A Case Study in Language-Based Security: Building an I/O Library for Wyvern

Jennifer A. Fish
jafish@andrew.cmu.edu
School of Computer Science,
Carnegie Mellon University
Pittsburgh, PA

Darya Melicher
darya@cs.cmu.edu
School of Computer Science,
Carnegie Mellon University
Pittsburgh, PA

Jonathan Aldrich
aldrich@cs.cmu.edu
School of Computer Science,
Carnegie Mellon University
Pittsburgh, PA

Abstract

As the impact of vulnerabilities increases in practice, it is imperative for programming languages to include security as a first-class design consideration. While a number of security-related language features have been proposed to address this need, in many cases, we do not know enough about whether it is practical and useful to build software systems in languages with these features.

In this paper, we begin to investigate this question, using a case study methodology. The setting of our case study is Wyvern, a recently designed language we selected because it incorporates three advanced security-related features: capability safety for enforcing the principle of least privilege, an effect system for tracking the secure use of resources, and a language extension feature that mitigates command injection. In our case study, we built a small standard I/O library, seeking to use the new language features to create a library that is less vulnerable to misuse and can serve as a building block for more secure programs, compared to conventional I/O library designs. Our study suggests that these features are indeed practicable and useful, and thus potentially promising for inclusion in other future language designs. It also sheds light on the value and cost of these features and suggests directions for future research on security-focused language design.

CCS Concepts: • **Security and privacy** → *Software security engineering*.

Keywords: Security, Programming Language, Libraries, Case Study

ACM Reference Format:
Jennifer A. Fish, Darya Melicher, and Jonathan Aldrich. 2020. A Case Study in Language-Based Security: Building an I/O Library for Wyvern. In *Onward! 2020*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Onward! 2020, November 15–20, 2020, Chicago, IL

https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Security continues to rise in importance to the users and developers of software. For more than 20 years, researchers have been exploring language-based security mechanisms as a way to make systems more secure [14]. A number of different security-related language features have been developed, including information flow types [24], capabilities [23], specified foreign function interfaces [27], effect systems [18], and safe language extensions [26]. However, in many cases, we do not know enough about whether it is practical and useful to build software systems in languages with these features.

In this paper, we begin to investigate this question, using a case study methodology. We decided to focus our case study on I/O functionality, for two reasons. First, I/O is a critical aspect of nearly all software systems, and thus a critical test of how well a language supports programmers. I/O allows programs to communicate over networks, obtain and display information via a user interface, and read from or store information in files. Second, I/O is also a major target of security-related attacks. For example, improper use of I/O can create security vulnerabilities such as command-injection attacks [33]. Another concern is privilege escalation, which occurs when an attacker has control of a part of a system, e.g., via a plugin or exploitation of a security vulnerability, and leverages that control to accomplish some malicious goal. Typically such a goal requires the attacker to gain access to I/O. When developing large-scale software applications, following the *principle of least privilege* can limit the misuse of I/O resources, as each module only has access to the minimum set of privileges needed for its intended functionality. In addition to mitigating security exploits, this principle also limits damage caused by improper or unwanted use of I/O and other sensitive resources [29].

We selected the Wyvern programming language as the setting for our case study, because Wyvern was designed to include several features that are intended to mitigate command-injection and privilege-escalation attacks. The first is Wyvern's capability-based module system: in order for

a given module to have access to certain system resources, capabilities must be passed as arguments upon instantiation of the module [20]. This allows developers to limit privileges at the module level, providing mitigation of privilege-escalation vulnerabilities. Second is Wyvern's effect system, which allows static reasoning about how a module uses the privileges it has. Since the effect system captures privilege use more directly than capabilities, it is a particularly effective way to identify privilege-escalation attacks, e.g., by observing that a module developed by a third party has more effects than expected given its advertised functionality. Third, Wyvern's type-specific language (TSL) extension feature addresses command-injection attacks [26]. This feature allows developers to import library support for new embedded languages, such as SQL, replacing the common but insecure existing practice of constructing SQL queries from strings.

Wyvern covers only a subset of the space of language-based security, of course, but we believe it is a subset that defends against important and interesting attacks: command-injection is currently the number one web application vulnerability according to OWASP [1], and privilege escalation can exploit a wide range of vulnerabilities. However, Wyvern's unique features—capability-based security, an effect system, and language extension—are still unusual, and we have little experience building libraries and applications based on these features. We do not know whether doing so is practical, nor how effectively these novel features can be applied in a real setting. Answering these questions is critical, as it will help developers decide whether to adopt languages with this features, and it will help the developers of future languages decide whether to incorporate them.

To start to answer these questions, this paper contributes a case study applying these security features to the construction of a standard I/O library for Wyvern. The library features a file system module with a permission-based capability hierarchy for access to combinations of read and write functionality, a network library with support for synchronous and asynchronous operations, and support for type-safe formatted printing. The library's design uses the highlighted security aspects of Wyvern for various purposes. Capabilities are used to restrict read and write permissions for objects that can modify file and network streams, TSLs are used to implement print formatting, and effects are used throughout to indicate observable side effects on sensitive I/O resources. In designing and implementing this library, we explored to which aspects of a library these language-design features could be effectively applied, as well as some of the drawbacks, syntactic overhead, and trade-offs with respect to traits such as readability, convenience, and practicality. Overall, the design of the library met our goals well, suggesting that these language features can support a functional and secure I/O library design. We present a discussion of the various design decisions that we made, as well as potential alternatives and future improvements that can be made to reduce the drawbacks of added security features.

While our case study is carried out in Wyvern, we believe that the results could be useful to designers who are evolving other languages and creating new ones. Capability-based security has been added to a number of other languages, including E [23], Joe-E [21], and Newspeak [9]. Numerous recent languages explore effect systems (e.g. Koka [16]) and while their focus has not been on security, Turbak and Gifford [34] argue that this is a natural application for future language designs with effects. Finally, many other researchers are exploring language extension [10, 30], and application of those facilities to security is a natural direction there too.

2 Background and Related Work

Security vulnerabilities in third-party software are a rising concern as reusable code becomes the only way to meet the demands of large-scale application development. While the majority of the responsibility for enforcing software security rests with the software architect, certain programming language features can make vulnerabilities easier to detect and mitigate.

2.1 Capabilities

One way to prevent certain classes of security vulnerabilities is to limit the authority of program modules, so that they cannot abuse access to system resources for malicious purposes. In the object-capability model, all resources are encapsulated as objects, and the only way to affect a resource is to invoke a method on that object. For example, resources might include the file system as well as individual files within it, so a capability-based design would include objects representing files and an object representing the file system.[1] Methods on the file system might provide access to files, and methods on files might support reading and writing to them.

In a capability system, since capabilities are objects, permission to access resources is represented by a reference graph. Objects are only able to communicate with other objects via references. In this reference-graph model, objects cannot be arbitrarily accessed, but rather can only be accessed if a capability is explicitly granted. "Only connectivity begets connectivity," all access is derived from previous accesses, and so, under this model, disjoint subgraphs cannot become connected [23]. In Wyvern, modules are treated as first-class capabilities. Global state is forbidden—it would break the "only connectivity begets connectivity" rule—and thus modules must be given all the external resources they need as either instantiation parameters or resources that are delegated to the module later on during execution [20].

In practice, the capability system prevents untrusted modules from accessing sensitive resources, since such accesses

[1] In our design, the file system is represented by a module, but modules are just a special case of objects.

cannot happen unless capabilities to those resources are passed to them directly. When capabilities are passed as arguments, the required permissions are also visible in the module's interface. In this way, Wyvern's capability model provides clarity about the resources used by a module, eliminating the need to examine the implementation directly and making it easier for software architects to detect security flaws.

Another security application of object capabilities is the isolation of third-party plugin modules. While delegating capabilities to third-party modules can isolate them from certain components of the host application, it does not prevent them from interfering with one another. This can be solved by providing each third-party plugin with a disjoint set of capabilities, which under the object-capability model prevents the reference graphs from being combined, thus preventing communication between the two plugins [31].

There are alternative methods of authority control implemented in other languages. In Java, permissions to access system resources, including access to network connections and files, are handled by the security manager [4]. Permission objects are used to represent permitted accesses to system resources, and are granted to code by a policy for the application environment specifying what permissions are allowed [2, 13]. The security manager checks this context before sensitive or unsafe operations are performed and can throw exceptions for forbidden actions [4, 12].

The object-capability model provides several useful security advantages compared to other models. Capabilities better support the principle of least privilege, because an object only has the privileges passed to it via capabilities, which can be made as few as possible. Capabilities support dynamic creation of objects and can enable and control dynamic delegation of authority between them. Capabilities also mitigate the confused-deputy problem, which can occur when a program is exploited to misuse its authority [22]. Capabilities also support reasoning based on object connectivity alone, without any knowledge of the code itself, and thus work well with unknown plugin code that may be controlled by an attacker. In contrast, the Java security model requires a complicated stack inspection algorithm in addition to the permission model. Object capabilities are much simpler, and therefore easier to use correctly by both system and application writers.

The earlier paper on capability-based modules in Wyvern described a fairly primitive, monolithic Wyvern `fileIO` library that did not support capability-based security at any finer granularity than the entire file I/O module [20]. In contrast, this paper presents a more mature I/O library design that supports fine-grained capabilities, and we give a much more detailed description and a discussion of the design tradeoffs involved.

Listing 1. Example of Reader interface and instantiation

```
type Reader
    effect Read
    ...

def makeReader() : Reader = new
    effect Read = fileEffects.Read
    ...
```

2.2 Effects

Wyvern's effect system is another feature that can be used to reason about code more easily and thereby enforce security properties. Much like capabilities, effect annotations make programs more "transparent," revealing potentially unsafe access to system resources. Rather than indicate the general ability of a module to access system resources, however, these annotations indicate specific uses of resources that arise from calling particular methods. Hence, effect annotations can provide a more direct and specific indication of the resource use within a module.

An unusual feature of Wyvern's effect system is *effect abstraction*. Types can declare abstract effects, which are given concrete definitions when an object of that type is instantiated. This allows for some flexibility in using a single, generalized type for multiple modules with different effects, improving code reusability. For example, the partial interface for the `Reader` type in Listing 1 defines an abstract effect `Read`, which is then given a concrete definition in the `makeFileReader` implementation.[2] When an object of type `Reader` is created, a concrete definition of the `Read` effect must be specified. In this way, if we were to have a `Reader` for reading from files and another for reading from network connections, they could both use the common `Reader` type, with the abstract `Read` effect referring to different underlying system effects. Usage of this feature with respect to Wyvern's I/O library will be covered in more detail later.

Wyvern also has support for parameterizing functions by effects. Observe the function in Listing 2.[3] The function `invokeTwice` is parameterized by an effect `E`, taking in a function that takes no input and produces no output and has effect `E`, and invokes the function argument twice. Effect parameterization improves reusability of functions with effects: rather than require multiple instances of the same function with different annotations to convey different effects, we can

[2]Code is simplified from https://github.com/wyvernlang/wyvern/blob/f3e4c9a26ab6f9d15fe836a1541ada3f2ade549a/stdlib/platform/java/fileSystem/Reader.wyt and https://github.com/wyvernlang/wyvern/blob/f3e4c9a26ab6f9d15fe836a1541ada3f2ade549a/stdlib/platform/java/fileSystem.wyv

[3]Code adapted from https://github.com/wyvernlang/wyvern/blob/3ad5c69ddfd90c01b50a452040f60f18c590538c/examples/effects/twice.wyv

Listing 2. Example of effect parameterization

```
def invokeTwice[effect E](f : Unit -> {E} Unit)
    f()
    f()

invokeTwice[{fileEffects.Write}](
    () -> file.write("Hello World!") )
```

create a generic function that takes in an effect as a parameter. In addition to function parameterization, types can be parameterized by effects; this has the effect of substituting the concrete effect given in the parameter for the abstract effect member of the type. Thus, in the earlier example, we could specify the return value of `makeFileReader` more precisely as `Reader[{fileEffects.Read}]`, indicating that the `Read` effect member of the returned `Reader` is equal to the set `{fileEffects.Read}`.

There are several trade-offs involved in using Wyvern's effect system. The effect system provides another round of checks on the operations performed by code and serves as another layer of security but comes at the cost of more syntactic overhead. Required annotations could also potentially result in a loss of flexibility, but this can be resolved with parameterization and abstraction of effects, as described above.

A concurrent submission contributes a detailed description of Wyvern's effect system, some effect idioms and a formalization and an application case study [6]. In contrast, this paper describes the detailed design of Wyvern's I/O library, including both effect features and other features. The two papers do have some relationship; this paper by necessity gives a brief overview of Wyvern's effect system and uses some of the effect idioms (but neither is a contribution of this paper). Similarly, the application case study in the other paper builds on our I/O library, but does not describe the I/O library's design other than a passing reference to `File` and its effects (and that is not a contribution of the other paper). Notably, the case study described in this paper motivated some of the unique effect features that are added to Wyvern in the concurrent submission.

2.3 Language Extension

Wyvern also has support for language extension, a feature which primarily enhances readability and syntax convenience but also has security applications, to be covered below. Type-specific languages (TSLs) in Wyvern associate logic with a type, determining how literals[4] of that type are parsed and elaborated. Essentially, this allows programmers to write literals of library abstract data types such as lists. The programmer can also define TSLs for user-defined types, greatly enhancing flexibility and syntactic convenience.

[4]Note that "literals" are not limited to just data; for example, literals of type `SQL` or `StateMachine` would include behavior.

For example, consider that you want to write a SQL statement for querying a database. If we imagine[5] an library-based interface to SQL, we might write a SQL statement like this:

```
let query : SQL = SelectStmt(["title", "author"],
    "posts",
    [WhereClause(InPredicate(StringLit(keyword),
                             "title"))])
```

This is difficult to comprehend and has low readability, while also being tedious for the programmer to write. If the language has built in notation for SQL, then the following could be written as syntactic sugar for the previous statement:

```
let query : SQL = ~
    SELECT title, author FROM posts
    WHERE {keyword} in title
```

While this is the most convenient as it is the most similar to the syntax of native SQL, such notation may not always be provided by the language, especially for types created by the programmer. If such notation is not available, it is common for developers to default to parsing string representations, as it is easy to implement and has better readability compared to the first (library-based) approach:

```
let query : SQL = parse_sql("SELECT title, author"
    + " FROM posts WHERE'"+keyword+"' in title")
```

While this method is convenient, it is a bit awkward in that it requires explicit casting to the SQL type, as well as escaping to switch between the language syntax and string literals. It can also lead to run-time errors, as the expression is parsed at run-time and thus the program cannot statically find syntax errors during compilation. Regarding security, this method of string parsing makes the program vulnerable to injection attacks.

Wyvern's TSL mechanism solves this problem by allowing developers to import language extensions as libraries. In the hypothetical example above (adapted from [26]), a TSL is associated with the `SQL` type, and the `~` in the example indicates that the following indented lines are expressed in the SQL language extension. This eliminates the possibility of command injection because the string data (the `keyword` variable) could be pasted in safely (e.g. escaping command symbols) by the TSL library. The original TSL paper [26] did not report on an implementation of the SQL TSL, so there is not as much experience with this idea as one might like. In the next section, we start to address this lack of experience, discussing how our I/O library for Wyvern uses a TSL extension to prevent format string attacks. Our formatstring library is the first published non-toy library based on TSLs, so our case study contributes validation that the feature is practicable and that its benefits extend to a domain different from those proposed in the earlier work.

[5]This is illustrative; a SQL library has not yet been implemented for Wyvern.

2.4 Previous Language-Based Security Case Studies

Other case studies have explored language-based security. For example, an early set of case studies investigated Java as a systems programming language, including some discussion of Java's support for security [7]. More recently, Askarov and Sabelfeld [5] implement cryptographic protocols in Jif, a language that supports static information flow types in Java. In contrast, a dynamic language-supported approach to information flow was evaluated in a case study implementing a secure conference review system [32]. Finally, Giffin et al. [11] implement a code-hosting website on top of Hails, a web framework with mandatory access control and a declarative policy language.

Many more such case studies could be cited. Our work is unique and interesting for two reasons. First of all, rather than exploring one language feature, we build a library that takes advantage of multiple security-related language features, including some aspects that are complementary (e.g. effects and capabilities help with privilege escalation in different ways). Second, the combination of the features we evaluate and our security focus is unique. In the setting of capabilities, I/O libraries were designed for a dynamically-typed language, E [23], and existing I/O libraries were adapted for Joe-E [21] (though there is little published discussion of the library design). However, our case study leverages the combination of types and a capability-safe module system, features that were not present together in the prior case studies. Similarly, although researchers have used case studies to explore and evaluate effect systems [16] and language extension mechanisms [17], we are not aware of case studies looking at these mechanisms specifically from a security point of view.

3 Library Design and Implementation

We implemented a standard I/O library for Wyvern, providing a consistent set of abstractions for accessing standard input and output as well as the file system (text and binary) and network (connection-oriented and datagram). This library implementation has now been accepted into the main Wyvern codebase, where it replaced various ad-hoc designs that existed before.

The I/O library is 1559 lines of code, including 828 lines of Wyvern and 731 lines of Java. The Wyvern code consists mostly of interfaces and thin wrappers to underlying Java functionality; the Java code is itself mostly wrappers to Java's extensive I/O library. Because of this leverage, we are able to get a fair amount of functionality with a relatively small amount of code. Although most of the actual functionality of the code is written in Java, the interfaces are the most interesting part from the perspective of evaluating the security features of the language, which is our goal in this case study.
‘

3.1 Capabilities

Wyvern's capability-based module design allows developers to limit the capabilities that are passed to other modules, which can enforce more secure and transparent usage of sensitive system resources. Capability-based modular design is particularly highlighted by the `fileSystem` module of the I/O library. Observing the principle of least privilege, we want modules to have only the minimal set of capabilities necessary for their desired functionality [29]. This ultimately limits the potential damage that can be caused by unwarranted access to system resources, as well as increases clarity regarding the access permissions of each module.

Design Considerations. Our primary design goal was to create a library that makes it easy for programmers to restrict the I/O access of untrusted modules. For example, if the programmer wants to use the functionality of a module, but does not fully trust it, we want the programmer to be able to limit that module's access to specific files or subdirectories—or even (in the extension we discuss below) reading from but not writing to a file. In a capability-based system, we do this by passing to each module only the capabilities it needs to do its task.

The primary design issue we considered is the granularity with which to support capabilities. Melicher et al. [20] described an earlier design for file I/O in Wyvern which supported only one, very coarse-grained capability: a `fileIO` module that provides access to the entire file system. While useful for controlling which parts of the program access files, this design does not support many useful forms of expressiveness, such as limiting an untrusted plugin to read-only access to particular files.

At the other extreme, a design focused on maximizing expressiveness could provide very fine-grained capabilities, for example supporting any combination of read, write, or append access to any file or directory subtree. A downside of this design is that it has the potential to be very complicated. For example, it might lead to 8 different `Directory` abstractions: one for each combination of read, write, and append permissions; in Joshua Bloch's terms, this would be an API with a poor *power-to-weight ratio* [8].

Our design resolves this tension by providing capability-based abstractions for the entire file system as well as files and directories. We support stream-based file I/O, and so `Reader` and `Writer` abstractions naturally support read-only and write-only access to a file that is open. To minimize conceptual weight, we do not explicitly support read/write/append capabilities to directories or unopened files, but as we will see, our design enables programmers to easily implement this functionality themselves if they want it.

```
type FileSystem
    def fileFor(path : String) : File
    def directoryFor(path : String) : Directory

type Directory
    def files() : List[File]

type File
    def makeReader() : Reader
    def makeWriter() : Writer

type Reader
    effect Read
    def read() : Int
    def close() : Unit

type Writer
    effect Write
    def write(s : String) : Unit
    def close() : Unit

module def fileSystem(java: Java): FileSystem
    import java:wyvern.stdlib.support.FileIO.
        file

    def fileFor(path : String) : File
       ...
    ...
```

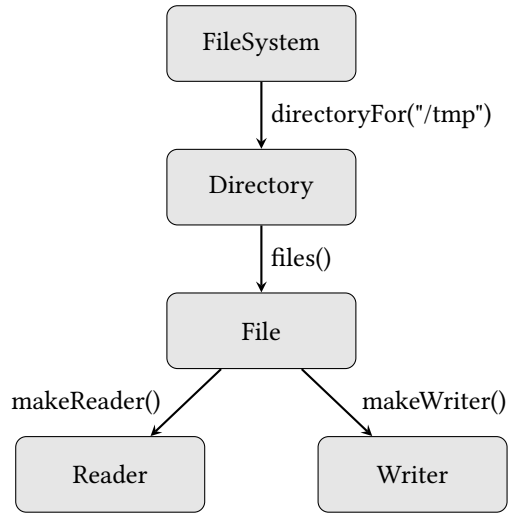

Figure 1. Partial interface for some of the Wyvern types used in File I/O operations. The diagram shows the hierarchy of module capabilities, where arrows indicate that a type can produce an instance of another type via the given method.

Concrete Design. As outlined in Figure 1, the `fileSystem` module is structured with support for four levels of access: to the entire file system, to a directory, to a single file, and read-only or write-only to a file.[6] The `fileSystem` module provides access to the entire file system; it can only be instantiated using a special *foreign function interface* capability `java` that is passed from the Wyvern run-time system to the top-level Wyvern script (Wyvern's equivalent of `main`). The `java` capability is used to import Java objects that implement the file system functionality at the lowest level. By controlling who is passed either the `java` capability or the `fileSystem` capability that is produced from it, the top-level Wyvern script has complete control over which parts of the program can access files.

The `fileSystem` module can be used to create objects of Wyvern's `File` type, representing specific files, via the `fileFor` method. A similar method allows searching for a `Directory`, which in turn allows listing its constituent files. The `File` type itself has methods for creating `Reader` and `Writer` types for that specific file in memory. Thus, the owner of the `fileSystem` module instance not only has control over which files can be modified by other modules, but can also directly grant read-only, write-only, or both read and write access to the file using its `makeReader` and `makeWriter` methods. By passing a `File` directly, the recipient of that `File` can also choose to grant read-only or write-only access to that file by passing a `Reader` or `Writer` for that file to another module. Hence this design for the `fileSystem` creates a hierarchy of access levels in which types with a higher permission level are able to produce types beneath it in the hierarchy via library methods.

From a security standpoint, restricting access to files can potentially minimize damage from malicious third-party modules by only giving them access to the file operations needed for their stated purpose, as well as by giving access to only the files that should be modified or read. Thus, these modules are unable to read or overwrite confidential information stored in other files.

Example Usage Scenario. Imagine that Alice needs to incorporate a third-party module into her code, but doesn't fully trust it. Alice wants the third-party module to provide data analysis; it needs read-only access to her data directory, but should not be able to modify the data, or read data outside that directory. As mentioned above, our library has native support for read-only access to individual open files, but not to entire directories. However, Alice can easily build read-only directory access on top of our library, as shown in Listing 3.[7] In her design, Alice defines two new

[6] Code is from https://github.com/wyvernlang/wyvern/blob/f3e4c9a26ab6f9d15fe836a1541ada3f2ade549a/stdlib/platform/java/fileSystem.wyv and https://github.com/wyvernlang/wyvern/tree/f3e4c9a26ab6f9d15fe836a1541ada3f2ade549a/stdlib/platform/java/fileSystem, simplified by removing some declarations and effect signatures.
[7] Simplified from https://github.com/wyvernlang/wyvern/blob/f3e4c9a26ab6f9d15fe836a1541ada3f2ade549a/examples/dirTest.wyv

Listing 3. Example of restricting a directory capability to provide read-only access to an untrusted third-party module

```
// read only types
type ReadOnlyFile
    def makeReader() : Reader

type ReadOnlyDir
    def files() : List[ReadOnlyFile]

// client code
require java
import fileSystem
import untrustedModule

val fileSystemInst = fileSystem(java)

val dataDir = fileSystemInst.directoryFor("/data")

def restrictDir(d:Directory):ReadOnlyDir = new
    def files() : List[ReadOnlyFile]
        d.files().map[ReadOnlyFile](
                    (f:File) => restrictFile(f))

def restrictFile(f:File):ReadOnlyFile = new
    def makeReader() : Reader = f.makeReader()

val restrictedDir = restrictDir(dataDir)

val untrustedInst = untrustedModule(restrictedDir)
untrustedInst.getProcessedData()
```

types: `ReadOnlyFile` and `ReadOnlyDirectory`. Her program `require`s a `java` capability from the operating system, imports `fileSystem`, and instantiates a `fileSystemInst` capability from it, passing in the `java` FFI capability as required. She also imports the untrusted module (which is technically a functor; it must be instantiated with an object representing its source directory). She uses the `fileSystem` capability to get a capability to her data directory using `directoryFor`. She then defines a `restrictDir` function that wraps an ordinary `Directory` so that the `files` accessor returns a list of `ReadOnly` files (using a companion `restrictFile` function). She restricts her data directory to read-only access, instantiates the third-party module with the restricted directory, and makes the data requests she needs.

Discussion, Alternatives, and Limitations. The scenario above illustrates the usage of our library, and shows that clients can easily implement additional capability-based security mechanisms on top of what we provide. Note that if read-only directories turn out to be widely useful, Alice could share her `restrict` functions with others, and they could even be incorporated into a future version of our library.

What alternative designs would solve this problem, especially in languages without capability-based security? We could read the data and pass the data to the untrusted module, without passing a `File` object. This is potentially a performance problem; perhaps the untrusted module only needs to read part of the `File`. But a bigger issue is that, without some security controls, the untrusted module could open the file anyway and write to it. Capability-based security is one way to do this, as shown in this example. Another would be access control mechanisms, such as Java's code permission model; but this approach would give up the pleasant simplicity of capabilities, and (as argued separately [23]) make it more difficult to ensure that confused deputies do not inadvertently give the untrusted module inappropriate privileges.

Could an attacker compromise this design, gaining the ability to write to unauthorized files? We can easily reason that this is impossible. First, any access to files must ultimately go through the foreign function interface, to Java in this case. But Wyvern's `import` statement can only import from Java using the `java` capability, and we didn't pass it to the untrusted module. The untrusted module could try to get a capability from a global variable–but Wyvern's type system prohibits global variables from holding any object with mutable state or with access to FFI functionality [20]. The only capabilities the untrusted module can get are the ones we pass to it, either when it is first created or when it is later used. By examining the interface of the `restrictedDir` object—the only object passed to the untrusted module—we can determine that all capabilities reachable from `restrictedDir` are ones we are willing to share with the untrusted module.

One limitation of this example is the need to explicitly wrap the directory using the `restrictDir` and `restrictFile` functions. This limitation is necessary in dynamically-typed capability-safe languages, such as E [23]. However, Wyvern is statically typed; it would be nice if we could omit the `restrict` functions entirely and simply write:

```
val restrictedDir : ReadOnlyDir = dataDir
```

counting on the type system to enforce the type signature of `ReadOnlyDir` and thus not allow calls to `restrictedDir.files().get(0).makeWriter()`. Unfortunately, there are currently two problems with this. First of all, Wyvern does not yet support covariant type parameters, so `ReadOnlyDir` is not a supertype of `Directory`. This observation suggests a pragmatic need for supporting covariant (and contravariant) type parameters in Wyvern—and in fact there has been recent theoretical work on this, but it is not yet implemented [19]. Second, although Wyvern does not support downcasts (except for explicitly tagged types [15]), the current implementation of Wyvern includes a dynamic type `Dyn` that allows the encoding of a downcast. This observation suggests that Wyvern should adopt recent proposals for incorporating parametricity into gradual typing [25]. While our example demonstrates that neither of these enhancements is strictly necessary to support capability-based security, together they would make programming with capabilities more efficient

both in lines of code and in run-time overhead (since it would be unnecessary to create wrapper objects)—and would also allow strong reasoning about capabilities based on examining the types of objects alone.

3.2 Effects

Capabilities are useful for enforcing least-privilege security constraints, but they do have limitations: reasoning about what code can do requires reasoning about what capabilities it has access to, which in turn requires some reasoning about the dynamics of the system. An alternative approach is to reason statically about what code can do using an effect system. We used Wyvern's effect system to annotate the I/O library to support this kind of reasoning.

Note that while the example of using Wyvern's capability-based I/O library incurred some overhead from object wrappers that restrict capabilities, static effect annotations can be erased at run time,[8] and so can be used without any performance cost.

Design Considerations. As with capabilities, our main goal is to allow programmers to restrict the I/O access of untrusted modules. However, with effects, we'd like this restriction to be done with strictly static reasoning. Just by looking at the declared effects of a module, it should be possible to reason about what it can do. As a concrete example, we'd like to reason that an untrusted module can only read from files, not write to files or access the network.

As with capabilities, the main design tradeoffs involve the granularity with which to track effects. Wyvern has only one native effect: `system.FFI`, which is triggered whenever Wyvern code calls code written in another language (Java, in our case) through the language's Foreign Function Interface (FFI). The `system.FFI` allows programmers to track what code might have some system-level effect, but says nothing about what that effect might be.

At the opposite extreme, Wyvern's effect system can differentiate reads and writes to each individual `File` object. Forcing the programmer to reason separately about effects to every file, however, quickly becomes unweildy; in fact, it's not even possible to describe all such effects unless there are a finite number of files and a variable for each one is in scope wherever the effect needs to be declared. So we take a middle ground: each `File` object declares its own read and write effects, but generally we treat those effects as equivalent to global file read and write effects.

Concrete Design. We annotated our I/O library with effects that describe read and write effects to files and the network. Figure 2 shows some of the key effect declarations in the initial implementation of our file system (note that the effect declarations were left out of our earlier code listings, to allow us to focus on capabilities there).[9]

```
resource type File
    effect Read
    effect Write
    def makeReader(): {} Reader[{this.Read}]
    def makeWriter(): {} Writer[{this.Write}]

resource type Writer
    effect Write
    def write(s: String):{this.Write}Unit

module fileEffects
    effect Read = {system.FFI}
    effect Write = {system.FFI}

module networkEffects
    effect Read = {system.FFI}
    effect Write = {system.FFI}

module def fileSystem(java: Java): FileSystem
    import java:wyvern.stdlib.support.FileIO.file
    def fileFor(path: String): {fileEffects.Read}
        File[
            {fileEffects.Read},
            {fileEffects.Write}]
        val f = file.createNewFile(path)
        ...

module def io(java: Java): IO
    type Socket
        effect Read
        effect Write
        def makeReader(): {} Reader[this.Read]
        def makeWriter(): {} Writer[this.Write]

    def makeSocket(...): {networkEffects.Read}
        Socket[{networkEffects.Read},{
        networkEffects.Write}]
```

Figure 2. Selected (and somewhat simplified) code showing effect declarations for File and Network I/O.

We used effect abstraction to create generalized `Reader` and `Writer` types to represent reads and writes on different I/O streams. The library currently has support for these types implemented on TCP/IP network connections and files. Reads and writes on the different kinds of streams have nearly identical usage patterns despite their differing implementations, so a generic interface that encapsulates the core functionality of input and output streams is more versatile. With effect abstraction, the abstract `Read/Write`

[8] Wyvern's compiler is currently focused on debugging and so maintains this information, but it is well-understood how to erase it.

[9] This code is available at the same links given in the previous subsection, plus https://github.com/wyvernlang/wyvern/blob/master/stdlib/platform/java/io.wyv and https://github.com/wyvernlang/wyvern/blob/master/stdlib/platform/java/io/networkEffects.wyv

effects can represent effects on either file or network streams depending on the underlying implementation, but it does not need to be specified in the interface. Modules or functions can also take in a `Reader` or `Writer` as a parameter and perform I/O operations on both network streams and file streams, depending on the implementation of the object passed in. Hence the generic `Reader` and `Writer` types allow for more versatile usage and promote code reusability.

The implementation of `fileSystem` imports the `wyvern.stdlib.support.FileIO.file` object from Java (this import can be used later using the shortened name `file`). All methods in the `file` object are automatically treated as if they have effect `system.FFI`—this ensures that effects resulting from native code cannot be forgotten, though they can be abstracted as some higher-level effect. Thus, when `fileFor` invokes `file.createNewFile(path)`, it acquires the effect `system.FFI`. Since `fileFor` is annotated with `fileEffects.Read`, which is known to be defined as equal to `system.FFI`, the effect checker succeeds.

In the implementation of network stream I/O, we had to make certain design decisions about the set of network effects which should be represented. Aside from the core `Read` and `Write` effects, it would also be useful to indicate establishing connections with a `Connect` effect. Typically a connection would be followed by read or write operations, but it would also be possible for a program to arbitrarily open connections and occupy system resources without performing any other operations. Establishing the connection itself also results in several bits being communicated along the connection, which is observable on both ends of the network. Hence it could be important to indicate that a connection was made and a port on the local machine is in use as an observable side effect.

However, in a practical setting, this hypothetical `Connect` effect could instead be represented by a `Read` effect for a more concise set of effects. The purpose of opening a socket connection would be to either read, write, or both to a socket channel. In the trivial case of opening a connection without performing reads or writes, a small amount of data is transferred to indicate a successful connection. Since this last case is uncommon, there would be a trivial loss in precision by using a `Read` or `Write` effect in place of `Connect`. We chose to replace it with a `Read` effect, since reads can be considered less dangerous than writes in that they do not physically modify data.

Major Limitation Observed. In our library implementation, we observed a major limitations of Wyvern's effect system, stemming from its inability to hide the concrete definition of an effect at any scope larger than a single file. This came up when defining read and write effects on the file system. We declare these effects in a globally visible module, called `fileEffects`, because any code in the file library, or which uses that library, needs to be able to refer to them. The implementation of the file system needs to know the definition of these effects–in this case, they are both implemented as Wyvern's primitive FFI-usage effect, `system.FFI`. The definition of these effects thus has to be exposed by the `fileEffects` module–but this means that *all* modules can see that definition. In practice, this means that the `fileEffects.Read` and `fileEffects.Write` effects are collapsed into one, and the effect checker does not distinguish them.

This is a very serious issue. To understand why, consider the following example code:

```
module def readonlyAnalysis(f:File[{fileEffects.
    Read}, {fileEffects.Write}]) {}
    def analyze() : { fileEffects.Read } Unit
        f.makeWriter().write("Hello")
```

If you look at the module's signature, it has only one function, `analyze`, which has the effect `fileEffects.Read`. However, the body of the `analyze` function calls `f.makeWriter().write()`, which has effect `fileEffects.Write`. So how can `analyze` typecheck? Well, `fileEffects.Read` and `fileEffects.Write` are both declared to be equal to `system.FFI`, so the effect checker allows this annotation.

Fixes and Discussion. A fix could be accomplished within Wyvern's existing type and effect theory; it is a matter of implementation to extend signature ascription to work across groups of modules. This fix would disallow the example above because the definition of `fileEffects.Read` and `fileEffects.write` would only be visible within the Wyvern I/O library, and would not be visible within module `readonlyAnalysis`.

An alternative fix is to support effect declarations that are not just abstract, but abstract and *bounded* by another effect. This second fix approach has now been prototyped [6]. It allows us to declare module `fileEffects` as follows:

```
module fileEffects
    effect Read >= {system.FFI}
    effect Write >= {system.FFI}
```

Now, instead of being declared to be equal to `system.FFI`, `fileEffects.Read` is declared as a super-effect of `system.FFI`. This allows the library implementation's `FFI` effect to be subsumed by `Read`, but would not allow mixing the `Read` and `Write` effects, since there is no known sub-effecting relationship between them. Thus, the second (implemented) fix also disallows the problematic example above.

With either of these fixes, we can achieve our design goals: by looking at the effects on functions in the `readonlyAnalysis` module, we can determine that even though it has the *capability* to read and write to the passed-in file, it only actually reads from it. We can have confidence that the declared effects are accurate without reading the

source code; the effect checker ensures this. An example correct implementation of the module is given below.

```
module def readonlyAnalysis(f:File[{fileEffects.
    Read}, {fileEffects.Write}]) {}
    def analyze() : { fileEffects.Read } Int
        f.makeReader().read()
```

Alternatives and Discussion. As with capabilities, one alternative is to use access control mechanisms to reason about the effects of code. Looking at Java's code permission model, we observe that Java's mechanism is much more coarse-grained, as it restricts all the code from a given source. Java's access control also does not help in reasoning about transitive effects in the way that our system does: even if a module `m` cannot write to files, if `m` invokes a more privileged module `p`, then `p` might write to files, and code in `m` might have the transitive effect of writing to modules.

Regarding transitivity, Wyvern's effect abstraction does allow us to rename an effect such as `fileEffects.Write` as a differently-named effect. This is a powerful feature, but one that users have to be aware of: to prohibit the `fileEffects.Write` effect in general, you also have to prohibit any effect that is defined in terms of `fileEffects.Write`.

Another alternative to effect systems would be to use monads to reason statically about I/O effects. Haskell's I/O monad [28] can be used to reason about the presence or absence of an I/O effect, but it combines all different forms of I/O and thus does not support statically distinguishing reads and writes. A different design for Haskell is possible that would make this distinction, but Haskell's monads do not obey the nice algebraic properties that allow our effects to be treated as sets, so a more fine-grained design would quickly become inconvenient. Monads also require programmers to structure their code in restrictive ways.

Other Limitations. A second, more minor limitation is that the developer must fix a choice about which effects to reason about. For example, we chose to treat network connections as a read effect; as a result, our library's effect declarations are not helpful to a client that wants to reason about connections separately from reads. In the context of the I/O library we've implemented so far, we believe this is a minor limitation, as it seems clear which effect declarations are useful to the majority of clients. However, it is possible that there are examples for which this choice is less clear. Parameterization may be able to help in those cases, at the cost of some additional complexity in the interface.

Readers may wonder about the syntactic overhead of effects. We believe the most appropriate way to measure syntactic overhead is not in libraries that use effects, but applications built on those libraries. In an application case study that builds on our library and uses effects fairly intensely, effect annotations were observed to increase lines of code by 9% [6].

A related issue is that effect annotations and effect checking are optional in Wyvern–thus, they impose no overhead on programmers who do not use them. Code that is not annotated with effects is not checked for effects. This brings up an interesting issue: if we *want* to make sure all code is effect-checked, as in our case study, how do we do so? Wyvern implements the special rule that if an effect is annotated on the module as a whole—for an example, note that `readonlyAnalysis` is annotated with the empty effect `{}`, meaning that no effect occurs when the module is instantiated—then every function in the module must also be annotated with an effect. For our case study, we ensured that all I/O library files have an effect annotation at the top; this in fact caught a few missing annotations. In principle, one could automatically check that all files in a directory tree have an annotation at the top, though this functionality has not yet been implemented in the Wyvern compiler.

Finally, it's worth noting that any code that uses the foreign function interface must be marked by a `system.FFI` effect, or some other effect that abstracts `system.FFI`. Programmers can thus trust the effect system to not forget any real system accesses, but must be responsible for reviewing code that does use the FFI to ensure that any effect abstraction it does is reasonable and appropriate. This responsibility is, in some sense, inevitable: every system must use lower layers of abstraction somewhere, and code that does so must always be inspected to make sure those uses are appropriate.

3.3 Language Extension

Another feature added to the I/O library is a TSL extension for format strings. Format strings provide a convenient syntax for programmers to insert formatted string representations of various types, typically using format specifiers. These can then be used to write formatted information to stdout or other output streams. Using Wyvern's support for language extensibility, we defined a TSL that can splice Wyvern primitives into a string. Several examples are given below:[10]

```
require stdout

val x : String = "World"
stdout.printf(~)
  Hello %{x}!
// output: "Hello World!"

val y : Int = 6
stdout.printf(~)
  The sum is %{y + 5}. y = 6 is %{y == 6}.
  We can also print %.2{1.6}.
// output: "The sum is 11. y = 6 is true.
//          We can also print 1.60."
```

[10] Similar code in the Wyvern repository can be found at https://github.com/wyvernlang/wyvern/blob/93464607b17f6d0092f9a5d3160d0bdfda38d1bf/examples/tsls/formatstringClient.wyv

Note that this differs from the traditional syntax for formatted strings in languages such as C in several ways. First, instead of providing a string containing format specifiers and additional arguments for the formatted expressions to `printf`, we only provide one string that contains the Wyvern expressions directly, indicated by a `%` and surrounded by curly braces. Second, whereas C requires the format specifiers to indicate the type of the argument inserted into the format string (i.e., `%d` for integers, `%c` for char values), Wyvern uses type inference to determine how the value should be formatted. Additional format specifier syntax exists for floating point precision, as shown above.

We are able to use features such as type inference, instead of tediously requesting type specifiers from the programmer, because of the way Wyvern expressions are spliced in the TSL syntax. As the argument string is parsed to construct an AST, we find the first valid format specifier (`%{...}`). We then process the code within the brackets as a Wyvern expression, which is typechecked and compiled. Using the inferred type of the expression, we wrap the AST with a call to the respective type's `toString` function, then a call to the concatenate function along with the part of the string that has been processed, then continue to parse the rest of the string.

Creating a TSL for format strings in this way prevents format string attacks, enhancing the security of application code. In languages like C, carefully picked values and format specifiers can cause the program to read from and modify the stack and other locations in memory. For example, if the programmer takes in a string as input from the user, the user can include format specifiers in their input to read beyond what is intended from memory.

For example, consider the following code in C that takes input from a user and prints out the contents of the input:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main (int argc, char **argv) {
    char buf[100];
    // write input into buffer
    snprintf(buf, sizeof buf, argv[1]);
    printf("Input is %s\n", buf);
    return 0;
}
```

With normal inputs, this would print out `Input is (input).` However, if the input string contains format specifiers, the user can unexpectedly read memory address contents.[11] If the string `"Hello world! %x %x"` is given to the program, the call to `snprintf` would write the input into the buffer `buf`. However, it interprets the `%x` format specifiers, and replaces them with the contents of a memory address. When the string is printed out in `printf`, the memory contents are exposed to stdout. A user could also cause a program to crash by providing numerous format specifiers such that the program eventually attempts to access an illegal memory address [3].

Due to the way Wyvern expressions are spliced in the TSL, they are processed and typechecked separately from the template string. Thus, expressing the above C example is impossible in Wyvern. The input from the user is of type `String`, not `FormatString`, and there is no coercion between them provided: all `FormatString`s must be literals in program text. It's also impossible to write a `FormatString` literal that declares specifiers for which data is not provided, because the data must be provided as a in-line Wyvern expression right at the format specifier. Wyvern's language extension feature allows us to bypass these kinds of format string vulnerabilities.

A notable benefit of using TSLs for format strings is that we were able to provide a nicer interface to format strings (e.g. by providing type inference) that is also more secure. Due to this language feature, the secure approach enhances usability, rather than the two being in conflict, as is the usual case!

Wyvern's type-specific language feature operates as a compile-time code generator; there is no overhead associated with its reflective features at run time.

Comparison to Built-In Format Specifiers. Some languages, such as Python, build a format specifier feature into the language. Our library provides a similar level of convenience, but we can provide additional checking compared to Python. For example, our format strings include details such as decimal precision which are applicable to some types and not others; if the developer passes something that's not of the right type, our library detects it statically, whereas Python would fail at run time. More broadly, we explore a much different point in the design space, which has not been previously evaluated: Python builds format specifiers into the language, whereas our format specifiers are implemented as a library. This keeps the language leaner, and more importantly, it means that our library can be modularly extended with other, similar features, whereas extending Python in a similar way would require invasive changes to the compiler and language specification.

3.4 A Challenge: Asynchronous I/O

In building the I/O library, we discovered certain design challenges that may require further work in the future to make Wyvern programs more precise and expressive. Notably, while designing the framework for asynchronous operations, we encountered the challenge of representing effects in an asynchronous context.

[11] Note that modern C compilers may very well warn about such code, but such warnings must be built into the compiler. In Wyvern, format strings can be expressed safely as a library. Nothing is built-in, so new things that are similar to formal strings can also be expressed safely, which would not be true in C.

From a functionality standpoint, we chose to represent asynchronous operations with the use of a `Future` type inspired by the corresponding type in Scala, as it has a more functional style and allows for the convenience of supplying callback lambdas to be executed upon completion of the asynchronous process. This is more concise than an iterative approach of using loops until the action is complete. However, a drawback of this approach is introduced when we attempt to explicitly annotate the Future methods with effects. Observe the following partial interface for the Future type:

```
type Future
    type T
    def get() : T
    def andThen[U](callback : T -> U) : Future[U]
```

While `andThen` provides a convenient means of composing callbacks that can be executed upon completion of the initial asynchronous process, it raises the question of when callback effects should be expressed. Consider a scenario in which numerous callbacks which each perform some I/O operation with a visible effect are chained together in the form `Future[T](...).andThen[U](...).andThen[V](...)` .... When should the effect of each callback be expressed?

There are several possible ways to express the effects which arise from this asynchronous chain of operations. One way is to associate the cumulative set of effects with the resulting Future, in which case the annotation would be provided with the `get()` method. However, this lacks precision in expressing the effect of each individual callback when it is called on the result of the previous Future. There is also no guarantee that the `get()` method would ever be called, and since the asynchronous operation is initiated upon creation of the initial Future, it is possible that these effects would occur but never be expressed in annotations.

Another option is to associate these effects with a parameterized `andThen` method, parameterizing it with the type and effect of the supplied lambda. This is most likely the best approach, as it more precisely expresses the effects of the callbacks as they are supplied. However, this degree of parameterization becomes very verbose, and our impression is that it would not be worthwhile until (for example) Wyvern provides better support for inferring the effect parameters to be used on each call to `andThen`.

At the present time, the effects for asynchronous operations as provided in the library are expressed upon creation of the resulting Future type; however, effect-annotated functions cannot be supplied as a callback. This would need to be handled in future work—perhaps based on enhanced effect parameter inference in Wyvern—to support fully effect-checked asynchronous I/O code.

3.5 Additional Library Features

The discussion above focuses on particular features of the library that illustrate lessons learned from our case study. However, while the library remains modest in size, it does include a number of additional features. These include reading and writing in both text and binary, an abstraction supporting random access files, and both TCP/IP and UDP networking. The library captures some finer distinctions as well—for example, standard input streams are conceptually infinite, but input streams derived from a file are bounded and the client can request to read the entire stream using a `readFully()` method. The approach to design using capabilities, effects, and the format string TSL that was outlined above extended smoothly to these additional features.

4 Conclusions and Future Work

Wyvern's I/O library serves as an example of applying various security-related language features to a small but real and common design problem. It provides a sense for what is possible to accomplish with capabilities, effects, and language extensions, and what the practical benefits and costs might be. In our view, the case study was largely successful: we were able to implement the core of a modern synchronous I/O library in Wyvern with minimal problems, and with no evidence that Wyvern's unique features are likely to cause problems as the library scales up. Through Wyvern's features—capabilities, effects, and language extensions—our library enables applications to directly express and enforce many security constraints that are implicit and can only be enforced via programmer discipline in conventional programming systems. Our case study thus provides preliminary evidence for the practicality and benefits of designing future languages with Wyvern's feature set: static types, capability safety, abstract effects, and language extensibility.

Concretely, we found that a capability-based module system supports the creation of a hierarchy in which objects with powerful capabilities are able to produce new objects with more restricted capabilities. This allows for either precise or granular restrictions on the privileges of new modules, depending on the designer's requirements. An effect system that requires explicit annotations is also beneficial for security, as it increases the transparency of operations on I/O resources and therefore can help limit the dangerous effects of third-party modules. Meanwhile, language extension can be used to defend against specific types of injection attacks—format strings in the case of our study. Each of these language design features contributes a means of defense against common security flaws, and as demonstrated by their use in Wyvern's I/O library, it is possible to use them to create code structured to prevent common security attacks.

We also observed limitations in the current design and implementation of Wyvern's security-related features. These

include the syntactic overhead of effects—particularly pronounced in the case of asynchronous I/O—as well as implementation-related limitations in Wyvern's abstraction facilities. We hope that our observations will inspire future research to overcome these limitations—and indeed, some recent and concurrent work has already begun to address them [6, 19].

References

[1] [n.d.]. OWASP Top 10 Web Application Security Risks. https://owasp.org/www-project-top-ten/

[2] [n.d.]. Permissions in the Java Development Kit (JDK). https://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html

[3] 2015. Format string attack. https://www.owasp.org/index.php/Format_string_attack

[4] 2019. SecurityManager (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/lang/SecurityManager.html

[5] Aslan Askarov and Andrei Sabelfeld. 2005. Security-Typed Languages for Implementation of Cryptographic Protocols: A Case Study. In *Computer Security – ESORICS 2005*, Sabrina de Capitani di Vimercati, Paul Syverson, and Dieter Gollmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 197–221.

[6] Anonymous authors. 2020. Bounded Abstract Effects. In *In submission to OOPSLA*.

[7] J. M. Bishop. 1998. Java as a Systems Programming Language: Three Case Studies. In *Proceedings of the IFIP TC2 WG2.4 Working Conference on Systems Implementation 2000 : Languages, Methods and Tools*.

[8] Joshua Bloch. [n.d.]. *How To Design A Good API and Why it Matters*. https://www.youtube.com/watch?v=heh4OeB9A-c

[9] Gilad Bracha, Peter Von Der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules as objects in newspeak. In *European Conference on Object-Oriented Programming*. Springer, 405–428.

[10] Matthew Flatt. 2012. Creating Languages in Racket. *Commun. ACM* 55, 1 (Jan. 2012), 48–56. https://doi.org/10.1145/2063176.2063195

[11] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. 2012. Hails: Protecting Data Privacy in Untrusted Web Applications. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 47–60.

[12] Li Gong. 2011. Java Security Architecture Revisited. *Queue* 9, 9 (Sept. 2011), 30–36. https://doi.org/10.1145/2030256.2034639

[13] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. 1997. Going beyond the Sandbox: An Overview of the New Security Architecture in the JavaTM Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems* (Monterey, California) *(USITS'97)*. USENIX Association, USA, 10.

[14] Dexter Kozen. 1999. Language-based security. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 284–298.

[15] Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. 2015. A Theory of Tagged Objects. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 174–197. https://doi.org/10.4230/LIPIcs.ECOOP.2015.174

[16] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types, In Mathematically Structured Functional Programming. https://www.microsoft.com/en-us/research/publication/koka-programming-with-row-polymorphic-effect-types-2/

[17] D. Liebgold. 2011. Functional mzScheme DSLs in game development. (2011). Presented at Commercial Users of Functional Programming.

[18] John M. Lucassen. 1987. *Types and Effects towards the Integration of Functional and Imperative Programming*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[19] Julian Mackay, Alex Potanin, Jonathan Aldrich, and Lindsay Groves. 2019. Decidable Subtyping for Path Dependent Types. *Proc. ACM Program. Lang.* 4, POPL, Article 66 (Dec. 2019), 27 pages. https://doi.org/10.1145/3371134

[20] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-Based Module System for Authority Control. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 20:1–20:27. https://doi.org/10.4230/LIPIcs.ECOOP.2017.20

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

[22] Mark Miller, Ka-Ping Yee, Jonathan Shapiro, and Combex Inc. 2003. *Capability Myths Demolished*. Technical Report.

[23] Mark Samuel Miller. 2006. *Robust composition: Towards a unified approach to access control and concurrency control*. PhD dissertation. Johns Hopkins University.

[24] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 228–241. https://doi.org/10.1145/292540.292561

[25] Max S. New, Dustin Jamner, and Amal Ahmed. 2019. Graduality and Parametricity: Together Again for the First Time. *Proc. ACM Program. Lang.* 4, POPL, Article 46 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371114

[26] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. 2014. Safely Composable Type-Specific Languages. In *ECOOP*.

[27] Daniel Patterson and Amal Ahmed. 2017. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 12:1–12:15. https://doi.org/10.4230/LIPIcs.SNAPL.2017.12

[28] Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/158511.158524

[29] J. H. Saltzer and M. D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (Sep. 1975), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[30] August Schwerdfeger and Eric Van Wyk. 2009. Verifiable Composition of Deterministic Grammars. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM.

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

[32] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible Dynamic Information Flow Control in Haskell. *SIGPLAN Not.* 46, 12 (Sept. 2011), 95–106. https://doi.org/10.1145/2096148.2034688

[33] Joel Kamdem Teto, Ruth Bearden, and Dan Chia-Tien Lo. 2017. The Impact of Defensive Programming on I/O Cybersecurity Attacks. In *Proceedings of the SouthEast Conference* (Kennesaw, GA, USA) *(ACM*

SE '17). ACM, New York, NY, USA, 102–111. https://doi.org/10.1145/3077286.3077571

[34] Franklyn A. Turbak and David K. Gifford. 2008. *Design Concepts in Programming Languages*. The MIT Press.