Ca' Foscari University of Venice
Department of Environmental Sciences, Informatics and Statistics

A Modern Approach to Cryptography in Swift: Vapor's JWTKit Project

Computer Science Bachelor Thesis
Academic Year 2022/2023

Graduand: Paul Toffoloni
Supervisor: Prof. Filippo Bergamasco
Co-Supervisor: Tim Condon

A Giovanni e Irene
per avermi nutrito, di affetto e non solo

Acknowledgements

First and foremost, I extend my gratitude to Professor Bergamasco, whose guidance has been important throughout the journey of completing this thesis. My heartfelt appreciation also goes to Tim, who introduced me to the world of server-side Swift and still manages to teach me something new every day.

I owe a profound debt of gratitude to my parents and my family, whose emotional, financial, and all-encompassing support have been the pillars of my strength throughout my university years.

Special thanks to Ginevra, who helps me manage life when it is not so easily manageable and for being my partner in exploring new culinary horizons, whether in Bologna, Rome or New York.

I am grateful to Gabriele, for always being there to lend a hand with my thesis work, and to Luca, with whom I have shared the entirety of my university journey.

Finally, I want to thank all my friends, both in Trieste and in Venice, who I have shared so many important moments with, even if at distance.

Abstract

With Server-Side Swift gaining prominence in the realm of back-end development, the presence of a good JWT library has become extremely important as it enables safe and efficient data transfer between two parties, allowing for the verification of integrity and origin of the data. Vapor's JWTKit package has been the go-to library for developers who need JWT-based authentication in their Swift-based backend. The project behind this thesis aims to improve JWTKit's structure, mainly under the hood.

The thesis begins with an introduction to server-side Swift, explaining how the Apple-made language has evolved from being a closed-source Apple-only-client language to being open-sourced and being used on the server side by several companies. Vapor, although not the only one, is the most popular Swift-based web framework, used by big names, first of them being Apple itself.

The main focus of this thesis is redesigning Vapor's JWT library. This new version brings a number of improvements to the repository, the main one being the removal of Google's BoringSSL C library and its replacement with Swift-only implementations for cryptographic operations. This transition allows the library to reduce external dependencies and potential points of failure, also enhancing maintainability and scalability. Sendable conformance in the package is also explored, which aligns the package with Swift's latest concurrency model and enables thread-safe usage of the library. Finally, the new version also brings an improved API to the package, allowing for custom token de/serialisation, new signature algorithms, and more.

In conclusion, this thesis offers a detailed examination of JWTKit's version 5 release, making it a more valuable asset for developers working on server-side Swift applications.

Contents

Introduction **1**

1 An introduction to Swift **3**
1.1 Type System . . . 3
1.1.1 Type Safety . . . 3
1.1.2 Generics . . . 4
1.1.3 Protocols . . . 5
1.1.4 Type Erasure and Opaque Types . . . 6
1.2 Memory Management . . . 6
1.2.1 Automatic Reference Counting (ARC) . . . 7
1.2.2 Copy-on-Write Semantics (CoW) . . . 7
1.2.3 Optionals . . . 7
1.3 Swift's Concurrency Model . . . 8
1.3.1 Asynchronous Programming . . . 9
1.3.2 Structured Concurrency . . . 10
1.3.3 Actors . . . 11
1.3.4 Sendable Types . . . 13

2 Server Side Swift **15**
2.1 SwiftNIO . . . 15
2.1.1 Non-blocking I/O . . . 15
2.1.2 Event Driven Architecture . . . 16
2.1.3 EventLoops . . . 16
2.1.4 Channels and ChannelHandlers . . . 16
2.1.5 EventLoopFutures . . . 17
2.2 Vapor . . . 18
2.2.1 Application and entrypoint . . . 18
2.2.2 Routing . . . 19
2.2.3 Fluent . . . 19
2.2.3.1 Database Drivers and Setup . . . 20
2.2.3.2 Model . . . 20
2.2.3.3 Migrations . . . 21
2.2.3.4 Query . . . 22
2.2.4 Authentication and Authorisation . . . 23
2.2.4.1 Authentication . . . 23

2.2.4.2 Authorisation 24

3 JSON Web Tokens **27**
3.1 Structure of a JWT 28
3.2 Stateless Authentication 29
3.2.1 Cross-Site Request Forgery (CSRF) 30
3.2.2 Cross Site Scripting (XSS) 30
3.3 Security 30
3.3.1 Hashing 31
3.3.2 Signing Algorithms 32
3.3.2.1 HMAC 32
3.3.2.2 ECDSA 33
3.3.2.3 EdDSA 35
3.3.2.4 RSA 36
3.4 JWTs in Vapor 38

4 JWTKit v5 **41**
4.1 Package Structure and Public API 41
4.1.1 Version 4 41
4.1.2 Version 5 42
4.2 BoringSSL vs SwiftCrypto 43
4.2.1 BoringSSL 44
4.2.2 SwiftCrypto 44
4.2.2.1 CryptoExtras 44
4.2.3 Key Creation 45
4.2.3.1 Signing 49
4.2.3.2 Verification 50
4.2.4 C vs Swift 52
4.2.5 RSA "Raw" API 52
4.3 PSS Addition 54
4.4 Modern Concurrency Adoption 55
4.5 Customisation 56
4.5.1 Custom Header Fields 57
4.5.2 Serialising and Parsing 59
4.6 X5C Verification 61

5 Results **63**
5.1 Signing 64
5.2 Verification 66
5.3 Token Lifecycle 68
5.4 Lines of Code 71

5.5 End result 72

Conclusion **73**

A Appendixes **75**
A.1 RSA Prime Factors Calculation 75

Introduction

When Apple made their Swift language open source in 2015 giving it cross-platform support, Swift developers started putting their efforts into shifting some of Swift's focus to the server side. Given Swift's strict and powerful type-system, efficient and safe memory management and modern concurrency model, it is not only a great tool for making client side apps, but also a perfect fit for the server. That's why Tanner Nelson created Vapor, currently the most popular server side swift framework. As with every respectable server-side framework, Vapor too needed a safe and efficient JWT library. JSON Web Tokens, or JWTs, are the go-to technology for enabling secure communication between parties. This is where JWTKit comes in. Since its creation, JWTKit has been a crucial library for Swift on the server developers and has been long due for a re-evaluation.

JWTKit was initially constructed as a wrapper for Google's BoringSSL, a famous C library based on top of OpenSSL. This approach worked well for the first few versions up until version 4. Maintaining a Swift library dependent on C is not modern, safe nor efficient, this is why the decision was made to eradicate BoringSSL from JWTKit. With Swift being run on Linux more and more, Apple published an open source version of CryptoKit, SwiftCrypto, which provides cryptographic operations to Apple devices. Being maintained by Apple, available on Linux and having a C free API makes SwiftCrypto the best companion for JWTKit.

Additionally, with Swift's new concurrency system, maintaining a library which was not thread-safe by default didn't make sense, that's why during JWTKit's v5 redesign another one of the upgrades the package got was integration with the Sendable protocol and Swift 5.5's new concurrency model, which guarantees thread-safety and eliminates data races from the library's types. Then, the missing RSA-PSS algorithm was added for signing and verifying tokens, in addition to the already present RSASSA-PKCS1-v1_5. X5C header verification, which was previously based on BoringSSL too, was switched over to use Apple's swift-certificates package, which allows working with X.509 Certificates. Finally, the possibility was added for users to implement their own parsing and serialising of the tokens, allowing for custom header fields like the zip header, which modifies the structure of the token by compressing the payload.

All of these changes and more are explained in detail in this thesis after an introduction to Swift and the server side Swift ecosystem, the JSON Web Token technology, and how these two merge. Finally the thesis will explore how the updates impact performance relatively to the previous version.

1. An introduction to Swift

In 2014, Apple made a groundbreaking announcement at their Worldwide Developer Conference (WWDC), introducing Swift, a new powerful and modern language initially designed to replace Objective-C on iOS, macOS, watchOS and tvOS. Swift is a general purpose, statically typed and compiled programming language with modern, user friendly yet expressive and powerful syntax. Following are some of Swift's syntax's peculiarities, listed to introduce the reader to the language and to prepare them for other code examples later on, when these concepts are going to be taken for granted.

1.1 Type System

Swift is a statically typed language, which means that each variable and constant has an explicit type, whether declared by the programmer or inferred by the compiler. This strict typing helps prevent runtime errors such as type mismatches. Swift's type system also supports advanced features like generics which enable the creation of flexible and reusable functions and types while maintaining strong type safety. Additionally, Swift's protocols enable the creation of blueprints which any type can adopt, ensuring that the type conforms to the specified behaviours.

1.1.1 Type Safety

"Swift is a type-safe language."[1] A language is type safe when the operations performed on data are only allowed if they are consistent with the data's type. In other words, a type safe language does not allow for type errors at runtime, as every type error is caught at compile time. While type checks are great to catch mismatched types, type safety can also lead to more verbose code as every variable and object have to have an explicitly defined type. In Swift this is solved by type inference, a technique compilers use to deduce the type of an expression by analysing the values provided to it.

```
let publicExponent = 65537
```

In this example the compiler can infer the `Int` type of the constant without it having to be explicitly defined. Swift also allows for some extra formatting specifically for number literals which allow for better readability, for example different base integers, which can be represented like:

```
let dec42 = 42
let bin42 = 0b101010 // binary representation of 42
let hex42 = 0x2A // hexadecimal representation of 42
```

or big numbers which can contain underscores:

```
let currentEpochTime = 1_707_928_991 // time in seconds from 1. Jan 1970
```

[1] To handle null values, Swift uses Optionals.

1.1.2 Generics

Generics are the implementation of the so-called "parametric polymorphism". This programming technique allows developers to write reusable code which can work with any type subject to user-defined requirements. Generics solve the problem of having multiple implementations for a method which only differ by the type of the data they work on. Thanks to Swift's strict type system, the same code can be reused for different types:

```
struct IntQueue {
    private var elements: [Int] = []

    mutating func enqueue(_ element: Int) {
        elements.append(element)
    }

    mutating func dequeue() -> Int? {
        guard !elements.isEmpty else { return nil }
        return items.removeFirst()
    }
}
```

would have to be rewritten for each type which is not an `Int` (`Double`, `Float` etc.). With generics the code can be rewritten to:

```
struct Queue<Element> {
    private var elements: [Element] = []

    mutating func enqueue(_ element: Element) {
        elements.append(element)
    }

    mutating func dequeue() -> Element? {
        guard !elements.isEmpty else { return nil }
```

```
            return elements.removeFirst()
        }
    }
```

which enables the stack to be used with any type, e.g. `Queue<String>` or `Queue<Double>`.

1.1.3 Protocols

A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to conform to that protocol. [1] A protocol can be defined as:

```
protocol MyProtocol {}
```

A class can then extend said protocol as follows:

```
class MyClass: MyProtocol {}
```

Protocols can define both methods and properties which the conforming class has then to implement:

```
protocol Countable {
    var count: Int { get set }
}

struct Queue<Element>: Countable {
    private var elements: [Int]

    var count: Int {
        elements.count
    }

    ...
}
```

The earlier created `Queue` class can even be extended using one of Swift's standard library protocols: `Numeric`, which all basic number types conform to:

```
struct NumericQueue<Element>: Countable where Element: Numeric {
    ...
}
```

This assures that the class can not be used with elements that are not numbers.

1.1.4 Type Erasure and Opaque Types

A common pattern to use the property defined in the `Countable` protocol might be:

```
func printCountOf<Collection>(collection: Collection)
    where Collection: Countable
{
    print("The count of your collection is \(collection.count)")
}
```

While this works, this method looks more complex than needed, that's why Swift introduced the `some` keyword to make this exact pattern easier to express:

```
func printCountOf(collection: some Countable) {
    print("The count of your collection is \(collection.count)")
}
```

Here `some Countable` indicates that the method expects a type which conforms to `Countable`, without having to specify the specific type. This is called an **opaque** type. Finally, it might be quite intuitive to write something like:

```
func printSumCountOfAll(collections: [some Countable]) {
    var count = 0
    for collection in collections {
        count += collection.count
    }
    print("The sum of all collections count's is \(count)")
}
```

This is not going to work, since the underlying type (which is the concrete type that gets passed to the method instead of `some Countable`) is fixed, which means that all of the elements of the array must be of the same type. Although, the need is to have an array of arbitrary `Countable` types, as the method doesn't care which type it is, just that it conforms to `Countable` This is where another one of Swift's keywords comes in handy: `any`. By using `[any Countable]` the array represents any arbitrary type of `Countable` and the underlying type can vary at runtime. This doesn't give the same compile time type-safety that `some` has, though it allows to represent different types of `Countable` at once, solving the problem at the top. This is called **type erasure**.

1.2 Memory Management

Memory management is a crucial aspect of programming languages which can lead to errors like memory leaks and buffer overflows if not handled correctly. Swift incorporates a number of mechanisms to avoid these types of errors.

1.2.1 Automatic Reference Counting (ARC)

Swift uses Automatic Reference Counting (ARC) to track and manage the app's memory usage. In most cases, this means that memory management "just works" in Swift, and developers don't need to think about memory management themselves. ARC automatically frees up the memory used by class instances when those instances are no longer needed.[2] ARC is a memory management feature which relies on reference counting. This system keeps track of all memory allocations that happen during runtime and automatically deallocates the memory of elements no longer needed, eliminating the programmer's burden to manually handle memory allocation and deallocation.

1.2.2 Copy-on-Write Semantics (CoW)

For some of its value types such as `Array`, `String` and `Dictionary` Swift implements Copy-on-Write semantics, a memory management technique which allows efficient management of mutable data structures. CoW is particularly useful in managing collections such as arrays, sets or strings. The goal is to minimise unnecessary copying of objects between different callers and, as the name suggests, this is done by copying the object in a new, private copy visible to the caller *only* when the object is written-on. If the passed object is never modified by the caller, no copy gets created and the caller gets to keep a reference to the original object without having to allocate new, useless, memory for data they already have. [3]

1.2.3 Optionals

In many languages such as C or C++ null references can lead to runtime errors such as null pointer dereferences, which can cause crashes or undefined behavior. The compiler prevents access to uninitialised or deallocated memory by implementing optionals. Optionals are designed to handle the absence of a value: formally, for a non optional type `T`, `T?` is the corresponding optional type representing either a value of type `T` or the absence of a value (`nil`). Following is an example of an optional:

```
var optionalString: String? = "Hello"
optionalString = nil // This is allowed and safe
```

Optionals are implemented as an enum with two cases: `none` and `some(Wrapped)`, which represents a wrapped value of type `T`. To use the wrapped value, the optional must be unwrapped first. This can be done in different ways:

- **Optional Binding**:

```
if let unwrappedValue = optionalValue {
    // Use unwrappedValue safely within this block
```

```
} else {
    // optionalValue is nil
}
```

This technique allows to conditionally unwrap an optional and bind its value to a new constant or variable within a conditional statement. Depending on if the optional contains a value or not, either the first block or the second will be executed.

- **Nil Coalescing**

```
let unwrappedValue = optionalValue ?? defaultValue
```

This method provides a default value to fallback to if the optional doesn't contain a value.

- **Guard Statement**

```
guard let unwrappedValue = optionalValue else {
    // Handle the case where optionalValue is nil
    return
}
// Use unwrappedValue safely beyond this point
```

This statement is similar to Optional Binding but instead of continuing in the `if` block, it exits early (in fact, inside of a `guard` statement, there must be either a `return` or a `throw` expression) if the value is `nil` and continues on outside otherwise.

- **Force Unwrapping**

```
let unwrappedValue = optionalValue!
```

This forcefully unwraps the optional bypassing Swift's safety checks and crashes the runtime if the optional is `nil`. Using force unwrapping is a bad practice and developers should choose other strategies unless they are sure the optional is never going to be empty.

1.3 Swift's Concurrency Model

Usually, when a method gets called, the thread on which the method is executed gets blocked until the end of the methods' execution. That is why, in modern languages and especially in server-side oriented ones, the ability to execute more than one task at a time is fundamental. This programming technique is called **concurrency**, and aims to increase responsiveness, scalability and resource utilisation of programs by splitting up the work

and executing more than one operation at a time. With the release of version 5.5, Swift introduced new patterns to enable developers to write safer, simpler and more efficient concurrent code.

1.3.1 Asynchronous Programming

While not *enabling* concurrency per se, one of the most notable new features is async/await. This pattern is quite common in other languages like JavaScript, Python and C# and allows writing asynchronous code using normal, synchronous-looking constructs. With async/await functions can be marked as `async`, which means that the method can suspend its execution awaiting the completion of another subroutine without blocking the thread it is running on. Inside of the method, the `await` keyword is used to execute the function that is going to pause current `async` function. For example, this completion-based code block:

```
func fetchData(completion: @escaping (Result<Data, Error>) -> Void) {
    let task = URLSession.shared.dataTask(with: url) { data, _, error in
        completion(data.map(Result.success) ?? .failure(error ??
        ↪  NetworkError.unknown))
    }
    task.resume()
}

func getData() {
    return fetchData { result in
        switch result {
        case .success(let data):
            processData(data) { processedResult in
                // Handle processedResult
            }
        case .failure(let error):
            // Handle error
        }
    }
}
```

can be rewritten with async/await as:

```
func fetchData() async throws -> Data {
    let (data, _) = try await URLSession.shared.data(for: url)
    return data
}
```

```
func getData() async throws {
    do {
        let data = try await fetchData()
        let processedData = try await processData(data)
        // Handle processedData
    } catch {
        // Handle error
    }
}
```

Using completions works, though it can get messy quite fast. Using async/await avoids the use of the so called "callback-hell", when a pyramid of callbacks is created by nesting callbacks in each call, making the code much clearer and easier to debug. Another advantage is getting rid of the `Result<Data, Error>` type and being able to throw errors normally like in any other synchronous code block. [4]

1.3.2 Structured Concurrency

While async/await introduced a simpler way to write asynchronous code, the `getData()` method is not yet *concurrent*: it is asynchronous, though it is still sequential, meaning that `fetchData()` and `processData()` are still going to get executed one after the other. To be concurrent, the tasks need to be able to be executed at the same time: this is the reason Swift introduced Structured Concurrency.
Put simply, running code concurrently means splitting up the workload of a function on a number of threads rather than using just one, and executing those threads at the same time, gaining a significant time advantage: this is called "unstructured concurrency". Unstructured concurrency *works*, the problem is that various issues can arise from threads being spawned independently, such as race conditions and deadlocks. With the "structured" adjective, concurrency becomes not only about randomly splitting up work, but doing so in a structural, hierarchical manner. In particular, the work is split up into **tasks**. *A task is the basic unit of concurrency in the system*[5], and each time a task is created, it becomes a child of the task where it was spawned. This way, tasks inherit their parents' lifecycle and are not left dangling. This also means that when a parent task gets cancelled, the cancellation gets propagated to its children. Errors on the other hand get propagated up to their parent task, centralising and simplifying error handling. Furthermore, since tasks are managed hierarchically, data sharing between tasks gets much easier. [5] Using the same example as earlier but supposing that multiple calls have to be made concurrently to different URLs, in Swift this can be done as:

```
func getData(from urls: [URL]) async throws {
    // Create a task group for concurrent data fetching
```

```
    let fetchedData = try await withThrowingTaskGroup(of: Data.self) {
    ↪  group -> [Data] in
        var results = [Data]()

        // Add a fetching task for each URL
        for url in urls {
            group.addTask {
                try await fetchData(from: url)
            }
        }

        // Collect results
        for try await data in group {
            results.append(data)
        }

        return results
    }

    // Process all fetched data
    for data in fetchedData {
        let processedData = try await processData(data)
        // Handle processedData
    }
}
```

In this example, a task group is created, and, for each URL, a task is added to the task group to fetch data from that URL. Then, in the for loop, each task is executed and the result of each API call is added to the `results` array, finally to be processed by the `processData(_:)` function.

1.3.3 Actors

While Structured Concurrency is great for ensuring safety against data races between concurrently executing code, that paradigm doesn't allow for safely sharing mutable state, using, for example, a class. Classes are reference types which means they *can* be used to share state between different tasks, the problem is that they require code-explicit-synchronisation to be used safely across concurrent domains, and even then safety is not guaranteed. This is why Swift 5.5 also introduced actors. Actors, just like classes, are reference types, though they differ in how tasks can access their data: access to an actor's properties is **asynchronous**. This means that actors only allow one task to modify their state at a time (in

fact, any operation performed on an actor is done via the async/await syntax), while other tasks have to wait for the task before to finish. This ensures (at compile time) that there can not be any race conditions on the actor's state. Actors also encapsulate their state to be private, only allowing operations they decide (for example, via getters and setters), denying direct access to external parts of the program. [1] The following example demonstrates how an actor dramatically simplifies a class that was made data-race safe using a `DispatchQueue`:

```
class Counter {
    private var value = 0
    private let queue = DispatchQueue(label: "com.example.counterQueue")

    func increment(completion: @escaping () -> Void) {
        queue.async {
            self.value += 1
            completion()
        }
    }

    func getValue(completion: @escaping (Int) -> Void) {
        queue.async {
            completion(self.value)
        }
    }
}

let counter = Counter()

counter.increment {
    counter.getValue { currentValue in
        print("Current Value: \(currentValue)")
    }
}
```

Using actors, this gets to become:

```
actor Counter {
    private var value = 0

    func increment() {
        value += 1
    }
```

```
    func getValue() -> Int {
        return value
    }
}

let counter = Counter()

await counter.increment()
let currentValue = await counter.getValue()
print("Current Value: \(currentValue)")
```

1.3.4 Sendable Types

The last concurrency related idiom introduced by Swift 5.5 discussed in this thesis is `Sendable`. `Sendable` is a marker protocol that indicates that the conforming type is safe to be sent across actor boundaries or to be used in concurrency. While actors and tasks create safe concurrency domains, a mechanism is required to safely pass data *across* said domains: by marking a type as `Sendable`, the compiler automatically knows that that type should be safe, and is going to warn (error in future Swift releases) if it is not. Additionally, thanks to structured concurrency and actors, the compiler is going to notice and warn when non-`Sendable` types are passed across different concurrency-domains. This helps catch race conditions at compile time. As of Swift 5.5, some types are even `Sendable` by default, such as structures (which are copy-value types) with only `Sendable` properties and enumerations with only `Sendable` associated values. [1]

```
actor SomeActor {
    func doThing(string: SomeNonSendableType) async {...}
}

func useActor(someActor: SomeActor, myString: SomeNonSendableType) async {
    await someActor.doThing(string: myString)
}
```

Actors are always implicitly `Sendable` since they handle their own mutable state, that's why the first method can be declared. The problem here is with the second method, as that is going to throw a warning:

```
Passing argument of non-sendable type 'SomeNonSendableType' into
   actor-isolated context may introduce data races
```

This example shows how the compiler behaves when a non-`Sendable` type is used concurrently. While it's just a warning, this code could introduce data races, that's why the warning will become an error in future Swift releases. [6]

2. Server Side Swift

With features such as type safety, automatic memory management and compile time safety for concurrency, Swift has become the perfect tool for writing server-side applications. The evolution of Swift for server-side development is significantly marked by the introduction of SwiftNIO, a non-blocking I/O framework that enables the development of high-performance and scalable network applications.

2.1 SwiftNIO

SwiftNIO is a cross-platform asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers and clients. It is like Netty, but written for Swift.[7] Published by Apple, this low-level networking package aims to be the foundation of Swift based web frameworks such as Vapor. It is equipped with a non blocking architecture which allows execution of operations without having to block the main thread, resulting in ease of scaling and handling of high volumes of traffic.

2.1.1 Non-blocking I/O

In a traditional, blocking, Input-Output (I/O) model, the thread to which a job is assigned (such as a disk read or a network request) gets blocked until the operation completes. This is a straightforward approach which can unfortunately lead to a number of issues:

- **inefficient resource utilization**: spawning a thread for each operation, especially in network servers which handle many simultaneous requests, can consume more system resources than needed;
- **poor scalability**: the system might hit its thread number limit long before reaching CPU or network capacity;
- **reduced responsiveness**: applications based on blocking I/O can become unresponsive, which is especially bad in Graphical User Interface (GUI) based applications.

These issues are solved by using non-blocking I/O. In an input-output environment it is better to let a single thread handle multiple connections, hopping from one to another when needed. This means that the program can perform other tasks while waiting for one to finish, which happens often in networking applications as data might not be available

right away. For example, reading from a socket might take time and until the whole of the data has been read, it can not be used. The system has two ways of handling this: while doing some other work it might decide to check back every once in a while if the data is available (polling) or it might create a notifier to be alerted when the data is ready (event-driven) and go back and use it once it is. [8] This last approach, the event-driven one, is what is used in SwiftNIO. [9]

2.1.2 Event Driven Architecture

If a client is making an HTTP POST request with a large body over a slow network, the server might be able to read the data faster than the client can post it. Rather than waiting for the operation to finish, it makes sense for the system to notify the program when the operation is completed so that the program can dedicate time and resources to other processes and come back to the initial one when the data is ready. This is better than polling, as that would require a structure such as a `while` loop that periodically checks whether the data is ready. [8] SwiftNIO implements this architecture using `EventLoop`s.

2.1.3 EventLoops

`EventLoop`s are one of the foundational blocks of SwiftNIO. An `EventLoop` is a programming construct that works like a `while` loop, waiting for events to come in and dispatching them to the appropriate event handler. [10] `EventLoop`s monitor socket descriptors for readability or writability, using system mechanisms like `epoll` on Linux, `kqueue` on macOS and `IOCP` on Windows. When an I/O operation is ready (for example, data has arrived on a socket), the `EventLoop` triggers the execution of callback functions or channel handlers associated with that operation. `EventLoop`s run on a single dedicated thread managing multiple connections and ensuring thread safety. In SwiftNIO, `EventLoop`s are grouped into `EventLoopGroup`s, which are used to spread work amongst different `EventLoop`s, and thus threads. These event loops work independently, each managing its own set of tasks. [9]

2.1.4 Channels and ChannelHandlers

While `EventLoop`s are the fundamental entity of SwiftNIO, the main structure users have to interact with are `Channel`s and `ChannelHandler`s. A `Channel` represents a single network connection on a socket, acting as the pipeline through which input and output data flows. They are responsible for handling all operations on a connection, such as opening it, closing it, reading from and writing to the network. Although the `Channel` reads and writes from and to the connection, it does not process any data, as that is done by the `ChannelPipeline`. A `ChannelPipeline` is an ordered sequence of `ChannelHandler`s, these are simple, reusable constructs which process the data in some way before passing

it along to the next `ChannelHandler`. When an event is received, it travels through the `ChannelPipeline`, with every `ChannelHandler` doing its processing on the data. Once the data is processed it is sent to the *network* if the handler is and Outbound handler, and to the *system* if the handler is an Inbound handler. The former ones are used for operations initiated from the inside and that have to go outside, such as network writes, the latter ones are used for the opposite, such as reading data from a socket. [9]

2.1.5 EventLoopFutures

The final SwiftNIO-related argument discussed in the thesis are `EventLoopFutures`. The `EventLoopFuture` is a generic type that encapsulates a future. *A future is an instance of a class that will have a value in some point in the future.*[10] It is used to encapsulate a value that is not known yet but will be defined in the future by the `EventLoop`, and it is particularly useful when writing asynchronous code, as often network operations can take some time and their result can not be known right away. [9] All asynchronous operations in SwiftNIO return an `EventLoopFuture`, that encapsulates the eventual result of that operation. The future is created when the operations starts and fulfilled when it ends, either containing the result when it was successful, or an error if it was not. Either way, a callback can be attached to execute when the operation ends, allowing to handle the result. For example:

```
let futureResult: EventLoopFuture<String> = ...

futureResult.whenComplete { result in
    switch result {
    case .success(let value):
        print("Operation succeeded with \(value)")
    case .failure(let error):
        print("Operation failed with \(error)")
    }
}
```

This snippet creates a future with a `String` which is tied to some asynchronous operation, such as a database or a network read. Once the operation is completed, it calls the `whenComplete` method, passing in a closure to handle the result.

While using SwiftNIO can be interesting as it is really powerful, it also has some justified complexity, as users should not be writing low level networking code directly. That's why server-side frameworks exist: to abstract away the complexities of libraries such as SwiftNIO. All of the past topics about SwiftNIO are the building blocks for modern Swift-based server side frameworks, in fact this exploration sets the stage for the next section of the thesis: Vapor.

2.2 Vapor

Vapor, built by Tanner Nelson in 2016 and now maintained by its core team and hundreds of contributors, is a modular Swift-based framework for writing server-side applications. It is and always has been an open source package, meaning that everyone can contribute to it, and since version 3 it is based on SwiftNIO, Apple's own open source networking I/O library. This part of the thesis will go about explaining the key features of Vapor and the reasons why it is as popular as it is.

2.2.1 Application and entrypoint

In Vapor, everything revolves around the `Application` type. This type represents the actual server being run. The application is usually started using an `entrypoint.swift` file which in the template looks like this:

```
@main
enum Entrypoint {
    static func main() async throws {
        // detect environment
        var env = try Environment.detect()
        // initialise logging system using environment
        try LoggingSystem.bootstrap(from: &env)

        // create application
        let app = Application(env)
        defer { app.shutdown() }

        do {
            // call the configure method on the application
            try await configure(app)
        } catch {
            app.logger.report(error: error)
            throw error
        }
        // run the application
        try await app.execute()
    }
}
```

This is the basic starting block of a Vapor server. Inside the `configure(:)` method the app can be set up instantiating various services such as controllers, database connections and more.[11]

2.2.2 Routing

The likely more apparent feature in a backend framework is routing, which is defining how the application responds to an incoming HTTP request. For example, to create an endpoint for `GET` requests on the ”hello” route in Vapor it is as simple as:

```
app.get("hello") { req in
    "Hello, world!"
}
```

Now when a `GET` requests comes in at the ”hello” endpoint, the server will return ”Hello, world”. With route parameters this can be simply expanded as

```
app.get("hello", ":name") { req in
    let name = req.parameters.get("name")!
    return "Hello, \(name)!"
}
```

At the `GET` ”hello/foo” route, the server will return ”Hello, foo!” The last example explains how Vapor uses its `Content` API to read a number of query parameters:

```
struct SomeContent: Content {
    // query parameters should always be optional
    let name: String?
    let age: Int?
}

app.get("hello") { req in
    let decoded = try req.query.decode(SomeContent.self)
    return "Hello, \(decoded.name ?? "No name")! You are \(decoded.age ??
    ↪  99) years old."
}
```

At the ”hello” `GET` route with the following query:

```
http://localhost:8080/hello?name=foo&age=42
```

the server will output ”Hello, foo! You are 42 years old.” [11]

2.2.3 Fluent

Any respectable server-side library needs access to persistent storage, usually SQL or NoSQL databases. In Vapor this role is played by the Fluent package, a database-agnostic library which provides APIs to connect to various databases. More precisely, Fluent is

an ORM or object relational mapping tool, which is a library that creates a relationship between the instances of a database to objects in code. Using an ORM can make the developer's life much easier as it can abstract away the complexity of having to use the database directly by allowing to write database code using a high level language rather than SQL queries directly. Plus, Swift's strict type safety ensures that only valid queries are constructed.

2.2.3.1 Database Drivers and Setup

Thanks to the different drivers provided by Vapor, Fluent can connect to different databases:

- PostgreSQL
- SQLite
- MySQL

Fluent does not care about which database is being used in a certain query as it will be the underlying driver which will translate the Swift code into the respective data manipulation language. Each database conforms to the `Database` protocol which allows for interchangeability between different underlying databases. For a database to be used it's enough for it to be registered in the `configure(app:)` method, for example, using a Postgres database:

```
import Fluent
import FluentPostgresDriver

try app.databases.use(.postgres(url: "<connection string>"), as: .psql)
```

[11]

2.2.3.2 Model

Each table in the databases that needs to be mapped to an object can be represented in code as a `Model`. Models can have one ore more properties which represent the columns in the table, one of which is the identifier required by a `Model`. For example:

```
final class User: Model {
    static let schema = "users"

    @ID(key: .id)
    var id: UUID?

    @Field(key: "name")
    var name: String
}
```

In this example the first two fields are required while the third one is optional. The `schema` indicates the name of the table in the database while the `id` is the identifier of the model. The `@ID` and `@Field` annotations are property wrappers added to the fields which indicate how they are stored in the database. [11]

2.2.3.3 Migrations

The migrations API allows implementing structural changes to the database using DDL (Data Definition Language). Migrations are used for all those operations that *change* the structure of the database rather than working on the data itself. For example they can be used to create tables, delete them or modify their structure such as adding or removing columns. Fluent automatically keeps track of all migrations run so there's no need to access the database directly for such operations, allowing for reversion of already run migrations, similar to how a version control system works. All migrations are atomic, meaning that if an error is encountered during a migration, the system will be reverted to the state before the migration, keeping the database intact and safe. Migrations can be defined as follows:

```
struct CreateUserMigration: AsyncMigration {
    func prepare(on database: Database) async throws {
        try await database.schema(User.schema)
            .id()
            .field("name", .string, .required)
            .create()
    }

    func revert(on database: Database) async throws {
        try await database.schema(User.schema).delete()
    }
}
```

The `prepare` method does the actual changes to the database, while the `revert` method is called when the changed needs to be, in fact, reverted. In the example above the `User` table is created with a required "name" column of type string. The `revert` simply deletes the user table, which is the reversion of what the `prepare` method does. Migrations can easily be added to the `configure(app:)` method:

```
app.migrations.add(CreateUserMigration())
```

This adds the `CreateUserMigration` to the list of migrations of the app. Migrations can then be either executed using the `migrate` command (with the `--revert` flag if it's to revert the migration):

```
swift run App migrate [--revert]
```

or automatically using:

```
try await app.autoMigrate()
```

By adding this command to the `configure(app:)` method, migrations are going to be run (more specifically, the `prepare` method) the next time the app is spun up. Since Fluent keeps track of all executed migrations, they are going to be executed only once unless they get reverted. [11]

2.2.3.4 Query

Fluent also provides an API to use DML (Data Manipulation Language) on the database. It supports most of SQL's (and NoSQL's, but being database agnostic, the high level API is the same) constructs to create, read, update and delete models. Using the same `User` model as before, all users called "Foo", ordered alphabetically by name and with their `Profile`s can be retrieved using:

```
let users = try await User.query(on: database)
    .join(Profile.self, on:  \User.$id ==  \Profile.$userID)
    .filter(\.$name == "Foo")
    .sort\.$name)
    .all()
```

The `User.query` expression returns a `QueryBuilder` for the given model, which allows the creation of a type safe query, providing the user compile time information of whether the query will yield an error or not when executed on the database. It will be translated to something like:

```
SELECT users.* FROM users
JOIN profiles ON users.id = profiles.user_id
WHERE users.name = 'Foo'
ORDER BY users.name ASC;
```

[11]

By abstracting direct database access and leveraging Swift's strong typing, Fluent makes persistent storage management using Vapor both efficient and enjoyable. While it introduces an additional layer of abstraction, the benefits in terms of development speed, code quality, and maintenance are significant. Being a standalone package, Fluent (or rather FluentKit, as Fluent is the integration package which bridges Fluent to Vapor) can even be used without Vapor.

2.2.4 Authentication and Authorisation

Authentication and authorisation are both key concepts used to protect systems and information. While authentication is the act of identifying a user, authorisation means granting them access to determined resources or actions around them. [12] Some resources need to be protected and not accessible at all times and that's where authentication and authorisation come in. Vapor has built-in support for both.

2.2.4.1 Authentication

Usually authentication revolves around validating user credentials, such as email and password, against stored data. To implement authentication a data model is needed which conforms to `Authenticatable`:

```
final class User: Model, Content {
    static let schema = "users"

    @ID(key: .id)
    var id: UUID?

    @Field(key: "email")
    var email: String

    @Field(key: "passwordHash")
    var passwordHash: String
}

extension User: Authenticatable {}
```

This snippet takes the `User` model created earlier, adds email and hashed password fields and conforms it to the `Authenticatable` protocol. This protocol defines that a model can be authenticated and can be integrated with Vapor's various authentication providers and middleware, allowing for flexible and secure authentication mechanisms. Afterwards the mode of authentication needs to be specified, for example using password authentication:

```
extension User: ModelAuthenticatable {
    static let usernameKey = \User.$email
    static let passwordHashKey = \User.$passwordHash

    func verify(password: String) throws -> Bool {
        try Bcrypt.verify(password, created: self.passwordHash)
    }
}
```

This conforms the model to `ModelAuthenticatable`, which requires the definition of the two fields that will be verified during authentication and a method defining how the password should be verified, in this case using `Bcrypt`. Finally, the user can be authenticated during login:

```
let passwordProtected = app.grouped(User.authenticator())
passwordProtected.post("login") { req -> User in
    try req.auth.require(User.self)
}
```

When POSTing the "login" route, if the credentials are correct the user will be logged in and their data will be returned, otherwise an unauthorised error will be thrown. [11]

2.2.4.2 Authorisation

Authorisation is the act of giving a user the privileges they deserve. This can be done, for example, using role-based access. This pattern involves giving each user a role and authorising certain operations based on that role. A role field could be added to the previous model:

```
enum UserRole: String {
    case admin
    case user
}

final class User: Model, Content {
    ...

    @Field(key: "role")
    var role: UserRole
}
```

And then adding some `Middleware` which checks that the user is an admin:

```
struct EnsureAdminUserMiddleware: AsyncMiddleware {
    func respond(
        to request: Request,
        chainingTo next: AsyncResponder
    ) async throws -> Response {
        guard
            let user = request.auth.get(User.self),
            user.role == .admin
        else {
```

```
            throw Abort(.unauthorized)
        }
        return try await next.respond(to: request)
    }
}
```

This middleware ensures that the user accessing a certain resource is an admin and throws an unauthorised error otherwise. It can be registered to a certain route or route group using:

```
let protectedRoutes = app.grouped(EnsureAdminUserMiddleware())
protectedRoutes.get("admin-only", use: adminOnlyHandler)
```

This snippet will protect the GET route on the ”admin-only” endpoint from being accessed by non-admin users. [11]

Having explored the foundational elements of authentication and authorisation within Vapor, the next chapter in this thesis will delve into another, more language agnostic aspect of modern web security: JSON Web Tokens (JWTs).

3. JSON Web Tokens

In the context of web development, ensuring secure communication and efficient authentication mechanisms is important. There are many technologies that allow that, though the JSON Web Token (JWT) specification has become one of the standard ways of communicating, likely due to its simplicity and versatility. *JWT is a compact, URL-safe means of representing claims to be transferred between two parties.* [13] JWTs are based on JSON, which is a format used to serialise structured data.[14] It is widely used in a number of scenarios such as configuration files, storage (some NoSQL databases use it to store data) and, finally, data interchange. JWTs allow transferring of JSON data between parties in a safe way as the tokens are digitally signed using a signing algorithm.

Base64 and Base64URL Encoding

Before delving into the specifics of a JWT's structure, it is essential to understand the encoding techniques that make JWTs both compact and web-safe: base64 and base64URL encoding. These encoding schemes allow JSON data to be easily transformed into a format that can be safely transmitted over web protocols.

Base64 Encoding Base64 is a binary-to-text encoding scheme that represents binary data in an ASCII string format by translating it into a radix-64 representation. It is widely used in various applications, including email via MIME, as well as storing complex data in XML or JSON formats. The process involves dividing the input binary data into sequences of 24 bits (3 bytes). Each 24-bit sequence is then split into four 6-bit groups, with each group mapped to a single character in the base64 alphabet, which consists of 64 characters: A-Z, a-z, 0-9, +, and /. This method allows binary data to be represented in a text format, which is particularly useful for data transmission over text-based protocols. [15]

Base64URL Encoding While base64 encoding is versatile, it includes characters (+ and /) that can have special meanings in URLs and file systems. Base64URL solves this issue by replacing + with -, and / with _, making the encoded data safe for URL and filename usage. Additionally, base64URL encoding omits padding characters (=). This URL-safe variant is important for JWTs, as it ensures the token's components can be safely transmitted in URLs. This is fundamental as JWTs are often used to authenticate users in web applications and in such an environment data is transmitted using URLs. [15]

3.1 Structure of a JWT

A JWT typically consists of three base64URL encoded strings divided by a dot (".") character:

- Header: a number of JSON fields which reveal some information about the token itself, for example what algorithms was used to sign it and the type of the token (JWT, JWE etc). The only mandatory field is the `alg` one, but more optional fields can be added too. This is how a JWT header typically looks:

  ```
  {
    "alg": "HS256",
    "typ": "JWT"
  }
  ```

  This header contains two fields: the `alg` field indicating that the token was signed using the HMAC algorithm with SHA-256, and the `typ` field, indicating that that is a JWT. This JSON is then base64URL encoded and will look like this:

  ```
  eyJhbGciOiAiSFMyNTYiLCJ0eXAiOiAiSldUIn0
  ```

  > **Notable fields**
  > Some notable fields which will be used later on are:
  > - `kid`: the `kid` or key identifier field is used to identify the key which the token was signed with and needs to be verified with out of a key set;
  > - `x5c`: a DER formatted X.509 certificate chain used to certify the authenticity of the token, with each certificate certifying the next one up to the root Certificate Authority (CA);

  [16]

- Payload: contains the actual data that needs to be transmitted in the form of JSON claims. There are some default claims, such as the token's expiry and issuer and there can be custom claims. This is how a payload typically looks:

  ```
  {
    "sub": "1234567890",
    "name": "John Doe",
    "iat": 1516239022
  }
  ```

  In this example there is three claims: the first one is the `sub` which indicates the subject or whom the token refers to. The third one is the `iat` claim, or "issued at",

which indicates when the token was signed using Unix time. The second one, `name`, is a custom claim indicating some name. When base64URL encoded, this payload looks like:

```
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4iLCJpYXQiOjE1MTYyMzkwMjJ9
```

- Signature: string used to secure and verify that the token has not been altered. When using a private key to sign the token, the signature also asserts that the issuer has the private key and can therefore be trusted. The signature is created by concatenating the base64URL encoded header and payload using a dot (”.”). Afterwards a cryptographic algorithm is applied to that string resulting in the signature, which is appended to the initial token using another dot (”.”) character. Using HMAC-256 and the *very* secure secret key ”secret”, the final token looks like this (newlines are inserted for display purposes):

  ```
  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
  .
  eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4iLCJpYXQiOjE1MTYyMzkwMjJ9
  .
  4hrUcwXkLeE3O-pwPHgS860Nt-Lq_3PNoyHXnBTyjIY
  ```

  In this example the token parts are on different lines for readability but one can exactly see the header, payload and signature, all split by a dot (”.”). This is known as the compact serialisation form of the JWT.

[13] [17]

The main use case of JWTs is authentication, more specifically stateless authentication.

3.2 Stateless Authentication

JWTs can be, and are most commonly used for authentication. Using JWT authentication, servers can create a JWT when the user logs in and return that token to the user. The user can then store the token on the client side (some complications may happen which will be discussed later on) and attach the JWT as a header field in every request they send; this allows the server to verify the token and therefore approve or deny (in case validation fails) access to the user. Since JWTs contain actual data, the payload can store the role of the user and the server can therefore check for authorisation privileges too.

This mechanism is advantageous because it allows for stateless authentication, this means that the server does not have to store the token or any other login data in a persistent storage, since the token is stored on the client side, they just have to validate it on each request.

Although stateless authentication has its pros it also has its cons. First of all data on the client side is hard to keep safe, meaning that a number of different approaches exist to tamper with client side-saved tokens. [17]

3.2.1 Cross-Site Request Forgery (CSRF)

Cross Site Request Forgery essentially means that if a user is logged in into a website (which usually means they have some authentication cookies stored in the browser) and visits a malicious site, then the malicious site can use the user's cookies or local storage data to forge a request and make requests to the original website, authenticating as that user. The solution to this attack is splitting the token parts by saving the signature in the cookies and the header and payload in the client's local storage. [17]

3.2.2 Cross Site Scripting (XSS)

Cross Site Scripting attacks consist of injecting malicious JavaScript into trusted sites. An example might be a website that takes unsanitised user input and stores it in storage, then later on renders it as HTML and therefore potentially renders a malicious `<script>` tag, which could access tokens stored in cookies and in local storage. To mitigate this attack developers should always sanitise inputs and use the `HTTPOnly` tag on the cookie containing the signature. This makes the cookie inaccessible to JavaScript. To mitigate both issues the use of HTTPS is strongly recommended. [17]

Both authentication and authorisation processes are closely tied to the safety of a web application, and JWTs help promote this safety. The next section will dive into why and how JWTs help improve the security of a web application.

3.3 Security

This next section will delve into how JWTs provide security using digital signatures and how signatures are created. A digital signature is a piece of information that can be generated by encrypting a digest of the message with a private key. Afterwards, using the signature, the recipient can check the authenticity of the message with the public key, verifying that the message has not been altered since it was signed and that it was signed by the holder of the private key. This technology provides several key benefits:

- **Authenticity**: digital signatures verify the source of the message, ensuring that it comes from the claimed sender;
- **Integrity**: any alteration to the message after signing invalidates the signature, protecting against tampering;

- **Non-repudiation**: the signer cannot deny the authenticity of the message they signed, as only their private key could have been used to generate the signature.

For JWTs, digital signatures are defined by the JSON Web Signature (JWS) specification (RFC 7515) [16], which specifies how to digitally sign or create message authentication codes (MACs) for JSON based data. Following is a list of tools necessary to understand how signing works.

3.3.1 Hashing

All signing algorithms depend on a hash that is performed on the data to be signed before it is encrypted. A hash is a function that converts a message into a fixed-size string of bytes. Rather than signing the data directly which could be of any size, the data gets hashed before, resulting in a string of fixed length that allows the signing algorithm to have a more uniform distribution of efficiency for any data being signed. Often data can be large which means that hashing it also reduces the complexity for the signing algorithm. The hashing process is also collision resistant, which means that it is computationally infeasible to find two different inputs that produce the same output. Therefore the hashed data is uniquely tied to the data it represents, and any minor change to the data would result in a change of hash, invalidating the signature. Furthermore, this property also ensures confidentiality, as the original data can not be retrieved from the hash in a reasonable amount of time. [18]

Different hashing algorithms exists, although the ones used in signing algorithms are all variations of the SHA (Secure Hash Algorithm) family: SHA256, SHA384 and SHA512. Following is an overview of how these work:

1. **preparation**: all algorithms start with padding the message so that its length becomes of a certain size, afterwards the message is divided into blocks of a certain length;

2. **initialisation**: then, a set of initial hash values is defined $(H_0, H_1, ..., H_n)$ based on square roots of prime numbers. These initial values are the same for SHA256 and SHA512 but slightly different for SHA384;

3. **computation**: each of the padded blocks is processed in multiple rounds in which bitwise operations such as AND, OR, NOT and XOR and bit shifting are applied, also mixing in constants derived from the cube roots of primes. The output of each block becomes the input for the next round;

4. **end**: once computation is done, the blocks get concatenated and the hash value is ready.

The algorithms work in basically the same way, although the results differ in size, meaning that a hash created using SHA256 will be 256 bits long, one created with SHA384 will be 384 bits and SHA512 will result in 512 bits. SHA384 is a truncated version of SHA512 and also differs from the other two in how initial values for the blocks are chosen, though the algorithm remains the same. [19]

3.3.2 Signing Algorithms

Two types of algorithms can be used to sign JWTs:

- **Symmetric**: the same secret is used by both parties, to both sign and verify the signature;
- **Asymmetric**: a pair of keys is used, one private and one public, the former for signing and the latter for verifying.

The latter should be preferred when the token issuer is not the same entity as the token consumer.

Following is a list of most, though not all, of the signing algorithms defined by the JSON Web Algorithms (JWA) (RFC 7518) [20] specification and currently in use by JWTs. This is to give an overview to the reader about what algorithms are used in JWTs and how signing and verifying works for each of them. Each algorithm has three different "versions" depending on which hashing algorithm (SHA256, SHA384 and SHA512) is used, though the signing algorithm itself is independent of the hashing algorithm, that is why it is abstracted away from the following explanations.

3.3.2.1 HMAC

HMAC or (Hash-based Message Authentication Code) is currently the only symmetric algorithm used to sign JWTs. The algorithm creates a Message Authentication Code (MAC) using a hash function and a secret key. A MAC is somewhat similar to a digital signature in the sense that it is a piece of information used to verify the authenticity a message. It is however similar in that does not use a public and private key but it relies on the two parties exchanging information to share the same secret key. The sender will generate a MAC, also known as tag, by combining the message with a secret key and then sends both the original message and the MAC tag to the receiver. The receiver then applies the algorithm again to the received message and obtains a new MAC, if that matches to the one they received, then the message is authentic, otherwise it has likely been tampered with. HMAC specifically works the following way:

1. **preparation**: if the secret key is too short then it gets padded. Then the it gets hashed with a hashing algorithm such as SHA256, SHA384 or SHA512;

2. **process**: two constants, `opad` (output padding) and `ipad` (inner padding) are used. These get XORed with the secret key, resulting in two different keys. The message is then processed twice using the hash function, the first time with the `ipad`-XORed key and the second time with the `opad` one. This results in the final HMAC value.

[21] Depending on the hashing algorithm used, the JWS specification defines three algorithms which use HMAC: HS256, HS384 and HS512. [16]

3.3.2.2 ECDSA

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a variant of the Digital Signature Algorithm (DSA) that uses elliptic curve cryptography to provide a secure way of generating and verifying signatures. An elliptic curve is a smooth non-singular curve defined over a field and following a mathematical equation. The equation has the following general form

$$y^2 = x^3 + ax + b$$

where a and b are coefficients which satisfy the $4a^3 + 27b^2 \neq 0$ equation to ensure the curve is non-singular. In cryptography, elliptic curves are defined over a finite field, allowing the creation of discrete logarithm problems that are currently computationally infeasible to solve. They can be defined in either one of these ways:

- prime curves over a prime field $\mathbb{F}_n$ where the equation takes the form

  $$y^2 \equiv x^3 + ax + b \; mod \; p$$

  where p is a prime;
- binary curves over a binary field F_{2^n}, with the equations modified to fit the structure of the field.

[22]

Key Generation
Following is the explanation on how keys for ECDSA are generated:

1. **selection of the curve**: the curve $\mathbb{E}$ (on a finite field) has to be selected along with its parameters: (p, a, b, G, n, h) where p is the size of the field, a and b are defined above, G is a base point on the curve, n indicates the order of G and h is the co-factor;
2. **selection of the private key**: the private key d is selected randomly in the range $[1, n-1]$;
3. **calculation of the public key**: the public key Q is calculated by simply multiplying d and G.

Finally the two points on the curve representing private and public key are ready for signing. [22] [23]

Signing
The process of creating a digital signature with ECDSA involves the following steps:

1. hash the original message m, getting $H(m)$;
2. choose a random nonce k such that $1 \leq k < n$;
3. compute $R = kG = (x, y)$. If $x = 0$ then a new k should be chosen;
4. compute $s = k^{-1}(H(m) + xd)\ mod\ n$. If $s = 0$ then a new k should be chosen.

The final signature for the document will be

$$(R, s).$$

This can then be attached to the document and transmitted together. [24]

Verification
Following is the verification process for ECDSA keys:

1. let x be the x coordinate of R;
2. verify that x and s are in $[1, n-1]$, otherwise the signature is invalid;
3. hash the original message m, getting $H(m)$;
4. compute
   - $u_1 = s^{-1}H(m)\ mod\ n$
   - $u_2 = s^{-1}x\ mod\ n$
5. compute $V = u_1G + u_2Q$
6. declare the signature valid if and only if $V = R$.

This process ensures that the signature was made by the private key holder and that the message was not altered since.[24]

The reason elliptic curves are so useful in cryptography is the Elliptic Curve Discrete Logarithm Problem (ECDLP). This problem is formulated in the following way: given two points P and Q on the curve, find the integer d such that $Q = dP$, if d exists. This operation is easy to follow in the other direction: given P and d, compute Q (in fact, this really just returns the public key from the private one), but no polynomial-time algorithm has been found to solve the original question, making it computationally impossible (even with quantum computers) to retrieve the private key from the public one. [22]

Furthermore, there are three common versions of ECDSA recommended by National Institute of Standards and Technology (NIST) and used in JWTs which differ in what curve is used:

- **ES256**: ECDSA version using the P-256 (secp256r1) curve. This curves operates on a prime field over a 256-bit prime number and provides a security level of 128 bits.

This is widely used around SSL/TLS cryptography. This algorithm uses the SHA256 hashing algorithm;

- **ES384**: ECDSA version using the P-384 (secp384r1) curve. This operates over a prime field with a 384-bit prime and provides a security level of 192 bits. This uses SHA384;
- **ES521**: ECDSA version using the P-521 (secp521r1) curve. This provides a security level of 260 bits and should be used where maximum security is required and computational efficiency is secondary. This uses SHA512.

[22] [23] [16]

> **Note on security levels**
> The security level indicates how strong and resilient the algorithm is. A security level of n bits means that the best theoretical attack would require 2^n operations to succeed. This is balanced out with computational efficiency, meaning that as the security level grows, the speed with which the signature is calculated drops.

3.3.2.3 EdDSA

The Edwards-Curve Digital Signature Algorithm (EdDSA) is a variant of the Digital Signature Algorithm (DSA) that uses twisted Edwards curves. Just as ECDSA, the EdDSA algorithm also relies on the complexity of the ECDLP problem, although it has a performance advantage over ECDSA because of the use of twisted Edwards curves. A twisted Edwards curve is a curve defined over a (finite) field K with a d element such that $d \neq 0 \neq 1$ in K. The curve follows the given equation

$$ax^2 + y^2 = 1 + dx^2y^2$$

where a and d are constants in K and a is a non-square in K. In cryptography K is usually chosen as a large prime field $\mathbb{F}_p$ or a binary field, just like in ECDSA. As can be seen in the equation, the twisted Edwards curve is really a specific type of elliptic curve, though a generalisation of Edwards curves, which are twisted Edwards curves with $a = 1$. There is two specific curves used in cryptography, namely Ed25519 and Ed448, with Ed25519 being much more popular in terms of usage. This curve is based on a field $\mathbb{F}_p$ with $p = 2^{255} - 19$. [25] The following paragraphs are going to focus on Ed25519 exclusively.

Key Generation
Key generation is much simpler than with ECDSA. The private a key is simply an integer selected at random from a defined range, while the public key A is calculated as $A = aB$ where B is the base point of the curve. [26]

Signing
Given the message m to be signed and the private key a, the signing process goes as follows:

1. compute $H = \text{SHA512}(a)$ and split it into two 256 bit-parts: H_0 and H_1;
2. compute the nonce $r = \text{SHA512}(h_0 \,||\, m)$;
3. calculate the point $R = rB$;
4. calculate $h = \text{SHA512}(R_x \,||\, A \,||\, m)\ mod\ q$, where q is the order of the base point B;
5. calculate $s = (r + hH_1)\ mod\ q$.

The resulting signature will be

$$(R, s).$$

In this algorithm, the nonce r is calculated in a deterministic way rather than randomically as in ECDSA, this improves on ECDSA's possible poor random number generation vulnerability. Then the challenge h is bound to the signature ensuring integrity. Finally, the signature is built using all components and making it hard to compute it without a private key. [26] [27]

Verification
Given the message m, the public key A and the signature R, the verification process goes as follows:

1. calculate $h = \text{SHA512}(R \,||\, A \,||\, m)\ mod\ q$
2. calculate $P_1 = sB$
3. calculate $P_2 = R + hA$
4. check if $P_1 = P_2$ and return the result.

If the result is true, it means that P_1 and P_2 are the same EC point, but P_1 was calculated using the private key and P_2 using the corresponding public key, this means that the keys actually correspond. [26] [27] The JWS signing algorithm is EdDSA.

3.3.2.4 RSA

The last algorithm discussed in this thesis is Rivest-Shamir-Adleman (RSA). While probably the most known and common, this algorithm has been deemed insecure by NIST for a while and is not recommended to use anymore. There are two versions of RSA depending on what padding is used: Probabilistic Signature Scheme (PSS) and PKCS#1-v1.5. With the first one being the *slightly* more secure one of the two, this is the only one that will be discussed here. [28] RSA is based on the mathematical difficulty of factoring the product of two numbers.

Key Generation What follows is the process of generating an RSA key pair:

1. choose p and q such that they are two sufficiently and distinct large prime numbers;
2. compute $n = pq$. This will become the modulus of the keypair and its length will indicate the key length;
3. compute $\phi(n) = (p-1)(q-1)$;
4. choose an integer e such that $1 < e < \phi(n)$ and e is coprime to $\phi(n)$. This will become the key's public exponent and is usually chosen as 65537;
5. determine $d = e^{-1} \; mod \; \phi(n)$. This will be the private key exponent.

The result will be the public key defined as

$$(n, e)$$

and the private key defined as

$$(n, e, d).$$

[28] [29] [30]

Signing Signing using RSA is quite straightforward:

1. hash the message m producing H;
2. a random salt s is added to the message and they both get padded using the PSS padding;
3. the private key is used to sign the padded message producing the signature.

[30]

Verification Finally, verification goes as follows:

1. compute the encoded message with signature exponentiation: $S_e \; mod \; n$ using e and n which can be found in the public key;
2. extract the hash of the original message H' and the salt s';
3. hash the received message M' to get H;
4. construct a verification block with H and s' with PSS padding, if this block matches the received block and $H = H'$ then the signature is valid.

[30] The JWS signing algorithms using RSA are PS256, PS384 and PS512 for PSS padding, RS256, RS384 and RS512 for normal RSA PKCS#1-v1.5. [16]

3.4 JWTs in Vapor

Support for JWTs in Vapor is provided through the JWTKit package and the JWT wrapper package. JWTKit is the main library providing the API used for signing and verifying tokens while the JWT package provides first class integration to use JWTKit with Vapor. As explained in the authentication part of this chapter, the server creates a token after the user has logged in and returns it to the user, who then sends it to the server inside the header of every request. First of all a payload has to be created:

```
struct UserPayload: JWTPayload, Authenticatable, Content {
    var email: String
    var exp: ExpirationClaim
}
```

This payload contains the email of the user and an `ExpirationClaim`, which is a claim type provided by JWTKit and contains the expiration of the token in Unix time.

> **Note**
> This code will not compile directly as a `verify` method is required by the `JWTPayload` protocol, though it is omitted as it is not relevant in this specific context.

The token can be used like this:

```
app.post("login") { req -> String in
    let loginData = try req.content.decode(LoginData.self)
    // Authenticate the user with the provided credentials...

    let userPayload = UserPayload(
        email: loginPayload.email,
        exp: ExpirationClaim(value: Date().addingTimeInterval(3600))
    )
    let token = try req.jwt.sign(userPayload)
    return token
}
```

In this route the user is logged in (the login code is omitted and can be lifted from the second chapter of this thesis), afterwards the token is created with their data and is returned. Finally, the token can be used to protect routes:

```
let secure = app.grouped(
    UserPayload.authenticator(),
    UserPayload.guardMiddleware()
)
```

```
secure.post("validateLoggedInUser") { req -> HTTPStatus in
    let token = try req.auth.require(UserPayload.self)
    return .ok
}
```

Here an authenticator is added to the `"validateLoggedInUser"` route which checks for the presence of the token in the request header, in particular in the `Authorization` field. Since the token is `Authenticatable`, the authenticator is created automatically. Then, `guardMiddleware()` is called, a middleware which returns an unauthorised error if authentication failed. [11]

The next chapter will discuss how JWTKit works internally, with focus on its latest version rewrite, including the switch from BoringSSL to SwiftCrypto, new customisation features and more.

4. JWTKit v5

This chapter will speak about the rewrite of JWTKit which lead to its major version number 5. During the rewrite, the package underwent a major overhaul which updated internals to switch from the underlying BoringSSL library to use SwiftCrypto, the native Swift library created by Apple providing cryptographic operations. Additionally, the public API also got an update, with the addition of the RSA-PSS algorithm, custom header and custom serialising and parsing and more. In this chapter the differences and improvements between version 4 and version 5 will be discussed, starting with the package structure and the public API changes.

4.1 Package Structure and Public API

First of all, this section will take a look at how the project structure changed from version 4 to version 5; this will be useful to understand the next sections of this chapter too.

4.1.1 Version 4

While an anticipation was given in section 3.4, the type used to represent JWTs is the `JWTPayload`: this protocol represents the payload of the token. Users can create types conforming to this protocol and can add the custom claims they desire, also specifying a custom `verify` method. For example:

```
struct UserPayload: JWTPayload {
    var email: String
    var exp: ExpirationClaim

    func verify(using signer: JWTSigner) throws {
        try self.exp.verifyNotExpired()
    }
}
```

This token is similar to the one in section 3.4, but with the addition of the `verify` method which is required. The payload can both get signed, returning the compact serialisation format of the token as a `String`, and be used to verify other string tokens. Signing and verification happens through a `JWTSigners` type, which is basically a collection of `JWTSigner` instances. Each one of them is identified by a `kid` and represents a, usually different from the others, signing algorithm. Keys can be created and then added to the signers collection:

```
let ecdsaPublicKeyPEM = """
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE2adMrdG7aUfZH57aeKFFM01dPnkx
C18ScRb4Z6poMBgJtYlVtd9ly63URv57ZW0Ncs1LiZB7WATb3svu+1c7HQ==
-----END PUBLIC KEY-----
"""
// Initialise an ECDSA key with public PEM
let key = ECDSAKey.public(pem: ecdsaPublicKeyPEM)
// Create signers
let signers = JWTSigners()
// Add the key to the signers
try signers.use(.es256(key: key), kid: "some-kid")
```

This snippet creates a public ECDSA key from a Privacy Enhanced Mail (PEM) file and adds it to a new signer collection, adding the "some-kid" identifier to it so the correct signer can be chosen between the other ones. This can then be used to sign a payload:

```
let payload = UserPayload(
    email: "some-email",
    exp: .init(value: .distantFuture)
)
let token = try signers.sign(payload, kid: "some-kid")
```

The `token` constant will then contain the compact serialisation form of the token and can be verified using the same signers:

```
let payload = try signers.verify(
    token,
    as: UserPayload.self,
    kid: "some-kid"
)
```

If verification succeeds, the `payload` constant will contain the original payload.[31]

4.1.2 Version 5

In version 5, the payload API remains the same, besides the changes to the `verify` method:

```
func verify(using key: JWTAlgorithm) async throws {
    try self.exp.verifyNotExpired()
}
```

Since `JWTSigner` became internal, `JWTAlgorithm` can be used instead. This protocol represents a signing algorithm. The main change in the new API is the replacement of the `JWTSigners` class with an `actor` named `JWTKeyCollection`. They work in a similar way, both encapsulating a collection of signing algorithms, though the main difference, since the new structure is an `actor`, is that it is inherently thread safe, as explained in the first chapter of this thesis. This is how a key collection is created:

```
// Create a key using the same PEM string as above
let key = ES256PublicKey(pem: ecdsaPublicKeyPEM)

// Create a key collection and add the key to it
let keys = JWTKeyCollection()
try await keys.addES256(key: key, kid: "some-kid")
```

Afterwards, the key collection can be used both to sign:

```
let token = try await keys.sign(payload, header: ["kid": "private"])
```

and verify:

```
let payload = try await keys.verify(token, as: UserPayload.self)
```

The main differences here are that, since the key collection has become an `actor`, the methods on it are all `async` and must therefore be awaited `await`; also, the header does not receive the token's header parameters as function parameters, but it takes a `JWTHeader`, this is dictionary containing header fields to be added to the token, including the possibility to add custom ones which will be discussed later on.
Each signer is now a key and has therefore a different API, for example, other than the ES256 algorithm shown above:

```
- EdDSAKey.PublicKey(x: eddsaPublicKeyBase64, curve: .ed25519)
+ EdDSA.PublicKey(x: eddsaPublicKeyBase64, curve: .ed25519)

- RSAKey.private(pem: pem)
+ Insecure.RSA.PrivateKey(pem: pem)
```

Since `RSA` is not recommended besides out of compatibility reasons, the key is now gated behind an `Insecure` namespace. [32]

While the public API changes are not huge, they were certainly necessary due to the restructuring of the internals, which will be discussed next.

4.2 BoringSSL vs SwiftCrypto

Internally, the most important change which version 5 brings is the eradication of BoringSSL, which got replaced by SwiftCrypto. This section will speak about the two different libraries and how the transition to SwiftCrypto makes JWTKit's maintenance much easier.

4.2.1 BoringSSL

Until version 4, JWTKit was really a wrapper around a vendored version of BoringSSL. BoringSSL is Google's fork on OpenSSL, a low level C library providing cryptographic operations. As stated on their Git repository, BoringSSL was built to comply with Google's needs, which OpenSSL didn't completely satisfy. Once the differences started becoming big, Google decided to open source the project and provide users with vendored copies of the library, although not providing API stability and thus allowing breaking changes. [33] Thanks to Swift's interoperability with C, functions from BoringSSL can be easily called from Swift code using unsafe constructs with direct access to memory such as `OpaquePointer`s, `UnsafePointer`s and `UnsafeMutablePointer`s. Having to use unsafe, not type checked and not memory safe constructs in a modern language such as Swift is not ideal, that's why a different approach was chosen.

4.2.2 SwiftCrypto

SwiftCrypto is Apple's first open source cryptographic library. Apple has had its CryptoKit library for a long time, providing native cryptographic operations to its platforms (iOS, macOS etc.). This library is closed source, which means that its code is not public and it is only maintained by Apple itself. In 2020, Apple released SwiftCrypto, an open source library providing cryptographic operations to both Apple platform and Linux: this was a giant step for server side Swift. The library closely follows the CryptoKit API and, on Apple platforms, actually uses the CryptoKit code directly. On Linux, on the other hand, it uses BoringSSL, the library mentioned above which JWTKit managed to get away from. While this may seem just a distancing rather than a proper farewell to BoringSSL, being able to abstract away low-level C calls from a modern Swift library is a big advancement.

4.2.2.1 CryptoExtras

SwiftCrypto has been built with the idea of creating an easy to use cryptographic library, this means that it had to strip away complexities from normal crypto libraries such as BoringSSL, and these complexities also include some old and/or unused algorithms, as having too many algorithms for hashing, encrypting or signing may be a source of confusion for some users. One of these algorithms is RSA. RSA is not included in CryptoKit because of security and efficiency reasons, therefore it is not in SwiftCrypto either. This is the reason the `_CryptoExtras` module was created, basically an extension of the package which is not directly part of the main SwiftCrypto API, but provides support for less safe and modern algorithms and operations, mainly for compatibility reasons. Since often JWTs require RSA and PSS, support was needed for those and the `_CryptoExtras` package was used. Unfortunately one specific API was missing, which creates RSA keys based on modulus, public exponent and private exponent, that's why this API was directly implemented in

JWTKit, though more on this will be discussed later on.

Following are some practical code examples on how the code changed with the transition.

4.2.3 Key Creation

Version 4

Before the key code itself, a method must be introduced that takes a PEM formatted key and processes it using a passed in closure:

```
static func load<Data, T>(
    pem data: Data,
    _ closure: (UnsafeMutablePointer<BIO>) -> (T?)
) throws -> T
    where Data: DataProtocol
{
    // Convert the input Data to a byte array for C API compatibility
    let bytes = data.copyBytes()

    // Create a new BIO memory buffer with the PEM data.
    // BIO is a fundamental I/O abstraction in OpenSSL/BoringSSL.
    // `CJWTKitBoringSSL_BIO_new_mem_buf` creates a BIO
    // that operates in memory with the given data
    let bio = CJWTKitBoringSSL_BIO_new_mem_buf(
        bytes,
        numericCast(bytes.count)
    )

    // Ensure that the BIO is freed after use to prevent memory leaks.
    defer { CJWTKitBoringSSL_BIO_free(bio) }

    // Ensure the BIO was successfully created and that the closure doesn't
    // return nil. If either fails, throw an error indicating the failure.
    guard let bioPtr = bio, let c = closure(bioPtr) else {
        throw JWTError.signingAlgorithmFailure(
            OpenSSLError.bioConversionFailure
        )
    }

    // Return the result of the closure if everything succeeds
    return c
```

```
}
```

Every BoringSSL function called has a `CJWTKitBoringSSL_` prefix, as that is what the vendored library is called. Then, following is the code to actually create a public key from a PEM file:

```
public static func `public`<Data>(pem data: Data) throws -> ECDSAKey
    where Data: DataProtocol
{
    // Calls the 'load' method defined previously,
    // passing the PEM data and a closure.
    let c: OpaquePointer = try self.load(pem: data) { bio in
        // Attempts to read an EC public key from the BIO memory buffer.
        // The 'nil' arguments are placeholders for callback functions
        // and user data, which are not needed for this operation.
        CJWTKitBoringSSL_PEM_read_bio_EC_PUBKEY(bio, nil, nil, nil)
    }
    return self.init(c)
}
```

Afterwards, the `c` constant will contain an `OpaquePointer` with the initialised public key. JWTKit used to simply keep an `OpaquePointer` reference to pass around to the various C functions that needed it.
Finally, an interesting method in BoringSSL is the creation of an EC curve using its coordinates, which in JWTKit v4 was done as follows:

```
public convenience init(
    parameters: Parameters, // public key coordinates (x, y)
    curve: Curve = .p521,
    privateKey: String? = nil
) throws {
    // Create a new EC key object based on the specified curve
    let c = CJWTKitBoringSSL_EC_KEY_new_by_curve_name(curve.cName)

    // Convert the base64URL encoded 'x' and 'y' coordinates to a BigNumber
    let bnX = BigNumber(base64URL: parameters.x)
    let bnY = BigNumber(base64URL: parameters.y)

    // Set the public key coordinates for the EC key
    CJWTKitBoringSSL_EC_KEY_set_public_key_affine_coordinates(
        c,
        bnX.c,
```

```
            bnY.c
        )

        // If a private key is provided, attempt to set it for the EC key.
        if let privateKey = privateKey {
            let bnPrivate = BigNumber(base64URL: privateKey)
            CJWTKitBoringSSL_EC_KEY_set_private_key(c, bnPrivate.c)
        }

        // Initialise the EC key object with the created EC key.
        self.init(c)
    }
```

Disclaimer: this snippet was greatly simplified from the original for demonstration purposes: all the C functions return optionals and each C call is wrapped around a guard statement which throws an error if the function fails.
The example shows how an `OpaquePointer` containing a key can be initialised using the keys coordinates and an optional private key. [31]

Version 5
For dealing with EC keys, SwiftCrypto provides namespaces for the most common ones, namely `P256`, `P384` and `P521`. Unfortunately these do not share a common protocol as JWTKit would need, that's why a custom one had to be build in JWTKit directly:

```
public protocol ECDSACurveType: Sendable {
    associatedtype Signature: ECDSASignature
    associatedtype PrivateKey: ECDSAPrivateKey
    static var curve: ECDSACurve { get }
    static var byteRanges: (x: Range<Int>, y: Range<Int>) { get }
}
```

Each of SwiftCrypto's curves then conforms to that:

```
extension P256: ECDSACurveType, @unchecked Sendable {
    public typealias Signature = P256.Signing.ECDSASignature
    public typealias PrivateKey = P256.Signing.PrivateKey

    public static let curve: ECDSACurve = .p256

    /// Specifies the byte ranges in which the X and Y coordinates
    /// of an ECDSA public key appear for the P256 curve
    public static let byteRanges: (x: Range<Int>, y: Range<Int>) =
```

```
        (1 ..< 33, 33 ..< 65)
}
```

The curve type is then used in the `ECDSAKey` protocol:

```
public protocol ECDSAKey: Sendable {
    associatedtype Curve: ECDSACurveType
}
```

This then allows to create a namespace for ECDSA:

```
public enum ECDSA: Sendable {}
```

which will contain two structures, one for the public key and one for the private one, each one equipped with a generic curve. Finally, keys can be created like:

```
public init(pem string: String) throws {
    backing = try PublicKey(pemRepresentation: string)
}
```

which is possible thanks to the `PublicKey` and `PrivateKey` associated types both having an `init(pemRepresentation:)` initialiser. In comparison to C code, the only complication here is the abstraction of code using protocols to create a clean API, but if it were just about creating a P256 public key from a PEM encoded String, in SwiftCrypto it is a one-liner:

```
let privateKey = try P256.Signing.PublicKey(pemRepresentation: string)
```

which, compared to the BoringSSL code from earlier, is quite neat. The last method of the previous paragraph is also much simpler using SwiftCrypto:

```
public init(parameters: ECDSAParameters) throws {
    guard
        let x = parameters.x.base64URLDecodedData(),
        let y = parameters.y.base64URLDecodedData()
    else {
        throw JWTError.generic(
            identifier: "ecCoordinates",
            reason: "Unable to interpret x or y as base64 encoded data"
        )
    }
    backing = try PublicKey(x963Representation: [0x04] + x + y)
}
```

This creates a new ECDSA key using the x and y coordinates. [32]

4.2.3.1 Signing

Version 4

To use the stored `OpaquePointer` for signing, JWTKit 4 used this method:

```
func sign<Plaintext>(_ plaintext: Plaintext) throws -> [UInt8]
    where Plaintext: DataProtocol
{
    // Compute the digest (hash) of the plaintext
    let digest = try self.digest(plaintext)

    // Sign the digest using ECDSA.
    // This function requires the digest, its length, and the EC key
    let signature = CJWTKitBoringSSL_ECDSA_do_sign(
        digest,
        numericCast(digest.count),
        self.key.c
    )

    // Free the signature structure to prevent memory leaks
    defer { CJWTKitBoringSSL_ECDSA_SIG_free(signature) }

    // Extract the r and s values from the signature
    let r = CJWTKitBoringSSL_ECDSA_SIG_get0_r(signature)
    let s = CJWTKitBoringSSL_ECDSA_SIG_get0_s(signature)
    // Determine the size of the result based on the signing curve
    // This affects how many bytes are needed for r and s
    let rsSize = self.curveResultSize

    // Prepare byte arrays to hold the binary representation of r and s,
    // initialised with zeros
    var rBytes = [UInt8](repeating: 0, count: rsSize)
    var sBytes = [UInt8](repeating: 0, count: rsSize)

    // Convert r and s from BIGNUM to binary form
    // and store them in the prepared byte arrays
    let rCount = Int(CJWTKitBoringSSL_BN_bn2bin(r, &rBytes))
    let sCount = Int(CJWTKitBoringSSL_BN_bn2bin(s, &sBytes))

    // Assemble the signature by concatenating r and s,
    // ensuring each is zero-prefixed to the required length if necessary.
    // This step handles potential issues with byte representation
```

```
    // in ECDSA signatures, ensuring the signature format
    // conforms to expectations (e.g., as per RFC 7515)
    return rBytes.prefix(rCount).zeroPrefixed(upTo: rsSize)
        + sBytes.prefix(sCount).zeroPrefixed(upTo: rsSize)
}
```

This creates the signature of the plaintext which can then be attached to the token. [31]

Version 5
The same code in JWTKit 5 is the following:

```
func sign(_ plaintext: some DataProtocol) throws -> [UInt8] {
    // Compute the digest (hash) of the plaintext
    let digest = try self.digest(plaintext)

    // Check if the key is a private key, throw otherwise
    guard let privateKey else {
        throw JWTError.signingAlgorithmFailure(ECDSAError.noPrivateKey)
    }

    // Calculate the signature
    let signature = try privateKey.backing.signature(for: digest)

    // Convert it to [UInt8]
    return [UInt8](signature.rawRepresentation)
}
```

The code is quite a bit shorter, easier to understand and uses a more modern syntax. The `some` keyword is also in action here, representing a `DataProtocol`-conforming opaque type. It's worth noting how few comments are in the second version compared to the first one: this is an indicator of how clear the code itself is. [32]

4.2.3.2 Verification

Version 4
Finally, using BoringSSL, ECDSA verification is done as follows:

```
func verify<Signature, Plaintext>(
    _ signature: Signature, // The signature to be verified
    signs plaintext: Plaintext // The data that was supposedly signed
) throws -> Bool
    where Signature: DataProtocol, Plaintext: DataProtocol
```

```
{
    // Compute the digest (hash) of the plaintext
    let digest = try self.digest(plaintext)

    let signatureBytes = signature.copyBytes()
    // Determine the size of r and s based on the elliptic curve used
    let rsSize = self.curveResultSize
    // If the signature size doesn't match the expected, return false
    guard signatureBytes.count == rsSize * 2 else {
        return false
    }

    // Create a new ECDSA signature object
    let signature = CJWTKitBoringSSL_ECDSA_SIG_new()
    defer { CJWTKitBoringSSL_ECDSA_SIG_free(signature) }

    // Split the signature bytes into r and s components
    // and set them on the ECDSA signature object
    try signatureBytes.prefix(rsSize).withUnsafeBufferPointer { r in
        try signatureBytes.suffix(rsSize).withUnsafeBufferPointer { s in
            // Convert the byte buffers for r and s into BIGNUMs
            // and set them on the signature object
            CJWTKitBoringSSL_ECDSA_SIG_set0(
                signature,
                CJWTKitBoringSSL_BN_bin2bn(r.baseAddress, rsSize, nil),
                CJWTKitBoringSSL_BN_bin2bn(s.baseAddress, rsSize, nil)
            )
        }
    }

    // Perform the verification using the signature object
    // and the digest of the plaintext.
    // Returns true if the signature is valid, false otherwise
    return CJWTKitBoringSSL_ECDSA_do_verify(
        digest,
        numericCast(digest.count),
        signature,
        self.key.c // The EC key associated with the signature.
    ) == 1 // A return value of 1 indicates success (valid signature).
}
```

This snippet shows the verification function used by JWTKit v4. [31]

Version 5
The same verification using SwiftCrypto is done as:

```
public func verify(
    _ signature: some DataProtocol,
    signs plaintext: some DataProtocol
) throws -> Bool {
    // Calculate the digest (hash)
    let digest = try self.digest(plaintext)

    // Return whether the signatu
    return try publicKey.backing.isValidSignature(signature, for: digest)
}
```

[32] The difference between the two is blatant and explains in a practical way how and why SwiftCrypto is a better fit for any modern Server Side Swift (SSS) library in need of cryptographic support.

4.2.4 C vs Swift

A note has to be made on the replacement of C with Swift. The safety of Swift in general was discussed in the first chapter, but one cannot emphasise enough how advantageous it is to replace all C calls with Swift. C, being a low level language, is powerful but, at the same time, brings many disadvantages. Using C inside of Swift implies constructs such as `UnsafePointer`s, of which the name is enough to discourage their use. Removing usage of such constructs allows safe memory usage, without having to deal with unfreed memory and having to copy data into a buffer. Swift's ARC reduces the risk of dangling pointers and memory leaks and Swift's type safety allows the compiler to clearly know the type of each variable, therefore usage of constructs like `OpaquePointer`s, which do not reveal the type of a variable and can result in problems at runtime, is avoided. Finally, error handling is improved, as calling BoringSSL functions often implied checking for optionals rather than throwing errors with Swift's normal `do`, `try` and `catch` mechanisms.

4.2.5 RSA "Raw" API

For handling JWKs (which are not discussed in this thesis, even though JWTKit supports them) the previous version of JWTKit included an RSA key initialiser which used its raw parameters, modulus, exponent and optionally public exponent, to create a key. SwiftCrypto, or rather CryptoExtras, doesn't provide this API, that is why it had to be

added manually. This section explains how it was added.
As explained earlier, RSA keys are based on the calculation, or rather the search, of two prime factors, p and q. A similar algorithm to the Miller-Rabin primality test was used to calculate those: it can be found in the Appendix A.1. [34] After having calculated both prime factors, the chinese remainder theorem [35] is used to get the remaining parts of the private key, such as dp, dq and $qInv$. These are all components used by RSA internally and contribute to its Abstract Syntax Notation One (ASN.1) form. In fact, with these components the ASN.1 form can be created, specifically using the Swift ASN1 package. This provides utilities to handle all kinds of ASN1 forms using a `DERSerializable` protocol to which such structures can conform to. In JWTKit two were created, one for the RSA public key and one for the private one, following is the private key one:

```
struct ASN1: DERSerializable {
    let version: UInt8
    let modulus: ArraySlice<UInt8>
    let publicExponent: ArraySlice<UInt8>
    let privateExponent: ArraySlice<UInt8>
    let prime1: ArraySlice<UInt8>
    let prime2: ArraySlice<UInt8>
    let exponent1: ArraySlice<UInt8>
    let exponent2: ArraySlice<UInt8>
    let coefficient: ArraySlice<UInt8>
}
```

[36] The key can then be constructed using this structure:

```
var serializer = DER.Serializer()

guard
    let privateKeyDER = try Insecure.RSA.calculatePrivateDER(
        n: n,
        e: e,
        d: d
    )
else {
    throw RSAError.keyInitializationFailure
}
try privateKeyDER.serialize(into: &serializer)
let privateKey = try _RSA.Signing.PrivateKey(
    derRepresentation: serializer.serializedBytes
)
```

and can be finally used with the usual structure, whether with PKCS#1-v1.5 or PSS. [32]

4.3 PSS Addition

Although not particularly new, JWTKit 5 introduced the PSS algorithm family to the package. PSS is a slightly more secure version of RSA as it tries to fill the gaps of the traditional RSA PKCS#1-v1.5 scheme. Chapter 3 already explained how it works, so following is how it was implemented in JWTKit using `_CryptoExtras`. The PSS algorithm family still uses RSA keys underneath, just the padding used in signing is different, that is why the key structure is the same as traditional RSA signing. All keys in JWTKit 5 basically have structure as the ECDSA one explained in the previous section:

```
public extension Insecure {
    enum RSA: Sendable {}
}
```

a namespace for RSA keys, gated behind the `Insecure` namespace to discourage people from using it, and:

```
public protocol RSAKey: Sendable {}
```

, a protocol for the keys to conform to. Then there are both a `PublicKey` and a `PrivateKey` structure inside of the namespace and conforming to `RSAKey`. The important difference with RSA-PKCS#1-v1.5 is in the `RSASigner`, as that now takes a `padding` argument in its initialiser:

```
struct RSASigner: JWTAlgorithm, CryptoSigner {
    let publicKey: Insecure.RSA.PublicKey
    let privateKey: Insecure.RSA.PrivateKey?
    var algorithm: DigestAlgorithm
    let name: String
    let padding: _RSA.Signing.Padding

    init(
        key: some RSAKey,
        algorithm: DigestAlgorithm,
        name: String,
        padding: _RSA.Signing.Padding
    ) {
        ...
    }
```

This `padding` is of type `_RSA.Signing.Padding`, an enum inside of `_CryptoExtras` which can be either `insecurePKCS1v1_5` or `PSS`. Now, when adding a signer, JWTKit provides new methods to allow adding the PSS family algorithms, namely

```
@discardableResult
func addPS256(
    key: some RSAKey,
    kid: JWKIdentifier? = nil,
    parser: some JWTParser = DefaultJWTParser(),
    serializer: some JWTSerializer = DefaultJWTSerializer()
) -> Self {
    add(.init(
        algorithm: RSASigner(
            key: key,
            algorithm: .sha256,
            name: "PS256",
            padding: .PSS
        ),
        parser: parser,
        serializer: serializer
    ),
    for: kid)
}
```

and its other two PS384 and PS512 variations, which just differ in the hashing algorithm. This method also spoils two other version 5 novelties: the `JWTParser` and the `JWTSerializer`. While they were present in the previous version too, they are now public, but they will be explained in more detail later on.

4.4 Modern Concurrency Adoption

JWTKit 5 also brought `Sendable` conformance and modern concurrency to the package. As explained in chapter 1, the `Sendable` protocol denotes types which are safe to be sent across concurrency domains. Conforming types to `Sendable` is particularly useful in a library such as JWTKit because, in a Vapor server for example, multiple requests might need to sign or validate JWTs, and `Sendable` ensures that the requests access the common tokens in a safe way, preventing issues like data races. Distributed systems are also a big use case for JWTs and `Sendable` makes it easy to use these types across different, distributed and concurrent environments. Other than `Sendable`, the `JWTKeyCollection` (renamed from `JWTSigners`) is now an `actor`, meaning that access to its shared state is concurrency safe. This requires all methods defined on the actor to be `async`, therefore `sign` and `verify` APIs are now to be `await`ed. The little added complexity is an acceptable downside if it means that all the elements are concurrency safe, especially in a server environment. Adopting `Sendable` and modern concurrency is easy: it's enough to start from one type and the compiler warnings

will slowly propagate across the whole package, indicating which other types need to be conformed too. In some special cases conformance is not enough and the types' shared state has to be made explicitly thread safe. This did not happen in JWTKit, however it did when updating the JWT package to use the new version of JWTKit.
The idea is that a `JWTKeyCollection` instance has to be exposed to the user, and usually this is done by creating a `Storage` class to attach to Vapor's `Application`. Classes are somewhat problematic to use in concurrency, as they are reference types which are not thread safe by default. To solve this issue and make the access to the `JWTKeyCollection` thread safe, a type of the SwiftNIO's NIOConcurrencyHelper module was used:

```
private struct SendableBox: Sendable {
    var keys: JWTKeyCollection
}

private let sendableBox: NIOLockedValueBox<SendableBox>

var keys: JWTKeyCollection {
    get {
        self.sendableBox.withLockedValue { box in
            box.keys
        }
    }
    set {
        self.sendableBox.withLockedValue { box in
            box.keys = newValue
        }
    }
}
```

This snippet shows the creation of a `Sendable struct` *inside* of the class, containing the instance needed. To access the instance from the class, a computed property exists which does not reference the keys directly, but through `NIOLockedValueBox`. This helper from SwiftNIO is used to create a lock mechanism around the type it wraps, giving runtime thread safety to the keys' access. While it is not ideal, being the rest of the code *compile time* safe, using a lock is the more traditional way of handling sequential access and, while just at runtime, guarantees that there will not be any data races on the property. [9]

4.5 Customisation

As previously spoiled a few sections ago, JWTKit 5 introduced some new customisation features for parsing and serialising. The idea behind that is that a lot of definitions and

usages exists around the JavaScript Object Signing and Encryption (JOSE) standard, and maintaining them all in one package can become tricky. This is why it was decided to allow users to implement their own specifications where JWTKit stopped providing them explicitly.

4.5.1 Custom Header Fields

The first novelty in regards to customisation is custom headers. While the JWS (RFC 7515) specification lists a number of default headers, of which only the `alg` one is mandatory, JOSE headers can contain custom fields too. The header field in JWTKit 5 is a dictionary that looks like:

```
@dynamicMemberLookup
public struct JWTHeader: Sendable {
    public var fields: [String: JWTHeaderField]

    public init(fields: [String: JWTHeaderField] = [:]) {
        self.fields = fields
    }

    public subscript(dynamicMember member: String) -> JWTHeaderField? {
        get { fields[member] }
        set { fields[member] = newValue }
    }
}
```

There are two important observations to be made about this: the `JWTHeaderField` type and the `@dynamicMemberLookup` attribute. The `JWTHeaderField` is the actual field inside of the header. It is an `enum` that stores JSON data and it looks like this:

```
public indirect enum JWTHeaderField: Hashable, Sendable, Codable {
    case null
    case bool(Bool)
    case int(Int)
    case decimal(Double)
    case string(String)
    case array([JWTHeaderField])
    case object([String: JWTHeaderField])
}
```

There is actually a lot more code around it (like custom en/decoding and various extensions) but this represents the essence of it. Any JOSE-compliant header field can be

represented using this `enum` and there are some elegant extensions (which Swift's standard library provides) that allow header fields to be used as Swift primitives. For example, using the following extension:

```
extension JWTHeaderField: ExpressibleByStringLiteral {
    public init(stringLiteral value: StringLiteralType) {
        self = .string(value)
    }
}
```

the header field can be expressed as `String`s directly rather than having to pass through the `.string()` syntax. Practically speaking, this is now possible:

```
- token.header["alg"] = JWTHeaderField.string("HS256")
+ token.header["alg"] = "HS256"
```

Such an extension is provided for every type, namely `String`, `Int`, `Bool`, `Float`, `Array`, `Dictionary` and even `nil`.
`@dynamicMemberLookup` on the other hand is an attribute which can be added to types to enable property access as if the properties were directly defined on them, even if they are not. The access to those fields is handled through the `subscript(dynamicMember:)` method. In the case of the `JWTHeader`, the method accesses the `fields` dictionary which contains all header fields of the token. This means the following:

```
- token.header["alg"] = "HS256"
+ token.header.alg?.asString = "HS256"
```

This is even safely typed, which means that assigning anything else than a `JWTHeaderField` is not allowed. [37] Additionally, JWTKit provides extension for common header fields (which are enough for most users) that abstract away direct access to the dictionary:

```
var alg: String? {
    get { self[dynamicMember: "alg"]?.asString }
    set { self[dynamicMember: "alg"] = newValue.map { .string($0) } }
}
```

This is provided for fields like `alg`, `kid` and more.
Finally, all of this means that custom header fields can be implemented, for example the `b64` header which indicates whether the payload is base64URL encoded or not, simply by using:

```
let token = try await keyCollection.sign(payload, header: ["b64": false])
```

to add it to the token when signing, and

```
header.b64?.asString
```

to access it, in a type safe way. [32] [16]

4.5.2 Serialising and Parsing

While keeping the `b64` header field example in mind, some header fields require the structure of the token to change with them. For example, the `b64` header field indicates whether a token's payload is base64URL encoded or not. Usually the field is omitted and thus assumed to have the value `true`: that is what a standard JWT token looks like. When the field is present and set to `false` on the other hand, the token will look something like:

```
eyJiNjQiOmZhbHNlLCJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
.
{"name":"Foo","exp":2000000000,"admin":false,"sub":"vapor"}
.
NuN8EQPF7AwvnysdpSdFnjwpKO4gf4E2nkGggOPLrE8
```

As this is a somewhat unorthodox token, but still valid in some scenarios, JWTKit supports it by implementing custom parsing and serialising. Users can choose how to parse and serialise a token according to some standard or to their liking, simply by implementing their own `JWTSerializer` and `JWTParser`. The `JWTSerializer` is used to serialise the token from Swift constructs (which means `JWTPayload` and `JWTHeader`) into the compact serialisation form of the token. This is used to sign the token. The `JWTParser` on the other hand does the contrary, parsing the compact serialised token into data that can be used in Swift, usually for verification. This can be done as follows: first a custom `JWTSerializer` is created

```
struct CustomSerializer: JWTSerializer {
    // Here a custom encoder can be used
    var jsonEncoder: JWTJSONEncoder = .defaultForJWT

    // This method should return the payload the way the user wants
    func serialize(
        _ payload: some JWTPayload,
        header: JWTHeader
    ) throws -> Data {
        // Check if the b64 header is set.
        // If it is, base64URL encode the payload, don't otherwise
        if header.b64?.asBool == true {
            try Data(jsonEncoder.encode(payload).base64URLEncodedBytes())
        } else {
            try jsonEncoder.encode(payload)
        }
    }
}
```

Then, a parser needs to be implemented too:

```
struct CustomParser: JWTParser {
    var jsonDecoder: JWTJSONDecoder = .defaultForJWT

    // This method parses the token into a tuple
    // containing the various token's elements
    func parse<Payload>(
        _ token: some DataProtocol,
        as: Payload.Type
    ) throws -> (header: JWTHeader, payload: Payload, signature: Data)
        where Payload: JWTPayload
    {
        // A helper method is provided to split the token correctly
        let (encodedHeader, encodedPayload, encodedSignature) = try
        ↪  getTokenParts(token)

        // The header is usually always encoded the same way
        let header = try jsonDecoder.decode(
            JWTHeader.self,
            from: .init(encodedHeader.base64URLDecodedBytes())
        )

        // If the b64 header field is non present or true,
        // base64URL decode the payload, don't otherwise
        let payload = if header.b64?.asBool ?? true {
            try jsonDecoder.decode(
                Payload.self, from:
                .init(encodedPayload.base64URLDecodedBytes())
            )
        } else {
            try jsonDecoder.decode(
                Payload.self, from:
                .init(encodedPayload)
            )
        }

        // The signature is usually also always encoded the same way
        let signature = Data(encodedSignature.base64URLDecodedBytes())

        return (header: header, payload: payload, signature: signature)
```

```
    }
}
```

These methods are then both used internally to sign and verify the token, although they do have to be registered first:

```
let keyCollection = await JWTKeyCollection().addHS256(
    key: "secret",
    parser: CustomParser(),
    serializer: CustomSerializer()
)
```

This way, any token handled by that `keyCollection` instance will go through the parser and serialiser defined earlier, for example signing a token as was done in the previous subsection using the `b64` header. This API provides JWTKit with a solid foundation to support many options which other packages may be too limited to support, and even specifications that may come in the future, making it a much more flexible package than before. [32]

4.6 X5C Verification

The last one of JWTKit 5's novelties touched by this thesis is X5C verification. As shortly defined in chapter 3, the `x5c` header parameter in a token contains a chain of X.509 certificates used to validate the authenticity of the token, with each certificate validating the next one up to a root certificate issued by a trusted certificate authority. The root one is supposed valid, and if all other certificates are valid too, it means the token is authentic. JWTKit 4 had verification for X5C headers using BoringSSL, but since it was removed a new solution had to be found. Apple provides the swift-certificates package which allows for handling of X.509 certificates and even for verification of certificate chains. Exactly that API was used to rewrite the X5C verifier. It is a modern API, allowing for custom policies and a straightforward syntax. The first step is to parse the `x5c` header:

```
let parser = DefaultJWTParser(jsonDecoder: jsonDecoder)
let (header, payload, _) = try parser.parse(token, as: Payload.self)

guard let x5c = header.x5c, !x5c.isEmpty else {
    throw JWTError.missingX5CHeader
}
```

and then map it to an array of certificates

```
let certificates = try certificateData.map {
    try Certificate(derEncoded: [UInt8]($0))
}
```

Finally, a `Verifier` is set up from a trusted store (which is the root certificate) which verifies every certificate on the chain, using the certificate before

```
// Setup the verifier using the predefined trusted store
var verifier = try Verifier(rootCertificates: trustedStore, policy: {
    // Use the optional user's custom policy
    try policy()
})

// Validate the leaf certificate against the trusted store
let result = await verifier.validate(
    leafCertificate: certificates[0],
    intermediates: untrustedChain
)

if case let .couldNotValidate(failures) = result {
    throw JWTError.invalidX5CChain(reason: "\(failures)")
}
```

Once the end of the code is reached, the chain will be valid. [32]

While this chapter explained all of the changes made to the package, the following, last chapter will speak about the results of this work, diving into some performance benchmarks and talking about the possible future for the JWTKit library.

5. Results

After having spoken of the many maintenance and code quality advantages of version 5, this chapter includes some benchmarks made on the library, in particular it compares version 4 with version 5. All measurements were performed on a custom built PC running Linux, all virtualised in a Docker environment, specifically with the following Dockerfile:

```
# Use the official Swift image from Docker Hub
FROM swift:5.10-jammy

# Set the working directory inside the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . .

# Download Swift package dependencies
RUN swift package resolve

# Command to run the application
CMD ["swift", "package", "--allow-writing-to-package-directory",
↪  "benchmark", "--format", "jmh"]
```

For benchmarking, the package-benchmark library was used [38]. The benchmarks test signing, verification and token lifecycle, which includes key creation, signing and verification, using different algorithms and calculating different parameters:

- throughput: number of operations per second;
- `mallocs`: number of memory allocations, indicates effective memory usage.

HMAC was left out because version 5 uses the same implementation as version 4, therefore the comparison is not really useful. PSS was also not considered as it was not present in version 4. Additionally, total lines of code were also measured. Obviously the code is built using release mode, which optimises performance at the expense of build time and binary size.

5.1 Signing

This section shows the signing benchmarks is done. For each test, a key is constructed once, and then used to sign a token multiple times.

Version 4

Version 4 was tested using this code:

```
Benchmark("ES256") { benchmark in
    for _ in benchmark.scaledIterations {
        let signer = try JWTSigner.es256(key: .generate())
        let token = try await signer.sign(payload)
        _ = try await signer.verify(token, as: Payload.self)
    }
}
```

Version 5

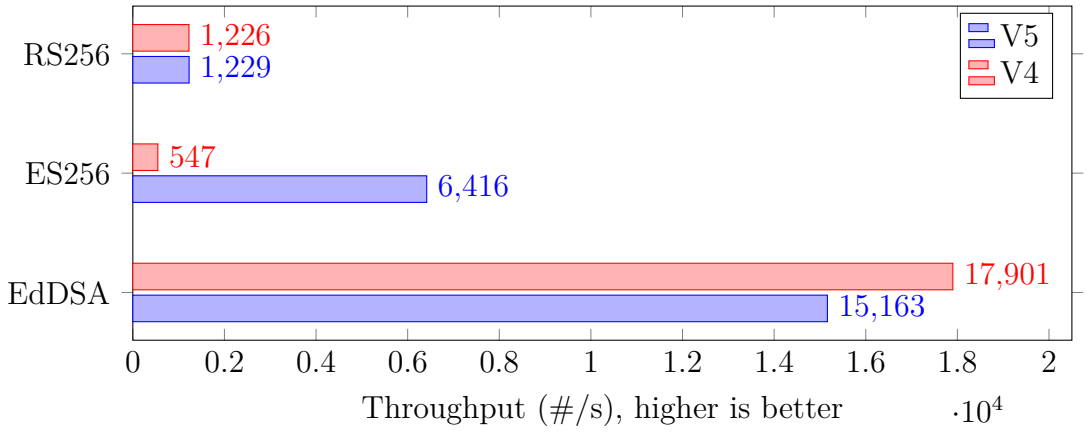
This is the code that was executed using version 5, is equivalent to version 4 from above:

```
Benchmark("ES256") { benchmark in
    let key = ES256PrivateKey()
    let keyCollection = await JWTKeyCollection().addES256(key: key)
    for _ in benchmark.scaledIterations {
        _ = try await keyCollection.sign(payload)
    }
}
```

Results

Following are the results.

Performance

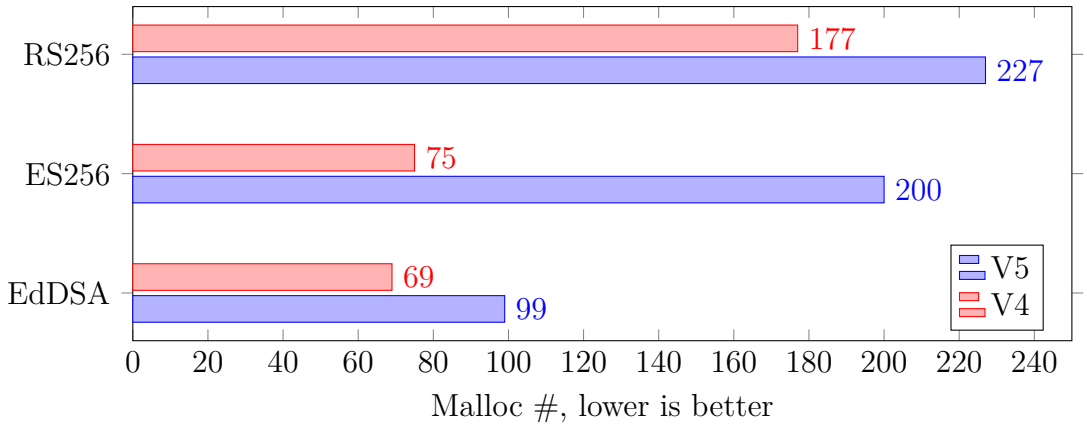


This chart shows the total number of operations performed per second. From the chart, the first noticeable thing is how slow both versions of the RS256 algorithm are compared to the elliptic curves ones. A significant improvement was made in signing using the ES256 algorithm and finally a noticeable regression in signing through EdDSA.

Algorithm	Version 4	Version 5	Change
RS256	1229	1226	+0.24%
ES256	6416	547	+1072.94%
EdDSA	15163	17901	-15.3%

Table 5.1: Signing throughput (#/s) difference

Memory Management



This graph shows the total number of memory allocations. Just as before, RSA is out-

performed by the elliptic curve algorithms, though not by much if inspecting the ES256 algorithm. EdDSA is the clear winner, though all three algorithms saw a regression in version 5. This is likely due to the usage of Swift types to get to C rather than directly accessing C constructs.

Algorithm	Version 4	Version 5	Change
RS256	177	227	+26.28%
ES256	75	200	+166.67%
EdDSA	69	99	+47.76%

Table 5.2: Signing `malloc` (#) difference

5.2 Verification

Version 5 saw a big drop in signature verification efficiency.

Version 4

This is the code used to verify a token using JWTKit 4:

```
Benchmark("ES256") { benchmark in
    let pem = """
    -----BEGIN PUBLIC KEY-----
    MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEEVs/o5+uQbTjL3chynL4wXgUg2R9
    q9UU8I5mEovUf86QZ7kOBIjJwqnzD1omageEHWwHdBO6B+dFabmdT9POxg==
    -----END PUBLIC KEY-----
    """
    let token = "eyJhbGciOiJFUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJuYW1lIjoiSm9obiBEb2UiLCJhZG1pbiI6dHJ1ZX0.
        vY4BbTLWcVbA4sS_EnaSV-exTZT3mRpH6JNc5C7XiUDA1PfbTO6LdO
        bMFYPEcKZMydfHy6SJz1eJySq2uYBLAA"
    let signer = try JWTSigner.es256(key: .public(pem: pem))
    for _ in benchmark.scaledIterations {
        _ = try signer.verify(token, as: Payload.self)
    }
}
```

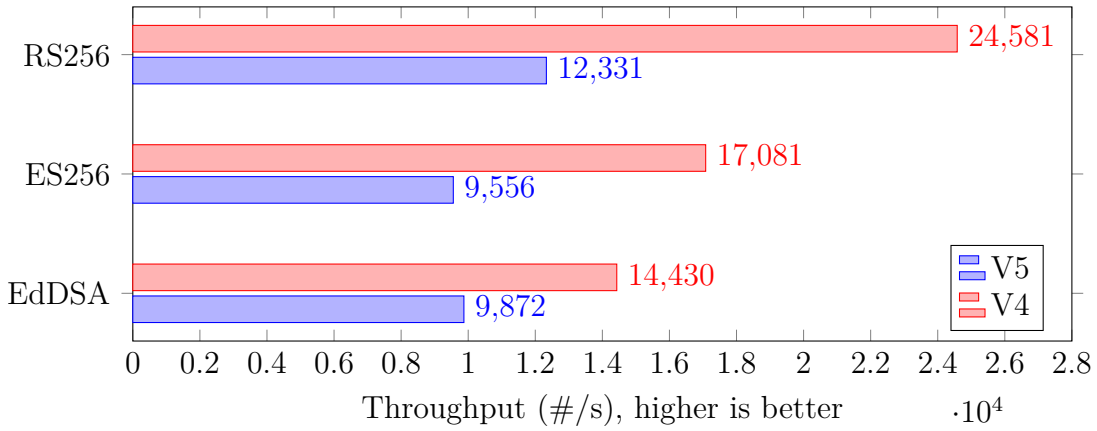
Version 5

This on the other hand is what was used in version 5:

```
Benchmark("ES256") { benchmark in
    let pem = """
    -----BEGIN PUBLIC KEY-----
    MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEEVs/o5+uQbTjL3chynL4wXgUg2R9
    q9UU8I5mEovUf86QZ7kOBIjJwqnzD1omageEHWwHdBO6B+dFabmdT9POxg==
    -----END PUBLIC KEY-----
    """
    let token = "eyJhbGciOiJFUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJuYW1lIjoiSm9obiBEb2UiLCJhZG1pbiI6dHJ1ZX0.
    vY4BbTLWcVbA4sS_EnaSV-exTZT3mRpH6JNc5C7XiUDA1PfbTO
    6LdObMFYPEcKZMydfHy6SJz1eJySq2uYBLAA"
    let key = try ES256PublicKey(pem: pem)
    let keyCollection = await JWTKeyCollection().addES256(key: key)
    for _ in benchmark.scaledIterations {
        _ = try await keyCollection.verify(token, as: Payload.self)
    }
}
```

Results

Performance

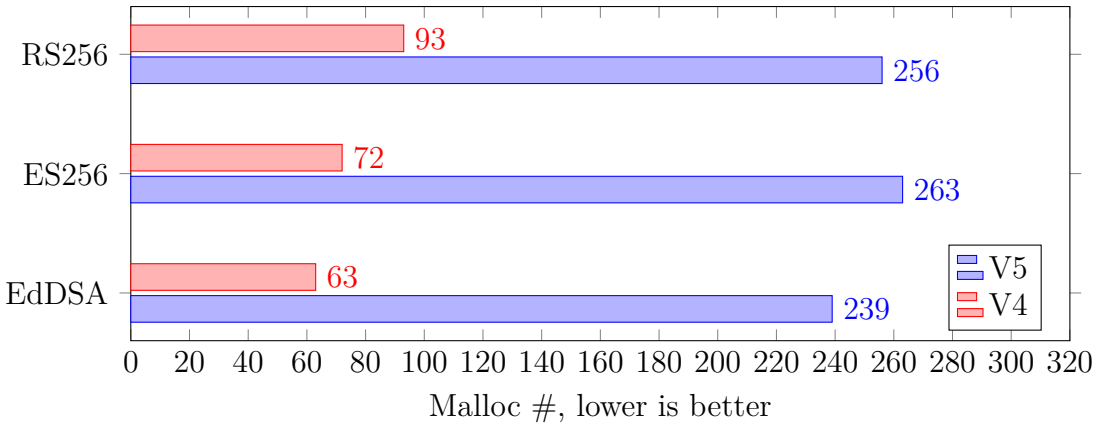


The chart shows a significant drop in throughput when using version 5. The RS256 algorithm sees its throughput halved in version 5. While the EC algorithms are not as drastic, the decrease is still noticeable and is likely due to the addition of code layers.

Algorithm	Version 4	Version 5	Change
RS256	24,581	12,331	-99.34%
ES256	17,081	9,556	-78.75%
EdDSA	14,430	9,872	-46.17%

Table 5.3: Verification `throughput` (#/s) difference

Memory Management



The chart above shows memory allocations. It shows a clear increase in memory usage from all three algorithms, and as before it is likely due to the fact that the new version uses safer and more abstract that increase memory footprint but also improve maintainability and security.

Algorithm	Version 4	Version 5	Change
RS256	93	256	+175.27%
ES256	72	263	+265.28%
EdDSA	63	238	+279.365%

Table 5.4: Verification `malloc` (#) difference

5.3 Token Lifecycle

Finally, this section sheds light on a more real life usage of the tokens, which includes both token signing and verification and also key creation. Since keys can be created different ways, it's correct for these ways to be differently measured, therefore the following graphs

will be slightly more complex. Following are code examples which showcase how data was gathered.

Version 4

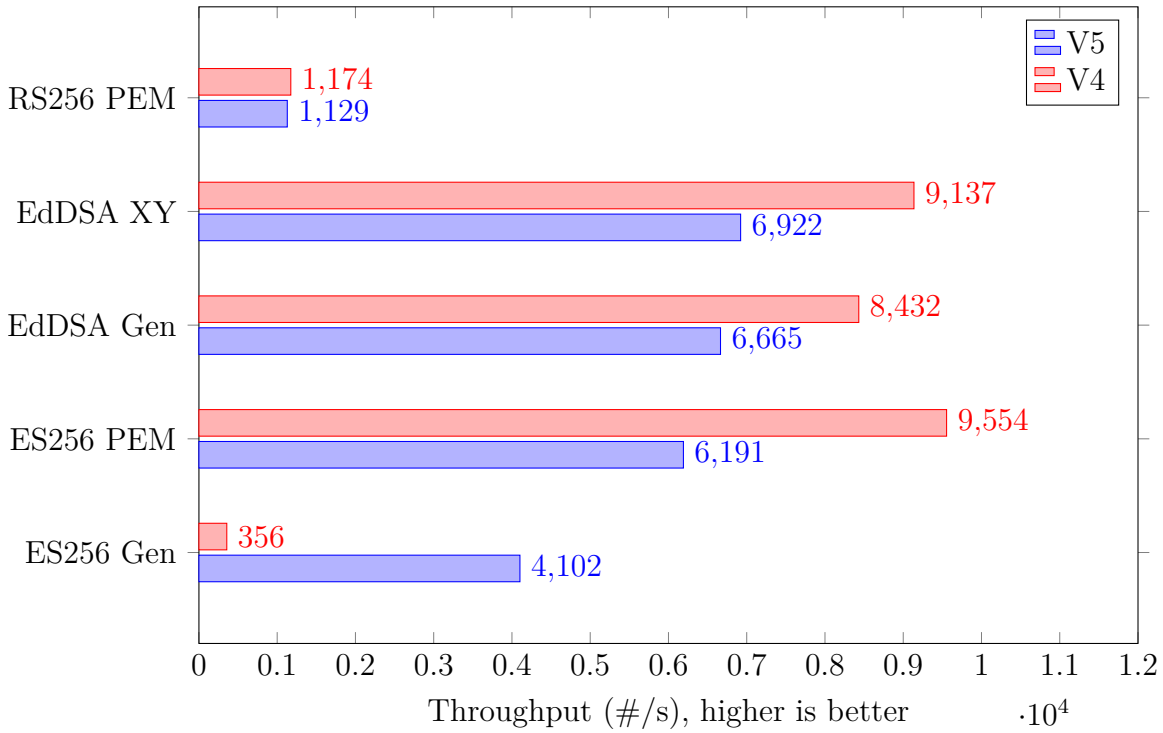
```
Benchmark("ES256 Generated") { benchmark in
    for _ in benchmark.scaledIterations {
        let signer = try JWTSigner.es256(key: .generate())
        let token = try await signer.sign(payload)
        _ = try await signer.verify(token, as: Payload.self)
    }
}
```

Version 5

```
Benchmark("ES256 Generated") { benchmark in
    for _ in benchmark.scaledIterations {
        let key = ES256PrivateKey()
        let keyCollection = await JWTKeyCollection().addES256(key: key)
        let token = try await keyCollection.sign(payload)
        _ = try await keyCollection.verify(token, as: Payload.self)
    }
}
```

Results

Performance

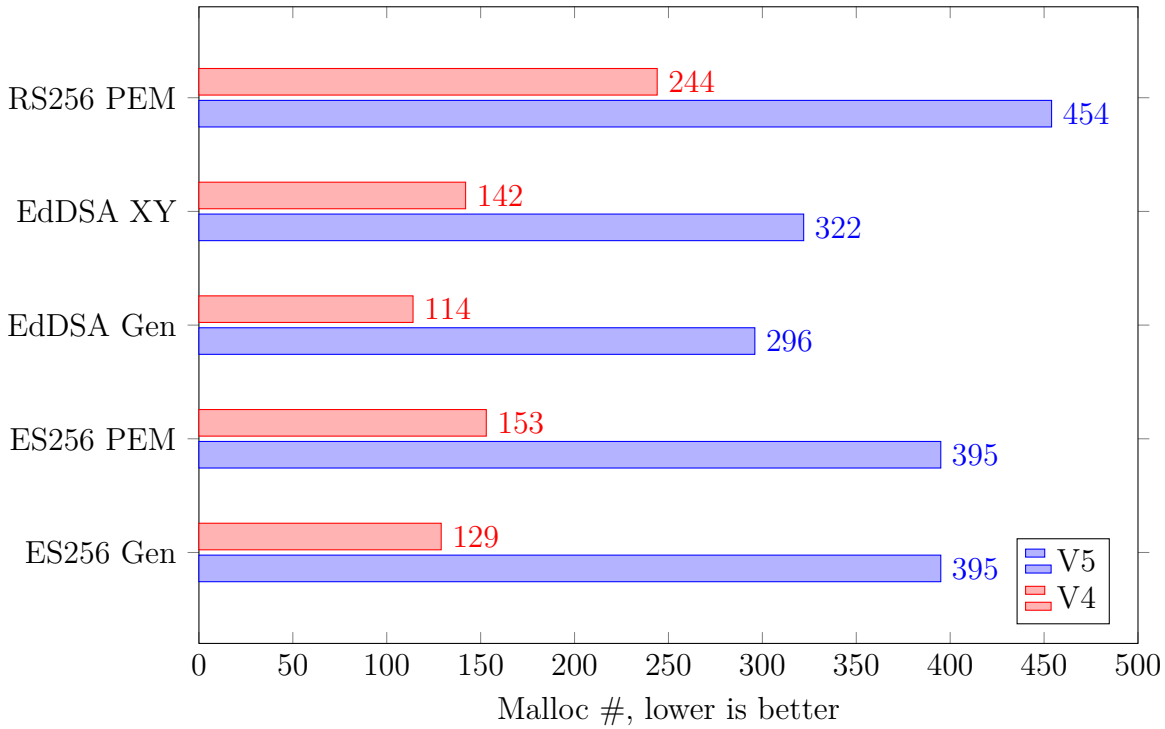


There is a substantial decrease in performance in the elliptic curves algorithms, except in the generated version of ES256, which doesn't use a PEM file but instead creates a key on the fly. RS256 performance on the other hand remains constant and, as usual, behind the rest.

Algorithm	Version 4	Version 5	Change
RS256	1,174	1,129	-3,83%
EdDSA XY	9,137	6,922	-24.24%
EdDSA Gen	8,432	6,665	-20.96%
ES256 PEM	9,554	6,191	-35.2%
ES256 Gen	356	4,102	+1052.25%

Table 5.5: Token lifecycle throughput (#/s) difference

Memory Management



Finally, this chart shows the total memory allocations which happened during the tests. This also shows a demerit for version 5, as there's much more memory usage with SwiftCrypto. This is likely due to the added complexity and abstraction of the code.

Algorithm	Version 4	Version 5	Change
RS256	244	454	+86.07%
EdDSA XY	142	322	+126.76%
EdDSA Gen	114	296	+159.65%
ES256 PEM	153	395	+158.17%
ES256 Gen	129	395	+206.20%

Table 5.6: Token lifecycle `malloc` (#) difference

5.4 Lines of Code

While benchmarks indicate a big gap in performance between the two versions, this will likely change with the passing of time and the gradual updating of the package, the beginning of which can be seen in Swift 5.10, which brings new features (such as `~Copyable`)

that can enhance performance. One change that can be seen immediately on the other hand is the size of the package, which decreased considerably in version 5. In particular, using the cloc package [39], the results were the following:

Version 4	Version 5	Change
311897	3160	-98.99%

Table 5.7: Lines of code (#) difference

This table indicates the total number of code lines in the project, before and after the transition to version 5. The difference is massive as the whole vendored version of BoringSSL is gone.

5.5 End result

While the benchmark results do not look good for version 5 of JWTKit, this is an acceptable result as code quality is greatly increased and with the latest Swift versions concurrency is ensured. A good example of code quality improvement is the drastic decrease in lines of code, and therefore quantity of code that has to be maintained. This also allows easier development and thus performance can be improved more easily, without having to touch low level C code.

Conclusion

This thesis has provided an examination of the Swift programming language, focusing on its use on the server and therefore listing numerous features on why the language is an optimal fit for said environment. In particular Swift's type safety, automatic memory management and modern concurrency model all contribute to it being the perfect fit for the server. Furthermore, the thesis discussed how the presence of the SwiftNIO library promotes the creation of server side frameworks such as Vapor and why this framework is a good choice for creating server side applications. Then, the intricacies of JSON Web Tokens were explored, with an overview of their structure and how their cryptography functions under the hood. Afterwards, the merge of these two technologies in the JWTKit package was discussed and how JWTKit provides great support for using JWTs in Swift based applications. Then, the transition to the last version of the package was evaluated. This transition was long overdue as the presence of C in a Swift library, despite Swift's C interoperability, is obsolete. Therefore, BoringSSL was removed from the library, and along with this the library underwent a number of changes, such as the adoption of Swift 5.5's concurrency model, `Sendable` conformance, new customisation APIs and more. Manually implementing functionalities which were available in BoringSSL but not in SwiftCrypto such as the raw RSA initialisation from modulus, exponent and private exponent was likely the most challenging part. This evolution brought more safety to the package, easier maintainability and general stability. Finally, the new version's performance was evaluated against version 4 and, with not much surprise, performance worsened quite a bit during the transition, likely due to added complexity and abstraction over C constructs. While this does not represent good news for the package, it also offers opportunities for further optimisation and innovation, and thanks to the eradication of C code and the adoption of modern Swift patterns, contribution from the Swift community is promoted and, more than ever, likely.

A. Appendixes

A.1 RSA Prime Factors Calculation

What follows is the algorithm used to calculate p and q when creating an RSA keys from n, e and d.

```
struct PrimeGenerator {
    static func calculatePrimeFactors(
        n: BigUInt, e: BigUInt, d: BigUInt
    ) throws -> (p: BigUInt, q: BigUInt) {
        let k = (d * e) - 1

        guard k & 1 == 0 else {
            throw RSAError.keyInitializationFailure
        }

        let t = k.trailingZeroBitCount, r = k >> t

        var y: BigUInt = 0
        var i = 1

        // If the prime factors are not revealed after 100 iterations,
        // then the probability is overwhelming that the modulus is not
        // the product of two prime factors, or that the public
        // and private exponents are not consistent with each other.
        while i <= 100 {
            let g = BigUInt.randomInteger(lessThan: n - 1)
            y = g.power(r, modulus: n)

            guard y != 1, y != n - 1 else {
                continue
            }

            var j = 1
            var x: BigUInt
```

```
            while j <= t &- 1 {
                x = y.power(2, modulus: n)

                guard x != 1, x != n - 1 else {
                    break
                }

                y = x
                j &+= 1
            }

            x = y.power(2, modulus: n)
            if x == 1 {
                let p = (y - 1).greatestCommonDivisor(with: n)
                let q = n / p

                return (p, q)
            }
            i &+= 1
        }

        throw RSAError.keyInitializationFailure
    }
}
```

Acronyms

ASN.1 Abstract Syntax Notation One. 53, *Glossary:* Abstract Syntax Notation One

CA Certificate Authority. 28

GUI Graphical User Interface. 15

I/O Input-Output. 15, 18

JOSE JavaScript Object Signing and Encryption. 57, *Glossary:* JavaScript Object Signing and Encryption

JWA JSON Web Algorithms. 32

JWS JSON Web Signature. 31, 33, 36, 37

JWT JSON Web Token. 27, 29–32, 38, 41

MAC Message Authentication Code. 32

NIST National Institute of Standards and Technology. 34, 36

PEM Privacy Enhanced Mail. 42, 45, 46, 70, *Glossary:* Privacy Enhanced Mail

PSS Probabilistic Signature Scheme. 36

RSA Rivest-Shamir-Adleman. 36

SSS Server Side Swift. 52

Glossary

Abstract Syntax Notation One standard interface description language for defining data structures that can be serialised and deserialised in a cross-platform way. It is widely used in telecommunications and computer networking, and extensively in cryptography, to specify the format of data being sent across networks or stored. 53

claim A piece of information asserted about a subject. A claim is represented as a name/value pair consisting of a Claim Name and a Claim Value [13]. 28

concurrency domain The scope in which mutable state resides and where concurrent operations occur on said state. Operations inside of a concurrency domain are usually safe. 13, 55

data manipulation language Subset of SQL (Structured Query Language) that is used for accessing and manipulating data in a database. 20

data race A specific race condition in which at least one of the tasks accessing the data needs to write on it. 11, 14

deadlock Standstill situation of two tasks not being able to continue their execution as both are waiting on each other to finish and free up a resource the other one need. 10

encoding process of converting data from one form into another according to a set of rules or a specific format. This is often done to prepare data for transfer, storage, or compatibility purposes across different systems. 27

JavaScript Object Signing and Encryption set of specifications designed to secure data through JSON-based mechanisms. It groups many standards under one umbrella, such as JWT, JWS and JWA . 57

lock synchronisation mechanism used in concurrent programming to prevent multiple threads from accessing a shared resource or a critical section of code at the same time. When a thread acquires a lock, it gains exclusive access to the protected resource. Other threads attempting to acquire the same lock are blocked until the lock is released by the holding thread. 56

marker protocol A protocol not used to implement some specific behaviour but rather to indicate that a type conforms to some rule or characteristic. 13

non-singular type of curve in algebraic geometry that does not have any singular points. A singular point on a

curve is a point where the curve either crosses itself, has a cusp (a point where the curve has a sharp turn), or the derivative (slope) of the curve does not exist. 33

Privacy Enhanced Mail base64 format for storing and transmitting cryptographic keys, certificates, and other sensitive data . 42

property wrapper Accessor type which can be added to a property to add additional behaviour to their getter and setter. 21

race condition Condition in which the outcome of a task is not deterministic because two tasks are accessing a resource at the same time and the output can change depending on which task finishes first. 10, 12, 13, 78

salt random value sometimes added to messages before they get processed by a cryptographic function. This is done to promote uniqueness of the resulting output. 37

socket descriptor Integer that uniquely identifies a socket. It's used to manage and reference open sockets for various network operations. 16

Unix time Number of seconds elapsed since the Unix epoch, which is 00:00:00 UTC on 1 January 1970. This representation was born as the Unix OS default system time and is widely used since. 29, 38

X.509 standard defining the format of public key certificates. An X.509 certificate contains information such as the certificate holder's name, the certificate's serial number, expiration dates, the public key, and the digital signature of the certificate authority (CA) that issued the certificate, thereby ensuring the authenticity of the public key for secure communications. 28, 61

Bibliography

[1] Apple Inc. *The Swift Programming Language (Swift 5.7)*. Apple Inc., 2014.

[2] Apple Inc. The Swift Programming Language (5.10 beta). https://docs.swift.org/swift-book/documentation/the-swift-programming-language/automaticreferencecounting/. Accessed: 14 Feb 2024.

[3] Jon Hoffman. *Swift Protocol-Oriented Programming: Increase productivity and build faster applications with Swift 5, 4th Edition*. Packt Publishing, 2019.

[4] John McCall and Doug Gregor. Async/await. https://github.com/apple/swift-evolution/blob/main/proposals/0296-async-await.md, 2022. Accessed: 17 Feb 2024.

[5] John McCall, Joe Groff, Doug Gregor, and Konrad Malawski. Structured concurrency. https://github.com/apple/swift-evolution/blob/main/proposals/0296-async-await.md, 2022. Accessed: 17 Feb 2024.

[6] Chris Lattner and Doug Gregor. Sendable and @Sendable closures. https://github.com/apple/swift-evolution/blob/main/proposals/0302-concurrent-value-and-concurrent-closures.md, 2022. Accessed: 19 Feb 2024.

[7] Apple Inc. Open Source at Apple. https://opensource.apple.com.

[8] Elliotte Rusty Harold. *Java Network Programming, 3rd Edition*. O'Reilly Media, 2004.

[9] Apple Inc. SwiftNIO Documentation. https://swiftpackageindex.com/apple/swift-nio/2.63.0/documentation/nio.

[10] Ralph Kuepper and Tanner Nelson. *Hands-On Swift 5 Microservices Development*. Packt Publishing, 2020.

[11] Tim Condon, Gwynne Rasking, et al. Vapor Documentation. https://docs.vapor.codes/. Accessed: 21 Feb 2024.

[12] Nigel Chapman and Jenny Chapman. *Authentication and Authorization on the Web*. Web Security Topics. MacAvon Media, 2012.

[13] Michael B. Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). RFC 7519, May 2015.

[14] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017.

[15] Simon Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648, October 2006.

[16] Michael B. Jones, John Bradley, and Nat Sakimura. JSON Web Signature (JWS). RFC 7515, May 2015.

[17] Sebastián E. Peyrott. *The JWT Handbook*. Auth0 Inc., 2016-2018.

[18] S. Marshall and C. Allen. *Digital Signatures for Dummies, Cryptomathic Special Edition (Custom)*. John Wiley & Sons, Incorporated, 2018.

[19] Helena Handschuh. *SHA Family (Secure Hash Algorithm)*, pages 565–567. Springer US, Boston, MA, 2005.

[20] Michael B. Jones. JSON Web Algorithms (JWA). RFC 7518, May 2015.

[21] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.

[22] Don Johnson; Alfred Menezes; Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International Journal of Information Security 2001-aug vol. 1 iss. 1*, 1, August 2001.

[23] Thomas Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979, August 2013.

[24] Lawrence C. Washington. *Elliptic curves: number theory and cryptography*. Discrete Mathematics and Its Applications. Chapman and Hall/CRC, 2 edition, 2008.

[25] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards Curves. In Serge Vaudenay, editor, *Progress in Cryptology – AFRICACRYPT 2008*, pages 389–405, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[26] Svetlin Nakov. *Practical Cryptography for Developers*. 2019.

[27] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017.

[28] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, November 2016.

[29] Steve Burnett; Stephen Paine; RSA Security. *RSA Security's official guide to cryptography*. Osborne/McGraw-Hill, 2001.

[30] Henk C. A. Tilborg, editor. *Encyclopedia of Cryptography and Security*. Springer New York, NY, 2005.

[31] Tanner Nelson, Scott Grosch, and Tim Condon. JWTKit v4, 2020.

[32] Paul Toffoloni, Tim Condon, and Gwynne Raskind. JWTKit, 2024.

[33] Google LLC. BoringSSL, 2023.

[34] National Institute of Standards and Technology (NIST). Recommendation for Pair-Wise Key Establishment Schemes Using Integer Factorization Cryptography (Revision 2). NIST Special Publication 800-56B Rev. 2, National Institute of Standards and Technology, mar 2019.

[35] Desi Wulansari, Much Aziz Muslim, and Endang Sugiharti. Implementation of RSA Algorithm with Chinese Remainder Theorem for Modulus N 1024 Bit and 4096 Bit. *International Journal of Computer Science and Security (IJCSS)*, 2016.

[36] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, November 2016.

[37] Chris Lattner. Introduce User-defined "Dynamic Member Lookup" Types. `https://github.com/apple/swift-evolution/blob/main/proposals/0195-dynamic-member-lookup.md`, 2021. Accessed: 5 Mar 2024.

[38] Joakim Hassila. package-benchmark, 2022.

[39] Al Danial. cloc, 2006.