Impact of architecture of Native Modules on the performance of Hybrid Mobile Applications

Łukasz Kurant[1][0000−0002−2523−5952]

University of Maria Curie-Sklodowska, Pl. M. Curie-Skłodowskiej 5, 20-031 Lublin, Poland. lukasz.kurant@mail.umcs.pl

Abstract. React Native enables the creation of mobile applications using JavaScript, but to take full advantage of native platforms, effective communication between the JavaScript layer and native code is required. This article presents a detailed comparative analysis of selected native module architectures such as Turbo Modules, Nitro Modules, Expo Modules, and Weles Modules – a proprietary framework that extends integration capabilities with technologies not supported by React Native by default, while the Swarogen code generator automates the process of creating an intermediate layer, significantly reducing programming effort and minimizing the risk of implementation errors. The research conducted shows performance statistics for selected data types and operations, taking into account the communication architecture.

Keywords: native modules · react native · code optimisation · hybrid mobile applications.

1 Introduction

Among mobile applications, a hybrid approach is becoming a very common choice, allowing applications to be created for different platforms using as much shared code as possible. Currently, one of the most popular frameworks enabling such a solution is React Native, in which most of the code is written in JavaScript. Unlike other frameworks such as Ionic, React Native uses JavaScript to control native components. This architecture requires the cooperation of different technologies depending on the platform, so it is based on a multilingual paradigm, combining JavaScript with native languages for mobile platforms (e.g. Java/Kotlin for Android or Objective-C/Swift for iOS). Until recently, the main element of communication was the so-called *Bridge*, enabling asynchronous sending of serialized messages between two execution environments. However, the introduction of the new *JSI Architecture*, enabled by default since 2025 [16], has changed this communication model, allowing synchronous calls between JavaScript and the native part using C++ code. This change highlights the importance of choosing the language and architecture of native modules for the overall performance of the application.

The main goal of this article is to conduct a detailed comparative analysis of selected methods of communication between the JavaScript engine and native

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_18

modules. Another goal is to propose a unified architecture called *Weles* for modules in other languages, systematizing the multilingual approach and enabling convenient and quick creation of custom libraries based on other technologies using the *Swarogen* code generator.

1.1 Related work

React Native performance studies have been conducted since the framework's inception. In the article [2], the authors described the detailed impact of the application on the CPU, memory, and GPU, while the publication [6] presented the most important advantages of using React Native. The publication [12] provides a detailed comparison of hybrid frameworks, describing their most important features and pointing out the performance advantage of solutions using native UI components over WebView-based solutions. The authors in [11] presented the impact of the choice of JavaScript engine on the performance of hybrid applications and compared the old architecture using bridges with the new JSI architecture and its impact on rendering performance and application size. Other studies on the impact of JavaScript engine selection have also been conducted; in [4], the authors analyzed the impact of bytecode generated by the Hermes engine on application performance in React Native.

Research in the field of multilingual applications is also being conducted. The study [9] presents an approach to system modularization for a deeper understanding of applications. In [15], the author showed a way to interface high-level languages with low-level languages at runtime, which is easier to use than specification-based solutions.

Despite the growing number of publications, the literature still shows a gap in the impact of native module architecture and programming language selection on various aspects of hybrid application quality. This article aims to fill this gap by providing a comprehensive analysis of the relationship between architectural decisions and their impact on business logic. The proposed solutions also aim to demonstrate the possibility of integration with other technologies and indicate a unified way to integrate with them.

2 JavaScript Interface architecture

JavaScript Interface (JSI), introduced as part of the so-called *New Architecture* in React Native, replaces asynchronous Bridge with a synchronous interface, enabling direct calls to native functions from JavaScript and manipulation of JavaScript objects from native code. One of the most important features of this architecture is memory sharing – objects can be passed by reference without copying values. Formally, JSI is therefore a homomorphic function that enables memory sharing between two runtime environments. If J is the JavaScript environment space, N is the native space, function $\chi : J \to N$ maps objects from space J to N, and $\phi : N \to J$, then for subinterfaces $J' \subset J$ and $N' \subset N$, we can define JSI as:

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_18

$$\forall j \in J' : \chi(j) \in N' \text{ and } \phi(\chi(j)) = j \tag{1}$$

$$\forall n \in N' : \phi(n) \in J' \text{ and } \chi(\phi(n)) = n \tag{2}$$

The JSI architecture is based on the Host Function pattern, where native functions are registered in the JavaScript runtime as executable objects. When JavaScript code executes such a function, JSI translates the arguments to native types, executes the native implementation, and converts the result back to the JSI representation — all within a single synchronous call. Another part of this architecture are Host Objects, which are special JavaScript objects whose properties and methods are implemented on the native side. Each such object must meet certain requirements, i.e. implement the *HostObject* interface, which enforces the implementation of specific methods responsible for retrieving, setting, or removing object fields. The appearance of such an interface is shown in Listing 1.1. In JavaScript, the field key can be either a string or a symbol [5], and the value can be any type supported in JavaScript.

One of the most important aspects of JSI is the ability to install objects and functions directly in the global scope of the JavaScript runtime. This mechanism allows to create native APIs that are globally available without having to import modules. This allows for slightly better performance, at the cost of no validation and no layer to control the data being sent.

2.1 Turbo Modules and New Architecture

Turbo Modules is a new native module system built on the JSI foundation, replacing the older Native Modules API. The key innovation of Turbo Modules is lazy loading – modules are initialized only when first used, unlike the old system, where all modules were loaded when the application was launched. Such a module can have a native implementation in both C++ and another language supported by the platform. JSI and Turbo Modules are part of a larger architecture presented in Figure 1. Other parts of this architecture include:

- React Library: a library responsible for managing rendering in the JavaScript.
- JavaScript Engine: an engine that executes JavaScript code. The current standard is the Hermes engine, but integration with JavaScriptCore, V8, or other engines is possible [11].
- Codegen: a program responsible for automatically generating C++ code for the communication layer based on TypeScript types.

```
class HostObject {
  virtual ~HostObject();
  virtual jsi::Value get(jsi::Runtime&, const jsi::PropNameID&);
  virtual void set(jsi::Runtime&, const jsi::PropNameID&, const jsi::Value&);
  virtual std::vector<jsi::PropNameID> getPropertyNames(jsi::Runtime& rt);
}
```

Listing 1.1: Virtual *HostObject* class code in C++

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_18

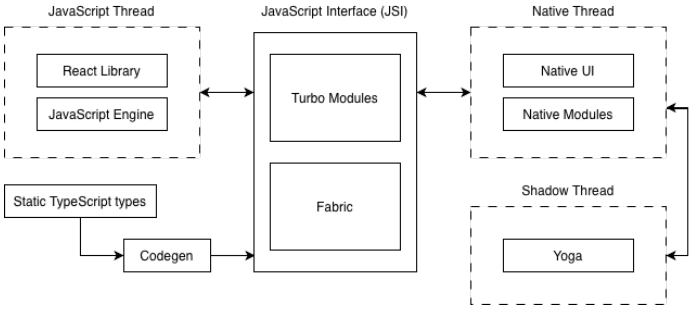


Fig. 1: Diagram of the new React Native architecture

– Yoga: an engine for calculations used in interface rendering, which works on separated thread.

2.2 Nitro Modules

Nitro Modules is a library for creating native modules in React Native, introduced in 2024, offering significantly better performance for operations requiring intensive data exchange, e.g. image and video processing, operations on large files, integration with Augmented Reality (AR) ot Virtual Reality (VR), or other operations requiring low latency [13]. Instead of serializing data and transferring it via JSI, Nitro Modules use shared memory and C++ smart pointers, allowing direct access to raw bytes from JavaScript and automatic object lifetime management. Unlike Turbo Modules, which use Host Objects, Nitro is built on *Native State*, i.e. native objects containing valid prototypes, and the size of such objects is known on the native side, which makes it easier for the garbage collector to remove unused objects [1]. Another important advantage is the Swift C++ interoperability layer [8], which allows Swift functions to be executed on iOS without going through the Objective-C layer.

3 Weles Modules

Weles Modules is our proprietary framework of native modules built on the top of JSI layer, enabling quick and easy creation of native modules for non-standard programming languages and integrating them with the React Native hybrid application. One of the main goals is to enable quick transfer of business logic to a mobile application without costly code rewriting, especially when using a non-standard runtime environment.

The prerequisite for using this framework is the ability to cross-compile the runtime environment in the case of interpreted languages or the code itself in the case of languages compiled for mobile device architectures. The consequence of this approach is a consistent communication interface with completely different languages and the use of code directly from JavaScript.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_18

```
interface LuaWelesModuleSpec extends WelesInterface<{language: 'lua'}> {
  setItem(value: string, key: string): number;
  processLargeArray(array: number[]): number[];
  heavyComputationAsync(): Promise<number>;
};

const LuaModule = getWelesModule<LuaWelesModuleSpec>('LuaWelesModule');
```

Listing 1.2: Example of a static TypeScript interface with method specifications for a module in the Weles architecture

3.1 Weles Architecture

The Weles library itself works based on Turbo module, installing a registry in the global JavaScript runtime state where installed modules are stored. The registry is a Host Object containing a map of modules and methods responsible for integration with them, such as *installModule()* or *uninstallModule()* to uninstall a module if necessary. Each module inherits from the *WelesModule* class, which is also a Host Object. This object contains methods that can be used from within the JavaScript environment. Figure 2 shows an example with the *LuaWelesModule* module containing integration with the Lua scripting language. This class contains all the necessary method implementations, and the *LuaWelesLinker* class is responsible for their correct installation, automatically performing linking when the module is first used. Modules can be loaded or removed as needed, allowing for better performance, especially when using more resource-intensive runtime environments.

It is important to note that the module does not need to be installed immediately after launching the application, but at a convenient time, which reduces the time needed to interact with the application (TTI), which is one of the main parameters used to evaluate a hybrid application.

In the case of scripting languages such as Lua, it is possible to inject code into the runtime environment while the mobile application is running. In such cases, it is sufficient to create an appropriate method in the module specification that accepts code in the form of a string and passes it to the runtime environment using the Weles architecture.

3.2 Swarogen

Swarogen is a generator that allows to create Weles Module code. It automatically creates a Linker class and a skeleton for implementing module methods based on the specifications provided in the TypeScript code. An example of such a specification is shown in Listing 1.2. Using the *getWelesModule()* method and passing such a specification creates a module and injects it into the Weles Modules registry. All basic Typescript types are supported, including the ability to create asynchronous methods. The generator itself was built in JavaScript based on the *ts-morph* library [14], which uses an Abstract Syntax Tree (AST) to create the required native equivalents of functions from the specification.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_18

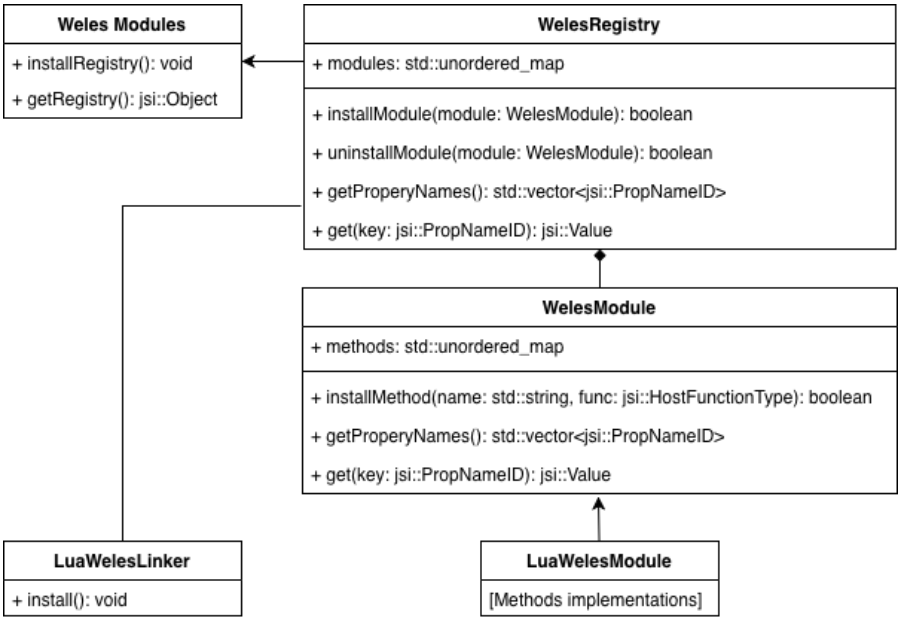


Fig. 2: Weles architecture diagram. The namespace *jsi* is used here for types such as *PropNameID*, which stores the object field key, or *HostFunctionType*, which is a reference to the Host function.

4 Methodology

The main goal of the experiments is to conduct a comparative analysis of selected communication architectures between JavaScript code and native technologies. To this end, we have prepared a series of tests to check performance in various contexts, from transferring data from one point to another, through performing simple and more advanced calculations, to processing asynchronous queries. Each of these experiments was prepared in native code along with the corresponding JavaScript code. To this end, the following solutions were tested:

- Expo Modules – modules for the Expo framework [3] built on React Native. These modules contain their own system of interaction with the application and are an alternative to Turbo modules for Expo. The technologies used are Kotlin (Android) and Swift (iOS).
- Turbo Modules – standard native modules in React Native. For testing purposes, projects were created for different types of communication: using code in C++, Kotlin (Android), Objective-C, and Swift with an interoperability layer to Objective-C (iOS). The case of code injected directly as Host Functions into the global Runtime object for JavaScript was also tested.
- Nitro Modules – modules using Kotlin (Android) and Swift with an interop layer for C++.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_18

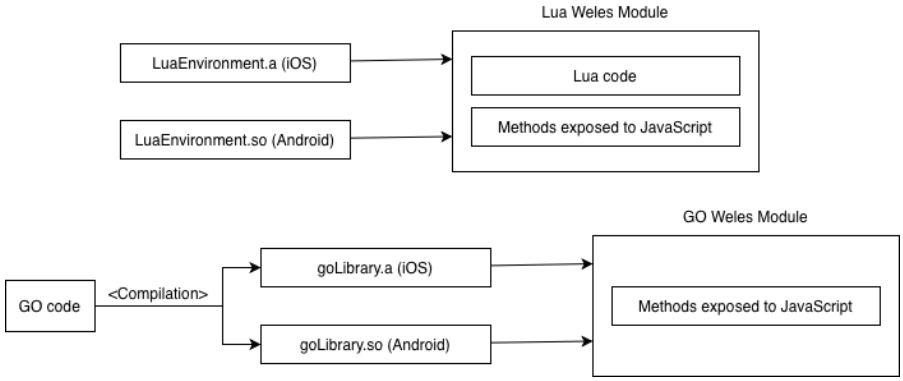


Fig. 3: Code architecture for Weles modules for Lua and GO languages

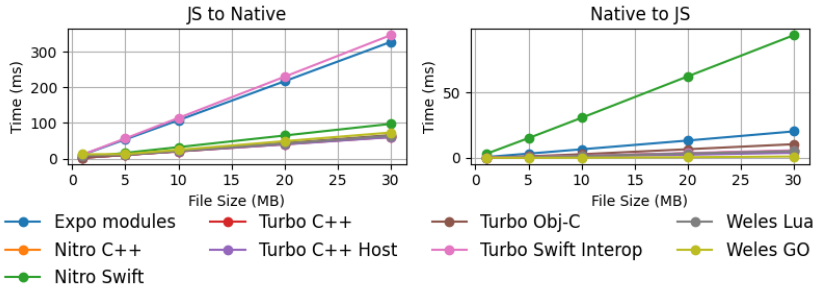
– Weles Modules – proprietary module architecture for non-standard languages. The final technologies selected were Lua (scripting language) injected directly into the runtime environment during module installation, and GO language, whose code was compiled during the compilation of the application itself, with functions called directly from the Weles library code (Figure 3).

The experiments were conducted on the following devices: iPhone 15 Pro (with an A17 Pro processor (6 threads), 8GB of RAM, and iOS 26.2) and Nubia Z70 Ultra (with a Snapdragon 8 Elite processor (8 threads), 12GB of RAM, and Android 15). Since background processes or thermal throttling can directly affect the accuracy of the results, we ensured a consistent runtime environment. The system ran without any additional applications running, with automatic updates disabled and battery saving mode turned off. In addition, we ensured cooling periods for the device by taking at least 5-minute breaks between successive test groups. Each test was performed 20 times. No significant differences in deviations between tests were observed, so there was no need to remove outliers. The arithmetic mean was calculated from the results, which became the final result.

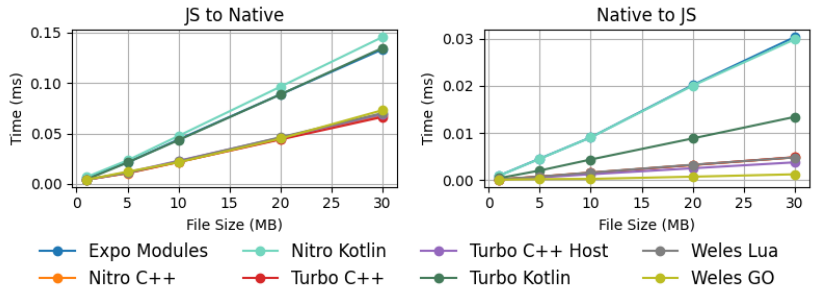
5 Experiments

The main benchmark we performed as part of the tests was to check the data transfer performance between solutions. To this end, we designed and created an experiment that measured the time needed to transfer data in both directions, i.e., first from JavaScript code to the final native code and vice versa. The transferred data consisted of character strings with sizes of 1, 5, 10, 20, and 30 MB. Detailed results are presented in the charts in Figure 4. In the case of iOS, Expo modules performed the worst when transferring data from JavaScript to the native module, along with the Turbo module with an interop layer between

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_18



(a) iPhone 15 Pro



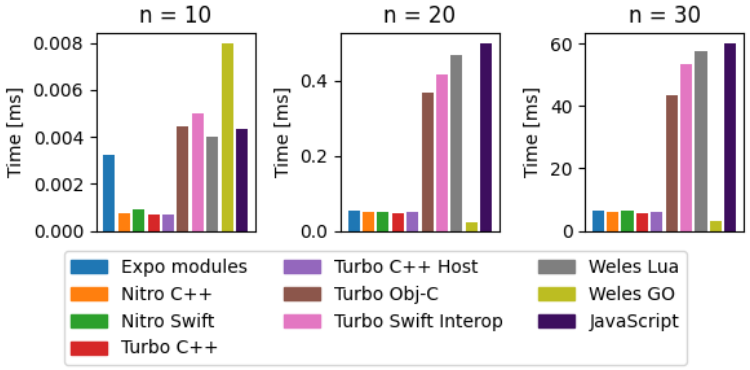
(b) Nubia Z70 Ultra

Fig. 4: Results of the experiment with data transfer in both directions

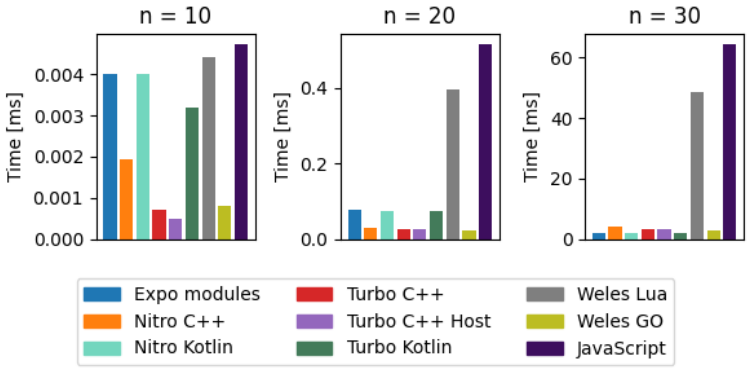
Objective-C and Swift. This is due to differences in data storage and the lack of a guaranteed continuous buffer in memory, which is particularly noticeable with larger data sets. In the case of Android, Kotlin-based modules are also not among the best, for similar reasons. The stage of data conversion between these languages and the C++ layer causes a significant drop in performance. Another conclusion is a more efficient solution using injected Host Functions to a global object in JavaScript Runtime – the lack of additional validation streamlines the transfer process, and although it does so only slightly, in the case of frequent function execution, it begins to matter.

Another experiment was to test a case where we can perform calculations using an algorithm that JavaScript may have more difficulty with. One such algorithm is the recursive calculation of the nth term of the Fibonacci sequence. Due to the fact that most engines still do not implement support for Tail Call Optimization (TCO) [7], in such cases it is worth using native-side calculations. The graphs in Figure 5 show the calculation time of the algorithm result for $n = 10$, $n = 20$, and $n = 30$. In this case, native modules clearly outperform code written in JavaScript, but it is GO in the Weles module that is able to perform the most aggressive optimizations, resulting in a significantly better result for larger n.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_18



(a) iPhone 15 Pro



(b) Nubia Z70 Ultra

Fig. 5: The result of the experiment of calculating the nth term of the Fibonacci sequence

Subsequent experiments involved the sequential transfer and addition of two numbers and two strings. The aim of the study was to examine the impact of data type representation on processing on the native module side. In each test, this task was repeated 10,000 times. The remaining series of experiments involved measuring other typical situations that mobile applications often encounter in a production environment when using native modules. The results of these experiments are presented in tables, broken down into data from the iPhone 15 Pro (Table 1) and Nubia Z70 Ultra (Table 2). Among these experiments, we can highlight the following:

– Photo blur – a photo is sent to the module in the form of a buffer of unsigned integers (in JavaScript, this is the *Uint8Array* type). The algorithm is supposed to accept the photo, transform it by performing a blur operation (for a kernel size of 3), and send it back in the form of a buffer.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_18

	Expo modules	Nitro C++	Nitro Swift	Turbo C++	Turbo C++ Host	Turbo Obj-C	Turbo Swift Int.	Weles Lua	Weles GO
String connection	24282	2386	4027	2308	**1165**	20941	19479	3254	4047
Number addition	20579	1514	1576	1426	**455**	11396	11701	1814	2257
Photo blur	149	180	212	167	**165**	167	190	1246	228
Process Array	3070	341	353	**311**	312	622	758	460	492
Allocate buffer	668	361	391	153	**152**	939	341	114285	338
Date manipulation	31214	**2726**	2799	3096	3069	19621	21786	3325	5370
Encryption	11641	12714	10187	11664	11661	15435	10998	589927	**8190**
Decryption	8656	12322	10660	11693	11671	11414	8480	582214	**8122**
Async computation	849	84	94	81	**63**	2509	1682	170	110

Table 1: Results of selected experiments for the iPhone 15 Pro device. Results are given in µs, with the best values in bold.

- Process Array – the module accepts an array of floating-point numbers and performs a simple transformation by increasing each element by a constant value, then returns the array in the same form.
- Allocate Buffer – the module allocates a 5 MB buffer of unsigned integers and returns it.
- Asynchronous Computation – the algorithm performs complex trigonometric function calculations but returns a Promise that is resolved after the calculations are complete. The test aims to use an additional thread to perform the calculations. Depending on the technology, it uses other available standard solutions, e.g., for C++ we use *std::thread*, *DispatchQueue* for Swift, or *Coroutines* for Kotlin.
- Date Manipulation – the aim of the experiment was to check how the native module would handle a *Date* object from JavaScript (based on the *Date* prototype). The module adds 24 hours to the date and then returns the *Date* object.
- Encrypt/Decrypt data – performing symmetric encryption, which combines several basic cryptographic operations, such as XOR operation with a key, left bit rotation, XOR with the encryption round value, and modulo key addition. The operation is performed byte by byte.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_18

	Expo modules	Nitro C++	Nitro Kotlin	Turbo C++	Turbo C++ Host	Turbo Kotlin	Weles Lua	Weles GO
String connection	7872	2469	6043	2220	**1266**	7640	7588	5432
Number addition	5582	1524	2445	1430	**510**	3398	5511	3421
Photo blur	145	141	323	132	**131**	139	610	228
Process Array	1964	282	313	**276**	279	1459	415	404
Allocate buffer	1959	482	2632	258	**189**	7704	82039	299
Date manipulation	26599	2727	5505	**3195**	3235	12070	4239	5370
Encryption	14888	10364	14656	10549	10250	16467	359081	**7630**
Decryption	15129	10509	14117	10197	10181	16581	357584	**7536**
Async computation	317	180	208	205	**165**	604	216	196

Table 2: Results of selected experiments for the Nubia Z70 Ultra device. Results are given in µs, with the best values in bold.

Experiments requiring greater interaction and direct action with the JSI layer show a significant advantage of C++ based modules, especially in the case of frequent integrations of those based on direct action on Host Functions. However, in some cases where time-consuming calculations are delegated directly to a native module, such as in the case of encryption and decryption, specific standard optimizations for the selected technology can result in significant acceleration (as in the case of the Weles module for the GO language).

6 Threats to validity

Our tests were conducted on iPhone 15 Pro and Nubia Z70 Ultra devices. The choice of devices can have a significant impact on the results, but in our tests we were guided by the desire to compare the performance of the application on both Android and iOS. The devices were selected because of their fairly popular component configuration (A17 Pro and Snapdragon 8 Elite processors). Each native module was tested in a separate application, completely isolated from the other modules. All tests were conducted in the standard configuration for the React Native framework version 0.83. Due to the intensive development of the framework, the choice of version may be of great importance in the context of the results. In our experiments, we used the Lua runtime environment version 5.5.0 and GO version 1.25.7. The other libraries and compilers would be standard for React Native.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_18

7 Conclusions

The choice of a native module will be important when performing certain types of tasks in hybrid mobile applications, and delegating more resource-intensive tasks from the JavaScript thread makes a lot of sense in mobile applications. Our research shows that it is possible to systematically integrate technology with a hybrid application, which we have achieved by designing our proprietary Weles Modules solution and the Swarogen code generator. Using this type of solution allows for the effective transfer of business logic to a hybrid application without the need to rewrite code for a different technology. For tasks that require extensive interaction with the JSI layer, the most efficient solution is to use C++ based modules, especially those that directly use Host Functions, at the expense of data validation. All our benchmarks have been made publicly available on our repository [10].

We also see many potential avenues for further research, including the impact of architecture choices on memory consumption and energy efficiency, which may be particularly relevant for weaker devices. It would be possible to attempt to integrate other languages or runtime environments, especially those implementing other paradigms such as Haskell (a functional language) or Mercury (a language implementing the logical paradigm).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Baets, P.D.: Hermes repository: Support nativestate in jsc engine (2023), https://github.com/facebook/hermes/pull/1151
2. Danielsson, W.: React native application development: A comparison between native android and react native (dissertation) (2016)
3. Expo: Expo: An open-source framework for making universal native apps with react, https://github.com/dsherret/ts-morph
4. Falvo, C.: Bn-hermes: a decompilation and analysis plugin for react native's hermes bytecode. In: Master thesis (2023)
5. Flanagan, D.: Javascript: The Definitive Guide. O'Reilly, Sebastopol, California, third edn. (2021)
6. Garg, P., Yadav, B., Gupta, S., Gupta, B.: Performance Analysis and Optimization of Cross Platform Application Development Using React Native, pp. 559–567. Springer Nature Singapore, Singapore (2023). https://doi.org/10.1007/978-981-19-9304-6_51
7. Harband, J., Rubanov, S.: Ecmascript 6 compatibility table (2025), http://compat-table.github.io/compat-table/es6/
8. Inc., A.: Mixing swift and c++ (2023), https://www.swift.org/documentation/cxx-interop/
9. Kargar, M., Isazadeh, A., Izadkhah, H.: Multi-programming language software systems modularization. Computers and Electrical Engineering **80**, 106500 (2019), https://www.sciencedirect.com/science/article/pii/S0045790618327095

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_18

10. Kurant, L.: Native modules in hybrid mobile applications repository (2025), https://github.com/lukaszkurantdev/native-modules-hybrid-apps
11. Kurant, L., Bylina, J.: Impact of selected javascript engines on the performance of mobile hybrid applications. Journal of Software: Evolution and Process **38**(2), e70086 (2026). https://doi.org/https://doi.org/10.1002/smr.70086
12. Nawrocki, P., Wrona, K., Marczak, M., Sniezynski, B.: A comparison of native and cross-platform frameworks for mobile applications. Computer **54**(3), 18–27 (2021). https://doi.org/10.1109/MC.2020.2983893
13. Rousavy, M.: What is nitro? (2025), https://nitro.margelo.com/docs/what-is-nitro
14. Sherret, D.: *ts-morph*: Typescript compiler api wrapper for static analysis and programmatic code changes (2025), https://github.com/dsherret/ts-morph
15. Toele, M.: Multi-Language Software Development with dix (2007)
16. Zaidman, V., Corti, N., Donadel, D.G., Houghes, A.: React native 0.82 – a new era (2025), https://reactnative.dev/blog/2025/10/08/react-native-0.82

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29909-3_18