Bachelor's Thesis Computing Science

Verifying Yivi's attribute-based signatures

Luca Nieuwenhuijzen
s1113467

June 9, 2026

First assessor/supervisor:
Dr. A.A. Westerbaan

Second assessor:
Prof. Dr. B.P.F. Jacobs

Radboud University

Abstract

This thesis presents a TypeScript implementation for verifying Yivi's attribute-based signatures. Designed for web applications, this library enables developers to perform verification entirely within the user's browser. By eliminating the need for server-side processing, the solution ensures that sensitive user data remains local, thereby reducing operational overhead and having potentially some privacy benefits. The implementation is a manual port of the verification logic originally written in Go. This thesis details the verification system and the porting process, discussing the challenges encountered when translating from a server-side Go environment to a browser-based TypeScript environment. Finally, the library is evaluated for correctness, performance, and security, demonstrating its viability as a client-side alternative for verifying Yivi's attribute-based signatures.

Contents

Chapter 1

Introduction

Motivation

Digital identification is becoming increasingly important as most, if not all, administrative activities have moved online. Traditional methods often require users to share more personal information than necessary. For example, to watch age-restricted videos on YouTube, a user must verify their age with Google. This can be done by providing a picture of oneself, executing a credit card transaction, or, most significantly, uploading a copy of a government-issued ID [1]. In addition to the exact date of birth, an ID typically reveals a full name, place of birth, and other details that are entirely unnecessary for verifying age. This creates unnecessary privacy risks [2, 3, 4] and puts sensitive personal data in the hands of Google and third parties, who may not need access to such detailed information. Even worse, in the event of a data breach, this information can be leaked and used for identity theft or other malicious purposes [5].

Yivi (previously known as IRMA) addresses this problem using an attribute-based authentication system backed by zero-knowledge proofs [6]. Yivi allows users to disclose only the specific attributes (e.g., "is over 18") required for verification, without revealing the underlying personal information present on an ID (e.g., their exact date of birth) [7, 8]. These attributes can also be used to sign messages in a privacy-preserving manner [9]. For example, a doctor could share their medical opinion in an online forum and prove their medical credentials without ever revealing their full name. This combines authenticity with anonymity in a way that traditional digital signatures cannot achieve.

Currently, adoption of these signatures is limited by the fact that verification can only be performed server-side using the existing Go implementation [10] (referred to as `irmago` from here on), because no working and up-to-date client-side verification implementation existed. Consequently, anyone wishing to verify a Yivi signature must set up and maintain a dedicated backend

server to handle the verification process. This introduces unnecessary complexity, as it requires setting up and maintaining server infrastructure, and adds latency, since signatures must be sent over the internet rather than verified locally. Both factors hamper adoption. This thesis addresses the limitation by porting the signature verification code to TypeScript, enabling it to run directly in web browsers. It describes the porting process, the challenges faced, and evaluates the performance and security of the resulting browser-based implementation.

The central research question this thesis addresses is:

How can Yivi's attribute-based signature verification be correctly and efficiently implemented in a browser environment, and what are the performance and security trade-offs compared to the existing server-side Go implementation?

To answer this, the thesis first, in Chapter 2, describes the details of a Yivi attribute-based signature and the mathematical operations a verifier must perform to ensure correctness. Next, Chapter 3 describes the steps taken to port `irmago` to TypeScript. After that, the performance of the resulting library is benchmarked and compared to the original Go implementation in Chapter 4. Finally, in Chapter 5, the security and correctness of the port are evaluated.

Chapter 2

Preliminaries

2.1 Introduction to Yivi

Yivi implements the Idemix [8] identity scheme, is controlled by the non-profit Privacy by Design Foundation, and is developed by Caesar Groep [11]. Its selective disclosure rests on zero-knowledge proofs [6] (henceforth ZKPs), which let a user prove possession of an attribute without revealing the attribute itself or any other personal information. These ZKPs are also used to link multiple disclosed attributes to the same user [12]. This prevents a user from taking one attribute (e.g., being over 18) from another user's credential and combining it with their own attributes (e.g., their email address) to falsely bypass age restrictions on services.

2.2 Yivi's Attribute Model

Yivi's attribute models are hierarchically organised using identifiers that follow the structure:

`scheme.issuer.credential.attribute`.

An example of such an identifier would be:

`pbdf.PubHubs.pubhubsaccount.email`,
or
`irma-demo.MijnOverheid.ageLower.over18`.

In order from the most specific to the most general, the components of this structure are:

Attributes are the fundamental pieces of information that the three core actions (issuing, disclosing, and signing) operate on, these actions will be detailed in Section 2.3. These attributes can include concrete personal information such as name, date of birth, and address, but also potentially more abstract properties such as "is over 18" or "has a valid driver's license". All attributes are stored locally on the user's device in a secure manner akin to a digital wallet. Attributes are bundled together into groups called credentials.

Credentials function as digital documents, similar to a government-issued ID card or a student card. For example, a student card credential might contain attributes such as the student's name, student ID number, and the institution they are enrolled in. Credentials are valid only when issued by trusted entities known as issuers.

Issuers are the trusted entities responsible for verifying user information and providing the corresponding credentials. For example, a university can act as an issuer by issuing student card credentials to its students after verifying their enrollment. The issuer's digital signature on a credential ensures its authenticity [13]. Through the use of ZKPs, verifiers can trust the attributes without needing to contact the issuer directly or even seeing the signature itself [12]. Issuers are organised into schemes.

Schemes represent the highest level of organisation in Yivi's attribute model. A scheme is responsible for indexing all participating issuers, the public keys for said issuers, and the credential definitions that those issuers can use. Each scheme is controlled by a scheme manager, who signs the entire directory to ensure its integrity and authenticity. This means scheme managers have ultimate control over the structure of the scheme and can add or remove issuers as needed. Currently, two schemes exist: the public `pbdf` scheme managed by the Privacy by Design Foundation and Caesar Groep, and the `irma-demo` scheme used for demonstration purposes. These schemes are hardcoded into the Yivi app, though it can be modified to load custom schemes, allowing for flexibility and adaptability to different use cases and contexts [14].

2.3 Yivi's Core Operations

As mentioned before, Yivi supports three core operations: issuing, disclosing, and signing.

Issuance is the process by which a user obtains credentials from an issuer. After authentication, where the user typically provides some form of identification to the issuer who verifies the information, the issuer issues the credential requested by the user. These credentials are then stored securely in the Yivi app on the user's device, allowing them to then manage and control their personal information without relying on a central authority. For example, when a Dutch citizen wants to add personal attributes related to their identity, they might authenticate with DigiD [15] (a Dutch digital identification system) to prove their identity to the government[1]. Once the government verifies the user's identity, it issues a credential containing the user's attributes that are typically contained on an ID card such as name, date of birth, and any abstract attributes like "is over 18".

Disclosure is the process by which a user reveals specific requested attributes from their credentials to a service provider. This operation is designed to minimise the amount of information shared. For example, when a user needs to prove they are over 18 to access a service, they can disclose the "is over 18" attribute from their credential without revealing their exact date of birth or other personal information. This is a privacy-preserving way to authenticate compared to traditional methods that often require sharing an entire (digital) government ID containing far more information than necessary. This process resembles linking a service to a Google account, where the user can review which specific attributes (like email or name) are shared with the service. Importantly, Yivi improves upon this model by removing the need for a central authority (e.g., Google) that tracks and verifies these disclosures. Authentication via Yivi disclosures involves communication only between the service provider and the user [16].

Signing in the context of Yivi, is the process by which a user can sign a document or message using attributes from their credentials. This operation produces a signature that can be verified by others while still preserving the user's privacy by not requiring full identity disclosure [9]. For example, a tenant could sign a rental contract using their verified name and email address from their Yivi wallet. The landlord can then verify the signature to ensure the contract was indeed signed by the person claiming to be the tenant, without needing to copy and store the tenant's passport or ID card.

[1] In actuality the municipality of Nijmegen in this instance handles the authentication with DigiD and is the issuer of the credential

This acts identically to any other digital signature [9], but with the added benefit of not needing to share or store sensitive personal information, thus reducing the risk of data breaches and identity theft.

2.4 Mathematical Description of Yivi Signatures

The privacy-preserving nature of Idemix, which Yivi is an implementation of [16], is rooted in the Camenisch-Lysyanskaya (CL) scheme [17]. This section provides the mathematical foundation required to understand what a Yivi credential is, how it is obtained, and, most importantly, how a verifier can check its validity without learning any secret information. The section is organised in three parts. It first sets up the shared groundwork, the public key parameters and how a credential is issued, and then splits the signature itself into the two halves that matter in practice: creating a signature (section 2.4.3) and verifying one (section 2.4.4).

2.4.1 Public Key Parameters

All arithmetic in this section is performed modulo a large RSA modulus $n = pq$, where p and q are secret primes known only to the issuer. The CL signature scheme relies on the Strong RSA Assumption [18, 19], which underpins the security of the signatures and the zero-knowledge proofs used in Yivi. The Strong RSA Assumption states that:

Given a random element z in the multiplicative group of integers $(\mathbb{Z}/n\mathbb{Z})^\times$ and n, no efficient algorithm can find a pair (A, e) with $e > 1$ satisfying $A^e \equiv z \pmod{n}$, unless the factorisation of n is known.

This assumption is crucial for the security of CL signatures. It means that without knowing the factorisation of n (which only the issuer knows), it is computationally infeasible to find any pair of values (A, e) that satisfy a specific equation involving exponentiation modulo n. Later it will be shown that forging a credential would require constructing such a pair of values that fit the issuer's cryptographic scheme. The Strong RSA Assumption makes credential forgery computationally difficult for any attacker who does not possess the issuer's private key. Said private key consists of the prime factors p and q. From these the issuer can compute Euler's totient $\phi(n) = (p-1)(q-1)$, which later will be shown, in section 2.4.2, to be used to calculate $d \equiv e^{-1} \pmod{\phi(n)}$ which is used to issue credentials. Anyone without the factorisation cannot compute $\phi(n)$, and so cannot produce these signatures.

The issuer publishes a public key of a credential consisting of the modulus n together with a set of generators $(S, Z, R_0, R_1, \ldots, R_L) \in \mathbb{Z}_n^*$, where L is the number of generators chosen by the issuer. An example public key

is published in the `irma-demo` scheme manager, which is used to issue the `irma-demo.RU.studentCard` credential.

The generator R_0 is permanently associated with the user's master secret key sk. The generator R_1 is similarly permanently associated with the credential's metadata, which includes the credential's identifier, issuance date, and expiration date [20]. The remaining generators $(R_2, \ldots, R_L)$ each correspond to at most one of the credential's other attributes.

2.4.2 Credential Issuance

Issuance is an interactive process between the user and the issuer that produces a CL signature over the user's master secret sk and a set of attribute values $(m_1, \ldots, m_L)$.

To prevent the issuer from ever learning sk, the user first creates a *commitment*

$$U = S^{v'} \cdot R_0^{sk} \bmod n,$$

where v' is a large random value chosen by the user. This value serves as a *blinding factor*: a random value that masks the secret, preventing the issuer from seeing it or linking this issuance to any future use of the credential. This commitment ensures that the issuer cannot see what sk is and that the issuer 'commits' to the secret key and cannot remove or change sk later in the process. The user sends the commitment U to the issuer, who selects an additional blinding factor v'' and computes

$$Q = \left(U \cdot S^{v''} \cdot \prod_{i=1}^{L} R_i^{m_i}\right)^{-1} \cdot Z \bmod n.$$

The issuer then picks a random prime e of a public, pre-defined, bit-length ℓ_e, generates d such that it is equivalent to $e^{-1} \pmod{\phi(n)}$, and computes $A = Q^d \bmod n$. The partial signature (A, e, v'') is returned to the user, who forms the total blinding factor $v = v' + v''$ and stores the complete CL signature tuple (A, e, v). Note that the CL signature is only part of the final credential.

By construction, any valid credential satisfies the following identity, called the *credential equation*:

$$A^e \cdot S^v \cdot R_0^{sk} \cdot \prod_{i=1}^{L} R_i^{m_i} \equiv Z \pmod{n}. \tag{2.1}$$

This identity is the cornerstone of all subsequent proofs and their verification.

The result is a credential that can be trusted and used for disclosure and signing, even without a verifier ever needing to contact the issuer again. This is because the credential equation can be verified using only the public key

parameters without revealing *sk* or any attributes, which is called a proof of knowledge. How this proof of knowledge is created and shared without revealing the underlying information will be discussed in the next section.

2.4.3 Creating a Signature

We split signature handling into two parts: creation and verification. This subsection covers *creation*, where the signer uses their credential to generate a proof without revealing the secret values, which remain on the signer's device. The verification process, covered in section 2.4.4, is what the verifier performs to check the proof's validity.

Constructing the Zero-Knowledge Proof

When a user wishes to present a credential and disclose some, but not all, attributes, they must never reveal the credential-unique (and thus user-unique) signature tuple (A, e, v) directly. Doing so would allow a verifier to track the user's activity across different platforms, linking distinct disclosures back to the same identity and destroying unlinkability. There should also be a way to verify the credential without revealing the attributes that are not being disclosed. To achieve both these goals, the user instead constructs a non-interactive Zero-Knowledge Proof (ZKP) via the Fiat-Shamir heuristic, proving possession of a valid credential without exposing the underlying data or it being linked to a single user [21].
This process involves three steps, detailed below: randomising the signature, forming a commitment, and computing the challenge and responses.

Randomising the signature. The user picks a fresh random value r_{blind} and computes a randomised $A_{proof} = A \cdot S^{r_{blind}} \bmod n$, together with an adjusted blinding factor $v_{proof} = v - e \cdot r_{blind}$. Crucially for the signature's validity, the randomised signature still satisfies the credential equation:

$$A_{proof}^{e} \cdot S^{v_{proof}} = A^{e} \cdot S^{e \cdot r_{blind}} \cdot S^{v - e \cdot r_{blind}} = A^{e} \cdot S^{v}$$

So the credential equation (2.1) continues to hold with the new A_{proof} and v_{proof}. However, the verifier does not see the original A or v. This step is needed for unlinkability, as it ensures that each proof instance appears unique to the verifier, even if the same credential is used multiple times.

Generating the commitment. Let H denote the set of hidden attribute indices and D the set of disclosed attribute indices, with $H \cup \{0\} \cup D = \{0, 1, \ldots, L\}$ (index 0 represents *sk*, which is always hidden). The user selects fresh random values r_e, r_v, and r_{m_i} for each $i \in H \cup \{0\}$, and computes the commitment

$$T = A_{proof}^{r_e} \cdot S^{r_v} \cdot \prod_{i \in H \cup \{0\}} R_i^{r_{m_i}} \bmod n.$$

This commitment encapsulates the user's knowledge of the hidden attributes and the credential's signature, without revealing any of these values directly. This commitment will be used in the challenge to prove knowledge of the hidden attributes and the validity of the credential. Note that no disclosed attributes but only the hidden attributes contribute to the commitment; the disclosed attributes are revealed in plaintext regardless and thus do not need to be included in the proof of knowledge.

Generating the challenge and responses. To make the proof non-interactive, the user generates the challenge c themselves using a hash function $\mathtt{H}$ that combines three elements: the context identifier *ctx* (see below), the proof contributions (A_{proof} and T), and a composite nonce η. See equation (2.2) for the full computation.

The context identifier *ctx* serves as a domain separator to prevent cross-protocol attacks, where a proof created for one protocol could be reused in another. Currently unused in Yivi, its value is fixed to 1 [22].

The proof contributions are not fixed in size, each credential contributes its own A_{proof} and T, and these can be concatenated into the hash regardless of how many there are. This allows a single proof to combine attributes from multiple credentials simultaneously. For clarity, this section covers the single-credential case, but the construction extends straightforwardly when multiple credentials are involved.

The composite nonce η is constructed differently from a traditional simple random value. It is a hashed combination of a 'server nonce', a hash of the message being signed, and the signature of an included timestamp. The server nonce itself is a random value generated by the requestor, the party that requests the signature from the user. It represents the nonce typically referred to when discussing non-interactive proofs via the Fiat-Shamir heuristic [21]. By including the message hash and timestamp signature alongside the server nonce in η, Yivi ensures that the proof is intrinsically bound to the specific message and time. This prevents replay attacks where an old proof could be reused to sign a different message or at a different time.

This results in the following challenge equation:

$$c = \mathtt{H}\Big(ctx \;\Big\|\; \underbrace{A_{proof} \;\|\; T}_{\text{contributions}} \;\Big\|\; \underbrace{\mathtt{H}\big(\eta_{server} \;\|\; \mathtt{H}(message) \;\|\; timestamp\big)}_{\eta} \Big) \tag{2.2}$$

With the challenge c fixed, the user computes one *response* s for each

value that must stay hidden:

$$\begin{aligned} s_e &= r_e + c \cdot \tilde{e}, && \text{where } \tilde{e} = e - 2^{\ell_e - 1}, \\ s_v &= r_v + c \cdot v_{proof}, \\ s_{m_i} &= r_{m_i} + c \cdot m_i, && \text{for each } i \in H \cup \{0\}. \end{aligned}$$

Every response follows the same pattern: a random value r chosen when forming the commitment T, plus the challenge c multiplied by the secret. This pattern is the heart of a Schnorr-style proof of knowledge. The random term r hides the secret, since adding a fresh random number to a value reveals nothing about that value on its own. The term $c \cdot \textit{secret}$ ties the response s to this particular challenge c, which is what makes the proof hard to forge without knowing the secret. The verifier's final check (section 2.4.4) is what relies on this.

The only response that does not mask the associated raw secret directly is s_e. Recall from section 2.4 that e is a prime of a fixed, public bit-length ℓ_e, so its value always lies in a known window just above $2^{\ell_e - 1}$. Proving knowledge of e itself would waste effort on bits that are already public. Instead the user proves knowledge of the offset $\tilde{e} = e - 2^{\ell_e - 1}$, the small distance between e and the bottom of that window. This keeps s_e compact and avoids redundant computations; the constant $2^{\ell_e - 1}$ that was subtracted here is added back by the verifier during reconstruction, as shown below.

The proof that the user sends to the verifier bundles together everything needed to check the credential without exposing any secret. It is a tuple consisting of:

- c: the challenge hash;
- A_{proof}: the randomised signature;
- s_e, s_v: responses for the exponent and blinding factor;
- $\{s_{m_i}\}_{i \in H \cup \{0\}}$: responses for all hidden attributes (including sk);
- $\{m_j\}_{j \in D}$: plaintext values of all disclosed attributes.

Note what is deliberately missing from this list: the original signature (A, e, v), the master secret sk, and the hidden attribute values themselves never appear. The verifier receives only the masked responses and the randomised signature, so it can confirm the credential is valid while learning nothing it should not.

2.4.4 Verifying a Signature

This subsection covers *verification*, where the verifier checks the proof using only public values and never learns any of the hidden ones. Verification proceeds in three steps, shown in Figure 2.1: reconstructing the commitment, checking that the reconstruction is exact, and a final challenge check that accepts or rejects the signature.

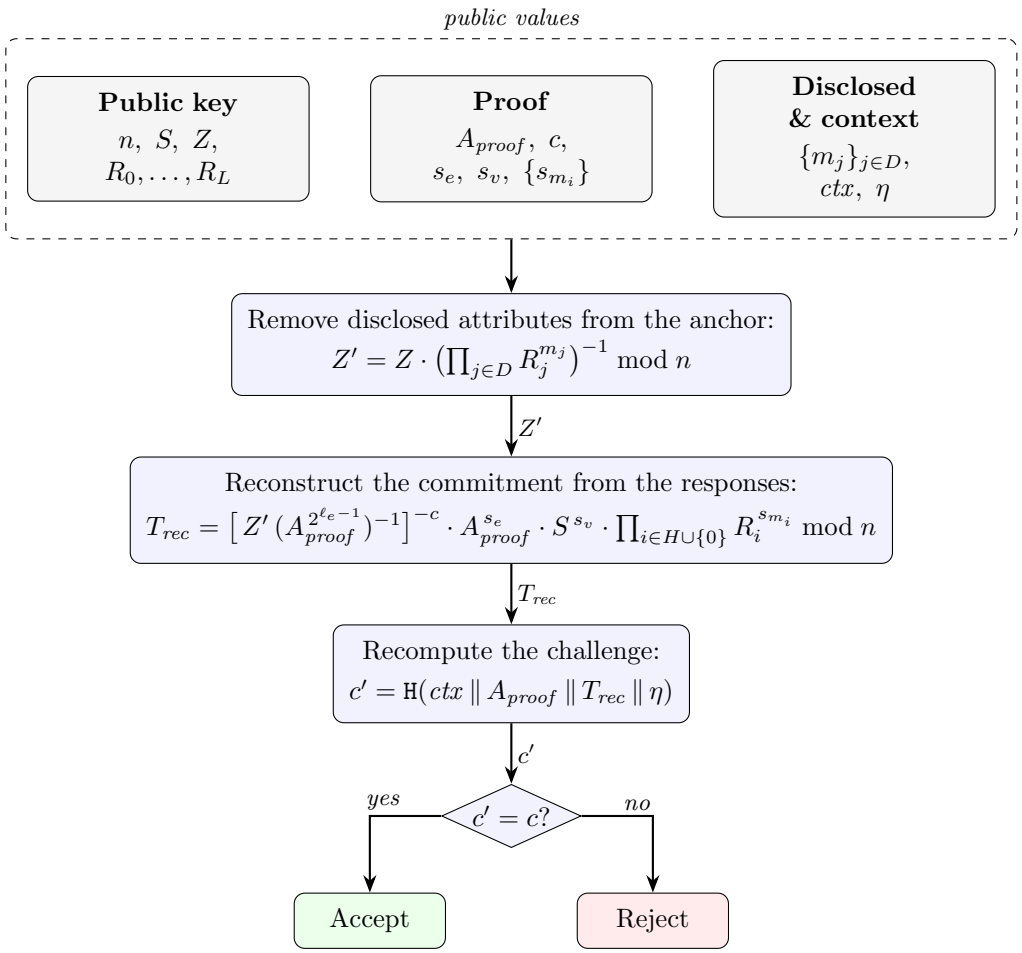

Figure 2.1: Verifying a Yivi signature (Section 2.4.4). The verifier removes the disclosed attributes from the anchor Z, reconstructs the commitment T_{rec} from the proof's responses, recomputes the challenge c', and accepts only if $c' = c$. Every input is public; no secret value (sk, e, v, or a hidden attribute) ever enters the computation.

Commitment Reconstruction

Having received the proof, the verifier must check that the responses are consistent with a genuine credential, and it must do so without ever seeing the hidden values. When the user built the commitment T, it raised the public generators to the random exponents r_e, r_v, and r_{m_i}. The verifier cannot repeat that computation directly, because it does not know those random values. What it can do is recompute the same quantity using the responses s in place of the randoms r, and then cancel out the extra challenge-dependent part using the public credential data. If the prover really knew the hidden values, this recomputation lands back exactly on the original commitment T. If it did not, the result will not match, and the final challenge check will reject the proof.

Adjusting the anchor. The credential equation (2.1) ties together every attribute, both hidden and disclosed, through the public value Z. The proof, however, only needs to reason about the hidden part, since the disclosed attributes $\{m_j\}_{j\in D}$ are already known to the verifier in plaintext. The verifier therefore removes the disclosed attributes from Z in advance, producing an adjusted anchor Z' that accounts for everything except the hidden values:

$$Z' = Z \cdot \Big(\prod_{j\in D} R_j^{m_j}\Big)^{-1} \bmod n.$$

Dividing Z by the disclosed contribution $\prod_{j\in D} R_j^{m_j}$ (which is what multiplying by the inverse does modulo n) leaves precisely the hidden side of the credential equation:

$$A^e \cdot S^v \cdot \prod_{i\in H\cup\{0\}} R_i^{m_i} \equiv Z' \pmod n.$$

In other words, Z' is the target the hidden values must reproduce, and it can be computed by the verifier using only public information.

Reconstructing T. The verifier now rebuilds the commitment as a product of four terms, all modulo n. Three of them mirror the structure of the original commitment $T = A_{proof}^{r_e} \cdot S^{r_v} \cdot \prod R_i^{r_{m_i}}$, but with each random exponent r replaced by its corresponding response s. The fourth term, $\mathcal{Z}$, undoes the surplus this substitution introduces, using the public anchor Z' and the

challenge c:

$$\begin{aligned}
\mathcal{Z} &= \left(Z' \cdot (A_{proof}^{2^{\ell_e-1}})^{-1}\right)^{-c}, && \text{(anchor \& lower-bound correction)}\\
\mathcal{A}_e &= A_{proof}^{s_e}, && \text{(exponent response)}\\
\mathcal{S}_v &= S^{s_v}, && \text{(blinding factor response)}\\
\mathcal{R}_s &= \prod_{i \in H \cup \{0\}} R_i^{s_{m_i}}, && \text{(hidden-attribute responses)}
\end{aligned}$$

$$T_{rec} = \mathcal{Z} \cdot \mathcal{A}_e \cdot \mathcal{S}_v \cdot \mathcal{R}_s \bmod n.$$

The three response terms $\mathcal{A}_e$, $\mathcal{S}_v$, and $\mathcal{R}_s$ each carry two parts inside their exponents, because every response has the form $s = r + c \cdot secret$: a part matching the original random exponent and an extra part scaled by the challenge c. The correction term $\mathcal{Z}$ exists to absorb that extra challenge-scaled part. It does two jobs at once. The factor $(Z')^{-c}$ cancels the challenge contribution of all the hidden values together, since by the previous paragraph the hidden values multiply out to exactly Z'. The factor $(A_{proof}^{2^{\ell_e-1}})^c$ adds back the public lower bound 2^{ℓ_e-1} that the user subtracted when forming $\tilde{e}$, restoring the full exponent e. If every response was computed honestly, these corrections balance the surplus exactly and T_{rec} collapses back to T. The next subsection verifies this algebraically.

Proof of Correctness

The reconstruction above only convinces a reader if it provably returns the original commitment whenever the prover was honest. This property is what the verifier's final equality check relies on: it must be the case that a correctly built proof always passes. The following derivation confirms it by starting from T_{rec}, substituting in the definitions of the responses, and simplifying until the expression reduces to T. Expanding the responses $s_e = r_e + c\tilde{e}$, $s_v = r_v + c\, v_{proof}$, and $s_{m_i} = r_{m_i} + c\, m_i$ into the reconstructed commitment gives:

$$T_{rec} = (Z')^{-c} \cdot A_{proof}^{c \cdot 2^{\ell_e-1} + r_e + c\tilde{e}} \cdot S^{r_v + c\, v_{proof}} \cdot \prod_i R_i^{r_{m_i} + c\, m_i}.$$

Next, the exponents of A_{proof} are combined. Since $c \cdot 2^{\ell_e-1} + c\tilde{e} = c(2^{\ell_e-1} + \tilde{e}) = c \cdot e$, this becomes:

$$T_{rec} = (Z')^{-c} \cdot A_{proof}^{r_e + c \cdot e} \cdot S^{r_v + c\, v_{proof}} \cdot \prod_i R_i^{r_{m_i} + c\, m_i}.$$

Each exponent now splits cleanly into a random part and a challenge-scaled part, so the whole product can be separated into the original commitment T (the random parts) and a leftover raised to the power c (the challenge parts):

$$T_{rec} = \underbrace{(A_{proof}^{r_e} \cdot S^{r_v} \cdot \prod_i R_i^{r_{m_i}})}_{=T} \cdot (Z')^{-c} \cdot (A_{proof}^{e} \cdot S^{v_{proof}} \cdot \prod_i R_i^{m_i})^c.$$

It remains to show that the leftover, $(Z')^{-c}$ times the bracketed term raised to c, equals 1. First, the bracketed term is simplified. Substituting $A_{proof} = A \cdot S^{r_{blind}}$ and $v_{proof} = v - e \cdot r_{blind}$, the blinding factor r_{blind} introduced for unlinkability cancels itself out:

$$\begin{aligned} A_{proof}^{e} \cdot S^{v_{proof}} &= (A \cdot S^{r_{blind}})^e \cdot S^{v - e\, r_{blind}} \\ &= A^e \cdot S^{e\, r_{blind} + v - e\, r_{blind}} = A^e \cdot S^v. \end{aligned}$$

Substituting this back, the bracketed term is now $A^e \cdot S^v \cdot \prod_i R_i^{m_i}$:

$$T_{rec} = T \cdot (Z')^{-c} \cdot (A^e \cdot S^v \cdot \prod_i R_i^{m_i})^c.$$

This is exactly the hidden side of the credential equation, which by the adjusted anchor of the previous subsection equals Z':

$$A^e \cdot S^v \cdot \prod_{i \in H \cup \{0\}} R_i^{m_i} = Z'.$$

The leftover therefore becomes $(Z')^{-c} \cdot (Z')^c$, which is 1, and the reconstruction collapses to the original commitment:

$$T_{rec} = T \cdot (Z')^{-c} \cdot (Z')^c = T.$$

This confirms that a proof built by a user who genuinely holds the credential always reconstructs the commitment the verifier expects.

Challenge Verification

The reconstruction gives the verifier a value T_{rec} that should equal the prover's original commitment T. The verifier never received T directly, however, so it cannot simply compare the two. What it does instead is recompute the challenge from T_{rec}, using the same hash and the same inputs the prover used, and check that the result matches the challenge c that came with the proof:

$$c' = \mathtt{H}(ctx \parallel A_{proof} \parallel T_{rec} \parallel \eta)\,.$$

The signature is accepted as cryptographically valid if and only if $c' = c$. The verifier additionally checks that each response lies within its expected range: the responses for the hidden attributes and for the exponent e must

fall inside the bit-length windows fixed by the public parameters. A response outside its window is rejected regardless of the challenge, since the proof of knowledge only binds when the responses stay within the sizes an honest prover would have produced.

The reason this single equality is enough rests on the Fiat-Shamir heuristic [21]. The original challenge c was computed by hashing the genuine commitment T. The recomputed challenge c' is obtained by hashing the reconstructed commitment T_{rec}. Because a hash function maps different inputs to different outputs, $c' = c$ can hold only if $T_{rec} = T$. The previous subsection showed that an honest prover always achieves $T_{rec} = T$, so a correct proof always passes. The converse is what makes the check hard to fool: producing responses that reconstruct to the very commitment that was hashed into c, without knowing the hidden values, would require predicting the hash output before choosing the inputs to it. This is the forgery difficulty noted during signature creation (section 2.4.3), now made concrete on the verifier's side, and it is why a passing check is taken as strong evidence that the prover knew the hidden values. This equality confirms that the prover possesses a credential satisfying the issuer's credential equation, and that the disclosed attributes are bound to that same credential, all without revealing sk, the exponent e, the blinding factor v, or any of the hidden attributes.

Chapter 3

Design and Implementation

This chapter explains how the verification library was built. It first sets out the design decisions behind the port, then the constraints imposed by running inside a browser, the strategy used to keep the port correct, and finally the parts of `irmago` that were left out of this version. The full source code is available online [23].

3.1 Design Decisions

Porting vs. Alternative Approaches Three approaches were considered to enable client-side signature verification: automated source-to-source translation from Go to JavaScript, compilation of the Go codebase to WebAssembly, and a manual port of the Go verification logic to TypeScript.

Automated translation could, in principle, handle isolated parts of a port. The only tool found and maintained that translates Go to JavaScript directly is GopherJS [24], but it is not designed to produce a standalone browser library: it ships a full Go runtime alongside the translated code and preserves the server-side structure of the original. Unfamiliarity with GopherJS could also risk spending more time on learning and debugging the translation process than on the port itself. GopherJS also did not support the version of Go that `irmago` requires, making it inapplicable without significant additional work.

WebAssembly was considered next. Compiling the existing Go code to a `.wasm` binary would, in theory, preserve all cryptographic logic without modification. This is not hypothetical: an earlier effort compiled the relevant verification routine of `irmago` to WebAssembly with the standard Go compiler, so that it could be called from JavaScript in the browser [25]. The approach works, but it carried practical drawbacks. `irmago` had to be patched to compile at all: its file system access for the scheme manager had

to be replaced by an in-browser stub, and the resulting binary was large, which matters because it must be downloaded in full before any verification can run. Given that native `BigInt` and the Web Crypto API are already fast enough for client-side verification at interactive speeds, as the benchmarks in Chapter 4 confirm [26, 27], the gain from compiling to WebAssembly did not justify its cost here.

Manual porting was therefore chosen. It is the option that keeps the verification logic in a high-level language that a reviewer can read and compare against `irmago` directly, which was a concern held throughout the process, and it keeps the result accessible to future contributors. This is more work than the WebAssembly approach, but it ensures the domain change to the browser is handled thoughtfully rather than just patching the existing code to run in a different environment. The cost is real and should be stated plainly. Manual porting is slow, it depends on the accuracy of the person doing it, and it creates an ongoing obligation: when `irmago` changes, the relevant logic must be found and ported by hand rather than recompiled automatically. This was an acceptable trade-off for a verification library, where auditability matters more than speed of delivery, and the structural choices described in section 3.3 are meant to keep the maintenance obligation manageable.

Language and Technology Stack JavaScript was the natural target language: it runs natively in all modern browsers, has a mature cryptography ecosystem, and is familiar to the web developers most likely to use this library. TypeScript [28] was chosen as the development language as it has multiple advantages over plain JavaScript for a project of this nature. Static typing catches a large number of errors at compile time and allows for an additional layer of documentation safety through type annotations.

The compiler targets the ECMAScript 11 standard (`ES2020`), which provides two features that are essential for this implementation. The first is native `BigInt`, which enables arbitrary-precision integer arithmetic directly in the language without any third-party dependency [26]. This is necessary for the modular exponentiations and large-integer operations that underpin the CL signature scheme described in Chapter 2. The second is that `ES2020`, and `BigInt` in particular, has been supported by every major browser since 2020, so targeting it does not restrict the set of users a developer can reach.

Vite [29] was chosen as the build tool for this project. It converts the TypeScript code into a standard JavaScript module (`ES2020`). This allows the final code to be used directly in a web browser or shared as an `npm` package, so other developers do not need to set up their own build tools. Tests are written with Vitest [30], as this seamlessly plugs into Vite.

3.2 Client-Side Constraints

File System Access and Key Retrieval The Go implementation caches Yivi's scheme on the local file system. Browsers do not permit this: the sandbox model intentionally prevents web pages from reading local paths without the user explicitly defining said path [31]. It is inconvenient to make the user supply the schemes themselves, so a different strategy is required.

The TypeScript implementation retrieves schemes and associated public keys on-demand using the Fetch API [32], requesting only the keys needed for the specific signature being verified. This means a user verifying a signature from a single issuer downloads only that issuer's public key, rather than pre-loading the entire scheme at startup. The trade-off is that the very first verification for a given issuer causes a network round trip, which can cause a noticeable delay. In practice however, this is largely avoided by relying on built-in browser HTTP caching: once a key has been fetched it is served from the local cache on subsequent page loads, making the latency cost a one-time expense per issuer per session.

Single-Threaded Execution Go's concurrency model, built on goroutines and shared data structures, has no direct equivalent in the browser. JavaScript executes on a single thread by design, and the concurrency primitives used in `irmago` are therefore not applicable. In practice this was not a significant constraint, because `irmago` does not make extensive use of concurrency during signature verification, and the parts that do use it parallelise operations that are independently fast. These were simplified to sequential equivalents.

Web Workers [33], which allow for background execution in a separate thread, were considered as a future optimisation for computationally intensive steps. They were left out because single-threaded performance already meets interactive requirements, as shown in Chapter 4, and because of time constraints. For now they remain an avenue for future work, as discussed in section 6.3.

3.3 Correctness Strategy

Porting cryptographic code introduces a specific risk: the ported code can look the same as the original yet still produce different results, because of incorrect bit widths or subtle differences in how integers are encoded. There is no indication that `irmago` itself contains errors; the porting process is the source of risk, and testing the port against the original's output is what catches such mistakes. To protect against this, the implementation uses two complementary strategies.

Structural Matching The TypeScript codebase mirrors the structure of `irmago` closely, organising code into two packages that mirror those in the original. The *gabi* package contains the core cryptographic operations: proof verification, key handling, and the zero-knowledge challenge reconstruction described in Chapter 2. The *irma* package contains the higher-level protocol logic: scheme management, attribute disclosure which is still needed during verification, and the orchestration of the verification flow.

This similarity serves two purposes. When a future change is made to `irmago`, a developer can locate the mirrored TypeScript code directly and choose to implement it with relative ease. This also makes it easier to review the code for correctness, as the logic can be directly compared to the original Go code.

Unit Testing Tests were generated by running the original Go implementation on a set of known inputs and recording its outputs. The same inputs were then fed into the TypeScript implementation, and the outputs were compared both manually and with unit tests. This gives strong evidence that the mathematical operations produce identical results across both languages on the cases tested. Additional tests cover TypeScript-specific logic that has no Go counterpart, such as the parsing of incoming verification requests and the handling of network errors when fetching keys. The functions covered include the encoding and decoding of attributes across all supported formats and versions, the zero-knowledge proof challenge reconstruction (`ReconstructZ`), and credential verification against issuer public keys. Coverage is measured with Vitest's built-in tooling. Across the library the test suite reaches 80% line coverage, 74% function coverage, and 70% branch coverage, with the core cryptographic modules covered more thoroughly than the surrounding request-handling code. The inputs are chosen to exercise the different attribute formats and credential versions that drive the main branches in the decoding logic. This reduces, but does not remove, the chance that a discrepancy hides on an input outside the test set, as discussed further in Chapter 5. All tests are run with Vitest inside a `jsdom` environment, which simulates the browser APIs available at runtime.

3.4 Known Limitations

Not every feature of `irmago` was ported within the scope of this project. This section describes the most significant exclusions, explains why each was excluded, and assesses the practical impact on real-world verification.

Timestamp Verification Yivi signatures include a timestamp produced by the `atum` library. Its purpose is to provide a safety guarantee for long-term validity in a potential post-quantum world, addressing a fundamental property of the CL signature scheme (see section 2.4): the scheme's security rests on the Strong RSA Assumption, which is known to fail against a sufficiently powerful quantum computer running Shor's algorithm [34, 35]. To this end, the `atum` timestamp defaults to the post-quantum `XMSSMT` signature scheme, so that once quantum computers become a realistic threat, a verifier can still know that any Yivi signature whose timestamp predates viable quantum computers is valid. Note, however, that `irmago` does not currently make use of this capability. When requesting a timestamp it sets its preferred algorithm to the classical `Ed25519` scheme, which is smaller and faster but provides no post-quantum guarantee, so Yivi signatures at present carry a non-post-quantum timestamp. Since `atum` is intended to be used with post-quantum signatures, the timestamp is assumed to always be signed with `XMSSMT` going forward.

No JavaScript library exists for parsing and verifying `atum` signatures, and implementing one from scratch fell outside the scope of this project. In the present, pre-quantum world, this lack of timestamp verification carries no practical consequence, because the timestamp signature is included in the nonce η from which the challenge c is derived, as described in section 2.4.3. The proof is therefore bound to that specific timestamp value, and any current attempt to change it causes challenge verification to fail immediately. As research into quantum computing and engineering progresses, the importance of this check will grow, and implementing full `atum` verification is identified as future work in section 6.3.

Revocation Checking Yivi supports credential revocation, allowing an issuer to invalidate a credential before it expires, for example if the underlying attributes are no longer accurate or the issuer was compromised. The verification of non-revocation proofs was not ported in this implementation.

In practice, this missing aspect has limited impact today, because credential revocation is not yet used by the majority of active issuers in either the `pbdf` or `irma-demo` schemes (which can be seen by the lack of specified `RevocationServer` in the `description.xml` of most issuers [36, 37]). However, if revocation adoption grows, a verifier that skips non-revocation checks could accept signatures from credentials that have been explicitly

invalidated. This should therefore be treated as a priority for future work, as discussed in section 6.3.

Scheme Signature Verification Yivi's scheme manager signs the entire scheme directory to guarantee its integrity, as described in Chapter 2. A verifier that does not check this signature must trust that the scheme files it downloads have not been tampered with, relying entirely on transport-layer security and the server sending the scheme rather than a cryptographic guarantee over the content itself. This check was not implemented in the current version. The risk is partly mitigated by the use of HTTPS when fetching scheme files, but a comprehensive implementation should verify the scheme manager's signature directly, and this is identified as high priority future work in section 6.3.

External Dependencies The implementation relies on two external libraries. `asn1js` [38] is used to parse ASN.1 DER-formatted structures embedded in the scheme files, and `fast-xml-parser` [39] is used to parse the XML scheme definitions that Yivi uses to describe its credential types. Both were chosen for their wide use and small, focused codebases. A direction for future work would be to replace these with small, purpose-built parsers to reduce supply-chain risk and exposure to faulty updates, as discussed in section 6.3.

Chapter 4

Performance

The performance of the TypeScript implementation was benchmarked against the original Go implementation to establish whether client-side verification is fast enough to be practical. Both benchmarks measure the complete signature verification function, with keys and scheme definitions pre-loaded into memory before each run. Fetching these resources over the network was excluded from measurement, since it falls outside the control of the verification logic itself and would obscure the computational comparison. In a real browser environment this introduces an additional cost per issuer, mitigated by HTTP caching on subsequent page loads, as discussed in Chapter 3. Running each test with and without an explicit warm-up resulted in no meaningful difference in either implementation, so all runs are combined in the results below. The raw benchmark data along with the code used to generate this data is available in the repository [23].

4.1 Benchmarking Environment

All benchmarks were conducted on the same machine running Linux (x86-64), equipped with an AMD Ryzen 7 8840HS processor and 16 GB of RAM. The TypeScript benchmarks were executed using Vitest `4.1.3` on Node.js `v22.22.3`, with Vite `7.1.11` as the build and transformation layer. The Go benchmarks were executed using Go `1.23.5` on the same machine, compiled and run natively without an intermediate runtime. Each test was repeated across a fixed number of outer iterations, with each iteration verifying a fixed number of signatures independently. Four independent TypeScript runs and three independent Go runs were collected.

4.2 Results

Table 4.1 shows the mean and standard deviation for each test configuration, averaged across all runs.

Test	TypeScript		Go	
	Mean (ms)	Std Dev	Mean (ms)	Std Dev
1 verification	11.70	0.49	2.36	0.20
100 verifications	1169	10.1	233	0.90
1000 verifications	11765	77	2338	12.0

Table 4.1: Results: TypeScript vs. Go (mean ± std dev, ms per iteration)

The per-signature cost is consistent across all batch sizes in both implementations. TypeScript averages approximately 11.7 ms per signature and Go approximately 2.3 ms, resulting in a **5**× performance penalty across every configuration. The standard deviation remains close to or below 1% in batch configurations and still under 5% for single verifications despite the smaller absolute times leading to more variability, indicating stable and predictable behaviour under sustained load.

4.3 Discussion

Google's RAIL performance model treats 100 ms as the limit below which a response feels instantaneous to the user, and recommends a 50 ms processing budget to leave room for the browser's other work [40, 41]. At 11.7 ms per verification, the implementation comfortably meets both for verifying one signature.

The raw comparison with Go also should not be taken at face value. A server verifying signatures on behalf of many users must share its resources across all concurrent requests, so the effective per-user latency grows under load. The client-side implementation runs on the user's own device, dedicated entirely to their session, which means the 5× difference in throughput does not translate directly into a 5× difference in user-perceived performance.

For use cases involving large batches of verifications, for example, a conversation where every message carries a Yivi signature, the linear scaling means that 1000 verifications would take approximately 11.7 seconds. This, on its own, would be unacceptable according to the RAIL model, and would likely lead to a poor user experience. Potential strategies for mitigating this delay (e.g. deferred verification) are discussed in section 6.3.

Chapter 5

Security and Correctness

The only task this library performs is to decide whether a Yivi signature is valid. Its security therefore rests on two properties. The first is soundness: the library must never accept an invalid signature, since a single false acceptance defeats the purpose of verifying at all. The second is faithfulness of the port: the move from Go to TypeScript must not introduce any weakness that is absent from the trusted `irmago` implementation. The sections that follow examine the threats the library must withstand, the integrity of the port and the platform primitives it builds on, the risks introduced by external dependencies and by running inside a browser, and the security consequences of the checks left unported in section 3.4.

5.1 Threats

A verification library is exposed to a narrower set of threats than a signing library, because it never handles secret key material. Throughout this chapter the verifier is assumed to hold the correct issuer public key; the integrity of that key is itself one of the unported checks, and its significance is revisited in section 5.5. Given a correct public key, three potential threats were considered relevant.

Forgery is the central threat. An attacker presents a signature that corresponds to no credential issued by a trusted issuer, or one that claims attributes the underlying credential does not contain, and tries to have it accepted. For example, an attacker might take a genuine credential that proves they are a student and attempt to construct from it a proof of an "is over 18" attribute they were never issued. Defending against this is exactly the soundness property described above. The defence is not handled in this library but is inherited from the proof system. Yivi's credential is an instance of the CL signature, which Camenisch and Lysyanskaya prove existentially unforgeable under the Strong RSA assumption [17]; the selective-disclosure

proof the verifier checks is a proof of knowledge of such a signature, made non-interactive with Fiat-Shamir [21]. A prover without a valid credential therefore cannot produce a proof whose reconstructed commitment matches the challenge, as established in section 2.4. The library's task is to carry out that reconstruction faithfully, which is the subject of section 5.2.

Denial of service aims to disrupt rather than deceive. Instead of forging a signature, an attacker feeds the verifier deliberately malformed data, such as a corrupt proof, a malicious XML scheme, or an invalid ASN.1 structure, hoping to crash it or even execute code. This threat is met by strict input handling, discussed in section 5.3.

Side-channel leakage targets secrets that a computation reveals indirectly, for instance through the time it takes. This threat does not apply here, for a reason developed in section 5.2: the verifier processes only public data, so there is no secret for a side channel to leak.

One concern is deliberately absent from this list. The confidentiality of the prover's hidden attributes is guaranteed by the zero-knowledge property of the proof itself, established in section 2.4, and not by any code in this library. A verifier never sees a hidden attribute, the master secret sk, or the signature tuple (A, e, v), and so it cannot leak what it never receives.

5.2 Integrity of the Port

The security of a port rests on two things: that the translation preserves the behaviour of the original, and that the primitives it builds on are themselves sound.

The choice of a manual port, set out in Chapter 3, was made partly for this reason. Because the verification logic is written by hand in a high-level language and mirrors the structure of `irmago`, a reviewer can read it and compare it against the reference line by line, rather than trusting the opaque output of a translation step that nobody inspects.

What the port reuses from Go, and what it does not, is worth making explicit. The verifier does not reimplement Go's `crypto` package. The only demanding primitive it needs is modular exponentiation on large integers, which it performs with the browser's native `BigInt`, alongside SHA-256 from the Web Crypto API for the challenge hash [26, 27]. Both are standard components of every modern browser.

These primitives differ from their Go counterparts in one respect that initially looks alarming. The cryptographic libraries in Go and C use constant-time algorithms to resist side-channel attacks, whereas standard JavaScript `BigInt` operations are not constant-time [42]. Constant-time arithmetic

exists to hide secret operands, so that the running time of an operation reveals nothing about the values it processes. A Yivi verifier, however, operates only on public keys and public signature components. The time an operation takes can therefore reveal only values an attacker could already read directly from the signature, and the threat that constant-time code is designed to prevent, recovery of a secret through timing, does not arise. This is the reasoning promised under the side-channel threat in section 5.1.

What equivalence testing can and cannot show. The port is checked against `irmago` by the unit tests described in section 3.3: the reference is run on a set of inputs, its outputs are recorded, and the TypeScript implementation is required to produce the same outputs for the same inputs. This gives strong evidence that the two agree on the cases tested, but by construction it cannot prove that they agree on every possible input. Two kinds of residual risk remain, and they differ sharply in severity. The first, fixed-width integer overflow, is the classic way two ports of the same arithmetic silently diverge, where a value wraps at a width in one language that it does not reach in the other. This risk is largely absent here, because both implementations use arbitrary-precision integers, `math/big` in Go and `BigInt` in TypeScript, so there is no fixed width at which a value can wrap. The second risk lies in encoding, length handling, and parsing, where a discrepancy could surface only on an input outside the test set. To narrow this, the tests cover attribute encoding and decoding across all supported formats and versions, and the structural mirroring of section 3.3 keeps the two codebases close enough to be compared function by function. Confidence in the port is as a result high, but not complete: testing a finite set of inputs and reviewing the code reduce the chance of an undetected discrepancy without removing it entirely.

5.3 Dependencies and Supply Chain

The library draws on two external packages: `asn1js` to parse the ASN.1 DER structures in the scheme files [38], and `fast-xml-parser` to parse the XML credential definitions [39]. Every such package is code that cannot be controlled, and so forms part of the attack surface.

To keep installations reproducible, the exact version of each package used is recorded in a lockfile. This pinning is a trade-off. On one side, every build then uses the same audited code instead of whatever a version range happens to resolve to on the day of installation. On the other, a pinned version does not receive upstream security fixes on its own, so pinning exchanges automatic updates for the obligation to update deliberately whenever a vulnerability in a dependency comes to light.

The verifier is written to ***fail closed***: any input it cannot parse is treated

as a verification failure, rather than letting verification proceed on partial or guessed data. This is the intended answer to the denial-of-service threat of section 5.1, since a malformed proof or scheme is rejected outright instead of crashing the verifier. The guarantee is only as strong as the parsers beneath it. A defect in a third-party parser could accept a malformed structure, or quietly reshape it into a different but well-formed one, so failing closed at the library level does not by itself rule out faults inside the parsers. This is part of why replacing both with small, purpose-built parsers is identified as future work in section 6.3.

5.4 Trust in a Client-Side Setting

Moving verification into the browser changes where the work is done, but it does not by itself remove the need to trust a server. The page that runs the verification, and the library code within it, is still delivered by a server, and a server that serves tampered JavaScript could report any result it chose, whatever the real signature said. The prover's attributes and signature never leave the user's device, and the server is relieved of the verification work it would otherwise perform on the user's behalf. The dependence on the serving server can be reduced, though not removed, by distributing the library as a fixed and reviewed bundle, such as an `npm` package which uses Subresource Integrity to ensure fetched packages are delivered without manipulation[43, 44]. This ensures that the code a user runs is tied to a known version rather than having to be trusted again on every page load.

The browser environment is shared in a second way that the server cannot fix. A script injected into the page, for instance through cross-site scripting, runs with the same privileges as the library and could overwrite the displayed result regardless of what the verification actually returned. This exposure is inherent to client-side web applications and lies beyond what a verification library can defend against on its own; guarding against it falls to the developer who embeds the library in a page. This however is not a new risk introduced by the client-side port, since any server-side verification also relies on the integrity of the page that displays its result.

5.5 Consequences of the Unported Checks

Section 3.4 describes what each unported check does and why it was set aside. The concern here is only what an attacker stands to gain from each omission.

Timestamp verification. Leaving the `atum` timestamp unverified has no effect on present-day soundness, because the timestamp is folded into the

nonce η, and thus into the challenge c, so it cannot be altered without invalidating the proof (section 2.4.3). What is given up is a long-term guarantee, and only a conditional one. Once a quantum computer can break the Strong RSA Assumption, CL signatures become forgeable, and a verifier without this check can no longer establish that a given signature was created before that point and was therefore genuine when it was made. This guarantee naturally depends on the timestamp itself being signed with a post-quantum scheme, which is currently not the case (see section 3.4). However, the check is not urgent today, since the timestamp is already bound into the proof, and the quantum threat is still years away according to national and supranational transition roadmaps [45, 46, 47].

Revocation checking. A verifier that omits the non-revocation proof will accept a credential its issuer has explicitly revoked (section 3.4). This is a real soundness gap rather than a theoretical one. It is minor today only because few issuers currently use revocation, and it widens in direct proportion to how broadly revocation is adopted.

Scheme signature verification. This is the most serious of the three gaps. Without verifying the scheme manager's signature, the issuer public keys are trusted on the strength of the HTTPS connection and security of the server sending the scheme, rather than a cryptographic guarantee over their contents (section 3.4). An attacker able to serve a modified scheme could substitute their own issuer public key and have signatures of their own making accepted. Because that key is the root of trust for every check the library performs, compromising it defeats verification entirely, which is why verifying the scheme signature is the highest-priority item in section 6.3.

Chapter 6

Conclusion and Future Work

This thesis ported Yivi's signature verification logic from Go to TypeScript, producing a library that verifies Yivi's attribute-based signatures directly inside the browser [23]. The resulting library implements the core cryptographic operations a verifier needs, and removes the requirement for a dedicated server to perform them.

6.1 Research Contributions

The main contribution of this work is the open-source library that enables verification of Yivi signatures in the browser [23]. This fills a gap in the current ecosystem, where verification was previously possible only on a server.

To keep the library reliable and easy to audit, its structure closely mirrors that of the original `irmago` codebase. By following the structure of the reference code, the complex parts, in particular the reconstruction used to verify the zero-knowledge proofs, could be compared against the original function by function. This made it practical to test every ported cryptographic function against `irmago`'s own output, and it leaves a maintainable foundation for future updates.

The work also shows what a modern browser is capable of on its own. Native web standards, namely `BigInt` for arbitrary-precision integer arithmetic and the Web Crypto API for hashing, were sufficient to implement the full verification path. This indicates that heavy external cryptographic dependencies, or a WebAssembly build, are not required for client-side verification.

6.2 Critical Evaluation

The evaluation compared the compiled, server-side reference with the interpreted, client-side port. As shown in Chapter 4, the TypeScript implementation is about five times slower than native Go, at roughly 11.7 ms

per signature against roughly 2.3 ms. In absolute terms this is comfortably below both RAIL's 50 ms processing budget and its 100 ms instantaneous-response limit, so a single verification is fast enough for interactive use. The limit is reached only for large batches: verifying a thousand signatures takes on the order of 11.7 seconds, which would harm the user experience. Strategies for handling that case are discussed in section 6.3.

From a security standpoint, detailed in Chapter 5, the library implements the core verification logic, including the zero-knowledge proof check on which soundness depends, and it matched the reference on every tested input. As that chapter makes clear, testing against a finite set of inputs cannot guarantee agreement on all inputs, so this is strong evidence rather than a proof of equivalence. The main residual gaps are the checks that were left unported: the `atum` timestamp, revocation, and the scheme manager's signature, the last being the most security-relevant because the issuer public key is the root of trust for every verification. These omissions were deliberate scope decisions and are the primary candidates for future work, addressed in section 6.3.

Finally, moving verification to the client changes the trust model rather than simply improving it. The user's attributes and signature stay on their own device, and the server is relieved of the verification work, but the server is still trusted to serve honest code, and the result remains exposed to page-level threats such as cross-site scripting. A developer integrating this library must therefore secure the surrounding page, since the integrity of the displayed result is only as strong as the environment it runs in.

In answering the central research question: verification can be done correctly and efficiently in the browser. The trade-offs are a roughly five-fold compute cost, offset by dedicated client resources and the removal of server-side verification; a reduced but not eliminated trust dependency on the serving server; and three unported checks whose security impact is documented and scoped.

6.3 Future Work

The current implementation is a functional and fast client-side verifier, but several directions would extend, strengthen, and optimise it.

6.3.1 Completing the Unported Checks

The three checks left out of this version, described in section 3.4 and analysed for their security impact in Chapter 5, are the most important work remaining.

Verifying the scheme manager's signature is the highest priority, because the issuer public key is the root of trust for every verification: without this check, anyone able to serve a modified scheme could substitute their own

key. Revocation checking is the next priority, and its urgency grows directly with how widely issuers adopt revocation. Verifying the `atum` timestamp is the least urgent of the three today, since the timestamp is already bound into the proof, but it matters for long-lived signatures: its importance scales with progress toward a cryptographically relevant quantum computer, which national and supranational transition roadmaps currently place years away [45, 46, 47]. No JavaScript library currently parses and verifies `atum` signatures, so this step also requires writing that parser.

6.3.2 Performance Optimisations

The performance evaluation in Chapter 4 shows that the library is fast enough for interactive use, but there is room to improve, especially when verifying many signatures at once. The dominant cost in verification is modular exponentiation on large integers during the zero-knowledge proof check, so that is the natural target for any optimisation.

Web Workers. JavaScript runs on a single thread [33], so a long run of verifications blocks the main thread and can freeze the interface. Web Workers move work to background threads, which would let independent signatures be verified in parallel and keep the interface responsive, for example when a chat history full of signed messages is loaded.

WebAssembly. As explained in Chapter 3, the library was written in pure TypeScript for readability and ease of audit. A hybrid design could keep the high-level logic in TypeScript while moving only the modular-exponentiation path into a WebAssembly module, narrowing the gap to native speed. The drawback is a more complex build and a larger bundle.

Robust caching. The current implementation relies on the browser's HTTP cache for schemes and public keys, which works but gives the application little control. A more robust caching layer could be implemented to give more control and potentially improve performance. However, this would add complexity currently not warranted by the potential performance gain, so it is a lower priority than the other optimisations.

6.3.3 Removing External Dependencies

The library depends on `asn1js` and `fast-xml-parser`, which, as discussed in Chapter 5, place third-party code in the trust path. Replacing them with small, purpose-built parsers would shrink the attack surface and remove a source of supply-chain risk. This has a clear drawback: a correct ASN.1 DER parser is hard to write, and a hand-rolled parser could introduce its

own bugs, so this trades a small, well-tested dependency surface against the risk and maintenance cost of bespoke code.

6.4 Final Remarks

By removing the need for a dedicated verification server, this library lowers the barrier to adopting Yivi's privacy-preserving signatures. It is a step towards a more privacy-centric web, in which verifying who signed something can happen locally, under the user's own control, rather than being handed to a remote service.

Bibliography

[1] Google, *Update your account to meet age requirements*, https://support.google.com/accounts/answer/1333913, 2026. (visited on 02/23/2026).

[2] O. Chia, "ID photos of 70,000 users may have been leaked, discord says," *BBC*, Oct. 2025. [Online]. Available: https://www.bbc.com/news/articles/c8jmzd972leo (visited on 02/23/2026).

[3] P. Okunytė, "252m identities dumped online in massive leak affecting 7 countries," *CyberNews*, Sep. 2025. [Online]. Available: https://cybernews.com/security/identity-records-global-data-leak/.

[4] V. Petkauskas, "IDMerit data breach, 1 billion records of personal data exposed in KYC data leak," *CyberNews*, Feb. 2026. [Online]. Available: https://cybernews.com/security/global-data-leak-exposes-billion-records/.

[5] F. Bisogni and H. Asghari, "More than a suspect: An investigation into the connection between data breaches, identity theft, and data breach notification laws," *Journal of Information Policy*, vol. 10, pp. 45–82, May 2020, ISSN: 2381-5892. DOI: 10.5325/jinfopoli.10.2020.0045. eprint: https://scholarlypublishingcollective.org/psup/information-policy/article-pdf/doi/10.5325/jinfopoli.10.2020.0045/1611448/jinfopoli_10_1_45.pdf. [Online]. Available: https://doi.org/10.5325/jinfopoli.10.2020.0045.

[6] U. Feige, A. Fiat, and A. Shamir, "Zero knowledge proofs of identity," in *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '87, New York, New York, USA: Association for Computing Machinery, 1987, pp. 210–217, ISBN: 0897912217. DOI: 10.1145/28395.28419. [Online]. Available: https://doi.org/10.1145/28395.28419.

[7] Privacy by Design Foundation and Caesar Groep, *Yivi documentation*. [Online]. Available: https://docs.yivi.app (visited on 02/25/2026).

[8] J. Camenisch and E. Van Herreweghen, "Design and implementation of the idemix anonymous credential system," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, ser. CCS '02, Washington, DC, USA: Association for Computing Machinery, 2002, pp. 21–30, ISBN: 1581136129. DOI: 10.1145/586110.586114. [Online]. Available: https://doi.org/10.1145/586110.586114.

[9] Privacy by Design Foundation and Caesar Groep, *Attribute-based signatures*, https://docs.yivi.app/technical-overview/#attribute-based-signatures, Jun. 2025. (visited on 02/25/2026).

[10] Privacy by Design Foundation, *Irmago*, https://github.com/privacybydesign/irmago. (visited on 05/19/2026).

[11] Privacy by Design Foundation, *Over yivi*, https://yivi.app/about_yivi/. (visited on 02/25/2026).

[12] Privacy by Design Foundation and Caesar Groep, *Zero-knowledge proofs*. [Online]. Available: https://docs.yivi.app/zkp.

[13] Privacy by Design Foundation and Caesar Groep, *Yivi technical overview*, https://docs.yivi.app/technical-overview#irma-security-properties. (visited on 05/20/2026).

[14] Privacy by Design Foundation and Caesar Groep, *Yivi scheme*, Jun. 2025. [Online]. Available: https://docs.yivi.app/schemes/.

[15] Logius, *What is digid*, https://www.digid.nl/en/about-digid/what-digid/. (visited on 05/20/2026).

[16] Privacy by Design Foundation and Caesar Groep, *What is yivi*, https://docs.yivi.app/what-is-yivi. (visited on 05/20/2026).

[17] J. Camenisch and A. Lysyanskaya, "A signature scheme with efficient protocols," in *Security in Communication Networks*, S. Cimato, G. Persiano, and C. Galdi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 268–289, ISBN: 978-3-540-36413-9.

[18] N. Barić and B. Pfitzmann, "Collision-free accumulators and fail-stop signature schemes without trees," in *Advances in Cryptology — EUROCRYPT '97*, W. Fumy, Ed., Berlin, Heidelberg: Springer, 1997, pp. 480–494, ISBN: 978-3-540-69053-5. DOI: 10.1007/3-540-69053-0_33.

[19] E. Fujisaki and T. Okamoto, "Statistical zero knowledge protocols to prove modular polynomial relations," in *Advances in Cryptology — CRYPTO '97*, B. S. Kaliski, Ed., Berlin, Heidelberg: Springer, 1997, pp. 16–30, ISBN: 978-3-540-69528-8. DOI: 10.1007/BFb0052225.

[20] Privacy by Design Foundation and Caesar Groep, *Special attributes*, https://docs.yivi.app/technical-overview#special-attributes, Jun. 2025. (visited on 06/05/2026).

[21] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Advances in Cryptology — CRYPTO' 86*, A. M. Odlyzko, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 186–194, ISBN: 978-3-540-47721-1.

[22] Privacy by Design Foundation and Caesar Groep, *IRMA protocol*, Jun. 2025. [Online]. Available: https://docs.yivi.app/irma-protocol/%5C#get-irmasessionclienttokenrequest.

[23] L. Nieuwenhuijzen, *IrmaTS: Client-side verification of yivi attribute-based signatures*, https://gitlab.science.ru.nl/lnieuwenhuijzen/irmats-abs-verification. (visited on 06/01/2026).

[24] GopherJS contributors, *GopherJS: A compiler from Go to JavaScript*, https://github.com/gopherjs/gopherjs. (visited on 05/31/2026).

[25] L. Botros and Radboud University iLab, *irma-signature-verify: Verify irma signatures in the browser*, https://gitlab.science.ru.nl/ilab/irma-signature-verify. (visited on 06/01/2026).

[26] Mozilla Developer Network, *Bigint*, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt, Jul. 2025. (visited on 05/26/2026).

[27] Mozilla Developer Network, *Web crypto API*, https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API, Sep. 2024. (visited on 05/26/2026).

[28] Microsoft, *Typescript for javascript programmers*, https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html, Feb. 2026. (visited on 02/12/2026).

[29] VoidZero, *Vite*, https://vite.dev/. (visited on 05/26/2026).

[30] VoidZero, *Vitest*, https://vitest.dev/. (visited on 05/26/2026).

[31] Mozilla Developer Network, *File system API*, https://developer.mozilla.org/en-US/docs/Web/API/File_System_API, May 2026. (visited on 05/26/2026).

[32] Mozilla Developer Network, *Fetch API*, https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API, May 2026. (visited on 05/26/2026).

[33] Mozilla Developer Network, *Web workers API*, https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API, Apr. 2025. (visited on 02/25/2026).

[34] P. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[35] M. Sharma, V. Choudhary, R. S. Bhatia, S. Malik, A. Raina, and H. Khandelwal, "Leveraging the power of quantum computing for breaking RSA encryption," *Cyber-Physical Systems*, vol. 7, no. 2, pp. 73–92, 2021. DOI: 10.1080/23335777.2020.1811384. eprint: https://doi.org/10.1080/23335777.2020.1811384. [Online]. Available: https://doi.org/10.1080/23335777.2020.1811384.

[36] Privacy by Design Foundation, *Pbdf-schemes*, https://github.com/privacybydesign/pbdf-schememanager. (visited on 05/19/2026).

[37] Privacy by Design Foundation, *Demo-schemes*, https://github.com/privacybydesign/irma-demo-schememanager/. (visited on 05/19/2026).

[38] Peculiar Ventures, *ASN.1js*, https://github.com/PeculiarVentures/asn1.js, Feb. 2026. (visited on 06/06/2026).

[39] Natural Intelligence, *Fast-xml-parser*, https://github.com/NaturalIntelligence/fast-xml-parser, Feb. 2026. (visited on 02/12/2026).

[40] Mozilla Developer Network, *RAIL*, https://developer.mozilla.org/en-US/docs/Glossary/RAIL. (visited on 05/28/2026).

[41] Google, *Measure performance with the RAIL model*, https://web.dev/articles/rail/. (visited on 05/28/2026).

[42] S. Mukherjee, C. Rechberger, and M. Schofnegger, "Cache timing leakages in zero-knowledge protocols," in *7th Conference on Advances in Financial Technologies (AFT 2025)*, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025, pp. 1–26.

[43] Mozilla Developer Network, *Subresource Integrity*, https://developer.mozilla.org/en-US/docs/Web/Security/Defenses/Subresource_Integrity. (visited on 06/03/2026).

[44] npm, *ssri*, https://github.com/npm/ssri. (visited on 06/03/2026).

[45] National Institute of Standards and Technology, *NIST IR 8547*, Transition to Post-Quantum Cryptography Standards, https://csrc.nist.gov/pubs/ir/8547/ipd. (visited on 05/28/2026).

[46] European Commission, *A coordinated implementation roadmap for the transition to post-quantum cryptography*, https://digital-strategy.ec.europa.eu/en/library/coordinated-implementation-roadmap-transition-post-quantum-cryptography. (visited on 05/28/2026).

[47] National Cyber Security Centre, *Timelines for migration to post-quantum cryptography*, https://www.ncsc.gov.uk/guidance/pqc-migration-timelines. (visited on 05/28/2026).