POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Development of a Secure and Licensed Mobile Framework for Real-Time Physiological Data Analysis

Supervisors

Prof. MASSIMO VIOLANTE

Candidate

AMIR HOSSEIN RAHMANZADEH

JULY 2025

Abstract

For companies active in the domain of developing data-driven and proprietary algorithms, a big challenge is always how to share or sell their solution to other companies without exposing the internal logic. This problem is more highlighted, especially in the use cases where the data processing should be done on the edge and in real time, even in remote areas, eliminating the possibility of using a server-side logic. The solution presented in this work provides a reliable way to securely share the physiological prediction algorithms for third-party mobile applications, in the domain of sleep, fatigue, and Alcohol misuse. It consists of a Swift XCFramework for iOS and an Android AAR library, each providing a clear and consistent public interface while keeping the logic secure and unreachable. The solution works independently of the source of health data and provides the result in a defined manner without exposing the core logic. To enhance the security and prevent misuse of the library or framework, a comprehensive offline licensing system is implemented to generate licenses to be shared with clients. Each license indicates a list of features available in it, the bundle ID or package name of the app it can run on, and the expiry date. To further increase the security of the solution, security checks for jailbroken or rooted devices are added, in addition to the detection of reverse engineering attempts.

Table of Contents

List of Figures v

Acronyms viii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 2

2 Background 3
- 2.1 Cloud, Fog, Edge Computing and On-Device Processing . . . 3
  - 2.1.1 Cloud Computing in Mobile Health . . . 3
  - 2.1.2 Fog Computing in Mobile Health . . . 4
  - 2.1.3 Edge Computing in Mobile Health . . . 5
  - 2.1.4 On-Device Computing in Mobile Health . . . 6
  - 2.1.5 Choice justification . . . 7
- 2.2 IP Protection and Licensing Methods . . . 8
  - 2.2.1 Hardcoded License Keys . . . 9
  - 2.2.2 Online License verification . . . 9
  - 2.2.3 Token-Based License Systems (JWT and Similar Tokens) . . . 10
  - 2.2.4 Offline Licensing with Digital Signatures . . . 12
  - 2.2.5 Choice justification . . . 13
- 2.3 Licensing Strategies for On-Device Frameworks . . . 14
  - 2.3.1 Hardware/Device-Bound Licensing . . . 14
  - 2.3.2 Feature-Limited and Modular Licensing . . . 15
  - 2.3.3 Time Limited Licensing . . . 16
  - 2.3.4 Floating (Concurrent) Licensing . . . 16
  - 2.3.5 License Servers on Device . . . 17
  - 2.3.6 Choice Justification . . . 17
- 2.4 Security Risks in Mobile License Enforcement . . . 18
  - 2.4.1 Reverse Engineering and Patching . . . 18
  - 2.4.2 Key Leakage . . . 18

2.4.3 Network Attacks . . . 19
2.4.4 Denial-of-Service or Bricking Risks . . . 19

3 System Overview 21
3.1 System Objectives and Use Context . . . 21
3.2 SDK Architecture (iOS and Android) . . . 22
3.2.1 iOS: XCFramework Packaging and Architecture . . . 22
3.2.2 Android: AAR Packaging, Native Code, and Architecture . . . 23
3.2.3 Cross-Platform Architectural Principles . . . 24

4 Framework and Library Implementation 26
4.1 Overview of SDK Structure . . . 26
4.2 iOS XCFramework . . . 28
4.2.1 Internal Modules (PredictS, License, Security, Features) . . . 28
4.2.2 Building and Integration in Customer's App . . . 33
4.2.3 What third-party developers have access to . . . 34
4.3 Android AAR Library . . . 35
4.3.1 Architecture and Components of the Android SDK . . . 35
4.3.2 Kotlin Layer . . . 36
4.3.3 C++ Native Layer . . . 36
4.3.4 Build Configuration: CMake and Gradle Integration for Native Libraries . . . 38
4.3.5 Integration in Customer's App . . . 39

5 Licensing System 41
5.1 Design Goals and Constraints . . . 41
5.2 License Creation and Verification Flow . . . 42
5.2.1 Cryptographic Primitives . . . 42
5.2.2 License Signing Workflow . . . 44
5.2.3 Offline Client Integration . . . 45

6 Conclusion and Future Work 47
6.1 Achievements . . . 47
6.2 Limitations . . . 48
6.3 Future Work . . . 48

Bibliography 50

List of Figures

2.1 High-level view of JWT flow 11

4.1 High-level Flowchart of system 27

Listings

4.1 init function of PredictS class 28
4.2 Drowsiness state detection function in PredictS file 29
4.3 Jailbreak check logic 30
4.4 Public Interface of Fatigue Prediction 32
4.5 Commands for distributing .xcframework 33
4.6 How developers can use the framework 34
4.7 Public interface shared with third-party developers 34
4.8 Kotlin interface of License checker 36
4.9 Native root detection code 37
4.10 Sample initializing of the library 39
5.1 License Parameters 45
5.2 License Sample 45

Acronyms

AAR
Android Archive

ABI
Application Binary Interface

API
Application Programming Interface

ECDSA
Elliptic Curve Digital Signature Algorithm

HMAC
Hash-based Message Authentication Code

HTTPS
Hypertext Transfer Protocol Secure

JNI
Java Native Interface

JSON
JavaScript Object Notation

JWT
JSON Web Token

MITM
Man-in-the-Middle

NDK
: Native Development Kit

OS
: Operating System

RSA
: Rivest–Shamir–Adleman

SDK
: Software Development Kit

TEE
: Trusted Execution Environment

TLS
: Transport Layer Security

XCFramework
: Xcode Framework (multi-architecture binary format)

SHA-256
: Secure Hash Algorithm 256-bit

SHA-2
: Secure Hash Algorithm 2

Base64
: Binary-to-text encoding scheme

RSASSA-PSS
: RSA Signature Scheme with Appendix – Probabilistic Signature Scheme

PKCS#1
: Public-Key Cryptography Standard #1

SHA-256
: Secure Hash Algorithm 256-bit

SHA-2
: Secure Hash Algorithm 2

Chapter 1

Introduction

1.1 Motivation

In recent years, wearable devices and mobile health (mHealth) technologies have seen remarkable growth, reshaping how physiological data is gathered and analyzed in real time. The global wearable tech market was valued at around USD 84 billion in 2024, and is projected to more than double by 2030, with an average annual growth rate of over 13%.[1] Even more impressive is the surge in medical-grade wearables, a category expected to grow from USD 91 billion in 2024 to more than USD 320 billion by 2032, driven by increasing demand for continuous monitoring and preventative care.[2]

The capabilities of these devices are no longer limited to heart rate monitoring, they now routinely track blood oxygen levels, sleep patterns, stress markers, and other vital health metrics. What powers this shift behind the scenes are complex machine learning and data processing algorithms capable of interpreting large streams of physiological data and extracting meaningful insights from them. But, building these algorithms is no small task. It requires time, domain expertise, carefully curated data sources gathered from reliable sources, and rigorous validation, often in clinical or field settings. The financial and technical investment involved makes these models extremely valuable assets the kind of intellectual property (IP) that companies are understandably eager to protect.And that's where a key challenge emerges. To be useful, these algorithms often need to be integrated into third-party apps. But distributing them in a way that preserves their value, while also allowing them to run on a user's device, is far from straightforward. This is especially true in environments where constant internet access can't be guaranteed, like in remote clinics, mobile units, or field applications, making on-device, offline execution essential.

Cloud-based deployments offer some protection, since the logic never leaves the server. But that model breaks down in low-connectivity contexts and can raise privacy or latency concerns. Ultimately, developers face a difficult trade-off: how to allow broader use of their algorithms without exposing them to reverse engineering, misuse, or unauthorized redistribution.

1.2 Objectives

The goal of this project is to create a secure and practical way to run proprietary physiological data analysis algorithms on mobile devices without putting the underlying logic at risk and without relying on a constant internet connection.

To achieve this goal, this thesis focuses on developing a separate mobile framework for iOS and a Library for Android, letting third-party companies and developers integrate the logic in their apps while keeping the core logic secure and safe.
More specifically, the project sets out to:

- Build a shared architecture with two components: a Swift XCFramework for iOS and an AAR library for Android, both exposing a simple and consistent interface for sending data and receiving results.
- Keep the system platform-agnostic by accepting health data from any source as long as it follows the expected format, whether it's Garmin, Apple Health, or something else.
- Protect the algorithm's logic through binary obfuscation, compiled delivery, and built-in checks that make reverse engineering significantly harder.
- Design and implement a license system that works entirely offline, where each license defines which features are enabled, which app it can run on, and how long it's valid.
- Implement various security checks at runtime to detect rooted Android or Jailbroken iOS devices, which are typically used for bypassing their OS standard security standards.
- Test the system thoroughly to make sure everything works as expected: from license validation to handling edge cases and security triggers.

Together, these objectives aim to provide a developer-friendly but secure way to deliver sensitive algorithmic functionality to mobile apps without giving up control over how and where it's used.

Chapter 2

Background

2.1 Cloud, Fog, Edge Computing and On-Device Processing

Mobile health (mHealth) applications increasingly rely on the continuous monitoring and real-time processing of physiological signals, such as heart rate, blood pressure, and electrocardiogram (ECG) data, captured through wearable sensors and smartphones. In such contexts, selecting an appropriate computational architecture is crucial to ensuring both performance and security. Four main models are typically employed: cloud computing, fog computing, edge computing, and fully on-device computation. Each paradigm presents a unique balance of factors, including latency, offline operability, and the degree of data privacy, considerations that carry particular weight in healthcare-related use cases.

2.1.1 Cloud Computing in Mobile Health

Cloud computing consists of delegating data storage and processing tasks to remote, centralized servers that are accessible over the internet. In the scope of mHealth applications, cloud-based platforms are often used to collect and analyze data from multiple users, enabling large-scale analytics and the development of heavy and expensive resource-intensive tasks. One of its key advantages is its virtually unlimited scalability and flexibility, making it ideal for computationally demanding operations.[3] Leading providers such as Amazon Web Services and Microsoft Azure support easy scaling solutions and "pay as you go" plans reducing the need for initial heavy investments both in terms of cost and time.[4]

Despite its scalability and processing power, cloud computing presents notable limitations for real-time mobile health applications, most critically, in terms of

latency. Since data must be transmitted from the user's device to remote data centers and then returned, inherent network delays are introduced. Empirical studies report that cloud-based processing typically incurs latencies in the range of 20 to 40 milliseconds, which may be insufficiently responsive for time-sensitive scenarios.[5] Indeed, several researchers emphasize that fully cloud-reliant architectures are "not suitable for real-time applications" due to these latency concerns.[6] In clinical situations such as detecting arrhythmias or responding to falling sleep, especially in dangerous scenarios such as driving, even minor delays can have serious consequences. Furthermore, the dependency on continuous internet connectivity renders cloud-based systems ineffective in offline or low-connectivity environments such as rural areas or during emergency events, where uninterrupted health monitoring is most needed. This lack of offline operability significantly undermines the reliability and ubiquity required for continuous physiological assessment.

Another issue with cloud-based mHealth systems is data privacy. Potential weaknesses in data security and regulatory compliance arise when extremely sensitive personal health data, like activity histories or heart rate patterns, is transmitted to distant servers. Users of these systems must have a great deal of faith in outside providers to protect their medical information. The act of moving data outside of the user's direct control is still a major concern, even though encryption protocols and adherence to laws like HIPAA can reduce some risks. Attempts to guarantee strong privacy protection are complicated by this trade-off, which involves outsourcing computation at the expense of data being sent to the cloud. In short, cloud computing has many benefits, such as centralized analytics and large-scale processing, but it also has some serious disadvantages, such as the need for reliable network connections, latency problems that prevent real-time feedback, and a greater risk of personal health data exposure.

2.1.2 Fog Computing in Mobile Health

Fog computing is built upon the traditional cloud model by introducing a distributed layer of intermediary nodes (commonly referred to as fog nodes or fog servers) which are closer to the data-generating devices.[6] These nodes are typically located within local or regional networks, allowing data to be processed closer to its source.
Handling computation at such closer distance, helps reducing the latency and the network bandwidth usage significantly. This architectural approach enhances the system's capacity for real-time responsiveness, particularly in scenarios where rapid feedback is essential. According to industry analyses, fog computing offers many of the low-latency advantages available in edge computing, while having the flexibility to send non-urgent processing tasks to the cloud. This hybrid capability makes

it especially valuable for applications that require both immediacy and scalability.[4]

One key advantage of fog computing is its capability to keep a part of its functionality even in the event of an internet outage. As long as the local fog node remains accessible, typically through Wi-Fi or a local network, data can still be processed and stored locally. This capability it possible for vital services, like alert generation and patient or driver conciseness monitoring, to function even when there isn't a reliable cloud connection.

By localizing computation inside a limited organizational or geographic boundary, fog computing presents clear benefits in terms of data privacy. Unlike public cloud solutions, fog architectures let sensitive health data be analyzed and pre-processed near its source without instantaneous transmission to outside servers. Since raw personal data can stay on the actual premises or institutional network, this proximity helps to improve data protection rules compliance. However, the introduction of fog nodes also expands the potential attack surface, given that these nodes serve as intermediaries between edge devices and the cloud. Without robust security measures, they may become susceptible to threats such as man-in-the-middle attacks.[4] Still, fog computing offers a convincing middle ground: it preserves access to cloud infrastructure for high-complexity chores or long-term storage while delivering lower latency and enhanced privacy than cloud-only systems. In healthcare IoT systems, where both strict data governance and real-time responsiveness are critical, this hybrid approach is especially suited.

2.1.3 Edge Computing in Mobile Health

Edge computing advances the distribution of computation by locating processing activities as near as possible to the source of data, usually at the boundary of the network or on the gateway devices that are in direct connection with sensor devices. In this architecture, processing occurs at edge nodes that are physically close to data-gathering peripherals.[4] These nodes can include cellular base stations, local servers on the same site, or specialized processing equipment co-located with the sensor array. While on-device processing is sometimes grouped under edge computing, it is treated as a distinct category in this thesis. The central idea behind edge computing is to allow computation and information interpretation to occur on-demand locally, in or near the location where it is generated, preventing the latency involved with the usage of remote servers. The edge systems can process data at ultra-low latency by storing and processing locally, making them essential in real-time systems.[3] Experimental findings confirm the assertion that edge computing has the potential to reduce latency to less than 5 milliseconds far beyond the performance of the cloud, which records an average of 20 to 40

milliseconds. Such degree of responsiveness is particularly crucial in time-dependent mobile health situations.

Edge computing improves system resilience by the help of enabling data processing on nearby nodes or devices, which reduces dependence on internet connectivity. This makes edge computing particularly valuable in remote or low-infrastructure environments. However, when the edge node is separate from the user's device, a local connection (e.g., LAN or cellular) is still required. Even though they are not completely independent, edge systems are far more self-sufficient than cloud-based models and, when paired with on-device processing, can continue vital monitoring even when the internet is unavailable.

By allowing sensitive data to be processed close to its source, edge computing improves data privacy. Edge systems can filter and analyze data locally, sharing only aggregated insights or alerts, rather than sending raw health data to external servers. This method lessens dependency on external cloud infrastructure while maintaining control over personal health information. As a result, in addition to enhancing privacy it also lowers the bandwidth usage, which is an important benefit in healthcare settings. According to a systematic review, edge architectures are widely employed in applications that have strict privacy and real-time requirements, like medical monitoring, because they can satisfy both.[3] However, the edge nodes' limited computational capacity is a significant drawback. Edge devices, in contrast to cloud systems, are limited in how many users or tasks they can manage. Several nodes may need to be deployed in order to scale such systems. Although, complex analyses can now be conducted at the edge, however, thanks to developments in mobile hardware and AI acceleration, which provide a workable balance between privacy, scalability, and speed in mHealth applications.

2.1.4 On-Device Computing in Mobile Health

The most localized model of data-processing architectures is on-device computing, using which all computations are carried out locally on the mobile or wearable sensor that is used to capture the data, as opposed to being outsourced to remote servers. The calculations are usually being done by the internal processors of a smartphone, a smartwatch, or of a medical sensor. With recent advances in mobile chipsets, running even heavy data processing tasks locally has become increasingly feasible.[7] It is now possible to perform real-time, fully on-device tasks like fall detection, driver drowsiness prediction, and heart rhythm classification. Processing delays are usually lowered to a few milliseconds, completely eliminating network latency.

The implications of on-device computing for mobile health are particularly impactful. It offers maximum offline availability: even in areas without network coverage, users can receive health insights directly from their devices. This capability is essential for continuous monitoring and timely alerts, regardless of connectivity. From a privacy standpoint, on-device processing is also highly advantageous, as sensitive physiological data remains confined to the device, minimizing risks of exposure or misuse.[7] As one industry report observes, on-device AI "eliminates the need to transmit data to the cloud," offering stronger privacy safeguards for sensitive information such as voice, images, or biomedical signals.[7] This localized model aligns well with data protection regulations by ensuring that raw data remains under the user's direct control.

These properties make this paradigm appealing to mHealth applications which need to serve privacy-aware users and contexts with inconsistent connectivity.

2.1.5 Choice justification

Out of all the computational models that were discussed, on-device computing was found to be the most suitable for the goals of this thesis. Although cloud, fog, and edge computing architectures each have unique benefits, none of them simultaneously meet the three crucial requirements of robust data privacy, offline operability, and real-time responsiveness.

To ensure that the technical design satisfies the demanding requirements of modern mHealth applications, this thesis uses on-device computation as the fundamental architectural principle for the suggested framework and library.

2.2 IP Protection and Licensing Methods

When a company distributes proprietary algorithms through a mobile software development kit (SDK) or library, implementing a robust licensing system is essential to protect its intellectual property and sustain its business model. Unlike standalone applications, SDKs are designed to be embedded within third-party apps, meaning the vendor's proprietary logic, or a part of it, is shared with external developers. Without effective licensing, there is little to prevent different cases of misuse, such as unauthorized deployment beyond agreed terms or repackaging of the SDK into unlicensed products. Licensing acts like a safeguard, enabling the vendors to enforce legal usage, restrict redistribution, and ensure that only authorized clients have access to their product.[8] This makes the SDK a regulated, marketable product rather than a resource that can be freely exploited.[8] Additionally, it is essential for discouraging software piracy, which continues to pose a serious risk to proprietary software. In order to reduce the risk of reverse engineering and unauthorized replication, academic literature on software protection emphasizes the significance of incorporating techniques like license verification, code obfuscation, and watermarking. Since all protection systems are, in theory, vulnerable to attacks over time, rigorous implementation is essential. A robust, structured licensing framework, therefore, not only boosts commercial enforcement by maintaining compliance with license terms and conditions but also keeps the IP safe.

In summary, licensing is the foundation of secure SDK distribution, providing the ability for vendors to share high-value libraries only with known and verified end users. Beyond protecting IP, it also enables the enforcement of revenue models, such as subscriptions or usage-based fees, by forcing technical controls based on the terms of the license.

Vendors of mobile and embedded software have created a variety of licensing techniques to address the difficulties of preserving intellectual property and enforcing usage terms. The most effective solutions usually involve a combination of techniques, suitable for the specific constraints of mobile environments, such as intermittent connectivity, limited resources, and considerable vulnerability to reverse engineering. The following section outlines commonly used licensing methods for mobile libraries, highlighting their practical applications, advantages, and disadvantages.

2.2.1 Hardcoded License Keys

One of the most straightforward licensing methods involves the use of static or hardcoded license keys. A unique key is assigned to each client by the SDK vendor. This key must be provided by the third-party application integrating the SDK, usually by passing it through a runtime initialization function. The key is then validated by the SDK using a validation algorithm or by comparing it to an internally stored list of permitted values. This approach is easy to implement and works completely offline.

However, static license keys provide only a minimum level of protection. Since the validation logic is simple, an attacker, with a few valid license keys, or moderate reverse engineering skills, can find out the pattern or validation logic. Additionally, hardcoded keys are not flexible by nature. Unless further validation layers are in place, once a key is leaked, it can easily be used by unauthorized parties without any restrictions. Due to the lack of built-in revocation or usage monitoring mechanisms, vendors are unable to identify or prevent misuse after a key has been compromised.

2.2.2 Online License verification

Online licensing operates via a real-time communication between the application and a remote server to check the license at runtime. In this model, the mobile app, or the SDK embedded within it, sends a request to a licensing server, usually operated by the SDK vendor or a third-party License-as-a-Service provider. The server may respond with a confirmation of the validity of the license or particular license parameters after authenticating the request using identifiers like an API key, app package name, or unique client ID. Commercial SDKs and platforms frequently use this technique as a component of all-inclusive licensing packages.

However, there are a number of drawbacks to depending on network connectivity. Applications that need to run in offline or low-connectivity settings might not be able to run license checks, which could result in startup errors or momentary functionality loss. This is a huge problem when it comes to mHealth scenarios, especially in cases where the patient or person of interest moves to areas with weak connectivity potential, like drivers. If the license server cannot be reached, online verification could introduce latency and possible points of failure even in normal circumstances. Many systems use a hybrid approach to address these issues: offline operation is allowed for a grace period after an initial online activation. For instance, Cryptolens advises caching a digitally signed license token that permits offline use for a specified amount of time (for instance, 30 days), after which the application needs to reconnect in order to renew the license.[9]

Another important factor in online licensing is security. To avoid spoofing and unwanted access, the client-license server communication must be secured. Using API keys for client authentication, sending data over HTTPS, and using a server-side private key to sign responses are examples of some common practices to mitigate this threat. Using an embedded public key, the client can then confirm the response's legitimacy, guaranteeing that only legitimate licenses from the authorized server are accepted. In most cases, the developer registers the app and receives a license or API key, which is then stored in the app. Upon launch, the SDK sends a request to the licensing server, including identifying information such as the app's package name and device metadata. After confirming the request, the server provides a signed license token with status and usage parameters, frequently in JSON format. The license server is still the final source of truth, even if this token is saved on the device and used again for offline verification.

In conclusion, online licensing gives SDK vendors a great deal of control and flexibility, but it needs to be properly planned to handle connectivity issues and ensure secure client-server communication. It provides an efficient balance between usability, security, and enforceability in mobile software licensing when put into practice with the right safeguards and contingency plans.

2.2.3 Token-Based License Systems (JWT and Similar Tokens)

One modern technique that combines online and offline licensing models is token-based licensing. This method encapsulates the license in a digitally signed token, usually a JSON Web Token (JWT) or a similar cryptographic payload. This token includes encoded parameters that define the license terms, such as the purchaser's identity, permitted app or device, expiration date, and enabled features. Crucially, the token is signed using the vendor's private key, allowing the SDK to verify its authenticity and integrity offline using the corresponding public key. This eliminates the need for constant connectivity and allows for safe, impenetrable license validation.

At its core, token-based licensing depends on the reliability of digital signatures. Forging a legitimate token becomes computationally impossible as long as the vendor's private signing key is kept private and a secure cryptographic algorithm (such as RSA or ECDSA) is employed. This counts as a major improvement over static keys, as it removes the need to embed any secret in the client application. There is no chance of client-side secret leakage because the SDK only contains the public key, which is used to validate the signature. A high-level view of a typical token-based licensing (here JWT) is shown at Figure 2.1.

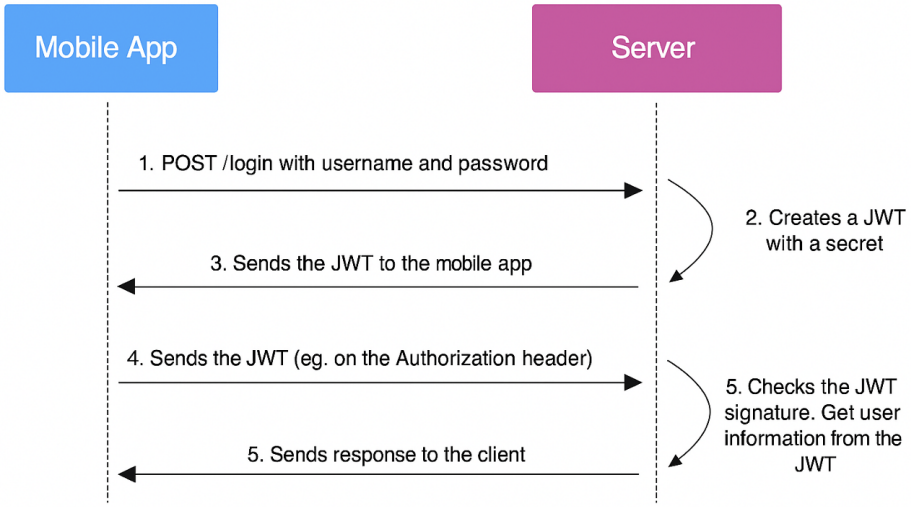

Figure 2.1: High-level view of JWT flow

However, implementation integrity becomes the primary vulnerability. By altering the SDK, for example, to accept invalid or expired tokens or to omit the signature check, an attacker could try to get around the verification logic. This can be lessened by obfuscating and deeply integrating the signature validation process into the codebase. A critical mistake to avoid is using symmetric cryptography (e.g., HMAC) for signing and embedding the shared secret in the SDK; doing so would allow attackers to generate valid tokens by themselves considering this, it is highly recommended to use asymmetric keys for signing and verifying.[10] To support long-term security, key rotation is also advisable. Vendors may choose to allow connected applications to retrieve the current public key from a trusted URL or, for offline use, to embed it in a protected area of the application.[11] In fully offline environments where public key updates are not feasible, including an expiration date in the token ensures that licenses must periodically be refreshed, thereby reestablishing trust under a new key if needed.

In conclusion, token-based licensing, particularly JWT, is well-liked in contemporary SDK licensing due to its flexibility, self-contentedness, and cryptographic security. It can encode device-bound or feature-bound restrictions, as it will be covered next, and it performs well for mobile SDKs that might not always be online.

2.2.4 Offline Licensing with Digital Signatures

Conceptually, offline license files are comparable to token-based licensing and frequently embody the same fundamental concept under a different moniker. This method is still widely used, especially in specialized SDKs and embedded systems, and it predates contemporary standards like JWT. Under this model, a license file, usually a signed or encrypted text or binary blob, or occasionally a digital certificate, is created by the vendor and sent to the client. The file is bundled with the mobile application or included as an asset. At runtime, the SDK loads and verifies the license file; if the signature is valid and the embedded parameters (such as expiration date, allowed features, or device restrictions) are deemed acceptable, the SDK enables its functionality. Otherwise, access is restricted or denied. This method supports fully offline operation while preserving strong authenticity and integrity guarantees.

The strength of the underlying cryptography and the privacy of the vendor's private signing key play a major role in the security of offline signed license files. Because it is computationally impossible to forge a legitimate signature without the private key, the system is extremely resistant to forgery when it is implemented using robust algorithms, like RSA-2048 or its equivalent. The main risk is bypass rather than cryptographic failure: an attacker can completely get around the system if they alter the SDK to omit or fabricate the license validation logic, and this risk applies to all licensing systems. This emphasizes how important tamper-proofing is, which is covered in more detail in a later section.

While some vendors choose to encrypt license contents to obscure internal parameters and make them unreadable to end user developers, the most crucial security element that guarantees the robustness of the licensing system is the digital signature. It ensures that any unauthorized modification to the file will be detected. License files can be distributed as binary blobs with extra opacity or in human-readable formats like signed XML or JSON, which make it simpler for clients to review the terms of the license and improves user experience. The SDK usually parses the license at runtime and verifies parameters like the expiration date, application or device identifiers, and allowed usage levels, regardless of the format. The SDK should reject the license and limit access appropriately if any of the conditions are not met or if the signature cannot be validated. Optionally, it can also notify why the license failed, which will significantly improve user experience but also lowers the security of the system, this is a known trade-off in the cybersecurity world.

2.2.5 Choice justification

Each of the licensing systems discussed have their own advantages and disadvantages, but when it comes to utilizing them for an on-device mHealth use case, the ability of utilizing the system offline becomes an important choice making factor, since as discussed, in many scenarios there might not be a stable network connection and failure to check license and subsequently to use the logic my result in life threating outcomes. Keeping all these in mind, the Offline Licensing with Digital Signatures has been chosen for this thesis due to its high security and offline using availability.

2.3 Licensing Strategies for On-Device Frameworks

2.3.1 Hardware/Device-Bound Licensing

Hardware-bound licensing links the use of an SDK to a particular physical device, extending the scope of traditional license validation. In addition to checking if the license is valid, it also checks if the device that is running the app is recognized and has access. This model is widely used in embedded systems and enterprise software, and can also apply in mobile contexts where the software is deployed on managed or dedicated hardware, such as a company-owned fleet of tablets.

A unique hardware identifier, such as a device serial number, IMEI, UDID, MAC address, or CPU ID, is usually retrieved for implementation and is embedded to the license file or. The SDK checks the licensed value against the identity of the current device during runtime. The SDK activates if they match; if not, access is restricted. By preventing license sharing or replication across unauthorized devices, this provides an extra layer of control. Another limitation similar to this, used most frequently in paid SDK sharing, is checking the application identifier rather than device. Doing so, the vendor can be sure the SDK is used only for the application that the client paid for, regardless of the device it's running on.

Device-bound licensing offers a strong deterrent against unauthorized redistribution by binding SDK functionality to a specific hardware identity. Even if an attacker obtains both the application and a valid license file, they would still need to spoof the licensed device's hardware signature which is an added barrier that discourages casual piracy. This method is especially effective in scenarios where the SDK is intended for deployment on a single device or appliance, such as in industrial or enterprise settings with per-device licensing models. This method adds several levels of protection by requiring both hardware identity matching and license validation, which makes it much more difficult for attackers to completely get past security measures.

Despite its advantages, for consumer-based application, hardware-bound licensing is rarely used since in such scenarios users and buyers expect to install the app and use the SDK freely on as many device as they want.

Therefore, in enterprise or embedded scenarios, where the "user" is an organization that manages a fixed set of devices, like kiosks, IoT hardware, or company-issued tablets, device-binding is more appropriate.

The growing privacy protections on contemporary mobile platforms also present technical challenges. Since iOS no longer allows access to the original UDID, developers are forced to use alternatives like "identifierForVendor" or demand that a device code be manually entered.

2.3.2 Feature-Limited and Modular Licensing

Tiered licensing model is extremely popular among SDK vendors, where different feature sets are enabled based on the purchased license level, for example, a tier may give you access to only 1 feature, but the higher tier to all features. These permissions are enforced at runtime by the SDK, which grants or prohibits access to particular features based on the access level.

An industry standard is adding feature flags and granted rights in the license token. The SDK determines whether the license contains the necessary rights when an application calls a feature. Otherwise, the SDK might raise an error or turn off the feature. This method eliminates the need for separate binaries by allowing access to a matrix of features to be controlled by a single license file. A different approach makes use of modular licensing, in which the main SDK components are licensed separately.

Feature-based licensing, like other restriction methods comes with some security challenges, particularly when the SDK uses on device computation method, which results in code for advanced functionalities being shipped within the application but gated by internal flags. In these situations, attackers might try to circumvent these limitations and unlock premium features without permission by manipulating the SDK, either by patching the binary or by changing memory during runtime. This again highlights the important of designing the SDK and its distribution method.

From a commercial perspective, feature-based licensing makes it possible for vendors to offer specialized packages according to client requirements by enabling flexible sales models. However, because every gated feature is now a possible target for manipulation, it increases the attack potential from a security standpoint. Feature flags should be used in combination with other security measures to reduce this risk. To ensure that new features cannot be enabled by simply editing the license file, for example, it should be digitally signed to prevent unauthorized modification. In order to prevent static analysis and patching, license checks should also be obfuscated and deeply ingrained in feature logic. Sensitive elements of premium features may be implemented in native code or rely on secure hardware calls in high-security scenarios, making them much more difficult to circumvent or reverse

engineer.

2.3.3 Time Limited Licensing

Most licensing systems limit the functionalities by time, supporting models like trial periods or time-bound subscriptions. This is usually accomplished in mobile SDKs by including an expiration timestamp in the license, usually via fields such as the exp claim in JWT-based tokens.[12] If the license has expired, the SDK disables or restricts functionality at runtime by comparing the current system date with the expiration value.

The potential for users to alter the system clock in order to prolong the validity of licenses is the primary problem in offline environments. Some SDKs handle this by flagging any detected clock rewinds and keeping track of the last known valid usage timestamp in secure local storage. Others require recurring online verification to verify license status, but this adds complexity and necessitates connectivity. In completely offline scenarios, vendors frequently issue short-duration licenses that need to be renewed frequently in order to mitigate or accept a certain amount of risk.

The grace period model, which requires revalidation after a limited period of offline validity (such as 30 days) following an initial online activation, is a commonly used compromise. This method provides a useful compromise between license integrity and user convenience, especially in mobile settings where sporadic connectivity is typical.

2.3.4 Floating (Concurrent) Licensing

To enable a restricted number of SDK instances to operate concurrently across a pool of devices, floating, also known as concurrent, licensing models are frequently used in enterprise settings. Usually, this method relies on real-time communication with a central license server that controls availability and license check-outs. Because it can be difficult to maintain constant network access, floating licenses are less common in mobile applications than in desktop software.

However, companies can use on-premise license servers, even in closed networks, to control floating license distribution for controlled deployments, like corporate fleets of mobile devices. Such configurations are supported for large-scale deployments by solutions such as those provided by Cryptolens.[9] Mobile apps can be set up to connect to a local license server on the client's internal network when offline access is required. This shows the flexibility needed to satisfy various licensing requirements in enterprise contexts, even though it differs from strictly on-device

enforcement.

2.3.5 License Servers on Device

Deploying a lightweight license manager directly on the device, where the SDK checks for validation, is a new strategy in embedded and high-security settings. The license management component of these architectures usually functions in a Trusted Execution Environment (TEE) or secure element, which are protected hardware. TEEs are already present in many mobile devices, such as the Secure Enclave on iOS and ARM TrustZone on Android. In theory, these devices could be used to safely store license keys or carry out validation tasks that are not accessible by standard user-level code.

Although this model provides robust defense against tampering and key extraction, it usually requires extensive device platform integration. Because of the intricacy and restricted accessibility of these environments, the majority of third-party SDKs do not directly utilize TEEs as of 2025. However, such mechanisms are commonly used in platform-level digital rights management (DRM) systems, such as those used in secure media playback. In the future, mobile SDK licensing may use more and more platform-provided secure storage options, like Apple's Secure Enclave or Android's KeyStore, to keep cryptographic license materials. By making key extraction and license forgery much more difficult, these technologies have the potential to greatly improve security. However, the majority of SDK licensing is still done at the user-space level for the time being.

2.3.6 Choice Justification

Since the aim of this thesis was to develop a secure framework and library for sharing the physiological state prediction with third party application, security was a very important subject to keep in mind. On the other hand, due to nature of the target audience of this SDK, the ability to work offline all the time was also a very important matter. Keeping these in mind, the thesis selected "Offline Licensing with Digital Signatures" approach. Furthermore, to keep the proposed system dynamic, "Feature-Limited" licensing is employed to give the vendor the ability to define different access levels with just one license. Finally, to enhance the security of this distribution system, similar to "Hardware/Device-Bound Licensing", the license is generated for a specific app identifier (bundle id for iOS and package name for Android), this ensures that only the company that have paid for this license can have access to it and sharing it with other companies will result in license validation failure.

2.4 Security Risks in Mobile License Enforcement

During the previous section, some security issues related to each licensing method and limitations were covered, in this section, a summary of those and generic ones are discussed.

2.4.1 Reverse Engineering and Patching

Reverse engineering is still a threat even with improvements in mobile application security. Attackers can identify and alter license enforcement mechanisms by analyzing app binaries, including embedded SDKs, using widely available tools like APKTool, Jadx, Hopper, and Ghidra. Expert reverse engineers can examine control flow, track execution paths, and try to disable protection mechanisms at the assembly level, even in cases where licensing logic is deeply embedded.

When developing and distributing SDKs, many vendors, including the present thesis, decide to distribute SDKs as precompiled binaries instead of source code in order to reduce this risk. This is typically accomplished on iOS using XCFramework bundles, and on Android, the AAR (Android Archive) library is the equivalent format. Compared to open or semi-obfuscated source code, these binary formats greatly complicate direct inspection or modification because they encapsulate the compiled logic. Sharing only compiled code increases the barrier to tampering and unauthorized use by limiting the exposure of proprietary algorithms and sensitive licensing logic.

It is known, however, that no client-side protection is completely secure. Therefore, the goal is to make reverse engineering unreasonably difficult and time-consuming. Raising the cost of attack to a level where most adversaries cannot afford it is the practical objective.

2.4.2 Key Leakage

The risk of key exposure is inherent in licensing schemes that depend on secret data embedded in the client application, such as static lists of valid license codes or symmetric decryption keys. By examining the binary's strings, breaking down the code, or performing runtime memory analysis, attackers can obtain this information. For instance, static or dynamic analysis can often reveal an embedded private key or a hardcoded license key, which puts the integrity of the licensing system in danger.

To address this issue, modern licensing architectures increasingly adopt public-key cryptography, which avoids placing sensitive secret material in the client app.

Instead, only a public key is embedded in the SDK, which is used to verify the signatures generated by the vendor's private key, which remains securely stored on the server and is not shared with the clients. Advanced methods like white-box cryptography are advised when a secret needs to be on the client, such as when it's required for decrypting encrypted code blocks. White-box cryptography seeks to obscure cryptographic operations to the point where an attacker cannot easily isolate the embedded key, even if they have complete visibility into the execution environment.

In order to prevent key extraction in situations where attackers "have full visibility and control" over the application, vendors like Thales incorporate white-box cryptography into their licensing solutions. These techniques greatly increase the technical barrier, which delays or discourages attacks by making key recovery more difficult, even though they are not perfect against highly skilled adversaries.[13]

2.4.3 Network Attacks

Network-level interference, particularly replay and man-in-the-middle (MITM) attacks, is a frequent threat vector in online licensing systems. For example, if the SDK casually asks a remote server, "Is license X valid?" and unlocks features based on any positive answer, a malicious actor could intercept that request and create a fake "yes," getting around licensing restrictions. To mitigate such risks, licensing servers typically rely on TLS to encrypt communications and authenticate the server, and often supplement this with signed responses or the use of one-time nonces to prevent replay attacks.

To fool the SDK into accepting fabricates responses, attackers working in a completely offline environment might try to replicate the license server entirely by rerouting DNS, creating a phony local server, or taking over endpoints. For this reason, it's essential that the SDK rigorously validates the authenticity of the server, either by verifying a cryptographic signature in the response

2.4.4 Denial-of-Service or Bricking Risks

How the system reacts when tampering or an invalid license is detected is a frequently disregarded aspect of SDK protection. An excessively aggressive SDK may unintentionally allow denial-of-service attacks, such as when it intentionally crashes the host application when it detects abnormalities. Malicious actors may purposefully set off the SDK's tamper response in order to destabilize the application and render it unreliable for authorized users, not to get around licensing.

Well-designed SDKs usually handle such cases with restraint to prevent this. The SDK should react to license problems or tampering attempts in a controlled, predictable manner, like throwing a clear exception or silently disabling premium features, rather than erasing data or violently stopping the process. The objective is not to penalize the user or jeopardize the stability of the entire application, but to ensure that protected features are unavailable while the rest of the app continues to function normally.

Chapter 3

System Overview

3.1 System Objectives and Use Context

The objective of the system presented in this work is to provide a secure and modular mobile software framework mainly designed for analyzing psychological health data and predicting different health states, such as drowsiness, fatigue, or Alcohol misuse state, without relying on any server-side infrastructure. It is worth mentioning that developing such IPs are beyond the scope of this thesis, as here the best way to distribute it with third-party developers as a plug and play solution is discussed.

Third-party mobile developers can integrate this system to add sophisticated health prediction algorithms to their own apps while strictly protecting the proprietary logic underlying the algorithms.

Each of the two platform-specific components that make up the system, a precompiled Android AAR library and a binary Swift-based XCFramework for iOS, encapsulates the essential detection logic in a format that is simple to embed but cannot be examined or changed. The internal implementation is deliberately kept totally hidden from the client application. The only interface that developers can use is a well-defined one that manages feature-specific prediction outputs, license validation, input ingestion, and security checks.

The SDK's intended integration into third-party applications, which are outside the direct control of its original developers, is a defining feature of the system's deployment. Significant difficulties in safeguarding the underlying intellectual property are brought about by this external integration. Unlike standalone mobile applications, where developers retain full authority over how the code is executed

and secured, this solution must function as a plug-and-play component delivered to external teams, many of whom will have access to the compiled binary. However, that access must not extend to the proprietary algorithms embedded within. As a result, safeguarding the algorithmic logic and enforcing licensing restrictions are not optional design features but essential pillars of the overall architecture.

The system is designed with the following limitations in mind in order to achieve these goals:

- No dependency on continuous connectivity to the internet, ensuring the framework can function in remote and offline environments.
- strong licensing restrictions that restrict feature access in accordance with the license terms and bind each instance to a particular application.
- Strong runtime security that can identify compromised environments and stop devices that have been jailbroken or rooted from being used.
- Cross-platform compatibility, which maintains the same security guarantees while enabling smooth deployment across iOS and Android.

This system's structure, design principles, and component interactions are thoroughly examined in the following sections.

3.2 SDK Architecture (iOS and Android)

This section presents the high-level technical architecture of the mobile framework as implemented on iOS and Android platforms. While the two ecosystems differ significantly in terms of packaging formats and development environments, the framework adheres to a common set of design principles—namely, modularity, encapsulation, and functional parity. In the following discussion, the structure of the framework on each platform is explained, along with the reasoning behind its distribution as a precompiled binary (using AAR for Android and XCFramework for iOS). It also describes the division of essential features into discrete parts, including licensing, security enforcement, and physiological feature detection. In addition to facilitating safe integration, this architectural strategy guarantees a standardized and efficient developer experience on both mobile platforms.

3.2.1 iOS: XCFramework Packaging and Architecture

In 2019, Apple released Xcode 11 with the XCFramework format, a cutting-edge way to distribute precompiled binaries across various architectures and platforms.[14]

This format was designed to replace older "fat" frameworks and address their limitations, particularly around multi-platform compatibility and architecture-specific packaging. XCFrameworks' support for true binary distribution is one of its main advantages; their implementation is completely closed-source, which is crucial for safeguarding proprietary SDK logic, and they only include compiled code and public interface definitions.

By combining several architecture slices, like arm64 and x86-64, into a single package, XCFrameworks also makes cross-platform support easier. This streamlines the development process and guarantees consistent behavior across devices and simulators by doing away with the need for manual merging with tools like Lipo. Furthermore, they support Swift module stability, which allows the binary to remain compatible with future Swift compiler versions and architectures without requiring the developer to recompile the framework. These advantages make XCFrameworks particularly well-suited for distributing secure, multi-platform SDKs in the iOS ecosystem.

Within this XCFramework, the internal architecture comprises:

1. Core module – Entry point for initialization, license and security enforcement, and feature invocation.

2. License module – Verifies digital licenses using embedded public keys.

3. Security module – Detects jailbreak and runtime tampering.

4. Feature modules – Swift-based logic for each physiological state prediction.

3.2.2 Android: AAR Packaging, Native Code, and Architecture

The framework is distributed as a precompiled AAR (Android Archive), which is the common packaging format for reusable libraries on the Android platform. The framework can be easily incorporated into client applications thanks to the AAR file, which contains compiled bytecode, resources, manifest entries, and, when required, native binaries. Both Gradle and Android Studio fully support AARs, which are the standard distribution method for modular Android components. This makes it easy to integrate and configure AARs in third-party projects. This format ensures that the SDK remains self-contained while adhering to Android development conventions, making it a reliable vehicle for delivering secure, plug-and-play functionality.[15] Android's Java and Kotlin code is by default compiled into .class files, which are then converted into Dalvik bytecode (.dex), which is

easily decompiled with widely accessible tools. This ease of reverse engineering poses a significant risk to intellectual property, particularly for proprietary frameworks. In order to enhance security and improve runtime efficiency, the framework implements its most sensitive components using the Android Native Development Kit (NDK). By using C or C++ to write the core logic of the system and then compiling them to .so shared libraries, we can increase the security of the system and make it way more difficult to decompile and reverse engineer.

The modular layout parallels the iOS implementation:

1. PredictS (Core) – serves as the system's initialization, license and security check, feature request routing, and public-facing API.
2. LicenseVerifier – uses an embedded public key to validate signed license files locally. Verifies feature authorization, package name, and expiration.
3. SecurityChecks – detects the presence of debuggers, runtime hooks such as Frida or Xposed, and the device root. For more difficult detection resistance, this logic can be implemented in native code.
4. Feature modules – A separate class for any feature that the SDK offers.

For performance and IP protection, the important logic of each module is written in C++, with only a thin JNI bridge exposed to the Kotlin layer. Performance is improved, and the bar for reverse engineering is raised when NDK is used.

3.2.3 Cross-Platform Architectural Principles

Both iOS and Android implementations of the framework follow a common architectural design based on four fundamental principles, despite their different packaging conventions:

- Modular structure: Clear division of responsibilities is made possible by the organization of functionality into distinct modules, such as Core, Licensing, Security, and Feature Detection. Code maintainability is enhanced, targeted testing is supported, and future scalability is made easier by this modularity.
- Encapsulation of logic: Private modules contain all proprietary algorithms and security measures. Internal logic is protected from external access and inspection by the public interface, which only exposes the necessary APIs for integration.
- Binary-only distribution: The SDK is only made available in compiled binary form to preserve intellectual property. Since there is no source code provided, the underlying implementation is kept closed and impenetrable.

- Platform parity: The same logic flow, licensing structure, and runtime security measures are enforced by both the iOS and Android versions. This simplifies cross-platform development and support while guaranteeing consistent behavior across platforms.

This consistent design philosophy allows the framework to deliver a secure, maintainable, and developer-friendly experience regardless of the target mobile platform.

Chapter 4

Framework and Library Implementation

4.1 Overview of SDK Structure

The SDK architecture is structured around four principal components: the Core controller, the License management module, the Security enforcement layer, and a collection of feature-specific prediction modules. This modular organization is mirrored across both the iOS and Android implementations, supporting the overarching goal of platform parity and ensuring uniform behavior across environments.

A high level flowchart of system could be find in Figure 4.1, the client's app embeds the SDK and calls the SDK initialization and passes license payload and signature, then the license is checked (to be discussed in depth in following sections), if not valid for any reason, SDK stops and returns an error, if valid it checks the list of available features and continues and runs security checks such as reverse engineering, root and jailbroken devices (to be discussed in depth in following sections). If it fails, the SDK again returns an error; otherwise returns the Predictor object on which the user can call different features such as Sleep detection, Fatigue, and Alcohol. Upon calling, each the SDK checks if it's licensed, otherwise returns an error.

The modular and single entry point makes the SDK more maintainable and scalable since for adding new prediction features we will add a separate file and class and just call it through a method in main PredictS class.

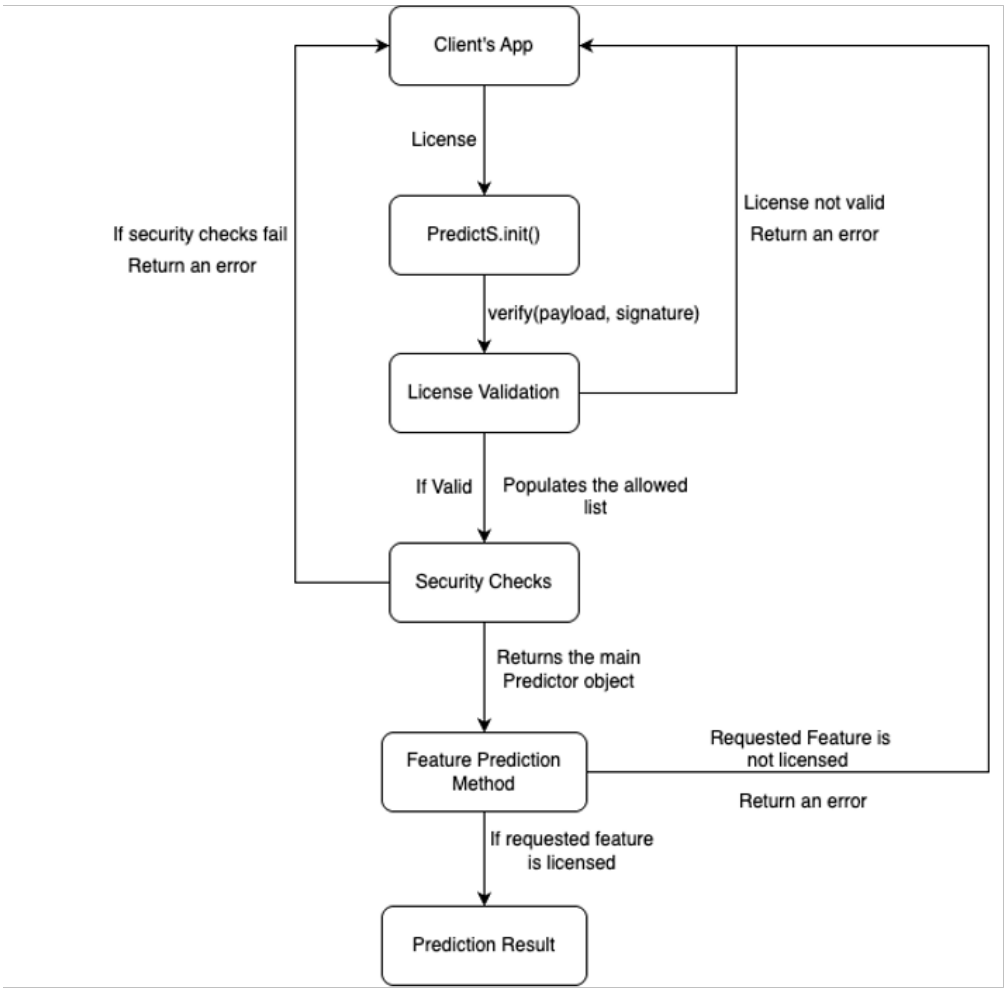

Figure 4.1: High-level Flowchart of system

4.2 iOS XCFramework

The PredictS SDK is provided on iOS as a precompiled XCFramework, which bundles all required binary slices, including those for simulators and physical devices, and the matching Swift module interfaces into a single bundle. This distribution format gives developers a simplified, plug-and-play experience by facilitating smooth integration into third-party iOS applications. The proprietary nature of the framework's core logic is maintained by using XCFramework to ensure that the internal implementation details are hidden by encapsulating only compiled binaries. In this section, different parts of the XCFramework are discussed. In order to keep this thesis concise and efficient, only important parts of the code are highlighted in the text.

4.2.1 Internal Modules (PredictS, License, Security, Features)

Within the XCFramework, functionality is cleanly divided into four main internal sections, each responsible for a distinct concern:

Core Controller (PredictS)

Located at PredictS.swift file, this is the start point of the SDK, as it can be seen in its init function in Listing 4.1, it receives the licensePayload and licenseSignature and passes them to the PredeictSLicense class, which will be discussed in depth in a different section, where the license pair is checked and the list of available features are extracted. After the license check, in case of success, a series of security checks are done to ensure a safe environment for running the SDK.

Listing 4.1: init function of PredictS class

```
1  /// Initializes PredictS with a license payload and signature
2  ///
3  /// - Parameters:
4  /// - licensePayload: The encoded license information
5  /// - licenseSignature: The cryptographic signature of the license
6  /// - Throws: `PredictSError.invalidLicense` if the license
       validation fails
7  public init(licensePayload: String, licenseSignature: String) throws
       {
8      print("[PredictS] new Initializing...")
9
10     // Validate the license: this checks the payload signature match,
       bundle ID, expiry, and features.
```

```
11      self.license = try PredictSLicense(payload: licensePayload,
        signature: licenseSignature)
12
13      // Run security checks to prevent use on compromised/jailbroken
        devices or with tampered
14      // libraries
15      SecurityChecks.performAllChecks()
16
17      // Log available features after license validation
18      print("[PredictS] License valid with features: \(license.
        allowedFeatures)")
19  }
```

In case of success in both license check and security check, then the third-party developer can call the features they want, for example, for sleep detection which is shown in Listing 4.2 user provided the needed inputs which are defined by the SDK through its public interface and then the SDK checks if sleep is in the licensed otherwise it will return an error. The same procedure is applied for other features and any feature to be added in the future.

Listing 4.2: Drowsiness state detection function in PredictS file

```
1  /// Performs sleep detection based on health data and current mode (e
       .g., night/day)
2  ///
3  /// - Parameters:
4  ///   - input: The health data used to assess sleep state
5  ///   - isNightMode: A boolean indicating if night mode is active
6  /// - Returns: A `SleepState` indicating the user's current sleep
       status
7  /// - Throws: `PredictSError.featureNotLicensed` if sleep detection
       is not licensed
8  public func sleepDetection(input: SleepHealthData, isNightMode: Bool)
       throws -> SleepState {
9      // Ensure the "sleep" feature is licensed before proceeding
10     guard license.allows("sleep") else {
11         throw PredictSError.featureNotLicensed("sleep")
12     }
13
14     return sleepDetector.predict(input: input, isNightMode:
       isNightMode)
15 }
```

License Management

Located at PredictSLicense.swift file, it acts as the single source of truth for all checks and processes regarding license and access management. Due to its

importance and its diverse concepts, it will be discussed in a separate section for both android and iOS.

Security Checks

In this file different tests are done in runtime to detect if the environment that the app and as a result the SDK is running is safe. Since checks in this section are OS specific, the contents of it are different on Android and iOS.
A main test employed within the SecurityChecks.swift module is to scan the file system for indicators of a jailbroken environment. The process of getting root access to an operating system on a closed-system device, like an iPhone, by getting around its built-in security features, is known as jailbreaking. With this elevated access, users can install unauthorized software, change system files, and exercise complete administrative control over the device, circumventing restrictions imposed by the manufacturer.[16] Searching for known filesystem artifacts that are frequently left behind during the jailbreak process is one popular technique for identifying an iOS device that has been jailbroken. Applications such as Cydia, a third-party package installer, or configuration files linked to the APT package manager are clear signs that the device has been compromised.

When such artifacts are found, platform integrity is usually compromised. These file-based checks are relatively simple to implement using standard APIs such as FileManager.fileExists(atPath:), or via lower-level system calls like stat() and access(). The code snippet responsible for checking jailbroken devices is presented at 4.3

Listing 4.3: Jailbreak check logic

```
/// Checks for signs of a jailbroken device
///
/// - Returns: `true` if the device is jailbroken, otherwise `false`
private static func isJailbroken() -> Bool {
    // list of known jailbreak file paths
    let jailbreakPaths = [
        "/Applications/Cydia.app",
        "/Library/MobileSubstrate/MobileSubstrate.dylib",
        "/bin/bash",
        "/usr/sbin/sshd",
        "/etc/apt",
        "/private/var/lib/apt/",
        "/private/var/tmp/cydia.log"
    ]

    // Check if any jailbreak file exists
```

```
17      for path in jailbreakPaths {
18          if FileManager.default.fileExists(atPath: path) {
19              print("[Security] Jailbreak file found: \(path)")
20
21              return true
22          }
23      }
24
25      // Also test if the app can write outside the sandbox
26      if canWriteOutsideSandbox() {
27          print("[Security] Device is jailbroken: can write outside
    sandbox.")
28
29          return true
30      }
31
32      return false
33  }
```

Another detection strategy implemented in SecurityChecks.swift targets potential violations of the iOS sandboxing model. Under Apple's security architecture, all applications are strictly confined to their own sandbox and executed under a non-privileged user account typically the mobile user. As part of this model, apps are prohibited from accessing system files or interacting with the data of other applications.[17] The SDK uses Apple's dynamic loader APIs, like _dyld_image_count() and _dyld_get_image_name(), to list all dynamically loaded libraries at runtime in order to detect this kind of tampering. The names of these libraries are then examined for known signs of compromise, specifically searching for substrings linked to popular instrumentation tools like "frida," "FridaGadget," "libcycript," and "MobileSubstrate." Any such module found in the application's memory is interpreted as a clear indication that the process is being actively instrumented or altered, and the check is initiated appropriately. This method essentially serves as a safeguard against dynamic analysis and runtime hooking, two common methods for evading licensing and deciphering proprietary logic. Combining runtime library scans, sandbox violation tests, and file-path inspections creates a layered security approach that complies with industry best practices.[18] Each technique focuses on a different type of compromise: dynamic code injection, privilege escalation, and filesystem anomalies, respectively. Combining these complementary methods results in a more robust detection framework, which raises the overall barrier to exploitation and significantly increases the effort needed for a successful attack.

Feature Files

Each feature has its own class, making the framework completely modular and free of entangled code. In each class the I/O of the class is defined as public so the third-party developers who use the framework will have access to it and know in advance how to provide data and what type of outcome to expect. For example, in Listing 4.4, the public interface of Fatigue prediction class can be observed, in this case the input is of type Struct which let's third-party developers provide required data in one object, and the output is of type enum which defines the possible states.

Listing 4.4: Public Interface of Fatigue Prediction

```
1
2  /// Represents the possible fatigue levels detected from health data
3  public enum FatigueState: String {
4      case normal     // No signs of fatigue
5      case mild       // Some indications of light fatigue
6      case severe     // Strong signs of severe fatigue
7  }
8
9  /// Container for health metrics relevant to fatigue detection
10 public struct FatigueHealthData {
11
12     public var heartRate: Int?    // Heart rate (beats per minute)
13     public var hrv: Int?          // Heart Rate Variability
14     public var accZ: Int?         // Z-axis accelerometer value (
       optional motion data)
15
16     /// Initializes the health data used for fatigue prediction
17     ///
18     /// - Parameters:
19     ///   - heartRate: Optional heart rate value
20     ///   - hrv: Optional heart rate variability
21     ///   - accZ: Optional Z-axis acceleration
22     public init(heartRate: Int?, hrv: Int?, accZ: Int?) {
23         self.heartRate = heartRate
24         self.hrv = hrv
25         self.accZ = accZ
26     }
27 }
```

4.2.2 Building and Integration in Customer's App

After implementing, modifying or extending the framework, In accordance with best practices in software release engineering, the vendor executes the following stages:

1. Archive for Device: compile and archive the framework for physical iOS devices.
2. Archive for Simulator: compile and archive the framework for the iOS Simulator.
3. Create XCFramework: merge the two archived slices into a single multi-architecture XCFramework.
4. Distribution: publish or share the finalized .xcframework bundle with third-party integrators.

The commands illustrated in Listing 4.5 show this sequence.

Listing 4.5: Commands for distributing .xcframework

```
1
2 # 1. Archive for iOS Devices
3 xcodebuild archive \
4   -scheme PredictSFramework \
5   -destination "generic/platform=iOS" \
6   -archivePath "./build/PredictSFramework-iOS.xcarchive" \
7   SKIP_INSTALL=NO BUILD_LIBRARY_FOR_DISTRIBUTION=YES
8
9 # 2. Archive for iOS Simulator
10 xcodebuild archive \
11   -scheme PredictSFramework \
12   -destination "generic/platform=iOS Simulator" \
13   -archivePath "./build/PredictSFramework-Sim.xcarchive" \
14   SKIP_INSTALL=NO BUILD_LIBRARY_FOR_DISTRIBUTION=YES
15
16 # 3. Create XCFramework
17 xcodebuild -create-xcframework \
18   -framework "./build/PredictSFramework-iOS.xcarchive/Products/
     Library/Frameworks/PredictSFramework.framework" \
19   -framework "./build/PredictSFramework-Sim.xcarchive/Products/
     Library/Frameworks/PredictSFramework.framework" \
20   -output "./PredictSFramework.xcframework"
```

Once the .xcframework is shared with the third-party developer they need to embed the framework into their code base by going to application target's General, Frameworks, Libraries, and Embedded Content pane, drag PredictSFramework.xcframework, and select Embed and Sign. This ensures that the framework's code signature is validated at install time and that its dynamic library is bundled within the app's .ipa. Once all these are done, they can use the framework as shown in the Listing 4.6.

Listing 4.6: How developers can use the framework

```
do {
    let predictor = try PredictS(
        licensePayload: "<Base64-payload>",
        licenseSignature: "<Base64-signature>"
    )
    let fatigueState = try predictor.fatigueDetection(
        input: FatigueHealthData(heartRate: 75, hrv: 45, accZ: nil)
    )
    print("Detected fatigue state: \(fatigueState.rawValue)")
} catch {
    // Handle invalid license or security violation
    print("PredictS error: \(error)")
}
```

4.2.3 What third-party developers have access to

After the successful distribution and embedding of the framework, if the third-party developer wants to open the framework file and classes, they will only be able to see the public interface of the system and not the actual core logic, as shown at Listing 4.7, this protects the IP and aligns with the goals of this thesis.

Listing 4.7: Public interface shared with third-party developers

```
public class SleepDetection {

    /// Default initializer
    public init()

    /// Predicts the current sleep/drowsiness state based on health
    data and mode
    ///
    /// - Parameters:
    ///   - input: Health data to be used in prediction
    ///   - isNightMode: A Boolean indicating whether it's currently
    night mode
```

```
11      /// - Returns: A `SleepState` indicating the current drowsiness
        level
12      public func predict(
13          input: PredictSFramework.SleepHealthData,
14          isNightMode: Bool
15      ) -> PredictSFramework.SleepState
16
17      @objc deinit
18  }
```

4.3 Android AAR Library

The Android implementation of the SDK is distributed as an Android Archive (AAR) library, which combines Kotlin-based interfaces with C++ native components to support a secure and modular design. The AAR format is the industry standard mechanism for packaging Android libraries, allowing developers to bundle compiled Java/Kotlin classes, resources, manifest entries, and native .so binaries into a single, reusable module.[15] By using this format, the SDK makes it easier to integrate into third-party applications by delivering all necessary components in a single package.

4.3.1 Architecture and Components of the Android SDK

The SDK uses a modular design that divides duties between a native C++ layer and a Kotlin-based layer. As the public-facing interface, the Kotlin module manages the integration of the Android framework and makes the SDK's API available to client applications. However, the Java Native Interface (JNI), which serves as a link between managed and native code, assigns security-critical tasks to the C++ module. This separation is intentional and strategic. Native code, once compiled into .so shared libraries, is translated into platform-specific machine code, making it considerably more difficult to reverse-engineer than standard Java or Kotlin bytecode.[19] As a result, implementing proprietary algorithms and license validation routines in C++ substantially increases the complexity of static analysis and tampering. By placing sensitive logic in the native layer, the SDK leverages one of the most effective code protection techniques available in the Android ecosystem.

The overall structure of files is the same as iOS, so to prevent repetition, the following sections go deeper in platform specific security and safeguard methods that are used to protect the IP of the SDK.

4.3.2 Kotlin Layer

The Kotlin layer of the SDK comprises the set of classes and methods exposed to the host application, forming the primary interface through which third-party developers interact with the framework. This component is responsible for managing Android-specific functionality. It also serves as the bridge to the underlying C++ logic, invoking native routines via the Java Native Interface (JNI). A Kotlin singleton or companion object loads the native library and declares a set of external methods that map to native C++ functions in order to initialize the SDK in a standard implementation. The API can then internally delegate sensitive operations to the native layer while maintaining a clear and idiomatic Kotlin interface by calling these methods from Kotlin code as though they were regular functions. For example, as shown in Listing 4.8, in the license checker file, instead of implementing the actual logic, the native library is loaded, and the isValidNative function is called to handle and check the authenticity of the license.

Listing 4.8: Kotlin interface of License checker

```kotlin
package com.sat.predictlibrary

import android.content.Context

object License {
    init { System.loadLibrary("predictslib") }

    /**
     * Returns true if:
     *  - license JSON is well-formed
     *  - signature is valid (RSA-SHA256)
     *  - bundle ID matches the app
     *  - expiry date is in the future
     */
    @JvmStatic external fun isValidNative(
        context: Context,
        payload: String,
        signature: String
    ): Boolean
}
```

4.3.3 C++ Native Layer

The native component, compiled using the Android NDK, is responsible for all security-critical functionality. This includes license key verification, runtime integrity checks, and the execution of proprietary algorithms that must remain protected from inspection or tampering. The SDK makes sure that its most delicate logic is hidden from common reverse engineering methods by separating these

operations in native code. Same as the iOS section, since the licensing and license check flows are similar, and due to its importance, it will be covered in a separate section. In this section only runtime integrity checks, which are OS specific, are covered.

A variety of security checks are implemented by the SDK's native layer to determine whether the device environment has been compromised, specifically through runtime instrumentation or rooting. Root detection usually involves scanning for well-known indicators, such as the presence of privileged binaries like su in standard filesystem locations, or packages like Superuser.apk, which are strong signals that the device has been rooted. To find anomalies suggestive of elevated privileges, further checks might analyze kernel-level data or query system properties. Low-level system calls, like access(), are used to carry out these validations within the C++ layer in order to confirm the existence of suspicious paths. Since it reduces the attacker's surface area within the managed runtime, implementing such logic in native code makes it much more resistant to tampering or bypass via Java-level hooks. The native layer carries out anti-instrumentation checks in addition to root detection. For example, it inspects /proc/self/status and flag the presence of a non-zero TracerPid value, an established method for identifying whether a debugger is attached. It also scans active processes and network ports for patterns associated with dynamic analysis tools like Frida, which injects processes (e.g., frida-server) or opens specific ports to hook application logic. The SDK is way more efficient at identifying and thwarting runtime manipulation attempts that might jeopardize license enforcement or reveal proprietary algorithms by carrying out these checks at the native level. A snippet of the native code in charge of root detection is shown at Listing 4.9

Listing 4.9: Native root detection code

```
// A) Root paths
const char* paths[] = {
    "/system/xbin/su", "/system/bin/su",
    "/system/app/Superuser.apk", "/sbin/su"
};
struct stat st;
for (auto p : paths) {
    if (stat(p, &st) == 0) {
        LOGE("root detected: %s", p);
        return JNI_FALSE;
    }
}
```

All native components developed in C++ are compiled into shared object libraries (.so files), which encapsulate the platform-sensitive logic of the SDK. During the build process, the system generates multiple versions of these binaries, each targeting a specific Application Binary Interface (ABI), such as ARMv7, ARM64, or x86. This multi-ABI compilation ensures that the final AAR package remains compatible with a broad range of Android devices, accommodating variations in processor architecture and delivering seamless integration across the Android ecosystem. Although their goals are very different, it is crucial to understand that both malicious actors and legitimate developers may use native libraries to hide code from scrutiny.[20] The use of native code in this framework is strongly focused on the former: protecting proprietary algorithmic logic and enforcing licensing restrictions with strong, technically sound mechanisms.

4.3.4 Build Configuration: CMake and Gradle Integration for Native Libraries

Building the SDK with its native component necessitated careful coordination between the C++ and Android Gradle build systems to generate an AAR package that includes the compiled .so binaries. The directory structure of the project adhered to the standard Android NDK structure. The C++ source files and a CMakeLists.txt file that specifies the native build configuration are located in the src/main/cpp/ directory within the SDK module. This CMake script includes all necessary source files, declares dependencies or compiler flags as necessary, and specifies the library type.[19] Only JNI-required symbols are exported in this implementation because the script sets the C++ language standard to C++17 and uses flags like -fvisibility=hidden to limit symbol exposure. On the Gradle side, the Android Gradle Plugin offers native support for integrating CMake through the externalNativeBuild block. Within the module's build.gradle file, CMake integration was enabled by referencing the appropriate CMakeLists.txt script and specifying any required NDK configurations. This setup allows Gradle to coordinate the native build process alongside the Java/Kotlin build pipeline, ensuring that the compiled native libraries are properly included in the resulting AAR artifact.

As a result of this configuration, when the library module is built, such as through the assembleRelease Gradle task, the native .so libraries are compiled and automatically packaged into the final AAR. Gradle's adaptability also makes it possible to customize the build output by defining build types or product flavors. For example, a debug version of the SDK might have more logging, but the release version might not have these security and performance checks. By passing conditional compilation flags from Gradle to CMake, features such as verbose native logging can be included selectively depending on the type of build.

4.3.5 Integration in Customer's App

The SDK can be made available as an AAR file or via a Maven repository. The recommended approach is to host the SDK on a Maven repository, like Maven Central, or a private Maven server. Alternatively, in scenarios where Maven distribution is not feasible, the AAR can be manually included by placing it in the application's libs/ directory and declaring it as a local file dependency within the Gradle build script. In both cases, the Android build system will correctly package the AAR into the final application.

Once successfully added to the project, customers can use the library similar to the flow of the iOS .xcframework, a sample of use case which is aimed at testing the license is shown at Listing 4.10. As it can be seen, license payload, signature and app context should be passed to the constructor of the library, then based on the required feature they can call the specific feature method that first checks if that feature is licensed or not and then proceeds with it.

Listing 4.10: Sample initializing of the library

```
try {
    val predictsS = PredictS(payload, signature, context = this)
    when (feature) {
        "sleep" -> {
            val result = predictsS.sleepDetection(SleepHealthData(),
    isNightMode = false)
            showResult(message = "License valid for
    $featureDisplayName.\nResult: $result", state = "success")
        }
        "fatigue" -> {
            val result = predictsS.fatigueDetection(FatigueHealthData
    ())
            showResult(message = "License valid for
    $featureDisplayName.\nResult: $result", state = "success")
        }
        "alcohol" -> {
            val result = predictsS.alcoholDetection(AlcoholHealthData
    ())
            showResult(message = "License valid for
    $featureDisplayName.\nResult: $result", state = "success")
        }
    }
} catch (e: PredictSError.FeatureNotLicensed) {
    showResult(message = "License is NOT valid for
    $featureDisplayName.", state = "fail")
} catch (e: PredictSError.InvalidLicense) {
    showResult(message = "License error: ${e.message}", state = "fail
    ")
```

```
21 } catch (e: Exception) {
22     showResult(message = "Unknown error: ${e.message}", state = "fail
       ")
23 }
```

Chapter 5

Licensing System

5.1 Design Goals and Constraints

Protecting the SDK from unauthorized use while preserving a seamless experience for authorized developers is the main goal of the licensing system. Full offline operability is a crucial prerequisite; license verification cannot be dependent on constant network access. For mHealth scenarios, where external connectivity may be limited or nonexistent, this design consideration is crucial. To uphold security in such conditions, the system relies on public-key cryptography. Each license file is digitally signed using the vendor's private key, ensuring that only authentic, vendor-issued licenses are accepted. Any attempt to modify the license contents invalidates the signature, effectively making the file tamper-evident and resistant to forgery.

By embedding fields like expiry timestamps and lists of permitted capabilities, the license structure supports a variety of distribution models, such as feature-based licensing and expiration-based (time-limited) licensing. Notably, the implementation is platform-agnostic, although the SDK is delivered in Swift for iOS and Kotlin (and native C++) for Android, the license verification logic remains consistent across both environments, with a shared focus on cryptographic integrity.

The licensing system is designed with a number of fundamental limitations and presumptions in mind. The most important of these is that the vendor's private key, which is used to sign all authentic license files, must be kept safe and never be made public. Compromise of this key would undermine the entire licensing mechanism, as it would enable malicious actors to generate counterfeit licenses that appear valid. The matching public key is included in the SDK on the client side only to confirm license signatures. The integrity of the system is maintained even

in untrusted environments thanks to this asymmetric cryptographic arrangement, which guarantees that although the SDK can authenticate a license, it cannot create or modify one. Second, the choice to allow offline license verification means that expiration dates must be enforced using the client device's system clock. Because of this reliance, there is a chance that a user could try to manually backdate the device's clock in order to avoid the license expiration date. One known drawback of offline licensing models is this kind of manipulation. To address this, the SDK can implement mitigating strategies—such as persisting the most recent valid timestamp in secure local storage or leveraging tamper-resistant time sources where available. Nonetheless, complete immunity to time spoofing is difficult without periodic online validation, and a degree of trust in the client environment is necessary.

Notwithstanding these drawbacks, the design effectively strikes a balance between security and usability by maintaining complete offline operability and enforcing license authenticity through digital signatures, thereby deterring unauthorized usage without imposing online activation requirements on legitimate users.

5.2 License Creation and Verification Flow

This licensing system uses a secure, script-driven license generation process that combines asymmetric cryptographic signing and structured data serialization. The system ensures the integrity and authenticity of license data by digitally signing each license using the private key of the vendor. This design guarantees that only licenses issued by the vendor can be recognized as valid while enabling the resulting license files to be validated locally on the device, without the need for online validation.

5.2.1 Cryptographic Primitives

The licensing mechanism used in this SDK is based on proven cryptographic primitives that were picked especially to guarantee the integrity, authenticity, and impenetrability of license files. Key elements include RSA digital signatures, SHA-256 cryptographic hashing, and Base64 encoding. Each of these components serves a distinct role: RSA enables the secure verification of license origin, SHA-256 guarantees the integrity of the license contents, and Base64 encoding facilitates safe transmission of the license data across text-based channels. Together, these elements support a robust and offline-capable license validation process.

RSA Public-Key Cryptography and Digital Signatures

At the heart of the licensing system is the RSA algorithm, an established asymmetric cryptographic method based on a pair of mathematically related keys: one public, one private. In this design, the private key is securely maintained by the software vendor and used solely to generate digital signatures, while the corresponding public key is embedded within the mobile SDKs for iOS and Android to verify those signatures.

The digital signature serves to attest to the integrity and genuineness of the licensed material. When issuing a license, the vendor signs the license payload with its private RSA key, producing a signature that cryptographically links the data to the vendor's identity. Only licenses that are actually issued by the vendor are accepted because the SDK uses the public key to validate this signature on the client side. The SDK would reject the license and the signature verification would fail if the license data were altered or falsified. This application of RSA is consistent with well-known standards like RSASSA-PSS and PKCS#1 v1.5, which are both extensively used in secure software licensing and distribution. The implementation adopts a 2048-bit RSA key length, in accordance with NIST SP 800-57 and prevailing industry recommendations for long-term cryptographic resilience.

SHA-256 Hashing

The SHA-256 algorithm, part of the SHA-2 family of cryptographic hash functions, is employed to compute a fixed-length digest of the license payload before it is digitally signed. An arbitrary-length input is converted into a fixed-size, deterministic, collision-resistant, and computationally irreversible output by a cryptographic hash function like SHA-256. This effectively renders the digital signature invalid since any alteration to the input, even a one-bit change, results in an entirely different hash.

This behavior is essential for maintaining the integrity of the license data. It ensures that an attacker cannot alter key fields, such as the bundle ID, expiry date, or enabled features, without detection. In the licensing workflow, the vendor computes the SHA-256 hash of the original license payload and signs it using RSA. The same hash is later recomputed and verified by the client using the embedded public key, ensuring that the license remains both authentic and untampered.

Base64 Encoding

The license payload and its digital signature are both fundamentally binary data structures. The payload is a JSON object encoded in UTF-8, and the signature is

a cryptographic byte sequence produced by RSA. To ensure these binary elements can be reliably embedded in license files and transmitted in JSON format they are encoded using Base64. This encoding transforms binary data into an ASCII string representation, preserving the integrity of the information across systems that may not support raw binary formats. It also enables seamless parsing and transport within standard software development workflows.[21]

5.2.2 License Signing Workflow

Building upon the cryptographic foundations discussed in the last section, this section details the practical implementation of the license generation process. By using public-key digital signatures, the process is intended to preserve the fundamental values of authenticity, integrity, and offline verifiability. The software vendor handles every step of license creation independently, so no client-side intervention is needed during generation. The outcome is a digitally signed JSON license file, which can be safely distributed to authorized third-party developers or end-users for integration into their applications.

License Parameter Definition

Each license file encapsulates a compact set of structured fields that define the usage rights granted to a specific application. These fields include:

- bundle_id: the unique identifier of the authorized application—corresponding to the Bundle Identifier on iOS or the Package Name on Android;
- Expiry: a date string in YYYY-MM-DD format indicating the license's expiration
- Features: an array specifying the enabled capabilities (e.g., "sleep", "fatigue", "alcohol"), which determine the functional modules accessible within the SDK.

A sample of such initialization is shown at Listing 5.1, as it can be seen, the section of parameter definition is completely non-technical personnel friendly and readable. After defining parameters, this data is encoded as UTF-8 bytes and serialized into a minimal JSON object. The serialized payload is then Base64-encoded to enable compatibility with text-based systems and enable dependable transmission or storage. The end product is a single-line ASCII string that can be easily embedded into files or API responses while maintaining the integrity of the binary data.

Listing 5.1: License Parameters

```
# Path to PEM-encoded RSA private key
PRIVATE_KEY_PATH = "private_key.pem"

# License parameters
BUNDLE_ID    = "com.example.app"    # The bundle ID the license is valid for
EXPIRY_DATE = "2026-12-31"          # Expiration date in "YYYY-MM-DD" format
FEATURES     = ["sleep"]            # List of enabled features ["sleep","alcohol","fatigue"]
```

Digital Signature Generation

To ensure both the authenticity and integrity of the license payload, the vendor applies a digital signature to the original UTF-8–encoded license data prior to Base64 encoding. This process begins by computing a SHA-256 hash of the raw payload. The resulting digest is then signed using the vendor's private RSA key, employing the PKCS#1 v1.5 signature scheme. This cryptographic operation binds the payload to the issuer and makes any subsequent modification to the data reliably detectable.

To ensure safe transport as a text string, the digital signature that is created as a binary byte array is then Base64-encoded. Using the public key included in the SDK, this encoded signature permits verification and binds the license payload to the issuer in a unique way. The final JSON structure, which contains the Base64-encoded payload and its matching signature, is what makes up the entire license file that is sent to clients. A sample of what third-party developer receives as a license is shown at Listing 5.2.

Listing 5.2: License Sample

```
{
  "payload": "
    eyJjdW5kbGVfaWQiOiJiJ2b2uc2F0LmV4cGVyYWl1bnRhbCIsImV4cGlyeSI6Ij
  IwMjY....",
  "signature": "
    MhE3a8luefqb7o4BilHbisxIgrxuIm3aClVfvDbzuji1xycE3aYhWoq4YXDtd1dy3o
  4vRGcSXS....."
}
```

5.2.3 Offline Client Integration

Authorized clients receive the generated license file for integration. Using the embedded public key, the client-side SDK validates the accompanying signature,

decodes the payload, and recalculates the SHA-256 hash of its original content at runtime. This enables secure, fully offline validation. If the signature fails to verify, the license is immediately rejected. If the verification is successful, the SDK carries out further checks, such as making sure the license hasn't expired and comparing the declared bundle_id with the host application's identifier. The corresponding features are only activated when all requirements are met.

Strong protection for proprietary logic is provided by this division of duties, where the SDK enforces licenses and the vendor solely signs them. This keeps the architecture simple and network-independent, making it ideal for mobile environments.

Chapter 6

Conclusion and Future Work

6.1 Achievements

This thesis detailed the design and implementation of a secure, offline-capable licensing system developed for a cross-platform mobile framework dedicated to real-time physiological data analysis. The primary objective was to safeguard proprietary algorithms, focused on detecting conditions such as fatigue, sleep disruption, and alcohol misuse, while allowing their controlled deployment to third-party developers. Although the real IP algorithms were not implemented, but as safeguarded place holder for each was implemented. The system was engineered to balance robust intellectual property protection with ease of integration, ensuring both commercial viability and adherence to privacy standards.

To achieve these goals, a dual-platform architecture was implemented, comprising a Swift-based XCFramework for iOS and an Android AAR library. Both SDKs are designed to expose only the necessary interfaces for physiological prediction, with all proprietary logic securely encapsulated within precompiled binaries to deter reverse engineering. SHA-256 hashing, Base64 encoding, and RSA digital signatures were used to create a safe offline licensing system. The framework can verify the integrity and authenticity of license files using this cryptographic technique without the need for internet access.

The license generation tool, implemented as a script-driven utility, enables the software vendor to issue secure, parameterized license files tailored to individual clients. On the client side, both the iOS and Android SDKs integrate robust signature verification routines and enforce license checks at runtime. To further strengthen protection against tampering and unauthorized use, the system incorporates device-level security measures, such as jailbreak detection on iOS and root

detection on Android, aimed at identifying compromised environments.

When combined, these components show a full-stack approach to safe distribution of mobile SDKs. With a distinct division between proprietary algorithmic logic and third-party application code, the system effectively strikes a balance between technical security requirements and pragmatic limitations like offline operation and developer usability.

6.2 Limitations

The system has some drawbacks despite its strong technical foundation, especially because of its offline architecture. Its reliance on the local system clock of the device to ascertain whether a license has expired is one significant issue. This creates a potential loophole: users could intentionally set the clock back in an effort to extend license validity beyond its intended period. Since the framework is built to function entirely without server interaction, it lacks a direct mechanism to detect or prevent such manipulation. In the future, this restriction might be overcome by integrating reliable time sources, like timestamps from safe hardware components, or by creating a hybrid model that, when practical, allows for sporadic online validation.

A second limitation lies in the fact that license enforcement, though grounded in strong cryptographic principles, is performed entirely on the client device. This design choice inherently exposes the verification process to tampering by technically adept adversaries. The Kotlin-layer functions on Android, in particular, can be accessed using reverse engineering tools and modified to get around checks, such as by imposing return values that mimic legitimate licenses. Although offloading sensitive operations to native code via the NDK strengthens resistance to manipulation, purely client-side enforcement remains fundamentally vulnerable in the absence of hardware-backed security measures such as Trusted Execution Environments (TEEs) or server-side verification. In essence, while the current setup offers meaningful deterrence, it cannot guarantee absolute protection against sophisticated modification attempts.

6.3 Future Work

Several technically feasible and impactful enhancements are envisioned to extend this work. The enhancement of tamper resistance is the most important of these. More advanced code obfuscation techniques, internal integrity verification methods, and runtime detection methods to spot debugging or code injection attempts may

all be included in future iterations of the system.

A graphical user interface (GUI) or web-based license management system could be used in place of the current script-based license generation tool, which is another promising area for development. By allowing non-technical stakeholders to issue, manage, and revoke licenses without needing to understand the underlying cryptographic processes, such an interface would streamline operations. A web-based dashboard could further enhance usability by providing centralized oversight of all issued licenses, tracking their validity periods, and optionally capturing activation events in connected environments. Over time, this system could be extended to support more flexible licensing models, such as floating licenses, tiered feature sets, or usage-based metering.

Lastly, secure license update mechanisms could be included in later versions of the system, allowing vendors to remotely send notices of revocation, feature upgrades, or renewals. Secure channels, like digitally signed license update files or authenticated in-app update endpoints, could be used to distribute encrypted payloads. Crucially, such functionality would need to be designed in a way that maintains the framework's offline-first philosophy, ensuring that core runtime capabilities remain accessible even in the absence of connectivity, while still allowing dynamic updates when a secure network connection is available. This hybrid model would offer greater operational flexibility without compromising on the system's core security and usability goals.

Bibliography

[1] Grand View Research. *Wearable Technology Market Size, Share & Trends Analysis Report*. 2024. URL: https://www.grandviewresearch.com/industry-analysis/wearable-technology-market (cit. on p. 1).

[2] Fortune Business Insights. *Wearable Medical Devices Market Size, Share & Industry Analysis*. 2024. URL: https://www.fortunebusinessinsights.com/industry-reports/wearable-medical-devices-market-101070 (cit. on p. 1).

[3] F. Andriulo, M. Fiore, M. Mongiello, E. Traversa, and V. Zizzo. «Edge Computing and Cloud Computing for Internet of Things: A Review». In: *Informatics* (2024) (cit. on pp. 3, 5, 6).

[4] M. Elissen. *Fog Computing vs. Edge Computing: Their Roles in Modern Technology*. June 5, 2025. URL: https://www.akamai.com/blog/edge/fog-computing-edge-computing-roles-modern-technology (cit. on pp. 3, 5).

[5] L. Fourrage. *Edge Computing in 2025: Bringing Data Processing Closer to the User*. Feb. 24, 2025. URL: https://www.nucamp.co/blog/coding-bootcamp-full-stack-web-and-mobile-development-2025-edge-computing-in-2025-bringing-data-processing-closer-to-the-user (cit. on p. 4).

[6] Y.-A. Daraghmi, E. Daraghmi, R. Daraghma, H. Fouchal, and M. Ayaida. «Edge–Fog–Cloud Computing Hierarchy for Improving Performance and Security of NB-IoT-Based Health Monitoring Systems». In: *Sensors* (2022) (cit. on p. 4).

[7] P. Szczygło. *Beyond the Cloud: Pioneering Local AI on Mobile Devices with Apple, Nvidia, and Samsung*. Mar. 9, 2025. URL: https://www.netguru.com/blog/beyond-the-cloud-pioneering-local-ai-on-mobile-devices-with-apple-nvidia-and-samsung (cit. on pp. 6, 7).

[8] Thales Group. *SDK Licensing: Everything You Need to Know*. URL: https://cpl.thalesgroup.com/software-monetization/sdk-licensing (cit. on p. 8).

[9] A. Los. *Offline License Verifications*. July 2, 2024. URL: https://cryptolens.io/2024/07/offline-license-verifications (cit. on pp. 9, 16).

[10] B. Rocha. *Creating a Licensing System for Paid Apps in Swift*. Apr. 6, 2021. URL: https://swiftrocks.com/creating-a-license-system-for-paid-apps-in-swift (cit. on p. 11).

[11] 10Duke. *Handle and Store JWT License Tokens*. URL: https://docs.enterprise.10duke.com/developer-guide/consuming-licenses/handle-and-store-jwts (cit. on p. 11).

[12] Wacom. *SDK Licensing Overview*. URL: https://developer-support.wacom.com/hc/en-us/articles/9354475998103-SDK-Licensing-Overview (cit. on p. 16).

[13] Thales. *White Box Cryptography*. URL: https://cpl.thalesgroup.com/software-monetization/white-box-cryptography (cit. on p. 19).

[14] Apple. *WWDC 2019 – Secure Enclave White-Box Cryptography*. 2019. URL: https://developer.apple.com/videos/play/wwdc2019/416/ (cit. on p. 22).

[15] Android Developers. *Create an Android Library*. URL: https://developer.android.com/studio/projects/android-library (cit. on pp. 23, 35).

[16] McAfee. *What Is Jailbreaking?* URL: https://www.mcafee.com/learn/what-is-jailbreaking/ (cit. on p. 30).

[17] Apple. *Security of Runtime Process in iOS, iPadOS and visionOS*. Dec. 19, 2024. URL: https://support.apple.com/en-au/guide/security/sec15bfe098e (cit. on p. 31).

[18] Talsec. *How Can Mobile Developers Detect Jailbroken Devices?* May 2025. URL: https://docs.talsec.app/glossary/jailbreak-detection/how-can-mobile-developers-detect-jailbroken-devices (cit. on p. 31).

[19] Adjoe. *How Native Code Protects Sensitive Logic in Android*. Apr. 28, 2025. URL: https://adjoe.io/company/engineer-blog/mobile-app-security-android-native-code (cit. on pp. 35, 38).

[20] M. Stone. *VB2018 Paper: Unpacking the Packed Unpacker: Reversing an Android Anti-Analysis Native Library*. https://www.virusbulletin.com/virusbulletin/2019/01/vb2018-paper-unpacking-packed-unpacker-reversing-android-anti-analysis-native-library. 2018 (cit. on p. 38).

[21] A. Critelli. *Base64 Encoding: What Sysadmins Need to Know*. Aug. 10, 2022. URL: https://www.redhat.com/en/blog/base64-encoding (cit. on p. 44).