A Comparative Study of Traditional versus Capability-Based Module Systems for Modern Programming Languages

Abhaas Goyal [1], **Alex Potanin** [2] **and Jonathan Aldrich** [3]

[1, 2] *Australian National University, Australia*
[3] *Carnegie Mellon University, USA*

Abstract

The principle of least privilege serves as an essential guideline in designing secure computing systems. However, implementing this in real-world systems through various programming languages has proved to be difficult and has allowed for many vulnerabilities in privilege escalation. One proposed solution is to have capability-based security primitives in programming languages for modules and objects. A capability is a unique token that provides the authority to perform a specific set of actions on a selected resource. However, its effectiveness as a language design choice for real-world applications remains to be seen.

To answer this question, we designed a comparative study to compare programmer productivity, security of the designs, and extensibility of packages in capability-based module systems vs. others. Our main goal was to determine whether module systems/packages having capabilities from the ground up provide usability and security advantages compared to their absence. The study used two programming languages - one with object capabilities (Wyvern) and the other with support for capabilities via external libraries (Rust).

Preliminary findings show that programs designed in Wyvern provided higher security guarantees in some cases, and users found using object capabilities an easy-to-use secure abstraction layer for managing critical resources. However, a lack of tooling for showing appropriate errors or code completion introduced challenges in writing code. Hence, future work requires a more in-depth study to define and validate current user-centric methods of designing capability-based languages. Further work also involves classifying security vulnerabilities solved by capabilities and building the necessary tools in Wyvern to make capabilities more viable as a design choice in programming languages.

Keywords: Object Capabilities. Language-based Security. Module Systems. Rust. Wyvern.

1 Introduction

A rise of sophistication in cyber-security attacks has led to increased research in language-based security [1], which attempts to provide computer security for applications at an architectural level. One of the core principles to follow in their designs is the principle of least privilege - which states that every user and process should be provided with minimal authority over the underlying system resources.

Initial attempts to achieve authority control in systems centered around two symmetric approaches. [2] The first was the identity-centric model, which was implemented using access-control lists. The other one was the authority-centric model known as capabilities, which kept designation and authority together. The former became the industry standard for various historical reasons leading to myths around using capabilities in production systems [3].

The Open Worldwide Application Security Project (OWASP) provides a regularly updated list of the top 10 list of "most critical security risks faced by organizations in web applications" [4]. Within the list, **Broken Access Control** has been put at the number one spot [5]. Several subsets of vulnerabilities exist within this category- such as *Execute Code*, *Directory Traversal*, and *Gaining Privilege*. Recently, research into capabilities has been re-assessed as a potential solution by the research community. Capabilities have been implemented in a wide variety of contexts, such as standard library packages in various programming languages ([6]–[8]), Operating Systems such as sel4 [9], and even in hardware architectures [10].

In addition to building capability-based systems based on libraries, we can design programming languages that use capabilities from the ground up to design secure programs. In the past, this

PLATEAU
13th Annual Workshop at the Intersection of PL and HCI

DOI: 10.35699/1983-3652.yyyy.nnnnn

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

This work is licensed under a Creative Commons Attribution 4.0 International License.

has been attempted both in older languages such as E [11] and Newspeak [12], as well as modern languages such as Pony [13] and Wyvern [14].

However, it remains to be seen whether regular software developers can use capability-based languages in teams to design applications and the effect of doing so on introduced security vulnerabilities, compared with a library-based approach. To answer this question, we frame the following research questions for the project:

1. **RQ1**: How usable are capability-based languages when designing an architecture?
2. **RQ2**: What security vulnerabilities do programmers expose when designing resource-critical objects?

To answer these questions, we set up a comparative study between two languages - one with object capabilities (Wyvern) and the other with support for capabilities via external libraries (Rust). The two languages were chosen based on the following parameters:

Reasons to choose Wyvern

1. Provides a capability-based module-system with formalized authority-control [15]. Here, system resources are represented as object capabilities and must explicitly be passed as arguments, limiting access to certain system resources. This feature will help us to design a multi-tiered architecture while controlling security in each layer.
2. Domain-specific syntax support within the language provides good support for embedding other languages (such as in the case of web development HTML/SQL), so users can have an easier time designing architecture with high safety/security properties.
3. A potential future scope of the Wyvern project lies in studying the usability and security of Effect Systems in conjunction with capabilities. A comparable language is Koka [16], but it's effect system does not have a focus on security since it does not support the principle of information hiding.

Reasons to choose Rust

1. In terms of choosing a capability library among modern languages, the Rust library was found to be the most stable. (Object-Capabilities in Scala [7] was another candidate but the library is currently in beta).
2. Considered as the most loved language on Stack Overflow [17], so a strong case for increasing future user adoption rates.
3. Supports multiple paradigms of programming, such as imperative and functional programming, which increases suitability for getting more users for the study.
4. Systems language with a strong type system, known to be secure due to ownership, so it would be interesting to see security vulnerabilities arising from using this language.

Our findings are in their preliminary stages. They show that programs designed in Wyvern provided higher security guarantees, and users found that object capabilities provided an easy-to-use yet a secure abstraction layer for critical resource management. However, a lack of tooling for showing appropriate errors or code completion introduced ease of use challenges when writing programs. Informed by these findings, future work is required primarily in designing a more in-depth study to understand the user-centric methods needed to design capability-based languages. Further work also involves classifying security vulnerabilities solved by capabilities and building the necessary static analysis tools/debugging aids in Wyvern to make capabilities more viable as a design choice in programming languages.

2 Background and Related Work

We provide a background on the motivation for using capability-based design and a sample capability pattern used in one of the study designs.

2.1 Capabilities: Motivation

To start with why we need capabilities, we first need to look at the Confused deputy problem [18]. In Figure 2, Bob acts as the deputy and is deemed trustworthy by every other entity (hence the term "deputy"). He is communicating with Alice and Carol. Alice then provides Bob with a sensitive resource (in this case `/etc/passwd` with specific permissions) and trusts Bob enough not to pass the sensitive resource to Carol. The question then arises whether Carol can "trick" Bob into access to the underlying resource, thus leading to broken access control. Currently, many real-world vulnerabilities are present in this class (due to ambient authority) with common examples being Cross-Site Request

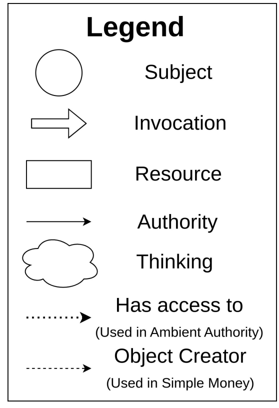

Figure 1. Legend for various entities in object-capability diagrams

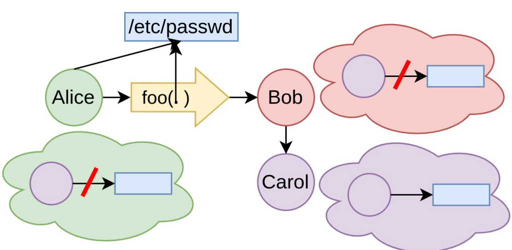

Figure 2. The Confused Deputy Problem represented as a Granovetter Diagram [18]

Forgery (CSRF) [19] and VSCode extensions ([20]).

However, Rust provides module systems that don't have ambient authority, for which one could argue that these bugs would not happen in the future (i.e., we consider Bob won't be pinned as a reason for broken access control). This further propagates the question of whether capabilities as a language design choice by itself provide additional advantages.

[18] further discusses object-capabilities security as one of the methods to avoid the confused deputy problem. Capabilities can be viewed as one of the interpretations of the Granovotter diagram. It is defined as the following:

If Bob does not have access to Carol (which is a critical resource), he can only gather access to it from someone else. Let us assume it's Alice; she can only give the resource to Bob only if:

- Alice already has the authority to Carol object (via a reference)
- Alice has a reference to Bob
- (Key part) Alice decides to voluntarily share the reference to Carol with Bob

Capabilities can be summarised as - "Don't separate authority from designation." [3]. The design choice in creating programs with capabilities is for a main trusted program to provide the correct amount of capability (which is implemented as non-forgeable references) to each specific object.

2.2 Capability Patterns: Sealer-Unsealer

According to [21], sealer/unsealer pairs (Figure 3) can be conceptually viewed as similar to public/private key pairs. They can be used to control rights to access a certain object as follows:

1. **Initialisation** The main program creates the sealer/unsealer pair, and passes it to Alice.
2. **Sealing the resource** Alice invokes `seal` from a sealer object. Internally, `Sealer` creates a Sealed box, and returns it to Alice. This is represented by steps (1)-(3) in Figure 3. Considering that Alice only has access to the Sealed Box, she can pass it around without having to worry about others accessing the critical resource.
3. **Unsealing the resource** However, Alice must be careful with passing the Unsealer around. If she passes it to another entity (this is acceptable within the design of the language since behavior is defined in trusted code, and not at runtime), they now have the ability to call unseal and gain access to the inner object. This is represented by steps (4)-(6) in Figure 3.

This pattern will be used in the study design of Simple Money, which is given in section 4.3.

3 Methodology

To gain insight into the usability of capable and incapable module systems, we set up a qualitative language usability study. For that, we applied for Human Ethics approval from ANU Research Information Enterprise System [22] for approval. The resulting methodology comprised the following aspects:

Recruitment Participants were recruited within the School of Computing at the Australian National University. The method of recruitment was to contact potential participants via email or face-to-face interaction. Some of the places include:

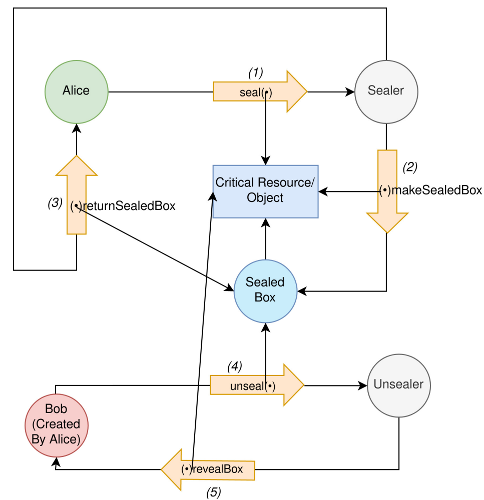

Figure 3. Sealer-Unsealer Architecture

- Personal/Professional connections
- Face-to-face interactions with club members of the Computer Science Student Association (CSSA)
- Students completing research in programming languages

In recognition of the participant's time, we renumerated them with a AUD$20 or AUD$30 Westfield voucher, depending on availability.

Participant	Background	Programming Experience	Task
P1	Parallel Systems	5 years	Logger Editor
P2	Systems/Security	7 years	Network Pool
P3	Formal Methods	3 years	Network Pool
P4	Data Science	5-6 years	Simple Money

Table 1. Participant Details

Participants Participants at ANU have been chosen because of close proximity to the place where the interviews had to be conducted. To maintain an appropriate level of experience and skills for the pilot study, senior undergraduate/postgraduate students were chosen. The participants should have the necessary background to be able to solve the proposed problem and answer questions during the interview time frame. They had chosen from varied backgrounds to get a more representative study for language designers. Their details are provided in Table 1.

Considering that there are only four participants, the reader may assume a low number of participants is a threat to the validity of the result. However, previous successful studies using thematic analysis have been conducted with only four participants [23], and using think-aloud protocol with six participants [24]. The papers had the central theme of designing the setup of the study design open-ended to conduct an in-depth analysis of individual studies; and screening tests of the participants.

Procedure We conducted the study in the format of a semi-structured interview and following the think-aloud protocol. A set of tasks were designed that can be given to either software architects or experienced software engineers that involved designing or extending a small product architecture.

We use the think-aloud protocol by building on studies done to understand novice programers' strategy who had little experience in programming ([24], [25]). They partially apply in our case, considering the participants have had no experience in Wyvern, and two out of four participants had prior experience with Rust. However, it should be noted that our pre-screening of participants requires that they have the appropriate level of knowledge in computer science. [24] also used narrative analysis, whereas this research project aims to generate results by thematic analysis. [25] is in the context of developing computational thinking in general; however, our goal is much more specific - to use computational thinking at a higher abstraction level to design modular architectures.

We asked participants to first use a language with support for modules and object capabilities (e.g.

Wyvern), and then use a more traditional language with external capability libraries (e.g. Rust/Java) to design the same architecture. Documentation links related to standard capability libraries in Rust ([26]–[28]) were provided as starting points. After this, we asked them to break the security of the written program itself for any/both languages, to reveal potentially overlooked vulnerabilities.

Finally, we asked post-interview questions consisting of a survey (details of questions are provided in appendix A). The total length of each study was 90 minutes, with each section having a rough outline of the overall time distribution:

1. **Rust and Wyvern Implementation** (60 mins)
2. **Trying to break security of the Program** (20 mins)
3. **Post-study survey** (10 mins)

Occasionally, we took notes regarding how the discussion went, and marked points of potential interest to double down on when conducting thematic analysis. If a student was stuck on a part, we asked them to explain their thinking process using the think-aloud protocol.

Data Collection The data being collected was personal information about the interviewee's previous experience in software development, domain of expertise, and current role. During the interview, the information being collected was: (a) Screen recording of the desktop environment (b) Audio transcript (c) A survey at the end of the interview (full-list of questions provided in appendix A).

Analysis Thematic analsysis is now commonly used in the programming languages research community for various use cases, such as highlighting key challenges in language design [29] and its tooling, and debugging [23]. The main goals of the referenced studies also apply to our overarching topic of determining the suitability of a language for a given task.

We conducted a thematic analysis of the code, audio-recorded interview and the post-interview survey to derive potential hypotheses. The analysis was conducted by keeping two major themes in mind: The usability of both languages and any Security Vulnerabilities that could have arisen. In our case, the procedure would look like the following: If one is taking an audio-recorded interview, the initial step is transcribing the sampled data. Our data sources also consist of screen sharing, so it is essential to mark points of interest (errors in code, completion, etc.) as well. The next step is to look for recurring themes from each interview. We follow this up in Section 5.

4 Study Design

The overall study consists of three parts. In each part, we ask participants to implement specific parts of the architecture using capabilities via external libraries or via module systems in languages. Each part was designed to primarily test one of the facets of the overall goal (usability, extensibility, and security); however, all facets were considered in the analysis of each part.

Each study part was divided into three steps for participants - designing the initial architecture, finding vulnerabilities, and the post-study survey.[1]

4.1 Logger Editor

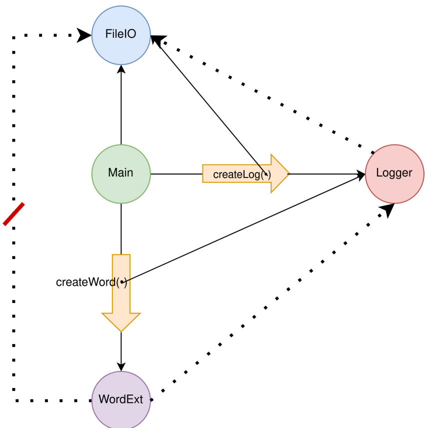

Figure 4. Logger Editor

1 Code templates for the studies are available at https://github.com/abhaasgoyal/module-systems/tree/plateau-2024

Purpose To assess the security of participant code, ensuring that authority is non-transitive.

Background In figure 4, the `Main` application has a `Logger` module that it can trust. The logger module has access to `FileIO`, providing it the authority to read/write to files. However, there also exists an extension named `wordCloud` module, which needs to utilize the `Logger` library to read/write to a place that only `Logger` can allow. Here, `Main` passes `Logger` when creating `wordCloud`. The goal of the user is to design `Logger` in such a way that `WordExt` does not have access to the underlying module `fileIO` and, in the process, escalate its privilege.

4.1.1 Instructions

Rust Implementation The implementation in Rust was based on the `cap-std` module [6]. The implementation should contain the functionalities for the following (note that one can change the function names in the Extension depending on how the Logger is structured):

- `create_logger(logFile: String)` - A constructor which returns a new logger object with the name logFile
- `logFile.append_to_log(entry : String)` - Append a new entry to the logFile

Given a possibly malicious `extension.rs`, design the corresponding Logger module with capability library in Rust.

Wyvern Implementation The extension library is `wordCloud`, and the users need to design the logger library from scratch with the `appendLog` function. Since capabilities are hierarchical here, the users were given a more open-ended specification of the parameters of the logger library; the solution is that the underlying `fileIO` system resource is passed exclusively to the logger from Main.

There was no developer-friendly documentation available for Wyvern's standard I/O library, so we provided the implementation, hoping that it will act as self-documenting code. The required method definitions for `fileIO` are provided in listing A.2.

Trying to Break security Upon completing the corresponding functions, users were asked to find a way to bypass the restriction of accessing `fileIO` in the corresponding programs, while only modifying `Extension.rs` (for Rust) and `wordCloud` module (for Wyvern).

4.2 Network Pool

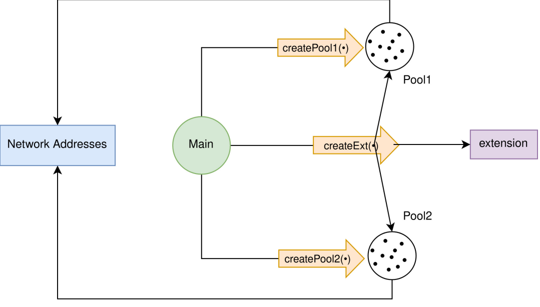

Figure 5. Network Pool (A dotted circle is the capability to connect to a specific device)

Purpose To assess the ease of developing appropriate implementations from the specification. A high degree of freedom should be provided in how to design programs.

Goal Firewalls are essential to control incoming and outgoing network traffic. The user has to implement a basic firewall by making two kinds of network pools that allow an untrusted extension to use the network in limited ways. The extension promises to only connect to the website `example.com`. The architecture can be represented with figure 5. There are two types of pools used in the Extension. The pools should limit the extension to the following authorities, respectively:

- **TCP-Port** Only allow connections to an IP address of `93.184.216.14` using a TCP port (0-65535). For Wyvern, it can be assumed that the IP address is a string of fixed length i.e. it has a length of 15. For example, `192.68.1.1` is represented as `192.068.001.001`
- **Net-Port** Only allow connections to a small range of IP addresses (but with any port allowed). The last 8 bits of the IP addresses should be in the range `93.184.216.<0-255>`

4.2.1 Instructions

Rust Implementation Within `pool_auth.rs`, create the respective network pools by looking at the necessary documentation. Then, call in the Extension by passing in the Pools with the required IP address and HTTP port in both cases as the input. In this case, the necessary documentation was regarding Network Pools implementation in capability standard library of Rust [30].

Wyvern Implementation The `makePool` module should have 4 input parameters - `startIp`, `endIp`, `startPort`, and `endPort`. Then, come up with an abstraction of functions for Net-Port and TCP-Port, which call `makePool`. This should be within the main function. Finally, the function `connect(addr, port)` should consist of a guard which checks whether the `addr` and `port` are within the acceptable range, and connects if the check succeeds.

Trying to Break security This exercise was similar to the Logger-Editor design, but now considering multiple capabilities that are passed to the Extension. Upon completing the corresponding functions, try to break the security of the filesystem in the corresponding programs by modifying only `Extension.rs` (for Rust) and the `cloud` module (for Wyvern).

4.3 Simple Money

Note: The design was motivated by the simple money example [21].

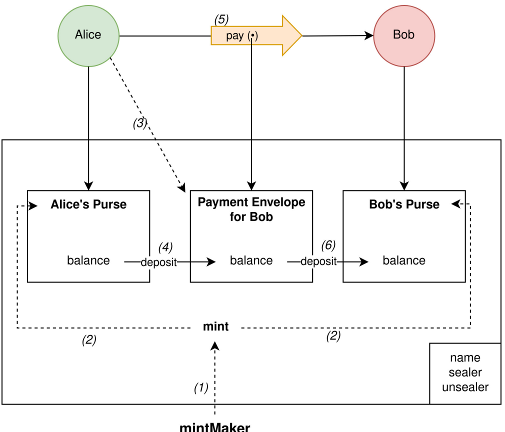

Figure 6. Simple Money

Goal Extensibility of adding code to existing codebases.

Background Given a money minter and two purses A and B, design a transaction where user A can securely send money to user B, using the capability pattern of sealer-unsealer.

Architecture The architecture (figure 6) consisted of the main entity `mintMaker`, making the `Mint` Object, representing a new currency. It has a fixed amount of total balance. It employs the Factory Pattern to further create two `Purse` objects, Alice and Bob, which it can initialize with a certain balance. This is implemented in steps (1)-(2). Now, the question is whether Alice can pay some of her money to Bob while conserving the total currency. Some of the goals the architecture needs to achieve [21] are:

- (T1) Only someone with the mint has the power to change the total balance of that currency
- (T2) `Purse A` cannot change the balance of `Purse B`
- (T3) Balances should always be positive
- (T4) If a successful deposit gets reported, Alice should be guaranteed that the deposit was made to the other wallet

4.3.1 Instructions

The participant is given implementations of the Sealer and Unsealer primitives, as well as the Mint object (steps (1)-(2) in the diagram). The user tasks for both languages were to securely transfer money via an intermediate Purse Object with the Sealer-Unsealer pattern. The purse should have the following methods:

1. `purse.getBalance(): Int` - Get the current balance in the purse
2. `sprout(): Purse` - Create a new empty purse

3. `getDecr(): SealedBox[Int -> void]` - Get a sealed version of `decr`. A hint was provided that should be used to validate (T4) during a `deposit` to Bob's purse. `decr` is a function that subtracts the balance in the current Purse
4. `dst.deposit(amount:Int,src:Purse):void` - Securely transmits money from one wallet to another
5. `print():void` - (Optional) Print debugging information

The programmer's expected steps are to understand the respective codebase and extend the program's functionality. Instructions were the same for Wyvern and Rust implementation, with slightly different filenames. They were to implement the architecture above and then come up with potential vulnerabilities in their implementation.

5 Results

After conducting the study with four participants, we now construct a thematic analysis and derive hypotheses for each research question.

5.1 RQ1: Usability of capabilities in the language vs. library

5.1.1 Understandability of existing code and Extensibility

(P4) mentioned that understanding the code for Simple Money was highly complicated in Rust, even though they had past experience. In particular, it was related to `dyn` keyword and its use in traits. This proved to be much easier in Wyvern; however, it may be because Wyvern's implementation matches better with the original problem in terms of objects. Rust's implementation of Simple Money also had all three objects for building objects and their behavior: `trait`, `struct`, and `impl`, which in general, caused a higher cognitive load for the users. This confused the participant as to how different modules were interacting. Another major conceptual hurdle three participants found in Rust was `.unwrap()` vs `?` operator (i.e. the `Option` vs `Result` type).

In comparison, most concepts in Wyvern came more easily to participants. An example of this would be `var` vs. `val`. (P2, P4) mentioned they were easy to understand and helped them gain quicker results in Wyvern's implementation. However, there were some concepts in Wyvern that participants had difficulty understanding. A major one was the difference between `module def` and using plain `module` when defining abstractions. For `module def`, most participants were provided with the analogy of the parameters as constructors (data members) along with function definitions. However, (P1) raised an interesting question on `wordCloud`, where both `WordFactory.Word` and `WordFactory` types had been passed as parameters. They asked why both needed to be passed when only `Word` was needed in the implementation function. Here, the capability to access a value of type `WordFactory.Word` itself depends on whether we have the existing capability for `WordFactory` - thus following the rule of no ambient authority. Here, the Main module has to pass both the top-level capability object and sub-capability object to different modules. This could potentially lead to many parameters in module definitions in large systems, which can be considered as a case of ***overhead*** of using capabilities.

5.1.2 Writing Syntax

One of the hurdles faced by participants was that they needed to learn the language syntax better, so the process of writing up the correct syntax could have been faster for both languages. Even though tutorials were designed for each language to implement the specific question to improve productivity, they were limited in scope due to the time constraints of the study itself (we acknowledge this as a limitation of the study design). Although the intention of the study was designed for participants with a general computer science background, having some pre-reading on the documentation of the specific languages before the interview would save significant time and help with more complex studies. This could be achieved with a pre-survey on those specific languages before recruiting.

(P1, P3) felt that a lot of the code in Rust was boilerplate in terms of using appropriate type definitions. This was due to Rust's complex set of borrowing rules. For example, (P1)'s implementation of the logger module's had to change the type of extension to mutable as well (since they structured their logger with the file reference instead of location, and File is mutable in `cap-std`. In comparison, users liked that they had to write less code in Wyvern without sacrificing stability.

Finally, users sometimes found it helpful to have code completion hints in Rust, of which none exist in Wyvern. However, participants faced no significant problems in terms of the time taken to

write the program and complete the study on time.

5.1.3 Errors and Debugging

Rust Most of the participants found Rust's error messages to be helpful in figuring out bugs, but only once they understood the specific concept. A participant even mentioned that the error messages popping up and fixing them helped the participant learn the language in a short period of time. However, the author would like to be cautious about being totally dependent on errors generated by the compiler. For example, (P2) faced an error in Rust, which suggested that the error was related to invalid types; however, the actual error was that they had missed a semicolon (;) in the previous line. This simple error, with an unrelated message, took a lot of time for the participant to debug.

Wyvern All participants noted that Wyvern was highly sensitive to bugs in lexical analysis and parsing, which made it hard to debug the exact cause of the issue. This has to do with Wyvern being in its initial stages of development. However, they used line numbers in the error message to estimate where the bug resided and tried to change the code on a trial-and-error basis. In cases where the participant could not solve the issue, we acted as the manual compiler and helped the participant fix errors in this case. It is also essential to see that Rust's documentation is more comprehensive than Wyvern's, with answers to specific questions on websites like StackOverflow. Hence, if the participant was stuck for a long in a problem with Wyvern, we acted as the StackOverflow entity in that scenario. One of the most common errors faced by participants was mismatching function/type definitions in Wyvern's specification and implementation. For example, (P3) realized that they needed to change the following module definition when creating connect:

```
// Original code
module def poolMaker(startIp: String, endIp: String, startPort: Int, endPort: Int): PoolMaker
def connect(address: String, port: String): String
// Proposed change
def connect(address: String, port: Int): String
```

However, this change requires changes in `PoolMaker` module as well:

```
resource type PoolMaker
   def connect(address: String, port: Int): String
```

From the points above, we arrive at a hypothesis on the usability of capable and incapable languages:

> **Hypothesis 1** (Usability)**.** *With fewer keywords in the language specification, Wyvern is easier to learn in terms of syntax. Furthermore, compared to Rust, it provides better structure in terms of reducing cognitive overhead when designing programs. However, there is a desire for (a) minimizing the capability overhead on expressing capabilities in terms of language syntax and (b) improving the debugging, documentation, and code completion, which accounts for factors in the lack of usability for the language.*

5.2 RQ2: Analysing Security of Capability-Designed Languages

In total, two types of vulnerabilities were found among the participants:

- One of the participants found the vulnerability in Network Pool (4.2) in Rust. They circumvented the need to use a specific Pool object by importing `cap-std` within the extension. When the extension function is called, they can now create a new Network Pool with **any** privilege and connect to the network from there. There are no checks by the network connector whether the extension module itself had the capability to access specific addresses. This could also be carried on for the Logger Editor Example in terms of file systems. The participant could not think of a solution in the time frame. Now, this raises an interesting point. The Main module has to look for all types of imports within the whole codebase. Based on that, trust the extension. However, this is not feasible in large software, where developers interact via function API documentation. This problem was not faced in Wyvern, which could provide a potential use case for using capability-based language design.
- (P4) found a vulnerability in creating additional Minters in both Rust and Wyvern; however could not break the security of a single transaction. As such, both architectures were equally secure in terms of implementation.

Thus, we reach the following hypothesis on the security of capable and incapable module systems:

> **Hypothesis 2** (Security). *It was seen that capabilities ensured additional levels of security compared to other modern programming languages in the context of trusting the API definition. However, to see wider adoption from language designers, more work is required to classify more security bugs that capabilities are particularly suited to solve.*

5.3 Threats to Validity

It should be noted that the hypotheses state are derived from preliminary results. Therefore, a more thorough study is required to validate them, which would address the following threats to validity:

Participants All recruited participants were from the same institution, all of whom were recent graduates or undergraduates. We attempted to resolve this by having a brief pre-screening with the participants. We asked them general questions regarding computer science concepts, which would ensure that participants had the appropriate knowledge to implement concepts in the study. However, it should be seen that the participants did not have much experience with designing software architecture in the industry, which would lead to more security bugs in the underlying code and a limitation in detecting potential bugs.

Prior Experience Considering that Wyvern is currently under research and Rust is an upcoming language in the mainstream, not many people have had prior programming experience in the specific languages. This could have made programming slower.

Study Procedure: Considering that we were asking the same subjects to break their own code, most of them (except one participant) could not think of a way on how to do this. The order in which these languages were studied is also important since participants already had a fair idea of the problem when implementing it in a different language. This could be one of the reasons why participants found it easier to code in Wyvern than in Rust. More anecdotal evidence conducted in a suitable design would answer this question. For this, a potential alternative study design is to give participants another participant's code to break the security (similar to penetration testing in companies) since finding vulnerabilities from a different solution is from a different experience. An experimental design that limits cross-condition effects would be suitable here.

6 Concluding Remarks

6.1 Conclusion

We conducted a comparative study of traditional versus capability-based module systems by interviewing four upper-level undergraduate students. In doing so, we identified two main hypotheses relating to the usability and security of programs in capability-based languages to help programmers be productive and motivate further work in researching further secure systems.

We hope these hypotheses will improve the understanding of reasons to design secure programming languages via capability-based language systems. Furthermore, if principles from the study are used to provide better tooling for existing capability-based languages, we hope that it will have broad benefits in terms of writing better software in terms of programmer productivity, security, and extensibility.

6.2 Future Work

Current results show that we have only scratched the surface in this problem domain. Further avenues for research mainly include improving the design of the study based on the following:

Larger target audience Preferably working professionals who are domain experts with at least one of the programming languages being surveyed. We found that our study needed quantitative analysis and collect more anecdotal evidence to support our claims. The current sample size is small for doing any of the two. This would also help us use more complex software design problems during interviews.

Including more modern programming languages For a more comprehensive comparison and seeing what features are needed in capability-based designed languages to make it more usable for general programmers without sacrificing security. The current study has only two programming languages with widely different syntaxes, so having different studies with closer languages would provide a better benchmarks in evaluating usability.

Finding more Security Vulnerabilities The study designs should be based on existing CVE vulnerabilities. Vulnerabilities classified in CVEs are based on large-scale software, where decisions are made on an architectural level. We can design studies about employing capabilities patterns on larger systems to achieve this.

References

[1] A. Askarov and A. Sabelfeld, "Security-typed languages for implementation of cryptographic protocols: A case study," in *Computer Security – ESORICS 2005*, S. d. C. di Vimercati, P. Syverson, and D. Gollmann, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 197–221, ISBN: 978-3-540-31981-8.

[2] M. S. Miller, *MarkM's Opening Statement on SOSP Panel*, 2018. [Online]. Available: https://www.youtube.com/watch?v=br9DwtjqmVI.

[3] M. Miller, K.-p. Yee, and J. Shapiro, "Capability myths demolished," Dec. 2003.

[4] OWASP, *OWASP Top Ten | OWASP Foundation — owasp.org*, https://owasp.org/www-project-top-ten/, [Accessed 29-May-2023], 2021.

[5] OWASP, *Broken Access Control | OWASP Foundation — owasp.org*, https://owasp.org/Top10/A01_2021-Broken_Access_Control/, [Accessed 29-May-2023], 2021.

[6] Rust, *Capability-oriented version of the rust standard library*, https://github.com/bytecodealliance/cap-std, 2023.

[7] Scala, *Ocaps: Object-capabilities in scala*, https://github.com/tersesystems/ocaps, 2023.

[8] Golang, *Cap'n proto library and code generator for go*, https://github.com/capnproto/go-capnp, 2023.

[9] G. Klein, J. Andronick, K. Elphinstone, *et al.*, "seL4: Formal verification of an operating-system kernel," *Communications of the ACM*, vol. 53, no. 6, pp. 107–115, Jun. 2010. DOI: 10.1145/1743546.1743574.

[10] R. N. Watson, J. Woodruff, M. Roe, S. W. Moore, and P. G. Neumann, "Capability hardware enhanced risc instructions (cheri): Notes on the meltdown and spectre attacks," University of Cambridge, Computer Laboratory, Tech. Rep., 2018.

[11] J. E. Richardson, M. J. Carey, and D. T. Schuh, "The design of the e programming language," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 15, no. 3, pp. 494–534, 1993.

[12] G. Bracha, "Newspeak programming language draft specification version 0.06," Technical report, Ministry of Truth, Tech. Rep., 2009.

[13] G. Steed and S. Drossopoulou, "A principled design of capabilities in pony," *Master's thesis, Imperial College*, 2016.

[14] D. Kurilova, A. Potanin, and J. Aldrich, "Wyvern: Impacting software security via programming language design," in *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools, Portland, OR, USA, October 21, 2014*, J. Sunshine, T. D. LaToza, and C. Anslow, Eds., ACM, 2014, pp. 57–58. DOI: 10.1145/2688204.2688216. [Online]. Available: https://doi.org/10.1145/2688204.2688216.

[15] D. Melicher, Y. Shi, A. Potanin, and J. Aldrich, "A capability-based module system for authority control (artifact)," *Dagstuhl Artifacts Ser.*, vol. 3, no. 2, 02:1–02:2, 2017. DOI: 10.4230/DARTS.3.2.2. [Online]. Available: https://doi.org/10.4230/DARTS.3.2.2.

[16] D. Leijen, "Koka: Programming with row polymorphic effect types," *arXiv preprint arXiv:1406.2061*, 2014.

[17] Stack Overflow, *Developer survey*, 2022. [Online]. Available: https://survey.stackoverflow.co/2022/.

[18] M. S. Miller, *The confused deputy*, 1997. [Online]. Available: http://erights.org/elib/capability/deputy.html.

[19] J. Blatz, "Csrf: Attack and defense," *McAfee® Foundstone® Professional Services, White Paper*, 2007.

[20] CVE, *Vscode: Cve vulnerabilities*, 2023. [Online]. Available: https://www.cvedetails.com/vulnerability-list/vendor_id-26/product_id-50646/Microsoft-Visual-Studio-Code.html.

[21] M. S. Miller, C. Morningstar, and B. Frantz, "Capability-based financial instruments," in *Financial Cryptography, 4th International Conference, FC 2000 Anguilla, British West Indies, February 20-24, 2000, Proceedings*, Y. Frankel, Ed., ser. Lecture Notes in Computer Science, vol. 1962, Springer, 2000, pp. 349–378. DOI: 10.1007/3-540-45472-1_24. [Online]. Available: https://doi.org/10.1007/3-540-45472-1%5C_24.

[22] ARIES, *ANU Research Information Enterprise System*, 2023. [Online]. Available: https://services.anu.edu.au/training/human-ethics-aries-training.

[23] R. (Huang, E. Pertseva, M. Coblenz, and S. Lerner, “How do Haskell programmers debug?,” Mar. 2023. DOI: 10.1184/R1/22277347.v1. [Online]. Available: https://kilthub.cmu.edu/articles/conference_contribution/How_do_Haskell_programmers_debug_/22277347.

[24] J. Whalley and N. Kasto, “A qualitative think-aloud study of novice programmers' code writing strategies,” in *Proceedings of the 2014 conference on Innovation & technology in computer science education*, 2014, pp. 279–284.

[25] S. Y. Lye and J. H. L. Koh, “Review on teaching and learning of computational thinking through programming: What is next for k-12?” *Computers in Human Behavior*, vol. 41, pp. 51–61, 2014.

[26] Rust, *A capability-based API modeled after std*, 2023. [Online]. Available: https://docs.rs/cap-std/0.26.1/cap_std/.

[27] Rust, *Capability-based standard directories*, 2023. [Online]. Available: https://docs.rs/cap-directories/0.26.1/cap_directories/.

[28] Rust, *The rust standard library*, 2023. [Online]. Available: https://doc.rust-lang.org/std/.

[29] M. Coblenz, A. Porter, V. Das, T. Nallagorla, and M. Hicks, “A Multimodal Study of Challenges Using Rust,” Mar. 2023. DOI: 10.1184/R1/22277326.v1. [Online]. Available: https://kilthub.cmu.edu/articles/conference_contribution/A_Multimodal_Study_of_Challenges_Using_Rust/22277326.

[30] Rust, *Capability-based network pools*, 2023. [Online]. Available: https://docs.rs/cap-std/1.0.15/cap_std/net/struct.Pool.html.

A Appendix

A.1 Post-study survery questions for Study Designs

1. How useful do you think capabilities are?
2. How much did you like working on Wyvern?
3. How much did you like working on Rust?
4. How much did you think you understand the concept of capabilities?
5. What are some things that you would have wanted to improve in the survey if you had the chance to do it all over again?

A.2 FileIO in Wyvern

```
import fileSystem.BoundedReader
import fileSystem.Writer
import fileSystem.RandomAccessFile
import fileSystem.BinaryReader
import fileSystem.BinaryWriter
resource type File
    effect Read
    effect Write
    effect Append
    def makeReader(): {} BoundedReader
    def makeWriter(): {} Writer[{this.Write, this.Append}]
    def makeAppender(): {} Writer[{this.Append}]
    def makeBinaryReader(): {} BinaryReader
    def makeBinaryWriter(): {} BinaryWriter
    def makeRandomAccessFile(mode : String): {} RandomAccessFile
```

A.3 Post-Study Survey

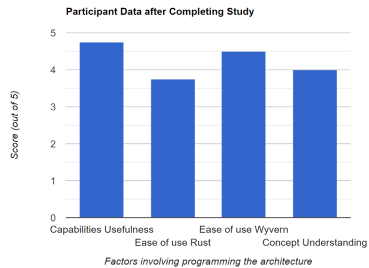

Figure 7. Quantitative Results for the Post-Study Survey (in order of questions asked in section 3)