Reef: Fast Succinct Non-Interactive Zero-Knowledge Regex Proofs

Sebastian Angel, Eleftherios Ioannidis, and Elizabeth Margolin, *University of Pennsylvania;* Srinath Setty, *Microsoft Research;* Jess Woods, *University of Pennsylvania*

https://www.usenix.org/conference/usenixsecurity24/presentation/angel

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Reef: Fast Succinct Non-Interactive Zero-Knowledge Regex Proofs

Sebastian Angel⋆ Eleftherios Ioannidis⋆ Elizabeth Margolin⋆ Srinath Setty† Jess Woods⋆
⋆*University of Pennsylvania* †*Microsoft Research*

Abstract

This paper presents Reef, a system for generating publicly verifiable succinct non-interactive zero-knowledge proofs that a committed document matches or does not match a regular expression. We describe applications such as proving the strength of passwords, the provenance of email despite redactions, the validity of oblivious DNS queries, and the existence of mutations in DNA. Reef supports the Perl Compatible Regular Expression syntax, including wildcards, alternation, ranges, capture groups, Kleene star, negations, and lookarounds. Reef introduces a new type of automata, *Skipping Alternating Finite Automata* (SAFA), that skips irrelevant parts of a document when producing proofs without undermining soundness, and instantiates SAFA with a lookup argument. Our experimental evaluation confirms that Reef can generate proofs for documents with 32M characters; the proofs are small and cheap to verify (under a second).

1 Introduction

Regular expressions (regex) are used to represent and match patterns in text documents in a variety of applications: content moderation, input validation, firewalls, biology, and more. Existing use cases assume that the regex and the document are both readily available to the querier so they can match the regex on their own with standard algorithms. But what about situations where the document is actually held by someone else who does not wish to disclose to the querier anything about the document besides the fact that it matches or does not match a particular regex? While slightly unusual, the ability to prove such facts enables interesting new applications:

• *Proving strong passwords.* Asymmetric or Augmented Password Authenticated Key Exchange (aPAKE) [43, 70, 73, 78] allow clients to register and authenticate to a server without disclosing their password to the server. However, aPAKE protocols have no mechanism for the server to confirm that the client chose a "strong password". This feature is crucial in corporate settings where password policies help prevent account compromise. Clients could convince the server of this fact with a proof that their secret password satisfies a password strength regex chosen by the server (e.g., at least 10 alphanumeric and one special character).

• *Disclosing content with redactions.* DomainKeys Identified Email (DKIM) [41] is a protocol whereby a sending mail server signs the header and payload of an email so that recipients can verify its authenticity. Journalists use DKIM signatures to establish the veracity of leaked emails. It might often be desirable to release a redacted version of an email (e.g., an email without a name) while allowing the public to confirm, via DKIM, the authenticity of the redacted email. By creating a regex that expresses the public content of the email, with redactions being expressed as wildcards with Kleene star, it is possible to show that the redacted email is derived from an email whose DKIM signature verifies under the sending mail server's public key. A similar idea is that of selectively disclosing fields in JSON web tokens [45] or verifiable credentials [10, 57] by "redacting" all other entries.

• *ODoH blocklisting. Oblivious DNS over HTTPS* [49] allow clients to obtain a domain's IP address without revealing which domain they are accessing. This technology improves privacy for users, but network administrators within organizations lose the ability to block certain sites (e.g., known malware domains) as they can no longer see which domains users query. One can reintroduce this functionality by asking clients to generate ZKPs showing that their DNS queries do not match a set of forbidden regexes before those packets are allowed through to the ODoH proxy. The same idea applies more generally to TLS traffic through middleboxes [40].

• *Proofs about genes.* DNA is used to establish ancestry or the presence of particular mutations. If sequencing companies (e.g., 23andme) were to provide users a signed commitment to their sequenced genome, users would be able to prove properties of their DNA (expressed as a regex) without having to disclose it in full. For instance, users could prove the presence of a certain genetic mutation when they order personalized medicine online or sign up to clinical trials.

In theory, the above applications can be designed with some suitable combination of encryption, commitments, signatures, and zero-knowledge proofs. In practice, creating efficient proofs over arbitrary unstructured text is far from trivial.

This is precisely the problem we tackle with *Reef*, a compiler and runtime system that allows an entity to *commit to a secret document and then subsequently prove that the document matches or does not match one or more public regexes without revealing anything else about the document.* Building Reef requires answering the following research questions:

1. How should one commit to a text document D?
2. Given a commitment to a document D, how can one *arithmetize* (i.e., express as some type of circuit) the statements "D matches/does not match a regex $\mathcal{R}$"?

3. What regex features are needed to enable realistic applications (e.g., quantifiers, alternation, lookarounds, etc.) and what is the best way to arithmetize these features?
4. What kind of zero-knowledge proof systems work well with the chosen commitment and arithmetization schemes?

To answer these questions, Reef marries new theoretical ideas and low-level techniques into a compiler that automatically arithmetizes arbitrary regexes. In particular, Reef:

Exploits NP checkers. Reef uses the common observation that checking the answer of a computation is often cheaper than finding the answer in the first place (either asymptotically or concretely). As a result, Reef does not arithmetize algorithms for finding regex matches/non-matches (e.g., Thompson's NFA [74], recursive backtracking). Instead, the prover in Reef computes the answer (i.e., finds the match and the relevant locations within the document, or establishes that there is no match) with a fast regex engine we built, and then proves that this answer satisfies criteria that implies the document has a match (or no match). Only this *NP checker* needs to be arithmetized and proven with a ZK proof system.

Reef's NP checker supports a wider class of regexes than all prior works, while also producing smaller arithmetizations. In particular, some works [11, 32] transform the regex into a DFA or NFA, and then prove that if one feeds the *entire* document into the automaton the final state is accepting/non-accepting. This approach results in $\mathcal{O}(|\mathsf{D}| \cdot |Q_{DFA}| \cdot |\Sigma|)$ constraints (or gates in some arithmetic or boolean circuit) to prove that there is a match, where D is the secret document, Q_{DFA} is the set of states in the DFA, and Σ is the alphabet. Three recent proposals, ZK-Regex [56], Zombie [79], and `zkreg` [64] reduce these costs: ZK-Regex and Zombie leverage Thompson's NFA (TNFA) and produce $\mathcal{O}(|\mathsf{D}| \cdot |Q_{TNFA}|)$ constraints, while `zkreg`'s use of Aho-Corasick DFA (ADFA) leads to $\mathcal{O}(|\mathsf{D}| + |Q_{ADFA}|)$ constraints.

Reef's NP checker is fundamentally different from the above approaches: it does not require feeding the entire document into the automata, only the relevant characters. This allows the prover to skip vast amounts of unnecessary work.

Introduces skipping automata. Above we allude to the idea of "skipping" irrelevant characters whenever possible. But how do we rigorously define this notion and what does "whenever possible" mean? To answer these questions, we introduce a new type of finite automata for regexes that we call *Skipping Alternating Finite Automata* (SAFA). SAFA generalize NFA to include the ability to change the *cursor* (i.e., the index of the next character to read in the input) following certain rules. SAFA allow Reef's prover to skip processing entire chunks of a document when the regex contains wildcard ranges such as "`.*`" or "`.{4,100}`" and let Reef handle *lookarounds*, which are common in password strength regexes and which no prior work supports.

Compared to prior works, Reef's NP checker can be expressed in $\mathcal{O}(\alpha \log(|\mathsf{D}| + |Q_{SAFA}| \cdot |\Sigma|))$ constraints, where $|Q_{SAFA}| \leq |Q_{TNFA}| \leq |Q_{ADFA}|$ and $\alpha = \mathcal{O}(|\mathsf{D}| \cdot L)$, where L is the number of lookarounds in the regex. There are two points worth emphasizing about the complexity of Reef's checker. First, SAFA have exponentially fewer states than TNFA and ADFA for many common regexes (§3.2). Second, α is much smaller than the above worst-case upper bound whenever Reef can skip characters. For instance, if $\Sigma = \{a, b, c\}$, the regex $\mathcal{R} =$ "`a.*b`" (meaning D has "a" eventually followed by "b") results in $\alpha = 2$ regardless of the size of D because Reef can skip all the wildcard characters. In contrast, $\mathcal{R} =$ "`^[a-b]*$`" (meaning D can contain any number of "a" or "b" characters but no "c") results in $\alpha = |\mathsf{D}|$ because we fundamentally have to check every character in the document to make sure it is not "c".

Leverages recursion. We observe that Reef's NP checker essentially performs the same high-level operations (looking up a character in the document and then transitioning to a new state) over and over. Such repeated structure is suitable for *recursive* zkSNARKs such as Nova [52], where the prover establishes that it ran some *step function* F, each time on a different input, until some terminating condition holds. Reef's termination condition is designed to allow the prover to safely stop proving as soon as the SAFA reaches an accepting state and the cursor points to the last character. This frees the prover from having to process the entire document (since in many SAFA the prover can skip to the last character without changing states) while hiding how many times F executes.

Commits to the document. Before Reef can be used, the document D needs to be committed in a form that allows Reef's NP checker to cheaply read arbitrary entries in D. Who generates the commitment depends on the application. In the gene example, the commitment is generated and signed by a trusted party (23andme). In the other applications, the commitment is generated by the user who must also supply a proof that ties the underlying document to the data in the application (e.g., the DKIM signature).

Reef uses a *polynomial commitment* [13, 23, 37, 54, 76, 80] for multilinear polynomials to commit to D, and a *lookup argument* [51] compatible with recursive proof systems. A lookup argument is a cryptographic protocol for proving that some entry exists in a public or committed table (polynomial) without revealing the entry. When the lookup argument is integrated into the step function F, it allows F to access any entries in D without revealing them to the verifier.

Supports table projections. Reef modifies the `nlookup` argument [51] to support lookup operations on *table projections*. Given a commitment to a table such as the document D, a projection is a smaller table D_{proj} derived from one or more contiguous chunks of D (the choice of which chunks of D are projected is public information). Reef then runs `nlookup` on D_{proj}, which incurs costs that are proportional to $|\mathsf{D}_{proj}|$. The verifier can still check that all lookups to D_{proj} were done correctly by using the original commitment to D.

$r, s ::=$	α	$\alpha \in \Sigma$
|	`^` / `$`	document start / end
|	`.`	wildcard character
|	rs	concatenation
|	$r \mid s$	alternation
|	$r?$ / $r*$ / $r+$	quantifiers
|	$[\alpha_i - \alpha_j]$	character classes
|	$[\hat{}\alpha_i \ldots \alpha_j]$	negation of characters $\alpha_i \ldots \alpha_j$
|	r`{m}` / r`{m,}` / r`{m,n}`	repetition ranges
|	`(?=r)` / `(?<=r)`	lookahead / lookbehind

FIGURE 1—Reef supports the entire PCRE syntax [7] except for backreferences and subroutine references.

Table projections are a powerful construct in Reef and might be of independent interest. For example, a DNA chromosome results in a document D with tens of millions of entries. However, regexes on DNA usually have the form: $\mathcal{R}$ = "`.{1000}TT(T|C).{5000}CT(T|C|A|G).*`", which says that the first thousand entries are irrelevant, but right after we should see *TTT* or *TTC*, and 5000 entries later we should see *CTT*, *CTC*, *CTA*, or *CTG*; beyond that is irrelevant. SAFA allows Reef to skip all the irrelevant entries. However, in each step of the recursive proof system, `nlookup` internally invokes the *sum-check protocol* [55] which incurs costs linear in $|\mathsf{D}|$ (millions of entries) in order to prove the table accesses. With projections, `nlookup` runs the sum-check protocol over D_{proj} (under 10 entries).

Combines private and public tables. To efficiently express the state transitions and complex skipping rules in SAFA, Reef again uses a lookup argument. In particular, Reef stores SAFA's states, transitions, and skipping rules in a public table that both the prover and the verifier can derive from the regex. Given this table, the prover can, with one lookup, prove that it transitioned to the next state in the SAFA and advanced the cursor following the prescribed rules.

Having both a private table and a separate public table is undesirable because lookup arguments amortize their costs over many lookups (i.e., the more lookups to a table, the cheaper the per-lookup cost). If one has two tables, then queries to one table do not apply towards the amortization of queries to the other table. To remedy this situation, Reef shows how to combine both private and public tables into a single *hybrid* table (without leaking the contents of the private table) so that all lookups can be done on this combined table, improving amortization and eliminating repeated fixed costs.

We evaluate Reef on the applications described earlier and find that it can generate small proofs (tens of KB) in a few seconds, even for large documents such as DNA chromosomes.

2 Background

This section reviews regex matching, rank-1 constraint satisfiability (R1CS), NP checkers, and zero knowledge succinct non-interactive arguments of knowledge (zkSNARKs).

2.1 Regular Expression Matching

Given an alphabet Σ, a regex $\mathcal{R}$ is a pattern matching a set of strings, called the *language* of $\mathcal{R}$ or $\mathcal{L}[\![R]\!] \subseteq \Sigma^*$. Figure 1 outlines the basic syntax for the creation and combination of regexes that Reef supports.

Regexes are converted to *deterministic finite automata* (DFA) with known techniques [16, 22, 38, 46, 60, 72]. One can determine if a document matches a regex $\mathcal{R}$ by starting with the initial state and transitioning states on each character of the document until reaching a final state. If the final state an accepting states in the DFA, the document matches $\mathcal{R}$.

A common extension to regexes that Reef supports is *lookarounds* (e.g., positive or negative lookaheads and lookbehinds), a way to only match a pattern if is lead (or followed) by another pattern. For example, a password strength regex with two lookaheads might look like `^(?=.*[A-Z])(?=.*[!@#$&^*]).{10,}`, meaning it contains an upper case letter (`[A-Z]`), a special character from `{!,@,#,$,&,^,*}`, and has length at least 10 characters. The way to think about a lookaround such as "`^(?=`$\mathcal{R}$`)`" for some regex $\mathcal{R}$ is that $\mathcal{R}$ should be matched against the input string in the usual way, but once the match has been found, the *cursor* (i.e., the next position to process in the input string) should be reset back to what it was before the lookaround was processed. DFA/NFA have no notion of "resetting the cursor" and hence must simulate it by increasing the number of states exponentially [31].

2.2 zkSNARKs

A *zero-knowledge succinct non-interactive argument of knowledge* (zkSNARK) is a cryptographic protocol where a prover $\mathcal{P}$, convinces a verifier $\mathcal{V}$, that it knows a satisfying witness to some NP statement without revealing the witness. zkSNARKs typically target some variant of the NP complete problem of *circuit satisfiability* (e.g., R1CS [36, 67], Plonkish [35], AIR [17], CCS [68]), as one can represent arbitrary computations in this form. Informally, zkSNARKs are:

1. **Zero-knowledge:** The proof reveals no information to $\mathcal{V}$ beyond the fact that $\mathcal{P}$ knows a satisfying witness.
2. **Succinct:** The size of the proof and its verification is sublinear in the size of the satisfiability instance.
3. **Non-interactive:** No interaction between $\mathcal{P}$ and $\mathcal{V}$ besides the transferring of the computation's output and proof.
4. **Argument of knowledge:** $\mathcal{P}$ must convince $\mathcal{V}$ that it knows a witness that satisfies the instance. This argument is complete and computationally sound.

- *Perfect completeness:* If $\mathcal{P}$ knows a satisfying witness, $\mathcal{P}$ can always generate a proof that convinces $\mathcal{V}$.
- *Knowledge Soundness:* If $\mathcal{P}$ does not know a satisfying witness, it cannot produce a proof that $\mathcal{V}$ will accept, except with negligible probability.

2.3 Rank-1 Constraint Satisfiability (R1CS)

We focus on *rank-1 constraint satisfiability* (R1CS) as this is the arithmetization supported by the particular implementation of the zkSNARK we use [52], but all of our ideas apply to more general arithmetizations (e.g., CCS [68]). R1CS generalizes arithmetic circuit satisfiability, and an R1CS instance is given by a tuple $(\mathbb{F}, A, B, C, io, rows, cols)$, where $\mathbb{F}$ is a finite field, io is the public input and output of the instance, $A, B, C \in \mathbb{F}^{rows \times cols}$ are matrices, and $cols \geq |io| + 1$. The instance is satisfiable if and only if there exists a witness $w \in \mathbb{F}^{cols-|io|-1}$ that makes up a solution vector $z = (io, 1, w)$ such that $(A \cdot z) \circ (B \cdot z) = (C \cdot z)$, where $\cdot$ is the matrix-vector product and $\circ$ is the Hadamard product. The entry of z fixed at 1 allows constants to be encoded.

R1CS Arithmetization. Here we briefly explain how to turn a simple program into R1CS. Other works [14, 67, 69] have more complex examples. Suppose that $\mathcal{P}$ holds two elements $x_0, x_1 \in \mathbb{F}$ and wishes to convince $\mathcal{V}$ that y is the output of the following computation without leaking anything about x_0 or x_1 beyond what is implied by the result.

```
field foo(field x0, field x1) {
    field y;
    if (x0 == 30) { y = x1; } else { y = x0/x1; }
    return y;
}
```

To do so, we first express this function as a set of *constraints* (or equations) over elements in $\mathbb{F}$ that contain additions, subtractions, multiplications by constants, and at most one multiplication between variables. The result is:

$$\begin{aligned} guard \times (x_0 - 30) &= 0 \\ guard \times (y - x_1) &= 0 \\ (1 - guard) \times (y - prod) &= 0 \\ x_0 \times inv - prod &= 0 \\ x_1 \times inv - 1 &= 0 \end{aligned}$$

To see why this represents the original computation, observe that we introduced auxiliary variables called *guard*, *inv*, and *prod*. Here, $\mathcal{P}$ is allowed to assign any values it wishes to y and the auxiliary variables, but let us assume that $\mathcal{P}$ provides the right values for x_0 and x_1 (this is usually enforced through the use of commitments). The only way that all six constraints are simultaneously satisfied is when: (1) $x_0 = 30$, $y = x_1$, and $guard = 1$ (there are many suitable values for the remaining variables); or (2) $x_0 \neq 30$, $guard = 0$, $y = prod = x_0 \times inv$, and $inv = x_1^{-1}$. As a result, if $\mathcal{P}$ claims that the output is y, and $\mathcal{P}$ can convince $\mathcal{V}$ that it knows a satisfying assignment for variables in the constraints given y, then $\mathcal{V}$ is assured that y is correct.

Our tech report [15] shows how to convert these constraints into matrices A, B, and C. The solution vector z is $(y, 1, w)$, where $w = (x_0, x_1, guard, prod, inv)$ is $\mathcal{P}$'s secret witness.

2.4 NP checkers

While the above example is relatively simple it employs some clever tricks. In particular, it leverages *non-determinism* to transform expensive computations (branches and inverses) into cheap checkers that merely confirm the answers. For instance, if $\mathbb{F} = \mathbb{Z}_p$, *computing* $1/x$ with only additions and multiplications requires $\log(p)$ constraints via Fermat's little theorem (basically computing x^{p-2}). But in R1CS, we can just ask $\mathcal{P}$ to supply the inverse of x, *inv*, and simply *check* that *inv* is indeed the multiplicative inverse of x with one constraint: "$inv \times x - 1 = 0$". This is an example of an *NP checker*. There are many others used in SNARKs [14, 21, 44, 69, 77, 81].

In this work, we construct a novel NP checker for regex matching/non-matching based on a new type of automata.

3 Goals and standard approach

In Reef there are three parties: a committer $\mathcal{G}$, a prover $\mathcal{P}$, and a verifier $\mathcal{V}$ (in many cases $\mathcal{G}$ and $\mathcal{P}$ are the same entity). $\mathcal{G}$ generates a commitment *comm* for document D using random blind r, and provides $(comm, \mathsf{D}, r)$ to $\mathcal{P}$, and *comm* to $\mathcal{V}$. Later, $\mathcal{P}$ wishes to prove that D either does or does not match a regex $\mathcal{R}$ that is public and known to both $\mathcal{P}$ and $\mathcal{V}$. Given this setting, Reef has the following goals:

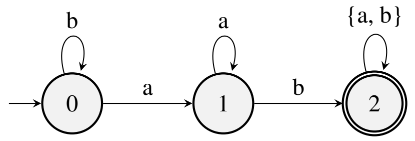
- **Completeness, Soundness, Succinctness, ZK**. These are analogs of the definitions given for zkSNARKs (§2.2) for the concrete R1CS instance that represents the statement "I know an opening of *comm*, and it matches $\mathcal{R}$" (or not).
- **Public verifiability**: The proof should be verifiable by anyone who has a commitment of the document and $\mathcal{R}$.
- **Expressiveness**: Reef should be able to support any regex written in PCRE syntax [7].

Additionally, our implementation of Reef achieves the following goal, though some settings might not need this and could use more efficient cryptographic primitives.

- **Transparency**: All cryptographic parameters for Reef should be generated without requiring a trusted setup.

3.1 A standard approach

As mentioned in Section 2.1, one can convert a regex into a DFA and then arithmetize its transition function δ. It boils down to a chain of if statements that takes as input the current state and current character in the document (both represented as field elements) and outputs the next state. For example, if the alphabet is $\Sigma = \{a, b\}$, and the regex is $\mathcal{R} =$ "`a+b.*`", the corresponding DFA would be:



Assuming that "a" maps to the field element 0, and "b" to 1, the corresponding δ transition is given by:

```
field delta(field state, field cur_char) {
    if (state == 0 && cur_char == 0) return 1;
    if (state == 0 && cur_char == 1) return 0;
    if (state == 1 && cur_char == 0) return 1;
    if (state == 1 && cur_char == 1) return 2;
    if (state == 2 && cur_char == 0) return 2;
    if (state == 2 && cur_char == 1) return 2;
    return -1; // invalid state or character
}
```

To express the computation of finding whether a committed document matches the regex, one would then: (1) open the commitment to obtain the document (an array of field elements); (2) call δ once for every character in the document in order; and (3) add a check at the end to see if the final state is one of the accepting states (another chain of if statements). The resulting `match` function is:

```
field match(field commit, field blind) {
  // commit is public input, blind is secret
  field[SIZE] document = open(commit, blind);
  field state = 0; // initial state

  for (i = 0; i < SIZE; i++) {
    state = delta(state, document[i]);
  }

  if (state == 2) { // accepting state in example
    return 1; // match
  } else {
    return 0; // no match
  }
}
```

One would then arithmetize this `match` function like in the example in Section 2.3. Indeed, this what some prior works do [11, 32]. Two recent works [56, 79] improve upon this design by converting the regex to a Thompson NFA (TNFA) [74] and performing additional optimizations.

3.2 Limitations of the standard approach

The previous standard approach has many drawbacks. We list the most salient ones here.

Insufficient Regex Expressiveness. Directly arithmetizing traditional finite state machines such as DFA, TNFA or Aho-Corasick DFA (AC-DFA) [12] fails to meet Reef's expressiveness goals. The most recent works in this area lack support for several common regex features.

For example, Zombie [79] lacks support for lookarounds. ZK-Regex [56] does not handle lookarounds, negations in character classes such as "`a[^[:space:]b`", or negations of entire matches (i.e., proving a non-match). Finally, `zkreg` [64], which is based on AC-DFA, only supports matching on a fixed set of strings. Unbounded repetition such as "`ab*c`" is unsupported, and negation of character classes, negation of entire matches, or wildcard ranges such as "`a.{100}b`" lead to an exponential number of states (2^{100}).

Poor scalability. The number of R1CS constraints produced by the standard approach for proving that a document D matches is $\mathcal{O}(|\mathsf{D}| \cdot |Q_{DFA}| \cdot |\Sigma|)$, where $|Q_{DFA}|$ is the number of states of the corresponding DFA. Zombie [79] improves this to $\mathcal{O}(|\mathsf{D}| \cdot |Q_{TNFA}|)$. But for applications where the document is millions of characters this still results in *billions of constraints*, even when the regex is small. In contrast, Reef's NP checker—based on SAFA (§5)—has $\mathcal{O}(\alpha \log(|\mathsf{D}| + |Q_{SAFA}| \cdot |\Sigma|))$ constraints, where $|Q_{SAFA}| \leq |Q_{TNFA}|$. As we discuss in Section 6.2, in the worst case $\alpha = \mathcal{O}(|\mathsf{D}| \cdot L)$, where L is the number of lookarounds in the regex; but in practice α is small (under 100 for even our largest document).

4 Improving the standard approach

One way to improve on the standard approach is to observe that the `match` function is well suited for a recursive proof system (this observation has been made many times in the context of other state machines such as blockchain rollups). In a *recursive zkSNARK* [18–20, 24–26, 50, 51], instead of arithmetizing the entire `match` function, we arithmetize one *step* of it. The result is:

```
field[3] match_step(field[] commit, field[] blind,
    field state, field cursor) {

  field cur_char = open_at(commit, blind, cursor);

  // accepting state and end of document (EOD)
  if (cur_char == EOD && state == 2) {
    return {0, 0, 1}; // match
  }

  state = delta(state, cur_char);
  return {state, cursor + 1, 0}; // not yet
}
```

The above `match_step` function takes as input a public *polynomial* [13, 23, 37, 47, 54, 76, 80] or *vector* [59] commitment (which could consist of multiple field elements) and the corresponding secret blind(s). These types of commitments have the nice property that they allow opening a particular entry within the commitment rather than having to open the entire document at once. `match_step` additionally takes the current state and the current cursor. If the current state is accepting and the cursor points to the end of D ("\$" in PCRE syntax, denoted by a special field element that the committer $\mathcal{G}$ appends to D to mark the end), D is a match and the return value is [0, 0, 1]. Else, `match_step` executes the DFA's δ function and returns the tuple [`state`, $cursor + 1$, 0].

A prover $\mathcal{P}$ in a recursive zkSNARK would then take the R1CS instance representing the `match_step` function, and produce a proof π_0 that establishes that running `match_step` correctly on a public commitment, private blinds, $state = 0$, and $cursor = 0$, produces the output out. Of course, proving a single step is not very useful (we could have done this without recursion); the key benefit is that a recursive proof system allows $\mathcal{P}$ to prove that it verified a prior proof (π_0 in this context) in addition to proving another `match_step` on the same public commitment, but the state and cursor returned

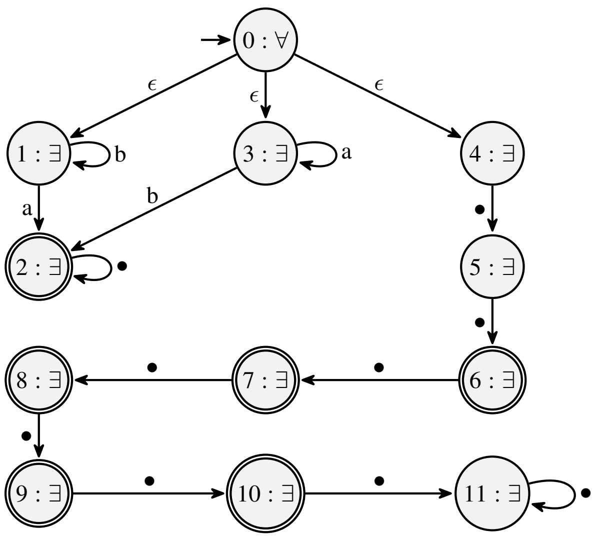


FIGURE 2—AFA for regex $\mathcal{R}$ = `^(?=.*a)(?=.*b).{2,6}$`.

by the prior step (*out*) which are bound by π_0. In this way, $\mathcal{P}$ can prove that, starting with $state = 0$ and $cursor = 0$, if $\mathcal{P}$ runs `match_step` *some* number of times, eventually $out = [0,0,1]$. The verifier $\mathcal{V}$ only learns this final value of *out* (and none of the intermediate values), in addition to a proof π_{final} that establishes that $\mathcal{P}$ checked all prior proofs and the last step was executed correctly.

This approach has four benefits. First, there is no need to unroll the loop and therefore the number of R1CS constraints is no longer fundamentally tied to the size of the document. This enables the second benefit: $\mathcal{P}$ can stop proving as soon as `match_step` outputs [0, 0, 1]. While in the construction presented so far $\mathcal{P}$ can only "stop" once it has gone through the entire document sequentially (so as to reach the EOD special character), Reef has the ability to skip many characters (possibly all the way to the end)—allowing the prover to stop without accessing the entire document. Third, breaking up the proof into small steps means that $\mathcal{P}$ can work on one step at a time, significantly reducing the amount of memory needed. Last, with recursive zkSNARKs like Nova [52], if $\mathcal{P}$ wants to prove the *same* step function many times (which is the case with `match_step`), there are significant performance gains.

5 Skipping Alternating Finite Automata

The use of recursion is a necessary first step in Reef, but it still falls short of our goals of expressiveness and efficiency.

In this section we introduce a new type of finite automaton called SAFA. The motivation for SAFA is twofold; avoid the state explosion problem for regex with lookarounds (§2.1) and capture the smallest set of characters within a document that must be checked in order to confirm that it matches a regex. We start by reviewing *Alternating Finite Automata* (AFA) which are a generalization of NFA. SAFA extend AFA to include the notion of *skips*.

5.1 Alternating Finite Automata (AFA)

AFA [27] are finite automata that generalize NFA by labeling states with an existential (∃) or a universal (∀) quantifier. An ∃ state is identical to a state in an NFA; the AFA merely reads the character at the current cursor, advances the cursor, and then transitions to any one of its possible next states. A ∀ state is very different. First, the AFA creates a *copy* of the remaining characters in the input string (starting at the current cursor until the end of the string) for *each* of its transitions (i.e., if there are 10 transitions it will create 10 copies of the input string). Then, in parallel, it transitions to every next state, and feeds each of those states their own independent copy of the input. For the AFA to accept an input string, all of the parallel branches need to end in accepting states. Intuitively, ∀ states capture the conjunction of multiple sub-automata, each of which operates independently on the provided input.

Formally, an AFA [27] is a 6-tuple $(Q,\Sigma,q_0,\lambda_q,\delta,F)$, where Q is the set of all states; Σ is the alphabet; $q_0 \in Q$ is the initial state; $\lambda_q : Q \rightarrow \{\forall, \exists\}$ is a labeling that assigns each state q either ∀ or ∃; $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation that defines final states with respect to initial states and input characters; $F \subseteq Q$ is the set of accepting states.

Example. Suppose we want to match documents of length between 2–6 that contain "a" and "b" defined over $\Sigma = \{a,b,c\}$. This is given by the regex $\mathcal{R}$ = "`^(?=.*a)(?=.*b).{2,6}$`". Representing $\mathcal{R}$ as an NFA requires creating an automaton that accepts the alternation of all strings that contain both "a" and "b" and have length between 2 to 6 ("`ab`", "`.ab`", "`a..b`", "`.a.b.`", etc.). The minimal NFA for this has 17 states (the 16 shown here [9] plus a sink state for all invalid characters). In contrast, one can match $\mathcal{R}$ with the 11-state AFA given in Figure 2.

To understand this AFA, first recall *epsilon transitions*, which AFA inherit from NFA and which mean that the automaton can take any transition with an ϵ label without advancing the cursor or reading any character from the document. Second, notice the state at the top is labeled ∀, which means that after processing the document, all of its transitions (the 3 vertical branches) should end in an accepting state. The transitions of ∀ states are special in that each creates a private copy of the cursor initialized to the value of the cursor when the ∀ state is reached. As a result, states 1, 3, and 5 will all have their own cursors (i.e., advancing the cursor of the left branch does not affect the cursor of the right branch).

Consider for example the document D = *acbcc* which is accepted since the three branches out of state 0 run in parallel and each branch terminates in an accepting state. If instead D = *bccbb*, the middle and right branches both terminate in accepting states, but the left branch does not.

The above example immediately shows that AFA could provide savings over the automata considered by prior works. Indeed, if a regex requires n states to be represented in an AFA, the same regex may require 2^{2^n} states in a DFA [31].

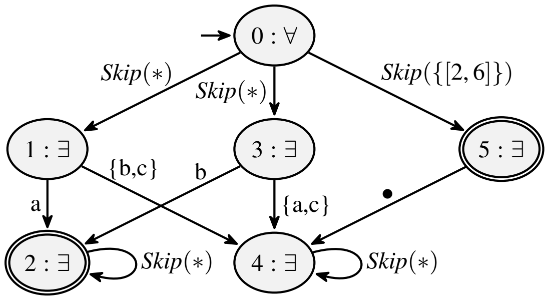


FIGURE 3—SAFA for regex $\mathcal{R}$ = `^(?=.*a)(?=.*b).{2,6}$` over alphabet $\Sigma = \{a,b,c\}$. $Skip(*)$ means the skip $\{[0,+\infty)\}$.

5.2 SAFA: Supporting Skips

AFA are a great way for Reef to increase the expressiveness of the supported regexes without incurring exponential costs, but AFA—just like DFAs and NFAs—are designed from the lens of "computation" rather than the lens of "verification". This fundamental distinction between compute and checking leaves a lot of opportunities unexplored.

As a concrete example, consider the regex $\mathcal{R}$ ="`.*ab$`" and the document D ="aaab". AFA (much like NFA) represent "`.*`" by a single, non-accepting state, with the option to loop or progress forward with an ϵ transition. Finding the solution to the question "is $D \in \mathcal{L}[\![\mathcal{R}]\!]$"? (meaning is D in the language defined by $\mathcal{R}$) requires computing both the case in which the first "a" in D matches the "`.*`" in $\mathcal{R}$ *and* the case in which it matches the "a" in $\mathcal{R}$. Confirming a match is simpler: given a path through the AFA for D, we just need to check that the path leads to an accepting state.

We can even take this concept further. When computing, bounded wildcard matching has to be explicitly unrolled. "`.{m,n}`", "`.{n}`", and "`.{n,}`" all require at least *n* transitions in an NFA or AFA. We see this in the right branch of the AFA in Figure 2 (states 4 through 10), where each state in "`.{2,6}`" has to be included explicitly.

But when checking, what if we could simply move the cursor forward by a number between 2–6 (inclusive), and carry on? Since "`.{2,6}`" is a wildcard, the content does not matter; what matters is that a *wildcard region* of the appropriate length exists. To express wildcard regions, we introduce *skips*. A skip is a finite set of non-overlaping intervals, $s = \{i_1,\ldots,i_n\}$, where each interval is of the form $i = [start, end]$ or $i = [start, \infty)$. Both *start* and *end* are non-negative integers and $start \leq end$; for $[start, \infty)$, the interval is unbounded on the right.

The idea is that when we reach a state that has a *skip transition* defined by some skip *s*, instead of reading a character from the input and transitioning to the next state based on the read value, the automaton advances the cursor by any amount within the intervals in *s*, and then moves to the next state.

Note that we need *s* to be a set rather than a single interval because of regexes such as "`(.{2,6}|.{8,10})a`" that have multiple acceptable disjoint wildcard regions. Also, observe that skips generalize epsilon transitions: we can simply define skip $\epsilon = \{[0,0]\}$. Third, we can support Kleene-star wildcard regions with $Skip(*) = \{[0,\infty)\}$, meaning any cursor less-than the length of the document works. The use of the ∞ symbol allows the separation of SAFA from the document to which it is applied.

SAFA. With the above notion of skips we can then define*Skipping Alternating Finite Automaton (SAFA)* as the 8-tuple $(Q, E, \Sigma, q_0, \lambda_q, \lambda_e, \delta, \mathcal{F})$, where Q is the set of all states (nodes); E is the set of all transitions (edges); Σ is the alphabet; $q_0 \in Q$ is the initial state; $\lambda_q : Q \to \{\forall, \exists\}$ defines the label for each node q to be either $\forall$ or $\exists$; $\lambda_e : E \to Skip \uplus \Sigma$ sets the label for each edge e as either a skip s or $\alpha \in \Sigma$; $\delta \subseteq Q \times E \times Q$ is the transition relation; and $\mathcal{F} \subseteq Q$ is the set of accepting states. The symbol $\uplus$ is the set *disjoint union*.

Much of this definition should look similar to the AFA in Section 5.1. The only difference is the addition of two new fields: E and λ_e. E is simply the set of all transitions. λ_e can be thought of as an analog of λ_q, but over transitions instead of states. It labels each transition $e \in E$ as taking a single step via a character (as is the case in AFA and NFA), or as a skip, which does not consume any characters from the document but increases the cursor non-deterministically by some amount in *s*.

Example. We defer the formal definition of skips and the various transitions to our tech report [15]. In Figure 3 we show the SAFA that corresponds to the AFA from Figure 2. The SAFA replaces the long chain of states (4–10) in the AFA with $Skip\{[2,6]\}$. This compression is possible because ϵ (the identity element) followed by skip *s* is just *s*.

The examples in Figures 2 and 3 provide the intuition for why SAFA might be cheaper to represent in an NP checker than AFA, while also being computationally equivalent (though SAFA requires the automaton to "know" how much to skip ahead of time). We formalize the equivalence between SAFA and AFA by direct translation.

Theorem 5.1. *Let $\mathcal{S}$ denote a SAFA. There exists an AFA $\mathcal{A}$ such that the language $\mathcal{L}[\![\mathcal{S}]\!] = \mathcal{L}[\![\mathcal{A}]\!]$ is regular.*

The proof is in our tech report [15].

5.3 Designing the SAFA `match_step` Function

Section 4 introduces a `match_step` function that is appropriate for recursive proof systems. Reef modifies this step function to support SAFA. Reef's `match_step` takes in two additional arguments: `cursor_move`, which is the quantity by which $\mathcal{P}$ plans on advancing the cursor in the next transition, and a *stack*. One can represent a stack very cheaply with a simple hash chain (a single field element). A new stack is simply the value $stack = 0$. To push a value *val* just append it to the hash chain $stack = H(stack||val)$. To pop a value from *stack*, $\mathcal{P}$ must supply a preimage of *stack*; the first part of the pre-image will be the new stack, the other part is the popped value. That said, in our specific setting we can implement an

```
field[4] match_step(field[] commit, field[] blind,
    field state, field cursor, field cursor_move,
    field stack) {

  field cur_char = open_at(commit, blind, cursor);

  if (cur_char == EOD) { // end of the document
    if !is_accept(state) {
      return {0, 0, 0, 0}; // no match
    }

    if (is_empty(stack)) {
      return {0, 0, 0, 1}; // match
    } else {
      // reached accepting state in one branch
      // but there are other branches.
      // process next branch
      stack, (state, cursor) = pop(stack);
      return {state, cursor, stack, 0};
    }
  }
  // special handling for forall state
  if (is_forall(state)) {
    for child in children(state) {
      stack = push(stack, (child, cursor));
    }
    stack, (state, cursor) = pop(stack);
    return {state, cursor, stack, 0};
  }
  // perform character or skip transition
  state, cursor = delta(state, cursor, cur_char,
    cursor_move);
  return {state, cursor, stack, 0};
}
```

FIGURE 4—Reef's step function using SAFA.

even more efficient version since we know ahead of time the maximum depth of the stack (which depends on the number of nested `forall` states and the number of transitions). The details are provided in our tech report [15].

Reef's `match_step` function is given in Figure 4. A key attribute for SAFA is that for a document D to be considered a match for regex $\mathcal{R}$, all children of a `forall` state must reach accepting states. Additionally, all of these children must start from the same cursor position, which is private. In Reef's `match_step` function, when a `forall` node is reached a copy of the cursor and the state ID is pushed onto the stack for each of the node's children. When one of the child branches terminates in an accepting state, its sibling and the original cursor position are popped from the stack.

Reef's `delta` function is then:

```
field[2] delta(field state, field cursor,
    field cur_char, field cursor_move) {
  field state, min, max = lookup(state, cur_char);
  assert(min <= cursor_move <= max);
  assert(cursor <= cursor + cursor_move);
  cursor = cursor + cursor_move;
  return {state, cursor};
}
```

Reef relies on *lookup tables* for determining whether a transition is valid. This is discussed more in-depth in Section 6, but in the context of our `delta` function, they work as follows: given a current state, character, and proposed quantity by which to move the cursor, we use a lookup table to validate the next state, as well as the minimum and maximum quantity the cursor is allowed to move, based on the type of skip. For example, if the transition is a skip "`{[n,m]}`", then `min`$= n$ and `max`$= m$. If the transition is `Skip(*)`, then `min`$= 0$ and `max`$= |\mathbb{F}| - 1$. In addition, we check that the new cursor position is greater than or equal to the current cursor position (i.e., that the prover did not decrement the cursor through an arithmetic overflow). In all other cases `max`$=$`min`$= 1$.

6 SAFA and Document Lookup Tables

Reef uses two lookup tables. One lookup table is public and represents the SAFA character and skip transitions; $\mathcal{V}$ can derive this public table from the regex. The other lookup table represents the document and is *private* (i.e., its contents cannot be revealed to the verifier). In each invocation of Reef's `match_step` (§5.3), the document table is accessed to read the character at the current cursor, and then the transition table is accessed to determine the next state.

This section reviews lookup arguments (§6.1), how Reef organizes the SAFA transitions table (§6.2); how it commits to the private table representing the document (§6.3); how it supports *table projections* that help filter which entries in the private table are relevant to a particular regex (§6.4); and how it combines both the public and private tables into one *hybrid* table that reduces the fixed costs of the lookup argument and improves its amortization (§6.5).

6.1 Lookup arguments

There are cases where one would want to check that a value v in an R1CS instance is contained in some table T of size n. A way to do this when T is public is to "hardcode" T in the R1CS instance by expressing it as a cascade of `if` statements similar to how we arithmetized the DFA's δ function (§3.1). Then, we check that v matches one of these `if` statements and not the final `return`. This requires $\mathcal{O}(|T|)$ constraints per lookup. An asymptotically cheaper (but sometimes concretely more expensive) solution is to use a Merkle Tree where the leaves represent T. One passes the root of the tree as a public input and a secret Merkle proof; the R1CS instance computes $\log(n)$ hashes to confirm there is a path to the root given v.

Lookup arguments [30, 34, 51, 71] generalize this idea: given m values $\{v_0, \ldots, v_{m-1}\}$ each in $\mathbb{F}$, lookup arguments check that all m values are entries in a table $T \in \mathbb{F}^n$. Crucially, lookup arguments amortize the costs over the m checks such that as m increases, the per-lookup cost decreases.

`nlookup` [51]. We briefly describe `nlookup`, which is designed for recursive proof systems such as the one we use (§4). For now, assume that the table is public. Section 6.3 describes additional techniques to handle private tables.

Let T be a table with $n = 2^\ell$ elements and let $\widetilde{T}$ be a multilinear polynomial in ℓ variables such that for all $i \in \{0,1\}^\ell$,

$\widetilde{T}(i) = T[\text{to-int}(i)]$, where to-int : $\{0,1\}^\ell \to \{0,1,\ldots,n-1\}$ is a function that maps ℓ-sized bit strings to ℓ-bit integers in a natural manner. Given $\widetilde{T}$, one can then prove that a value $v \in T$ by producing a point $q \in \{0,1\}^\ell$ such that $\widetilde{T}(q) = v$. `nlookup`'s core idea is to reduce the task of checking m of these lookup proofs to evaluating $\widetilde{T}$ at a single point. To do this, the `nlookup` prover proves:

$$\sum_{i=1\ldots m} \rho^i \cdot v_i = \sum_{i=1\ldots m} \rho^i \cdot \sum_{j\in\{0,1\}^\ell} \widetilde{eq}(q_i,j) \cdot \widetilde{T}(j)$$

where $v_i \in \mathbb{F}$ is the i-th value claimed by the prover to be in T, $\rho \in \mathbb{F}$ is a random challenge chosen by the `nlookup` verifier, and $\widetilde{eq}$ is a designated multilinear polynomial for performing Boolean equality checks. This equality can be proven using the sum-check protocol [55].

On its own, this is sufficient for proving membership of a set of elements in T. However, `nlookup` is particularly beneficial in the case where we would want to look up m elements *multiple times* (e.g., during different iterations of the step function of a recursive proof system). Readers familiar with the sum-check protocol can recall that in the above description, the verifier has to evaluate $\widetilde{T}$ at a random point at the end of the sum-check protocol.

In the case where we want to lookup m elements, k separate times, `nlookup` leverages a folding scheme to fold all k evaluations of $\widetilde{T}$ into a single one. It does this by initializing a running claim $v_r = \widetilde{T}(q_r)$ where $q_r, v_r \in \mathbb{F}^\ell$, and q_r is chosen arbitrarily. To incorporate new lookup claims (i.e., polynomial evaluations) into this running claim, `nlookup` makes a slight modification to the polynomial above. In particular, the sum-check protocol is now run over the polynomial:

$$v_r + \sum_{i=1\ldots m} \rho^i \cdot v_i =$$
$$\sum_{j\in\{0,1\}^\ell} \widetilde{eq}(q_r,j) \cdot \widetilde{T}(j) + \sum_{i=1\ldots m} \rho^i \cdot \sum_{j\in\{0,1\}^\ell} \widetilde{eq}(q_i,j) \cdot \widetilde{T}(j)$$

which incorporates the running claim over foldings.

Integrating `nlookup` into Nova. To use `nlookup` with Nova, we encode `nlookup`'s verifier as an R1CS NP checker. This involves implementing the sum-check verifier and the associated Fiat-Shamir transform involving hash computations as R1CS. We then invoke this NP checker inside Reef's step function whenever we want to enforce that a group of R1CS variables are set to values contained in a table.

The cost to represent the above NP checker is as follows. To look up m entries in a table of size n within a step function, the number of constraints depends on the above two components: (1) sumcheck verifier and variable assignment, which requires $\mathcal{O}(m \cdot \log n)$ constraints with small constant; and (2) Fiat-Shamir transform which requires representing $\mathcal{O}(\log n)$ hash function evaluations in constraints, and each hash function requires hundreds of constraints.

Since expressing the hash functions is the dominant cost, lookup arguments are designed to amortize this component over the batch of m lookups. This is in contrast to using Merkle Trees which requires $\mathcal{O}(m \log n)$ hash functions represented as constraints to handle m lookups.

Since the `nlookup` verifier is encoded as an NP checker in R1CS, the Nova prover actually needs to know the witness for this checker so that it can prove the satisfiability of the statement. To compute this witness, the Nova prover has to do $\mathcal{O}(n)$ finite field operations per series of m lookups. Also, outside of R1CS (after the Nova verifier has checked the proof), the verifier performs an additional $\mathcal{O}(n)$ finite field operations at the very end of the protocol. A more detailed explanation of the protocol can be found in [51, §7] and in our tech report [15].

6.2 SAFA Lookup table

The lookup table T that Reef uses to encode the SAFA has a row for each transition in the SAFA and 5 columns—current state, character, next state, minimum cursor move, and maximum cursor move. The function of each of these columns is covered in Section 5.3. To convert this into the multilinear polynomial $\widetilde{T}$ needed for `nlookup` we manifest T as a vector of elements; each element represents an entire row and is computed by hashing the corresponding 5 columns to produce a value in $\mathbb{F}$. After a lookup takes place in Reef's step function, the result is therefore a single hash digest. To obtain the columns, the step function has constraints that allow the prover to supply the five values of the column entries, followed by a check that confirms that the hash of these values matches the looked up digest.

Constraints for SAFA lookups. As we discuss in Section 6.1, the number of constraints required for m lookups in a table of size n using `nlookup` is $\mathcal{O}(m \cdot \log n)$ constraints plus $\mathcal{O}(\log n)$ hash functions expressed as constraints. Each of the m lookups represents one SAFA transition. The SAFA table is of size $\mathcal{O}(|Q_{SAFA}| \cdot |\Sigma|)$ in the worst case—a transition for every character from every state. If the step function processes one SAFA transition at a time then $m = 1$ and the number of constraints to represent the single lookup is $\mathcal{O}(\log(|Q_{SAFA}| \cdot |\Sigma|))$ plus $\mathcal{O}(\log(|Q_{SAFA}| \cdot |\Sigma|))$ hashes.

Constraints across all steps. While it may seem that the total number of transitions (and therefore steps) should be at most $\mathcal{O}(|\mathsf{D}|)$, that is not always the case. With no lookarounds, the total number of transitions is $\leq |\mathsf{D}|$. However, because SAFA may have multiple branches for lookarounds, certain parts of D may be looked up more than once. We thus upper bound the number of transitions by $\alpha = \mathcal{O}(|\mathsf{D}| \cdot L)$, where L is the number of lookarounds in the regex. The number of constraints needed to check all of the transitions is thus $\mathcal{O}(\alpha \log(|Q_{SAFA}| \cdot |\Sigma|))$ plus $\mathcal{O}(\alpha \log(|Q_{SAFA} \cdot |\Sigma|))$ hashes.

Of course, the whole point of using a lookup argument is to benefit from its amortization, which is why Reef places multiple SAFA transitions within a single step function based on the results of our optimizing compiler (§7). As a result, $m \geq 1$, so each step function has m transitions but Reef needs to run m times fewer steps. In this case, the total number of constraints across all steps is $\mathcal{O}(\alpha \log(|Q_{SAFA}| \cdot |\Sigma|))$ plus $\mathcal{O}(\frac{\alpha}{m} \log(|Q_{SAFA} \cdot |\Sigma|))$ hash functions. One might think that the optimal case is to have all lookups in a single step (which maximizes the amortization), but this is not so because there are other considerations as we explain in our tech report [15].

6.3 Committing to a document

To commit to a document D over an alphabet Σ, the committer $\mathcal{G}$ first maps each character in Σ to an element in $\mathbb{F}$. Then, $\mathcal{G}$ simply treats D as a vector in $\mathbb{F}^n$. At this point, $\mathcal{G}$ can commit to D using any vector or polynomial commitment [23, 54, 62, 76]. That said, we choose a polynomial commitment since Reef uses a lookup argument to access SAFA transitions anyway, so using a lookup argument to access D allows us to combine both lookup tables to get lower costs (§6.5).

Note that if the optional transparency goal is desired, then the commitment scheme must be transparent (§3).

Polynomial commitment. $\mathcal{G}$ treats the vector D as a multilinear polynomial $\widetilde{T}$ in evaluation form and commits to $\widetilde{T}$ with a polynomial commitment. A polynomial commitment is a tuple of algorithms (*Setup*, *Commit*, *ProveEval*, *VerifyEval*). Informally, *Setup* outputs public parameters *pp*; *Commit* takes *pp*, a polynomial $\widetilde{T}$, and outputs a hiding and binding commitment to $\widetilde{T}$, $C_{\widetilde{T}}$; *ProveEval* takes *pp*, $\widetilde{T}$, a point q, value v, and outputs a proof π_{poly} that $\widetilde{T}(q) = v$; *VerifyEval* takes *pp*, $C_{\widetilde{T}}$, q, π_{poly}, and v and outputs whether $\widetilde{T}(q) = v$.

In our implementation we use the Hyrax polynomial commitment (Hyrax-PC) [76, §6.1], but one could make other choices to get different tradeoffs (e.g., Dory [54] has smaller commitments but its *ProveEval* algorithm results in larger proofs and is more expensive).

Let the Pedersen commitment for a vector $x \in \mathbb{F}^n$ be:

$$Pedersen(x, b) = h^b \cdot \prod_{i=1}^{n} g_i^{x_i}$$

where $g_1, \ldots, g_n$ and h are public random generators of the group over which the zkSNARK is defined (Pallas elliptic curve [42] in our case) and $b \in \mathbb{F}$ is a secret random blind. Hyrax-PC treats T as the column-major order of a $\sqrt{n}$-by-$\sqrt{n}$ matrix M, and commits to each row of M using a Perdersen vector commitment. This means that the commitment in Reef is $\sqrt{n}$ group elements, and there are $\sqrt{n}$ random blinds.

Making `nlookup` zero-knowledge. `nlookup` [51] does not explicitly discuss a way to guarantee zero-knowledge during lookups. Here we give a concrete proposal, based on standard techniques [29, 66, 76]. As we describe in Section 6.1, the output of the recursive proof system will include an `nlookup` running value v_r purported to be the evaluation of the multilinear polynomial $\widetilde{T}$ at a public random point $q_r \in \mathbb{F}$ specified by the Fiat-Shamir transform. When T is public, $\mathcal{V}$ can simply compute $\widetilde{T}(q_r)$ and check if it equals v_r. This is what we do with the SAFA table (§6.2). However, when T is private, there are two issues: (1) $\mathcal{P}$ cannot give $\mathcal{V}$ the claim v_r in the clear, as v_r is a weighted sum of the contents of T and would leak information; and (2) $\mathcal{V}$ does not have access to T and hence cannot compute $\widetilde{T}(q_r)$ on its own.

We address these issues as follows. First, instead of outputting v_r in the clear, we have the `match_step` function output d, where $d = H(v_r||s_1)$ and s_1 is a random secret value that $\mathcal{P}$ chooses. $\mathcal{P}$ can make d available to $\mathcal{V}$ without revealing anything about v assuming H heuristically instantiates a random oracle. $\mathcal{P}$ then computes another proof, $\pi_{consistency}$, with a separate non-recursive zkSNARK (we use Spartan [66]) for the statement: "given commitment c and public input d, I know a v_r such that $d = H(v_r||s_1)$ and $c = g^{v_r}h^{s_2}$ for some s_1 and s_2", where g and h are appropriate generators of the polynomial commitment. In effect, $\pi_{consistency}$ establishes that $\mathcal{P}$ correctly transformed one type of commitment (d) that is cheap to compute in R1CS but is not useful to verify polynomial evaluations, into another type of commitment (c) for the same value v_r that can be used to verify polynomial evaluations. Furthermore, $\pi_{consistency}$ is very cheap to compute ($\approx$ 300 constraints) as we make c an *outer commitment* [28] (i.e., a commitment that is native to the underlying proof system) and does not need to be expressed in R1CS at all.

Second, recall that $\mathcal{V}$ has access to a polynomial commitment of $\widetilde{T}$, $C_{\widetilde{T}}$. $\mathcal{P}$ can then give $\mathcal{V}$ a proof $\pi_{poly} = ProveEval(\widetilde{T}, q_r, v_r)$, which $\mathcal{V}$ can use alongside q_r, c, and $C_{\widetilde{T}}$ to confirm that $\widetilde{T}(q_r) = v_r$. The key idea is to realize that, in Hyrax [76] and similar polynomial commitments [23, 54], the first step of $VerifyEval(C_{\widetilde{T}}, q_r, \pi_{poly}, v_r)$ is for $\mathcal{V}$ to turn the claim v_r into the Pedersen commitment $g^{v_r}h^{s_3}$ for some s_3. However, $\mathcal{V}$ already has $c = g^{v_r}h^{s_2}$ and a proof $\pi_{consistency}$ that establishes that c is a valid Pedersen commitment for v_r. Hence, $\mathcal{V}$ can simply use c instead.

Security. Observe that the verifier sees d, c, $C_{\widetilde{T}}$, q_r, $\pi_{consistency}$ and π_{poly}. From this information, the verifier learns nothing about v_r beyond the fact that d and c commit to the same value, and that c is a commitment to a correct evaluation of a polynomial underneath the commitment $C_{\widetilde{T}}$ at point q_r. This is because $\pi_{consistency}$ and π_{poly} are both zero-knowledge arguments, and the three commitments d, c, and $C_{\widetilde{T}}$ are hiding.

6.4 Table projections

For proving m lookups over a committed document of size n, `nlookup`'s prover incurs $\mathcal{O}(n)$ operations over $\mathbb{F}$. Although these are not expensive group operations, when n is large (e.g., billions), this can be expensive. On the other hand, in some applications, it is public information that lookups will be made to particular portion of the document (though the actual content within that portion of the document is private). For example, a study may just care about DNA regions that start at publicly known offsets.

To address this, we describe an approach to run `nlookup` on a *projected* table (one that contains one or more "chunks" of an original table) such that the prover incurs costs proportional to the size of the projected table. Furthermore, the verifier still only needs a commitment to the original table. The core idea is to leverage certain basic facts about multilinear polynomials to reduce claims about a projected table to claims about the original table.

We begin with an overview, which we then generalize. Let T be the original table with $n = 2^\ell$ elements, and $\widetilde{T}$ be its multilinear extension as described in Section 6.1. Suppose we project T into a smaller table T'; $\widetilde{T'}$ is then a multilinear polynomial in $\ell' < \ell$ variables. It turns out that $\widetilde{T'}$ and $\widetilde{T}$ are related in a fundamental way. This is what enables us to run `nlookup` on T'. At the end of `nlookup`, the verifier is left with a claim about T', of the form $\widetilde{T'}(q_r) = v_r$. However, the verifier only has a commitment to the original table T. To address this, we transform this claim to an equivalent claim about an evaluation of $\widetilde{T}$, allowing the verifier to check the claim about $\widetilde{T}$ using a commitment to T. We now elaborate.

We use a concrete example, to provide intuition. Suppose that $T = [a,b,c,d,e,f,g,h]$, so $\widetilde{T}$ is a multilinear polynomial in $\ell = 3$ variables. Suppose the projected table is $T' = [c,d]$, so $\ell' = 1$. For this example, it follows that for all $q_r \in \mathbb{F}^{\ell'}$ $\widetilde{T'}(q_r) = \widetilde{T}(s,q_r)$, where $s = 01 \in \{0,1\}^2 = \{0,1\}^{\ell-\ell'}$. In the context of `nlookup`, to check that $\widetilde{T'}(q_r) = v_r$, the verifier can instead check $\widetilde{T}(s,q_r) = v_r$, where $s = 01$. A key take-away here is that for $0 \leq \ell' \leq \ell$, observe that a specified prefix $s \in \{0,1\}^{\ell-\ell'}$ "selects" a unique chunk of T and specifies a particular projection of size $2^{\ell'}$.

Note that this approach generalizes to project non-contiguous chunks of T. For simplicity, suppose that we want to project two chunks of T, specified with two selectors $s_1 \in \{0,1\}^{\ell'}$ and $s_2 \in \{0,1\}^{\ell'}$, where $0 \leq \ell' \leq \ell$. The projected table $T' = (L,R)$ is a vector of size $2^{\ell-\ell'+1}$ and L and R are vectors of size $2^{\ell-\ell'}$, so $\widetilde{T'}$ is a multilinear polynomial in $\ell-\ell'+1$ variables. When we run `nlookup` with the projected table T', the verifier ends up with a claim about the projected table of the form $\widetilde{T'}(q_r) = v_r$, where $q_r \in \mathbb{F}^{\ell-\ell'+1}$. Again, derived from the properties of multilinear polynomials,

$$\begin{aligned}\widetilde{T'}(q_r) &= (1-q_r[0]) \cdot \widetilde{L}(q_r[1..]) + q_r[0] \cdot \widetilde{R}(q_r[1..]) \\ &= (1-q_r[0]) \cdot \widetilde{T}(s_1, q_r[1..]) + q_r[0] \cdot \widetilde{T}(s_2, q_r[1..])\end{aligned}$$

Thus to check if $\widetilde{T'}(q_r) = v_r$, the verifier can instead check if $(1-q_r[0]) \cdot \widetilde{T}(s_1,q_r[1..]) + q_r[0] \cdot \widetilde{T}(s_2,q_r[1..]) = v_r$, which makes two evaluation queries to $\widetilde{T}$. Note that this idea generalizes to projecting $k > 2$ non-contiguous chunks of T.

Low-cost padding to hide document size. In many settings, one would like to hide not just the content of D, but also its size. For example, if D is a password, revealing its size reveals the password's length. Projections allow the commitment generator $\mathcal{G}$ to pad the document to some upper bound (essentially for free) while allowing $\mathcal{P}$ to perform operations proportional to the unpadded document and without having to reveal the selector s to $\mathcal{V}$. Our tech report [15] has the details.

6.5 Hybrid private/public lookup argument

Reef's step function (§5.3) looks up values from two tables: the public SAFA table (S) and the private document table (D). We can do this with two separate instances of `nlookup`, one for each table. However, this requires $m\log(|\mathsf{D}| \cdot |S|) + \mathcal{O}_H(\log(|\mathsf{D}| \cdot |S|))$ constraints where m is the number of lookups to each table per step.

Instead, we combine both tables into a single *hybrid* table, all while preserving the privacy requirements of the document table. Accessing this hybrid table requires only $2m\log(|D| + |S|) + \mathcal{O}_H(\log(|\mathsf{D}| + |S|))$ constraints. This optimization does not pay off only when one of the tables is multiple orders of magnitude larger than the other. But we never encountered an imbalance between $|\mathsf{D}|$ and $|S|$ large enough to nullify the benefits in any of our experiments.

$\mathcal{P}$ has access to S and D and can merge the tables by pretending they are two halves of a large table T and running the `nlookup` prover. At the end, $\mathcal{V}$ will end up with a single claim about the multilinear extension of T: $\widetilde{T}(q_r) = v_r$, where $q_r \in \mathbb{F}^\ell$ and $\ell = \log(2 \cdot \max(|\mathsf{D}|, |S|))$. Since T in this case includes private data, $\mathcal{V}$ should not see v_r in the clear, and instead receives: $d = H(v_r||s_1)$, C_{v_r} (a Pedersen commitment to v_r), and a proof $\pi_{consistency}$ as we discuss in Section 6.3.

To verify $\widetilde{T}(q_r) = v_r$, $\mathcal{V}$ must treat the public and private parts of the large table as separate "indexable" chunks, similar to the way projections work. We define $\widetilde{T}(q_r)$ as:

$$\widetilde{T}(q_r) = (1-q_r[0]) \cdot \widetilde{S}(q_r[1..]) + q_r[0] \cdot \widetilde{\mathsf{D}}(q_r[1..]) = v_r$$

Notice that this means we need to arrange T such that it can be divided equally into a public half (indexed by $q_r[0] = 0$) and a private half ($q_r[0] = 1$). The smaller of the two tables will be padded to the size of the other, which is why $\ell = \log(2 \cdot \max(|\mathsf{D}|, |S|))$ above, and why the hybrid table becomes inefficient if one table is extremely larger than the other. Lookups to the public half of the table use exactly the same indices as before. Lookups to the private half will use the same indices as before added to $2 \cdot \max(|D|, |S|)$.

Given this structure, $\mathcal{P}$ evaluates $\widetilde{D}$ at the point $q_r[1..]$ and obtains a value $v_d \in \mathbb{F}$. $\mathcal{P}$ then generates a commitment C_{v_d} to v_d, and a proof $\pi_{poly} = ProveEval(\widetilde{D}, q_r[1..], v_d)$ that establishes that $\widetilde{D}(q_r[1..]) = v_d$. For its part, $\mathcal{V}$ computes $\widetilde{S}(q_r[1..]) = v_s$ on its own, and runs *VerifyEval* on π_{poly} using the document commitment, $C_{\widetilde{D}}$, and C_{v_d}.

So far, we have proceeded very similarly to the verification of the running claim in the non-hybrid model. But notice that $\mathcal{V}$ must still relate v_s and C_{v_d} to C_{v_r} in the following way:

$$(1-q_r[0]) \cdot v_s + q_r[0] \cdot v_d = v_r$$

This is done as follows. $\mathcal{V}$ computes C_L, which is a Pedersen commitment to the value on the left-hand-side of the above equation using v_s and C_{v_d} (this requires only linear operations on Pedersen commitments, which are linearly homomorphic). $\mathcal{P}$ then proves that C_L and C_{v_r} commit to the same value using a Schnorr [65] zero-knowledge proof of equality π_{eq}.

Security. When the verifier computes the commitment C_L, it does not learn any additional information about v_d as the operations are done using C_{v_d} (C_{v_d} is a commitment that hides the underlying value v_d). Furthermore, π_{eq} proves that the values under the commitments C_L and C_{v_r} are the same without revealing any additional information.

7 Implementation

Reef is implemented in 14K lines of Rust and is open source [8]. We discuss the main components here and optimizations in our tech report [15].

7.1 Compilation: from regex to R1CS

Reef has two levels of compilation. First, Reef compiles regexes written in standard PCRE syntax [7] (Figure 1) and produces a SAFA. From this SAFA, Reef generates the SAFA's transition lookup table and the `match_step` function discussed in Section 5.3. Since the `match_step` function uses lookups it also contains the checks that the `nlookup` verifier [51] must perform in each step. In particular, it contains a series of Fiat-Shamir challenges that we generate with the Poseidon hash function [39] using the Neptune library [3]. Finally, Reef uses the CirC [61] compiler to output R1CS instances that we convert to Bellman [1] instances.

7.2 Solving: finding the satisfying witness

Reef, given a document D, finds the witness to the R1CS instance representing `match_step` in two parts. First, Reef derives which paths in the SAFA to take, the skip values, the entries in D to read, and the rows in the transition table to look up. Reef's solver might be of independent interest and we discuss it in our tech report [15]. This solver only needs to run once and tells $\mathcal{P}$ how many steps to prove.

Second, for each step, Reef runs the `nlookup` prover, which we implement as there was no prior implementation, to generate the values that will satisfy the `nlookup` checks that were inserted in the corresponding `match_step`. The result of this and the SAFA solver are sufficient to construct the entire solution vector $z_i = (y_i, 1, w_i)$ where w_i is the witness and y_i is the output of step i.

7.3 Proving knowledge of the witness

For the proving and verifying, we use Nova [4], which we modify to make it zero-knowledge (the existing implementation was only succinct). This required changing 1.6K lines of Rust to hide the number of steps executed, and making the commitments hiding, and the folding scheme, sumcheck protocol, inner product argument, and SNARK zero-knowledge. Our modified version of Nova is open source [5].

8 Costs and Complexity analysis

In this section we discuss the asymptotic costs of all of the components of Reef. The analysis below considers the case where Nova [52] uses Pedersen commitments to commit to vectors, and Spartan [66] uses an IPA-based polynomial commitment scheme to compress incrementally generated proofs. Furthermore, for `nlookup` [51], the analysis considers the case where documents are committed with Hyrax's polynomial commitment scheme [76]. Finally, one of the basic operations of the above proof systems are *multiexponentiations*: given generators $g_1, \ldots, g_n$, and exponents $e_1, \ldots, e_n$, compute $g_1^{e_1} \cdot g_2^{e_2} \cdots g_n^{e_n}$. These are also called *multi-scalar multiplications* (MSM). These proof systems typically use Pippenger's algorithm [63] which can compute a size-n MSM in $\mathcal{O}(n\lambda/\log(n\lambda))$ group operations. We will ignore the security parameter λ and just treat a size-n MSM as $\mathcal{O}(n/\log n)$ group operations.

For simplicity, let $T = |\mathsf{D}| + |Q_{SAFA}|$ be the sum of the size of both the document and the SAFA lookup tables.

Committer's costs. Committing to a document D with Hyrax's polynomial commitment [76] requires the committer $\mathcal{G}$ to perform $\mathcal{O}(|\mathsf{D}|/\log\sqrt{|\mathsf{D}|})$ group operations.

Prover's costs. Ignoring the distinction between the arithmetization of hash functions and other operations, the contribution of the lookup argument towards Reef's step function is $\mathcal{O}(m\log T)$ R1CS constraints; Reef requires a total of $\mathcal{O}(\alpha/m)$ steps to finish processing a document. Nova performs $\mathcal{O}(m\log(T)/\log(m\log T))$ group operations per step. This results in $\mathcal{P}$ performing a total of $\mathcal{O}(\alpha\log(T)/\log(m\log T))$ group operations. The resulting proof π is of size $\mathcal{O}(\log(m\log T))$.

In addition, during each step, Reef needs to run the `nlookup` prover in order to generate the relevant portion of the satisfying witness for the R1CS instance. This requires computing the sumcheck protocol over the hybrid table, which necessitates $\mathcal{O}(T)$ field operations. If projections are used, then D is substituted with D_{proj} in the definition of T.

At the end of the protocol $\mathcal{P}$ needs to compute *ProveEval* in order to generate π_{poly} so that $\mathcal{V}$ can verify the private component of the hybrid table. This requires $\mathcal{P}$ to perform $\mathcal{O}(\sqrt{|\mathsf{D}|}/\log\sqrt{|\mathsf{D}|})$ group operations. The proof, π_{poly}, is of size $\mathcal{O}(\log|\mathsf{D}|)$.

Finally, our zero-knowledge extension to the lookup argument for D requires generating the proof $\pi_{consistency}$, which is done with a constant-size R1CS instance, and therefore $\mathcal{O}(1)$ group operations in Spartan [66].

$\mathcal{P}$ performs $\mathcal{O}(\alpha\log(T)/\log(m\log T) + \sqrt{|\mathsf{D}|}/\log\sqrt{|\mathsf{D}|})$ group and $\mathcal{O}(T)$ finite field operations in total.

Verifier's costs. The cost to the verifier $\mathcal{V}$ is $\mathcal{O}(m\log(T)/\log(m\log T))$ group operations in Nova to verify π. Further, $\mathcal{V}$ must invoke Hyrax's *VerifyEval* to check π_{poly}, which requires $\mathcal{O}(\sqrt{|\mathsf{D}|}/\log\sqrt{|\mathsf{D}|})$ group

Application	Document Size	SAFA States	SAFA Transitions
Redactions			
Small Email	415	331	42,318
Large Email	1,000	908	116,751
ODoH	128	36	4,012
Passwords			
Match	12	21	1,188
Non-Match	9	21	1,188
DNA			
Match	32.3×10^6	976	4,861
Non-Match	32.3×10^6	976	4,861

FIGURE 5—Document and SAFA size for evaluated applications

operations. Lastly, the verifier needs to evaluate the public component of the hybrid table which requires $\mathcal{O}(|Q_{SAFA}|)$ finite field operations.

$\mathcal{V}$ performs $\mathcal{O}(m\log(T)/\log(m\log T)+\sqrt{|\mathsf{D}|}/\log\sqrt{|\mathsf{D}|})$ group and $\mathcal{O}(|Q_{SAFA}|)$ finite field operations in total.

9 Evaluation

This section answers Reef's motivating questions: is proving general regular expression matching in zero knowledge practical for various applications and do Reef's optimizations meaningfully reduce the costs? Our results indicate that this is indeed the case.

9.1 Experimental Setup

Reef runs fine on a laptop (Intel Core i7 1.9 GHz, 16GB RAM) since its use of recursion means that $\mathcal{P}$ proves one step at a time and therefore uses little memory; at most 5.1 GB in our largest experiment. However, in order to run the baselines which require more memory, we run all of our experiments (including Reef) on a 16-core Intel Xeon Platinum 8253 CPU (2.20GHz) with 764 GB of RAM. We evaluate Reef over the applications discussed in Section 1: proving password strength, disclosing redacted emails, ODoH blocklisting, and genetic proving. For each of our use cases we evaluate documents and regexes of varying sizes. Figures 5 and 6 report the document sizes, SAFA sizes, and results for the largest instances based on SAFA size. However, full results, all document sizes, and a list of all regexes can be found in our tech report [15].

9.2 Overall Performance

We start by showing the end-to-end results of Reef on our applications, averaged over 10 runs, and then later break down some of these costs to show the benefits of each of Reef's optimizations.

Compilation. Compiling a regex to R1CS is the most time consuming part since it requires parsing the regex and generating the SAFA, lookup tables, and R1CS matrices. This includes the generation of the document commitment. However, this is typically a one-time cost and can be done in advance since the regex is public.

Solving (witness generation). Reef's witness generation includes the time to find the regex match, the right values for all the skips in SAFA, running the `nlookup` prover (whose output becomes a witness value to the step function), and finding the satisfying assignment to all R1CS variables. In most cases, all of this can be done in a few milliseconds; the exceptions are large documents (e.g., DNA or large emails) which require considerable time.

Proving. Proving time depends on document length, the regex complexity, how many steps the prover needs to run, and the size of each step. It includes the time to generate all the proofs, including the consistency and equality proofs of the hybrid table (§6.5). In our tech report [15] we discuss how Reef often batches many character and skip transitions into one step (leading to a larger step function but fewer total steps). Reef generally performs worse on regexes where the regex is similar to the document, as it gives Reef's prover fewer opportunities to skip and stop early. For example, the email redaction regexes are very similar to the original document, and hence result in more proving steps than some of the other regexes, and consequently larger proving time.

Reef's benefits are best exemplified with the DNA matching application, in which the document has over 32 million characters. Reef is able to generate succinct proofs for DNA in under 30 seconds (including both solving and proving) because it can avoid processing most of the document, thanks to its use of skips and projections.

Verification. The verifier's costs depend on the number of R1CS constraints for a single step (since Nova folds all steps into one), as well as the cost to evaluate the SAFA polynomial at a random point, and check the consistency polynomial evaluation, and the equality proof. Nova's current implementation uses Bulletproofs's [23] linear-time inner product argument on the folded instance (which we made zero-knowledge in our evaluation); so while it has logarithmic proofs it still has verification linear in the size of one step. This could be expensive when the step function is large, but our step functions are relatively small (under 100K constraints). As a result, verification in Reef takes less than 1 second in all of our applications and workloads.

Proof size. The proof column includes all materials needed for the verifier to check the prover's claim. This includes all commitments and auxiliary proofs (e.g., $\pi_{consistency}, \pi_{poly}, \pi_{eq}$). Reef is succinct so all proof sizes are sublinear (logarithmic) in the size of the statement being proven. However, Reef's use of Hyrax means that document commitments consist of $\sqrt{|\mathsf{D}|}$ group elements. When the document is very large, such as in DNA, this can be sizable.

9.3 Comparative Performance

To contextualize the benefits of Reef, we compare it against several alternatives:

Application	Constraints per step	# of Steps	Compiler Time (s)	Solver Time (s)	Prover Time (s)	Verifier Time (s)	Proof Size (KB)	Commitment Size (KB)
Redactions								
Small Email	46,655	4	36.947	0.760	3.169	0.553	32.609	0.512
Large Email	65,727	7	217.628	3.221	5.923	0.701	33.361	1.024
ODoH	22,692	2	19.650	0.213	1.709	0.435	31.889	0.512
Passwords								
Match	19,982	5	17.960	0.067	2.573	0.418	31.665	0.128
Non-Match	20,728	6	18.636	0.357	2.963	0.416	31.761	0.128
DNA								
Match	81,722	8	62.351	12.830	17.708	0.908	34.417	131.072
Non-Match	81,722	1	62.357	3.006	10.838	0.915	34.417	131.072

FIGURE 6—Summary of all costs for the largest instance of each application evaluated in Reef. R1CS Constraints are for one step in Nova. Proof sizes include all the Nova zkSNARK proof as well as all auxiliary proofs (e.g., $\pi_{consistency}$). Commitment size measures the size of the document commitment. Reported times are the mean across 10 runs, and the standard deviation was less than 5% of the mean for all components and applications.

Application	DFA	DFA + Recursion	SAFA + nlookup	Reef
Redactions				
Small Email	76.300	1.721	0.760	0.733
Large Email	—	5.848	1.067	1.051
ODoH	2.064	0.640	0.409	0.362
Passwords				
Match	—	—	0.351	0.347
Non-Match	—	—	0.343	0.330
DNA				
Match	—	—	9.392	5.091
Non-Match	—	—	8.389	5.032

FIGURE 7—Maximum memory used (GB) by the Prover in Reef and baselines for our applications. Verifier's memory use is lower.

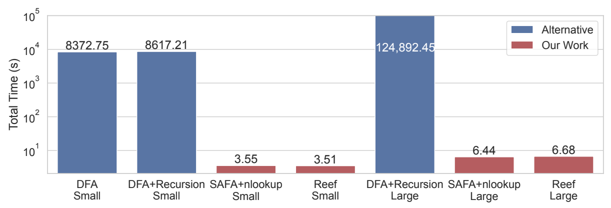
- **DFA**. This is the standard approach articulated in Section 3.1. To our knowledge, this is also the approach taken by the ZK-Email project [11]. We use Circom [2] to compile the match function and solve the corresponding R1CS instance since CirC [61] is not presently capable of compiling such large statements due to memory issues.
- **DFA + recursion**. This is the approach described in Section 4, which adds recursion and processes one character at a time. It uses a hash-chain as a vector commitment, which we believe is optimal (exactly one hash invocation) when accessing entries in the committed document sequentially. We use Circom and NovaScotia [6] to compile the step function and connect it with our zero-knowledge version of Nova (§7.3). Again, we are unable to compile these R1CS instances with CirC since they require expressing the (large) DFA delta function in constraints.
- **SAFA + lookup.** This is our implementation of Reef (§7) with SAFA and `nlookup`, but without projections (§6.4) or the hybrid table optimization (§6.5).

The metrics that we will consider in this section are memory usage and *end-to-end* completion time for the Prover, which includes both the time to solve and generate all witness values, and prove the satisfiability of the R1CS instance. Our tech report [15] has additional graphs for these same experiments but separates the time for solving and proving for readers interested in understanding the contribution of each component towards the end-to-end time. One thing to consider is that Reef pipelines the generation of a proof for step i with the generation of the witness for step $i + 1$ in parallel, as we discuss in our tech report [15]. As a result, the end-to-end time can sometimes be lower than the sum of the corresponding proving and solving times.

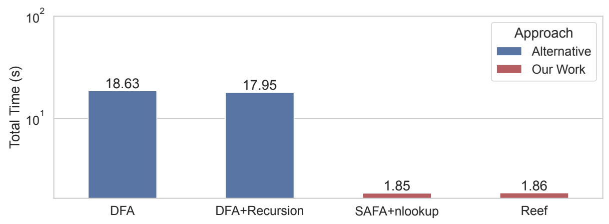
Results. Figure 7 shows the maximum memory use of Reef and the baselines for the same documents and regexes found in Figure 6. We are unable to run the password matching application with either of the DFA baselines due to its use of lookaheads, and the DNA application due to the massive R1CS instances (or number of steps) that are required. There are two observations: (1) using a recursive proof system has significant benefit in keeping the amount of memory required by the prover small since the prover only needs to prove one step at a time; and (2) the use of table projections in the DNA application means that the prover does not need to compute an expensive and memory-intensive sumcheck over the entire document, but rather works only over the projected table. This is why Reef uses less memory than SAFA + `nlookup`.

Figure 8 shows the end-to-end performance results. Across the board, SAFA +`nlookup` and Reef both dramatically outperform the DFA and DFA+Recursion approaches. Take for example *Redactions Small*. SAFA +`nlookup` and Reef took 3.55 and 3.51 seconds generate witnesses and prove, while the DFA baselines took over an hour. This suggests that Reef's ability to skip irrelevant parts of the document and the use of our zero-knowledge version of `nlookup` provides benefits.

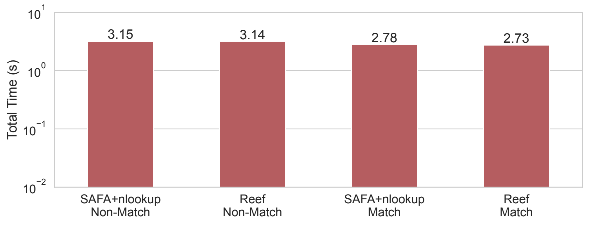
One might notice that DFA + recursion actually performs *worse* than just DFA in the case of small email redactions. There are a few reasons for this. First, while each step can process multiple characters, because the circuit still relies



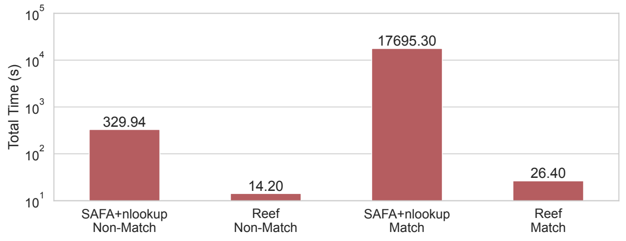
(a) Proof that a (small / large) committed email matches a redaction regex. DFA was unable to handle the large email.



(b) Proof that a committed document matches an ODoH regex.



(c) Proof that a committed password matches/does not match a password strength regex.



(d) Proof that a committed DNA document matches/does not match a DNA regex. Neither DFA nor DFA+recursion can handle this application.

FIGURE 8—Mean end-to-end completion time (which includes witness and proof generation) across 10 runs for proving that some committed document matches/does not match a regex with Reef and various alternatives. Standard deviations were less than 5% of the mean. Each subfigure describes a different application (regex) and type of document. The corresponding document sizes are found in Figure 6.

on `for` loops, it can only process a few characters per step before the circuit becomes too large. Second, in each step, there is some non-trivial work that is performed to check if the document is a match (§3.1). Third, each step of Nova adds a check to ensure foldings are correct (≈20,000 constraints).

Note that Reef also suffers from the latter two overheads (though the specific invariants for checking a match in a SAFA are different than in a DFA). However, Reef's use of lookup tables allow it to efficiently process large numbers of characters per step, which amortizes the latter two overheads over a batch of characters. Indeed, one of our optimizations (tech report [15]) is to process the optimal number of characters per step for a given regex in the SAFA's `match_step`, which amortizes these costs over the batch. We find this optimal value with a cost model that we have implemented in Reef's compiler.

The final impact to consider is that of Reef's additional optimizations. As discussed in Section 6.5, using hybrid tables reduces the number of constraints needed. This reduction is usually 1K–3K fewer constraints in the step function; full results are in our tech report [15]. This reduction in the size of the step function results in some small performance gains.

More significant is the impact of document projections in our DNA applications. Because common variants in the genome occur at known, fixed locations, by using projections (§6.4) Reef can skip over large parts of the genome directly to the start of the variant of interest. In the case of DNA matching, this results in a 50% reduction in proving time, and an over 99% reduction in solving time. While SAFA +`nlookup` can avoid the costs of a large document when it comes to proving, it still has to evaluate the sum-check protocol on the entire document for each step. When working with a document as large as DNA, this rapidly becomes prohibitive.

Takeaway. Reef handles a wide class of regexes at reasonable cost while producing succinct proofs. Each of Reef's optimizations provide benefits: SAFA allows expressing complicated regexes and skipping irrelevant parts of the document; recursion unleashes the power of SAFA by allowing the prover to prove only for as long as needed, and requiring much fewer memory; Reef's compiler picks the optimal number of characters to process per step for a regex to reduce the penalty of non-uniformity during recursion; hybrid lookup tables reduce the size of the step function; and projections make it possible for the prover to solve more efficiently when the location of relevance within the document is public.

10 Related Works

Reef relates to a series of very recent works on building proof systems for regexes [11, 56, 64, 79]. Reef aims to be as general as possible—targeting complex PCRE expressions and arbitrarily long documents. Reef achieves this by introducing SAFA, a brand new automata. In contrast, these other works target particular applications (middlebox packet inspection, malware hash membership tests) and use existing automata (DFAs or NFAs) enhanced with various encoding optimizations for their application domains. Reef can also handle these applications (and many others). It is unclear whether Reef would achieve better performance on these applications over these tailored proposals as we have not yet done an empirical comparison (they were all developed concurrently with Reef). One exception is ZK Regex from the ZK Email Verify project [11], which is in effect the "standard" approach in our evaluation, and which Reef outperforms in all applications.

Another related area is that of *secure regex evaluation* [33, 48, 53, 56, 58, 75]. Here the goal is for one party to supply the regex $\mathcal{R}$ and another party the document D, and to determine whether $\mathsf{D} \in \mathcal{L}[\![\mathcal{R}]\!]$ without revealing their inputs. This is a multi-party computation, and the techniques used in this domain aim to express *computation* rather than *verification*, which is the main theme in our work (via NP checkers).

11 Discussion and Future Work

Reef is the most expressive zero-knowledge proof system for regexes to date. It excels in situations where the document is large and the match is small, or when the regex gives Reef many opportunities to skip unnecessary work. In contrast, works like Zombie [79] excel in the opposite regime (small documents or when the document closely matches the regex). We think there are opportunities to combine the techniques in these two approaches to obtain the best of both worlds.

Reef has the ability to prove regex matches (and non-matches), but an interesting extension is to support "search and replace". In such a setting, the prover would prove not whether there is a match for some regex but rather that some committed document is the result of performing a regex search and replace transformation on some other committed document. Another extension to Reef is to support context free grammars. We think a similar approach of developing a custom automata would work there, and Reef already uses a stack for SAFA, which we show is quite efficient.

Source Code

Our code is available at:
https://github.com/eniac/Reef.

Acknowledgements

We thank Justin Thaler for insightful discussions on the problem of matching with wildcards and Riad Wahby for clarifying some questions on Hyrax and its polynomial commitment. We also thank Zachary DeStefano, Paul Grubbs, and Michael Walfish for helping us better understand Zombie, and Xiang Fu for pointing out a typo in an earlier version of this paper and for helping us better understand `zkreg`. Weidong Cui encouraged us to formalize our automata, which ultimately led to our formal development of SAFA. This work was funded in part by NSF grants CNS-2045861, CNS-2107147, CNS-2124184, a gift from the Arcological Swiss Association, and DARPA contract HR0011-17-C0047. Any views expressed herein are solely those of the authors listed.

References

[1] bellman. https://github.com/zkcrypto/bellman.

[2] Circom. https://github.com/iden3/circom.

[3] Neptune. https://github.com/lurk-lab/neptune.

[4] Nova: Recursive SNARKs without trusted setup. https://github.com/microsoft/Nova.

[5] Nova: Recursive SNARKs without trusted setup. https://github.com/sga001/Nova.

[6] Nova-scotia. https://github.com/nalinbhardwaj/Nova-Scotia.

[7] Perl-compatible regular expressions (PCRE). https://www.pcre.org/original/doc/html/pcresyntax.html.

[8] Reef: A zkSNARK system for proving that a committed document matches a regex. https://github.com/eniac/Reef.

[9] https://bit.ly/reef-min-dfa, 2023.

[10] Introduction to Microsoft Entra Verified ID. https://learn.microsoft.com/en-us/azure/active-directory/verifiable-credentials/decentralized-identifier-overview, 2023.

[11] Zk email verify. https://github.com/zkemail/zk-email-verify, 2023.

[12] A. V. Aho and M. J. Corasick. Efficient string hatching: an aid to bibliographic search. *Communications of the ACM*, 18, 1975.

[13] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[14] S. Angel, A. J. Blumberg, E. Ioannidis, and J. Woods. Efficient representation of numerical optimization problems for SNARKs. In *Proceedings of the USENIX Security Symposium*, 2022.

[15] S. Angel, E. Ioannidis, E. Margolin, S. Setty, and J. Woods. Reef: Fast succinct non-interactive zero-knowledge regex proofs. Cryptology ePrint Archive, Paper 2023/1886, 2023. https://eprint.iacr.org/2023/1886.

[16] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

[17] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046, 2018. https://eprint.iacr.org/2018/046.

[18] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2014.

[19] D. Boneh, J. Drake, B. Fisch, and A. Gabizon. Halo Infinite: Recursive zk-SNARKs from any Additive Polynomial Commitment Scheme. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2021.

[20] S. Bowe, J. Grigg, and D. Hopwood. Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Paper 2019/1021, 2019. https://eprint.iacr.org/2019/1021.

[21] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[22] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[23] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[24] B. Bünz and B. Chen. ProtoStar: generic efficient accumulation/folding for special sound protocols. Cryptology ePrint Archive, Paper 2023/620, 2023. https://eprint.iacr.org/2023/620.

[25] B. Bünz, A. Chiesa, W. Lin, P. Mishra, and N. Spooner. Proof-carrying data without succinct arguments. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2021.

[26] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner. Proof-carrying data from accumulation schemes. In *Proceedings of the Theory of Cryptography Conference (TCC)*, 2020.

[27] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM (JACM)*, 28(1), 1981.

[28] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2015.

[29] R. Cramer and I. Damgård. Zero-knowledge proofs for finite field arithmetic, or: Can zero-knowledge be for free? In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1998.

[30] L. Eagen, D. Fiore, and A. Gabizon. cq: Cached quotients for fast lookups. Cryptology ePrint Archive, Paper 2022/1763, 2022. https://eprint.iacr.org/2022/1763.

[31] A. Fellah, H. Jürgensen, and S. Yu. Constructions for alternating finite automata. *International journal of computer mathematics*, 35(1-4):117–132, 1990.

[32] N. Franzese, J. Katz, S. Lu, R. Ostrovsky, X. Wang, and C. Weng. Constant-overhead zero-knowledge for ram programs. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–191, 2021.

[33] K. B. Frikken. Practical private dna string searching and matching through efficient oblivious automata evaluation. In *Data and Applications Security XXIII: 23rd Annual IFIP WG 11.3 Working Conference, Montreal, Canada, July 12-15, 2009. Proceedings 23*, pages 81–94. Springer, 2009.

[34] A. Gabizon and Z. J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Paper 2020/315, 2020. https://eprint.iacr.org/2020/315.

[35] A. Gabizon and Z. J. Williamson. Proposal: The turbo-PLONK program syntax for specifying SNARK programs. https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo_plonk.pdf, 2020.

[36] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2013.

[37] A. Golovnev, J. Lee, S. Setty, J. Thaler, and R. S. Wahby. Brakedown: Linear-time and post-quantum snarks for r1cs. *Cryptology ePrint Archive*, 2021.

[38] G. Gramlich and G. Schnitger. Minimizing nfas and regular expressions. In *STACS 2005: 22nd Annual Symposium on Theoretical Aspects of Computer Science, Stuttgart, Germany, February 24-26, 2005. Proceedings 22*, pages 399–411. Springer, 2005.

[39] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In *USENIX Security Symposium*, volume 2021, 2021.

[40] P. Grubbs, A. Arun, Y. Zhang, J. Bonneau, and M. Walfish. Zero-knowledge middleboxes. In *Proceedings of the USENIX Security Symposium*, 2022.

[41] T. Hansen, D. Crocker, and P. Hallam-Baker. Domainkeys identified mail (DKIM) service overview. https://www.rfc-editor.org/rfc/rfc5585.html, 2009. RFC 5585.

[42] D. Hopwood. The Pasta curves for Halo 2 and beyond. https://electriccoin.co/blog/the-pasta-curves-for-halo-2-and-beyond/, 2020.

[43] S. Jarecki, H. Krawczyk, and J. Xu. Opaque: An asymmetric PAKE protocol secure against pre-computation attacks. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2018.

[44] K. Jiang, D. Chait-Roth, Z. DeStefano, M. Walfish, and T. Wies. Less is more: refinement proofs for probabilistic proofs. Cryptology ePrint Archive, Paper 2022/1557, 2022. https://eprint.iacr.org/2022/1557.

[45] M. Jones, J. Bradley, and N. Sakimura. JSON web token (JWT). https://datatracker.ietf.org/doc/html/rfc7519, 2015. RFC 7519.

[46] A. R. Karlin, H. W. Trickey, and J. D. Ullman. Experience with a regular expression compiler. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1983.

[47] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2010.

[48] F. Kerschbaum. Practical private regular expression matching. In *IFIP International Information Security Conference*, pages 461–470. Springer, 2006.

[49] E. Kinnear, P. McManus, T. Pauly, T. Verma, and C. A. Wood. Oblivious DNS over HTTPS. https://www.rfc-editor.org/rfc/rfc9230, 2022. RFC 9230.

[50] A. Kothapalli and S. Setty. SuperNova: proving universal machine executions without universal circuits. Cryptology ePrint Archive, Paper 2022/1758, 2022. https://eprint.iacr.org/2022/1758.

[51] A. Kothapalli and S. Setty. HyperNova: recursive arguments for customizable constraint systems. *Cryptology ePrint Archive*, 2023.

[52] A. Kothapalli, S. Setty, and I. Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In *Advances in Cryptology–CRYPTO 2022: 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15–18, 2022, Proceedings, Part IV*, pages 359–388. Springer, 2022.

[53] P. Laud and J. Willemson. Universally composable privacy preserving finite automata execution with low online and offline complexity. *Cryptology ePrint Archive*, 2013.

[54] J. Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In *Proceedings of the Theory of Cryptography Conference (TCC)*, 2021.

[55] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1990.

[56] N. Luo, C. Weng, J. Singh, G. Tan, R. Piskac, and M. Raykova. Privacy-preserving regular expression matching using nondeterministic finite automata. Cryptology ePrint Archive, Paper 2023/643, 2023. https://eprint.iacr.org/2023/643.

[57] J. Miller, D. Waite, and M. Jones. JSON web proof. https://www.ietf.org/archive/id/draft-ietf-jose-json-web-proof-00.html, 2023.

[58] P. Mohassel, S. Niksefat, S. Sadeghian, and B. Sadeghiyan. An efficient protocol for oblivious dfa evaluation and applications. In *Topics in Cryptology–CT-RSA 2012: The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27–March 2, 2012. Proceedings*, pages 398–415. Springer, 2012.

[59] A. Nitulescu. SoK: Vector commitments. https://www.di.ens.fr/~nitulesc/files/vc-sok.pdf, 2021.

[60] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives re-examined. *Journal of Functional Programming*, 19(2):173–190, 2009.

[61] A. Ozdemir, F. Brown, and R. S. Wahby. Circ: Compiler infrastructure for proof systems, software verification, and more. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022.

[62] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2001.

[63] N. Pippenger. On the evaluation of powers and related problems. In *Proceedings of the Annual Symposium on Foundations of Computer Science (SFCS)*, 1976.

[64] M. Raymond, G. Evers, J. Ponti, D. Krishnan, and X. Fu. Efficient zero knowledge for regular language. Cryptology ePrint Archive, Paper 2023/907, 2023. https://eprint.iacr.org/2023/907.

[65] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4:161–174, 1991.

[66] S. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2020.

[67] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[68] S. Setty, J. Thaler, and R. Wahby. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552, 2023. https://eprint.iacr.org/2023/552.

[69] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *Proceedings of the USENIX Security Symposium*, 2012.

[70] S. Shin, K. Kobara, and H. Imai. Security proof of AugPAKE. Cryptology ePrint Archive, Paper 2010/334, 2010. https://eprint.iacr.org/2010/334.

[71] T. Solberg. A brief history of lookup arguments. https://github.com/ingonyama-zk/papers/blob/main/lookups.pdf, 2023.

[72] C. Stanford, M. Veanes, and N. Bjørner. Symbolic boolean derivatives for efficiently solving extended regular expression constraints. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 620–635, 2021.

[73] T. Taubert and C. A. Wood. SPAKE2+, an augmented PAKE. https://www.rfc-editor.org/rfc/internet-drafts/draft-bar-cfrg-spake2plus-08.html, 2022.

[74] K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6), 1968.

[75] J. R. Troncoso-Pastoriza, S. Katzenbeisser, and M. Celik. Privacy preserving error resilient dna searching through oblivious automata. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 519–528, 2007.

[76] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. Doubly-efficient zkSNARKs without trusted setup. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[77] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In *Proceedings of the USENIX Security Symposium*, 2021.

[78] T. Wu. The secure remote password protocol. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 1998.

[79] C. Zhang, Z. DeStefano, A. Arun, J. Bonneau, P. Grubbs, and M. Walfish. Zombie: Middleboxes that don't snoop. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024.

[80] J. Zhang, T. Xie, Y. Zhang, and D. Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[81] L. Zhao, Q. Wang, C. Wang, Q. Li, C. Shen, and B. Feng. Veriml: Enabling integrity assurances and fair payments for machine learning as a service. *IEEE Transactions on Parallel and Distributed Systems*, 2021.