Cross-platform Identity-based Cryptography using WebAssembly

Ádám Vécsi*, Attila Bagossy† and Attila Pethő‡
Department of Computer Science,
University of Debrecen
H-4028 Debrecen, Kassai str. 26.
*vecsi96@mailbox.unideb.hu, †bagossyattila@mailbox.unideb.hu, ‡Petho.Attila@inf.unideb.hu

***Abstract*—The explosive spread of the devices connected to the Internet has increased the need for efficient and portable cryptographic routines. Despite this fact, truly platform-independent implementations are still hard to find. In this paper, an Identity-based Cryptography library, called CryptID is introduced. The main goal of this library is to provide an efficient and open-source IBC implementation for the desktop, the mobile, and the IoT platforms. Powered by WebAssembly, which is a specification aiming to securely speed up code execution in various embedding environments, CryptID can be utilized on both the client and the server-side. The second novelty of CrpytID is the use of structured public keys, opening up a wide range of domain-specific use cases via arbitrary metadata embedded into the public key. Embedded metadata can include, for example, a geolocation value when working with geolocation-based Identity-based Cryptography, or a timestamp, enabling simple and efficient generation of single-use keypairs. Thanks to these characteristics, we think, that CryptID could serve as a real alternative to the current Identity-based Cryptography implementations.**

I. Introduction

Identity-based cryptography (IBC) is a novel branch of public-key cryptography. Although its foundations were established in 1985 by Shamir [1], who managed to build an identity-based signature (IBS) scheme, identity-based encryption (IBE) remained an open problem until Boneh and Franklin [2] created their pairing-based scheme in 2001, which was fast enough for practical use.

IBC's uniqueness lies in the fact that its public key is a string clearly identifying an individual or organization in a certain domain. Such string can be an email address or a username. The core purpose behind the IBC was to simplify the certificate management and eliminate the need for certification authorities. In a standard scenario, when employing the public key infrastructure (PKI), the key is bound to its user's identity with a public key certificate, however with IBC the user's identity is the public key itself, thus there is no need for a certificate. Despite this advantage, IBC still requires trusted third-party servers as private key generation and distribution can only be done by a so-called private key generator (PKG).

Even though there are some ideal use cases for IBC [3], [4], solution providers are not widely available. The most known one is the former Voltage Security Inc., now part of Micro Focus. This company provides a SecureMail service, which is based on IBE [5].

On the other hand, one can find several IBC implementations on the Internet. However, most of these libraries are focused on the desktop. Unfortunately, applications developed for the desktop can not be directly adapted for mobile devices, so in our opinion, this is a disadvantage, because nowadays there is an increasing number of mobile devices connecting to the Internet making use of apps or web-based services. Our motivation was to create a cross-platform, portable IBC solution targeting such devices.

One popular technology for development with such goals is JavaScript. One early library of IBC is WebIBC [6], which was developed in 2008 using JavaScript. The authors of the paper concluded that the web browsers and the JavaScript environment were not powerful enough, to implement a standard IBC library, which is based on pairing, because it is "too complex and overkill". Instead, they built a combined scheme, which requires much less computation-power and yet, the performance on a desktop was barely satisfying (1.5-2.5 seconds on average for encryption, using a 192-bit integer as the key).

Of course, since 2008, the performance of JavaScript engines significantly increased. An article written by a developer of the V8 JavaScript engine [7] points out that the performance of V8 quadrupled over the last ten years, which may inspire us to give a chance to implement a standard IBC library with the listed goals, using JavaScript. However, over the years a new technology, called WebAssembly came into the picture, which seems even more promising.

Developed by the W3C WebAssembly Community Group since 2015, WebAssembly is a virtual instruction set architecture, aiming to provide a basis for fast computations on the web, while also giving a solution which is embeddable into any environment [8]. Albeit being a quite young technology, already 86% of the internet users have a compatible browser enjoying its benefits [9]. Therefore, we can consider this technology as the perfect choice for the development of a cross-platform IBC solution.

In this paper, we will introduce our open source solution for a cross-platform IBC implementation using WebAssembly, which is available at https://github.com/cryptid-org. Our solution, called CryptID, is on the one hand small enough to be stored on a device with limited storage capacity, while, on the other hand, its performance is acceptable even on a device with

limited computational power. Our experiments proved that IBC based public key solutions are similarly efficient, as the PKI and also provides additional possibilities.

The rest of the paper is organized as follows. Section II contains some discussion about the relevance of IBC and WebAssembly and also describes Pairing-based cryptography and the standard IBC scheme. Our IBC implementation and its novelties are presented in Section III. Afterwards, Section IV outlines the performance of our library on multiple platforms. Section V gives a conclusion and contains some future development plans.

II. Preliminaries

A. Pairing

For $q = p^k$ with a prime p denote $\mathbb{F}_q$ the finite field with q elements. Let $E = E(\mathbb{F}_q)$ be an elliptic curve over $\mathbb{F}_q$. For the subgroup $G_1 \subseteq E$ the mapping $e : G_1 \times G_1 \mapsto \mathbb{F}_{q^\ell}$ with $\ell \geq 1$ is called *pairing* if

bilinear: For all $P, Q \in G_1$ and for all $a, b \in \mathbb{Z}_q^*$ we have $e(aP, bQ) = e(P, Q)^{ab}$.

non degenerated: If P is a non-zero element of G_1 then $e(P, P)$ generate $\mathbb{F}_{q^\ell}$.

Pairing is a rich theory and has numerous applications in cryptography, see the book of Cohen et al [10]. Its first celebrated application is due to Menezes, Okamoto and Vanstone [11], who proved that for hyperelliptic curves the discrete elliptic logarithm problem can be reduced in polynomial time to a discrete logarithm problem. To prove this result they used the efficiently computable Weil pairing. To avoid technical difficulties we do not define the Weil pairing, but refer to the paper of Boneh and Franklin [2]. There you may find not only the exact definition of the Weil pairing, but also its application to the identity based cryptography.

B. Identity-based Cryptography

In a public-key cryptography system, one very important task is key management. Nowadays, it is mostly handled by the PKI, which seems to work well, however, it has some shortcomings. In the white paper published by Micro Focus International plc [4] six important requirements are specified for enterprise key management.

- Deliver encryption keys.
- Authenticate users and deliver decryption keys.
- Jointly manage keys with partners.
- Deliver keys to trusted infrastructure components.
- Recover keys.
- Scale for growth.

The paper also clearly points out the shortcomings of the PKI. In many ways, it is difficult to use, implement and manage. This difficulty mainly comes from the need of maintaining enormous databases, which can be compromised or damaged, leading to severe data breaches or data loss. Additionally, maintaining such databases can get very expensive.

IBC may offer an obvious solution to these problems. IBC is a type of public-key cryptography in which the public key is a string clearly identifying an individual or organization in a certain domain. It is important to mention that not just the identifier can be arbitrary, but also the domain which specifies the scope of the identifier. This domain can be a global or even a local one, with only a few people in it.

The attractiveness of IBC comes from the previously mentioned properties of the public key, making it possible to establish systems without certification authorities and with simpler key management. Thus, IBC satisfies all six requirements in a cost-effective and user-accessible way.

From the point of this paper, two applications of IBC are relevant, encryption (IBE) and digital signature creation (IBS). Figure 1 shows how a standard IBE scheme works. The main participants are as follows: those want to exchange encrypted messages with each other and a third party, which handles the authentication and the private key generation. For authentication purposes, any already deployed resource can be reused, since this aspect is not limited by the scheme itself. The private key generation is performed by a trusted third party called the Private Key Generator (PKG).

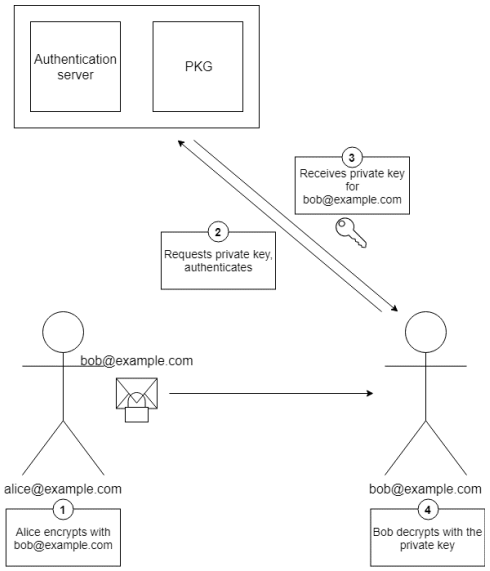

Fig. 1: How IBE works

The IBE scheme is based on four algorithms.

- *Setup*. Responsible for the initialization of the system. It generates the public parameters of the system and the master secret.
- *Extract*. This is the algorithm for calculating the private key from the public parameters, the user's identity, and the master secret.
- *Encrypt*. The algorithm for message encryption. It produces a ciphertext from the public parameters of the system, the public key and the plaintext message.
- *Decrypt*. The algorithm for message decryption. It uses the public parameters of the system, a private key generated by the PKG and an encrypted message.

It should be noted, that *Encrypt* and *Decrypt* are the inverse of each other. This means, if the message space is $\mathcal{M}$, then $\forall M \in \mathcal{M} : \mathit{Decrypt}(\mathit{Encrypt}(M, ID), sQ_{ID}) = M$, where ID is the user's identity and sQ_{ID} the corresponding secret key.

There are already multiple types of implementations of the IBE. The most popular variants are based on factoring, discrete logarithm or pairing. The first implementation worth mentioning is the Cocks IBE [12], which is based on integer factorization and the quadratic residuosity problem. Unfortunately, this solution produces long ciphertexts and suffers from long runtimes, rendering it inadequate for practical use. The real break-through came with the Boneh-Franklin IBE [2], which is based on pairing. This scheme is appropriate for practical use, but there exist other well-known IBE schemes with better performance, such as the Boneh-Boyen IBE [13], Sakai-Kasahara IBE [14] and TinyIBE [15].

The standard IBS scheme is similar to the IBE scheme. The first schemes were based on factoring or RSA [1], [16], [17], but these were not practical. Nowadays the ones based on pairing seem to be the preferred schemes, for instance [18], [19] to name a few.

To assess the security provided by the schemes, Bellare, Namprempre and Neven compared most of the existing IBS schemes with their framework [20].

C. WebAssembly

In this section, we would like to provide a short overview of WebAssembly, which is the key technology behind our work.

During the past decades, the Web has become a ubiquitous application platform, allowing developers to target a huge audience of users in a platform-independent manner. Thus, one can see more and more use cases for the Web platform, even in computation-intensive niches such as games, computer-aided design or audio and video manipulation software. On the other hand, efficient and at the same time, secure code execution remained an issue: technologies such as ActiveX [21], PNaCl [22] or asm.js [23] failed to consistently deliver these properties.

Therefore, a new Web specification, WebAssembly has born, aiming to securely speed up code execution on the Web [24]. The design goals of WebAssembly revolve around two key points, semantics and representation [8].

1) Design Goals: Regarding semantics, WebAssembly aims to execute code with near-native performance in a safe, sandboxed environment, while being hardware-, language- and platform-independent. As WebAssembly can be seen as a compilation target, language-independence means the lack of a privileged programming or object model.

Considering representation, WebAssembly offers a compact, modular binary format, that can be efficiently decoded, validated and compiled. The specification also considers streaming and parallel compilation of modules.

2) Targeting WebAssembly: Several popular programming languages offer WebAssembly as a compilation target, such as C/C++, Rust or C#. As our library is written in C, here we would only cover targeting facilities for that case.

Emscripten is a compiler toolchain built on top of LLVM, that can create WebAssembly modules from C and C++ source files [25] [26]. It should be noted, however, that Emscripten is capable of much more than simply emitting WebAssembly modules. As the browser (which is the main target of Emscripten) is a vastly different environment than the one assumed by most C applications, Emscripten offers the Emscripten Runtime Environment including, for example, a virtual file system, libc and libcxx implementations and tailored input and output handling.

3) Embedding WebAssembly: Despite its name, WebAssembly was designed considering server-side deployments from the ground up. The specification explicitly states openness as one of its design goals, which is achieved by providing a small, well-defined interface between the host environment and the WebAssembly semantics. Since the birth of the specification, several server-side runtimes have appeared, such as Lucet [27] or Wasmer [28].

Regarding Emscripten, we have previously highlighted, that it provides its own runtime environment in the browser. As the WebAssembly specification does not cover interfacing with system resources (such as files), currently each host has to define its own, incompatible runtime environment. In the future, this is going to change, since the WebAssembly System Interface (WASI) specification aims to cover this area [29].

III. CryptID

In most of the cases, it is not obvious how to implement a reliable cryptosystem, even if a mathematically proved secure cryptography protocol is available to build on. During the implementation, it is easy to make mistakes that open vulnerabilities. These vulnerabilities could come from programming negligence (incorrect input validation), or mathematical inattention, ignorance (using unsafe elliptic curves).

Most of the mistakes could be prevented, by using the standards during the implementation. In the case of IBC, multiple standards assist and guide the implementation [30], [31], [32], [33], [34], [35].

This section of the paper is about our solution, called CryptID, which is, in brief, an IBC implementation based on the RFC 5091. Nevertheless, it is not just a usual implementation, its novelty can be approached from two directions.

The novelty in the implementation is that CryptID is based on WebAssembly. Thanks to this property, CryptID is able to work on both the server-side and the client-side, or even completely separated from the web, providing a truly cross-platform and efficient IBC solution.

The novelty in the IBC scheme can be found in the public key. CryptID uses structured public keys, which may contain any kind of metadata with the identity string. This opens up many kinds of domain-specific opportunities. For example, if the current time is part of the metadata, then the keypair is devised for one-time use.

A. Cross-platform operation

With CryptID we wanted to create a library which provides efficient client-side IBC mainly targeting web browsers. Furthermore, we intended to implement a solution that can be used on the server-side, and even on IoT and alike. The motivation behind this was that we did not know about any open source IBC implementation which is out-of-the-box compatible with these platforms.

Earlier, the only technology which was able to serve these needs, was JavaScript. Unfortunately, JavaScript is far from ideal regarding the performance of computation-intensive tasks, which is just made worse by the fact that every browser has different optimizations, meaning, if something runs fast in one browser, it may be slow in the other. One solution for this problem is asm.js [23], which is a carefully chosen, easy-to-optimize subset of JavaScript. However, asm.js is not a well-established standard.

WebAssembly, on the other hand, is a great choice for projects like CryptID, because it is designed from the ground up as a performance and secure target platform. Moreover, WebAssembly made it possible for us to use GMP [36] as our arbitrary-precision arithmetic library, providing a stable, thoroughly tested foundation to our library.

B. Structured public key

As it was written earlier, the essence of IBC is that the public key clearly identifies an individual in a certain domain. D. Boneh and M. Franklin in their work [2] mentioned that public keys are expandable with any kind of metadata. It can be, for example, a year, which assigns a limited period of validity to the public and private keys. In that paper they simply concatenate the metadata to the identifier:

"bob@company.com $\|$ *current-year"*

In our opinion, this is a brilliant idea, with one serious flaw. By using concatenation, flexibility suffers greatly: everything needs to be in a fixed order. Of course, this cannot be changed, as the public key needs to be the same on the bit level both at encryption and extraction time. Still, we can accept arbitrary-ordered JSON documents from the clients of our library. The idea is simple: as the order of keys in a JSON object does not carry any meaning, we are free to reorder them. When given a public key, we always use the same key-sorting algorithm, making it possible to feed bit-accurate public keys to the lower layers of our library. Thus, the clients of CryptID do not need to worry about the way they structure the public key.

C. Library structure

CryptID can be didived into two main components: CryptID.wasm, which is a WebAssembly module, containing the IBC routines. The source code is written in C and is compiled to WebAssembly via Emscripten. The second part of the library is CryptID.js, which is a wrapper on top of the WebAssembly module, written in JavaScript. It provides an easy to use interface for the developers.

The library formed by these parts can, in turn, be divided into five smaller layers, shown in Figure 2.

Elliptic-curve arithmetics: Most of the popular IBC schemes are based on elliptic-curve cryptography, so the core part of our library is the elliptic-curve arithmetics. The reason behind writing our implementation is that we could not find a third-party solution that is well-tested and compilable to WebAssembly.

This layer is optimized to Type-1 curves, as recommended in the RFC 5091. The class of curves of Type-1 is defined as

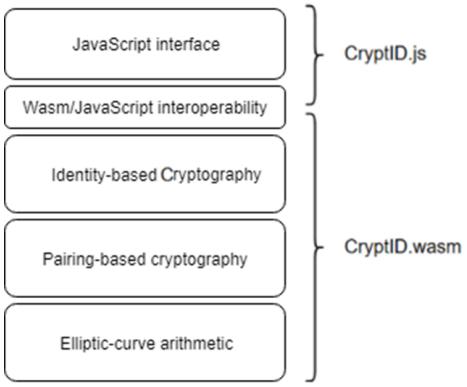

Fig. 2: The structure of CryptID.

the class of all elliptic curves of equation $E(\mathbb{F}_p) : y^2 = x^3 + 1$ for all primes $p \equiv 11 (\mathrm{mod}\ 12)$. This class forms a subclass of the class of supersingular curves.

To represent big numbers, we are using the GMP [36] arithmetic library. The elliptic-curve points are represented on the affine plane. The layer contains only the necessary methods, namely point doubling, point addition and point scalar multiplication.

Pairing-based cryptography: The majority of IBC protocols are using the pairing operation. As the RFC 5091 recommends, this layer provides a custom implementation of the Tate pairing. The implementation maps two points of $E(\mathbb{F}_p)$ to an element of $\mathbb{F}_{p^2}$. The implementation is based on Lynn's dissertation [37] and Martin's book [38].

Identity-based cryptography: The IBC routines are embedded into this layer. It includes the implementation of the Boneh-Franklin IBE [2] and the Hess IBS [18]. It also contains some miscellaneous helper functions, like an SHA implementation based on the RFC 6234 standard [39], and other hash functions based on SHA.

Wasm/JavaScript interoperability: The previous three layers together form an IBC implementation, which can even be consumed by native applications, without the need for WebAssembly compilation. However, when targeting the Web, these layers are hidden behind a JavaScript interface, as a WebAssembly module.

This abstraction is not absolutely necessary, because the WebAssembly embedding environment allows to call the functions of the module directly. However, CryptID is designed for those embedding environments, which grant the WebAssembly module calls via JavaScript. Such environments include web browsers or Node.js. This way CryptID can be called like any other JavaScript library rendering WebAssembly a simple implementation detail for clients.

To support this approach, it is crucial to implement an interoperability layer, which is responsible for the following tasks.

- Wrapping the C functions. To make the C functions callable from JavaScript, they need to be wrapped with the *cwrap* function of Emscripten's *Module* object.
- Conversion between datatypes. In our case two conversions were necessary, in both cases back and forth. The

first one is the converson of GMP's *mpz_t* big number type to JavaScript strings. The other one is the conversion betwen raw C byte arrays and JavaScript *ArrayBuffer*s.
- Bidirectional dataflow. It is not possible to use complex types like structs as parameters or return values, so the only way to exchange values is to copy between the isolated memory spaces accessible to JavaScript and to WebAssembly.

JavaScript interface: The JavaScript interface is a group of functions and datatypes which are public for outside clients. While previous layers can all be seen as implementation details, this layer is the actual interface that clients may consume.

Besides being a facade to lower layers, this layer has further responsibilities.

- Input validation. Being the public interface of the library, this is the only point where invalid input may enter into the system. Such values include *null* values or objects and strings with incorrect structure. Thanks to the validation performed by this layer, malformed values cannot form the basis of any computation.
- Key conversion. One of CryptID's novelties was the structured public key, which is able to contain any kind of metadata. Currently, structured public keys are handled in the interoperability layer, as JSON values. As there can be multiple JSON representations of the semantically same information, it is an important task to always convert JSON strings with the same content to the same bitstream. Our solution is to first create a new JavaScript object from the JSON string, with keys added in alphabetical order. Afterwards, `JSON.stringify` is called on this object to produce a new JSON string. Since `JSON.stringify` is guaranteed to preserve the original key addition order when producing JSON documents, we will always get the same bitstream from documents with the same keys. Thanks to this solution the lower layers do not need to know anything about the structure of the public key.

D. Development with CryptID

In this section, we present the basic usage of the library in the form of a small JavaScript program. Our goal is to illustrate the simplicity of integrating CryptID. The functionality of the sample code is as follows: First, it generates new public parameters and the master key. Then, a simple message is encrypted with an e-mail address being the public key. After the encryption, the private key corresponding to the public key is extracted, and lastly, the ciphertext is decrypted. Please be aware, that the sample code does not contain any error handling for better readability.

@cryptid/cryptid-js exports an instance of the *CryptID-Factory* interface, which contains only one function, namely *getInstance*. This is factory function, capable of creating *CryptID* instances asynchronously.

In the example, we are using a fixed message and public key, therefore they appear in the code as *const* values. The public key is a JavaScript object, which, for the sake of simplicity, does not contain any metadata this time, only an identifier, which is an e-mail address. We would like to note, that this example also leaves out authentication, in order to stay short.

After importing and obtaining a *CryptID* instance, one can use the standard IBE methods. First, we need to set up the system with the *setup* function, which generates the public parameters and the master secret. This function has only one parameter, the security level (`lowest`, `low`, `medium`, `high` and `highest`, based on the RFC 5091).

Now that we generated the public parameters, we can call the *encrypt* function with the previously specified public key and message. The resulting object contains a *ciphertext* field with the encrypted message in it.

To decrypt the ciphertext, we first need to acquire the private key via the *extract* function. In practical use, this is the point, where public key validation and authentication should be performed. As CryptID is only concerned with the encryption itself, these matters are left to the clients.

When the private key extraction is finished, the *decrypt* function can be called to obtain the original message from the ciphertext.

```
1  const CryptID = require('@cryptid/cryptid-js');
2
3  const message = 'Two hashes walk into a bar, one was a
       salted.';
4  const identity = {
5      email: 'flashandchill@example.com'
6  };
7
8  (async function main() {
9      const instance = await CryptID.getInstance();
10
11     const setupResult =
           instance.setup(CryptID.SecurityLevel.LOWEST);
12
13     const encryptResult =
           instance.encrypt(setupResult.publicParameters,
           identity, message);
14
15     const extractResult =
           instance.extract(setupResult.publicParameters,
           setupResult.masterSecret, identity);
16
17     const decryptResult =
           instance.decrypt(setupResult.publicParameters,
           extractResult.privateKey,
           encryptResult.ciphertext);
18
19     instance.dispose();
20 })();
```

IV. Performance

In the next section, the performance of the CryptID library is covered. We ran several benchmarks in multiple different environments while exercising the most performance-critical parts of the codebase. Where appropriate, we also compared the performance of our solution with the native version of other, well-established libraries.

First, we briefly outline the benchmark environments, which is followed by a detailed description of the performed experiments and their results.

A. Environments

Proving the platform-independent nature of our library, we aimed to benchmark it on a variety of platforms and

WebAssembly embedders. On the desktop, we performed experiments in three different embedders (Mozilla Firefox, Google Chrome, Node.js), and we also included the performance of the native version of the library as a baseline result. The exact hardware and software specifications can be seen in Table I. Regarding the mobile, we executed measurements in a single embedder (Google Chrome). Detailed specifications are available in Table II.

On all platforms, we used the Google Benchmark library [40] for our experiments. An experiment comprises twenty performance tests, where each test contains multiple executions of the same code on the same input. The result of the experiment is calculated as the average of the execution times. Inputs were chosen randomly for the four RFC defined security levels shown in Table III. Here, p is the order of the base finite field, the elliptic curve is defined over, while k stands for the RSA keylength providing comparable security [41].

Parameter	Value
Model	Dell Inspiron 5567 (2017)
CPU	i7-7500U, 2,7 GHz
OS	Ubuntu 16.04.4 LTS
emscripten	1.38.8
gcc	5.4.0 20160609
Node.js	v8.9.1
Firefox Quantum	62.0.3

TABLE I: Desktop hardware and software configuration.

Parameter	Value
Model	Nokia 6.1 TA-1043
CPU	Qualcomm Snapdragon 630, 2.2 GHz
OS	Android 8.1.0 - Kernel 4.4.78-perf+
Chrome for Mobile	68.0.3440.91

TABLE II: Mobile hardware and software configuration.

Security Level	p **bitlength**	k
LOWEST	512	1024
LOW	1024	2048
MEDIUM	1536	3072
HIGH	3840	7680

TABLE III: Security levels and values for appropriate parameters as stated in RFC 5091.

B. Benchmark Results

1) Elliptic Curve Scalar Multiplication: Considering the elliptic curve arithmetics, the scalar multiplication of elliptic curve points is a key operation to IBE. This operation can be found in a vast number of libraries, from which we chose MIRACL [42] and PARI [43] for our benchmarks because these are well-known and thoroughly tested solutions. In our experiments, we compared the performance of four different configurations: native MIRACL, native PARI, native CryptID and CryptID WebAssembly (Node.js).

In Figure 3, we graphed the benchmark results on a logarithmic scale, where the vertical axis represents the runtime in nanoseconds. It is clear, that MIRACL is several magnitudes faster than any other solution. On the other hand, we would like to highlight, that the native version of CryptID is just a little behind PARI, which is promising. Being this close results from the same choice of algorithm (double-and-add) and arithmetic library (GMP). In the case of the WebAssembly version, a somewhat consistent performance penalty can be noticed, compared to the native version. As the specification and the implementations mature, we expect this gap to decrease gradually.

2) Encrypt: CryptID was primarily designed for client-side use cases, where encryption and decryption take place on the user's device. Optimizing the performance of these operations is crucial, as we expect them to executed be frequently in a wide variety of browsers and devices. Thus, we first measured the runtime of the encrypt method in four different configurations: desktop Node.js, desktop Firefox browser, desktop Chrome browser, and again, desktop native as a baseline.

The logarithmically graphed results can be seen in Figure 4, where the vertical axis represents the runtime in milliseconds. On the desktop, the same sized performance gap is present between the native and WebAssembly versions, as in the case of elliptic curve scalar multiplication. Executing the WebAssembly code is consistently three to four times slower than the native program. However, we were quite surprised to discover, that our experiments took approximately the same time to finish in Node.js and Firefox, considering

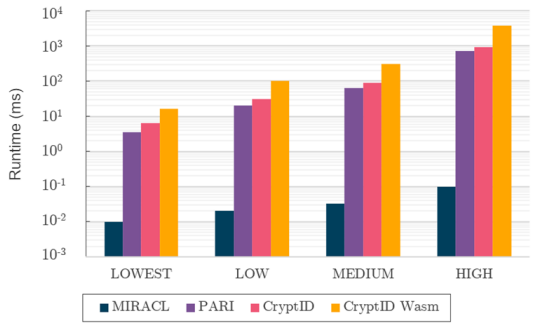

Fig. 3: Performance comparison of elliptic curve scalar multiplication solutions.

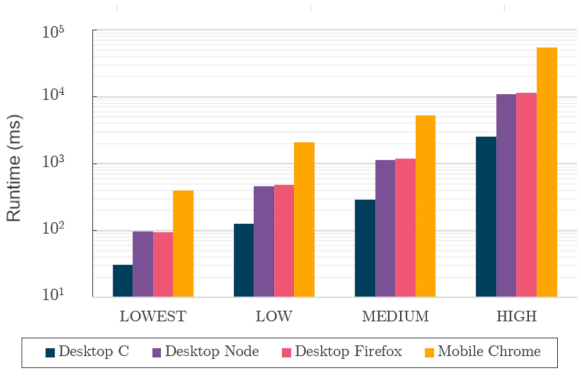

Fig. 4: Performance comparison of the *encrypt* function in across various environments.

the difference between the WebAssembly runtimes of these environments.

Unfortunately, encryption on the mobile is around four to five times slower than on the desktop. The difference results from the gap between desktop and mobile computational power. In spite of that, execution time of the `low` and `medium` security level can still be considered acceptable in practice.

3) Components of Encrypt: After outlining and comparing the performance of the encrypt operation in different environments, we would also like to further break this operation down into smaller components. The pie charts of Figure 5 show the results of profiling an experiment on a `high` input in the desktop Firefox environment.

The run time of encrypt is impacted by the performance of the Tate pairing and the *HashToPoint* function. As it can be seen on the bottom right chart, the execution time of the latter is approximately equivalent to that of a single elliptic curve scalar multiplication. Regarding the Tate pairing, modular exponentiation and multiplicative inverse have the largest influence on the performance.

4) Decrypt: We also performed experiments on the decrypt function in the same configurations as in the case of encryption. Based on the previous tests, we already had an approximate expectation regarding the performance of this function.

The actual results are shown in Figure 6, graphed on a logarithmic scale. Just as expected, the native desktop version has the best performance, while desktop Firefox is on par with Node.js. The performance gaps are of the same size as observed in the case of encryption.

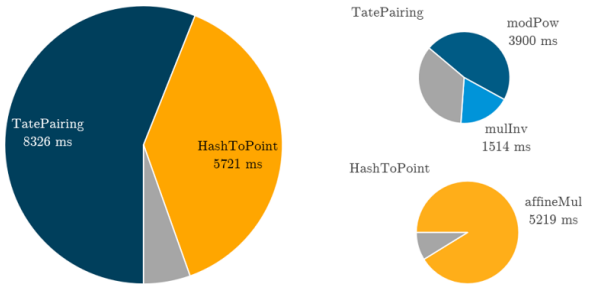

Fig. 5: Profiling results of a single *encrypt* execution on HIGH input in the desktop Firefox environment.

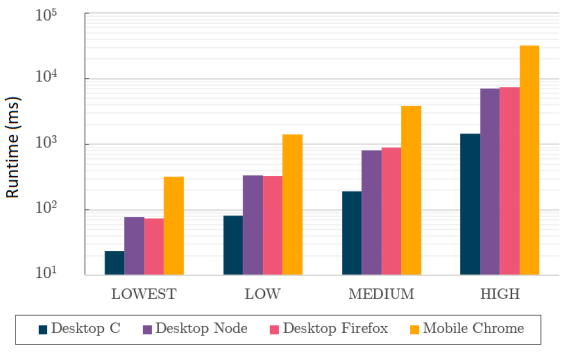

Fig. 6: Performance comparison of the *decrypt* function in across various environments.

Comparing the performance of decryption and encryption, one can notice, that *decrypt* runs for two-thirds of the run time as of encrypt. This is caused by the fact, that decrypt is equal to a single execution of the Tate pairing.

V. Conclusion and Future Work

We created a novel IBC library, which serves as a real alternative to the current implementations. One of the library's unique characteristics lies in its portability. Thanks to WebAssembly's platform-independent nature, CryptID offers an IBC solution on desktop, mobile, and IoT. The portability is combined with simple integrability, which makes for an appealing application development experience. CryptID also extends the already appealing identity-based public keys with optional metadata, in a structured, easy-to-manage way, giving an opportunity for many domain-specific use cases.

Several applications can be built on the above-mentioned novelties of the library. The client-side execution of the cryptographic functions provides a secure use of IBC, for example as a secure e-mail service. Besides the domain-specific options, structured public keys can also be used for the creation of single-use public keys, which is another important potential application of the library.

Considering future work, there are multiple possibilities to improve the performance of the library. With the implementation of better-performing arithmetic functions, we can optimize CryptID. Our main goal is to improve the elliptic-curve arithmetic layer, with the implementation of the Heuberger-Mazzoli scalar multiplication [44], and with some useful tricks, like precalculations.

Furthermore, another direction is the binary size reduction. Even though our bare library itself is lightweight, our dependency on GMP increases the linked binary size to a few hundred kilobytes. Unfortunately, dropping this dependency would cost us a lot of work, thus we are thinking of different approaches. Such an approach, for example, is called *tree-shaking*, which means the disposal of the unused code, potentially reducing the size of the linked binary even further.

Acknowledgment

This work was partially supported by the construction EFOP-3.6.3-VEKOP-16-2017-00002. The project was supported by the European Union, co-financed by the European Social Fund.

Research of the first and third authors was partially supported by the 2018-1.2.1-NKP-2018-00004 *Security Enhancing Technologies for the Internet of Things* project.

References

[1] A. Shamir, "Identity-based Cryptosystems and Signature Schemes," in *Proceedings of CRYPTO 84 on Advances in Cryptology*. New York, NY, USA: Springer-Verlag New York, Inc., 1985, pp. 47–53. [Online]. Available: http://dl.acm.org/citation.cfm?id=19478.19483

[2] D. Boneh and M. K. Franklin, "Identity-Based Encryption from the Weil Pairing," in *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '01. London, UK, UK: Springer-Verlag, 2001, pp. 213–229. [Online]. Available: http://dl.acm.org/citation.cfm?id=646766.704155

[3] J. Crampton, H. W. Lim, and K. G. Paterson, "What can identity-based cryptography offer to web services?" in *Proceedings of the 2007 ACM workshop on Secure web services - SWS '07*. ACM Press, 2007. [Online]. Available: https://doi.org/10.1145/1314418.1314424

[4] (2018) The Identity-Based Encryption Advantage. [Online]. Available: https://www.microfocus.com/media/white-paper/the_identity_based_encryption_advantage_wp.pdf

[5] (2018) On-Premises Email Encryption. Micro Focus. [Online]. Available: https://www.microfocus.com/media/white-paper/the_identity_based_encryption_advantage_wp.pdf

[6] Z. Guan, Z. Cao, X. Zhao, R. Chen, Z. Chen, and X. Nan, "WebIBC: Identity based cryptography for client side security in web applications," in *2008 The 28th International Conference on Distributed Computing Systems*. IEEE, Jun. 2008. [Online]. Available: https://doi.org/10.1109/icdcs.2008.24

[7] M. Bynens. (2018) Celebrating 10 years of V8. [Online]. Available: https://v8.dev/blog/10-years

[8] (2018) WebAssembly Specification. WebAssembly Community Group. [Online]. Available: https://webassembly.github.io/spec/core/_download/WebAssembly.pdf

[9] (2018) Can I use WebAssembly? [Online]. Available: https://caniuse.com/#feat=wasm

[10] H. Cohen, G. Frei, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren, Eds., *Handbook of Elliptic and Hyperelliptic Curve Cryptography, Second edition*. Chapman & Hall/CRC, 2012.

[11] A. Menezes, T. Okamoto, and S. Vanstone, "Reducing elliptic curve logarithms to logarithms in a finite field," *IEEE Transactions on Information Theory*, vol. 39, pp. 1639 – 1646, 1993.

[12] C. Cocks, "An identity based encryption scheme based on quadratic residues," in *In IMA International Conference*. Springer-Verlag, 2001, pp. 360–363. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-45325-3_32

[13] D. Boneh and X. Boyen, "Efficient selective-ID secure identity-based encryption without random oracles," in *Advances in Cryptology - EUROCRYPT 2004*. Springer Berlin Heidelberg, 2004, pp. 223–238. [Online]. Available: https://doi.org/10.1007/978-3-540-24676-3_14

[14] R. Sakai and M. Kasahara, "Id based cryptosystems with pairing on elliptic curve," 2003. [Online]. Available: http://eprint.iacr.org/2003/054

[15] P. Szczechowiak and M. Collier, "TinyIBE: Identity-based encryption for heterogeneous sensor networks," in *2009 International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*. IEEE, 2009. [Online]. Available: https://doi.org/10.1109/issnip.2009.5416743

[16] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Advances in Cryptology — CRYPTO' 86*. Springer Berlin Heidelberg, 2000, pp. 186–194. [Online]. Available: https://doi.org/10.1007/3-540-47721-7_12

[17] T. Okamoto, "Provably secure and practical identification schemes and corresponding signature schemes," in *Advances in Cryptology — CRYPTO' 92*. Springer Berlin Heidelberg, 2001, pp. 31–53. [Online]. Available: https://doi.org/10.1007/3-540-48071-4_3

[18] F. Hess, "Efficient identity based signature schemes based on pairings," in *Selected Areas in Cryptography*. Springer Berlin Heidelberg, 2003, pp. 310–324. [Online]. Available: https://doi.org/10.1007/3-540-36492-7_20

[19] P. S. L. M. Barreto, B. Libert, N. McCullagh, and J.-J. Quisquater, "Efficient and provably-secure identity-based signatures and signcryption from bilinear maps," in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 515–532. [Online]. Available: https://doi.org/10.1007/11593447_28

[20] M. Bellare, C. Namprempre, and G. Neven, "Security proofs for identity-based identification and signature schemes," *Journal of Cryptology*, vol. 22, no. 1, pp. 1–61, Aug. 2008. [Online]. Available: https://doi.org/10.1007/s00145-008-9028-8

[21] (2017) Introduction to ActiveX Controls. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/aa751972%28v%3dvs.85%29

[22] (2017) NaCl and PNaCl. [Online]. Available: https://developer.chrome.com/native-client/nacl-and-pnacl

[23] (2014) asm.js Specification. [Online]. Available: http://asmjs.org/spec/latest/

[24] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," *SIGPLAN Not.*, vol. 52, no. 6, pp. 185–200, jun 2017. [Online]. Available: http://doi.acm.org/10.1145/3140587.3062363

[25] A. Zakai, "Emscripten: An LLVM-to-JavaScript Compiler," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 301–312. [Online]. Available: http://doi.acm.org/10.1145/2048147.2048224

[26] A. Zakai. (2015) Big Web App? Compile It! [Online]. Available: kripken.github.io/mloc_emscripten_talk/

[27] (2019) Lucet, the Sandboxing WebAssembly Compiler. [Online]. Available: https://github.com/fastly/lucet

[28] (2019) Wasmer – Universal WebAssembly Runtime. [Online]. Available: https://wasmer.io/

[29] L. Clark. (2019) Standardizing WASI: A system interface to run WebAssembly outside the web. [Online]. Available: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

[30] X. Boyen and L. Martin, "Identity-based cryptography standard (IBCS) #1: Supersingular curve implementations of the BF and BB1 cryptosystems," *RFC*, vol. 5091, pp. 1–63, 2007. [Online]. Available: https://doi.org/10.17487/RFC5091

[31] G. Appenzeller, L. Martin, and M. Schertler, "Identity-based encryption architecture and supporting data structures," Tech. Rep., Jan. 2009. [Online]. Available: https://doi.org/10.17487/rfc5408

[32] L. Martin and M. Schertler, "Using the boneh-franklin and boneh-boyen identity-based encryption algorithms with the cryptographic message syntax (CMS)," Tech. Rep., Jan. 2009. [Online]. Available: https://doi.org/10.17487/rfc5409

[33] "IEEE standard for identity-based cryptographic techniques using pairings," 2013. [Online]. Available: https://doi.org/10.1109/ieeestd.2013.6662370

[34] "Iso/iec 18033-5:2015: Information technology - security techniques - encryption algorithms - part 5: Identity-based ciphers," 2015. [Online]. Available: https://www.iso.org/standard/59948.html

[35] "Iso/iec 15946-1:2016: Information technology - security techniques - cryptographic techniques based on elliptic curves - part 1: General," 2016. [Online]. Available: https://www.iso.org/standard/65480.html

[36] T. Granlund and the GMP development team. (2016) GNU MP: The GNU Multiple Precision Arithmetic Library. [Online]. Available: http://gmplib.org/

[37] B. Lynn, "On the implementation of pairing-based cryptosystems," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 6 2007. [Online]. Available: https://crypto.stanford.edu/pbc/thesis.pdf

[38] L. Martin, *Introduction to Identity-based Encryption*. Boston: Artech House, 2008.

[39] D. Eastlake and T. Hansen, "US secure hash algorithms (SHA and SHA-based HMAC and HKDF)," Tech. Rep., May 2011. [Online]. Available: https://doi.org/10.17487/rfc6234

[40] (2019) Google Benchmark - A microbenchmark support library . [Online]. Available: https://github.com/google/benchmark

[41] (2009) The Case for Elliptic Curve Cryptography. National Security Agency. [Online]. Available: https://web.archive.org/web/20090117023500/http://www.nsa.gov/business/programs/elliptic_curve.shtml

[42] MIRACL Cryptographic SDK.

[43] (2018) PARI/GP version `2.11.0`. The PARI Group. Univ. Bordeaux. [Online]. Available: http://pari.math.u-bordeaux.fr/

[44] C. Heuberger and M. Mazzoli, "Symmetric digit sets for elliptic curve scalar multiplication without precomputation," *Theoretical Computer Science*, vol. 547, pp. 18–33, aug 2014. [Online]. Available: https://doi.org/10.1016/j.tcs.2014.06.010