Zero-Knowledge Proofs of Generalized Regular Expression Matching for Anonymized Email Verification

Shreyas Londhe
ZK Email
shreyas@zk.email

Aayush Gupta
ZK Email
aayush@zk.email

Sora Suegami
Ethereum Foundation
sora.suegami@ethereum.org

Yogesh Shahi
ZK Email
yogesh@zk.email

Rute Figueiredo
ZK Email
rutefig@zk.email

Parisa Hassanizadeh
IPPT PAN / Zero Savvy
parisa@zerosavvy.xyz

Shahriar Ebrahimi
Alan Turing Institute / Zero Savvy
shahriar@zerosavvy.xyz

Abstract

Digital communication increasingly underpins identity, financial transactions, and regulatory compliance. In many settings, possession of a DKIM-signed email serves as evidence of account control, transaction confirmation, or institutional affiliation. Yet demonstrating such properties typically requires revealing the full email or relying on centralized intermediaries, introducing privacy risks and additional trust assumptions. A framework called ZK Email addresses this limitation by applying zero-knowledge proofs (ZKPs) to email verification, enabling publicly verifiable proofs of authenticity while preserving message confidentiality. However, its existing implementations struggle to support complex, real-world messages due to the inefficiency of regular-expression verification over structured formats and rich alphabets.

We address this limitation with a new ZKP system for regex matching based on path verification over ε-free NFAs, yielding prover complexity linear in the captured path and independent of the original email's size. This approach enables practical validation of expressive standard structures required for full DKIM-signed email verification. To fully integrate our constructions into ZK Email, we design complete end-to-end ZK circuits that combine (i) DKIM signature verification, (ii) an arbitrary-length SHA-256 circuit with partial precomputation for `rsa-sha256` under RFC 6376, and (iii) a general-purpose regex primitive enforcing structural constraints over email headers and body. We formalize the associated zero-knowledge relations and analyze their security under realistic adversary models. We implement the system (fully integrated with ZK Email and released under the MIT license) in `Circom` and `Noir`, targeting `Groth16` and `UltraHonk` backends, and evaluate it in both client-side and zkVM (SP1) deployment settings. Experimental results on commodity hardware demonstrate substantial efficiency improvements over prior DFA-based approaches, achieving a 2–6× speedup in proving time using the `UltraHonk` backend, while supporting a significantly richer class of regex languages.

Proceedings on Privacy Enhancing Technologies 2026(4), 740–771

https://doi.org/10.56553/popets-2026-0143

Keywords

Zero-knowledge proofs, regex, finite automata, email provenance

1 Introduction

Digital systems increasingly depend on structured user-submitted data, such as emails, for identity verification, tax and residency declarations, and KYC/AML compliance. In most deployments, however, such messages lack portable cryptographic guarantees that can be verified independently of the original service provider. Authenticity is therefore established through ad hoc checks or reliance on trusted intermediaries. This limitation is amplified by generative tools capable of producing synthetic yet highly realistic emails and other formally structured communications [30].

Email infrastructure already provides a widely deployed provenance mechanism through DomainKeys Identified Mail (DKIM) [1, 8]. A DKIM-signed email contains a verifiable signature issued by the sending domain over selected header fields and the message body. This signature can serve as a root of authenticity. In many emerging applications, however, email-derived claims must be verified in decentralized or publicly verifiable settings, such as smart contracts (e.g., user-friendly cryptocurrency wallet management) or shared execution environments (e.g., Cloud virtual machines) where no single party is trusted to perform the validation. In these contexts, naively transmitting the full content of a specific email and verifying its DKIM signature directly is impractical. The message size can lead to high data transmission costs, signature verification might be computationally expensive in constrained environments, and in most of the scenarios, applications often require selective disclosure of specific fields rather than full message publication.

An immediate mitigation is to rely on centralized attestations, whereby external providers validate emails or account ownership and issue confirmations. Although straightforward to deploy, this model introduces additional trust assumptions, privacy risks, and operational constraints. Verifiers often require direct access to parts of user's email account and may retain sensitive correspondence or become single points of failure. Enabling self-contained cryptographic proofs of origin and structural validity for email offers a more robust and independently verifiable alternative [4].

Recent work [4] has demonstrated that privacy-enhancing primitives such as zero-knowledge proofs (ZKPs) can integrate DKIM verification to enable selective and privacy-preserving email authentication. The objective is to allow a user to prove properties of a received email—such as sender authenticity, structural validity of headers, or the presence of specific fields—without disclosing the full message. Proof generation is performed client-side, while verification requires only a succinct proof and limited public inputs. Such a system enables trustless verification of DKIM-signed communications for applications including privacy-preserving identity proofs (e.g., account control or organizational membership), decentralized financial bridges and email-based wallets via on-chain verification of payment confirmations, and reusable anonymous KYC through selective disclosure [15]. It further can be employed to support pseudonymous legal and financial attestations, such as transaction receipts, tax filings, or proof of residence.

However, the main technical challenge of this approach is efficient enforcement of highly expressive, structured textual constraints required for DKIM compliance within the ZK domain. Email headers and bodies are defined by regular-language grammars and are commonly validated using complex regular expressions (regexes). Existing approaches to zero-knowledge regex verification [4, 11] present limitations in this context. Some incur prover costs that scale unfavorably with the input size [4], while others are optimized for applications that rely on skipping techniques enabled by efficient random access to the committed document [11]. Such assumptions are incompatible with the DKIM setting, where message integrity is bound to a SHA-256 digest and therefore inherits the sequential structure of the Merkle–Damgård construction. As a result, random-access optimizations cannot be directly applied without introducing additional mechanisms to establish consistency between the alternative commitment representation and the DKIM-signed SHA-256 digest, leading to substantial overhead.

We propose a general-purpose zero-knowledge regex primitive (ZK-Regex) specifically tailored to the requirements of DKIM verification. Given an ε-free nondeterministic finite automaton (NFA), ZK-Regex proves membership of a private string in the corresponding language with prover complexity linear in the length of an accepting run and independent of the size of the document. The construction decouples matching from verification: an accepting run is computed off-circuit by the prover, while the ZK circuit verifies correctness of the claimed state transitions and acceptance conditions. We integrate this primitive with DKIM signature verification to obtain end-to-end proofs for real-world emails.

We implement the proposed framework as a complete system comprising all zero-knowledge circuits required for DKIM signature and regex matching verification. To increase efficiency and practicality of our constructions, we designed an arbitrary-length `SHA-256` ZK circuit with support for partial precomputation (`rsa-sha256` under RFC 6376 [34]). The circuits are realized in two domain-specific ZK languages, `Circom` [16] and `Noir` [37], targeting the `groth16` [28] and `ultra_honk` [13] proving backends, respectively. At the application layer, the system includes smart contracts for on-chain verification and SDKs supporting custom circuit compilation, client-side proof generation, and selective disclosure workflows. All components are released as open-source under the MIT license.

We deploy and evaluate the system in realistic settings on commodity hardware (MacBook Pro M4 Pro) and formally reduce the security of our constructions to well-established cryptographic assumptions. Our benchmarks show that the proposed ZK-Regex technique delivers a 2–6× speedup in proving time compared to the previous approach on ZK Email [4] using the UltraHonk ZK backend, achieving end-to-end client-side ZK email proof generation in as low as 3.8 s and verification times of approximately 200 ms. We further evaluate deployability, including setup requirements and on-chain verification complexity, indicating low maintenance overhead and practical deployment costs.

The main contributions of this work are as follows.

- **ZKPs for DKIM verification.** We design and implement an end-to-end ZK construction for verifying DKIM-signed emails that is fully compatible with existing standards, enabling proofs of sender authenticity and structured message properties without revealing full email content.
- **Efficient dynamic hashing of large messages in ZK.** We introduce an arbitrary-length SHA-256 circuit with partial precomputation that reduces constraint overhead while remaining fully compliant with RFC 6376.
- **Input-size independent ZK regex verification.** To the best of our knowledge, we are the first to develop a DKIM-friendly ZK regex verification primitive whose circuit complexity scales linearly in the accepting path length and is independent of the document size.
- **Implementation and evaluation.** We provide a production-ready implementation in `Circom` and `Noir`, evaluate performance across `Groth16` and `UltraHonk` for client-side proving and a zkVM (SP1) [35] server-side proving. We release all of the circuits, contracts, and SDKs under the MIT license.

The remainder of the paper is organized as follows. Section 2 reviews the necessary background. Section 3 presents the protocol design. Section 4 defines the system and adversary models and analyzes the security of the top-level NIZK relations, including unforgeability, anonymity, and linkability. Section 5 describes implementation details and optimizations. Section 6 reports experimental results. Section 7 discusses related work.

2 Background

2.1 Finite Automata

Definition 2.1. (Deterministic Finite Automaton (DFA)) We formally define a DFA as a 5-tuple $M = \langle Q, \Sigma, \delta, q_0, F \rangle$, where Q is a finite set of states, indexed $0, \ldots, |Q| - 1$; Σ is a finite input alphabet (e.g. ASCII bytes); $\delta : Q \times \Sigma \to Q$ is the total transition function; $q_0 \in Q$ is the start state; and $F \subseteq Q$ is the set of accepting states.

A *transition table* is a tabular representation of δ:

$$\mathsf{table}[i][j] = k \quad \text{iff} \quad \delta(q_i, \sigma_j) = q_k,$$

where $\{\sigma_j\}$ enumerates Σ. Because δ is total and functional, each row has exactly $|\Sigma|$ entries.

Example. Consider the regular expression `ab*c` over $\Sigma = \{a, b, c\}$. Let us enumerate the states as $Q = \{q_0, q_1, q_2, q_{\text{dead}}\}$ with q_0 start, q_2 accepting, and q_{dead} a sink state. Figure 1 visualizes this DFA

using [5]. The transition matrix $T \in Q^{|Q|\times|\Sigma|}$ is:

$$T = \begin{array}{c|ccc} & a & b & c \\ \hline q_0 & q_1 & q_{\text{dead}} & q_{\text{dead}} \\ q_1 & q_{\text{dead}} & q_1 & q_2 \\ q_2 & q_{\text{dead}} & q_{\text{dead}} & q_{\text{dead}} \\ q_{\text{dead}} & q_{\text{dead}} & q_{\text{dead}} & q_{\text{dead}} \end{array}$$

Definition 2.2. (Nondeterministic Finite Automaton (NFA)) We formally define an NFA machine as a similar 5-tuple $M = \langle Q, \Sigma, \delta, q_0, F\rangle$, with the same Q, Σ, q_0, F as above, but where $\delta : Q \times \Sigma \to \mathcal{P}(Q)$.

Here, $\mathcal{P}(Q)$ denotes the *power set* of Q, and δ is a *transition relation*, not necessarily functional. From state $q \in Q$ and symbol $\sigma \in \Sigma$, there may be zero, one, or multiple next states $\{q' \mid (q, \sigma, q') \in \delta\}$.

Definition 2.3. (ε-NFA) An ε-NFA generalizes an NFA by allowing transitions on the empty string:

$$N_\varepsilon = \langle Q, \Sigma, \delta_\varepsilon, q_0, F\rangle, \quad \text{where} \quad \delta_\varepsilon : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q).$$

An ε-move (q, ε, q') consumes no input; the extended transition relation δ_ε^* is the reflexive transitive closure including such moves.

Definition 2.4. (Acceptance conditions) Let $w = a_1 a_2 \dots a_n$ be a string over the alphabet Σ. The automaton M accepts w if we could see a perfect, continuing path of state transitions: $q_0 \xrightarrow{a_1} q_i \xrightarrow{a_2} \cdots q_k$, where q_k is one of the possible acceptance states. In other words and more formally, there exists a sequence of states $r_0, r_1, \dots, r_n$ with the following properties:

(1) $r_0 = q_0$, (2) $\forall\, i \in [n] : \; r_{i+1} = \delta(r_i, a_{i+1})$, (3) $r_n \in F$.

The first condition specifies that the machine starts in the initial state q_0. The second condition requires that, for each input symbol a_i, the machine transitions to the next state according to δ. The final condition requires that, after reading the entire string, the automaton halts in an accepting state. If these conditions are met, M accepts w; otherwise, it rejects w. The set of strings that M accepts is called the *language recognized by* M, denoted $L(M)$.

Equivalently, acceptance can be defined using the extended transition function $\delta^* : Q \times \Sigma^* \to \mathcal{P}(Q)$:

$$\delta^*(r, \epsilon) = \{r\}, \qquad \forall (x \in \Sigma^*, a \in \Sigma) : \; \delta^*(r, xa) = \bigcup_{s \in \delta^*(r,x)} \delta(s, a).$$

Here, $\delta^*(r, x)$ is the set of all states reachable from r by consuming x. A string w is accepted if: $\delta^*(q_0, w) \cap F \neq \emptyset$.

Implementation Notes. For zero-knowledge circuit embedding, these automata can be represented using:

- a dense transition matrix for DFAs (constant-time lookup);
- per-state sparse lists or range compression for NFAs;
- flattened arrays with offsets for compact storage of Δ or Δ_ε.

Such representations allow circuit constraints to enforce valid state transitions succinctly while supporting cryptographic commitments to the automaton structure.

2.2 Verification of Email-Originated Data

Public-key cryptography provides a well-established method for verifying the authenticity of data: a party can confirm that data originated from a specific entity by verifying a digital signature created with the signing key of the entity. Although this principle underpins many secure systems, it has seen limited deployment in user-facing provenance workflows.

However, the electronic mail system offers an underutilized vector for cryptographic attestations. Most email traffic today includes domain-authenticated digital signatures via the DKIM standard. DKIM signs a canonicalized hash of selected email headers and message body using the domain's signing key. The core form of the signature can be summarized as follows.

```
rsa_sign(sha256(from:. . . , to:. . . , subject:. . . ,
        <body_hash>, . . . ), signing_key)
```

Since approximately 2017, major email providers have consistently signed outbound messages using DKIM. As a result, a received email from `someperson@mit.edu` to `you@yourdomain.com` can be cryptographically validated by fetching `mit.edu`'s public key from its DNS TXT records and verifying the DKIM signature.

Because signed headers include a hash of the body, one can also prove the content of the email. For example, the receipt of a Twitter verification email can be used to attest to ownership of a Twitter account, using only the DKIM header and relevant email content.

In the following, we formalize the syntax of the signature scheme specified by the DKIM standard as in Definition 2.5.

Definition 2.5 (DKIM-RSA Signature Scheme**).** We define this signature scheme as a tuple of algorithms following the standard [8, 31]:

- $(sk, pk) \xleftarrow{\$}$ DKIM-RSA.KeyGen(1^λ): The key generation algorithm takes as input a security parameter $\lambda \in \mathbb{N}$ and outputs a signing key sk and a public key pk.
- $\sigma \xleftarrow{\$}$ DKIM-RSA.Sign(sk, m): The signing algorithm takes as input a signing key sk and a message m and outputs a signature σ.
- $b \leftarrow$ DKIM-RSA.Vfy(pk, m, σ): The verification algorithm takes as input a public key pk, a message m, and a signature σ and outputs a bit $b \in \{0, 1\}$.

We require DKIM-RSA to satisfy *correctness* and *existential unforgeability under chosen-message attacks* (EUF-CMA) [31]. These properties are formally defined in Appendix D.

2.3 Regex for DKIM-Signed Content

Regular expressions are commonly used to safely extract structured information from DKIM-signed emails. Since DKIM signs canonicalized headers and body content, regex-based extraction enables fine-grained disclosure of authenticated data. Below we provide several representative examples.

1) Extracting the Sender Address. The following expression extracts the sender email address from the `From` header:

```
(?:\r\n|^)from:[^\r\n]*<([a-zA-Z0-9._%+-]+@[a
    -zA-Z0-9.-]+\.[a-zA-Z]{2,})>\r\n
```

The regex anchors either at the beginning of the header block or immediately after a CRLF sequence. It then matches the `from:` header and captures email address enclosed in angle brackets. This can be used where a user proves ownership of an email from a particular domain without revealing the remaining message contents.

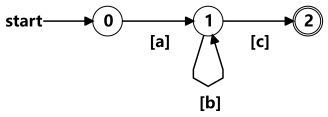


Figure 1: DFA for the Regex ab*c, generated by [5].

2) Extracting the DKIM Body Hash. The following expression extracts the bh field from the `DKIM-Signature` header:

```
(?:\r\n|^)dkim-signature:(?:[a-z]+=[^;]+; )+
    bh=([a-zA-Z0-9+/=]+);
```

The pattern iterates through DKIM tag-value pairs until reaching the bh= field, whose value is captured. The bh uniquely binds the signed email body to DKIM signature and can therefore be used to prove integrity of an email body without disclosing body itself.

3) Extracting the DKIM Timestamp. The following expression extracts the signing timestamp from the `DKIM-Signature` header:

```
(?:\r\n|^)dkim-signature:(?:[a-z]+=[^;]+; )+t
    =([0-9]+);
```

The captured value corresponds to the DKIM signing timestamp encoded in Unix time format. This field may be used in applications requiring temporal validity proofs, such as demonstrating that a signed email existed before a specific deadline or event.

2.4 Non-Interactive Zero-Knowledge Argument Systems

We follow the formulation of Groth [28] to define non-interactive zero-knowledge (NIZK) argument systems. In our security analysis, we use a restricted oracle-aided knowledge-soundness notion, in the spirit of O-SNARK [27]: the adversary may adaptively query an oracle $\mathcal{O}$, and the extractor receives the resulting oracle transcript in addition to non-black-box access to the adversary's state and random coins.

Before giving the definition, we fix the notation for relations and oracle families. Let $\mathcal{R}$ be a relation generator that, on input 1^λ for a security parameter $\lambda \in \mathbb{N}$, samples a binary relation R decidable in polynomial time together with auxiliary information z. For any pair $(\phi, w) \in R$, we call ϕ the statement and w the witness. Let $\mathcal{R}_\lambda$ denote the support of relations that may be output by $\mathcal{R}(1^\lambda)$. Let $\mathbb{O}$ be a family of oracles. We write $O \xleftarrow{\$} \mathbb{O}$ to denote sampling an oracle O from this family.

Definition 2.6. We define an NIZK argument system for $\mathcal{R}$ as a quadruple of (probabilistic) polynomial algorithms NIZK:=(Setup, Prv, Vfy, Sim) with the following syntax:

- $(\mathrm{crs}, \mathrm{td}) \xleftarrow{\$} \mathrm{Setup}(R)$: it takes as input an NP relation R and outputs a common reference string (CRS) crs and a trapdoor td.
- $\pi \xleftarrow{\$} \mathrm{Prv}(R, \mathrm{crs}, \phi, w)$: it takes as input an NP relation R, a CRS crs, a statement ϕ, and a witness w and outputs a proof π.
- $b \leftarrow \mathrm{Vfy}(R, \mathrm{crs}, \phi, \pi)$: it takes as input an NP relation R, a CRS crs, a statement ϕ, and a proof π and outputs $b \in \{0, 1\}$.
- $\pi \xleftarrow{\$} \mathrm{Sim}(R, \mathrm{td}, \phi)$: it takes as input an NP relation R, a trapdoor td, and a statement ϕ, and outputs a proof π.

The properties of NIZK argument systems used in this work, namely perfect completeness, perfect zero-knowledge, and knowledge soundness with respect to an oracle family $\mathbb{O}$, are defined in Appendix E.

We require knowledge soundness in the presence of a signing oracle for the DKIM-RSA signature scheme, defined as follows:

Definition 2.7 (Signing Oracle)**.** Let $\mathbb{O}_{\mathrm{DKIM}}$ denote the family of signing oracles with respect to a security parameter $\lambda \in \mathbb{N}$ and a DKIM-RSA signature scheme (Definition 2.5). Each oracle $O_{\mathrm{DKIM}} \xleftarrow{\$} \mathbb{O}_{\mathrm{DKIM}}$ works as follows.

(1) Upon initialization, it runs $(sk, pk) \xleftarrow{\$} \mathrm{DKIM\text{-}RSA.KeyGen}(1^\lambda)$ and stores the resulting key pair.
(2) On the special input "pk", it returns the stored public key pk.
(3) On each message query m, the oracle returns a signature $\sigma \xleftarrow{\$} \mathrm{DKIM\text{-}RSA.Sign}(sk, m)$.

For our concrete Groth16 and UltraHonk instantiations of the NIZK argument system, we conjecture that knowledge soundness continues to hold even when the adversary is given access to the signing oracle O_{DKIM}; we discuss the status and justification of this conjecture in Appendix E.1.

3 Constraints for DKIM-Signed Emails

We assume a setting where the prover aims to demonstrate receipt of an email from a known domain (e.g., `google.com`), prove that its message body satisfies a given regular expression, and selectively reveal specific portions of the message. We construct an NP relation R_{Email} that verifies the authenticity and structure of an email protected under the DKIM protocol. The prover demonstrates both that the signature is cryptographically valid and that the email is syntactically valid, using an embedded regex verifier. To do that, the prover generates a ZK proof π_σ for the **statement** $\phi = (pk_{\mathrm{srvr}}, m^*, \mathrm{rgx}_{\mathrm{ptrn}})$, where pk_{srvr} is the public key of the mail server[1], $\mathrm{rgx}_{\mathrm{ptrn}}$ is a masking regex tailored to the expected format of a message m, producing a regex-consistent view of the message m^*, and **witness** $w = (\sigma, m)$, consisting of a DKIM signature σ and a message m in the email. Formally, the relation R_{Email} returned by $\mathcal{R}_{\mathrm{Email}}$ is defined as follows:

$$R_{\mathrm{Email}} := \begin{pmatrix} \phi{=}(pk_{\mathrm{srvr}}, m^*, \mathrm{rgx}_{\mathrm{ptrn}}), \\ w{=}(\sigma, m) \end{pmatrix}, \tag{1}$$

$(\phi, w) \in R_{\mathrm{Email}} \Leftrightarrow$

$\mathrm{DKIM\text{-}RSA.Vfy}(pk_{\mathrm{srvr}}, m, \sigma) = 1 \wedge\ m^* = \mathrm{match}(m, \mathrm{rgx}_{\mathrm{ptrn}})$,

where the function $m^* = \mathrm{match}(m, \mathrm{rgx}_{\mathrm{ptrn}})$ is introduced immediately below.

More precisely, relation R_{Email} checks the following conditions:

(1) **Common DKIM verification:** The constraints that are independent of the user-defined regex pattern $\mathrm{rgx}_{\mathrm{ptrn}}$ are

[1]The sender's address follows the standard `username@domain` format, where the `domain` is represented by the mail server's public key pk_{srvr}, whose corresponding signing key sk_{srvr} signs the message. In practice, the public key also contains corresponding RSA modulus and public exponents pp_{RSA}, published in the DNS TXT records as part of the DKIM public key standard.

EmailVerifier(m=$\langle Hd, Bd\rangle$, ℓ_m=$\langle \ell_h, \ell_b\rangle$, σ, pk_{srvr}):

1: **assert** ValidFormat$(Hd, Bd, \ell_h, \ell_b) = 1$;
2: $h_H \leftarrow \text{SHA256}(Hd, \ell_h)$;
3: **assert** RSA.Verify$(h_H, \sigma, pk_{\text{srvr}}) = 1$;
4: $bh_H \leftarrow \text{ExtractBodyHash}(Hd)$;
5: $bh_B \leftarrow \text{SHA256}(Bd, \ell_b)$;
6: **assert** $bh_H = bh_B$;

Figure 2: High-level construction of the VerifyDKIM **circuit.**

represented by a VerifyDKIM circuit, whose high-level construction is provided in Figure 2. Concretely, the verifier checks that the SHA-256 hash of the message m [2] and the RSA signature[3] verification both succeed according to PKCS#1 v1.5 [31], while also ensuring that the message satisfies the canonicalization and formatting requirements defined by the DKIM standard. This validation procedure is implemented by checking whether the message m conforms to a fixed regex pattern rgx_{DKIM}, which captures the syntactic structure mandated by the DKIM specification. Listing 2 in Appendix A presents a high-level view of the corresponding circuit realization.

(2) **User-defined regex matching and capturing:** The constraints that depend on the user-defined regex pattern rgx_{ptrn} ensure that the masked message m^* revealed in the instance ϕ is obtained by applying the matching and capturing regex pattern rgx_{ptrn} to m. In what follows, we denote such capturing by the deterministic function $m^* = \text{match}(m, \text{rgx}_{\text{ptrn}})$.

3.1 Arbitrary-Length Hash Verification

A central challenge in supporting general email verification is handling variable-length message bodies within a single verifier circuit. To this end, we extend the `circomlib`[2] implementation of SHA-256 to support arbitrary-length inputs up to a configurable bound.

To mitigate the prohibitive constraint cost of hashing entire messages on-chain, we employ a modular precomputation strategy inspired by Merkle–Damgård and sponge constructions[17, 22]. Let the email body be split into blocks $m_1, \ldots, m_t$. The hash is computed iteratively as $h_i = H(h_{i-1}, m_i)$. The prover computes h_{i-1} off-circuit for all but the final few blocks and provides h_{i-1} as a public input. The circuit then verifies that applying SHA-256 to the final sequence of blocks produces the claimed digest. Formally, the circuit proves knowledge of $m_k, \ldots, m_t$ such that:

$$H^{(t-k+1)}(h_{k-1}, m_k, \ldots, m_t) = h_t.$$

It is important to note that this optimization preserves both completeness and soundness. While the intermediate chaining value h_{k-1} is supplied as a public input, the circuit verifies that it correctly extends to the DKIM-bound final digest by checking $H^{(t-k+1)}(\cdot) = h_t$, where h_t is the final SHA-256 digest authenticated by DKIM signature. Thus, h_{k-1} is not assumed to be trustworthy in isolation. Rather, soundness follows from the requirement that any prover must demonstrate a valid continuation from h_{k-1} to the signed digest h_t. Consequently, an adversary cannot benefit from supplying an arbitrary intermediate state unless it can also construct a corresponding message suffix that yields the same DKIM-bound digest.

3.2 Regex Matching in Zero Knowledge

Correct parsing of email headers requires more than substring detection. The verifier must ensure that specific fields (e.g., `From`, `To`, `Subject`) appear in valid positions and are delimited by canonical separators such as `\r\n`. Naïve substring checks are insufficient, as an adversary may embed misleading header-like strings within other fields (e.g., inside the message body). To enforce proper parsing semantics, we require zero-knowledge verification of regular-language constraints over private text. This enables us to prove the presence of structured patterns in private text without revealing the text itself. This primitive enables privacy-preserving proofs of account ownership, transaction confirmation, regulatory notice receipt, and secure recovery workflows.

3.2.1 ***Direct DFA Encoding (Baseline Approach).*** A straightforward method for ZK regex verification is to compile the regular expression into a DFA and embed its transition function directly into the arithmetic circuit. Prior work [4] follows this approach. Each transition is encoded as arithmetic constraints, and the circuit simulates the DFA step-by-step over the input string.

This design supports selective disclosure by decomposing the entire regex into smaller sub-patterns ("*part regexes*") and deciding on the visibility of each independently, e.g. as *public*/*private* and therefore, *exposing*/*hiding* the corresponding matched substrings. Figure 3 illustrates this approach. The DFA structure is embedded directly into the arithmetic circuit, enabling in-circuit evaluation of transitions. Listing 1 provides an example of dividing the input regex into multiple parts with varying configurations. We provide more details on the DFA-to-circuit translation technique used in [4] in Appendix B.

However, directly encoding the DFA transition table introduces structural limitations. The circuit size scales with both the number of DFA states $|Q|$ and the input length $|w|$, since each character requires evaluation against the full transition logic. For complex expressions, this results in large constraint systems and high proving costs. Furthermore, DFA-based constructions restrict support for richer regex features, such as non-greedy quantifiers (`*?` or `+?`), alternations (`|`), and require careful manual composition when combining multiple sub-patterns to preserve semantic equivalence [4].

```
{ "parts": [
    { "is_public": false,
      "regex_def": "email:" },
    { "is_public": true,
      "regex_def": "[a-z]+@[a-z]+.com" },
    { "is_public": false,
      "regex_def": "." } ] }
```

Listing 1: Example JSON structure with regex definitions.

[2]The VerifyDKIM circuit additionally takes the message length ℓ_m as input, and the message is zero-padded to its maximum size because the input size of the circuit must be fixed. In the NP relation $\mathcal{R}_{\text{Email}}$, we interpret the message m as implicitly including ℓ_m.

[3]In practice, the verifier (for instance, a smart contract) additionally confirms that the RSA public key pk_{srvr} included in the public statement matches the legitimate DKIM key that the sender's domain has published in DNS.

3.2.2 ***Off-Circuit Matching, In-Circuit Verification.*** In the proposed construction, we avoid simulating the full automaton inside the circuit. Instead, we separate matching from verification. Figure 4 illustrates the architectural differences between the baseline and proposed approaches. The proposed method consists of three main steps, as follows.

1) Regex Compilation: The input regex is compiled into an NFA and a corresponding verification circuit. NFA structure is derived using a Thompson-style construction [41] via `regex_automata` library [3], supporting ε-transitions for native capture group boundaries.

2) Off-Circuit Matching: The prover runs the NFA on the input text to compute a match and produces a traversal path: a sequence of transitions (q_i, c_i, q_{i+1}), i.e. (`current_state`, `input_byte`, `next_state`) that should be accepted by the NFA machine:

$$\big((p_0, a_1, p_1), (p_1, a_2, p_2), \ldots, (p_{k-1}, a_k, p_k)\big),$$

with $p_0 = q_0'$, $p_k \in F'$, and substring $y = a_1 a_2 \ldots a_k$ of the original sequence x.

3) In-Circuit Verification: The circuit takes the claimed match and path as private inputs and verifies the exact acceptance criteria formalized in Definition 2.4 as follows:

- the path starts in a valid initial state,
- each claimed transition is valid according to NFA rules,
- the state sequence is contiguous, and
- the final state is accepting.

Formally, the circuit verifies the following NP relation:

$$R_{\text{regex}} := \begin{pmatrix} \phi{=}(C_{[c_1,\ldots,c_k]}), \\ w{=}(w_{[a_1,\ldots,a_k]}, P_{[p_0,\ldots,p_k]}) \end{pmatrix}, \tag{2}$$
$$(\phi, w) \in R_{\text{regex}} \Leftrightarrow p_0 = q_0 \ \wedge \ p_k \in F$$
$$\wedge \ \forall i \in [k] : (p_{i-1}, a_i, p_i) \in \delta \ \wedge \ c_i = \text{Cap}(a_i, p_{i-1}).$$

Here, the only data in the statement ϕ is the vector $C = [c_1, \ldots, c_k]$, representing the claimed capture-group activity for each transition in the matched path. The private witness consists of the full regex-compatible string $w = [a_1, \ldots, a_k]$ and a valid state path $P = [p_0, \ldots, p_k]$ such that the sequence of transitions follows the automaton and terminates in an accepting state ($p_k \in F$). The function $\text{Cap}(a_i, p_{i-1})$ extracts the capture information associated with symbol a_i at state p_{i-1}, as defined by the regex semantics.

The constraint on $m^* = \text{match}(m, \text{rgx}_{\text{ptrn}})$ is enforced by verifying the NP relation R_{regex}. This prevents the following attack that attempts to forge the regex matching result.

- *Invalid transition injection:* Each claimed transition is checked against the immutable transition relation embedded in the circuit, preventing the introduction of non-existent edges.
- *Input substitution:* Transitions are verified with respect to the provided byte sequence, binding the proof to a specific input and preventing reuse across different messages.
- *Capture group manipulation:* Capture boundaries are enforced by the NFA structure and validated at each transition, preventing tampering with selectively revealed substrings.

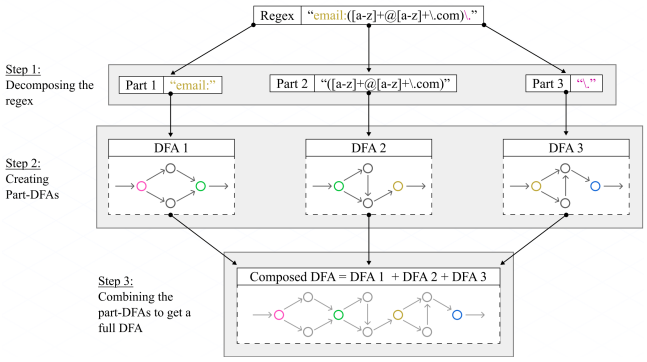


Figure 3: [Baseline approach] Decomposition of a regex into part-DFAs with varying visibility.

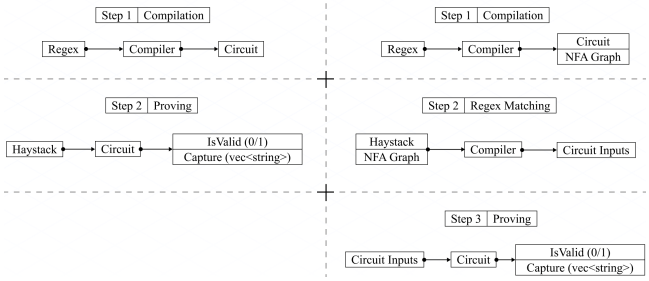


Figure 4: ZK Regex pipeline in baseline (left side) vs. the proposed approach (right side).

4 Security Analysis

4.1 System and Threat Model

We assume that an adversary $\mathcal{A}$ may corrupt and fully control a prover attempting to convince an honest verifier of a false claim about a target email. Consistent with real-world deployments, we assume that the sending and receiving mail servers are honest and uncompromised; in particular, their signing keys and authentication mechanisms remain secure. When controlling an entity, $\mathcal{A}$ may:

- Observe, intercept, modify, delay, or inject arbitrary messages on behalf of the corrupted actor.
- Attempt to forge zero-knowledge proofs.
- Mount substitution attacks by generating proofs that bind to a different account than the one that legitimately sent or received the email.
- Produce multiple conflicting proofs for the same email or replay previously valid proofs in a different context.
- Adaptively choose inputs, queries, and proof statements in order to maximize the probability that an honest verifier accepts without knowledge of a valid witness.
- Attempt to extract or infer sensitive information from proofs generated by honest provers.

We assume that $\mathcal{A}$ cannot:

- Break the existential unforgeability of the signature scheme underlying DKIM. In our instantiation, DKIM uses RSA

PKCS#1 v1.5 [31], and we therefore assume EUF-CMA[4] security of the deployed DKIM-RSA scheme. We formally define this scheme in Appendix D.
- Violate properties of the underlying NIZK argument systems, e.g., Groth16 [28] or UltraHonk [13], as stated in Subsection 2.4.

Furthermore, we assume a realistic deployment setting consistent with the current email ecosystem. In particular, DKIM public keys are obtained from DNS records and are assumed to correspond to the legitimate domain unless otherwise stated [34]. We do not assume universal deployment of DNSSEC, and therefore key authenticity may rely on auxiliary trust mechanisms, such as governance-based validation or hard-coded key registries, depending on the deployment model [36]. We also assume that mail servers operate according to the standard DKIM specification, including correct canonicalization and signature generation. Finally, we acknowledge that real-world deployments introduce additional trust and governance considerations, including DNS key rotation, censorship resistance, decentralized key validation, and limitations arising from mail server visibility [12, 38, 43]. For conciseness, we omit a detailed treatment of these operational assumptions in the main body of the paper. A comprehensive discussion of the system model, deployment considerations, and trust assumptions is provided in Appendix C.

4.2 Correctness

We say that our construction is correct if for all $\lambda \in \mathbb{N}$, $R_{\mathsf{Email}} \in \mathcal{R}_{\mathsf{Email},\lambda}$, all regex patterns $\mathsf{rgx}_{\mathsf{ptrn}}$, and all messages m, the following holds:

$$\mathsf{NIZK.Vfy}\left(R_{\mathsf{Email}}, \mathsf{crs}, \left(pk_{\mathsf{srvr}}, m^*, \mathsf{rgx}_{\mathsf{ptrn}}\right), \pi\right) = 1, \tag{3}$$

where

$$(\mathsf{crs}, \mathsf{td}) \xleftarrow{\$} \mathsf{NIZK.Setup}(R_{\mathsf{Email}}),$$
$$(sk_{\mathsf{srvr}}, pk_{\mathsf{srvr}}) \xleftarrow{\$} \mathsf{DKIM\text{-}RSA.KeyGen}(1^\lambda),$$
$$m^* = \mathsf{match}(m, \mathsf{rgx}_{\mathsf{ptrn}}),\ \sigma \leftarrow \mathsf{DKIM\text{-}RSA.Sign}(sk_{\mathsf{srvr}}, m),$$
$$\pi \xleftarrow{\$} \mathsf{NIZK.Prv}\left(R_{\mathsf{Email}}, \mathsf{crs}, \left(pk_{\mathsf{srvr}}, m^*, \mathsf{rgx}_{\mathsf{ptrn}}\right), (\sigma, m)\right).$$

By the correctness of the DKIM-RSA scheme, the above instance and witness satisfy R_{Email}. Hence, by the perfect completeness of the NIZK argument system, Equation (3) holds.

4.3 Unforgeability

We establish the unforgeability of our construction by analyzing the forgery probability of the top-level NIZK relation $\mathcal{R}_{\mathsf{Email}}$. Here, we provide only a sketch of the formal analysis used to prove the theorems. Full proofs appear in Appendix F.

Theorem 1. *Assuming that the DKIM-RSA scheme satisfies EUF-CMA security (Definition 2.5), and the NIZK argument system is knowledge-sound (Definition E.3) with respect to the signing oracle family $\mathbb{O}_{DKIM}$ in Definition 2.7, our construction is unforgeable.*

[4]Existential Unforgeability under Chosen Message Attack

$\mathsf{GameUnforge}^{\mathsf{Email}}_{\mathcal{A}}(1^\lambda, \mathsf{rgx}_{\mathsf{ptrn}})$:

1 : $(R_{\mathsf{Email}}, z) \xleftarrow{\$} \mathcal{R}_{\mathsf{Email}}(1^\lambda), O_{\mathsf{DKIM}} \xleftarrow{\$} \mathbb{O}_{\mathsf{DKIM}}$;
2 : $(\mathsf{crs}, \mathsf{td}) \xleftarrow{\$} \mathsf{NIZK.Setup}(R_{\mathsf{Email}})$;
3 : $pk_{\mathsf{srvr}} \leftarrow O_{\mathsf{DKIM}}(\text{"pk"})$;
4 : $(\pi^*, m^*) \xleftarrow{\$} \mathcal{A}^{O_{\mathsf{DKIM}}(\cdot)}(\mathsf{rgx}_{\mathsf{ptrn}}, R_{\mathsf{Email}}, z, \mathsf{crs}, pk_{\mathsf{srvr}})$;
5 : $\phi \leftarrow \left(pk_{\mathsf{srvr}}, m^*, \mathsf{rgx}_{\mathsf{ptrn}}\right)$;
6 : **return** $\mathsf{NIZK.Vfy}(R_{\mathsf{Email}}, \mathsf{crs}, \phi, \pi^*) = 1 \wedge$
7 : $\forall (m, \cdot) \in \mathsf{qt} : \ m^* \neq \mathsf{match}(m, \mathsf{rgx}_{\mathsf{ptrn}})$;

Figure 5: Game-based definition of unforgeability for our construction, where $\mathsf{qt} = \{m_i, O_{\mathsf{DKIM}}(m_i)\}$ denotes the oracle transcript as defined in Definition E.3.

Formally, for all security parameters $\lambda \in \mathbb{N}$, non-uniform polynomial-time adversaries $\mathcal{A}$, and regex patterns $\mathsf{rgx}_{\mathsf{ptrn}}$, it holds that:

$$\Pr[\mathsf{GameUnforge}^{\mathsf{Email}}_{\mathcal{A}}(1^\lambda, \mathsf{rgx}_{\mathsf{ptrn}}) = 1] \leq \mathsf{negl}(\lambda),$$

where $\mathsf{GameUnforge}^{\mathsf{Email}}_{\mathcal{A}}(1^\lambda, \mathsf{rgx}_{\mathsf{ptrn}})$ is defined in Figure 5, and $\mathcal{A}$ can make at most Q queries to the oracle O_{DKIM}.

Sketch. By the knowledge-soundness of the NIZK argument system, no adversary can produce an accepting proof without knowledge of a valid witness for the specified relation R_{Email}. This implies that $\mathcal{A}$ must know a signature $\sigma_{m'}$ on the challenge message m'. By EUF-CMA security of the DKIM-RSA scheme, since $\mathcal{A}$ does not have access to the signing key sk_{srvr}, $\mathcal{A}$ cannot produce a valid $\sigma_{m'}$ without submitting m' to the oracle O_{DKIM}. Therefore, $\mathcal{A}$ cannot forge a proof π^* that satisfies the conditions on both lines 6 and 7. In Appendix F, we prove this by constructing an adversary that breaks EUF-CMA security by internally invoking another adversary that wins $\mathsf{GameUnforge}^{\mathsf{Email}}_{\mathcal{A}}(1^\lambda, \mathsf{rgx}_{\mathsf{ptrn}})$ with non-negligible probability. □

4.4 Anonymity

We require that the proposed scheme provide anonymity in the following sense: given two valid claims produced by two different users, if the publicly disclosed message component m^*, i.e. captured group, is identical in both cases, then no PPT adversary can distinguish between the proofs. An example of such two different messages with the same captured content is as follows:[5]

```
(?:\r\n|^)from:[^\r\n]*<[a-zA-Z0-9._%+-]+@([a
    -zA-Z0-9.-]+\.[a-zA-Z]{2,})>\r\n
```

As in the unforgeability analysis, we provide only a proof sketch here, while the complete proofs are given in Appendix F.

Theorem 2. *Assuming that the NIZK argument system is zero-knowledge, our construction fulfills anonymity. Formally, for all security parameters $\lambda \in \mathbb{N}$, adversaries $\mathcal{A}$, and regex patterns $\mathsf{rgx}_{\mathsf{ptrn}}$, it*

[5]Note the contrast with the examples in Section 2.3, which use a different capture group due to a different placement of parentheses.

$\mathsf{GameAnon}^{\mathsf{Email}}_{\mathcal{A}}(1^\lambda, \mathsf{rgx}_{\mathsf{ptrn}})$:

$(R_{\mathsf{Email}}, z) \xleftarrow{\$} \mathcal{R}_{\mathsf{Email}}(1^\lambda)$;
$(\mathsf{crs}, \mathsf{td}) \xleftarrow{\$} \mathsf{NIZK.Setup}(R_{\mathsf{Email}})$;
$(sk_{\mathsf{srvr}}, pk_{\mathsf{srvr}}) \xleftarrow{\$} \mathsf{DKIM\text{-}RSA.KeyGen}(1^\lambda)$;
$\langle m^*, m_0, m_1 \rangle \xleftarrow{\$} \mathcal{A}(\mathsf{rgx}_{\mathsf{ptrn}}, R_{\mathsf{Email}}, z, \mathsf{crs}, pk_{\mathsf{srvr}})$;
assert $m^* = \mathsf{match}(m_0, \mathsf{rgx}_{\mathsf{ptrn}}) = \mathsf{match}(m_1, \mathsf{rgx}_{\mathsf{ptrn}})$;
$b \xleftarrow{\$} \{0, 1\}$;
$\sigma \xleftarrow{\$} \mathsf{DKIM\text{-}RSA.Sign}(sk_{\mathsf{srvr}}, m_b)$;
$\phi \leftarrow \left(pk_{\mathsf{srvr}}, m^*, \mathsf{rgx}_{\mathsf{ptrn}}\right),\ w \leftarrow (\sigma, m_b)$;
$\pi \xleftarrow{\$} \mathsf{NIZK.Prv}(R_{\mathsf{Email}}, \mathsf{crs}, \phi, w)$;
$b^* \xleftarrow{\$} \mathcal{A}(\pi)$;
return $b == b^*$;

Figure 6: Game-based definition of anonymity for our construction.

holds that:

$$\left|\Pr[\mathsf{GameAnon}^{\mathsf{Email}}_{\mathcal{A}}(1^\lambda, \mathsf{rgx}_{\mathsf{ptrn}}) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda),$$

where $\mathsf{GameAnon}^{\mathsf{Email}}_{\mathcal{A}}(1^\lambda, \mathsf{rgx}_{\mathsf{ptrn}})$ *is defined in Figure 6.*

Sketch. We need to prove that two valid proofs of two different original messages m_0 and m_1, both satisfying the regex pattern $\mathsf{rgx}_{\mathsf{ptrn}}$, are indistinguishable from each other. By the zero-knowledge property of the NIZK argument system, the proof π reveals nothing beyond the associated public statement. Because the public statement ϕ is independent of the chosen message m_b in the corresponding witness w, the distribution of the resulting proof π does not depend on b. Thus, no adversary can win the game with non-negligible advantage, implying the anonymity of the construction.

□

4.5 Linkability

Finally, we require that our scheme is linkable, implying that an adversary cannot create two different extracted (captured) messages $m_0^* \neq m_1^*$, given the same original message m and matching the same $\mathsf{rgx}_{\mathsf{ptrn}}$. We provide the complete proofs in Appendix F.

Theorem 3. *Assuming that the DKIM-RSA scheme satisfies EUF-CMA security (Definition 2.5), and the NIZK argument system is knowledge-sound (Definition E.3) with respect to the signing oracle family $\mathbb{O}_{DKIM}$ in Definition 2.7, our construction fulfills linkability. Formally, for all security parameters $\lambda \in \mathbb{N}$, non-uniform polynomial-time adversaries $\mathcal{A}$, and regex patterns $\mathsf{rgx}_{\mathsf{ptrn}}$, it holds that:*

$$\Pr[\mathsf{GameLink}^{\mathsf{Email}}_{\mathcal{A}}(1^\lambda, \mathsf{rgx}_{\mathsf{ptrn}}) = 1] \leq \mathsf{negl}(\lambda),$$

where $\mathsf{GameLink}^{\mathsf{Email}}_{\mathcal{A}}(1^\lambda, \mathsf{rgx}_{\mathsf{ptrn}})$ *is defined in Figure 7.*

Sketch. Recall that, under the same assumptions, our construction is unforgeable, as stated in Theorem 1. We prove that this unforgeability implies that no adversary can win GameLink with non-negligible probability. This is shown by constructing an adversary $\mathcal{B}$ for GameUnforge that internally invokes the adversary $\mathcal{A}$ for GameLink and simulates its view. □

$\mathsf{GameLink}^{\mathsf{Email}}_{\mathcal{A}}(1^\lambda, \mathsf{rgx}_{\mathsf{ptrn}})$:

$(R_{\mathsf{Email}}, z) \xleftarrow{\$} \mathcal{R}_{\mathsf{Email}}(1^\lambda)$;
$(\mathsf{crs}, \mathsf{td}) \xleftarrow{\$} \mathsf{NIZK.Setup}(R_{\mathsf{Email}})$;
$(sk_{\mathsf{srvr}}, pk_{\mathsf{srvr}}) \xleftarrow{\$} \mathsf{DKIM\text{-}RSA.KeyGen}(1^\lambda)$;
$m \xleftarrow{\$} \mathcal{A}(\mathsf{rgx}_{\mathsf{ptrn}}, R_{\mathsf{Email}}, z, \mathsf{crs}, pk_{\mathsf{srvr}})$;
$\sigma \leftarrow \mathsf{DKIM\text{-}RSA.Sign}(sk_{\mathsf{srvr}}, m)$;
$(\pi_0^*, \pi_1^*, m_0^*, m_1^*) \xleftarrow{\$} \mathcal{A}(\sigma)$;
$\forall\, b \in \{0, 1\} \quad \phi_b \leftarrow \left(pk_{\mathsf{srvr}}, m_b^*, \mathsf{rgx}_{\mathsf{ptrn}}\right)$;
return $(m_0^* \neq m_1^*) \wedge$
$\left(\forall\, b \in \{0, 1\} : \mathsf{NIZK.Vfy}(R_{\mathsf{Email}}, \mathsf{crs}, \phi_b, \pi_b^*) = 1\right)$;

Figure 7: Game-based definition of linkability for our construction.

5 Implementation Details

Encoding Strategy for `Noir`. `Noir` employs a sparse array data structure to efficiently encode and verify transition rules. A sparse array stores only non-zero entries together with their indices, making it well-suited for NFAs, where most state-to-state transitions are undefined. Each valid transition (s, b, s') is uniquely mapped to an index key via a *Random Linear Combination (RLC)*:

$$\mathsf{key} = s + b \cdot 257 + s' \cdot 257^2.$$

This encoding provides three key properties:

- **Unique positioning:** Each component occupies a distinct digit position in base-257 arithmetic.
- **Collision-freedom:** Since $257 > 255$ (the maximum byte value), no two distinct transitions produce the same key.
- **Constant-time lookup:** The circuit can check a claimed transition by computing its key and querying the sparse array at that index.

For example, the transition $(s = 5, b = 65, s' = 12)$ yields

$$\mathsf{key} = 5 + 65 \cdot 257 + 12 \cdot 257^2 = 807{,}378.$$

Each sparse array entry stores a validity flag and any associated capture-group markers, enabling $O(1)$ lookup time rather than iterating through all possible transitions. This strategy substantially reduces circuit complexity for regexes with large transition sets.

Performance Characteristics. The total number of transitions is proportional to the regex complexity (e.g., use of character classes, alternations, and repetitions). The performance implications differ across circuit DSLs:

- **Circom:** Here the circuit size scales as $O(\mathit{match\_length} \times \mathit{total\_transitions})$, since each input byte requires iterating over all possible transitions.

- **Noir:** Circuit size scales as $O(\textit{match\_length} \times \textit{lookup\_cost})$, where the lookup cost is effectively constant due to the direct key-based access of the sparse array.

For regexes with large transition sets, `Noir`'s sparse-array encoding yields significantly more efficient circuits compared to the exhaustive iteration model used in `Circom`.

5.1 Proposed zk-Regex Capabilities

The proposed zk-Regex architecture introduces several significant improvements over the *baseline* [4] in terms of expressiveness, security, performance, and modularity, by decoupling regex matching from in-circuit verification.

Expanded Regex Support. The primary advantage of the proposed scheme is its ability to support a much wider range of regular expression syntax. By moving the complex matching logic off-circuit, proposed zk-Regex can handle any pattern that the `regex_automata` library can compile into a NFA. This enables robust support for features that were difficult or impossible to implement in the baseline [4] approach's direct-to-circuit model, including:

- Non-greedy quantifiers (e.g., `*?`, `+?`);
- Complex alternations (`|`) and nested grouping;
- Unicode character classes and properties (e.g., `\p{L}`).

We provide a detailed comparison of different regex patterns supported by each method in Appendix J (See Table 16 for different patterns). This approach also eliminates the need for the manual "decomposed regex" strategy from baseline. Users can now specify a single, complete regex pattern—simplifying circuit construction and reducing potential implementation errors.

Unsupported Features. zk-Regex inherits the deliberate design constraints of `regex_automata` [3] library, which omits certain features to ensure linear-time matching performance. As a result, the proposed zk-regex does not support:

- Lookarounds (e.g., `(?=...)`, `(?!...)`, `(?<=...)`, `(?<!...)`);
- Backreferences (e.g., `\1`, `\k<name>`);
- Other complex PCRE [7] constructs such as atomic groups.

Regex patterns containing these features will fail to compile.

Performance and Scalability. The proposed method architecture fundamentally changes the scaling behavior of zk-Regex:

- **Decoupled complexity:** Circuit size is no longer tied to regex complexity, but primarily to the maximum match length and the number of capture groups.
- **Targeted proving:** The proving cost scales with the length of the matched substring, not the full haystack.
- **Efficient resource use:** CPU-intensive matching occurs off-circuit, while circuit handles only path verification.

Although direct quantitative comparison with the *baseline* approach is non-trivial due to architectural differences, replacing in-circuit automaton simulation with explicit path verification yields substantially improved scalability for complex regular expressions. We report comprehensive benchmark results in Appendix J.

Proving-System-Agnostic Core. The Rust-based compiler underlying the proposed zk-Regex framework performs regex parsing, NFA construction, and ε-elimination. The design is proving-system agnostic and produces a standardized intermediate representation, `NFAGraph`, which can be compiled into backend-specific formats for systems such as `Circom` and `Noir`. The same implementation can also be executed directly within zkVM environments, including SP1. Additional details on supported proving backends and their evaluation are provided in Appendix H.

Simplified and Accurate Capture Groups. In *baseline* [4], capture groups were implemented through manual regex decomposition. In our method, they are handled natively by the NFA, and therefore, capture groups are intrinsic to the NFA produced by `regex_automata` as the epsilon elimination preserves their start and end boundaries. This ensures semantically correct and predictable behavior consistent with standard regex engines [29].

5.2 Epsilon-Free NFA Generation

To make NFA verification efficient within a ZK circuit, the automaton must be transformed into an equivalent one that does not use ε-transitions. Here we describe the transformation process.

From Regex to an Initial NFA. The process begins by constructing a NFA from the input regex string. Proposed `zk-Regex` employs Rust's `regex_automata` library [3] for this task. This approach provides two key benefits:

- **Optimized Construction:** The library implements a variant of Thompson's NFA construction, producing NFAs that are structurally aligned with the regex and typically more compact than equivalent DFAs.
- **Native Capture Groups:** The generated NFAs include special ε-transitions (via the `State::Capture` variant) to mark capture-group boundaries. This yields a unified automaton that encodes capture-group semantics directly, avoiding manual and fragile composition of part-DFAs required in *baseline*.

The output of this stage is an NFA that correctly represents the regex but still contains ε-transitions, which are incompatible with direct circuit verification.

The Problem with ε-transitions. An ε-transition allows an NFA to move between states without consuming an input byte. While such transitions are standard for modeling alternation (|), repetition (*, +), and grouping, they pose challenges for circuit verification:

(1) **Invisible Steps:** Modeling state changes that do not consume input bytes introduces conditional logic and variable steps in circuit, significantly inflating constraint counts.
(2) **Verification of Gapped Paths:** A traversal path containing ε-transitions has "gaps" in the byte sequence. The circuit would need to prove that these gaps correspond exactly to valid ε-paths—an expensive and complex process.

Thus, all ε-transitions should be removed before circuit generation.

The ε-elimination Algorithm. The algorithm converts the initial NFA into an equivalent ε-free NFA while preserving both matching semantics and capture-group boundaries. The transformation, illustrated in Figure 8, operates under the assumption that every meaningful capture group consumes at least one byte. The algorithm proceeds as follows:

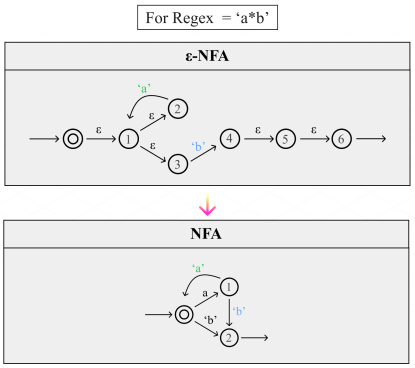


Figure 8: Epsilon elimination process.

(1) **Compute ε Closures:** For each state S, compute its ε-closure transitions (set of all states reachable from S via only ε). During this traversal, record any encountered capture-group events (start/end markers) and accept states.
(2) **Rewire Transitions:** For every state S and each state R in its ε-closure, if R has an outgoing byte transition $R \xrightarrow{\text{byte}} T$, add a direct transition $S \xrightarrow{\text{byte}} T$. This bypasses ε-only paths from S to R while maintaining equivalent behavior.
(3) **Preserve Capture-Group Semantics:** When adding a direct transition $S \xrightarrow{\text{byte}} T$:
   - *Start Captures:* Attach any start markers found along the ε-path from S to R to this new transition, indicating that the capture begins with consumption of this byte.
   - *End Captures:* Attach any end markers found in the ε-closure of T, ensuring that the capture ends when the byte leading to T is consumed.

   This two-sided transfer correctly delimits captured substrings in the resulting automaton.
(4) **Update Start and Accept States:** If the ε-closure of a state includes an accept state, that state itself becomes an accept state. Similarly, start states are updated to include all states reachable via ε-transitions from the original start state.
(5) **Prune Unreachable States:** Perform a reachability analysis from the updated start states and remove any states that can no longer be reached.

The resulting NFA is **behaviorally equivalent** to the original but entirely ε-free [29]. Each state transition now consumes a byte, and all capture-group boundaries are precisely preserved. This makes the structure suitable for direct encoding as arithmetic constraints within a zero-knowledge circuit.

5.3 Mapping NFAs to Arithmetic Circuits

With an ε-free NFA constructed, the next step is to encode its semantics into an arithmetic-circuit-compatible form. This section describes how the NFA is represented within the circuit and how the off-circuit matching logic interacts with in-circuit verification.

Flattening the NFA into a Transition Set. An ε-free NFA is a graph-based structure. To embed it within a circuit, we first *flatten* its transition rules into a structured set of tuples representing all valid state changes shown in Figure 9. The process produces three data components: 1) set of valid start states, 2) set of valid accept states, and 3 complete set of byte-level transitions, where each transition is a tuple (`source_state`, `byte_range`, `destination_state`) optionally annotated with capture-group metadata.

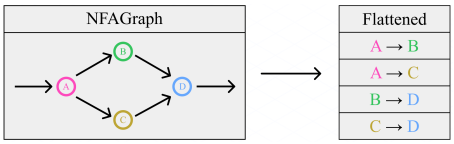


Figure 9: Flattening Process

This transition set forms an immutable ruleset embedded within the zero-knowledge circuit. The method for enforcing these rules depends on the capabilities of the target proving system — in particular, whether it natively supports operations such as associative lookups, unconstrained witness computation, and efficient range checks. When such features are available, the circuit can verify transitions with significantly fewer constraints.

Circom (R1CS). R1CS represents computations as flat collections of rank-1 quadratic equations over field elements [28, 42]; higher-level control flow, memory, and data structures (e.g., hash maps, dynamically-sized arrays) must be encoded manually into these constraints rather than being supported natively. As a result, the circuit must iterate through the entire transition set for each input byte, arithmetically checking that the claimed state transition matches at least one valid tuple. Each check instantiates comparison templates (`IsEqual`, `LessThan`) that decompose into multiple R1CS constraints, producing a cost of $O(T \times M)$ constraints where T is the number of NFA transitions and M is the match length.

Noir (UltraHonk). UltraHonk's arithmetization supports lookup gates and unconstrained witness computation, enabling a fundamentally different encoding [13]. All valid transitions are packed into a sparse array keyed by a composite (`current_state`, `byte`, `next_state`) value. At proving time, a binary search is performed in an unconstrained function — executing outside the constraint system at zero gate cost — and the result is verified with two native range-check assertions at a fixed cost of approximately 14.5 gates per lookup, independent of the number of NFA transitions. This reduces the per-byte verification cost from $O(T)$ to $O(1)$.

6 Experimental Results

6.1 Setup and Benchmark Configurations

All benchmarks were executed on a Macbook Pro M4 (14-core) with 48 GB of RAM. Table 3 in Appendix H.1 summarizes the exact hardware and software configuration.

Methodology. All measurements were collected on a single *Apple M4 Pro* (14 cores, 48 GB unified memory) running macOS (Darwin). Table 3 lists the exact toolchain. Two measurement campaigns are reported: a *regex-engine* comparison of the v1 (DFA) and v2 (NFA) back-ends (Appendix J), and a *full email-verification* comparison of the Circom/Groth16 and Noir/UltraHonk stacks (Appendix H). Each configuration is compiled once and proven multiple times; we discard one warm-up run and report the median to reduce

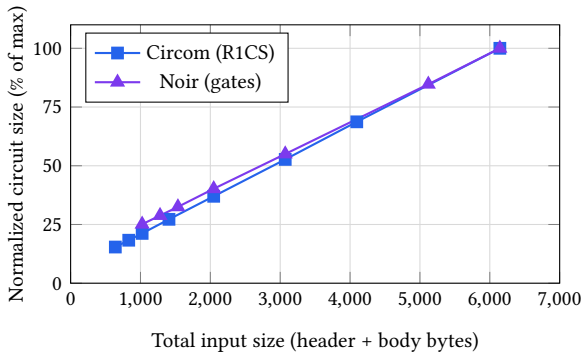


Figure 10: Circom vs. Noir: normalized circuit size.

sensitivity to system-level variance. Appendix I provides a detailed statistical analysis of benchmarking quality and consistency[6].

6.2 ZK Circuit Complexity

Constraint and Gate Scaling. We analyze prover complexity by measuring the number of constraints as the total email input size (header + body) increases. We evaluate circuit complexity scaling with input length across both proving systems. Detailed system-specific results are provided in Appendix H (Figures 21 and 22). Figure 10 shows both systems exhibit near-identical linear scaling when normalized. The slopes are similar despite different absolute units (R1CS constraints vs. UltraHonk gates). Exact values are reported in Appendix H, Table 4. Both Circom and Noir achieve $R^2 \approx 0.99$ with respect to total input size. The per-byte cost is approximately 545 constraints/B in Circom and 87 gates/B in Noir.

Feature Flag Overhead. To have a better understanding of the individual factors affecting circuit complexity, we isolate the cost of individual features by varying a single flag in Circom or substituting a single circuit template in Noir against a common baseline. In Circom, the baseline is FEAT-BASE (640h/768b, all flags disabled, 960,202 constraints). In Noir, the baseline is FEAT-BASE (512h/1024b, `verify_email`, 192,406 gates). Figures 11 and 12 report feature overhead as the percentage change relative to FEAT-BASE (960,202 constraints for Circom and 192,406 gates for Noir)[7].

We provide a detailed analysis of the impact of individual feature flags on circuit complexity in Appendix H, Figure 23 and Table 5. Overall, header and body masking introduce negligible overhead (less than 2%) in both systems. Skipping or partially computing the body hash yields the largest reduction in constraints and gates (33–44%).

RSA Key Size Impact. We evaluate the impact of the RSA public key size on circuit complexity in Appendix H, Table 7. Doubling the RSA key size from 1024 to 2048 bits adds only ∼10% gate overhead.

[6]In `Circom`, Groth16 requires a per-circuit trusted setup (zkey), which is performed once per configuration and is excluded from the reported proving-time measurements.
[7]In Circom, †FEAT-SOFT is confounded: `maxBodyLength` is set to 1408 bytes, compared to 768 bytes in all other configurations. This increases the constraint count independently of the `removeSoftLineBreaks` flag and inflates the apparent overhead. In contrast, Noir's FEAT-SOFT uses the same `maxBodyLength=1024` as the baseline, providing a clean measurement of the soft line break removal cost (+13.3%).

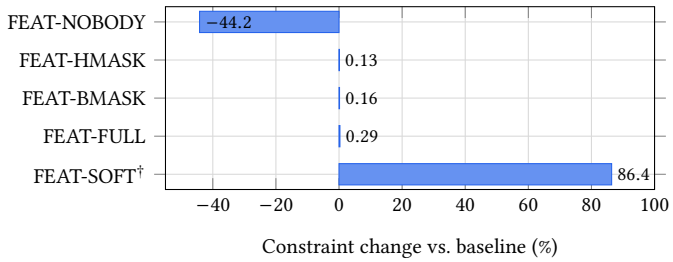


Figure 11: Circom: feature flag impact on constraints.

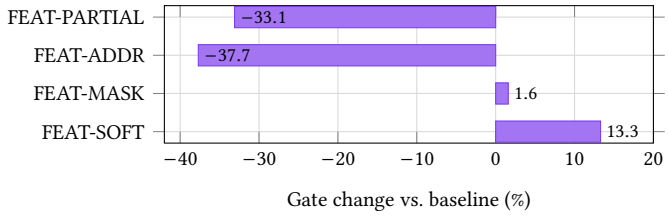


Figure 12: Noir: feature flag impact on constraints.

RSA verification is a relatively small fraction of the total circuit (SHA-256 hashing dominates).

SHA-256 Precomputation. We evaluate the impact of the SHA-256 partial precomputation technique described in Section 3.2. A complete analysis is provided in Appendix H (Figure 25, Table 8, and Figure 26). In Noir, 75% precomputation results in **33% fewer gates** and **39% faster proving**. In Circom, constraint reduction requires manually decreasing `maxBodyLength`, yielding −18% at 50% and −25% at 80%. Due to the fixed overhead of the partial-hash template in Noir, precomputation becomes net beneficial only beyond approximately 40% off-circuit hashing.

6.3 Prover Complexity

Figure 14 shows that proving time scales approximately linearly with input size in both systems. Circom/Groth16 exhibits a steeper slope and visible steps (e.g., between 3.000 and 4.000 bytes), corresponding to a ptau power-of-two boundary. For a better comparison, Figure 15 reports the prover performance of Noir relative (at the same input size) to Circom. Proving is consistently 1.8–2.3× faster in Noir, while witness generation is 3.2–6.3× faster. We provide a more detailed analysis in Appendix H.3.

To evaluate the efficiency of the proposed ZK regex construction, we compare it against the DFA-based method from [4] over several representative patterns, as summarized in Table 16. As shown in Table 16, for certain regex patterns the DFA-based approach of [4] reaches practical limits and cannot feasibly support them due to the increased complexity encountered during compilation. Figure 13 reports proving times for inputs of 64 bytes. Our Noir-based implementation achieves a 2–6× speedup over [4] under these settings. Moreover, the Noir backend consistently outperforms our Circom-based implementation by a factor of 5–15×. Additional experimental details are provided in Appendix J.5.

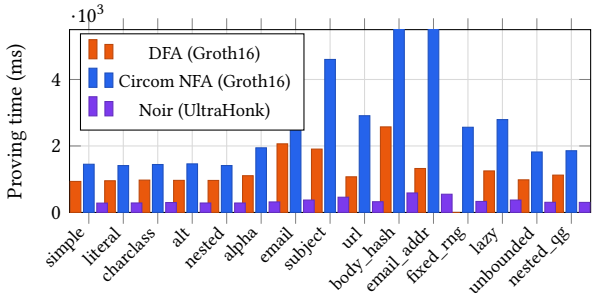
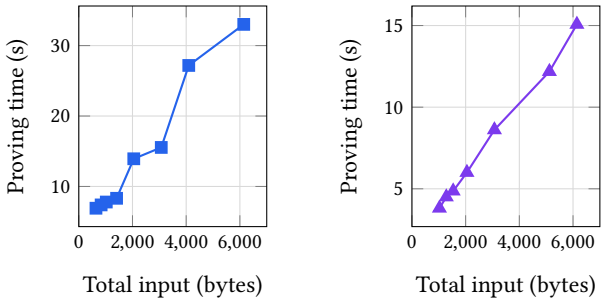


Figure 13: ZK-regex proving time at 64-byte input.



(a) Circom/Groth16. **(b) Noir/UltraHonk.**

Figure 14: Proving time vs. total input size for both systems.

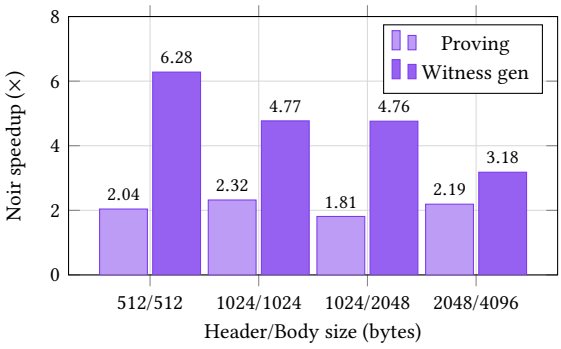


Figure 15: Noir speedup ratios over Circom.

6.4 Verification and Communication Complexity

Groth16 verification remains effectively constant at approximately ~188 ms, independent of circuit size. Although this behavior is expected, for completeness we benchmark detailed verifier performance under different configurations (see Appendix H.4, Table 9). In contrast, as shown in Figure 16, Noir/UltraHonk verification time scales linearly with circuit size, increasing from 1.7 s at 148 k gates to 5.8 s at 592 k gates. This differs from Groth16's constant $O(1)$ verification complexity.

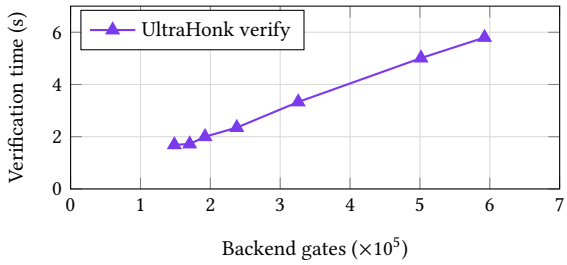


Figure 16: Noir/UltraHonk verification time.

Table 1: Groth16 vs. UltraHonk system-level tradeoffs.

Criterion	Groth16 (Circom)	UltraHonk (Noir)
Proof size	805 B	14,080 B
Verification time	188 ms (constant)	1.7 s–5.8 s (scales)
Proving time	6.9 s–33.0 s	3.8 s–15.1 s
Witness generation	1.3 s–4.8 s	250 ms–1.5 s
Compilation	8 s–68 s	1.7 s–5.4 s
Trusted setup	Required (per-circuit)	Not required
Max circuit size	~4.19 × 10^6 constraints	No hard limit
On-chain suitability	Excellent	Limited (large proofs)

Regarding proof size, as shown in Appendix H.4, Table 10, Groth16 also benefits from a constant-size proof, whereas UltraHonk produces proofs that are 17× larger. We note, however, that even in this case the proof remains relatively small at roughly ~14KB. In Appendix H.5, Table 11, we further provide detailed benchmarks of the on-chain verification costs for both ZK backends. As a rule of thumb, the gas cost to verify Groth16 proofs is byte-identical (318,559) across all seven evaluated scenarios—it depends only on the 20 public inputs and is independent of circuit size. In contrast, UltraHonk's gas cost follows a step function in the padded domain size N: each additional sumcheck round incurs ≈ 29,869 **gas**, with the total cost reaching approximately 1.8 M.

6.5 End-to-End Performance

We evaluate both systems with respect to deployment-related overhead, including compilation time and, for Groth16, the additional cost of deriving a new zkey for each circuit as part of the final trusted setup phase (see Appendix H.6).

We compare Circom and Noir at identical input sizes (header/body bytes). Four configurations share matching dimensions: 512/512, 1024/1024, 1024/2048, and 2048/4096. Figure 17 reports the total end-to-end time, including witness generation, proving, and verification, for both backends. Table 1 summarizes the benchmark results and highlights the main tradeoffs of each implementation.

7 Related Work

Verifiable Parsing and zk-Regex. Reef [11] explores zk-regex using skipping alternating finite automata (SAFA) and recursive proofs based on Nova [33]. However, its design is not directly applicable in the DKIM setting. SAFA's sublinear complexity relies on efficient random access into the committed document, whereas DKIM requires a SHA-256 commitment of the email body, which is

Table 2: Overview of state-of-the-art zk-regex approaches.

System	Automata	Complexity	Proof System	Trusted Setup	Proof Size	Features
Reef [11]	SAFA	$O(\alpha \times \log N)$	Nova (IVC)	No	Sublinear	Full PCRE
Zombie [44]	Thompson NFA	$O(\|D\| \times \|Q\|)$	Various	Varies	Varies	No lookaround
zkreg [39]	AC-DFA	$O(\|D\| + \|Q\|)$	Groth16	Yes	~660 B/proof	Fixed strings only
zk-regex (baseline) [4]	Min-DFA	$O(\|D\| \times \|Q_{\text{DFA}}\|)$	Groth16	Yes (per-circuit)	805 B	Restricted
Ours (Circom)	ε-free NFA	$O(m \times T)$	Groth16	Yes (per-circuit)	805 B	Most regex
Ours (Noir)	ε-free NFA	$O(m \times \text{lookup})$	UltraHonk	No	14,080 B	Most regex

Where: m = match length, T = transitions, α = non-skip steps, N = document + automaton size.

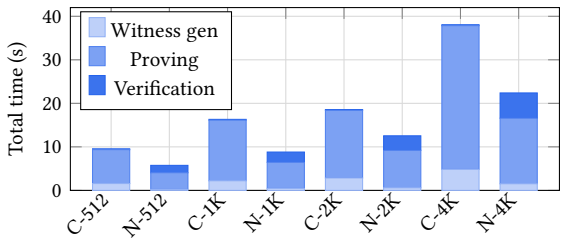


Figure 17: End-to-end performance. C = Circom, N = Noir.

a sequential Merkle–Damgård hash without random-access properties. Adapting Reef to DKIM would therefore require an additional zero-knowledge proof linking the SHA-256 commitment to a random-access-friendly commitment, introducing substantial overhead. Moreover, DKIM regexes are dominated by delimiter-safe bounded negations such as `[^;]+;` that must consume and verify each byte up to the RFC-defined tag separator `;`. In contrast, SAFA skipping is most effective for patterns with skippable regions, e.g., `a.+b`. For example, the DKIM-safe pattern `d=[^;]+;` matches exactly `d=example.com;`, whereas a relaxed pattern such as `d=.+;` may incorrectly absorb subsequent DKIM fields. While Reef supports broader PCRE-style expressiveness, its current design does not provide practical advantages for end-to-end DKIM verification compared to our Noir-based [13, 37] circuits. Table 2 summarizes state-of-the-art zk-regex systems across automata models, asymptotic complexity, proof assumptions, and supported features.

Privacy-Preserving Identity and Token-Based Proofs. The European Digital Identity Wallet (EUDIW) Architecture and Reference Framework emphasizes user control, data minimization, and interoperability in a standardized wallet-based architecture [25]. These goals have motivated research into using zero-knowledge proofs for selective disclosure and unlinkability in digital identity systems [10]. zkLogin [15] operationalizes this approach by using ZKPs to prove control over OpenID Connect (OIDC) [40] identity tokens (JWT) and derive unlinkable blockchain signing keys via Tagged Witness Signatures. While zkLogin demonstrates the feasibility of proving statements about identity provider–issued tokens, it remains inherently tied to third-party authentication providers and blockchain-specific account bootstrapping.

A recent study analyzes different vulnerabilities in zkLogin [20]. Some of these weaknesses can be mitigated by using our constructions. Most importantly, we avoid ad-hoc substring parsing and claim shadowing by enforcing strict regular-language constraints through $\mathcal{R}_{\text{regex}}$, where parsing semantics are encoded in a fixed automaton and each transition is verified in-circuit. This prevents duplicate-field ambiguity, character smuggling, and malformed-structure attacks similar to those identified in JWT parsing.

Internet Trust Infrastructure. FreeAuth [26] addresses privacy and security limitations in conventional email ownership verification, which typically relies on sending verification emails that expose full addresses and introduce interception, spam, and tracking risks. The protocol leverages existing SMTP/IMAP/POP3 authentication flows (via SASL) to prove email ownership without revealing the underlying address or requiring challenge emails. It employs TLS-oracle techniques to disclose only the authentication outcome and combines 2PC with ZKPs to produce statements about email identifiers. While FreeAuth demonstrates that ownership of an email account can be proven without direct disclosure, it does not leverage cryptographic artifacts already embedded in emails—such as DKIM signatures and does not support selective disclosures over structured message content or metadata.

NOPE [24] proposes a server authentication mechanism that reduces reliance on the traditional CA ecosystem by requiring domains to prove, via a zkSNARK, that their TLS public key is bound to a valid DNSSEC chain of trust. NOPE demonstrates how ZKPs can strengthen web PKI without disrupting existing TLS infrastructure. Similar works in this domain include [23, 36]. While NOPE strengthens domain-level authentication in the transport layer, our scheme enables privacy-preserving provenance proofs at the application layer over real-world communications.

Decentralized Storage and Provenance. Hello [32] studies abuse in IPFS-based content distribution, demonstrating practical attacks enabled by anonymous pinning and gateway caching, and advocates stronger accountability mechanisms such as stricter KYC, content moderation, and ZK–based identity tools. This line of work underscores the importance of provenance in decentralized storage systems. Our method complements this direction by providing cryptographic provenance for message-origin data itself, for example, enabling an email that seeds an IPFS object to carry a verifiable, privacy-preserving attestation of origin when stored or redistributed through IPFS.

DFA over secret shares. A related line of work studies secure DFA evaluation in the MPC setting [18], typically using interactive protocols based on oblivious transfer or homomorphic encryption and often aiming to keep the automaton private. Our setting is different: the prover locally computes a matching run and generates a NIZK proving its correctness. While MPC-based techniques could potentially be adapted, they would introduce interaction during automaton evaluation and entail different performance and trust assumptions.

Acknowledgment

The authors thank the PETS reviewing team for their valuable feedback and suggestions. The authors used generative AI-based tools to assist with text revision, including improving readability and correcting typographical, grammatical, and stylistic issues. This work was supported by the ZK Email team through funding received from the Ethereum Foundation.

References

[1] 2011. *Domainkeys identified mail (DKIM) signatures.* Technical Report.
[2] 2024. CircomLib. https://github.com/iden3/circomlib/. Accessed: 2026-02-07.
[3] 2024. regex automata library in Rust. https://docs.rs/regex-automata/latest/regex_automata/. Accessed: 2026-02-17.
[4] 2024. ZK Email Baseline. https://zk.email/blog/zkemail#user-content-fnref-3. Accessed: 2026-02-17.
[5] 2025. Regex DFA Graph Visualizer. https://zkregex.com/min_dfa. Accessed: 2025-09-10.
[6] 2026. How does DNSSEC work? https://www.cloudflare.com/learning/dns/dnssec/how-dnssec-works/. Accessed: 2026-02-20.
[7] 2026. Perl-compatible regular expressions (PCRE). https://www.pcre.org/original/doc/html/pcresyntax.html. Accessed: 12 February 2026.
[8] 2026. What is a DNS DKIM record? https://www.cloudflare.com/learning/dns/dns-records/dns-dkim-record/. Accessed: 2026-02-20.
[9] Marius A Aardal, Diego F Aranha, Katharina Boudgoust, Sebastian Kolby, and Akira Takahashi. 2024. Aggregating falcon signatures with LaBRADOR. In *Annual International Cryptology Conference*. Springer, 71–106.
[10] Iván Abellán Álvarez, Pol Hölzmer, and Johannes Sedlmeir. 2025. Privacy evaluation of the European Digital Identity Wallet's architecture and reference framework. *Computers & Security* (2025), 104707.
[11] Sebastian Angel, Eleftherios Ioannidis, Elizabeth Margolin, Srinath Setty, and Jess Woods. 2024. Reef: Fast succinct {Non-Interactive}{Zero-Knowledge} regex proofs. In *33rd USENIX Security Symposium (USENIX Security 24)*. 3801–3818.
[12] Mohammad Ishtiaq Ashiq Khan. 2026. Empirical Insights into the Operational Dynamics of DNS-Reliant Security Protocols: A Longitudinal and Qualitative Study of Email and DNS Security. *ACM SIGMETRICS Performance Evaluation Review* 53, 3 (2026), 52–55.
[13] AztecProtocol. 2025. Barretenberg: An optimized elliptic-curve library and PLONK SNARK prover. https://github.com/AztecProtocol/aztec-packages/tree/next/barretenberg.
[14] Christoph Bader, Dennis Hofheinz, Tibor Jager, Eike Kiltz, and Yong Li. 2015. Tightly-secure authenticated key exchange. In *Theory of Cryptography Conference*. Springer, 629–658.
[15] Foteini Baldimtsi, Konstantinos Kryptos Chalkias, Yan Ji, Jonas Lindstrøm, Deepak Maram, Ben Riva, Arnab Roy, Mahdi Sedaghat, and Joy Wang. 2024. zklogin: Privacy-preserving blockchain authentication with existing credentials. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3182–3196.
[16] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. 2022. Circom: A Circuit Description Language for Building Zero-knowledge Applications. *IEEE Transactions on Dependable and Secure Computing* (2022), 1–18. https://doi.org/10.1109/TDSC.2022.3232813
[17] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2007. Sponge functions. In *ECRYPT hash workshop*, Vol. 2007.
[18] Marina Blanton and Mehrdad Aliasgari. 2010. Secure outsourcing of DNA searching via finite automata. In *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 49–64.
[19] Jan Camenisch and Anna Lysyanskaya. 2002. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Annual international cryptology conference*. Springer, 61–76.
[20] Sofia Celi, Hamed Haddadi, and Kyle Den Hartog. 2026. Analysis and Vulnerabilities in zkLogin. *Cryptology ePrint Archive* (2026).
[21] Miranda Christ, Kevin Choi, and Joseph Bonneau. 2024. Cornucopia: Distributed randomness at scale. In *6th Conference on Advances in Financial Technologies (AFT 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 17–1.
[22] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. 2005. Merkle-Damgård revisited: How to construct a hash function. In *Annual International Cryptology Conference*. Springer, 430–448.
[23] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, and Bryan Parno. 2016. Cinderella: Turning shabby X. 509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 235–254.
[24] Zachary DeStefano, Jeff J Ma, Joseph Bonneau, and Michael Walfish. 2024. NOPE: Strengthening domain authentication with succinct proofs. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 673–692.
[25] European Commission. 2024. European Digital Identity Wallet Architecture and Reference Framework. Technical architecture and specifications for the European Digital Identity Wallet.
[26] Yijia Fang, Bingyu Li, Jiale Xiao, Bo Qin, Zhijintong Zhang, and Qianhong Wu. 2024. FreeAuth: Privacy-Preserving Email Ownership Authentication with Verification-Email-Free. In *2024 Annual Computer Security Applications Conference (ACSAC)*. IEEE, 336–352.
[27] Dario Fiore and Anca Nitulescu. 2016. On the (in) security of SNARKs in the presence of oracles. In *Theory of Cryptography Conference*. Springer, 108–138.
[28] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 305–326.
[29] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2001. *Introduction to Automata Theory, Languages, and Computation* (2 ed.). Addison-Wesley.
[30] IBM X-Force Red. 2023. AI vs. human deceit: Unravelling the new age of phishing tactics. Available at: https://www.ibm.com/think/x-force/ai-vs-human-deceit-unravelling-new-age-phishing-tactics/ (accessed 25 February 2026).
[31] Burt Kaliski and Jakob Jonsson. 2016. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.2. RFC 8017. https://doi.org/10.17487/RFC8017 https://www.rfc-editor.org/rfc/rfc8017.
[32] Christos Karapapas, Iakovos Pittaras, George C Polyzos, and Constantinos Patsakis. 2025. Hello, won't you tell me your name?: Investigating Anonymity Abuse in IPFS. In *International Conference on Availability, Reliability and Security*. Springer, 187–204.
[33] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. 2022. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. 359–388. https://doi.org/10.1007/978-3-031-15985-5_13
[34] Murray Kucherawy, Dave Crocker, and Tony Hansen. 2011. DomainKeys Identified Mail (DKIM) Signatures. RFC 6376. https://doi.org/10.17487/RFC6376
[35] Succinct Labs. 2026. SP1. https://github.com/succinctlabs/sp1. Accessed: Mar 1, 2026.
[36] Netlas.io. 2025. An Expert's View on DNSSEC: Pros, Cons, and When to Implement. https://netlas.io/blog/what_is_dnssec/. Accessed: 27 February 2026.
[37] Noir Development Team. 2025. Noir: A Domain-Specific Language for Zero-Knowledge Proofs. https://noir-lang.org/. Version v1.0.0-beta.8 (as of July 2025).
[38] PowerDMARC. 2026. 4 Easy Ways To Rotate DKIM Keys. https://powerdmarc.com/dkim-key-rotation-explained/. Accessed: 27 February 2026.
[39] Michael Raymond, Gillian Evers, Jan Ponti, Diya Krishnan, and Xiang Fu. 2023. Efficient zero knowledge for regular language. In *International Conference on Security and Privacy in Communication Systems*. Springer, 369–394.
[40] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. 2014. *OpenID Connect Core 1.0 incorporating errata set 1.* Technical Report. OpenID Foundation. Accessed: Oct 27, 2023.
[41] Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422.
[42] Alin Tomescu. 2025. Quadratic Arithmetic Programs (QAPs) and Rank-1 Constraint Systems (R1CS). https://alinush.github.io/r1cs. Accessed: 28 February 2026.
[43] Matthias Wächs, Martin Schanzenbach, and Christian Grothoff. 2014. A Censorship-Resistant, Privacy-Enhancing and Fully Decentralized Name System. In *Proceedings of the 13th International Conference on Cryptology and Network Security (CANS 2014)*. https://grothoff.org/christian/gns2014wachs.pdf.
[44] Collin Zhang, Zachary DeStefano, Arasu Arun, Joseph Bonneau, Paul Grubbs, and Michael Walfish. 2024. Zombie: Middleboxes that {Don't} snoop. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1917–1936.

A Email Verifier ZK Circuit Realization

Here, we provide a high-level abstraction of the circuit implementation available at: https://github.com/zkemail/zk-email-verify/blob/main/packages/circuits/email-verifier.circom

```
template EmailVerifier(...) {

// Validate header formatting and padding
assertValidHeader(emailHeader, emailHeaderLength);

// Compute SHA-256 hash of signed headers
headerHash = SHA256(emailHeader);

// Verify DKIM RSA signature over header hash
assert RSAVerify(headerHash, signature, pubkey);

```

```
// Optionally verify body hash consistency
if (checkBodyHash) {
    // Extract bh= value from DKIM-Signature header
    headerBodyHash = extractBodyHash(emailHeader);

    // Compute SHA-256 hash of email body
    computedBodyHash = SHA256(emailBody);

    // Ensure both hashes match
    assert(headerBodyHash == computedBodyHash);
}

// Optionally normalize quoted-printable soft breaks
if (removeSoftLineBreaks) {
    decodedBody = normalizeBody(emailBody);
}

// Optionally reveal only selected bytes
maskedHeader = applyMask(emailHeader, headerMask);
maskedBody   = applyMask(emailBody, bodyMask);
}
```

Listing 2: High-level structure of the DKIM email verification circuit.

B Direct DFA-to-Circuit Translation

Direct DFA-to-Circuit Translation. The combined DFA was translated into a `Circom` circuit that explicitly encoded its state transition logic. For each input character, the circuit evaluated the current DFA state, checked the symbol against all valid transitions, and activated the corresponding next state. This approach required a large number of arithmetic components (e.g., `IsEqual`, `AND`) to represent the entire transition table, as illustrated in the simplified excerpt in Listing 3.

```
// ...
template BodyHashRegex(msg_bytes) {
  signal input msg[msg_bytes];
  signal output out;
  // ...
  signal states[num_bytes+1][35]; // Represents 35 DFA states over num_bytes
  // ...
  for (var i = 0; i < num_bytes; i++) {
    // ... (input character validation) ...
    // Example of transition logic from state 0 for character 13 ('\r')
    eq[0][i] = IsEqual();
    eq[0][i].in[0] <== in[i];
    eq[0][i].in[1] <== 13;
    and[0][i] = AND();
    and[0][i].a <== states[i][0]; // Is current state 0 active?
    and[0][i].b <== eq[0][i].out; // Is current char '\r'?
    // states_tmp[i+1][1] would be set based on and[0][i].out
    // ... many similar lines for all transitions from all states ...
    states[i+1][1] <== MultiOR(2)([states_tmp[i+1][1], from_zero_enabled[i] * and[0][i].out]); // Activate next state
  }
  // ...
  // Check if any accept state is reached
  component is_accepted = MultiOR(num_bytes+1);
  for (var i = 0; i <= num_bytes; i++) {
    is_accepted.in[i] <== states[i][34]; // State 34 is an accept state
  }
  out <== is_accepted.out;
  // ...
}
```

Listing 3: Encoding the entire transition table of a DFA in baseline.

C Detailed Discussion on Realistic System Model and Trust Assumptions

DNS Public Keys. Verifying DKIM signatures requires knowledge of the domain's public signing key, which is published via the Domain Name System (DNS). These keys are subject to periodic rotation (typically every 6–24 months). In the current implementation, public keys are embedded in plaintext within the smart contract and verified by inspection. However, this model lacks cryptographic provenance.

To improve verifiability, we propose leveraging DNSSEC [6] to establish cryptographically authenticated DNS responses. Although DNSSEC adoption remains limited (notably, most major providers including Twitter do not support it), it provides a trust-minimized path to verifying key authenticity. Without DNSSEC, responses may be vulnerable to manipulation, allowing an adversary in a privileged network position to spoof the DKIM public key. To mitigate this, current contracts hard-code DNS keys, but this approach does not scale well across key rotations. Ultimately, on-chain verification of DNS records via DNSSEC or alternative trust frameworks is necessary for long-term resilience.

Censorship Resistance. To ensure censorship resistance, the system is designed such that no external party can unilaterally prevent proof generation. Notably, revocation of previously valid keys is only permitted if the corresponding signing key is disclosed, which acts as a form of cryptographic revocation proof. While this behavior is uncommon in practice, some literature has proposed deliberate key leakage as a key rotation strategy to avoid provenance retention. In such cases, older keys remain valid indefinitely, unless explicitly revoked via signing key disclosure. New keys are finalized only once all subsequent proofs use the updated key, preventing rollback attacks.

In scenarios with contested key updates, governance mechanisms may be invoked to resolve disputes, for example, through an on-chain n-of-m attestation quorum. Until DNSSEC adoption improves, the proposed approach anticipates the use of hybrid governance models to arbitrate key state transitions.

Decentralized Governance. We propose a composite governance framework for DNS key validation consisting of:

- Validator attestations from an EigenLayer-secured quorum,
- HTTPS-based TLS notary attestations from diverse endpoints,
- Enclave-based attestations from secure hardware environments,
- Threshold multisignature authorization.

While each mechanism has limitations when considered in isolation, the composite design ensures robust defense-in-depth. A successful

adversary would be required to compromise all verification layers simultaneously. Furthermore, public verifiability of these attestations enables detection of compromise, facilitating rapid migration or system forking in adversarial conditions.

Comparative Advantage. In the absence of DNSSEC, proposed approach provides a stronger security model than traditional attestations by amortizing trust: a one-time attestation to a public DNS key enables the generation of verifiable ZKPs for an unbounded number of private email statements. This reduces the frequency and scope of consensus operations and allows for scalable and censorship-resistant provenance guarantees.

Self-Hosted DKIM Registries. For scenarios where reliance on the DNS hierarchy is undesirable, proposed scheme supports alternative key publication mechanisms. Specifically, domains may publish DKIM registries signed by an application-specific ECDSA keypair. This enables fully self-sovereign key management, suitable for protocol communities, organizations, or individual users.

Users may also rotate their email identity keys in a privacy-preserving manner. This is accomplished by sending a private email to a secondary address they control, attesting ownership of the original address. Upon DNS key rotation, a follow-up SNARK can prove prior access and authorize migration to the new key. Due to DKIM limitations, self-signed emails are not viable, so a secondary email address is required for this flow.

C.1 Mail Servers

Sending Mail Server. By design, the DKIM signing key is held by the domain's outbound mail server, which enables it to sign any message purporting to originate from a user under that domain. This is consistent with the existing trust model of email infrastructure: users implicitly trust their email provider not to forge messages on their behalf. Thus, proposed scheme does not introduce new trust assumptions in this regard.

A more practical risk is that changes to the formatting of outgoing emails may invalidate previously functional regular expressions used for matching and proof construction. This necessitates periodic updates to the zk-circuit's regex configuration and associated proving keys.

Receiving Mail Server. The receiving mail server has visibility into both the content and metadata of incoming emails. Consequently, any party with access to the receiving server may generate valid proofs for messages it observes. This limits the system's ability to guarantee proof exclusivity for a recipient unless end-to-end encryption is employed. While the S/MIME standard could mitigate this by encrypting headers and payloads, adoption is currently limited.

This trust limitation is again congruent with existing email privacy expectations, where users trust providers (e.g., Gmail) not to misuse access to verification codes or sensitive messages.

C.2 Anonymity Considerations

In applications using our constructions for anonymous credentials or pseudonymous identity proofs, nullifier-based constructions may enable linkability. For example, if a user posts an on-chain nullifier derived from the email body, the receiving mail server—having access to plaintext—may associate the nullifier with a specific user identity. Thus, anonymity and uniqueness guarantees are in tension: nullifiers help prevent Sybil attacks but may allow deanonymization by privileged servers.

To mitigate this, users are encouraged to employ encrypted email providers (e.g., Skiff, ProtonMail). However, providers such as ProtonMail modify MIME boundaries with nondeterministic identifiers, rendering message bodies unverifiable. In such cases, only header-based proofs are feasible.

For enhanced privacy, we support workflows where structured data (e.g., authorization codes) is embedded in encrypted email subject lines. If delivery is delayed sufficiently, this technique prevents correlation between content and recipient behavior.

$\mathsf{Game}^{\text{EUF-CMA}}_{\mathcal{A}}(1^\lambda)$:

1 : $(sk, pk) \xleftarrow{\$} \text{DKIM-RSA.KeyGen}(1^\lambda)$;
2 : $\mathsf{S} \leftarrow \emptyset$;
3 : $(m', \sigma') \xleftarrow{\$} \mathcal{A}^{O_{\text{Sign}}(\cdot)}(pk)$;
4 : **return** $\text{DKIM-RSA.Vfy}(pk, m', \sigma') = 1 \wedge m' \notin \mathsf{S}$;

$O_{\text{Sign}}(m)$:

1 : $\sigma \xleftarrow{\$} \text{DKIM-RSA.Sign}(sk, m)$;
2 : $\mathsf{S} \leftarrow \mathsf{S} \cup \{m\}$;
3 : **return** σ;

Figure 18: EUF-CMA security of the DKIM-RSA signature scheme.

D Properties of the DKIM-RSA Signature Scheme

Correctness. We say that the DKIM-RSA scheme is correct if the DKIM-RSA.Vfy algorithm returns $b = 1$ for the signature that is honestly generated for a message m with a signing key sk. Formally, it holds that

$$\Pr[\text{DKIM-RSA.Vfy}(pk, m, \sigma) = 1] = 1,$$

where

$$(sk, pk) \xleftarrow{\$} \text{DKIM-RSA.KeyGen}(1^\lambda),$$
$$\sigma \xleftarrow{\$} \text{DKIM-RSA.Sign}(sk, m).$$

EUF-CMA Security. For any security parameter $\lambda \in \mathbb{N}$ and any PPT adversary $\mathcal{A}$, let $\mathsf{Game}^{\text{EUF-CMA}}_{\mathcal{A}}(1^\lambda)$ be the game defined in Figure 18, where $\mathcal{A}$ can make at most Q queries to the oracle O_{Sign}. The advantage of $\mathcal{A}$ is defined as

$$\text{Adv}^{\text{EUF-CMA}}_{\text{DKIM-RSA},\mathcal{A}}(1^\lambda) = \Pr\left[\mathsf{Game}^{\text{EUF-CMA}}_{\mathcal{A}}(1^\lambda) = 1\right].$$

We say that DKIM-RSA is existentially unforgeable against chosen-message attacks (EUF-CMA) if, for every security parameter $\lambda \in \mathbb{N}$ and every PPT adversary $\mathcal{A}$, the advantage $\text{Adv}^{\text{EUF-CMA}}_{\text{DKIM-RSA},\mathcal{A}}(1^\lambda)$ is negligible in λ.

MU-EUF-CMA Security. We further introduce a multi-user extension of EUF-CMA security, capturing a setting in which the adversary may observe N public keys and signatures under the

$\mathsf{Game}^{\text{MU-EUF-CMA}}_{\mathcal{A}}(1^\lambda, N)$:

$\forall i \in [N], \quad (sk_i, pk_i) \xleftarrow{\$} \text{DKIM-RSA.KeyGen}(1^\lambda);$
$\mathsf{S}_1, \ldots, \mathsf{S}_N \leftarrow \emptyset;$
$(i', m', \sigma') \xleftarrow{\$} \mathcal{A}^{O_{\mathsf{Sign}}(\cdot,\cdot)}(pk_1, \ldots, pk_N);$
return $\text{DKIM-RSA.Vfy}(pk_{i'}, m', \sigma') = 1 \wedge m' \notin \mathsf{S}_{i'};$

$O_{\mathsf{Sign}}(i, m)$:

$\sigma \xleftarrow{\$} \text{DKIM-RSA.Sign}(sk_i, m);$
$\mathsf{S}_i \leftarrow \mathsf{S}_i \cup \{m\};$
return σ;

Figure 19: MU-EUF-CMA security of the DKIM-RSA signature scheme.

corresponding signing keys [14]. However, since we assume that mail servers cannot be corrupted by the adversary, the following definition does not give the adversary access to any signing key.

Let N be the number of users. For any security parameter $\lambda \in \mathbb{N}$ and any PPT adversary $\mathcal{A}$, let $\mathsf{Game}^{\text{MU-EUF-CMA}}_{\mathcal{A}}(1^\lambda, N)$ be the game defined in Figure 19, where $\mathcal{A}$ can make at most Q queries to the oracle O_{Sign}. The advantage of $\mathcal{A}$ is defined as

$$\mathsf{Adv}^{\text{MU-EUF-CMA}}_{\text{DKIM-RSA},\mathcal{A}}(1^\lambda, N) = \Pr\left[\mathsf{Game}^{\text{MU-EUF-CMA}}_{\mathcal{A}}(1^\lambda, N) = 1\right].$$

We say that DKIM-RSA is existentially unforgeable against chosen-message attacks in a multi-user setting (MU-EUF-CMA) if, for every security parameter $\lambda \in \mathbb{N}$, every PPT adversary $\mathcal{A}$, every natural number $N = \text{poly}(\lambda)$, the advantage $\mathsf{Adv}^{\text{MU-EUF-CMA}}_{\text{DKIM-RSA},\mathcal{A}}(1^\lambda, N)$ is negligible in λ.

E Properties of the NIZK Argument Systems

Definition E.1 (Perfect Completeness [28])**.** An honest prover with a valid witness should convince an honest verifier. Formally, for all $\lambda \in \mathbb{N}$, $R \in \mathcal{R}_\lambda$, $(\phi, w) \in R$, it holds that

$$\Pr\left[\text{Vfy}(R, \text{crs}, \phi, \pi) = 1 \;\middle|\; \begin{array}{c}(\text{crs}, \text{td}) \xleftarrow{\$} \text{Setup}(R) \\ \pi \xleftarrow{\$} \text{Prv}(R, \text{crs}, \phi, w)\end{array}\right] = 1.$$

Definition E.2 (Perfect Zero-knowledge [28])**.** The argument does not leak anything beyond the truth of statement. We say that the NIZK argument system is perfect zero-knowledge if for all security parameters $\lambda \in \mathbb{N}$, $(R, z) \xleftarrow{\$} \mathcal{R}(1^\lambda)$, $(\phi, w) \in R$, and all adversaries $\mathcal{A}$, it holds that:

$$\Pr\left[\begin{array}{c}\mathcal{A}(R, z, \text{crs}, \\ \text{td}, \pi) = 1\end{array} \;\middle|\; \begin{array}{c}(\text{crs}, \text{td}) \xleftarrow{\$} \text{Setup}(R) \\ \pi \xleftarrow{\$} \text{Prv}(R, \text{crs}, \phi, w)\end{array}\right] = \Pr\left[\begin{array}{c}\mathcal{A}(R, z, \text{crs}, \\ \text{td}, \pi) = 1\end{array} \;\middle|\; \begin{array}{c}(\text{crs}, \text{td}) \xleftarrow{\$} \text{Setup}(R) \\ \pi \xleftarrow{\$} \text{Sim}(R, \text{td}, \phi)\end{array}\right]$$

Definition E.3 (Knowledge Soundness for $\mathbb{O}$ [27])**.** Let $\mathbb{O}$ be a family of oracles. We write $O \xleftarrow{\$} \mathbb{O}$ to denote sampling an oracle O from $\mathbb{O}$. We say that the NIZK argument system is an argument of knowledge with respect to an oracle family $\mathbb{O}$ if for all non-uniform polynomial-time adversaries $\mathcal{A}$ there exists a non-uniform polynomial-time extractor $\mathcal{X}_{\mathcal{A}}$ such that

$$\Pr\left[\begin{array}{c}\text{Vfy}(R, \text{crs}, \phi, \pi) = 1, \\ (\phi, w) \notin R\end{array} \;\middle|\; \begin{array}{c}(R, z) \xleftarrow{\$} \mathcal{R}(1^\lambda) \\ O \xleftarrow{\$} \mathbb{O} \\ (\text{crs}, \text{td}) \xleftarrow{\$} \text{Setup}(R) \\ (\phi, \pi) \xleftarrow{\$} \mathcal{A}^O(R, z, \text{crs}) \\ w \xleftarrow{\$} \mathcal{X}_{\mathcal{A}}(R, z, \text{crs}, \text{qt})\end{array}\right] \leq \text{negl}(\lambda),$$

where the extractor $\mathcal{X}_{\mathcal{A}}$ has full access to the internal state of the adversary $\mathcal{A}$, including its random coins, and is given the oracle transcript $\text{qt} = \{q_i, a_i\}$, consisting of each oracle query q_i made by $\mathcal{A}$ and the corresponding response $a_i = O(q_i)$.

E.1 Oracle-Aided Knowledge Soundness for Our Concrete Instantiations

For the concrete instantiations considered in this work, namely the constructions based on Groth16 and UltraHonk for the relation R_{Email}, we conjecture that these NIZK argument systems remain knowledge-sound even when the adversary is given access to the DKIM-RSA signing oracle O_{DKIM}. In other words, we expect that the extraction guarantees of Groth16 and UltraHonk continue to hold in the presence of the DKIM-RSA signing oracle used in our construction.

In the standard model, Fiore and Nitulescu [27] show a negative result: knowledge soundness does not generally extend to adversaries interacting with arbitrary, adversarially designed signing oracles. However, their result does not rule out extraction for natural, concrete signing oracles such as the DKIM-RSA signing oracle considered here. Our use of the oracle-aided knowledge-soundness definition follows this perspective: we do not claim that every NIZK argument system is secure with every signing oracle, but only assume the required oracle-aided knowledge soundness for the concrete DKIM-RSA signing oracle used in our construction.

This type of conjecture is not new; closely related conjectures have already been adopted in prior works that combine NIZK argument systems with signature verification. A closely related precedent appears in Cinderella [23], which uses verifiable computation to prove statements about PKCS#1-based X.509 certificates and explicitly relies on a knowledge-soundness assumption in the presence of signing oracles. A similar oracle-aided extraction issue also arises in zkLogin [15], which uses Groth16 to prove statements about signed JWTs. Although zkLogin formulates its requirement as classical knowledge soundness, its security argument likewise relies on knowledge extraction in a setting where the adversary can obtain valid signatures from an external issuer.

Importantly, these systems and our construction all involve RSA signatures encoded according to the PKCS#1 standard and applied to hashed messages. Specifically, each considers an NP relation that computes a SHA-256 hash of the message and then verifies the given RSA signature on that hash. Therefore, our conjecture should be viewed as a natural continuation of assumptions already used in these prior works, rather than as a fundamentally new conjecture.

Additional supporting evidence comes from the positive results of Fiore and Nitulescu [27]. In the random-oracle model, they show

that if signatures are generated by applying a hash-then-sign transformation to a random salt and the message, then classical knowledge soundness can be lifted to knowledge soundness in the presence of the corresponding signing oracle. LaBRADOR [9] further develops this line of work in a related deterministic hash-then-sign setting without a signing salt, which is closer to the PKCS#1-based signatures appearing in Cinderella, zkLogin, and our DKIM-RSA instantiation.

However, these analyses do not resolve the issue that arises when the hash computation used in signature verification is performed inside the relation proved by the NIZK argument system. While the signature verification algorithm obtains the hash value by querying the random oracle, the NP relation for that algorithm replaces this oracle query with an explicit circuit evaluation of a concrete hash function. To apply their ROM-based security argument to such concrete circuit implementations, one must therefore make the heuristic conjecture, explicitly noted in [27], that this in-circuit hash evaluation continues to play the role of the random oracle.

This is a general modeling gap for NIZK applications of ROM-based hash-then-sign signatures, not a peculiarity of our construction. We leave a fully formal treatment of this heuristic in an appropriate idealized model, such as an arithmetized random-oracle model, as an open problem.

F Full Security Proofs

In this section, we provide full proofs of Theorems 1, 2, and 3.

Unforgeability. We begin by proving Theorem 1.

Proof. We show that if there exists an adversary $\mathcal{A}$ that wins $\mathsf{GameUnforge}^{\mathsf{Email}}_{\mathcal{A}}(1^\lambda, \mathsf{rgx}_{\mathsf{ptrn}})$ in Figure 5 with non-negligible probability, then there exists an adversary $\mathcal{B}$ that, by internally invoking $\mathcal{A}$, breaks the EUF-CMA security in Definition 2.5 with non-negligible probability. $\mathcal{B}$ acts as the challenger for $\mathcal{A}$ as follows.

(1) $\mathcal{B}$ receives a public key pk_{srvr} from the challenger in $\mathsf{Game}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A}}(1^\lambda)$.
(2) $\mathcal{B}$ runs
$$(R_{\mathsf{Email}}, z) \xleftarrow{\$} \mathcal{R}_{\mathsf{Email}}(1^\lambda),$$
$$(\mathsf{crs}, \mathsf{td}) \xleftarrow{\$} \mathsf{NIZK.Setup}(R_{\mathsf{Email}}).$$
(3) For a query "pk" to the oracle O_{DKIM}, $\mathcal{B}$ returns pk_{srvr}.
(4) $\mathcal{B}$ provides $\mathcal{A}$ with $\mathsf{rgx}_{\mathsf{ptrn}}$, R_{Email}, z, crs, and pk_{srvr}.
(5) For each query m_i to the oracle O_{DKIM}, $\mathcal{B}$ forwards it to the signing oracle O_{Sign} in $\mathsf{Game}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A}}(1^\lambda)$, returning its output σ to $\mathcal{A}$.
(6) $\mathcal{A}$ returns π^* and m^* to $\mathcal{B}$.
(7) Because $\mathsf{NIZK.Vfy}(R_{\mathsf{Email}}, \mathsf{crs}, \phi, \pi^*) = 1$ holds with non-negligible probability, the knowledge-soundness of the NIZK argument system (Definition E.3) guarantees that there exists an extractor $\mathcal{X}_{\mathcal{A}}$ that on input R_{Email}, z, crs, and the oracle transcript qt, outputs a witness w such that $(\phi, w) \in R_{\mathsf{Email}}$ holds.
(8) $\mathcal{B}$ parses w as (σ', m') and outputs (m', σ').

By the definition of R_{Email}, $(\phi, w) \in R_{\mathsf{Email}}$ implies that
$$\mathsf{DKIM\text{-}RSA.Vfy}(pk_{\mathsf{srvr}}, m', \sigma') = 1$$
holds. We can also confirm that $m' \notin \mathsf{S}$; otherwise, there exists $(m_i, \cdot) \in \mathsf{qt}$ such that $m^* = \mathsf{match}(m_i, \mathsf{rgx}_{\mathsf{ptrn}})$. Therefore, $\mathcal{B}$ can break EUF-CMA security by outputting (m', σ'). Hence, by contradiction, the theorem is proved. □

Anonymity. We provide a full proof of Theorem 2.

Proof. We prove the theorem via a sequence of game hops, defined as follows:

$\underline{\mathsf{Game}_0}$: This is identical to $\mathsf{GameAnon}^{\mathsf{Email}}_{\mathcal{A}}(1^\lambda, \mathsf{rgx}_{\mathsf{ptrn}})$ in Figure 6.

$\underline{\mathsf{Game}_1}$: This is the same as Game_0 except that the proof π is replaced with the simulated one:
$$\pi \xleftarrow{\$} \mathsf{NIZK.Sim}(R_{\mathsf{Email}}, \mathsf{td}, \phi).$$

The perfect zero-knowledge property of the NIZK argument system immediately implies that Game_0 and Game_1 are identically distributed. Recall that the statement ϕ is parsed as $\left(pk_{\mathsf{srvr}}, m^*, \mathsf{rgx}_{\mathsf{ptrn}}\right)$; therefore, π in Game_1 no longer depends on the choice of $b \in \{0, 1\}$. This concludes the proof. □

Linkability. We provide a full proof of Theorem 3.

Proof. We show that if there exists an adversary $\mathcal{A}$ that wins GameLink with non-negligible probability, then there exists an adversary $\mathcal{B}$ that wins GameUnforge with non-negligible probability by internally invoking $\mathcal{A}$.

$\mathcal{B}$ first forwards $(\mathsf{rgx}_{\mathsf{ptrn}}, R_{\mathsf{Email}}, z, \mathsf{crs}, pk_{\mathsf{srvr}})$ that it receives from its challenger to $\mathcal{A}$. After $\mathcal{A}$ returns m, $\mathcal{B}$ submits it to the signing oracle O_{DKIM}, forwarding its output σ to $\mathcal{A}$. $\mathcal{A}$ finally returns $(\pi^*_0, \pi^*_1, m^*_0, m^*_1)$.

Let $m^*_\perp \leftarrow \mathsf{match}(m, \mathsf{rgx}_{\mathsf{ptrn}})$, and let $b \in \{0, 1\}$ be such that $m^*_b \neq m^*_\perp$, which exists because $m^*_0 \neq m^*_1$. The adversary $\mathcal{B}$ outputs (π^*_b, m^*_b). Since $\mathsf{qt} = \{m, O_{\mathsf{DKIM}}(m)\}$, such an output constitutes a valid forgery in GameUnforge. Therefore, $\mathcal{B}$ wins GameUnforge with non-negligible probability. This proves the theorem by contradiction. □

G Multiple Mail Servers

A more user-friendly variant of the NP relation R_{Email} avoids exposing the public key of the mail server pk_{srvr} directly in the public statement. Instead, the verifier can publish a cryptographic accumulator [19] over the set of eligible mail server public keys, and the prover demonstrates that the signer public key pk_{srvr} belongs to this set. This enables users to authenticate using different trusted mail providers, e.g., `google`, `apple`, or `outlook`, while preserving a uniform verification interface.

The accumulator can be instantiated with a Merkle tree or any other suitable construction; its concrete realization is orthogonal to our protocol. We adopt the formal definition of accumulators given in [19, 21].

Definition G.1 (Cryptographic Accumulator [19, 21])**.** We define a (static) accumulator scheme ACC as a tuple of the following polynomial-time algorithms:

- $pp \xleftarrow{\$} \mathsf{ACC.Setup}(1^\lambda)$: It takes as input a security parameter $\lambda \in \mathbb{N}$ and outputs public parameters pp.
- $v \leftarrow \mathsf{ACC.Accumulate}(pp, S)$: It takes as input public parameters pp and a set S. It outputs an accumulator value v.
- $w_x \leftarrow \mathsf{ACC.Wit}(pp, S, v, x)$: It takes as input public parameters pp, a set S, an accumulator value v for S, and an element $x \in S$. It outputs a membership witness w_x.
- $b \leftarrow \mathsf{ACC.Vfy}(pp, v, x, w)$: It takes as input public parameters pp, an accumulator value v, an element x, and a membership witness w_x. It outputs a bit $b \in \{0, 1\}$.

The correctness of the ACC scheme guarantees that for all sets S and all $x \in S$, we have

$$\Pr\left[\mathsf{ACC.Vfy}(pp, v, x, w_x) = 1 \;\middle|\; \begin{array}{c} pp \xleftarrow{\$} \mathsf{ACC.Setup}(1^\lambda) \\ v \leftarrow \mathsf{ACC.Accumulate}(pp, S) \\ w_x \leftarrow \mathsf{ACC.Wit}(pp, S, v, x) \end{array}\right] = 1.$$

The security property is that it is computationally infeasible to construct a membership witness for an element $x \notin S$ and an accumulator value v for S. Formally, we say that the ACC scheme is secure if for every PPT adversary $\mathcal{A}$, it holds that

$$\Pr\left[\begin{array}{c} \mathsf{ACC.Vfy}(pp, v, x, w_x) = 1 \\ \wedge\ x \notin S \end{array}\;\middle|\; \begin{array}{c} pp \xleftarrow{\$} \mathsf{ACC.Setup}(1^\lambda) \\ (S, x, w_x) \xleftarrow{\$} \mathcal{A}(pp) \\ v \leftarrow \mathsf{ACC.Accumulate}(pp, S) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

We extend the original relation R_{Email} (Equation 1) as follows, so that the NP relation verifies that the public key pk_{srvr} is contained in the accumulator v.

$$\begin{aligned} &R_{\mathsf{MultiEmail}} := \begin{pmatrix} \phi{=}(pp, v_{pk}, m^*, \mathsf{rgx}_{\mathsf{ptrn}}), \\ w{=}(\sigma, m, pk_{\mathsf{srvr}}, w_{pk}) \end{pmatrix}, \\ &(\phi, w) \in R_{\mathsf{MultiEmail}} \Leftrightarrow \\ &\mathsf{DKIM\text{-}RSA.Vfy}(pk_{\mathsf{srvr}}, m, \sigma) = 1 \\ &\wedge\ m^* = \mathsf{match}(m, \mathsf{rgx}_{\mathsf{ptrn}}) \\ &\wedge\ \mathsf{ACC.Vfy}(pp, v_{pk}, pk_{\mathsf{srvr}}, w_{pk}) = 1, \end{aligned}$$

where v_{pk} denotes the accumulation over the set of all valid public keys of the mail servers.

We can accordingly extend the security definition of unforgeability to a multi-server setting, where the NIZK proof is verified with respect to the NP relation $R_{\mathsf{MultiEmail}}$. The adversary can obtain N public keys and signatures under the corresponding signing keys; however, since we assume that mail servers cannot be corrupted by the adversary, the adversary cannot access any signing key.

To define this extended notion of unforgeability, we also extend the signing oracle O_{DKIM} as follows.

Definition G.2 (Multi-Server Signing Oracle)**.** Let $\mathbb{O}_{\mathsf{Multi\text{-}DKIM}}$ denote the family of multi-server signing oracles with respect to a security parameter $\lambda \in \mathbb{N}$, the number of servers $N \in \mathbb{N}$, and a DKIM-RSA signature scheme (Definition 2.5). Each oracle $O_{\mathsf{Multi\text{-}DKIM}} \xleftarrow{\$} \mathbb{O}_{\mathsf{Multi\text{-}DKIM}}$ works as follows.

(1) Upon initialization, for every $i \in [N]$, it runs $(sk_i, pk_i) \xleftarrow{\$} \mathsf{DKIM\text{-}RSA.KeyGen}(1^\lambda)$ and stores the resulting key pairs.
(2) On the special input "pk", it returns the stored public keys $\{pk_i\}_{i\in[N]}$.
(3) On each message query (i, m), the oracle returns a signature $\sigma \xleftarrow{\$} \mathsf{DKIM\text{-}RSA.Sign}(sk_i, m)$.

Theorem 4. *Assuming that the DKIM-RSA scheme satisfies MU-EUF-CMA security (Definition 2.5), the NIZK argument system is knowledge-sound (Definition E.3) with respect to the multi-server signing oracle family $\mathbb{O}_{\mathsf{Multi\text{-}DKIM}}$ in Definition G.2, and the ACC scheme is secure (Definition G.1), our construction is unforgeable in a multi-server setting. Formally, for all security parameters $\lambda \in \mathbb{N}$, non-uniform polynomial-time adversaries $\mathcal{A}$, regex patterns $\mathsf{rgx}_{\mathsf{ptrn}}$, and the number of servers $N = poly(\lambda)$, it holds that:*

$$\Pr[\mathsf{GameMultiUnforge}^{\mathsf{Email}}_{\mathcal{A}}(1^\lambda, \mathsf{rgx}_{\mathsf{ptrn}}, N) = 1] \leq \mathsf{negl}(\lambda),$$

where $\mathsf{GameMultiUnforge}^{\mathsf{Email}}_{\mathcal{A}}(1^\lambda, \mathsf{rgx}_{\mathsf{ptrn}}, N)$ is defined in Figure 20, and $\mathcal{A}$ can make at most Q queries to the oracle $O_{\mathsf{Multi\text{-}DKIM}}$.

Proof. Recall that by the knowledge-soundness of the NIZK argument system (Definition E.3), if the adversary $\mathcal{A}$ outputs a proof π^* such that $\mathsf{NIZK.Vfy}(R_{\mathsf{MultiEmail}}, \mathsf{crs}, \phi, \pi^*) = 1$ holds with non-negligible probability, there exists an extractor $\mathcal{X}_{\mathcal{A}}$ that on input $R_{\mathsf{MultiEmail}}$, z, and crs, outputs a witness w such that $(\phi, w) \in R_{\mathsf{MultiEmail}}$ holds. We parse w as $(\sigma', m', pk_{\mathsf{srvr}}, w_{pk})$. We first consider the case in which $pk_{\mathsf{srvr}} = pk_{i'}$ for some $i' \in [N]$, and then the case in which no such i' exists.

In the first case, as in the proof of Theorem 1, we can show that if there exists an adversary $\mathcal{A}$ that wins $\mathsf{GameMultiUnforge}^{\mathsf{Email}}_{\mathcal{A}}$ in Figure 20 with non-negligible probability, then there exists an adversary $\mathcal{B}$ that, by internally invoking $\mathcal{A}$, breaks the MU-EUF-CMA security in Definition 2.5 with non-negligible probability. The proof differs from the previous one only in the following respects. First, $\mathcal{B}$ computes v_{pk} from the public keys $\{pk_i\}_{i\in[N]}$ provided by the challenger, and provides v_{pk} to $\mathcal{A}$ as an additional input. After the extractor $\mathcal{X}_{\mathcal{A}}$ outputs a witness $w = (\sigma', m', pk_{\mathsf{srvr}}, w_{pk})$, $\mathcal{B}$ finds an index i' such that $pk_{\mathsf{srvr}} = pk_{i'}$. Finally, $\mathcal{B}$ outputs (i', m', σ').

By the definition of $R_{\mathsf{MultiEmail}}$, we have

$$\mathsf{DKIM\text{-}RSA.Vfy}(pk_{i'}, m', \sigma') = 1.$$

Moreover, for every queried message m, we have

$$m^* \neq \mathsf{match}(m, \mathsf{rgx}_{\mathsf{ptrn}})$$

so that $m' \notin S_{i'}$ holds. Therefore, this output of $\mathcal{B}$ implies that $\mathcal{B}$ breaks the MU-EUF-CMA security with non-negligible probability. It follows by contradiction that, in the first case, $\mathcal{A}$ wins $\mathsf{GameMultiUnforge}^{\mathsf{Email}}_{\mathcal{A}}$ only with negligible probability.

We next consider the second case, where $pk_{\mathsf{srvr}} \neq pk_i$ for all $i \in [N]$. We show that there exists an adversary C that, by internally invoking $\mathcal{A}$, breaks the security of the ACC scheme in Definition G.1. C acts as the challenger for $\mathcal{A}$ as follows.

(1) C receives public parameters $pp \xleftarrow{\$} \mathsf{ACC.Setup}(1^\lambda)$ from the challenger of the ACC scheme security.

$\mathsf{GameMultiUnforge}^{\mathsf{Email}}_{\mathcal{A}}(1^\lambda, \mathrm{rgx}_{\mathrm{ptrn}}, N)$:

$(R_{\mathsf{MultiEmail}}, z) \xleftarrow{\$} \mathcal{R}_{\mathsf{MultiEmail}}(1^\lambda)$, $O_{\text{Multi-DKIM}} \xleftarrow{\$} \mathbb{O}_{\text{Multi-DKIM}}$;
$(\mathrm{crs}, \mathrm{td}) \xleftarrow{\$} \mathsf{NIZK.Setup}(R_{\mathsf{MultiEmail}})$;
$\{pk_i\}_{i\in[N]} \leftarrow O_{\text{Multi-DKIM}}(\text{"pk"})$;
$pp \xleftarrow{\$} \mathsf{ACC.Setup}(1^\lambda)$;
$v_{pk} \leftarrow \mathsf{ACC.Accumulate}(pp, \{pk_i\}_{i\in[N]})$;
(π^*, m^*)
$\xleftarrow{\$} \mathcal{A}^{O_{\text{Multi-DKIM}}(\cdot)}(\mathrm{rgx}_{\mathrm{ptrn}}, R_{\mathsf{MultiEmail}}, z, \mathrm{crs}, \{pk_i\}_{i\in[N]}, pp, v_{pk})$;
$\phi \leftarrow \left(pp, v_{pk}, m^*, \mathrm{rgx}_{\mathrm{ptrn}}\right)$;
return $\mathsf{NIZK.Vfy}(R_{\mathsf{MultiEmail}}, \mathrm{crs}, \phi, \pi^*) = 1 \wedge$
$\forall((i, m), \cdot) \in \mathrm{qt} : m^* \neq \mathsf{match}(m, \mathrm{rgx}_{\mathrm{ptrn}})$;

Figure 20: Game-based definition of unforgeability in a multi-server setting for our construction, where qt $= \{(i, m_{i,j}), O_{\text{Multi-DKIM}}((i, m_{i,j}))\}$ denotes the oracle transcript as defined in Definition E.3.

(2) C runs

$$(R_{\mathsf{MultiEmail}}, z) \xleftarrow{\$} \mathcal{R}_{\mathsf{MultiEmail}}(1^\lambda),$$
$$(\mathrm{crs}, \mathrm{td}) \xleftarrow{\$} \mathsf{NIZK.Setup}(R_{\mathsf{MultiEmail}}),$$
$$\forall i \in [N], \quad (sk_i, pk_i) \xleftarrow{\$} \text{DKIM-RSA.KeyGen}(1^\lambda),$$
$$v_{pk} \leftarrow \mathsf{ACC.Accumulate}(pp, \{pk_i\}_{i\in[N]}).$$

(3) For the special query "pk" to the oracle $O_{\text{Multi-DKIM}}$, C returns $\{pk_i\}_{i\in[N]}$.
(4) C provides $\mathcal{A}$ with $\mathrm{rgx}_{\mathrm{ptrn}}$,$R_{\mathsf{MultiEmail}}$,$z$,crs,$\{pk_i\}_{i\in[N]}$,$pp$,$v_{pk}$.
(5) For each query (i, m) to the oracle $O_{\text{Multi-DKIM}}$, C simply returns a signature $\sigma \leftarrow \text{DKIM-RSA.Sign}(sk_i, m)$, which is feasible because C has the signing key sk_i.
(6) $\mathcal{A}$ returns π^* and m^* to C.
(7) The extractor $\mathcal{X}_{\mathcal{A}}$ outputs a witness $w = (\sigma', m', pk_{\mathrm{srvr}}, w_{pk})$. C outputs $(\{pk_i\}_{i\in[N]}, pk_{\mathrm{srvr}}, w_{pk})$.

By the definition of $R_{\mathsf{MultiEmail}}$, we have

$$\mathsf{ACC.Vfy}(pp, v_{pk}, pk_{\mathrm{srvr}}, w_{pk}) = 1.$$

However, $pk_{\mathrm{srvr}} \notin \{pk_i\}_{i\in[N]}$ in the second case. Therefore, this output of C implies that C breaks the security of the ACC scheme with non-negligible probability. It follows by contradiction that, in the second case, $\mathcal{A}$ wins $\mathsf{GameMultiUnforge}^{\mathsf{Email}}_{\mathcal{A}}$ only with negligible probability.

Since the first and second cases are disjoint and exhaustive, the overall winning probability of $\mathcal{A}$ is bounded by the sum of its winning probabilities in the two cases. As both terms are negligible, $\mathcal{A}$ wins only with negligible probability. □

H Detailed Analysis

H.1 System Configuration

Table 3: Client-Side Proving Benchmark Environment

Property	Circom/Groth16	Noir/UltraHonk
CPU	Apple M4 Pro (14 cores)	
Memory	48 GB	
OS	Darwin 25.3.0, arm64	
Runtime	Node.js v24.0.1	
Compiler	circom 2.1.9	nargo 1.0.0-beta.5
Prover	snarkjs 0.7.6 (Groth16)	bb v0.84.0 (UltraHonk)
Trusted setup	ptau-22*	N/A
Runs (median)	5 + 1 warm-up	

* $2^{22} \approx 4.19 \times 10^6$ max constraints. ZK-Regex only: ptau-21.

H.2 Circuit Complexity Scaling

Figure 21 shows that the number of R1CS constraints in Circom increases linearly with the total input size. The HEADER-HEAVY configuration (4096h/320b) lies on the same trend line, indicating that header and body bytes contribute approximately equally on a per-byte basis. The dashed line indicates the ptau-22 trusted setup limit (4.19×10^6 constraints). Figure 22 shows that the number of UltraPlonk gates in Noir also increases linearly with input size. In contrast to Circom, there is no fixed upper bound imposed by a trusted setup ceiling.

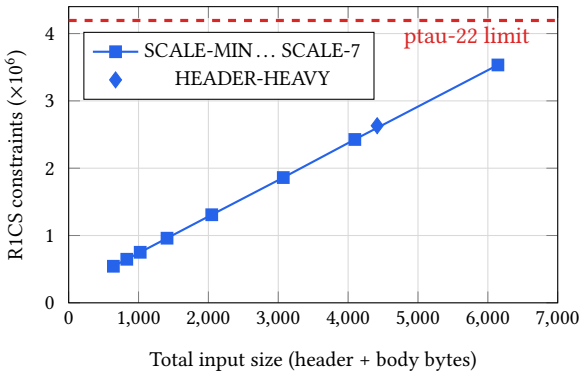


Figure 21: Circom: R1CS constraints vs. input sizes.

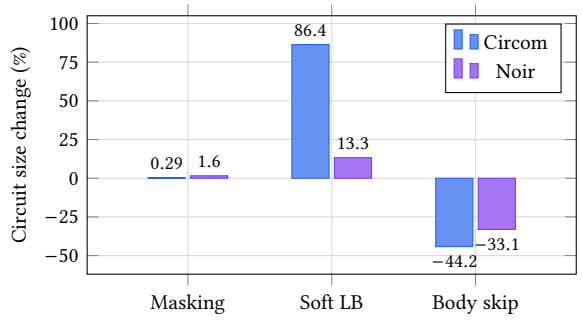


Figure 23: Cross-system feature comparison.

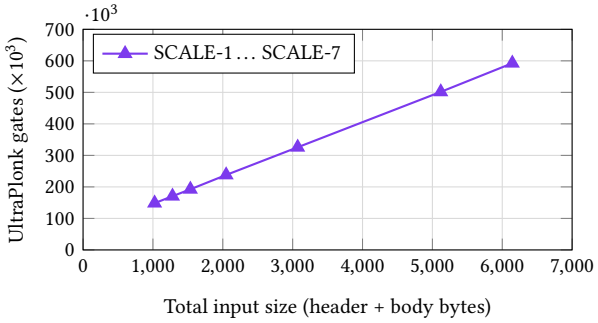


Figure 22: Noir: backend gates vs. input sizes

Figure 23 provides a direct side-by-side comparison. "Masking" corresponds to FEAT-FULL (Circom) versus FEAT-MASK (Noir); both introduce less than 2% overhead. "Soft LB" corresponds to FEAT-SOFT; the 86.4% overhead in Circom is confounded by the larger body allocation. "Body skip" compares FEAT-NOBODY (Circom, skipping body hashing) with FEAT-PARTIAL (Noir, partial hashing at 75%). The Circom FEAT-SOFT result is confounded by the increased body allocation (1408 vs. 768 bytes) and is not directly comparable to Noir's +13.3% measurement.

Table 5 reports detailed measurements of proving time and witness generation time for different feature flags relative to the baseline [4]. Additional breakdowns are provided in Table 6. Masking constraints compose additively: header masking introduces exactly 1,280 constraints, body masking introduces exactly 1,536 constraints, and enabling both equals their sum.

When `ignoreBodyHashCheck`= 1, body masking becomes a compile-time no-op, as the body is not loaded into the circuit. This confirms the correctness of the circuit's conditional compilation logic.

Table 6: Circom masking constraints are perfectly additive.

Config	Constraints	Delta from BASE
FEAT-BASE	960,202	—
FEAT-HMASK	961,482	+1,280
FEAT-BMASK	961,738	+1,536
FEAT-FULL (H+B)	963,018	+2,816 = 1280 + 1536
FEAT-NOBODY	535,950	−424,252
FEAT-NOBODY-BMASK	535,950	= NOBODY†
FEAT-NOBODY-HMASK	537,230	= NOBODY + 1,280
FEAT-NOBODY-FULL	537,230	= NOBODY-HMASK✯

† body mask is compile-time no-op. ✯ body mask no-op.

Our Noir implementation supports both RSA-1024 and RSA-2048 key sizes. Circom's standard RSA library enforces $n \times k > 2048$, and therefore RSA-1024 is supported only in Noir. We evaluate the impact of the RSA public key size on both prover and verifier complexity, as reported in Table 7. RSA-2048 costs approximately 10% more in gates and proving time vs. RSA-1024.

Table 7: RSA key size impact (Noir, `verify_email`, 512h/1024b).

Config	Gates	Prove	WitGen	Verify
RSA-1024	173,847	4.78 s	303 ms	1.93 s
RSA-2048	192,406	5.32 s	380 ms	2.07 s
Δ	−9.6%	−10.1%	−20.3%	−6.6%

Figure 24 shows RSA-2048 approximately consumes 10% more gates and proving time compared to RSA-1024. The results related to Noir implementation; Circom's `email-verifier` template enforces $n \times k > 2048$.

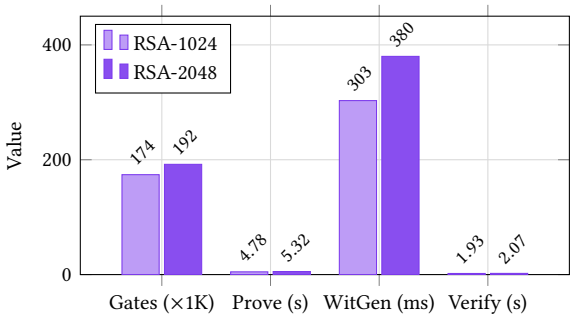


Figure 24: RSA-2048 vs. RSA-1024 costs.

Figure 25 shows the Noir SHA-256 precomputation sweep. The `partial_hash` circuit template introduces a fixed overhead due to additional partial-hash verification logic. At 25% precomputation, this overhead exceeds the gate savings obtained by reducing `maxBodyLength` from 1024 to 768 bytes, resulting in a net increase in gate count. The crossover point occurs between 25% and 50%

Table 4: Circuit size scaling data (both systems).

	Circom				Noir			
Config	**H/B**	**Constraints**	**Compile**	**Proof (B)**	**H/B**	**Gates**	**Compile**	**Proof (B)**
SCALE-MIN	384/256	543,299	9.0 s	805	384/256	115,131	1.0 s	14,080
SCALE-1	448/384	647,205	10.2 s	805	512/512	148,468	1.1 s	14,080
SCALE-2	512/512	750,853	12.0 s	805	512/768	170,437	1.2 s	14,080
SCALE-3	640/768	960,202	16.3 s	805	512/1024	192,406	1.3 s	14,080
SCALE-4	1024/1024	1,308,426	22.7 s	805	1024/1024	237,880	1.6 s	14,080
SCALE-5	1024/2048	1,860,876	35.1 s	805	1024/2048	325,756	2.0 s	14,080
SCALE-6	2048/2048	2,426,639	46.0 s	805	1024/4096	501,508	3.2 s	14,080
SCALE-7	2048/4096	3,533,585	69.1 s	805	2048/4096	592,456	3.6 s	14,080
HDR-HEAVY	4096/320	2,631,086	50.8 s	805	4096/320	450,309	2.8 s	14,080

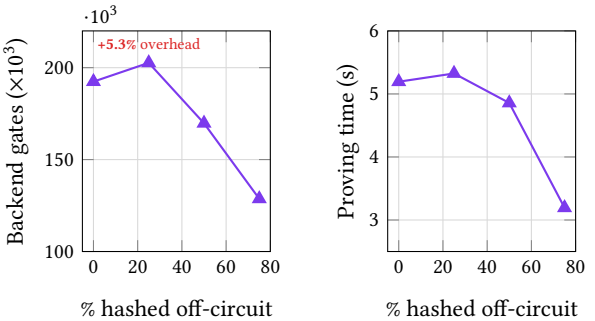
Comp.=Compile time, **Consts.**=Number of constraints. Proof size is constant in circuit size (succinct proofs): Groth16 ≈805 B (802–808 B across configs; [C]); UltraHonk 14,080 B (exact at every size; on-chain measurement [G], as the Noir off-chain harness did not record it).

Table 5: Feature flag impact on proving time (median).

	Circom (Groth16)			Noir (UltraHonk)		
Feature	**Prove**	**WitGen**	**Δ**	**Prove**	**WitGen**	**Δ**
BASE	8.30 s	1.80 s	—	5.15 s	396 ms	—
NOBODY	6.98 s	1.31 s	−15.9%	—	—	—
HMASK	8.49 s	1.80 s	+2.2%	—	—	—
BMASK	8.25 s	1.81 s	−0.7%	—	—	—
FULL	8.28 s	1.80 s	−0.3%	—	—	—
SOFT†	16.37 s	2.66 s	+97.1%	5.55 s	474 ms	+7.8%
PARTIAL	—	—	—	3.38 s	271 ms	−34.4%
MASK	—	—	—	5.33 s	636 ms	+3.4%
ADDR	—	—	—	3.12 s	331 ms	−39.4%

† Circom FEAT-SOFT uses `maxBodyLength=1408` (others use 768).

precomputation. At 75% precomputation, the circuit achieves a reduction of −33.1% in gate count and −38.5% in proving time.



(a) Gate count. **(b) Proving time (median).**

Figure 25: Impact of SHA-256 pre-computation in Noir on gate count and proving time.

In Circom, the precomputation position (10%, 50%, 80%) affects only the *input data*, i.e., the location at which the SHA intermediate state is injected, and does not modify the *circuit topology*. All PRECOMP configurations with `maxBodyLength=768` yield identical constraint counts. Reducing the number of constraints requires explicitly lowering `maxBodyLength`. This behavior is expected, as the circuit is parameterized by maximum input sizes rather than by the actual precomputation offset.

Table 8: Circom SHA precomputation: constraints and proving times.

Config	**Pos.**	**Body**	**Consts.**	**Δ**	**Prove**	**WitGen**
PRECOMP-NONE	0%	768	960,202	—	8.19 s	1.80 s
PRECOMP-EARLY	10%	768	960,202	0%	8.16 s	1.79 s
PRECOMP-MID	50%	768	960,202	0%	8.26 s	1.80 s
PRECOMP-LATE	80%	768	960,202	0%	8.21 s	1.81 s
PRECOMP-MID-SM	50%	448	787,752	−18.0%	7.64 s	1.59 s
PRECOMP-LATE-SM	80%	320	719,080	−25.1%	not proven⋆	

⋆ PRECOMP-LATE-SMALL compiled successfully but proving was not completed. **Consts.**=Number of Constraints. **Pos.**=Position.

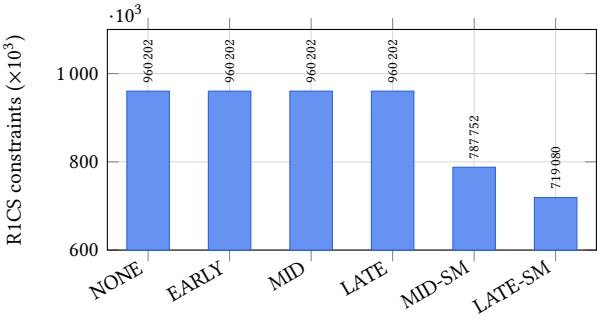


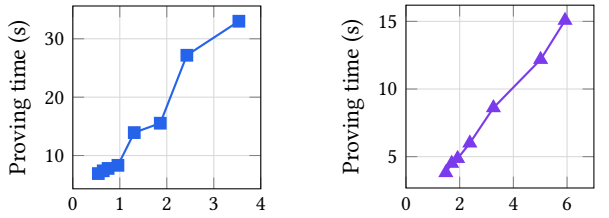
Figure 26: Circom precomputation effect: only reduced body lowers constraints.

Table 9: Circom/Groth16 verification time across all configs (median, ms).

Config	Verify (ms)	Config	Verify (ms)
SCALE-MIN	182	FEAT-BASE	185
SCALE-1	184	FEAT-NOBODY	181
SCALE-2	188	FEAT-HMASK	186
SCALE-3	186	FEAT-BMASK	185
SCALE-4	184	FEAT-SOFT	184
SCALE-5	181	FEAT-FULL	186
SCALE-6	206	PRECOMP-NONE	184
SCALE-7	205	PRECOMP-MID-SM	184
HDR-HEAVY	206		
Range: 181 ms–206 ms, **Mean:** 188 ms			

H.3 Prover Complexity

Figure 27 shows that proving time scales approximately linearly with circuit size in both systems. Circom/Groth16 exhibits a steeper slope and a visible step between 960 k and 1.3 M constraints, corresponding to a ptau power-of-two boundary.



(a) Circom/Groth16. **(b) Noir/UltraHonk.**

Figure 27: Proving time vs circuit size.

For a direct comparison, Figure 28 reports proving time at identical input sizes. Overall, Noir/UltraHonk achieves **2.0–2.3× faster** proving than Circom/Groth16. The compared configurations are Circom SCALE-2/4/5/7 and Noir SCALE-1/4/5/7, evaluated at identical header and body sizes.

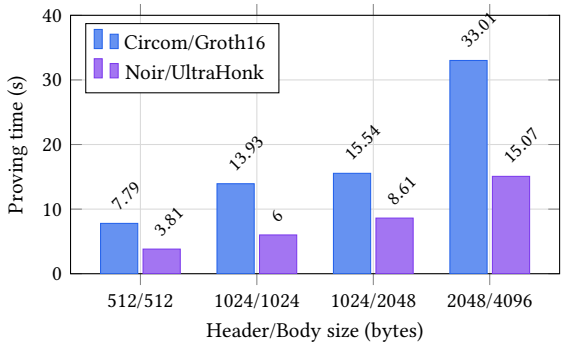


Figure 28: Head-to-head proving times at matching input sizes.

Figure 29 provides a detailed comparison of witness generation time across the same configurations. It shows Noir generates witnesses **3.2–6.3× faster** than Circom across all input sizes. Circom uses WASM-based witness computation; Noir uses native code via `nargo`.

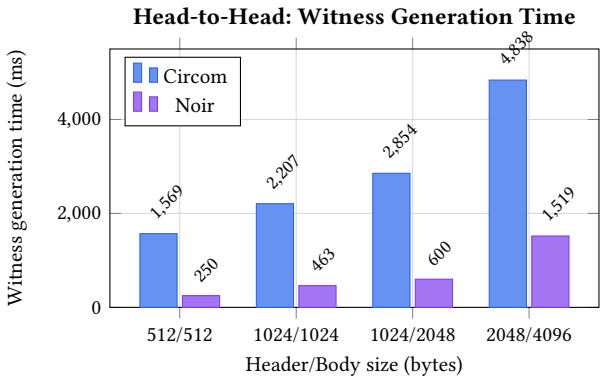


Figure 29: Witness generation time: Noir vs. Circom.

H.4 Verifier and Communication Complexity

Table 10: Proof size and verification cost comparison.

Property	Groth16	UltraHonk	Ratio
Proof size	~805 B	14,080 B	Groth16 **17.5× smaller**
Verification (small)	185 ms	1.69 s	Groth16 **9.1× faster**
Verification (large)	205 ms	5.80 s	Groth16 **28.3× faster**
Trusted setup	Required (ptau ceremony)	Not required	UltraHonk advantage
Max circuit	~4.19×10^6 (ptau-22)	No hard limit	UltraHonk advantage

H.5 On-Chain Verification Cost

We measure Ethereum-mainnet gas (Cancun EVM, `solc 0.8.28`, `forge 1.5.0`) for verifying a proof on chain. Gas is the `gasleft()` delta bracketing the `verify`/`verifyProof` call; it excludes the 21,000-gas transaction intrinsic and outer-tx calldata (added analytically). The Noir verifier uses the `keccak` transcript required for EVM verifiability.

Table 11: On-chain verifier cost. Source: `gas-benchmark/results/{groth16,honk}-gas.json` [G].

Config	System	Proof (B)	Pub.	Verify gas	×Groth16	Bytecode (B)	Pad N
SCALE-1 (512/512)	Groth16	805	20	318,559	1×	3,142	–
	UltraHonk	14,080	3	1,885,076	5.92×	21,585	2^{18}
SCALE-2 (512/768)	Groth16	805	20	318,559	1×	3,141	–
	UltraHonk	14,080	3	1,885,076	5.92×	21,583	2^{18}
SCALE-3 (512/1024)	Groth16	805	20	318,559	1×	3,141	–
	UltraHonk	14,080	3	1,885,076	5.92×	21,584	2^{18}
SCALE-4 (1024/1024)	Groth16	805	20	318,559	1×	3,141	–
	UltraHonk	14,080	3	1,885,076	5.92×	21,585	2^{18}
SCALE-5 (1024/2048)	Groth16	805	20	318,559	1×	3,143	–
	UltraHonk	14,080	3	1,914,944	6.01×	21,582	2^{19}
SCALE-6 (1024/4096)	Groth16	805	20	318,559	1×	3,142	–
	UltraHonk	14,080	3	1,914,944	6.01×	21,583	2^{19}
SCALE-7 (2048/4096)	Groth16	805	20	318,559	1×	3,141	–
	UltraHonk	14,080	3	1,944,814	6.11×	21,585	2^{20}

Groth16 verify gas is byte-identical (318,559) across all seven sizes—it depends only on the 20 public inputs, not circuit size. UltraHonk gas is a step function of the padded domain N: each extra sumcheck round adds ≈ 29,869 **gas**. SCALE-1 – 4 pad to 2^{18}, SCALE-5/6 to 2^{19}, SCALE-7 to 2^{20}; within a tier the gas is identical regardless of header/body bytes. Pad N labels follow the matched Noir SCALE mapping.

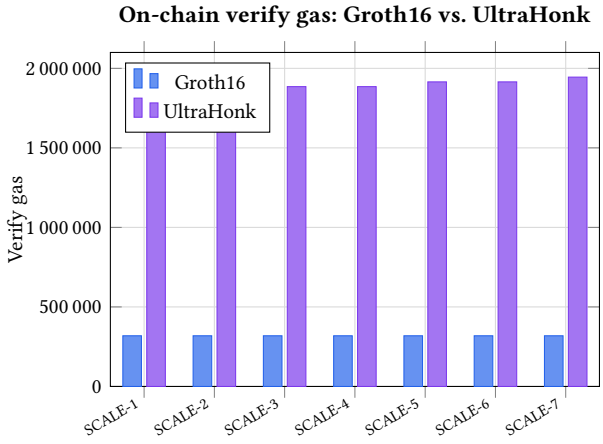


Figure 30: Groth16 verification gas is constant in circuit size (318,559; depends only on the 20 public inputs). UltraHonk is ∼6× more expensive and steps with the padded domain: 2^{18} for SCALE-1 – 4 (1,885,076), 2^{19} for SCALE-5/6 (1,914,944), 2^{20} for SCALE-7 (1,944,814)—each doubling adds ≈ 29,869 gas. Source: [G].

Observation. For applications where (*gas* × *verifications/day*) dominates, Groth16 is the economic choice (∼6× cheaper, 17.5× smaller proof, 6.9× smaller verifier bytecode). UltraHonk is preferred when the circuit exceeds the ptau-22 ceiling or when per-circuit trusted-setup ceremonies are undesirable.

H.6 Setup and Compilation Performance

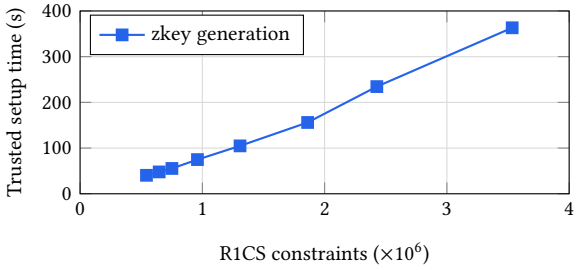


Figure 31: Groth16 trusted setup (zkey generation) scales linearly with constraint count. The largest circuit (3,533,585 constraints) requires over 6 minutes. UltraHonk has no equivalent cost.

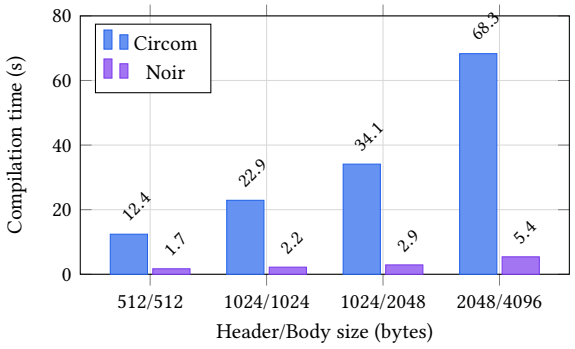


Figure 32: Noir compiles 7.3–12.7× faster than Circom at every input size. Circom compilation involves R1CS constraint generation and WASM output; Noir uses native ACIR compilation via `nargo`.

Table 12: Head-to-head comparison at matching input sizes (median values).

Size	System	Circuit Size	Prove (s)	WitGen (ms)	Verify (ms)	Proof (B)
384/256	Circom	543,299 R1CS	6.97	1308	184	805
	Noir	115,131 gates	1.36	187	561	14,080
512/512	Circom	750,853 R1CS	7.60	1557	183	805
	Noir	148,468 gates	2.15	226	975	14,080
1024/1024	Circom	1,308,426 R1CS	14.08	2210	186	805
	Noir	237,880 gates	2.90	355	1283	14,080
1024/2048	Circom	1,860,876 R1CS	15.85	2871	185	805
	Noir	325,756 gates	4.29	509	1813	14,080
2048/4096	Circom	3,533,585 R1CS	32.87	4830	208	805
	Noir	592,456 gates	7.82	1055	3208	14,080

Proof size is constant in circuit size: Groth16 ≈805 B (802–808 across all configs; [C]) vs. UltraHonk 14,080 B (exact, all sizes; on-chain measurement [G]) — Groth16 is **17.5**× smaller.

I Data Quality and Variance

I.1 Circom Proving Time Variance

Circom benchmarks use 3 runs per configuration. Table 13 shows the spread across runs.

Table 13: Circom proving time variance (3 runs, milliseconds).

Config	Min	Median	Max	CV (%)
SCALE-MIN	6908	6911	6920	0.1
SCALE-1	7371	7383	7409	0.3
SCALE-2	7647	7790	7836	1.3
SCALE-3	8289	8320	8400	0.7
SCALE-4	13889	13930	14004	0.4
SCALE-5	15447	15544	15559	0.4
SCALE-6	27091	27184	27895	1.6
SCALE-7	32219	33015	33714	2.3
FEAT-BASE	8245	8303	8339	0.6
FEAT-NOBODY	6963	6977	7072	0.8
FEAT-SOFT	16262	16366	16433	0.5
PRECOMP-NONE	8188	8189	8236	0.3
PRECOMP-MID-SM	7643	7644	7648	0.0

CV = coefficient of variation (stddev / mean × 100%). All configs show CV < 3%, indicating stable measurements.

I.2 Noir Proving Time Variance

Noir benchmarks use 5 runs per configuration.

Table 14: Noir proving time variance (5 runs, milliseconds).

Config	Min	Median	Max	CV (%)
SCALE-1	3693	3813	4439	8.5
SCALE-2	4327	4501	4587	2.7
SCALE-3	4587	4864	5583	7.7
SCALE-4	5965	6000	6328	2.4
SCALE-5	8360	8610	9195	3.5
SCALE-6	11270	12179	13559	7.3
SCALE-7	14552	15070	15483	2.9
RSA-1024	4392	4780	5117	6.6
RSA-2048	5229	5319	5604	3.0
FEAT-BASE	4762	5153	5569	5.9
FEAT-PARTIAL	3202	3378	4224	10.7
FEAT-SOFT	5471	5550	6786	8.5
PRECOMP-0	4784	5194	5405	4.7
PRECOMP-75	2943	3194	3247	3.5

Noir shows higher variance (CV up to 10.7%) than Circom (< 3%), likely due to UltraHonk's memory allocation patterns and OS-level scheduling on Apple Silicon.

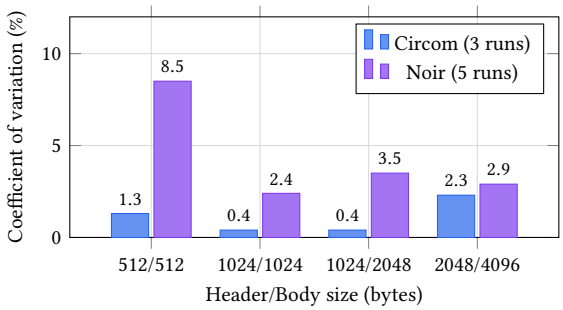


Figure 33: Proving time coefficient of variation at matching input sizes. Circom/Groth16 is more deterministic (< 3% CV), while Noir/UltraHonk shows higher variance (up to 8.5%).

I.3 Circom Public Output Sizes

Table 15: Circom public output sizes by feature configuration.

Config	Public Size (B)	Notes
BASE / NOBODY / SOFT	~874	Standard public signals
FEAT-HMASK	5244	Includes masked header bytes
FEAT-BMASK	6370	Includes masked body bytes
FEAT-FULL	10740	Both masked header + body

Masking features expose the masked content as public outputs, increasing the public signal size by 5–12×.

J ZK Regex Benchmarks

This section presents standalone benchmarks for the zk-regex compiler described in the zk-regex construction, evaluating its scaling claims. We compare three backends – DFA encoding (Circom/-Groth16), NFA encoding (Circom/Groth16), and NFA encoding

(Noir/UltraHonk) – across 15 regex patterns of increasing complexity, at input capacities from 64 to 512 bytes. All data and reproduction scripts are available in `benchmarks/` of the `zk-regex` repository.

J.1 Test Suite and Methodology

We select 15 patterns spanning four complexity tiers (Table 16): four *simple* (literals, basic character classes), four *medium* (alternation, non-capturing groups, real-world identifiers), four *complex* (DKIM headers, URL matching, capture groups), and three *NFA-only* (lazy quantifiers, unbounded ranges, nested quantified groups).

Table 16: Regex test suite. "DFA" indicates the DFA compiler can handle the pattern. Complexity is assigned by feature set, not pattern length.

Pattern	Regex	Complexity	Category	DFA
`simple`	`a*b`	simple	synthetic	✓
`literal`	`hello`	simple	synthetic	✓
`char_class`	`[a-z]+`	simple	synthetic	✓
`alternation`	`foo|bar|baz`	simple	synthetic	✓
`nested_nc`	`(?:a(?:b|c)d)+`	medium	synthetic	✓
`alphanumeric`	`[A-Za-z0-9_]+`	medium	real-world	✓
`email_basic`	`[a-zA-Z0-9._%+-]+@...`	medium	real-world	✓
`subject`	`(?:\r\n|^)subject:...\r\n`	medium	real-world	✓
`url`	`(?:https?|ftp)://...`	complex	real-world	✓
`body_hash`	`...bh=[a-zA-Z0-9+/=]+;`	complex	real-world	✓
`email_addr`	`(?:\r\n|^)to:...@...\r\n`	complex	real-world	✓
`fixed_range`	`[a-z]{3}[0-9]{2,4}[A-Z]{1,2}`	complex	synthetic	—
`lazy_quant`	`<.*?>`	NFA-only	synthetic	*
`unbounded`	`[a-z]{3,}`	NFA-only	synthetic	*
`nested_qg`	`(?:[a-z]+[0-9]+)+`	NFA-only	synthetic	*

* DFA compiler accepts these patterns unexpectedly; see §J.8.

Input scaling. Each input length is filled with *regex-matching content*, not zero-padded short strings. Three strategies are used: *repeat* (tile a template string), *extend* (grow a variable-length match portion), and *pad-with-match* (place an anchored template at the start, fill the remainder with safe filler). This ensures benchmarks measure the cost of real regex work at each capacity.

Methodology. Each pattern is benchmarked at four input capacities (64, 128, 256, 512 bytes) across all three backends. We report the median of 5–10 runs per backend per pattern, with standard deviations. Hardware: Apple M4 Max, 14 cores, 36 GB RAM, macOS Darwin 25.3.0. Software: Bun 1.3.10, `circom` 2.2.2, `nargo` 1.0.0-beta.5, Barretenberg v0.84.0.

J.2 NFA Transition Structure

The NFA compiler generates an epsilon-free NFA and encodes it differently for each backend. In **Circom**, each byte position iterates all NFA transitions; the generated template declares the count as `var numTransitions = N`. In **Noir**, the NFA is expanded into a byte-level sparse array of size N where each possible (state, byte, next_state) triple becomes a separate entry. These counts differ: a single Circom NFA transition on `[a-z]` expands to 26 Noir sparse entries (one per letter). Table 17 shows both counts for all 15 patterns.

Table 17: NFA transition counts extracted from generated circuit files. Circom transitions (`numTransitions`) count NFA-level tuples; Noir sparse entries count the expanded byte-level lookup table. Circom NFA R1CS at 64-byte input is shown for correlation.

Complexity	Pattern	Circom Trans.	Noir Entries	NFA R1CS	Noir Gates
simple	`char_class`	3	78	51,892	49,322
	`simple`	6	6	41,096	48,964
	`literal`	6	6	41,092	48,962
	`alternation`	9	9	51,892	48,977
medium	`nested_nc`	5	6	44,692	48,962
	`alphanumeric`	12	189	127,492	49,877
	`email_basic`	37	487	339,892	51,557
	`subject`	43	688	441,901	63,031
complex	`url`	31	211	217,496	49,989
	`body_hash`	66	910	630,301	65,819
	`email_addr`	74	1,062	779,101	65,275
	`fixed_range`	12	248	151,192	50,472
NFA-only	`lazy_quant`	20	501	185,092	51,641
	`unbounded`	5	130	73,492	49,582
	`nested_qg`	6	124	84,292	49,552

The Noir sparse entry count is the better proxy for Circom's actual per-byte circuit cost, because it counts the expanded byte-level transition checks that Circom must iterate. The correlation between Circom NFA R1CS and Noir sparse entries is clear: patterns with 6 entries have ~41,000 R1CS, patterns with ~500 entries have ~340,000–440,000 R1CS, and patterns with 1,000+ entries reach ~780,000 R1CS. Meanwhile, **Noir gates remain nearly flat** (48,962–65,819) across the same 6–1,062 entry range – confirming that the sparse-array lookup cost is independent of table size.

J.3 Circuit Complexity

Figure 34 shows circuit size at 64-byte input across all three backends. Two fundamental scaling behaviors emerge.

Circom NFA scales as $O(\text{match\_length} \times \text{total\_transitions})$. As stated in §5, the Circom backend iterates all NFA transitions at each byte position. The benchmark data confirms this: Circom NFA R1CS constraints range 19× across patterns (41,092–779,101) at fixed 64-byte input, directly correlated with the Noir sparse entry count (Table 17). The three most complex patterns (`body_hash`: 910 entries, `email_addr`: 1,062) require >15× more constraints than `simple` (6 entries).

Table 18 provides the complete cross-backend comparison.

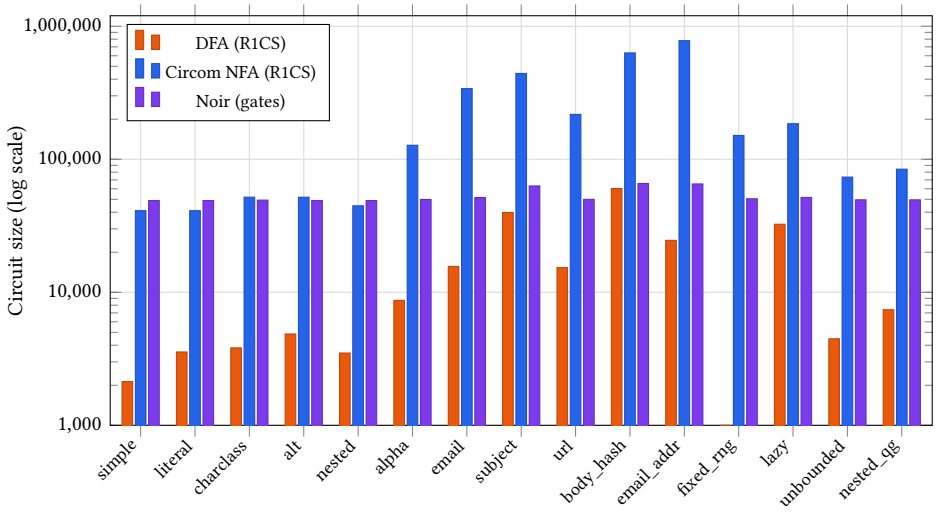


Figure 34: Circuit size at 64-byte input across 15 patterns (log scale). DFA R1CS constraints are omitted for `fixed_range` (compilation failure). Circom NFA R1CS spans a 19× range across patterns; Noir gates span only 1.34×.

Table 18: DFA vs Circom NFA constraint counts and Noir gates at 64-byte input. The NFA/DFA column shows the NFA blowup factor. `fixed_range` fails DFA compilation (accept-node limitations).

Complexity	Pattern	DFA R1CS	NFA R1CS	NFA/DFA	Noir Gates
simple	`simple`	2,131	41,096	19.3	48,964
	`literal`	3,561	41,092	11.5	48,962
	`char_class`	3,821	51,892	13.6	49,322
	`alternation`	4,861	51,892	10.7	48,977
medium	`nested_nc`	3,496	44,692	12.8	48,962
	`alphanumeric`	8,696	127,492	14.7	49,877
	`email_basic`	15,651	339,892	21.7	51,557
	`subject`	39,822	441,901	11.1	63,031
complex	`url`	15,326	217,496	14.2	49,989
	`body_hash`	60,295	630,301	10.5	65,819
	`email_addr`	24,589	779,101	31.7	65,275
	`fixed_range`	—	151,192	—	50,472
NFA-only	`lazy_quant`	32,486	185,092	5.7	51,641
	`unbounded`	4,471	73,492	16.4	49,582
	`nested_qg`	7,396	84,292	11.4	49,552

Noir scales as $O(match\_length \times lookup\_cost)$ with effectively constant lookup. Also as stated in the zk-regex construction, Noir replaces per-transition iteration with a single sparse-array lookup per byte, keyed by key $= s + b \cdot 257 + s' \cdot 257^2$. The total gate count ranges only 1.34× across all 15 patterns (48,962–65,819). Twelve of fifteen patterns cluster in the narrow band 48,962–51,641 gates; the three with slightly higher counts (`subject`, `body_hash`, `email_addr`) share large character-class alphabets and capture-group structures that expand the fixed lookup-table overhead.

The most direct test of "constant lookup cost" is how circuit size scales with input capacity (§J.4).

J.4 Scaling with Input Length

We measure circuit size at four input capacities: 64, 128, 256, and 512 bytes. Figure 35 shows three representative patterns.

DFA scales linearly with input length. DFA R1CS doubles at each doubling of input (ratio 1.99×), as expected from in-circuit DFA simulation where each byte adds a fixed per-state cost.

Circom NFA is constant within a template capacity. Circom NFA produces identical circuit sizes from 64 to 256 bytes, then steps up at 512 bytes when the compiler generates a new template with larger `max_byte_len`. This step-function behavior confirms the architecture described in §3.2: the circuit is compiled once for a fixed capacity, and all inputs within that capacity share the same constraint structure.

Noir is constant across all input capacities. Noir produces *identical* gate counts from 64 to 512 bytes for every pattern tested. Unlike Circom NFA, there is no step-up at 512 bytes – the circuit structure is entirely independent of input capacity. This is the strongest possible evidence for the constant-cost sparse-array lookup: the marginal per-byte cost is *zero*.

Marginal per-byte cost. Table 19 shows the additional circuit cost per byte when the template capacity increases from 256 to 512 bytes. This isolates the per-byte verification cost from the fixed overhead (lookup table setup, input validation, state initialization).

Table 19: Marginal per-byte cost when template capacity increases from 256 to 512 bytes. Noir gate counts are identical at both capacities (zero marginal cost). The Circom range (18.6×) reflects iteration over all NFA transitions.

		Circom NFA (R1CS)			Noir (gates)		
Complexity	Pattern	@256 B	@512 B	R1CS/byte	@256 B	@512 B	gates/byte
simple	`simple`	41,096	70,654	115.5	48,964	48,964	0.0
	`char_class`	51,892	89,082	145.3	49,322	49,322	0.0
medium	`email_basic`	339,892	580,602	940.3	51,557	51,557	0.0
	`subject`	441,901	755,203	1,223.8	63,031	63,031	0.0
complex	`body_hash`	630,301	1,076,739	1,744.7	65,819	65,819	0.0
	`email_addr`	779,101	1,330,691	2,154.6	65,275	65,275	0.0

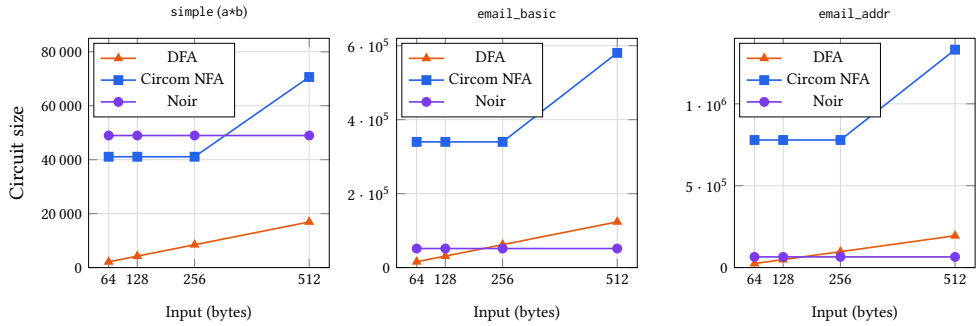


Figure 35: Circuit size scaling with input length for three representative patterns. DFA (orange) grows linearly with a 1.99× ratio at each doubling. Circom NFA (blue) is constant from 64 to 256 bytes (same compiled template), with a step increase at 512 bytes when a new template is generated. Noir (purple) is constant across all four input capacities.

In Circom, the marginal per-byte cost ranges from 115.5 R1CS/byte (`simple`, 6 sparse entries) to 2,154.6 R1CS/byte (`email_addr`, 1,062 entries) – an 18.6× range that mirrors the NFA transition count. In Noir, the marginal cost is *zero* – gate counts are identical at 256 and 512 bytes for every pattern. This is the most direct evidence that Noir's circuit structure is entirely independent of input capacity: the sparse-array lookup table is compiled once and does not grow with the input buffer.

J.5 Proving Performance

Figure 13 compares proving times at 64-byte input. We use Groth16 for Circom (DFA and NFA) and UltraHonk for Noir.

Noir is 5–15× faster than Circom NFA (Groth16). The speedup ranges from 4.8× (`char_class`: 298 ms vs. 1,442 ms) to 15.5× (`email_addr`: 548 ms vs. 8,469 ms), and correlates with regex complexity. For simple patterns, the advantage comes mainly from UltraHonk's faster per-gate proving. For complex patterns, it compounds: both the circuit is smaller *and* the prover is faster per gate.

Noir is 2–6× faster than DFA.. Even compared to the DFA's much smaller R1CS circuits, Noir proves faster thanks to UltraHonk's efficient backend. The speedup at 512 bytes grows further: 3.9× for `simple` and 7.9× for `body_hash`.

Context: regex-only vs end-to-end speedup. The 5–15× Noir speedup measured here applies to the *regex component only*. In our full pipeline (§6), SHA-256 hashing dominates circuit size and both backends have similar hashing costs, yielding the 3.5–5.1× end-to-end speedup reported in §6. The regex component exhibits a larger speedup because the bottleneck – per-transition iteration – is exactly what Noir's sparse-array encoding eliminates.

Table 20: Proving time scaling by input length (ms). Circom NFA is stable from 64–256 bytes (same template), stepping up at 512 bytes. Noir remains stable across all input sizes.

Pattern	Input (B)	DFA	Circom NFA	Noir	Noir/NFA
`simple`	64	933	1,450	281	5.2
	128	992	1,502	282	5.3
	256	1,034	1,534	282	5.4
	512	1,108	1,813	283	6.4
`email_basic`	64	2,064	4,197	373	11.3
	128	1,193	4,238	371	11.4
	256	1,392	4,230	373	11.3
	512	3,351	7,150	369	19.4
`email_addr`	64	1,322	8,469	548	15.5
	128	1,437	7,746	544	14.2
	256	1,847	8,380	547	15.3
	512	2,670	13,872	544	25.5

J.6 Verification and Proof Size

Table 21 shows the Noir backend details at 64-byte input. UltraHonk produces constant-size proofs of 14,080 B (≈13.8 kB) with verification time of ≈32 ms across all patterns.

Table 21: Noir/UltraHonk verification details at 64-byte input. Proof size is constant at 14,080 B for all patterns. Verification time is constant at ≈32 ms.

Pattern	ACIR	Gates	Prove (ms)	Verify (ms)
`simple`	26,417	48,964	281 ± 1	31 ± 1
`literal`	26,415	48,962	285 ± 3	33 ± 1
`char_class`	26,487	49,322	298 ± 2	32
`alternation`	26,418	48,977	284 ± 2	32 ± 1
`nested_nc`	26,415	48,962	284 ± 2	33 ± 1
`alphanumeric`	26,598	49,877	316 ± 1	32 ± 1
`email_basic`	26,896	51,557	373 ± 2	33 ± 1
`subject`	38,570	63,031	458 ± 3	31 ± 1
`url`	26,622	49,989	322 ± 2	31 ± 1
`body_hash`	40,008	65,819	586 ± 5	31 ± 1
`email_addr`	38,944	65,275	548 ± 5	32 ± 1
`fixed_range`	26,957	50,472	331 ± 2	32
`lazy_quant`	26,910	51,641	373 ± 2	32 ± 1
`unbounded`	26,539	49,582	305 ± 2	31 ± 1
`nested_qg`	26,533	49,552	302 ± 2	31 ± 1

The ACIR opcode counts show a bimodal distribution: 12 patterns cluster around 26,400–27,000 opcodes, while three (`subject`, `body_hash`, `email_addr`) require 38,500–40,000 opcodes. These three share large character-class alphabets and capture-group structures. Despite this, the resulting gate overhead is modest (1.34× worst case).

Note on verification time. The ≈32 ms verification measured here for 49,000–66,000 gate circuits is substantially lower than the 0.56 s–3.21 s range reported in §6 for full circuits with 115,000–592,000 gates. This difference likely reflects invocation overhead (CLI process startup, file I/O, verification-key loading) that dominates at small circuit sizes, as well as potential differences in recursive vs non-recursive verification modes.

J.7 Memory Usage

Table 22 compares compilation memory across backends for selected patterns at 64-byte input.

Table 22: Compilation memory (MB) at 64-byte input. Circom NFA requires up to 25× more memory than DFA for complex patterns. Noir compilation memory is more uniform.

Pattern	DFA Compile	NFA Compile	Noir Compile
`simple`	28	194 ± 9	385 ± 12
`literal`	36 ± 1	189 ± 8	384 ± 13
`char_class`	30 ± 1	157 ± 1	390 ± 15
`email_basic`	91 ± 2	1,064 ± 37	394 ± 26
`subject`	206 ± 7	1,923 ± 93	452 ± 50
`body_hash`	200 ± 2	2,737 ± 93	460 ± 26
`email_addr`	119 ± 4	2,968±228	485 ± 38

Circom NFA compilation memory varies widely (157 MB to 2,968 MB), scaling with NFA transition count. Noir compilation memory is more uniform (372 MB to 485 MB), reflecting the sparse-array compression. However, Noir's *proving* memory is significant: 597 MB to 1,335 MB, dominated by UltraHonk's polynomial commitment operations.

J.8 DFA Compatibility

We tested all 15 patterns against the DFA compiler. Only `fixed_range_quantifier` (`[a-z]{3}[0-9]{2,4}[A-Z]{1,2}`) fails, producing a "DFA accept nodes" error from multiple bounded range quantifiers. Surprisingly, the three patterns classified as *NFA-only* – lazy quantifier (`<.*?>`), unbounded range (`[a-z]{3,}`), and nested quantified groups (`(?:[a-z]+[0-9]+)+`) – all compile under the DFA compiler.

This has two implications. First, the DFA compiler handles a broader regex subset than expected, likely by internally expanding quantifiers and converting lazy to greedy semantics. Alternations (`|`) are handled natively by DFAs and are not a DFA limitation. Second, the true differentiator for the NFA approach is not *whether* a pattern compiles, but that the NFA-based approach preserves exact match semantics (e.g., lazy vs. greedy) that the DFA expansion may silently alter. The genuine DFA limitation is *bounded range quantifiers* (`{n,m}`) with multiple ranges, which cause DFA state explosion.

J.9 Summary

Table 23: ZK regex backend comparison summary at 64-byte input. Ranges span all 15 tested patterns.

Metric	DFA (Groth16)	Circom NFA (Groth16)	Noir (UltraHonk)
Constraints/Gates	2,131–60,295	41,092–779,101	48,962–65,819
Variation	28×	19×	1.34×
Marginal cost/byte	—	115–2,155 R1CS/B	0 gates/B
Per-byte variation	—	18.6×	N/A (zero)
Prove time	933 ms to 2,574 ms	1,410 ms to 8,469 ms	281 ms to 586 ms
Verify time	<1 ms (Groth16)	<1 ms (Groth16)	≈32 ms
Proof size	128 B (Groth16)	128 B (Groth16)	14,080 B
Regex coverage	Most patterns	All patterns	All patterns
Match semantics	Greedy only (DFA)	Exact (NFA)	Exact (NFA)

These results validate the scaling analysis the zk-regex construction:

- **Circom scales as $O(match\_length \times total\_transitions)$.** Circom NFA R1CS constraints vary 19× across patterns at fixed input and the marginal per-byte cost varies 18.6×, proportional to the NFA transition count. Within a compiled template, circuit size is constant; it steps up when a new template is generated for larger capacity.
- **Noir's circuit is entirely independent of input capacity.** Gate counts vary only 1.34× across patterns, and the marginal per-byte cost is *zero* – Noir circuits are identical from 64 to 512 bytes. This confirms that the RLC-keyed sparse-array lookup produces a fixed circuit structure regardless of both NFA size and input capacity.
- **Noir's advantage grows with pattern complexity.** The Noir-over-Circom NFA (Groth16) proving speedup scales

from 4.8× (simplest pattern) to 15.5× (most complex), and from 15× at 64 bytes to 26× at 512 bytes for complex patterns, because Noir proving time stays flat while Circom NFA increases at higher input capacities.

- **Circuit size independence from regex complexity holds for Noir, not Circom.** The claim that "circuit complexity is independent of regex size" (the zk-regex construction) is confirmed for the Noir backend (1.34× range, zero marginal cost) but does not hold for the Circom NFA backend (19× range, 18.6× marginal variation). This distinction should be qualified when stated in general terms.

The cost of Noir's advantages is a larger proof (≈14 kB vs. 128 B for Groth16) and higher verification time (≈32 ms vs. sub-millisecond), which may be acceptable for applications where prover efficiency is the primary bottleneck.

K Reproducibility

All benchmark results can be independently reproduced using the exact repository snapshots, system-level dependency installer, and step-by-step instructions below.

K.1 Repository Snapshots

Table 24: Exact repository commits used to produce the benchmark data.

Repository	Branch	Commit Hash
zkemail/zk-email-verify	feat/benchmark-and-fixes	c901dab
zkemail/zkemail.nr	chore/code-quality-improvements	e694d59

Clone and checkout the exact commits (includes setup script and all benchmark configs):

```
# Circom
git clone https://github.com/zkemail/zk-
    email-verify.git
cd zk-email-verify
git checkout
    c901dab109f65cc1dc48a3120ecce3c5661c7995


# Noir
git clone https://github.com/zkemail/
    zkemail.nr.git
cd zkemail.nr
git checkout
    e694d59eb7880423804253b7e1ec8812a5f3fd55
```

K.2 Machine Specifications

Table 25: Benchmark machine specifications.

Component	Specification
CPU	Apple M4 Pro
Cores	14 (10 performance + 4 efficiency)
Memory	48 GB unified
OS	macOS (Darwin 25.3.0)
Architecture	arm64 (Apple Silicon)

K.3 Software Versions

Table 26: Software versions used in benchmarks. All versions are pinned in `setup-macos.sh` and in each project's `benchmark.config.ts`.

Tool	Circom Stack	Noir Stack
Compiler	circom 2.1.9	nargo 1.0.0-beta.5
Prover	snarkjs 0.7.6	bb (barretenberg) v0.84.0
Backend	Groth16 (BN254)	UltraHonk
Runtime	Node.js v24.0.1	Node.js v24.0.1
Trusted Setup	Hermez ptau-22	N/A
Package managers		
yarn	3.2.3 (root workspace)	—
pnpm	latest	latest
bun	latest (setup script)	—

K.4 Automated System-Level Setup (macOS)

A single idempotent setup script installs *all* system-level dependencies on macOS. It is located at:

`zk-email-verify/scripts/benchmark/setup-macos.sh`

The script performs the following in order:

(1) **Homebrew** — installs if not present; then ensures `git`, `curl`, `wget`, `jq` are available.
(2) **Rust** (`rustup`) — required to compile the `circom` binary from source.
(3) **circom 2.1.9** — cloned from the tagged release and built via `cargo install`.
(4) **Node.js 24** via `nvm` — the JavaScript runtime used by both benchmark harnesses.
(5) **yarn, pnpm, bun** — package managers used across the two repos.
(6) **nargo 1.0.0-beta.5** via `noirup` — the Noir compiler.
(7) **bb v0.84.0** via bbup — the Barretenberg proving backend.
(8) **Powers of Tau file** (ptau-22, ~700 MB) — downloaded from the Hermez ceremony if not already present.

Run the script:

```
cd zk-email-verify/scripts/benchmark
chmod +x setup-macos.sh
./setup-macos.sh
```

After completion, `circom`, `nargo`, `bb`, `node`, `pnpm`, `yarn`, and `bun` are all available on `$PATH` at the pinned versions.

K.5 Running the Circom Benchmarks

```
cd zk-email-verify

# 1. Install root workspace dependencies
yarn install

# 2. Install benchmark-specific
    dependencies
cd scripts/benchmark && pnpm install

# 3. One-time setup: generate DKIM keys,
    sign test emails,
#    generate circuits and circuit inputs
pnpm run setup

# 4. Run the full benchmark suite (compile
    + prove + verify)
pnpm run benchmark

# --- OR run individual categories ---
pnpm run benchmark:scaling       # scaling
    configs only
pnpm run benchmark:features      # feature-
    flag configs only
pnpm run benchmark:rsa           # RSA-1024
     vs RSA-2048
pnpm run benchmark:compile-only  # compile
    + constraint count, skip proving
```

The `setup` step is required only once. It generates DKIM RSA-2048 key pairs, signs test emails via `nodemailer` + `mailauth`, generates parameterized `.circom` files, and prepares JSON circuit inputs. Subsequent runs reuse the generated artifacts.

Each proving configuration runs 3 measurement iterations plus 1 discarded warmup. The harness reports the *median* of the 3 measured runs.

The Groth16 trusted setup (`.zkey` generation) is performed once per circuit and its time is recorded separately. The ptau-22 file (`powersOfTau28_hez_final_22.ptau`) supports circuits up to $2^{22} = 4{,}194{,}304$ R1CS constraints.

K.6 Running the Noir Benchmarks

```
cd zkemail.nr

# 1. Install JS library dependencies (
    prover uses bb.js WASM in-process)
cd js && yarn install && cd ..

# 2. Install benchmark-specific
    dependencies
cd scripts/benchmark && pnpm install

# 3. Run the full benchmark suite (setup +
    gate count + prove + report)
pnpm run benchmark:all

# --- OR run steps individually ---
pnpm run setup              # generate keys
    , emails, circuits
pnpm run gate-count         # nargo compile
     + bb gates
pnpm run generate-inputs    # prepare
    prover inputs
pnpm run benchmark:prove    # prove +
    verify (UltraHonk)
pnpm run report             # generate CSV/
    JSON results
```

Unlike Circom, Noir proving phase runs *in-process* using `@aztec/bb.js` (WASM). The gate-counting phase invokes `nargo compile` and `bb gates` as subprocesses. No trusted setup or ptau file is required (UltraHonk is a universal SNARK).

Each configuration also runs 3 measured iterations plus 1 discarded warmup, with the *median* reported.

K.7 Benchmark Configuration

Both benchmark harnesses are configured via TypeScript files. Key parameters are defined in `scripts/benchmark/config/benchmark.config.ts`:

Table 27: Benchmark harness parameters (identical for both systems).

Parameter	Circom	Noir
Measurement runs	3	3
Warmup runs	1 (discarded)	1 (discarded)
Statistic reported	Median	Median
Compile timeout	60 min	30 min
Prove timeout	30 min	30 min
Memory limit	32 GB (Node)	—
Trusted setup	ptau-22	N/A
Proving system	Groth16	UltraHonk (honk)

Circuit configurations (header/body sizes, feature flags) are defined in `scripts/benchmark/config/circuits.config.ts` in each repository.

K.8 Data File Locations

Table 28: Benchmark data files (relative to each repository root).

Data	Path
Circom (`zk-email-verify/`)	
Raw JSON (per-run)	`scripts/benchmark/results/raw/benchmark_2026-02-26*.json`
CSV (medians)	`scripts/benchmark/results/csv/benchmark_2026-02-26*.csv`
Summary report	`scripts/benchmark/results/summary_2026-02-26*.md`
Circuit configs	`scripts/benchmark/config/circuits.config.ts`
Harness config	`scripts/benchmark/config/benchmark.config.ts`
Noir (`zkemail.nr/`)	
Raw JSON (per-run)	`scripts/benchmark/results/raw/benchmark_2026-02-26*.json`
CSV (medians)	`scripts/benchmark/results/csv/benchmark_2026-02-26*.csv`
Circuit configs	`scripts/benchmark/config/circuits.config.ts`
Harness config	`scripts/benchmark/config/benchmark.config.ts`

K.9 Notes for Reproducers

(1) **Disk space.** The Circom stack (compiled circuits, `.zkey` files, ptau) may require up to 20 GB of disk space. The Noir stack requires approximately 5 GB.
(2) **Memory.** The largest Circom circuit (2048/4096, ~3.5M constraints) consumes ~29 GB peak RSS during Groth16 proving. A machine with at least 32 GB of memory is recommended.
(3) **Expected wall-clock time.** A full Circom run (all 19 configs, compile + setup + prove) takes approximately 2–3 hours. A full Noir run (18 configs) takes approximately 30–45 minutes. The Groth16 trusted setup (zkey generation) dominates Circom wall time for large circuits.
(4) **macOS `/usr/bin/time` for memory.** Circom peak memory is measured using `/usr/bin/time -l`, which reports maximum resident set size on macOS. This is not available on Linux; the measurement code would need adaptation for `/usr/bin/time -v` on Linux.
(5) **Non-macOS systems.** The `setup-macos.sh` script is macOS-specific (Homebrew, Darwin conventions). On Linux, replace Homebrew with your package manager, and install `circom`, `nargo`, and `bb` using the same version manager commands (`cargo install`, `noirup`, `bbup`).
(6) **Variance.** Close all other applications during benchmarking. All measured configurations exhibit a coefficient of variation (CV) $\leq 1.1\%$ across 3 runs (see Appendix I).