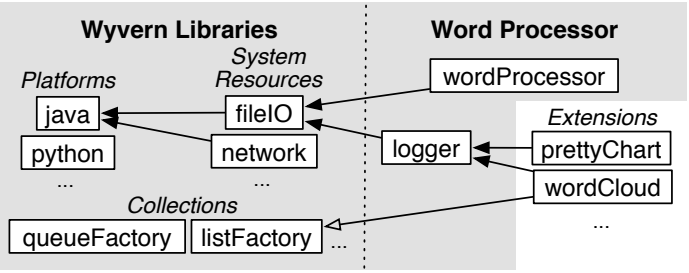

Figure 1 A module import diagram of a word processor application used in code examples. The boxes represent modules, and the arrows represent module imports. If an arrow goes from module A to module B, A imports B. The arrows with black arrowheads correspond to importing resource modules; the arrow with an unfilled arrowhead corresponds to importing a pure module. The dark background delineates the trusted code base.

- Modules are treated as capabilities in the style of [1], i.e., we unify the notion of having a reference to a module with the notion of having a capability to access that module. If a module can access another module, we say that the former module has a capability to use the latter module. (The same is true for objects.)
- Modules are divided into two categories: *resource modules*, i.e., security- or privacy-related modules (system resources, modules containing application data, or state-bearing modules), and *pure modules*, i.e., non-state-bearing utility modules.

To illustrate our approach, let us consider a sample application that allows third-party extensions. Figure 1 shows a module import diagram of a word processor application, similar to OpenOffice or MS Word, that extends its feature set by allowing third-party extensions. The vertical dotted line represents a virtual border between standard language-provided libraries and the word processor code. The boxes represent modules, which are clustered according to their conceptual type. The arrows represent module imports. If an arrow goes from module A to module B, module A imports module B. The arrows with black arrowheads correspond to importing resource modules, while the arrow with an unfilled arrowhead corresponds to importing a pure module. Being able to import a resource module, which corresponds to arrows with black arrowheads on the diagram, is equivalent to having unconditional control and thus authority over the imported module.

Wyvern provides a number of standard libraries: *Collections* refer to a set of pure modules that provide implementations of basic functionality, e.g., list and queue factories. *System Resources* refers to a set of language-provided modules that implement system-level functionality, e.g., file and network access. *Platforms* refer to the modules that implement the Wyvern back end. Platforms and system resources may be used to subvert the word processor, and thus access to them requires the possession of special capabilities.

The word processor system consists of core modules, which are considered trusted, and extension modules (marked so on the diagram), which are provided by third parties and considered untrusted. The diagram presents only a subset of modules of the word processor's core that are used in our examples: the `wordProcessor` module is the main module of the word processor, and the `logger` module provides a logging service and can be used by multiple word processor's modules.

We use the word processor example to introduce Wyvern's two types of modules—resource modules and pure modules—and to show how one can determine a module's authority. For brevity, all module definitions and their types in code examples are put together; however, in reality, each module definition and type resides in a separate file.

2.1 Threat Model

Our approach focuses on ensuring the principle of least authority and assumes a software system that is divided into a trusted code base [33] and untrusted peripheral code. All the code in the trusted code base is vetted by security or privacy experts. The untrusted code may be modules within the same code base or third-party extensions. Our module system aims at giving the untrusted modules the least possible authority over security- and privacy-related modules of the trusted code base, thus minimizing the possible damage if the untrusted code is malicious or vulnerable. The authority given to untrusted modules is scrutinized, but their code is not examined, except for their interfaces.

The following two common scenarios fit our threat model:

Malicious third-party code. In an extensible software system, an attacker writes a malicious extension and tricks the user into loading it into the system. We wish to limit the damage that such an extension can do.

Fallible in-house code. In a large software system, a trusted core is written by security experts, who have the knowledge to securely access sensitive resources, e.g., the network and file system, while the rest of the system is written by non-security experts, who may introduce vulnerabilities that could be exploited by an attacker. We wish to limit the damage that may result from exploits to the non-core parts of the system.

In both scenarios, modules written by less trusted parties can access security- and privacy-related modules, e.g., system resources, only via safe interfaces written by experts. We leverage module system capabilities to ensure that attackers cannot do anything to security- or privacy-critical resources beyond what is permitted by the safe interfaces. Vulnerabilities inside the trusted code base are explicitly outside of our security model. We discuss the limitations of this model more in Section 6.

The word processor example is presented as the first scenario, but it can be adapted to the second scenario as well. In Figure 1, the trusted code base is marked by the dark background.

2.2 Resource Modules

Resource modules are defined as modules that:

1. encapsulate system resources (e.g., `java` and `fileIO`),
2. use other resource modules (e.g., `wordProcessor` and `logger`), or
3. contain mutable state (e.g., `wordProcessor`).

A module is a resource if it has one or more of these characteristics. For example, the `wordProcessor` module is a resource module because it imports the system resource `fileIO` and has state (details upcoming). It is important for state-bearing modules to be resources, as they may contain private application data and also may facilitate communication between modules that import them, potentially allowing illegal sharing of capabilities.

Figure 2 presents a code example with several resource modules and types. By convention, module names start with lowercase letters, while type names are capitalized. The code snippet starts with the definition of the main module of the word processor application, `wordProcessor`, which is a resource module. The module imports a module instance of a resource type `FileIO` (defined on lines 5–7) via the argument passing mechanism. In Wyvern, each resource module is an ML-style *functor* [19], i.e., it is a function that accepts one or more arguments, each of which is a module instance of a required type, and produces a module instance as a result. In the case of `wordProcessor`, the module functor accepts a module instance of type `FileIO` and returns an instance of the `wordProcessor` module.

```
1  module def wordProcessor(io : FileIO) : WordProcessor
2    import logger
3    var log : Logger = logger(io)
4    ...
5  resource type FileIO
6    def read(file : File) : String
7    ...
8  resource type Logger
9    def appendToLog(entry : String) : Unit
10 module def logger(io : FileIO) : Logger
11   def appendToLog(entry : String) : Boolean
12     io.open("~/log.txt").append(entry)
```

Figure 2 A Wyvern code example demonstrating resource modules, their imports, and instantiations.

`FileIO` is a resource type that gives access to the file system, and since `wordProcessor` imports an instance of this type, `wordProcessor` is a resource module too. To access a resource module of the `FileIO` type, `wordProcessor` needs to have an appropriate capability. The capability must be passed into the `wordProcessor` module on its instantiation by either another module or top-level code.

The `wordProcessor` module instantiates the `logger` module (defined on lines 8–12) by, first, importing the definition of the `logger` module using the `import` keyword and then calling the imported `logger` functor definition with appropriate arguments to get an instance of the `logger` module. (Technically, `logger(io)` is syntactic sugar for `logger.apply(io)`, where `apply()` is a default method called on a resource module to instantiate it.) The argument that `logger` requires is a module instance of the `FileIO` type, and by passing in `io`, `wordProcessor` gives `logger` the capability to use the module instance of the `FileIO` type it received on instantiation. The created instance of `logger` is immediately assigned to a local variable `log`, which may be used later in the `wordProcessor`'s code. Note that `wordProcessor` *imports* a module instance of the `FileIO` type, but it *instantiates*, i.e., creates a local instance of, the `logger` module. Generally, any resource module can instantiate other resource modules from its initialization block and even provide them with access to resource modules to which it itself has access. Since `logger` is a resource module, instantiating it creates a capability for it, which, in this case, belongs to the `wordProcessor` module.

Alternatively, if `wordProcessor` did not want to provide `logger` access to the file system, `wordProcessor` could create and pass in a dummy module of type `FileIO` as follows:

```
module def wordProcessor(io : FileIO) : WordProcessor
  import logger
  var dio : FileIO = dummyIO
  var log : Logger = logger(dio)
  ...
```

This would disallow the `logger` module from having any access to the file system.

To run the program, the top-level code is as follows:

```
platform java
import fileIO
import wordProcessor
let io = fileIO(java) in
  let wp = wordProcessor(io) in ...
```

First, the back end to be used is specified using the `platform` keyword. This keyword can appear only on the top level and is used to create a resource module instance representing

the back-end implementation. Then, the definitions of the `fileIO` and `wordProcessor` module functors are imported, and the two modules are instantiated receiving the arguments they require. The two newly created module instances are assigned to two variables in two nested `let` constructs and can be used in the rest of the code contained in the inner `let`'s body.

The top-level code exercises high-level control over accesses to resource modules, performing two important functions. First, it instantiates resource modules, implicitly creating capabilities that allow using the instantiated modules. Second, it grants module access permissions (conceptually, in the Newspeak style [2]; syntactically, in the ML-functor style [19]): the instantiated modules (and implicit capabilities to use them) are passed as arguments to authorized modules.

For brevity, the top level code can be shortened as follows:

```
require fileIO : FileIO
import wordProcessor
let wp = wordProcessor(fileIO) in ...
```

Here we use syntactic sugar (the keyword `require`) for specifying the platform (the default platform is chosen), and importing the functor definition of and instantiating the `fileIO` module. This syntactic sugar can be used for resource modules that import only the resource module representing the back-end implementation, and is usually used for short programs, e.g., "Hello, World!"

Notably, two modules may share a module instance and potentially use it for communication. For example, if both extensions `prettyChart` and `wordCloud` would like to append to the word processor's log, they may share one instance of the `logger` module:

```
require fileIO
import wordCloud
import prettyChart
let log = logger(fileIO) in
  let wCloud = wordCloud(log) in
    let pChart = prettyChart(log) in ...
```

This makes the language more flexible and simplifies certain implementation tasks.

2.3 Pure Modules

The definition of a pure module is the opposite from the definition of a resource module. Pure modules are those modules that:

1. do not encompass system resources,
2. do not import any resource module instances,
3. do not contain or transitively reference any mutable state,
4. have no side effects.

For a module to be pure, all of these conditions must be satisfied. The third condition has a caveat: The prohibition is on whether a module and its functions *capture* state, not whether they *affect* it. Functions defined in a pure module may have side effects on state, but only if the state in question is passed in as an argument or created within the function itself.

Thus pure modules are harmless from the security perspective, and for more convenience, in Wyvern, any module can import any pure module.

Figure 3 shows an example of a pure module and how it can be imported. The `listFactory` module is the implementation of a list factory and belongs to the standard Wyvern library. It does not contain mutable state, but only creates new lists, and therefore is a pure module.

```
module listFactory : ListFactory
  def create() : List
    ...
module def wordCloud(log : Logger) : WordCloud
  import wyvern : listFactory as list
  var words : List = list.create()
  ...
```

Figure 3 A Wyvern code example demonstrating a pure module and its import.

```
module def wordCloud(log : Logger, list : ListFactory) : WordCloud
  var words : List = list.create()
  ...
// top level
require fileIO
import wordCloud
import listFactory as list
let log = logger(fileIO) in
  let wCloud = wordCloud(log, list) in ...
```

Figure 4 A Wyvern code example demonstrating how a pure module can be passed to a module as an argument.

In Wyvern, pure modules are *not* functors, and a module that imports a pure module receives an instance of the pure module.

The `wordCloud` module is a third-party extension module that creates a word cloud—an image composed of words used in a text passage, in which the size of each word indicates its frequency—and pastes it into a word processor document. The `wordCloud` module uses a list to store the words it operates on and therefore imports the `listFactory` module using the `import` keyword. Since, for pure modules, the import statement produces a module instance, it can be immediately assigned to a local variable using the `as` keyword. The import of `listFactory` by `wordCloud` is invisible to the module or top-level code that instantiates the `wordCloud` module.

Wyvern's module system includes additional features that are not essential to the capability model, but are useful for software engineering purposes. For example, pure modules can be assigned a resource module type, allowing them to be treated as resource modules, e.g., for testing purposes. Furthermore, we could make the `wordCloud` module generic in the particular implementation of lists that it uses by adding a pure module parameter of type `ListFactory`, as shown in Figure 4. We do not discuss these features further as they do not impact capability-based reasoning.

2.4 Authority Analysis

As stated in our threat model, we are concerned with the authority granted to third-party extensions, as well as minimizing access to system resources by all application modules. In this section, we demonstrate how an architect can verify that the authority of the modules in the word processor application matches the authority shown in Figure 5. (In Section 4, we will generalize authority to arbitrary objects and provide a formal definition.)

Since access to resources is mediated by modules, we can represent the authority of a given module as the set of resource modules it can access. In Figure 5, if an arrow goes from module A to module B, A imports B and has authority over B. If an arrow

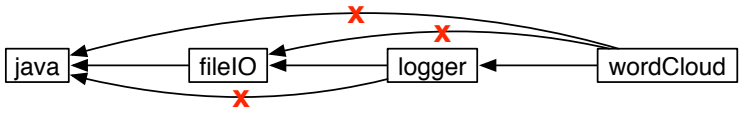

Figure 5 Authority distribution between `fileIO`, `logger`, and `wordCloud`. If an arrow goes from module A to module B, A has authority over B. Crosses on arrows mean that such authority is not granted. In Wyvern, authority is non-transitive.

is crossed, it means that such authority is not granted. Thus, `wordCloud` has authority to access `logger`, which in turn has authority to access `fileIO`, which ultimately has access to the `java` foreign function interface module. We want to verify that the transitive extension of these authority relationships does not hold, e.g., the `wordCloud` module does not have direct authority to do the file I/O operations supported by the `fileIO` module. In effect, we are verifying that `wordCloud` gets only an attenuated capability to do file I/O: it can perform the logging operations supported by the `logger` module, but nothing more. This facilitates a defense in depth strategy: if an attacker controls the `wordCloud` module and somehow subverts the `logger` module to get a `fileIO` capability, since `fileIO` itself attenuates the `java` foreign function interface capability, the attacker can do file I/O but cannot make arbitrary system calls supported by the Java standard library.

To verify that authority is property attenuated (thereby mitigating the attack mentioned above by ensuring that `wordCloud` cannot get a `fileIO` capability), we need to check that the `fileIO` module is properly encapsulated by the `logger` module, and that the `logger` module provides operations that are restricted appropriately to the intended semantics of logging and cannot be used to do arbitrary file I/O.

We can check encapsulation by inspecting the interface of `wordCloud` as well as the interfaces of the modules it imports: `Logger` and `ListFactory`. Since `ListFactory` is not a resource module, we do not have to look any further at its interface. (Note that, in contrast to dynamically typed, capability-safe languages such as E or Newspeak, Wyvern's type system aids our inspection here.) We inspect the interface of `logger` (lines 8–9 in Figure 2) and immediately observe that none of the types in `logger`'s interface are resource types. Thus, we verify that `logger` cannot leak a reference to the `fileIO` module that it uses internally—again, using only the type of the `logger` module, not its implementation.

Of course, encapsulation by itself is not enough: if `logger` provided the same operations as `fileIO`, it would essentially provide the same authority despite the actual `fileIO` being encapsulated. To this end, we check that `logger` attenuates the authority of `fileIO` and that `logger` can only do logging, instead of arbitrary file operations, by looking at the implementation of `logger`. Notably, this inspection is localized: we can use interfaces to reason about where capabilities can reach and then check the code that uses those capabilities to ensure it enforces the proper invariants. We do not have to inspect any code if we can show that the capability we are reasoning about does not reach that code. In this case, if we do inspect `logger` it is easy to see that it invokes `open()` and `append()` on a specific file, which is characteristic of the intended logging functionality.

This process would be more complicated in a language that is not capability-safe or even in a language that is capability-safe but does not have Wyvern's static typing support. In a language that is not statically typed, we could not so quickly exclude the possibility that a capability of interest is hidden in `ListFactory`, nor could we be sure that we know all of the operations available on an object unless we enforce that dynamically by imposing a wrapper. In a language that is not capability-safe, there is much more to worry about: `wordCloud` could

$$
\begin{array}{lcl}
p & ::= & \overline{md}\ \texttt{platform}\ x\ \ \overline{i}\ e \\
md & ::= & h\ \overline{i}\ \overline{d} \\
h & ::= & \texttt{module}\ x : \tau \\
 & | & \texttt{module def}\ x(\overline{y : \tau}) : \tau \\
i & ::= & \texttt{import}\ x\ [\texttt{as}\ y] \\
d & ::= & \texttt{def}\ m(\overline{x : \tau}) : \tau = e \\
 & | & \texttt{var}\ f : \tau = x
\end{array}
\qquad
\begin{array}{lcl}
e & ::= & x \\
 & | & \texttt{new}_s(x \Rightarrow \overline{d}) \\
 & | & e.m(e) \\
 & | & e.f \\
 & | & e.f = e \\
 & | & \texttt{let}\ x = e\ \texttt{in}\ e \\
 & | & \texttt{bind}\ \overline{x = e}\ \texttt{in}\ e \\
s & ::= & \texttt{resource} \mid \texttt{pure}
\end{array}
$$

Figure 6 Wyvern's abstract grammar.

get access to `fileIO` by reading a global variable, a reference to a file object could be smuggled in an apparently innocent variable of type `Object` and then downcast to type `File`, or reflection could be used to extract a `fileIO` reference from within the `logger` object. However, these are not possible in Wyvern: Wyvern does not support arbitrary downcasts but only pattern matching in a hierarchy where the possible child types are known. In addition, Wyvern's capability-safe reflection mechanism respects type restrictions [41], so that reflection cannot be used to do anything other than invoke the public methods of `logger`. Thus, Wyvern's capability-safe module system along with its static types greatly simplify reasoning about the authority of modules.

3 Wyvern Syntax and Semantics

Although modules are at the heart of our work, they are not central to Wyvern's formal system. Inspired by the Wyvern core work [29], our modules are syntactic sugar on top of an object-oriented core language and are available for developers' convenience. We present the Wyvern formal system in the following order: first, we describe the abstract grammar for writing modules in Wyvern, then the object-oriented core language syntax and module translation into it, and finally, Wyvern's static and dynamic semantics. This precisely defines our design and lays the groundwork for the definition and proof of authority safety in Section 4.

3.1 Module Syntax

Wyvern's abstract grammar is shown in Figure 6. A Wyvern program consists of zero or more modules followed by the top-level code that includes specifying the back end used to run the program using the `platform` keyword, zero or more module imports, and an expression e. Each module consists of a module header h, a list of imports $\overline{i}$, and a list of declarations $\overline{d}$. Module headers can be one of two types depending on whether the module is a resource module or a pure module. If a module is pure, its header consists of the `module` keyword, a name x that uniquely identifies the module, and a module type τ. If a module is a resource module, its header consists of the `module` keyword, followed by the `def` keyword, which signifies that it is a functor, a name x, which uniquely identifies the module functor, a list of functor parameters and their types, and a functor return type τ.

The module-import syntax is used for importing instances of pure modules or module functors for resource modules, and consists of the `import` keyword followed by the module or functor name x. In the case of importing an instance of a pure module, for convenience, the instance can be renamed using the `as` keyword.

$$
\begin{array}{rcl}
e & ::= & x \\
 & | & \texttt{new}_s(x \Rightarrow \overline{d}) \\
 & | & e.m(e) \\
 & | & e.f \\
 & | & e.f = e \\
 & | & \texttt{bind}\ x = e\ \texttt{in}\ e \\
 & | & l \\
 & | & l.m(l) \rhd e \\
s & ::= & \texttt{resource} \mid \texttt{pure}
\end{array}
\qquad
\begin{array}{rcl}
d & ::= & \texttt{def}\ m(x:\tau):\tau = e \\
 & | & \texttt{var}\ f:\tau = x \\
 & | & \texttt{var}\ f:\tau = l \\
\tau & ::= & \{\overline{\sigma}\}_s \\
\sigma & ::= & \texttt{def}\ m(x:\tau):\tau \\
 & | & \texttt{var}\ f:\tau \\
\Gamma & ::= & \varnothing \mid \Gamma,\ x:\tau \\
\mu & ::= & \varnothing \mid \mu,\ l \mapsto \{x \Rightarrow \overline{d}\}_s \\
\Sigma & ::= & \varnothing \mid \Sigma,\ l:\tau
\end{array}
\qquad
\begin{array}{rcl}
E & ::= & [\,] \\
 & | & E.m(e) \\
 & | & l.m(E) \\
 & | & E.f \\
 & | & E.f = e \\
 & | & l.f = E \\
 & | & \texttt{bind}\ x = E\ \texttt{in}\ e \\
 & | & l.m(l) \rhd E
\end{array}
$$

Figure 7 Syntax of Wyvern's object-oriented core.

A module can contain declarations of two kinds: method declarations and variable declarations. Method declarations are specified using the `def` keyword followed by the method name m, a list of method parameters and their types, the method's return type τ, and the method body e. Variable declarations are specified using the keyword `var` followed by the variable name f, the variable type τ, and the value x. We restrict the form of the initialization expression to simplify translation into the core, but this is relaxed in our implementation.

Wyvern expressions are common for an object-oriented programming language and include: a variable, the `new` construct, a method call, a field access, a field assignment, and the `let` and `bind` constructs. The `new` construct carries a tag s that indicates whether the object being created is pure or is a resource, which is at the core of our formalization of authority control. It also contains a self reference x that is similar to a `this`, but provides more flexible naming, and is used for tracking the receiver (discussed in more detail later). Finally, the `new` construct accepts a list of declarations $\overline{d}$. The `bind` construct is similar to a `let` with the difference that expressions in its body can access only the variables defined in it and nothing outside it (one can think of it as a Scala's spore [23] or an AmbientTalk's isolate [39]). The types of variables defined in a `let` or `bind` are inferred.

3.2 Core Language Syntax

For the sake of uniformity and to simplify reasoning about authority safety, Wyvern modules are translated into objects. The abstract grammar that has modules (Figure 6) is translated into the object-oriented core of Wyvern that does not have modules (Figure 7). Furthermore, in Wyvern's object-oriented core:

- Methods may have only one parameter.
- Expressions do not include the `let` construct.
- The `bind` construct may have only one variable.
- Expressions and declarations are extended with runtime forms that cannot appear in the source code of a Wyvern program.

To represent multiparameter methods, the `let` construct, and multivariable `bind` in the object-oriented core, we use a standard encoding (presented in the next section).

Expressions have two runtime forms: a location and a method-call stack frame. The location l refers to a location in the store μ (on the heap) that holds an object definition added at object creation. The method-call stack frame models the call stack and method calls on it, while preserving information about the receiver of the executing method. The

$$
\begin{array}{rcl}
\mathit{trans}(\overline{md}\ \texttt{platform}\ z\ \ \overline{i}\ e) & = & \begin{cases} \texttt{let}\ x = \mathit{trans}(md) & \text{if } \overline{md} = md\ \ \overline{md'} \\ \texttt{in}\ \mathit{trans}(\overline{md'}\ \ \texttt{platform}\ z\ \ \overline{i}\ e) & \\ \texttt{bind}\ z = \langle \mathit{constResObj} \rangle\ \mathit{trans}(\overline{i}) & \text{if } \overline{md} = \varnothing \\ \texttt{in}\ e & \end{cases} \\
\mathit{trans}(\texttt{module}\ x : \tau\ \ \overline{i}\ \ \overline{d}) & = & \texttt{bind}\ \mathit{trans}(\overline{i})\ \texttt{in}\ \texttt{new}_{\texttt{pure}}(x \Rightarrow \overline{d}) \\
\mathit{trans}(\texttt{module def}\ x(\overline{y : \tau}) : \tau\ \ \overline{i}\ \ \overline{d}) & = & \texttt{new}_{\texttt{resource}}(x \Rightarrow \texttt{def}\ \mathit{apply}(\overline{y : \tau}) : \tau \\
 & & \quad \texttt{bind}\ \overline{y = y}\ \mathit{trans}(\overline{i}) \\
 & & \quad \texttt{in}\ \texttt{new}_{\texttt{resource}}(\_ \Rightarrow \overline{d})) \\
\mathit{trans}(\overline{i}) & = & \begin{cases} y = x\ \ \mathit{trans}(\overline{i'}) & \text{if } \overline{i} = \texttt{import}\ x\ \texttt{as}\ y\ \ \overline{i'} \\ \varnothing & \text{if } \overline{i} = \varnothing \end{cases} \\
\texttt{let}\ x = e\ \texttt{in}\ e' & \equiv & \texttt{new}_s(\_ \Rightarrow \texttt{def}\ f(x : \tau) : \tau' = e').f(e) \\
\texttt{bind}\ \overline{x = e}\ \texttt{in}\ e & \equiv & \texttt{bind}\ x = (e_1, e_2, ..., e_n)\ \texttt{in}\ [x.n/x_n]e \\
\texttt{def}\ m(\overline{x : \tau}) : \tau = e & \equiv & \texttt{def}\ m(x : (\tau_1 \times \tau_2 \times ... \times \tau_n)) : \tau = [x.n/x_n]e
\end{array}
$$

Figure 8 Modules-to-objects translation rules, and encodings for `let`, multivariable `bind` and multiparameter methods.

expression $l.m(l_1) \triangleright e$ means that we are currently executing the method body e of a method m of the receiver l, and object l_1 was passed as an argument.

Since method bodies are evaluated lazily, i.e., only when an object calls the method, declarations have only one runtime form for object fields. Method bodies can never contain method-call stack frames. An object field in the source code can contain only a variable, which at runtime becomes a location in the store. Thus, the runtime form for an object field represents that a field f is referring to a location l.

A set of types of object fields and methods forms an object type, which is tagged as either pure or resource. We use standard typing contexts Γ for variables and Σ for the store, and to simplify Wyvern dynamic semantics, an evaluation context E.

3.3 Translation of Modules into Objects

Figure 8 presents modules-to-objects translation rules and encodings that are used in the translation but not expanded for brevity. A Wyvern program is translated into a sequence of `let` statements, where every variable in a `let` represents a module (the variable name x is the name of a module) and the body of the last `let` in the sequence is a `bind` expression containing the top-level code. The variables in this `bind` are a special constant resource object, representing the back-end implementation, and the translation of top-level imports. The body of the `bind` is the top-level expression.

In essence, modules are translated into objects: pure modules are translated into pure objects and resource modules and translated into resource objects. The exact translation of a Wyvern module depends on whether the module is a pure module or a resource module. If the module is pure, it translates into a `bind` construct, in which the module's imports become the `bind`'s variables, and the module's declarations are wrapped into a pure object of type τ in the `bind`'s body. If the module is a resource module, it is a functor, and it translates into a new resource object with a single method `apply()`. The `apply()` method takes as arguments the functor's arguments and, when called, returns a `bind` expression. The variables in the returned `bind` consist of variables that shadow the functor's arguments (since a `bind`'s body can access only the variables defined in the `bind` and no other, outside variables) and the imports of the resource module under translation. The body of the `bind` contains a resource object that encompasses the declarations of the translated resource module. The module's

```
module listFactory : ListFactory
  def create() : List
    ...
module def wordProcessor(io : FileIO)
                          : WordProcessor
  import wyvern : listFactory as list
  import logger
  var log : Logger = logger(io)
  var exts : List = list.create()
  ...
// top level
platform java
import fileIO
import wordProcessor
let io = fileIO(java) in
  let wp = wordProcessor(io) in ...
```

```
let listFactory = bind in new_pure(x ⇒
  def create() : List = ...) in
  let wordProcessor = new_resource(x ⇒
    def apply(io : FileIO) : WordProcessor
      bind
        io = io
        list = listFactory
        logger = logger
      in new_resource(_ ⇒
        var log : Logger = logger.apply(io)
        var exts : List = list.create()
        ...)) in
// top level
bind
  java = ⟨constResObj⟩
  fileIO = fileIO
  wordProcessor = wordProcessor
in
  let io = fileIO.apply(java) in
    let wp = wordProcessor.apply(io) in ...
```

Figure 9 A sample modules-to-objects translation.

declarations are prohibited from referring to the resource object itself (as it does not exist in the original code), and therefore we generate a fresh name for the self variable (in the translation, it is marked with an underscore). The `apply()` method of a functor's translation is invoked whenever the functor is invoked.

Importantly, the `bind` construct plays a significant role in Wyvern's module access control. Module imports are translated into variables in a `bind` construct. Since the body of a `bind` is disallowed to access anything outside the variables defined in the `bind`, a module can receive a capability to access a resource only via the import mechanism, as an argument to one of its methods, or as the return value from a method call on an imported module. This substantially limits the number of possible paths for acquiring module access.

The `let` construct, a multivariable `bind` construct, and multiparameter methods are provided only for developer convenience and are absent from Wyvern's core syntax; they are encoded instead. The `let` construct is encoded as a method call, and the multiplicity of variables in the `bind` construct and parameters in methods is achieved by bundling variables and parameters together in a tuple and then accessing them by their indices in the `bind` and methods' bodies.

Figure 9 shows an example of applying the translation rules from Figure 8. On the left is a code snippet as a developer would write it, and on the right is the same code written in Wyvern's core syntax without modules (the encodings are not expanded for conciseness, and we use the type abbreviations supported by our implementation rather than the less-readable structural types in our formalism). The snippet is a partial program; the `logger` and `fileIO` modules are assumed to be defined elsewhere.

The `listFactory` and `wordProcessor` modules are translated into variables defined in two nested `let`s. The outer `let` defines the `listFactory` module, which is translated into a `bind` expression. Since `listFactory` does not import any modules, the `bind` has no variables, and the `bind`'s body is a new pure object encompassing the `listFactory`'s `create()` method.

The inner `let` defines the `wordProcessor` module, which is translated into a resource object containing an `apply()` method. Similarly to the `wordProcessor` functor, the `apply()` method

takes an object of the `FileIO` type and returns an object of the `WordProcessor` type. The body of the `apply()` method is a `bind` expression, the variables of which are the `apply()`'s argument `io` as well as the two `wordProcessor`'s imports, `listFactory` and `logger`. The body of the `bind` expression has a resource object encompassing `wordProcessor`'s declarations. To get an instance of the `logger` module, the `logger`'s `apply()` method is called on it with an appropriate argument. Since the body of the `bind` is limited to access only the variables defined in the `bind`, `wordProcessor` has access to only three modules, `fileIO`, `listFactory`, and `logger`, and no other modules.

The top-level code is translated in the body of the inner `let` and is represented by a `bind` expression. The `bind` expression has all top-level imports as variable definitions and the top-level nested `let` expression in the body.

3.4 Static Semantics

The Wyvern static semantics are presented in Figure 10. The annotation underneath the turnstile—in the premise of T-New and declaration typing rules—is the same as the tag on the `new` construct in the syntax and serves to identify objects and their declarations as `pure` or `resource`. The annotation on top of the turnstile represents the current or future (in case of object creation) receiver of the enclosing method.

Tracking the receiver is used in lieu of making object fields private. Both mechanisms enforce non-transitivity of authority, but receiver tracking is simpler and is already implemented for authority safety. In the T-New rule, the receiver for the new object's declarations is the new object itself. In T-Field and T-Assign, the receiver is the object whose field is being accessed, which makes object field accesses private to the object to which they belong. For all declaration typing rules, the receiver is the object to which the declarations belong.

The T-Decls rule enforces that each declaration of an object is well-typed. DT-DefPure and DT-DefResource typecheck pure and resource object methods respectively. A pure method should be able to typecheck in a typing environment without any resource variables, except for the passed argument. The argument may be a resource, but because all other variables in the context are pure, it cannot be stored (e.g., be assigned to a variable) inside the method body. If all methods in an object are pure and the object does not have any fields, the object is pure. DT-DefResource has a standard, much less restrictive premise than DT-DefPure. If an object has a field, it is automatically declared a resource, and its typechecking proceeds as expected depending only on whether the field's value is a variable (DT-Varx) or a location (DT-Varl). The T-Store rule ensures that the store is well-formed and allocates new objects according to their types.

To summarize, an object is a resource if at least one of the following conditions is true:

1. The object contains a field (e.g., the object representing the `wordProcessor` module).
2. An object's method definition needs a resource variable to typecheck (e.g., the object representing `logger` needs an object of type `FileIO` to typecheck).

These conditions are checked *statically*. If neither of them are true, then the object is pure (e.g., the object representing the `listFactory` module).

The subtyping rules are standard, except for the S-State rule, which is used for the conversion between resource objects and pure objects:

$$\frac{}{\{\overline{\sigma_e}\}_{\texttt{pure}} <: \{\overline{\sigma_e}\}_{\texttt{resource}}} \text{ (S-State)}$$

$$\boxed{\Gamma \mid \Sigma \vdash^{e} e : \tau}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \mid \Sigma \vdash^{e} x : \tau} \text{ (T-Var)} \qquad \frac{\Gamma,\ x : \{\overline{\sigma}\}_s \mid \Sigma \vdash^{x}_{s} \overline{d} : \overline{\sigma}}{\Gamma \mid \Sigma \vdash^{e} \texttt{new}_s(x \Rightarrow \overline{d}) : \{\overline{\sigma}\}_s} \text{ (T-New)} \qquad \frac{\Gamma \mid \Sigma \vdash^{e'} e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \mid \Sigma \vdash^{e'} e : \tau_2} \text{ (T-Sub)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{e} e_1 : \{\overline{\sigma}\}_s \quad \texttt{def}\ \ m(x : \tau_2) : \tau_1 \in \overline{\sigma} \quad \Gamma \mid \Sigma \vdash^{e} e_2 : \tau_2}{\Gamma \mid \Sigma \vdash^{e} e_1.m(e_2) : \tau_1} \text{ (T-Method)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{e} e : \{\overline{\sigma}\}_s \quad \texttt{var}\ \ f : \tau \in \overline{\sigma}}{\Gamma \mid \Sigma \vdash^{e} e.f : \tau} \text{ (T-Field)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{e_1} e_1 : \{\overline{\sigma}\}_s \quad \texttt{var}\ \ f : \tau \in \overline{\sigma} \quad \Gamma \mid \Sigma \vdash^{e_1} e_2 : \tau}{\Gamma \mid \Sigma \vdash^{e_1} e_1.f = e_2 : \tau} \text{ (T-Assign)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{e} e_1 : \tau_1 \quad x : \tau_1 \mid \Sigma \vdash^{e} e_2 : \tau_2}{\Gamma \mid \Sigma \vdash^{e} \texttt{bind}\ x = e_1\ \texttt{in}\ e_2 : \tau_2} \text{ (T-Bind)} \qquad \frac{l : \tau \in \Sigma}{\Gamma \mid \Sigma \vdash^{e} l : \tau} \text{ (T-Loc)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{e'} l_1 : \{\overline{\sigma}\}_s \quad \texttt{def}\ \ m(x : \tau_2) : \tau_1 \in \overline{\sigma} \quad \Gamma \mid \Sigma \vdash^{e'} l_2 : \tau_2 \quad \Gamma \mid \Sigma \vdash^{l_1} e : \tau_1}{\Gamma \mid \Sigma \vdash^{e'} l_1.m(l_2) \rhd e : \tau_1} \text{ (T-StackFrame)}$$

$$\boxed{\Gamma \mid \Sigma \vdash^{z}_{s} \overline{d} : \overline{\sigma}} \quad \boxed{\Gamma \mid \Sigma \vdash^{z}_{s} d : \sigma}$$

$$\frac{\forall j,\ d_j \in \overline{d},\ \sigma_j \in \overline{\sigma},\ \Gamma \mid \Sigma \vdash^{z}_{s} d_j : \sigma_j}{\Gamma \mid \Sigma \vdash^{z}_{s} \overline{d}\ :\ \overline{\sigma}} \text{ (T-Decls)}$$

$$\frac{\begin{array}{c}\Gamma_{\texttt{resource}} = \{x : \{\overline{\sigma}\}_{\texttt{resource}} \mid x : \{\overline{\sigma}\}_{\texttt{resource}} \in \Gamma\} \\ \Gamma_{\texttt{pure}} = \Gamma \setminus \Gamma_{\texttt{resource}} \qquad \Gamma_{\texttt{pure}},\ y : \tau_1 \mid \Sigma \vdash^{z} e : \tau_2\end{array}}{\Gamma \mid \Sigma \vdash^{z}_{\texttt{pure}} \texttt{def}\ m(y : \tau_1) : \tau_2 = e\ :\ \texttt{def}\ m(y : \tau_1) : \tau_2} \text{ (DT-DefPure)}$$

$$\frac{\Gamma,\ x : \tau_1 \mid \Sigma \vdash^{z} e : \tau_2}{\Gamma \mid \Sigma \vdash^{z}_{\texttt{resource}} \texttt{def}\ m(x : \tau_1) : \tau_2 = e\ :\ \texttt{def}\ m(x : \tau_1) : \tau_2} \text{ (DT-DefResource)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{z} x : \tau}{\Gamma \mid \Sigma \vdash^{z}_{\texttt{resource}} \texttt{var}\ f : \tau = x\ :\ \texttt{var}\ f : \tau} \text{ (DT-VarX)}$$

$$\frac{\Gamma \mid \Sigma \vdash^{z} l : \tau}{\Gamma \mid \Sigma \vdash^{z}_{\texttt{resource}} \texttt{var}\ f : \tau = l\ :\ \texttt{var}\ f : \tau} \text{ (DT-VarL)}$$

$$\boxed{\mu : \Sigma}$$

$$\frac{}{\varnothing : \varnothing} \text{ (T-StoreEmpty)} \qquad \frac{\mu : \Sigma \quad x : \{\overline{\sigma}\}_s \mid \Sigma \vdash^{x}_{s} \overline{d} : \overline{\sigma}}{\mu,\ l \mapsto \{x \Rightarrow \overline{d}\}_s\ :\ \Sigma,\ l : \{\overline{\sigma}\}_s} \text{ (T-Store)}$$

Figure 10 Wyvern static semantics.

A pure object is a subtype of a resource object and, thus, can be used in place of a resource object, but not the other way around. Subtyping rules are presented in full in the technical report [21].

3.5 Dynamic Semantics

Figure 11 shows Wyvern's dynamic semantics. The E-Congruence rule subsumes all evaluation rules with non-terminal forms; the rest of the reduction rules deal with terminal forms. The E-New rule requires that the definition of the new object is closed, which is enforced in the progress theorem (below) and guarantees that the authority of the new

$\boxed{\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle}$

$$\frac{\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle}{\langle E[e] \mid \mu\rangle \longrightarrow \langle E[e'] \mid \mu'\rangle}\ \text{(E-Congruence)}$$

$$\frac{l \notin dom(\mu) \quad \texttt{new}_s(x \Rightarrow \overline{d})\ \textit{is closed}}{\langle \texttt{new}_s(x \Rightarrow \overline{d}) \mid \mu\rangle \longrightarrow \langle l \mid \mu, l \mapsto \{x \Rightarrow \overline{d}\}_s\rangle}\ \text{(E-New)}$$

$$\frac{l_1 \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu \quad \texttt{def}\ m(y : \tau_1) : \tau_2 = e \in \overline{d}}{\langle l_1.m(l_2) \mid \mu\rangle \longrightarrow \langle l_1.m(l_2) \rhd [l_2/y][l_1/x]e \mid \mu\rangle}\ \text{(E-Method)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu \quad \texttt{var}\ f : \tau = l_1 \in \overline{d}}{\langle l.f \mid \mu\rangle \longrightarrow \langle l_1 \mid \mu\rangle}\ \text{(E-Field)}$$

$$\frac{\begin{array}{c} l_1 \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu \qquad \texttt{var}\ f : \tau = l \in \overline{d} \\ \overline{d}' = [\texttt{var}\ f : \tau = l_2/\texttt{var}\ f : \tau = l]\overline{d} \quad \mu' = [l_1 \mapsto \{x \Rightarrow \overline{d}'\}_s/l_1 \mapsto \{x \Rightarrow \overline{d}\}_s]\mu \end{array}}{\langle l_1.f = l_2 \mid \mu\rangle \longrightarrow \langle l_2 \mid \mu'\rangle}\ \text{(E-Assign)}$$

$$\frac{}{\langle \texttt{bind}\ x = l\ \texttt{in}\ e \mid \mu\rangle \longrightarrow \langle [l/x]e \mid \mu\rangle}\ \text{(E-Bind)} \qquad \frac{}{\langle l.m(l_1) \rhd l_2 \mid \mu\rangle \longrightarrow \langle l_2 \mid \mu\rangle}\ \text{(E-StackFrame)}$$

Figure 11 Wyvern dynamic semantics.

object can be fully determined at its creation and onwards. To create a new object, a fresh store location is chosen, and the object definition is assigned to it. In E-Method, when the method argument is reduced to a location, a method-call stack frame is put onto the stack, the caller and the argument are substituted with corresponding locations in the method body, and the method body starts to execute. An object field is evaluated to the location that it holds (E-Field), and when an object field's value is reassigned, the necessary substitutions are made in the store (E-Assign). Similarly to methods, when the `bind`'s variable value is fully evaluated, variables in its body are substituted with their corresponding locations, and the `bind`'s body starts to execute (E-Bind). Finally, in the E-StackFrame rule, when a method body is fully executed, the method-call stack frame is popped from the stack and the resulting location is returned.

Notably, pure objects always remain pure, i.e., if a location l maps to a pure object in the store μ, then it always maps to a pure object in the store μ'. This can be proven by a simple induction on the reduction rules.

3.6 Type Soundness

The preservation and progress theorems are stated as follows. The proofs for both the theorems are fairly standard and are available in the technical report [21].

▶ **Theorem** (Preservation). *If* $\Gamma \mid \Sigma \vdash^{e''} e : \tau$, $\mu : \Sigma$, *and* $\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle$, *then* $\exists \Sigma' \supseteq \Sigma$, $\mu' : \Sigma'$, *and* $\Gamma \mid \Sigma' \vdash^{e''} e' : \tau$.

▶ **Theorem** (Progress). *If* $\varnothing \mid \Sigma \vdash^{e''} e : \tau$ *(i.e., e is a closed, well-typed expression), then either e is a value (i.e., a location), or* $\forall \mu$ *such that* $\mu : \Sigma$, $\exists e', \mu'$ *such that* $\langle e \mid \mu\rangle \longrightarrow \langle e' \mid \mu'\rangle$.

4 Authority Safety

We use the object-oriented core to prove our language authority-safe. Once modules are translated into objects, objects become the unit of reasoning, and thus our authority-related formalism is formulated in terms of objects.

In our system, a *principal* [5] is a resource object. An object—a principal or a pure object—can *directly access* a principal if the object has a reference to the principal, either by capturing it on object creation or acquiring it via a method call or return. The *authority* of an entity (an object or an expression) is the set of principals the entity can directly access, and we say that it has *authority over* those principals.

The *authority safety* property states that the authority of an object can only increase due to the creation of a new object, a method call, or a method return. More precisely, the situations in which authority can increase are:

1. **Object creation:** If a resource object A creates a new resource object B, then A gains authority over B.
2. **Method call:** If a resource object A does not have authority over a resource object B and receives B as an argument to one of A's methods, then A gains authority over B (perhaps only temporarily, while A's method is being executed).
3. **Method return:** If a resource object A does not have authority over a resource object B and B is returned from a method call that A invoked, then A gains authority over B (perhaps only temporarily, while A's method is being executed).

It is important to note that these must be *the only* situations when authority of an object increases (e.g., authority cannot increase due to side effects). The authority safety property is what assures us that all we need to reason about the authority of an object is to examine actions at its interface: method calls and returns; the case of object creation is usually not very interesting because the newly created object is born with no more authority than its creator had.

Note that the third case of authority safety is unique to our non-transitive definition of authority. In the transitive definitions of authority used in prior work, the caller of a method always already has the same authority as its callee, or more. This also means that if an object such as the `logger` is careful not to return a reference to the underlying file being used, then objects that use the `logger` will not have authority over that file, which matches our intuition about the role of the `logger` object as a gatekeeper.

For a pure object, an authority increase is inconsequential because a pure object cannot store mutable state. Thus the definition of authority safety focuses on principals—i.e., resource objects. On a technical level—as discussed in more detail below—we treat a pure object as being part of whatever resource object uses it.

4.1 Significance of Authority Safety

If a Wyvern program typechecks, it is authority-safe, i.e., authority gains are possible *only* in the three cases specified by the authority safety theorem. The type system *automatically, at compile time* enforces that a module *cannot* gain authority over and access to another module by any other means (e.g., via side effects). This property allows developers to reason effectively about the authority of program modules.

Consider reasoning about the authority of the `wordCloud` module. `wordCloud` is born with only the authority to access its required resources: due to the typechecking rule for `bind` and the way that modules are translated, these are the only resources in scope when `wordCloud` is instantiated. To see whether `wordCloud` gains any authority, the authority safety theorem

tells us we need only inspect its type (`WordCloud`) and that of its required resources (`Logger`). Together the types show over what resources `wordCloud` can gain authority via method calls and returns (cases 2 and 3 of the authority safety theorem). For example, it is easy to verify that no object representing `fileIO` can go across this interface and thus ensure that all file access done by `wordCloud` must go through the `logger`. Case 1 of authority safety allows `wordCloud` to create objects of its own that act as principals, but it cannot thereby gain access to system resources it did not already have. Notice that we can conclude all of this without even looking at the code in the `wordCloud` module—which is a useful property if this module is provided by a third party in compiled form and the source code is not available.

Authority safety also allows developers to reason about global invariants about the use of resources, while only needing to inspect part of the program. For example, to verify that the entire program only accesses the file system to write to log files, we first inspect the top-level code and observe that the `fileIO` resource is only passed to the `wordProcessor` module. We then inspect `wordProcessor` and observe that it passes the `fileIO` module exclusively into the `logger` module. Examining the `logger`'s code, we see that it enforces the desired invariant of writing only to log files, and does not provide clients with any means of accessing `fileIO` functionality. Since authority is *non-transitive* and neither `wordProcessor` nor `logger` expose `fileIO` via their methods, it is guaranteed that, besides `wordProcessor` and `logger`, no other program module has authority over `fileIO` module. It is unnecessary to inspect any other modules, which could make up an arbitrarily large fraction of the program, because we can rely on the authority safety property to ensure that those parts of the program can never acquire authority to `fileIO`.

Thus, our approach enables reasoning that is impossible in conventional languages, such as Java, without a global analysis that requires access to all code in the program, or use of the Java security manager (which is difficult to use correctly due to its excessive complexity [4]).

4.2 Formal Definition of Authority Safety

To formalize authority safety, we must first present a formal notion of authority. Our authority definition is given by two sets of rules—the $auth()$ and $pointsto()$ rules. Intuitively, $pointsto()$ captures references between objects, while $auth()$ is a higher-level relation that builds on $pointsto()$ to define authority. We describe the rules, give an example of how the rules are applied, state the authority safety theorem, and finally prove Wyvern authority-safe.

4.2.1 auth() Rules

The authority of an object is determined according to the functions and rules in Figure 12. Intuitively, our definition of authority has two parts. The first part, $auth_{store}$, captures the principals that an object has a reference to in the heap, either as one of its fields, or as a location captured in one of its methods (which act as closures in Wyvern). The second part, $auth_{stack}$, is more subtle: it captures the principals that an object has a reference to in an on-the-fly execution of one of the object's methods. More formally:

- $auth(l, e, \mu)$ takes a location l, an expression e, and a store μ, and returns a set of locations identifying principals that constitute the total authority of an object identified by l when an expression e is being executed in the context of memory μ.
- $auth_{store}(l, \mu)$ takes a location l and a store μ and returns a set of locations identifying principals to which an object identified by l has direct access by virtue of the object's

$\boxed{auth(l, e, \mu)}$ $\boxed{auth_{store}(l, \mu)}$ $\boxed{auth_{stack}(l, e, \mu)}$

$$\frac{}{auth(l, e, \mu) = auth_{store}(l, \mu) \cup auth_{stack}(l, e, \mu)} \text{ (AUTH-CONFIG)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\}_s \in \mu}{auth_{store}(l, \mu) = pointsto(l, \mu) \cup pointsto(\overline{d}, \mu)} \text{ (AUTH-STORE)}$$

$$\frac{l.m(l') \rhd e' \notin e}{auth_{stack}(l, e, \mu) = \varnothing} \text{ (AUTH-STACK-NOCALL)}$$

$$\frac{l.m'(l'') \rhd E' \notin E}{auth_{stack}(l, E[l.m(l') \rhd e'], \mu) = pointsto(e', \mu) \cup auth_{stack}(l, e', \mu)} \text{ (AUTH-STACK)}$$

Figure 12 Authority rules.

static state in the store μ. In other words, the function determines the object's authority that can be statically deduced by examining the code stored in the object.

- $auth_{stack}(l, e, \mu)$ takes a location l, an expression e, and a store μ, and returns a set of locations identifying principals to which an object identified by l has direct access by virtue of the execution state of methods of l executing in e in the context of memory μ. That is, the function determines the object's authority gained on the stack.

Since, in the process of evaluation, methods may have received new principals as arguments and method bodies may have been re-written to include new principals, the sets returned by $auth_{store}(l, \mu)$ and $auth_{stack}(l, e, \mu)$ may differ.

The AUTH-CONFIG rule defines the relation between the three functions: the total authority of an object consists of authority it has statically from the code it stores and authority it gained on execution. The AUTH-STORE rule defines $auth_{store}(l, \mu)$. It requires the object identified by l to be in the store μ and returns two sets of locations identifying principals to which an object identified by l has direct access via itself and its declarations.

The AUTH-STACK-NOCALL and AUTH-STACK rules define $auth_{stack}(l, e, \mu)$. The AUTH-STACK-NOCALL rule is used when there are no method-call stack frames with the receiver l on the stack ($l.m(l') \rhd e' \notin e$) and returns an empty set, as in such cases, l gains no authority from executing e. If the stack contains method-call stack frames where the receiver is l, the AUTH-STACK rule is used, and the authority is "collected" from the outermost such method-call stack frame (i.e., the furthest method-call stack frame from the expression that is being evaluated) up to the expression being evaluated. The condition $l.m'(l'') \rhd E' \notin E$ means that there must be no method-call stack frames with l as the receiver preceding the method call in consideration, which assures that, as we go down the stack, we do not miss any method calls with l as a receiver. The $auth_{stack}(l, e, \mu)$ returns a set of locations identifying the principals that the method body contains and the principals that l can access on the rest of the stack.

4.2.2 pointsto() Rules

Authority functions use *pointsto*() functions (Figure 13). The *pointsto*() functions take an expression e, a declaration d, or a list of declarations $\overline{d}$ and a store μ, and return a set of locations identifying principals to which the expression, the declaration, or the list of declarations point (i.e., have direct access) in the context of memory μ.

$\boxed{pointsto(e,\mu)}$ $\boxed{pointsto(\overline{d},\mu)}$ $\boxed{pointsto(d,\mu)}$

$$\frac{}{pointsto(x,\mu) = \varnothing} \text{ (POINTSTO-VAR)}$$

$$\frac{}{pointsto(\texttt{new}_s(x \Rightarrow \overline{d}),\mu) = pointsto(\overline{d},\mu)} \text{ (POINTSTO-NEW)}$$

$$\frac{}{pointsto(e.m(e'),\mu) = pointsto(e,\mu) \cup pointsto(e',\mu)} \text{ (POINTSTO-METHOD)}$$

$$\frac{}{pointsto(e.f,\mu) = pointsto(e,\mu)} \text{ (POINTSTO-FIELD)}$$

$$\frac{}{pointsto(e.f = e',\mu) = pointsto(e,\mu) \cup pointsto(e',\mu)} \text{ (POINTSTO-ASSIGN)}$$

$$\frac{}{pointsto(\texttt{bind } x = e \texttt{ in } e',\mu) = pointsto(e,\mu) \cup pointsto(e',\mu)} \text{ (POINTSTO-BIND)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{resource}} \in \mu}{pointsto(l,\mu) = \{l\}} \text{ (POINTSTO-PRINCIPAL)} \qquad \frac{l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{pure}} \in \mu}{pointsto(l,\mu) = \varnothing} \text{ (POINTSTO-PURE)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{resource}} \in \mu}{pointsto(l.m(l') \rhd e,\mu) = \{l\}} \text{ (POINTSTO-CALL-PRINCIPAL)}$$

$$\frac{l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{pure}} \in \mu}{pointsto(l.m(l') \rhd e,\mu) = pointsto(e,\mu)} \text{ (POINTSTO-CALL-PURE)}$$

$$\frac{}{pointsto(\overline{d},\mu) = \cup \bigcup_{d \in \overline{d}} pointsto(d,\mu)} \text{ (POINTSTO-DECLS)}$$

$$\frac{}{pointsto(\texttt{def } m(x:\tau_1):\tau_2 = e,\mu) = pointsto(e,\mu)} \text{ (POINTSTO-DEF)}$$

$$\frac{}{pointsto(\texttt{var } f:\tau = x,\mu) = \varnothing} \text{ (POINTSTO-VARX)}$$

$$\frac{}{pointsto(\texttt{var } f:\tau = l,\mu) = pointsto(l,\mu)} \text{ (POINTSTO-VARL)}$$

Figure 13 *pointsto*() rules.

A variable does not point to any location (POINTSTO-VAR). A new expression points to locations to which the new object's declarations points (POINTSTO-NEW). A method, an object field and its assignment, as well as a bind construct (POINTSTO-METHOD, POINTSTO-FIELD, POINTSTO-ASSIGN, and POINTSTO-BIND respectively) point to locations in their subexpressions. Depending on whether a location is identifying a principal or a pure object, it points to either itself (POINTSTO-PRINCIPAL) or nothing (POINTSTO-PURE) respectively. Depending on whether the method caller is a principal or a pure object, a method-call stack frame points to either itself (POINTSTO-CALL-PRINCIPAL) or a set of locations pointed to by the method body (POINTSTO-CALL-PURE) respectively.

POINTSTO-PRINCIPAL and POINTSTO-PURE look similar to $auth_{store}(l,\mu)$, but differ semantically: in these *pointsto*() rules, l is treated as an expression, not as a location identifying a principal, and so the only location l can access is itself.

A list of declarations points to a union of sets of locations to which each declaration in the list points (POINTSTO-DECLS). A method declaration points to the locations to which the method body points (POINTSTO-DEF). A field declaration points to locations to which

the field's value points: if the field's value is a variable, the field declaration does not point to any location (POINTSTO-VARX), and if the field's value is a location, the field declaration points to the same location as the value location (POINTSTO-VARL).

In our system, authority is non-transitive for principal objects and transitive for pure objects to which a principal points. As pure objects do not have fields, they cannot point to any resources and their methods cannot capture resources. Thus, POINTSTO-PRINCIPAL and POINTSTO-PURE do not involve declarations of the object identified by the location (cf. POINTSTO-NEW). However, an executing method of a pure object can have resources in it if they were passed as arguments. Since the pure object cannot own the resource arguments, in this case, the authority is transitive, and the resource arguments are owned by the resource caller down the stack. Therefore, POINTSTO-CALL-PRINCIPAL considers only the principal caller, whereas POINTSTO-CALL-PURE allows a principal caller down the stack to have authority over principals in a pure callee's method.

4.2.3 Determining Authority of an Object

To demonstrate how authority of an object is determined, consider the following definition of the `prettyChart` module:

```
module def prettyChart(logger : Logger) : WordCloud
  def updateLog(entry : String) : Unit
    logger.appendToLog(entry)
```

Assume that the definition of the `logger` module is as in Figure 2 and that the last line in the above code snippet is currently being executed, i.e., the method `appendToLog()` is called on the `logger` object. The `logger` object in the store μ looks like:

$$l_{logger} \mapsto \{\ x \Rightarrow \texttt{def}\ \ appendToLog(entry : String) : Unit \\ l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry)\ \}_{\texttt{resource}}$$

To find the authority l_{logger} has statically, i.e., from the code it contains, we apply AUTH-STORE, POINTSTO-PRINCIPAL, POINTSTO-DEF, POINTSTO-METHOD, POINTSTO-PRINCIPAL, and POINTSTO-VAR as follows:

$$\begin{aligned}
&auth_{store}(l_{logger}, \mu) \\
&\quad = pointsto(l_{logger}, \mu) \cup pointsto(\texttt{def}\ appendToLog(...)\ ..., \mu) \\
&\quad = \{l_{logger}\} \cup pointsto(\texttt{def}\ appendToLog(entry : String) : Unit \\
&\qquad\qquad l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry), \mu) \\
&\quad = \{l_{logger}\} \cup pointsto(l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry), \mu) \\
&\quad = \{l_{logger},\ l_{io}\}
\end{aligned}$$

To find the authority l_{logger} gained on the stack, we use AUTH-STACK, AUTH-STACK-NOCALL, POINTSTO-METHOD, POINTSTO-PRINCIPAL, and POINTSTO-VAR as follows:

$$\begin{aligned}
&auth_{stack}(l_{logger}, E[l_{logger}.appendToLog(l_{entry}) \triangleright l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry)], \mu) \\
&\quad = pointsto(l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry), \mu) \\
&\quad \cup auth_{stack}(l_{logger}, l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry), \mu) \\
&\quad = pointsto(l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry), \mu) \\
&\quad = \{l_{io}\}
\end{aligned}$$

Finally, by AUTH-CONFIG, the total authority of l_{logger} when executing the `appendToLog()` method is

$$\begin{aligned}
&auth(l_{logger}, E[l_{logger}.appendToLog(l_{entry}) \triangleright l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry)], \mu) \\
&\quad = auth_{store}(l_{logger}, \mu) \\
&\quad \cup auth_{stack}(l_{logger}, E[l_{logger}.appendToLog(l_{entry}) \triangleright l_{io}.open(\text{“}\sim/log.txt\text{”}).append(entry)], \mu) \\
&\quad = \{l_{logger},\ l_{io}\}
\end{aligned}$$

As expected, l_{logger} has authority over l_{io} and no other resource object.

This way, the $auth()$ and $pointsto()$ rules allow us to determine authority of every object on every step of execution, which serves as a basis for our formal system and the authority safety proof.

4.2.4 Authority Safety Theorem

We now state the authority safety theorem formally.

▶ **Theorem** (Authority Safety). *If*

1. $\Gamma \mid \Sigma \vdash^{e''} e : \tau$,
2. $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$,
3. $l_0 \mapsto \{x \Rightarrow \overline{d_0}\}_{\texttt{resource}} \in \mu'$,
4. $l \mapsto \{x \Rightarrow \overline{d}\}_{\texttt{resource}} \in \mu$, *and*
5. $auth(l, e', \mu') \setminus auth(l, e, \mu) \supseteq \{l_0\}$,

then one of the following must be true:

1. ***Object creation:***
   a. $e = E[l.m(l') \rhd E'[\texttt{new}_{\texttt{resource}}(x \Rightarrow \overline{d_0})]]$ *and*
   b. $e' = E[l.m(l') \rhd E'[l_0]]$, *where*
   c. $\forall l_a.m_a(l'_a) \rhd E'' \in E'$, $l_a \mapsto \{x \Rightarrow \overline{d_a}\}_{\texttt{pure}} \in \mu$
2. ***Method call:***
   a. $e = E[l.m(l_0)]$,
   b. $e' = E[l.m(l_0) \rhd [l_0/y][l/x]e'']$, *and*
   c. $y \in e''$
3. ***Method return:***
   a. $e = E[l.m(l') \rhd E'[l_a.m_a(l'_a) \rhd l_0]]$ *and*
   b. $e' = E[l.m(l') \rhd E'[l_0]]$, *where*
   c. $\forall l_b.m_b(l'_b) \rhd E'' \in E'$, $l_b \mapsto \{x \Rightarrow \overline{d_b}\}_{\texttt{pure}} \in \mu$

The formal statement of authority safety makes the informal statement above more precise, in that:

1. The principal gaining authority in the given evaluation step must be a receiver of a method-call stack frame on the stack, but not necessarily the immediate receiver for the expression under evaluation.
2. Receivers of all method-call stack frames between the principal receiver and the expression under evaluation must be pure.

These points allow us to define authority safety comprehensively, while treating pure objects as essentially a part of the principal that uses them. Below is a sketch of the proof of the authority safety theorem; the full proof is presented in the technical report [21].

Proof Sketch. The proof is by induction on a derivation of $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$. We start by considering E-Congruence and rely on the following fact (formally stated and proven in Lemma 8 in the technical report [21]):

- If there are only pure principals after the last method-call stack frame where l is the caller, i.e., l was the last principal caller on the stack, then
  $auth(l, E[e'], \mu') \setminus auth(l, E[e], \mu)$
  $= auth_{store}(l, \mu') \cup pointsto(e', \mu') \cup auth_{stack}(l, e', \mu')$
  $\setminus\ auth_{store}(l, \mu) \cup pointsto(e, \mu) \cup auth_{stack}(l, e, \mu)$

- Otherwise, if the last method-call stack frame where l is the caller is followed by a method-call stack frame with a principal caller that is not l, or if the stack has no method-call stack frames with principal callers, then
  $auth(l, E[e'], \mu') \setminus auth(l, E[e], \mu)$
  $= auth_{store}(l, \mu') \cup auth_{stack}(l, e', \mu') \setminus auth_{store}(l, \mu) \cup auth_{stack}(l, e, \mu)$

This implies that the changes in authority when $\langle E[e] \mid \mu \rangle \longrightarrow \langle E[e'] \mid \mu' \rangle$ depend on expressions in $\langle e \mid \mu \rangle \longrightarrow \langle e' \mid \mu' \rangle$. Next, we consider all possible terminal-form reduction steps and, using the *auth*() and *pointsto*() rules, calculate the difference in authority of the principals before and after the reduction step.

The subcases of E-New, E-Method, and E-StackFrame produce the three situations states in the theorem. The rest of the reduction rules do not cause any authority gains. ◀

5 Implementation

We have implemented the module system and core theory described in this paper as part of the open source Wyvern compiler and interpreter, available on GitHub: `https://github.com/wyvernlang/wyvern`. Although some features of a full-fledged language are missing, we have implemented examples from Figures 2, 3, and 4. The example code runs as part of the `wyvern.tools.tests.Figures` test suite and can be found in the `tools/src/wyvern/tools/tests/figs` subdirectory of the project. In ongoing development work, we are continuing to add features and improve the state of the implementation.

6 Limitations

Our threat model makes an important assumption that the code in the trusted code base of a software system is trustworthy. We assume that the security and privacy experts who are in charge of the trusted code base are honest and do not make mistakes. This may not be true in practice, and thus our approach is susceptible to insider attacks, which are common to systems that reason about trusted code bases and involve vulnerabilities inside the trusted code base.

For example, an expert responsible for the trusted code base may have a malicious intent and subvert the software system by exporting the functionality of system resources via wrapper functions. A wrapper function is a function of a module (e.g., `logger`) that "wraps" the functionality of a function of another module (e.g., a module of type `FileIO`), performing the same operations as the original function, e.g.:

```
module def logger(io : FileIO) : Logger
  def write(fileName : String, text : String)
    io.write(fileName, text)
```

By calling `logger.write()`, an extension importing `logger` could write to any file in the file system, and this would not be exposed in the `logger`'s type or interface. In a similar fashion, the malicious `logger` module may export functionality of an entire file I/O module, potentially changing function names to obfuscate the exposure. In such a case, an extension that is allowed to import `logger` would, in essence, have authority over a module of type `FileIO`.

Although insider attacks directed at the trusted parts of a system are beyond our reach, our approach allows developers to formally reason about the isolation of security- and privacy-related resources in a software system and gives developers a tool to enforce certain isolation properties. Also, the described limitations can be mitigated either by using more rigorous software development practices, e.g., code reviews, for critical parts of the system,

or by complementing our approach with more complex analyses, e.g., by using an effects system or an information flow analysis.

7 Related Work

Introduced to secure operating system resources [5], capabilities were later generalized to protect arbitrary services and resources [43], including programming language resources [28]. The object-capability model, in which capabilities guard more fine-grained programming language resources—objects—has recently been advocated by Miller [25]. The two pioneering languages that used object capabilities are E [24] and W7 [32]. Wyvern carries forward this line of work by exploring a statically typed, capability-safe language and providing support for modules as capabilities.

Our approach to modules was primarily inspired by the capability-passing modules design in Newspeak [2] and its predecessors, such as MzScheme's Units [13]. As in Newspeak, Wyvern modules are first-class. However, Wyvern's static types support reasoning about capabilities based on module interfaces (Newspeak is dynamically typed), and Wyvern reduces the overhead of ubiquitous module parameterization by allowing pure modules to be directly imported, rather than passed in as arguments (in Newspeak, all module dependencies must be passed in as arguments).

Several research efforts limited mainstream, non-capability programming languages to turn them into capability languages. Typically the imposed restrictions disallow mutable global state (e.g., static fields), tame the original language's APIs (e.g., reflection API), and prohibit ambient authority [40]. Sometimes sandboxing is used to facilitate isolation of program components (e.g., add-ons). Programming languages in this category include Joe-E [22] (a restricted subset of Java), Emily [37] (a performant subset of OCaml), CaPerl [17] (a subset of modified Perl), Oz-E [36] (a proposed variation of Oz), and Google's Caja [14, 26] (an enforced subset of JavaScript). In contrast, our work explores a module system with explicit support for capabilities without the constraint of adapting an existing language, enabling a cleaner design.

Shill [27] is a secure shell scripting programming language that takes a declarative approach to access control. In Shill, capabilities are used to control access to system resources, contracts are used to specify what capabilities each script requires, and capability-based sandboxes are used to enforce contracts at runtime. Shill supports compositional reasoning by tracing authority through program invocations and, if necessary, attenuating authority on every transition. The authority of the program's entry point is ambient, but its transition to other parts of the program is limited via contracts and sandboxes. Shill does not include mutable state (e.g., variables), which are part of Wyvern's model and make Wyvern's notion of authority safety more interesting; nor does Shill include a module system.

Maffeis et al. [20] formalized the notions of capability and authority safety and proved that capability safety implies authority safety, which in turn implies resource isolation. They showed that these semantic guarantees hold in a Caja-based subset of JavaScript and other object-capability languages. Maffeis et al.'s formal system defines authority topologically (objects are represented as nodes in a graph, and a path between two nodes implies that the source node can access the destination node) and thus transitive. In contrast, our formal definition of authority is non-transitive, enabling the important forms of reasoning discussed in Section 4.1.

Devriese et al. [6] presented an alternative formalization of capability safety that is based on logical relations. They argue that formalizations like Maffeis et al.'s [20] are too syntactic

and the topological definition of authority is insufficient to characterize capability safety as it leads to over-approximation of authority. Our non-transitive definition of authority is similarly more precise than prior, transitive topological definitions. However, our focus is on a relatively simple (compared to logical relations) type system that provides authority safety with respect to this more refined notion of authority, along with support for modules as capabilities.

Another line of related work assumes a capability-safe base language and develops logics or advanced type systems to state and prove properties that are built on capabilities. Drossopoulou et al. analyzed Miller's mint and purse example [25], rewrote it in Joe-E [8] and Grace [30], and based on their experience, proposed and refined a specification language to define policies required in the mint and purse example [9, 10, 11, 12]. Also, Dimoulas et al. [7] proposed a way to extend an underlying capability-safe language with declarative access control and integrity policies for capabilities, and proved that their system can soundly enforce the declarative policies. Dimoulas et al.'s formalization, like that of Maffeis et al. but unlike ours, formalizes authority transitively.

8 Conclusion

We presented a module system design that allows software developers to limit and control the authority granted to each module in a software system. Our module system supports first-class modules and uses capabilities to protect access to security- and privacy-related resource modules. It simplifies the reasoning for determining the authority of a module down to examining the module's interface, the module's imports, and the interfaces of the modules it imports, making security auditing more practical. Furthermore, unlike previous module systems (cf. Newspeak) that put significant overhead on developers by requiring all modules to be fully parameterized, in the Wyvern module system, parameterization is necessary only for resource modules, and the number of non-resource-module imports is unlimited. Our work also advances theoretical models of capabilities by modeling authority in a non-transitive way, which allows for attenuating a module's authority, such as when a powerful capability (e.g., file I/O) is encapsulated inside an attenuated capability (e.g., logging). We formally defined what it means for a module system to be authority-safe and proved that our module system possesses this property.

References

1 John Boyland, James Noble, and William Retert. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *European Conference on Object-Oriented Programming*, 2001.

2 Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as Objects in Newspeak. In *European Conference on Object-Oriented Programming*, 2010.

3 Shuo Chen, David Ross, and Yi-Min Wang. An Analysis of Browser Domain-isolation Bugs and a Light-weight Transparent Defense Mechanism. In *Conference on Computer and Communications Security*, 2007.

4 Zack Coker, Michael Maass, Tianyuan Ding, Claire Le Goues, and Joshua Sunshine. Evaluating the Flexibility of the Java Sandbox. In *Annual Computer Security Applications Conference*, 2015.

5 Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 9(3):143–155, 1966.

6 Dominique Devriese, Frank Piessens, and Lars Birkedal. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *European Symposium on Security and Privacy*, 2016.

7 Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. Declarative Policies for Capability Control. In *Computer Security Foundations Symposium*, 2014.

8 Sophia Drossopoulou and James Noble. The Need for Capability Policies. In *Workshop on Formal Techniques for Java-like Programs*, 2013.

9 Sophia Drossopoulou and James Noble. How to Break the Bank: Semantics of Capability Policies. In *Integrated Formal Methods*, 2014.

10 Sophia Drossopoulou and James Noble. Towards Capability Policy Specification and Verification. Technical report, Victoria University of Wellington, 2014.

11 Sophia Drossopoulou, James Noble, and Mark S. Miller. Swapsies on the Internet: First Steps Towards Reasoning About Risk and Trust in an Open World. In *Workshop on Programming Languages and Analysis for Security*, 2015.

12 Sophia Drossopoulou, James Noble, Toby Murray, and Mark S. Miller. Reasoning about Risk and Trust in an Open World. Technical report, Victoria University of Wellington, 2015.

13 Matthew Flatt and Matthias Felleisen. Units: Cool Modules for HOT Languages. In *Programming Language Design and Implementation*, 1998.

14 Google, Inc. Caja. `https://code.google.com/p/google-caja/`.

15 Michael Homer, Kim B. Bruce, James Noble, and Andrew P. Black. Modules As Gradually-typed Objects. In *Workshop on Dynamic Languages and Applications*, 2013.

16 Darya Kurilova, Alex Potanin, and Jonathan Aldrich. Modules in Wyvern: Advanced Control over Security and Privacy. In *Symposium and Bootcamp on the Science of Security*, 2016.

17 Ben Laurie. Safer Scripting Through Precompilation. In *Security Protocols*, 2007.

18 Michael Maass. *A Theory and Tools for Applying Sandboxes Effectively*. PhD thesis, Carnegie Mellon University, 2016.

19 David MacQueen. Modules for Standard ML. In *ACM Symposium on LISP and Functional Programming*, 1984.

20 Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object Capabilities and Isolation of Untrusted Web Applications. In *IEEE Symposium on Security and Privacy*, 2010.

21 Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. A Capability-Based Module System for Authority Control. Technical Report CMU-ISR-17-106, Carnegie Mellon University, 2017. URL: `http://reports-archive.adm.cs.cmu.edu/anon/isr2017/abstracts/17-106.html`.

22 Adrian Mettler, David Wagner, and Tyler Close. Joe-E: A Security-Oriented Subset of Java. In *Network and Distributed System Security Symposium*, 2010.

23 Heather Miller, Philipp Haller, and Martin Odersky. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *European Conference on Object-Oriented Programming*, 2014.

24 Mark S. Miller. The E Language. `http://erights.org/elang/`.

25 Mark S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

26 Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe Active Content in Sanitized JavaScript. Technical report, Google, Inc., 2008.

27 Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. SHILL: A Secure Shell Scripting Language. In *USENIX Symposium on Operating Systems Design and Implementation*, 2014.

28 James H. Morris, Jr. Protection in Programming Languages. *Communications of the ACM*, 16(1):15–21, 1973.

29 Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: A Simple, Typed, and Pure Object-Oriented Language. In *Workshop on Mechanisms for Specialization, Generalization and Inheritance*, 2013.

30 James Noble and Sophia Drossopoulou. Rationally Reconstructing the Escrow Example. In *Workshop on Formal Techniques for Java-like Programs*, 2014.

31 Martin Odersky, Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Philipp Haller, Stéphane Micheloud, Nikolay Mihaylov, Adriaan Moors, Lukas Rytz, Michel Schinz, Erik Stenman, and Matthias Zenger. Scala Language Specification. `http://scala-lang.org/files/archive/spec/2.11/`. Last accessed: May 2017.

32 Jonathan A. Rees. A Security Kernel Based on the Lambda-Calculus. Technical report, Massachusetts Institute of Technology, 1996.

33 John M. Rushby. Design and Verification of Secure Systems. In *Symposium on Operating Systems Principles*, 1981.

34 Jerome H. Saltzer. Protection and the Control of Information Sharing in Multics. *Communications of the ACM*, 17(7):388–402, 1974.

35 Z. Cliffe Schreuders, Tanya Mcgill, and Christian Payne. The State of the Art of Application Restrictions and Sandboxes: A Survey of Application-oriented Access Controls and Their Shortfalls. *Computers and Security*, 32:219–241, 2013.

36 Fred Spiessens and Peter Van Roy. The Oz-E Project: Design Guidelines for a Secure Multiparadigm Programming Language. In *Multiparadigm Programming in Mozart/Oz*, 2005.

37 Marc Stiegler. Emily: A High Performance Language for Enabling Secure Cooperation. In *International Conference on Creating, Connecting and Collaborating through Computing*, 2007.

38 Mike Ter Louw, Prithvi Bisht, and V Venkatakrishnan. Analysis of Hypertext Isolation Techniques for XSS Prevention. *Web 2.0 Security and Privacy*, 2008.

39 Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. AmbientTalk: Programming Responsive Mobile Peer-to-peer Applications with Actors. *Computer Languages, Systems and Structures*, 40(3–4):112–136, 2014.

40 David Wagner and Dean Tribble. A Security Analysis of the Combex DarpaBrowser Architecture. `http://combex.com/papers/darpa-review/security-review.pdf`, March 2002.

41 Esther Wang and Jonathan Aldrich. Capability Safe Reflection for the Wyvern Language. In *Workshop on Meta-Programming Techniques and Reflection*, 2016.

42 Robert N. M. Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *USENIX Workshop on Offensive Technologies*, 2007.

43 William A. Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–345, 1974.