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) [46, 72, 75, 80] 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) [45] 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 [48] or verifiable credentials [15, 60] by "redacting" all other entries.

• *ODoH blocklisting. Oblivious DNS over HTTPS* [52] 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 [44].

• *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 [76], 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 derives its NP checker from regexes given in PCRE syntax [7]. Reef's NP checker supports a wider class of regexes than prior work, while also producing smaller arithmetizations. In particular, some works [16, 36] 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 [59], `zkreg` [66], and Zombie [81] reduce these costs: `zkreg` uses Aho-Corasick DFA (ADFA) that lead to $\mathcal{O}(|\mathsf{D}| \cdot |Q_{ADFA}|)$ constraints, while ZK-Regex and Zombie [81] leverage Thompson's NFA (TNFA) and produce $\mathcal{O}(|\mathsf{D}| \cdot |Q_{TNFA}|)$ constraints.

Reef's NP checker improves things further as it 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 $\alpha = \mathcal{O}(\max(|\mathsf{D}|, |Q_{SAFA}|))$. Note that α is significantly smaller than this worst-case upper bound whenever the regex gives Reef opportunities to *skip* unnecessary work. For example, if $\Sigma = \{a, b, c\}$, the regex $\mathcal{R} =$ "`a.*b`" (meaning that D must have an "a" eventually followed by "b") results in $\alpha = 2$ because Reef can skip all characters except one instance of "a" and a subsequent "b". 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".

Introduces skipping automata. Above we allude to the idea of "skipping" irrelevant characters in the document 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.

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 [55], 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* [17, 26, 41, 57, 78, 82] for multilinear polynomials to commit to D, and a *lookup argument* [54] 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 [54] 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.

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).*`",

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

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

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* [58] which incurs costs linear in $|\mathsf{D}|$ (million 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 7 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 [19, 25, 42, 49, 63, 74]. 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 [35].

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 [40, 69], Plonkish [39], AIR [20], CCS [70]), 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 [55], but all of our ideas apply to more general arithmetizations (e.g., CCS [70]). 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 [18, 69, 71] 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:

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

To see why this represents the original computation, observe that we introduced auxiliary variables called *guard*, *tmp*, *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) $x_0 \neq 30$, $guard = 0$, $tmp = x_1$, $y = prod = x_0 \times inv$, $inv = tmp^{-1}$. 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.

Appendix E shows how to convert these constraints into matrices A, B, and C. The solution vector z is $(y, 1, w)$, where $w = (x_0, x_1, guard, tmp, prod, inv)$ 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 [18, 24, 47, 71, 79, 83].

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:

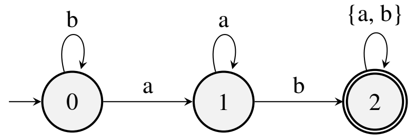
- **Completeness, Soundness, Succinctness, ZK**. These are analogs of the definitions given for zkSNARKs (§2.2) for the concrete R1CS instance that represents the statement "I know an opening of *comm*, and it matches $\mathcal{R}$" (or not).
- **Public verifiability**: The proof should be verifiable 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 [16, 36]. Two recent works [59, 81] improve upon this design by converting the regex to a Thompson NFA (TNFA) [76] 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 a DFA or TNFA fails to meet Reef's expressiveness goals. The most recent works in this area [59, 66, 81] lack support for at least one of: negations such as "`a[^[:space:]]b`", lookarounds, or unbounded repetition "`a*`".

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 [81] 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 $\alpha = \mathcal{O}(\max(|\mathsf{D}|, |Q_{SAFA}|))$; 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* [21–23, 27–29, 53, 54], 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* [17, 26, 41, 50, 57, 78, 82] or *vector* [62] 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 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 three 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. Last, with recursive zkSNARKs like Nova [55], if $\mathcal{P}$ wants to prove the *same* step function many times (which is the case with `match_step`), there are significant performance gains.

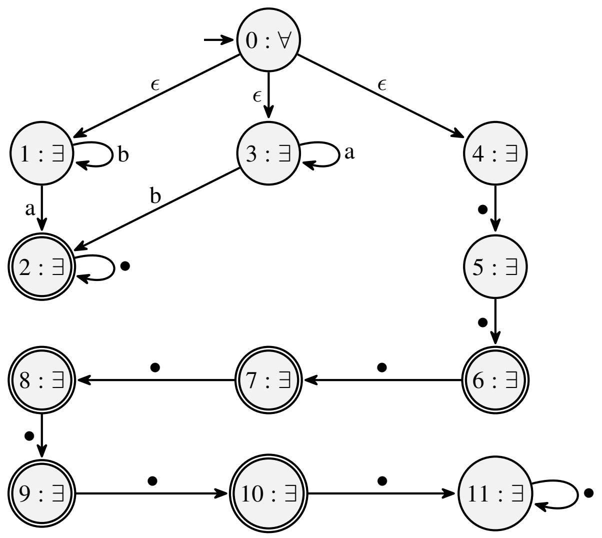


FIGURE 2—AFA for regex $\mathcal{R}$ = `^(?=.*a)(?=.*b).{2,6}$`.

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 [31] 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 [31] 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; $\lambda_q : Q \to \{\forall, \exists\}$ 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 [10] 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 $\mathsf{D} = acbcc$ which is accepted since the three branches out of state 0 run in parallel and each branch terminates in an accepting state. If instead $\mathsf{D} = bccbb$, 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 [35].

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 $\mathsf{D} \in \mathcal{L}[\![\mathcal{R}]\!]$”? (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.

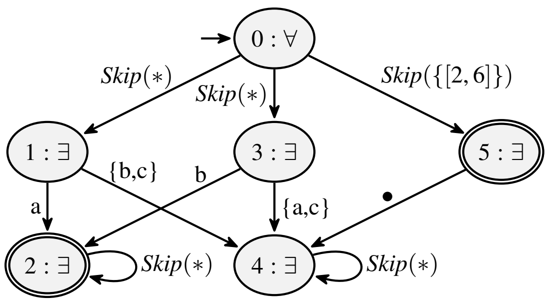


FIGURE 3—SAFA for regex $\mathcal{R}$ = `^(?=.*a)(?=.*b).{2,6}$` over alphabet $\Sigma = \{a,b,c\}$. *Skip*$(*)$ means the skip $\{[0,+\infty)\}$.

```
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.

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 intervals, $s = \{i_1, \dots, i_n\}$, 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.

A few remarks about skips. 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. Second, observe that skips generalize epsilon transitions: we can simply define skip $\epsilon = \{[0,0]\}$. Third, we can support Kleene-star wildcard regions with *Skip*$(*) = \{[0,\infty)\}$.

SAFA. With the above notion of skips we can then define *Skipping Alternating Finite Automaton (SAFA)* as the 7-tuple $(Q, E, \Sigma, q_0, bq, \lambda_q, \lambda_e, \delta, \mathcal{F})$, 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 $\exists$; $\lambda_e : E \to \mathit{Skip} \uplus \Sigma$ 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.

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. The symbol $\uplus$ is the *disjoint union*.

Example. We defer the formal definition of skips and the various transitions to Appendix B. 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*$\{[2,6]\}$ in the SAFA. This compression is possible because skips form a Monoid (Appendix A.1), so ϵ (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 Appendix B.6. The translation from AFA to SAFA is trivial as any AFA is a SAFA without skips.

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 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 Appendix G.

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 [34, 38, 54, 73] 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` [54]. 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 [58].

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 do k lookups of m elements, `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:

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

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.

When used with Nova, this results in $\mathcal{O}(m\cdot\log n)$ constraints for the step function. The Nova prover does a total of $\mathcal{O}(n)$ additional work in the form of finite field operations per series of m lookups, while the Nova 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 [54, §7].

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 with 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 $\mathcal{O}(m\cdot logn)$. The SAFA table is of size $\mathcal{O}(|Q_{SAFA}|\cdot|\Sigma|)$ in the worst case—a transition for every character from every state. While it may seem that the number of lookups m should be at most $\mathcal{O}(|\mathsf{D}|)$ that is not always the case. With no lookarounds, $m \le |\mathsf{D}|$. However, because SAFA may have multiple branches for lookarounds, certain parts of D may be looked up more than once. In that case $m \le |Q_{SAFA}|$. We thus upper bound m by $\alpha = \mathcal{O}(max(|\mathsf{D}|, |Q_{SAFA}|))$. The number of constraints needed to check all the lookups is therefore $\mathcal{O}(\alpha \log(|Q_{SAFA}|\cdot|\Sigma|))$.

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 [26, 57, 65, 78]. 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) [78, §6.1], but one could make other choices to get different tradeoffs (e.g., Dory [57] has smaller commitments but its *ProveEval* algorithm results in larger proofs and is more expensive).

Making `nlookup` zero-knowledge. `nlookup` [54] does not explicitly discuss a way to guarantee zero-knowledge during lookups. Here we give a concrete proposal, based on standard techniques [33, 68, 78]. 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 [68]) 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 (≈ 300 constraints) as we make c an *outer commitment* [32] (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 [78] and similar polynomial commitments [26, 57], 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 $\mathcal{V}$. Appendix F 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 expressing $O(\log(|\mathsf{D}| \cdot |S|))$ hashes (for the Fiat-Shamir transform in `nlookup`) and $m \cdot (\log(|\mathsf{D}| \cdot |S|))$ multiplications in 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 $O(\log(|\mathsf{D}| + |S|))$ hashes and $2m\log(|D| + |S|)$ multiplications per step. 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 $\ell = \log(2 \cdot \max(|D|, |S|))$. 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 $\pi_{polytail} = ProveEval(\widetilde{D}, q_r[1..], v_d)$ 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 $\pi_{polytail}$ 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 [67] 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 Appendix G.

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 [54] must perform in each step. In particular, it contains a series of Fiat-Shamir challenges that we generate with the Poseidon hash function [43] using the Neptune library [3]. Finally, Reef uses the CirC [64] 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 Appendix D.4. 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 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.

8.1 Experimental Setup

We run all of our experiments 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.

Application	Document Size (B)	# States	# Transitions	R1CS Constraints	# Steps	Compiler Time (s)	Solver Time (s)	Prover Time (s)	Verifier Time (s)	Proof Size (KB)
Redactions										
Small Email	415	331	42,315	52,631	4	340.953	1.243	3.580	0.521	33.665
Large Email	1,000	908	116,748	54,636	10	550.010	5.005	7.012	0.534	33.793
ODoH	128	36	4,007	22,692	2	58.029	0.215	1.912	0.420	34.193
Passwords										
Match	12	21	1,178	19,982	5	67.999	0.096	2.868	0.401	32.433
Non-Match	9	21	1,178	20,728	6	31.937	0.377	3.252	0.407	33.265
DNA										
Match	32.3×10^6	976	4,857	85,352	8	252.596	8.037	18.254	0.886	165.745
Non-Match	32.3×10^6	976	4,857	95,916	1	509.691	3.111	12.449	0.931	165.809

FIGURE 5—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., $\pi_{consistency}$) and commitments needed to verify the prover's claim. Times are averaged across 10 runs, standard deviation was less than 5% for all components and applications.

8.2 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. Figure 5 reports the results of the largest instances based on SAFA size. However, full results, all document sizes, and a list of all regexes can be found in Appendix H.

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 Appendix G 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.

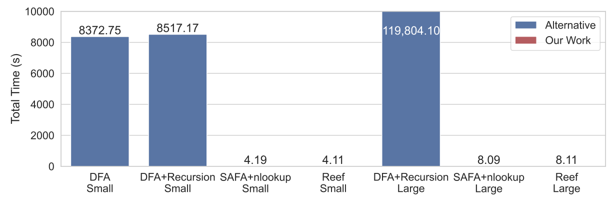
Verification. The verifier's costs depend on the number of R1CS constraints for a single step (since Nova folds all steps into one), as well as the cost to evaluate the SAFA polynomial at a random point, and check the consistency polynomial evaluation, and the equality proof. Nova's current implementation uses Bulletproofs's [26] 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., $\pi_{eq}, \pi_{consistency}$). 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.

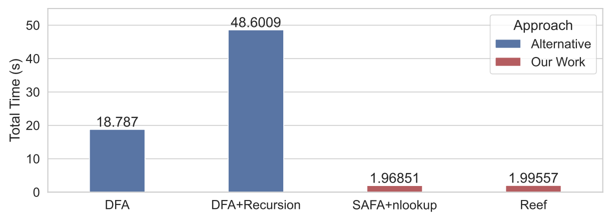
8.3 Comparative Performance

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

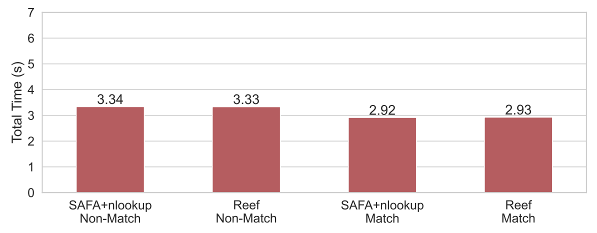
- **DFA**. This is the standard approach articulated in Section 3.1. To our knowledge, this is also the approach taken by the ZK-Email project [16]. We use Circom [2] to compile the match function and solve the corresponding R1CS instance since CirC [64] is not presently capable of compiling such large 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



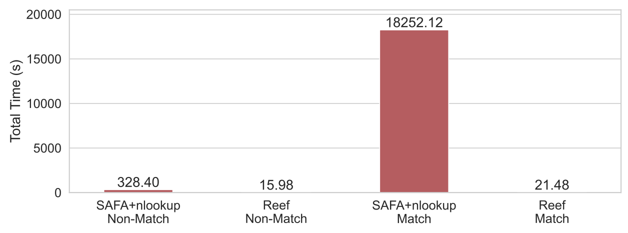
(a) Proof that a (small / large) committed email matches a redaction regex. DFA was unable to handle the large email.



(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 6—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 5.

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 circuits 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 metric that we will consider in this section is the *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. Appendix H 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 Appendix G. As a result, the end-to-end time can sometimes be lower than the sum of the corresponding proving and solving times.

Results. Figure 6 shows the results for the same documents and regexes found in Figure 5. 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. In the cases where we are able to do a comparison, SAFA +`nlookup` and Reef both dramatically outperform the DFA and DFA+Recursion approaches. Take for example *Redactions Small*. SAFA +`nlookup` and Reef took 4.12 and 4.10 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. There are a few reasons for this. First, each step processes a single character in a document, necessitating a large number of steps. 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 includes a verifier circuit (10,000 gates on each curve in the cycle).

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 does not process a single character in each step, allowing it to amortize the latter two overheads over a batch of characters. Indeed, one of our optimizations (Appendix G) 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. Of course, the baseline can also process a batch of characters in each step allowing it to amortize the latter two overheads over the batch. But we find that even if we ignore the latter two overheads, Reef is still orders of magnitude faster than DFA or DFA+recursion, not to mention that Reef is more expressive as well.

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 Appendix H. 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; 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.

9 Related Works

Reef relates to a series of very recent works on building proof systems for regexes [16, 59, 66, 81]. 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 and none of them have released their code). One exception is ZK Regex from the ZK Email Verify project [16], 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* [37, 51, 56, 59, 61, 77]. 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).

10 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 [81] 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 source code is open source and is freely 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. 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, 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] Regex filters for pi-hole. https://github.com/mmotti/pihole-regex/blob/master/regex.list.

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

[11] https://nordpass.com/most-common-passwords-list/, 2023.

[12] https://www.cs.cmu.edu/~enron/, 2023.

[13] https://www.ncbi.nlm.nih.gov/gene/672, 2023.

[14] https://www.ncbi.nlm.nih.gov/gene/675, 2023.

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

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

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] P. Caron, J.-M. Champarnaud, and L. Mignot. A general framework for the derivation of regular expressions. *RAIRO-Theoretical Informatics and Applications*, 48(3):281–305, 2014.

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

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

[47] 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.

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

[49] 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.

[50] 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.

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

[52] 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.

[53] 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.

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

[55] 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.

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

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

[58] 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.

[59] 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.

[60] 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.

[61] 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.

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

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

[64] 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.

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

[66] 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.

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

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

[69] 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.

[70] 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.

[71] 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.

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

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

[74] 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.

[75] 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.

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

[77] 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.

[78] 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.

[79] 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.

$r,s ::=$	$\emptyset$	Empty set
$\mid$	ϵ	Empty string
$\mid$	$\mathcal{C}$	Non-empty character set ($\emptyset \subset \mathcal{C} \subseteq \Sigma$)
$\mid$	rs	concatenation
$\mid$	$r + s$	Logical or (alternation)
$\mid$	$r \,\&\, s$	Logical and (conjunction)
$\mid$	r^*	Kleene-closure
$\mid$	$r\{n,m\}$	Bounded repetition ($n, m \in \mathbb{N}$)

FIGURE 7—Low-level Regex syntax in Reef. Additionally define the wildcard notation . to be the full character set Σ and n-repetition as $r\{n\} = r\{n,n\}$.

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

[81] 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.

[82] 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.

[83] 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.

A Preliminaries

A.1 Monoids

A monoid is a triple $(A, \cdot, \epsilon)$ where A is the carrier set, $\cdot$ is the *append* operation and ϵ is the identity element of append, such that the following lemmas apply

- Associativity $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
- Left-identity $\epsilon \cdot a = a$.
- Right-identity $a \cdot \epsilon = a$.

A.2 Boolean Algebras

A boolean algebra or boolean lattice is a familiar algebraic structure to computer scientists. A 6-tuple $(A, \top, \bot, \wedge, \vee, \neg)$ where A is the carrier set, $\top \in A, \bot \in A$ represent *true* or bit 1, and *false* or bit 0 respectively.

The binary combinators $\wedge, \vee$ are logical *and* and logical *or*, correspond to conjunction and disjunction respectively and the unary $\neg$ logical *not* corresponds to negation. A boolean algebra is closed in A under $\wedge, \vee, \neg$ and has the following lemmas.

- Associativity of $\vee, \wedge$.
- Commutativity of $\vee, \wedge$.
- Distributivity of $\wedge$ over $\vee$ and $\vee$ over $\wedge$.
- $\bot$ the unit of $\vee$.
- $\top$ the unit of $\wedge$.
- Annihilation for $\vee, \top$ and $\wedge, \bot$ respectively.
- Idempotence of $\vee, \wedge$.
- Complement rules for $\vee, \wedge$ and $\neg$.

Having a boolean algebra provides us with familiar intuitions, specifically DeMorgan's laws which are derivable from the lemmas above.

The free Boolean algebra over carrier set A is $B(A)$ is the most general boolean algebra such that the boolean algebra laws are satisfied and every other Boolean algebra can be derived from it. The positive free Boolean algebra over A is $B^+(A)$ is the free boolean algebra without using negation.

We combine Monoids and Boolean Algebras in definition A.3 by taking a Monoid $(A, \epsilon, \cdot)$ and the free positive Boolean algebra $B^+(A)$ over that monoid. We can show the monoid operation $\cdot$ can be overloaded to indicate elementwise concatenation over free Boolean algebras

$$\begin{aligned} \cdot \quad &: B^+(A) \to A \to B^+(A) \\ (a \wedge b) \cdot x &= (a \cdot x) \wedge (b \cdot x) \\ (a \vee b) \cdot x &= (a \cdot x) \vee (b \cdot x) \end{aligned}$$

$$\begin{aligned} \cdot \quad &: A \to B^+(A) \to B^+(A) \\ x \cdot (a \wedge b) &= (x \cdot a) \wedge (x \cdot b) \\ x \cdot (a \vee b) &= (x \cdot a) \vee (x \cdot b) \end{aligned}$$

A.3 Regular expressions

The low-level regular expression (regex) syntax used in Reef (fig. 7) is the one from [63] with the addition of bounded repetition ($r\{n,m\}$) which is useful to handle separately for performance reasons. We see how $r\{n,m\} = r\{n\} + \ldots + r\{m\}$ so this syntax reduces to exactly to regex with character sets from Owens et al [63], so by the same work its a regular language with a translation to DFA. For concatenation $r \cdot s$ we might write rs to mean the same.

Given an alphabet Σ, the language accepted by regex r is defined as $\mathcal{L}[\![r]\!] \subseteq \Sigma^*$.

Definition A.1.

$$\begin{aligned}
\mathcal{L}[\![\emptyset]\!] &= \emptyset \\
\mathcal{L}[\![\epsilon]\!] &= \{\epsilon\} \\
\mathcal{L}[\![\mathcal{C}]\!] &= \mathcal{C} \\
\mathcal{L}[\![r \cdot s]\!] &= \{uv \mid u \in \mathcal{L}[\![r]\!],\ v \in \mathcal{L}[\![s]\!]\} \\
\mathcal{L}[\![r + s]\!] &= \mathcal{L}[\![r]\!] \cup \mathcal{L}[\![s]\!] \\
\mathcal{L}[\![r \& s]\!] &= \mathcal{L}[\![r]\!] \cap \mathcal{L}[\![s]\!] \\
\mathcal{L}[\![r^*]\!] &= \{\epsilon\} \cup \mathcal{L}[\![r \cdot r^*]\!] \\
\mathcal{L}[\![r\{n,m\}]\!] &= \begin{cases} \{\epsilon\} & \text{if } n = m = 0 \\ \mathcal{L}[\![r \cdot r\{0, m-1\}]\!] & \text{if } n = 0 \\ \mathcal{L}[\![r \cdot r\{n-1, m-1\}]\!] & \text{otherwise} \end{cases}
\end{aligned}$$

A.4 Derivatives of regular expressions

Brzozowzki [25] first defined the derivative of a regex. The derivative $d_\alpha(r)$ of r, given a character $\alpha \in \Sigma$, is another regular expression, such that its language $\mathcal{L}[\![d_\alpha(r)]\!]$ contains all the suffixes $w \subseteq \Sigma^*$ of $\mathcal{L}[\![r]\!]$ with prefix α. Formally, $\mathcal{L}[\![r]\!] = \{\alpha w \mid w \in \mathcal{L}[\![d_\alpha(r)]\!]\}$.

The regex derivative is a recursive algebraic procedure, before we can define it introduce the *nullable* predicate $\nu(r)$ to be `true` only when a regex accepts the empty string. Nullable regex correspond to accepting states in finite automata.

Definition A.2.

$$\begin{aligned}
\nu(\epsilon) &= \mathit{true} \\
\nu(r^*) &= \mathit{true} \\
\nu(\emptyset) &= \mathit{false} \\
\nu(\mathcal{C}) &= \mathit{false} \\
\nu(rs) &= \nu(r) \wedge \nu(s) \\
\nu(r + s) &= \nu(r) \vee \nu(s) \\
\nu(r \& s) &= \nu(r) \wedge \nu(s) \\
\nu(r\{n,m\}) &= \begin{cases} \mathit{true} & \text{if } n = 0 \\ \nu(r) & \text{otherwise} \end{cases}
\end{aligned}$$

As a reference the Brzozowki derivative [25] of a regex r with respect to a character $\alpha \in \Sigma$ is defined as follows.

$$\begin{aligned}
d_\alpha(\emptyset) &= \emptyset \\
d_\alpha(\epsilon) &= \emptyset \\
d_\alpha(\mathcal{C}) &= \begin{cases} \epsilon & \text{if } \alpha \in \mathcal{C} \\ \emptyset & \text{otherwise} \end{cases} \\
d_\alpha(r \cdot s) &= \begin{cases} (d_\alpha(r)s) \mid d_\alpha(s) & \text{if } \nu(r) \\ d_\alpha(r)s & \text{otherwise} \end{cases} \\
d_\alpha(r + s) &= d_\alpha(r) + d_\alpha(s) \\
d_\alpha(r \& s) &= d_\alpha(r) \& d_\alpha(s) \\
d_\alpha(r^*) &= d_\alpha(r) r^* \\
d_\alpha(r\{n,m\}) &= \begin{cases} \emptyset & \text{if } n = m = 0 \\ d_\alpha(r \cdot r\{0, m-1\}) & \text{if } n = 0 \\ d_\alpha(r \cdot r\{n-1, m-1\}) & \text{otherwise} \end{cases}
\end{aligned}$$

Continuing in the same direction, Antimirov in his seminal work [19] notes that the upper semilattice $(r, +, \emptyset)$ induces a congruence with regards to the Associativity, Commutativity, Idempotence and Zero-element laws of regex, or the *ACIZ laws* for short. This congruence defines a normalization procedure for minimizing automata that both Brzozowski and Antimirov exploit.

Antimirov though gives an algebraic construction that admits ACIZ equivalence on derivatives "for free". *Partial derivatives* on regex given a character produce a set of regex, intuitively changing the meaning of a derivative from a one-to-one mapping to a one-to-many. This has a significant benefit, regex sets generated by partial derivatives, like all sets, admit the ACIZ lemmas. There is no need for a weaker equivalence relation, only set equality.

Caron et al. [30] generalize Antimirov's partial derivatives from sets to arbitrary *support structures*, a significant generalization used in Reef. The support structure we chose to observe as the result of our derivative is the free positive Boolean algebra A.2, the most general boolean algebra without the use of negation. This structure is convenient as Alternating Finite Automate (AFA) operate over states which are in $B^+(T)$ for some carrier set T.

Here is how our *Generalized Antimirov partial derivatives* $\partial_\alpha : \mathit{regex} \to B^+(\mathit{regex})$ are defined.

Definition A.3.

$$\begin{aligned}
\partial_\alpha(\emptyset) &= \bot \\
\partial_\alpha(\epsilon) &= \bot \\
\partial_\alpha(\mathcal{C}) &= \begin{cases} \epsilon & \text{if } \alpha \in \mathcal{C} \\ \bot & \text{otherwise} \end{cases} \\
\partial_\alpha(r \cdot s) &= \begin{cases} \partial_\alpha(s) \vee \partial_\alpha(r) \cdot s & \text{if } \nu(r) \\ \partial_\alpha(r) \cdot s & \text{otherwise} \end{cases} \\
\partial_\alpha(r + s) &= \partial_\alpha(r) \vee \partial_\alpha(s) \\
\partial_\alpha(r \& s) &= \partial_\alpha(r) \wedge \partial_\alpha(s) \\
\partial_\alpha(r^*) &= \partial_\alpha(r) \cdot r^* \\
\partial_\alpha(r\{n,m\}) &= \begin{cases} \bot & \text{if } n = m = 0 \\ \partial_\alpha(r \cdot r\{0, m-1\}) & \text{if } n = 0 \\ \partial_\alpha(r \cdot r\{n-1, m-1\}) & \text{otherwise} \end{cases}
\end{aligned}$$

We overload the monoidal append $\cdot$ operator to apply elementwise over the free positive boolean algebra $B^+(\mathit{regex})$ as indicated in A.2.

A.5 Automata

A Deterministic Finite Automaton (DFA) is a 5-tuple $(Q, \Sigma, q_0, \delta, \mathcal{F})$.

Q	:	the set of all states
Σ	:	The alphabet
$q_0 \in Q$	:	Initial state
$\delta : Q \times \Sigma \to Q$	:	Transition function
$\mathcal{F} \subseteq Q$	:	Set of accepting states

A Nondeterministic Finite Automaton (NFA) is a 5-tuple $(Q, \Sigma, q_0, \delta, \mathcal{F})$ where ϵ denotes the empty string

Q	:	the set of all states,
$\Sigma \cup \{\epsilon\}$	:	The alphabet with ϵ,
$q_0 \in Q$	:	Initial state
$\delta \subseteq Q \times \Sigma \times Q$	:	Transition relation
$\mathcal{F} \subseteq Q$	:	Set of accepting states

An Alternating Finite Automaton (AFA) [31] is a 6-tuple $(Q, \Sigma, q_0, \lambda_q, \delta, \mathcal{F})$ where

Q	:	the set of all states,
Σ	:	The alphabet,
$q_0 \in Q$	:	Initial state
$\lambda_q : Q \to \{\forall, \exists\}$	:	Label states $\forall$ or $\exists$
$\delta \subseteq Q \times \Sigma \times Q$	:	Transition relation
$\mathcal{F} \subseteq Q$	:	Set of accepting states

AFA are a generalization of Non-deterministic Finite Automata (NFA), which can be though as AFA with only existential states.

B Skipping Alternating Finite Automata

AFA enable the distinction between logical and/or branches. Next, to efficiently represent sparse matches we introduce an automaton that can ignore irrelevant parts the document. For example, the regex $\hat{}.\{1,1000\}ab\$$, indicates there is a cursor i such that $1 \leq i \leq 1000$ for a document d and $d_i d_{i+1} =$ "ab''. The contents of d in the range of i are irrelevant since wildcards match anything. We can not verify those 1000 equations and only verify the inequality on the cursor i above.

We must now give some preliminary definitions.

Interval sets are a collection of intervals of natural numbers in a light-weight memory representation that only stores the start and end points. Our implementation never iterates through an interval, thus allowing us to work with arbitrarily large intervals with nearly constant overhead.

B.1 Intervals

A left-closed, right-closed *interval* of natural numbers $[a, b]$ where $a \leq b$, $a, b \in \mathbf{N}$ represents the subset of natural numbers $\{\, i \mid a \leq i \leq b\}$ (inclusive in both ends).

A left-closed, unbounded interval $[a, +\infty)$ represents the subset of natural numbers $\{\, i \mid a \leq i\}$ (inclusive on the left, unbounded on the right). Those will be the two types of intervals we consider, the term *Intervals* will refer only to those and we call them *bounded* and *unbounded* intervals for brevity and represent them with the letters I, i, possibly using subscripts.

B.2 Skips/Interval sets

An *interval set* S or a *skip* is a Set datastructure. It is a collection of intervals $\{i_1, \ldots, i_n\}$ ordered by increasing starting point, such that there is no overlap between consecutive intervals.

We consider an overlap a difference of at most 1 between the end-point of the first and start point of the second. So for example $[1, 2], [4, 5]$ do not overlap but both $[1, 3], [2, 4]$ and $[1, 2], [3, 4]$ overlap and are both equivalent to $[1, 4]$. Formally, two intervals $[a_1, b_1],\ [a_2, b_2]$ overlap iff $max(a_1, b_1) + 1 \geq min(a_2, b_2)$.

Intervals and Interval sets admit the familiar set operations *union* ($\cup$), *intersection* ($\cap$), *complement* ($\neg$) and additionally the operation *append*($+$), which is element-wise addition of interval bounds. We use set notation $i \in s$ and $n \in s$, to mean i is an interval in s, or n is a number contained in any i.

B.3 Operations on Interval sets

We define the combinators $\cup, \cap, \neg$ on intervals as a preparation for defining the boolean algebra of Interval sets. Notice the return type of $\cup, \cap, \neg$ on Intervals is an Interval Set S, so while Intervals themselves are not closed under $\cup, \cap, \neg$ and as such, Intervals do *not* form a boolean algebra.

$$\cup \quad : I \to I \to S$$

$$[a_1, b_1] \ \cup [a_2, b_2] = \begin{cases} \{[min(a_1, a_2),\ max(b_1, b_2)]\} & max(a_1, a_2) \leq min(b_1, b_2) \\ \{[a_2, b_2], [a_1, b_1]\} & b_2 < a_1 \\ \{[a_1, b_1), (a_2, a_2)\} & \text{otherwise} \end{cases}$$

$$[a_1, \infty) \ \cup [a_2, b_2] = \begin{cases} \{[a_2, b_2], [a_1, \infty)\} & b_2 < a_1 \\ \{[a_1, \infty))\} & b_2 \geq a_1 \leq a_2 \\ \{[a_2, \infty))\} & \text{otherwise} \end{cases}$$

$$[a_1, b_1] \ \cup [a_2, \infty) = \begin{cases} \{[a_1, b_1], [a_2, \infty)\} & b_1 < a_2 \\ \{[a_2, \infty))\} & b_1 \geq a_2 \leq a_1 \\ \{[a_1, \infty))\} & \text{otherwise} \end{cases}$$

$$[a_1, \infty) \ \cup [a_2, \infty) = \begin{cases} \{[a_1, \infty))\} & a_1 \leq a_2 \\ \{[a_2, \infty))\} & \text{otherwise} \end{cases}$$

$\cap \quad : I \to I \to S$

$$[a_1,b_1] \cap [a_2,b_2] = \begin{cases} \{[max(a_1,a_2),\ min(b_1,b_2)]\} & max(a_1,a_2) \le min(b_1,b_2) \\ \{\} & \text{otherwise} \end{cases}$$

$$[a_1,\infty) \cap [a_2,b_2] = \begin{cases} \{[a_2,b_2]\} & b_2 \ge a_1 \le a_2 \\ \{[a_1,b_2]\} & b_2 \ge a_1 > a_2 \\ \{\} & \text{otherwise} \end{cases}$$

$$[a_1,b_1] \cap [a_2,\infty) = \begin{cases} \{[a_1,b_1]\} & b_1 \ge a_2 \le a_1 \\ \{[a_2,b_1]\} & b_1 \ge a_2 > a_1 \\ \{\} & \text{otherwise} \end{cases}$$

$$[a_1,\infty) \cap [a_2,\infty) = \begin{cases} \{[a_2,\infty)\} & a_1 \le a_2 \\ \{[a_1,\infty)\} & \text{otherwise} \end{cases}$$

$\neg \quad : I \to S$

$\neg\, [0,\ \infty) = \{\}$

$\neg\, [a,\ \infty) = \{[0, a-1]\}$

$\neg\, [0,\ a] = \{[a+1, \infty]\}$

$\neg\, [a,\ b] = \{[0, a-1], [b+1, \infty)\}$

Intervals are closed under the $+$ *append* operation and $\epsilon = [0,0]$ the identity interval and satisfy associativity, so intervals form a monoid.

$+ \quad : I \to I \to I$

$[a_1,\ \infty) + [a_2, \infty) = [a_1 + a_2, \infty)$

$[a_1,\ b_1] + [a_2, \infty) = [a_1 + a_2, \infty)$

$[a_1,\ \infty) + [a_2, b_2] = [a_1 + a_2, \infty)$

$[a_1,\ b_1] + [a_2, b_2] = [a_1 + a_2, b_1 + b_2]$

Interval sets also form a boolean algebra A.2, with $\top = \{[0,\infty)\}, \bot = \{\}$ and where the combinators $\vee = \cup, \wedge = \cap$ and $\neg$. We show that boolean algebra lemmas hold by simply observing that Interval sets are succinct representations of sets of natural numbers and all sets induce a boolean algebra with respect to set combinators.

$\cap \quad : S \to S \to S$

$\{i,\ \overline{i_n}\} \cup s = \{i \cup i_s \mid i_s \in s\} \cup \{\overline{i_n}\}$

$\{\} \cup s = s$

$\cap \quad : S \to S \to S$

$\{i,\ \overline{i_n}\} \cap s = \{i \cap i_s \mid i_s \in s\} \cap \{\overline{i_n}\}$

$\{\} \cap s = \{\}$

$\neg \quad : S \to S$

$\neg\{i_1, \overline{i_n}\} = \neg i_i \cap \neg\{\overline{i_n}\}$

$\neg\{\} = \{[0,\infty)\}$

$+ \quad : I \to I \to I$

$s_1 + s_2 = \bigcup \{i_1 + i_2 \mid i_1 \in s_1,\ i_2 \in s_2\}$

Notice the concatenation operation $+$ together with $\{[0,0]\}$ induce a Monoid structure, as for Interval sets, it is easy to show the associativity of $+$ and left/right identity laws hold. Interval sets form a boolean algebra with a monoidal structure – unsurprisingly the same algebraic structure enjoyed by regular expressions.

We can now formally define a finite automaton as the target of our regex compiler. As hinted earlier, SAFA are a generalization of AFA with *skips* on their transitions. A skip is an interval set s and when matching on a document at cursor position i, we can nondeterministically pick any $n \in s$ and continue matching at position $i + s$, without changing the outcome of the match. This is a powerful technique for matching on sparse documents utilizing non-deterministic witnesses.

B.4 SAFA formal definition

A Skipping Alternating Finite Automaton (SAFA) is a 8-tuple $(Q, E, \Sigma, q_0, bq, \lambda_q, \lambda_e, \delta, \mathcal{F})$ where

Q	:	the set of all states (nodes)
E	:	the set of all transitions (edges)
Σ	:	The alphabet
$q_0 \in Q$	:	Initial state
$\lambda_q : Q \to \{\forall,\ \exists\}$	:	Label nodes, $\forall$ or $\exists$
$\lambda_e : E \to S \uplus C$	:	Label edges, skip or character set
$\delta \subseteq Q \times E \times Q$	:	Transition relation
$\mathcal{F} \subseteq Q$	:	Set of accepting states

The only difference with the AFA definition is edge labels. In addition to a character set $C \subseteq \Sigma$ which matches one character from the document $\alpha \in C$ and increments the cursor by one, an edge might be labeled as a skip transition $s \in S$, which does not consume any characters from the document and increases the cursor nondeterministically by $n \in s \subseteq \mathbb{N}$.

For brevity, we overload the notation $(q, s, q') \in \delta$ to indicate a skip transition s, or more precisely there exists an edge $e \in E$ such that $(q, e, q') \in \delta$ and $\lambda_e(e) = s$. Also overload $(q, \alpha, q') \in \delta$ to indicate a character $\alpha \in \Sigma$ transition, or there exists an edge $e \in E$ and character set $C \subseteq \Sigma$ such that $(q, e, q') \in \delta$ and $\lambda_e(e) = C$ and $\alpha \in C$. As the labeling function λ_e maps to a disjoint union there is no confusion with this notation.

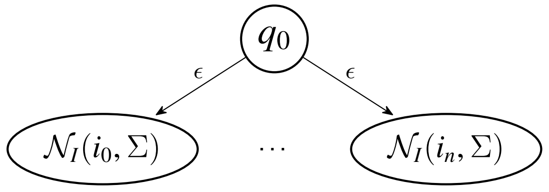


FIGURE 9—Combined NFA $\mathcal{N}(s,\Sigma)$ for interval set $s=\{\overline{i_n}\}$ with children the NFA $\mathcal{N}_I(\overline{i_n},\Sigma)$

.

Definition B.1.

$$\text{match}_{\mathcal{S}}(q,D,i) \triangleq \tag{1}$$
$$\begin{cases} q\in\mathcal{F}\wedge i=|D| & \text{(accept condition)}\\ \text{match}_{\forall}(q,D,i) & \text{if } \lambda_q(q)=\forall\\ \text{match}_{\exists}(q,D,i) & \text{if } \lambda_q(q)=\exists \end{cases}$$

$$\text{match}_{\forall}(q,D,i) \triangleq$$
$$\forall\, e,q',\ (q,e,q')\in\delta\rightarrow \text{match}_E(q',e,D,i) \tag{2}$$
$$\text{match}_{\exists}(q,D,i) \triangleq$$
$$\exists\, e,q',\ (q,e,q')\in\delta\wedge \text{match}_E(q',e,D,i) \tag{3}$$
$$\text{match}_E(q',e,D,i) \triangleq \tag{4}$$
$$\begin{cases} \text{match}_{\mathcal{S}}(q',D,i+1) & \text{if } \lambda_e(e)=D[i]\\ \exists n\in s,\ \text{match}_{\mathcal{S}}(q',D,i+n) & \text{if } \lambda_e(e)=s\in S\\ \mathit{false} & \text{otherwise} \end{cases}$$

FIGURE 8—SAFA semantics, mutually recursive predicate $\text{match}_{\mathcal{S}}$ is *true* iff document D at position $i\le|D|$ is accepted by SAFA $\mathcal{S}$.

B.5 SAFA Semantics

Now we can give the semantics of the language accepted by a SAFA $\mathcal{S}$. If $D\in\Sigma^*$ is a document with random-access, and $i\le|D|$ a cursor in the document, define the mutually-recursive decidable procedure $\text{match}_{\mathcal{S}}$ which returns *true* if at state q, a document D at index i is accepted by SAFA $\mathcal{S}$.

With the auxilary definitions of Figure 8 in place, finally define the language recognized by a SAFA $\mathcal{S}$ as

$$\mathcal{L}[\![\mathcal{S}]\!]=\{D\mid \text{match}_{\mathcal{S}}(q_0,D,0)\}$$

B.6 SAFA are regular

To prove SAFA are regular and have the same computational power as the source regex language, we must provide an isomorphism from SAFA to another finite automaton which is known to be regular, like DFA, NFA or AFA, as well as an isomorphism to our source language to show that no expressive power is lost during compilation.

Source language $\iff$ DFA The source language in fig. 7 is regular by giving a translation to the regular expression language with character sets in [63], which is shown to be regular by equivalence to a DFA. The only difference between our language and [63] is *bounded range* expressions $r\{a,b\}$ in fig. 7, which translate to an alternation of finite repetition $r\{a,b\}=r\{a\}+\ldots+r\{b\}$ where

$$\begin{aligned} r\{0\}&=\epsilon\\ r\{i\}&=r\cdot r\{i-1\} \end{aligned}$$

DFA $\iff$ AFA See [35] for a construction, the proof proceeds by constructing an intermediate NNFA where states are $Q\times Q$ sized boolean matrices which correspond to truth tables of $B^+(Q)$ of AFA states. By the above, a n-state AFA is equivalent to an at most 2^n-state NNFA and a 2^{2^n} DFA by the product construction as shown in [35]. The opposite direction is trivial, all DFA are AFA with only existential nodes.

AFA $\iff$ SAFA Finally, we show SAFA is regular by translation to AFA. We give a translation $\mathcal{N}(s,\Sigma)$ of skip s to NFA, given alphabet Σ. Then show substituting $\mathcal{N}(s,\Sigma)$ in place of s produces an AFA which recognizes the same language as the SAFA. The opposite direction, embedding an AFA to a SAFA is trivial, all AFA are SAFA without skips.

A skip is a finite set of intervals B.2, whose union is a possibly infinite subset of the natural numbers $s=\{\overline{i_n}\}\subseteq\mathbb{N}$. Before we give an NFA construction for s we construct a NFA $\mathcal{N}_I(i,\Sigma)$ for an interval i and alphabet Σ.

Intervals to DFA An NFA for a closed interval $[a,b]$ (where $a\le b$) is given by the following construction

Definition B.2.

$$\begin{aligned} \mathcal{N}_c(a,b,\Sigma)=(&\\ &Q:=\{q_1,q_2,\ldots,q_b\}\\ &\Sigma:=\Sigma\\ &q_0:=q_1\\ &\mathcal{F}:=\{q_a,\ldots,q_b\}\\ &\delta:=\{q_1\xrightarrow{\Sigma}q_2,q_2\xrightarrow{\Sigma}q_3,\ldots,q_{b-1}\xrightarrow{\Sigma}q_b\}\\ &) \end{aligned}$$

The NFA $N_{[a,b]}=\mathcal{N}_c(a,b,\Sigma)$ is a finite chain with accepting states $q_i,\ a\le i\le b$. Notice the final state q_b is an accepting state but does not transition. We will later step give a substitution function of an NFA in a SAFA $\mathcal{S}$ edge $q_A\xrightarrow{s}q_B$ which adds an ϵ-transition from q_b to SAFA state q_B.

The $\mathcal{N}_c$ construction will not work for an open interval $i=[a,\infty)$ as it will result in an infinite chain of states. Instead, construct a finite NFA with an accepting self-loop in the end that accepts infinite repeating strings.

Definition B.3.

$$
\begin{aligned}
&\mathcal{N}_o(a,\Sigma) = (\\
&\qquad Q := \{q_1, q_2, \ldots, q_a\} \\
&\qquad \Sigma := \Sigma \\
&\qquad q_0 := q_1 \\
&\qquad \mathcal{F} := \{q_a\} \\
&\qquad \delta := \{q_1 \xrightarrow{\Sigma} q_2, q_2 \xrightarrow{\Sigma} q_3, \ldots, q_{b-1} \xrightarrow{\Sigma} q_b, q_a \xrightarrow{\Sigma} q_a\} \\
&\qquad)
\end{aligned}
$$

Note the reflexive transition (q_a, Σ, q_a) on the only accepting state q_a means all a states must be traversed to accept. Now combine $\mathcal{N}_c(a,b,\Sigma)$ and $\mathcal{N}_o(a,\Sigma)$ to construct an NFA equivalent to an arbitrary interval, by conditionally branching on if the interval is closed or open.

Definition B.4.

$$
\mathcal{N}_I(i,\Sigma) = \begin{cases} \mathcal{N}_c(a,b,\Sigma) & \text{if } i = [a,b] \\ \mathcal{N}_o(a,\Sigma) & \text{if } i = [a,\infty) \end{cases}
$$

Interval sets to NFA Given skip $s = \{\overline{i_n}\}$ and alphabet Σ construct an $n+1$ state NFA $\mathcal{N}(s,\Sigma)$ as follows. Assume no capturing of state identifiers – state $q \in Q_{\mathcal{N}_I(i,\Sigma)}$ does not appear in another $\mathcal{N}_I(i',\Sigma)$ where $i \neq i'$.

Definition B.5.

$$
\begin{aligned}
&\mathcal{N}(s,\Sigma) = (\\
&\qquad Q := \{q_0\} \cup \bigcup_{i \in s} Q_{\mathcal{N}_I(i,\Sigma)} \\
&\qquad \Sigma := \Sigma \\
&\qquad q_0 := q_0 \\
&\qquad \mathcal{F} := \bigcup_{i \in s} \mathcal{F}_{\mathcal{N}_I(i,\Sigma)} \\
&\qquad \delta := \bigcup_{i \in s} \{(q_0, \epsilon, q_{0,\mathcal{N}(i,\Sigma)})\} \cup \bigcup_{i \in s} \delta_{\mathcal{N}(i,\Sigma)} \\
&\qquad)
\end{aligned}
$$

The initial state q_0 is non-accepting. Then add a nondeterministic choice from q_0 to each one of $\mathcal{N}_I(i_n,\Sigma)$. As long as any $\mathcal{N}_I(i_n,\Sigma)$ reaches an accepting state, then $\mathcal{N}(s,\Sigma)$ accepts. This concludes the construction or $\mathcal{N}(s,\Sigma)$.

Substitution in SAFA Now define formally a substitution procedure $\langle \mathcal{N}/e \rangle_{\mathcal{S}}$, for a SAFA $\mathcal{S}$, NFA $\mathcal{N}$ and SAFA edge $e \in E_{\mathcal{S}}$.

Remember that SAFA edges are a set $E_{\mathcal{S}}$ and map to either skips or character-sets via the labeling function $\lambda_e : E \to S \uplus C$. We build an edge set $E_{\mathcal{N}}$ and labeling function $\lambda_{\mathcal{N}} : E_{\mathcal{N}} \to C$ for NFA $\mathcal{N}$, such that

- No capturing: $E_{\mathcal{S}}$: $E_{\mathcal{N}} \cap E_{\mathcal{S}} = \emptyset$.
- $E_{\mathcal{N}}$ sound: $\forall e \in E_{\mathcal{E}},\ \exists q\ q'\ (q, \lambda_{\mathcal{N}}(e), q') \in \delta_{\mathcal{N}}$.
- $E_{\mathcal{N}}$ complete: $\forall q\ q'\ C,\ (q, C, q') \in \delta_{\mathcal{N}} \to \exists e \in E_{\mathcal{N}},\ \lambda_{\mathcal{N}}(e) = C$.

Also assume no capturing of states, $Q_{\mathcal{S}} \cap Q_{\mathcal{N}} = \emptyset$ are disjoint and their alphabets are equal. Let unique q_{src}, q_{dst}, such that $(q_{src}, e, q_{src}) \in \delta_{\mathcal{S}}$

Definition B.6.

$$
\begin{aligned}
&\langle \mathcal{N}/e_s \rangle_{\mathcal{S}} = (\\
&\qquad Q := Q_{\mathcal{N}} \cup Q_{\mathcal{S}} \setminus \{q_{src},\ q_{dst}\} \\
&\qquad E := E_{\mathcal{N}} \cup E_{\mathcal{S}} \setminus \{e_s\} \\
&\qquad \Sigma := \Sigma_{\mathcal{S}} \\
&\qquad \lambda_q := \lambda_{q,\mathcal{S}} \cup \{q \mapsto \exists \mid q \in Q_{\mathcal{N}}\} \\
&\qquad \lambda_e := \lambda_{e,\mathcal{S}} \cup \lambda_{\mathcal{N}} \\
&\qquad \delta := \delta_{\mathcal{S}} \\
&\qquad\quad \cup \{(q, e, q') \mid (q, C, q') \in \delta_{\mathcal{N}},\ C \subseteq \Sigma,\ \lambda_{\mathcal{N}}(e) = C\} \\
&\qquad\quad \cup \{(q_{src}, e_\epsilon, q_{0,\mathcal{N}})\} \cup \{(q_F, e_\epsilon, q_{dst}) \mid q_F \in \mathcal{F}_{\mathcal{N}}\} \\
&\qquad \mathcal{F} := \mathcal{F}_{\mathcal{S}} \\
&\qquad)
\end{aligned}
$$

The substitution construction may look complex in the formal definition but is intuitive to understand. For SAFA $\mathcal{S}$, NFA $\mathcal{N}$ and SAFA edge e_s with transition $(q_{src}, e, q_{src}) \in \delta_{\mathcal{S}}$, remove the transition (q_{src}, e, q_{src}) from $\delta_{\mathcal{S}}$ and replace it with $\mathcal{N}$ by connecting q_{src} and q_{dst} to the initial and accepting states of $\mathcal{N}$ respectively.

Notice the accepting states $\mathcal{F}_{\mathcal{S}}$ of $\mathcal{S}$ do not change, so if we can prove language equivalence of NFA $\mathcal{N}$ with the replaced edge e_s, we should be able to prove a key lemma for the SAFA $\iff$ AFA proof. The definition of a language for an edge $e \in E_{\mathcal{S}}$ comes directly from the match$_E$ rule in fig. 8.

Definition B.7.

$$
\mathcal{L}[\![e_{\mathcal{S}}]\!] = \begin{cases} C & \text{if } \lambda_e(e_{\mathcal{S}}) = C \subseteq \Sigma \\ \{\Sigma^n \mid n \in s\} & \text{if } \lambda_e(e_{\mathcal{S}}) = s \in S \end{cases}
$$

Either the edge e maps to a character set $C \subseteq \Sigma$ and the language is all the single characters in the set C, or a skip s and the language is all the n-length strings, for every $n \in s$.

The language of an NFA is the textbook definition, where δ^* is the transitive-reflexive closure of the δ relation.

Definition B.8.

$$
\mathcal{L}[\![\mathcal{N}]\!] = \{w \mid (q_0, w, q_F) \in \delta^*_{\mathcal{N}}\}
$$

Define the language *suffix* at state $q \in Q_{\mathcal{S}}$ for a SAFA as a generalization of $\mathcal{L}[\![\mathcal{S}]\!]$ to a given start state $q \in Q_{\mathcal{S}}$ instead of initial state q_0. This definition gives us a strong induction hypothesis to use in the following lemma.

Definition B.9.

$$\mathcal{L}[\![\mathcal{S} : q]\!] = \{D \mid \text{match}_{\mathcal{S}}(q, D, 0)\}$$

Taking a transition $e \in E_{\mathcal{S}}$ prepends the language $\mathcal{L}[\![e_{\mathcal{S}}]\!]$ to all the suffixes of the destination of e. Note the concatenation operator $\cdot$ is overloaded, to mean the pairwise concatenation of the product of two sets.

Lemma B.1.

$$(q, e, q') \in \delta_{\mathcal{S}} \to \mathcal{L}[\![\mathcal{S} : q]\!] = \mathcal{L}[\![e_{\mathcal{S}}]\!] \cdot \mathcal{L}[\![\mathcal{S} : q']\!]$$

Proof: The proof proceeds by induction on the derivation $\mathcal{L}[\![\mathcal{S} : q]\!]$. In the base case and the inductive case, perform case analysis on $\mathcal{L}[\![e_{\mathcal{S}}]\!]$.

1. If $\lambda_e(e_{\mathcal{S}}) = C \subseteq \Sigma$ then the language $\mathcal{L}[\![e_{\mathcal{S}}]\!] = C$. Prepend each character in the character set C to $\mathcal{L}[\![\mathcal{S} : q']\!]$. The documents $D \in C \cdot \mathcal{L}[\![\mathcal{S} : q']\!]$ are matched by $match_E(q', e_{\mathcal{S}}, D, 0)$ for the base case and $match_E(q', e_{\mathcal{S}}, D, i+1)$ in the induction step.

2. If $\lambda_e(e_{\mathcal{S}}) = s \in S$ then the language $\mathcal{L}[\![e_{\mathcal{S}}]\!] = \{\Sigma^n \mid n \in s\}$. The documents $D \in \Sigma^n \cdot \mathcal{L}[\![\mathcal{S} : q']\!]$ are matched by $match_E(q', e_{\mathcal{S}}, D, 0)$ for the base case and $match_E(q', e_{\mathcal{S}}, D, i+1)$ in the induction step.

Now what is left is to show the substitution operation respects language equivalence between edge $e_{\mathcal{S}}$ and NFA $\mathcal{N}$.

Lemma B.2.

$$\mathcal{L}[\![\mathcal{N}]\!] = \mathcal{L}[\![e_{\mathcal{S}}]\!] \to \mathcal{L}[\![\langle \mathcal{N}/e_{\mathcal{S}} \rangle_{\mathcal{S}}]\!] = \mathcal{L}[\![\mathcal{S}]\!]$$

Proof: This is a straightforward application of lemma B.1.

We need two more auxilary lemmas. First, describe how interval set composition translates to langage union.

Lemma B.3.

$$\mathcal{L}[\![\{i_{n+1}, \overline{i_n}\}]\!] = \{\Sigma^m \mid m \in i_{n+1}\} \cup \mathcal{L}[\![\{\overline{i_n}\}]\!]$$

The second is similar, it describes how an interval set composition in the NFA construction B.5 translates to language union.

Lemma B.4.

$$\mathcal{L}[\![\mathcal{N}(\{i_{n+1}, \overline{i_n}\}, \Sigma)]\!] = \{\Sigma^m \mid m \in i_{n+1}\} \cup \mathcal{L}[\![\mathcal{N}(\{\overline{i_n}\}, \Sigma)]\!]$$

Both lemmas are straightforward to prove from their definition.

Finally, prove the NFA construction for interval sets $\mathcal{N}_I(s, \Sigma)$ has the same language as skip s for all possible skips.

Lemma B.5.

$$\lambda_e(e_s) = s \in S \to \mathcal{L}[\![\mathcal{N}(s, \Sigma)]\!] = \mathcal{L}[\![e_{\mathcal{S}}]\!]$$

Proof: For skip $s = \{\overline{i_n}\}$ prove this statement by induction on the number of intervals n.

1. For the base case, $n = 1$ as skips are non-empty sets of intervals, then $\mathcal{L}[\![\mathcal{S}(e_s)]\!] = \{\Sigma^n \mid n \in i_1\}$ and $\mathcal{N}(s, \Sigma) = \mathcal{N}_I(i_1, \Sigma)$ as only one epsilon transition is possible from $\mathcal{N}(s, \Sigma)$, the one to $\mathcal{N}_I(i_1, \Sigma)$. By inspecting the δ relations in $\mathcal{N}_c(i_1, \Sigma)$ and $\mathcal{N}_o(i_1, \Sigma)$ in B.4, both recognize exactly $\{\Sigma^n \mid n \in i_1\}$.

2. For the inductive case, $n' = n + 1$, use the auxilary lemmas B.3, B.4 to translate composition of interval sets to language union, as well as composition of interval NFA to language union. Both lemmas produce a language union with $\{\Sigma^m \mid m \in i_{n+1}\}$ which cancel out. The result is exactly satisfied by the induction hypothesis.

SAFA to AFA recursive definition The last construction that converts a SAFA to an AFA is now possible. We give a well-founded recursion procedure *unskip*, based on the number of skips in SAFA $n_{\mathcal{S}} = |\{e \mid e \in E_{\mathcal{S}}, \lambda_e(e) = s \in S\}|$ which will substitute skip n on each iteration, for $0 \leq n \leq n_{\mathcal{S}}$.

Definition B.10.

$$\begin{aligned}
&\text{unskip}(0,\ \mathcal{S}) && = \mathcal{S} \\
&\text{unskip}(n+1,\ \mathcal{S}) && = \langle \mathcal{N}(s_n, \Sigma)/s_n \rangle_{\text{unskip}(n,\mathcal{S})}
\end{aligned}$$

This procedure runs once for all $e \in E_{\mathcal{S}}$ in $\mathcal{S}$, where $\lambda_e(e) = s \in S$ is a skip and substitutes s for its equivalent NFA $\mathcal{N}(s, \Sigma)$ until there are no more skips. By this definition unskip$(n_{\mathcal{S}}, \mathcal{S})$ is an AFA.

Next to show $\mathcal{S}$ and unskip$(n_{\mathcal{S}}, \mathcal{S})$ are equivalent in terms of the regular languages they recognize, we prove

Lemma B.6.

$$\mathcal{L}[\![\mathcal{S}]\!] = \mathcal{L}[\![unskip(n_{\mathcal{S}}, \mathcal{S})]\!]$$

Proof: We proceed by induction on the number of skips $n_{\mathcal{S}}$.

1. For $n_{\mathcal{S}} = 0$, there are no skips in $\mathcal{S}$, then unskip$(0, \mathcal{S}) = \mathcal{S}$ an AFA, the automata are equal and their languages are equal.

2. For $n_{\mathcal{S}} = n + 1$, assume $\mathcal{S}_n$ is an AFA with all skips already substituted and the induction hypothesis $\mathcal{L}[\![\mathcal{S}]\!] = \mathcal{L}[\![\mathcal{S}_n]\!]$. We must prove $\mathcal{L}[\![\mathcal{S}]\!] = \mathcal{L}[\![\mathcal{S}_{n+1}]\!]$.

   (a) Let s_n the current skip to substitute, then $\mathcal{L}[\![\mathcal{S}_{n+1}]\!] = \mathcal{L}[\![\text{unskip}(n+1, \mathcal{S}_n)]\!] = \mathcal{L}[\![\langle \mathcal{N}(s_n, \Sigma)/s_n \rangle_{\mathcal{S}_n}]\!]$ by unfolding the definition of *unskip*.

   (b) The key equality to conclude the proof is by lemma B.2 $\mathcal{L}[\![\langle \mathcal{N}/s_n \rangle_{\mathcal{S}_n}]\!] = \mathcal{L}[\![\mathcal{S}_n]\!]$, provided that $\mathcal{L}[\![\mathcal{N}(s_n, \Sigma)]\!] = \mathcal{L}[\![s_n]\!]$, which we proved already B.5.

   (c) All that is left is exactly the induction hypothesis $\mathcal{L}[\![\mathcal{S}]\!] = \mathcal{L}[\![\mathcal{S}_n]\!]$ which concludes the proof.

$$\begin{aligned}
r? &= (r|\epsilon) \\
r+ &= rr^* \\
[\,\overline{a_i}\,] &= \sum_{c\,\in\,\overline{a_i}} c \\
[\hat{}\ \overline{a_i}] &= \sum_{c\,\in\,(\Sigma\setminus\overline{a_i})} c \\
r\{n\} &= r\{n,n\} \\
r\{n,\} &= r\{n,n\}r^* \\
((?{<}{=}r)s &= rs
\end{aligned}$$

FIGURE 10—Syntactic sugar rules for PCRE expansion to low-level regex.

C Compiling Regular Expressions to SAFA

We present here recursive compilation procedure from regex to SAFA, based on generalized Antimirov derivatives A.3 [30]. Assume syntactic sugar expansion D.1 and regex normalization by weak equivalence $\simeq$ is already done.

Start with a fully normalized regex r, alphabet Σ. Create an empty SAFA given alphabet Σ and states of type $B^+(r)$ then run this recursive procedure.

Given a regex r,

1. If state r exists in the SAFA, return. Otherwise add the new state r to Q.
2. Extract a skip $r \xrightarrow{s} r'$ (D.3) from r, if possible. Then s is the skip interval set and r' is the remaining regex when no more wildcards can be extracted. Label state r an $\exists$ state by $\lambda_q(r) = \exists$ and add to it a new outgoing edge e such that $(r, s, r') \in \delta$ and $\lambda_e(e) = s$. Recurse for r'.
3. Otherwise, for each character $\alpha \in \Sigma$ take the derivative of r with respect to α to be a boolean algebra expression $\partial_\alpha(r)$ A.4 in disjunctive normal form A.2 and add one transition for each character $(r, \alpha, \partial_a(r)) \in \delta$.
   (a) In DNF, the derivative $\partial_\alpha(r) = \bigvee_i(\bigwedge_j r_{i,j})$ proceed to add i existantial $\exists$ states $(\bigwedge_j r_{i,j})$ and for each j add a forall $\forall$ state $r_{i,j}$.
   (b) Then add ϵ-transitions $(\partial_\alpha(r), \epsilon, \bigwedge_j r_{i,j}) \in \delta$ for each i, as well as $(\bigwedge_j r_{i,j}, \epsilon, r_{i,j}) \in \delta$ for each j.
   (c) Recurse for each leaf state $r_{i,j}$.

Note, the number of new states added in step 3(a) is $O(|\Sigma| * i * j)$, however, in practice we noticed regex are not nested as much so $i * j$ is small.

D Preprocessing a PCRE

D.1 Syntactic sugar

All of the PCRE syntax in fig. 1 can be expressed in terms of the simpler syntax in fig. 7. Reef performs the conversion in fig. 10 as soon as possible, reducing the domain of the compiler and thus simplifying the compilation process.

D.1.1 Anchor elimination

Anchors ($\hat{}$, \$) are zero-length assertions, indicating a regex matches the begining and end of a string respectively. In preprocessing we eliminate anchors by converting a *substring* regular expression to an *exact match* regular expression.

For example, the regular expression $(?{=}a).b$ will match the suffix *ab* in *aab* but $\hat{}(?{=}a).b\$$ will only match exactly the string *ab* and not *aab*. Consider now $\hat{}. * (?{=}a).b. * \$$, it will exactly match any string that has a substring *ab* – the boolean behavior of $(?{=}a).b$ and $\hat{}. * (?{=}a).b. * \$$ are equivalent. As Reef only cares about the boolean output of matching, this regex transformation is sound.

More generally, a substring match r is transformed into an exact match $\hat{}. * r. * \$$. A start-anchored match $\hat{}r$ is transformed into an exact match $\hat{}r. * \$$. An end-anchored match $r\$$ is transformed into an exact match $\hat{}. * r\$$. Finally, an exact match $\hat{}r\$$ remains as is. In all cases we are left with an exact match over a regex, so we can remove the anchors and only implement the solving algorithm for *exact matches*.

D.2 Regular expression normalization

Equivalence of regular expressions is computationally hard. We provide a weaker syntactic equivalence for regex in Figure 12 similar to Owens [63] and reuse their result

$$\forall r,\ s,\ r \equiv s \text{ iff } r \simeq s$$

The benefit of introducing this weaker notion of equivalence [63] is we get a syntactic normalization procedure for regex. As Owens [63] shows, this normalization procedure is fast and successfully minimizes the number of states of the compiled automaton (SAFA) and thus the final step function size, which is proportional to the number of states.

D.3 Extract skips

We define the rules for extracting skips from a regex compositionally. Compositionality possible by the Boolean algebra properties of Interval sets. Then we can define the partial function $r \xrightarrow{S} r'$ that extracts skips from the head of a regex (Figure 13).

D.4 SAFA solver

At a high-level, the SAFA solver algorithm is given by the SAFA semantics in Appendix B.5. A side-note, an additional advantage of non-determinism is the room for parallelization. We take advantage of non-determinism to parallelize the SAFA solver in at least three places using a threadpool.

1. On match$_\forall$ we parallelize solving across edges $e \in E$ then join and wait on the results.

$$\frac{}{. \xrightarrow{\{[1,1]\}} \epsilon}\ \textsc{Dot} \qquad \frac{r \xrightarrow{\{\}} \epsilon}{r^* \xrightarrow{\epsilon} \epsilon}\ \textsc{Empty*} \qquad \frac{r \xrightarrow{\epsilon} \epsilon}{r^* \xrightarrow{\epsilon} \epsilon}\ \textsc{Nil*}$$

$$\frac{r \xrightarrow{s} \epsilon}{r^* \xrightarrow{\{[0,\infty)\}} \epsilon}\ \textsc{Star*} \qquad \frac{r \xrightarrow{\{\}} \epsilon}{r\{0,j\} \xrightarrow{\epsilon} \epsilon}\ \text{RE\_1}$$

$$\frac{r \xrightarrow{\{\}} \epsilon \qquad i \neq 0}{r\{i,j\} \xrightarrow{\{\}} \epsilon}\ \text{RE\_2} \qquad \frac{r \xrightarrow{\epsilon} \epsilon \qquad i \leq j}{r\{i,j\} \xrightarrow{\epsilon} \epsilon}\ \textsc{RNil}$$

$$\frac{r \xrightarrow{s} \epsilon \qquad i \leq j}{r\{i,j\} \xrightarrow{s^i \cup \ldots \cup s^j} \epsilon}\ \textsc{Range} \qquad \frac{r_1 \xrightarrow{s_1} \epsilon \qquad r_2 \xrightarrow{s_2} r'}{r_1 r_2 \xrightarrow{s_1+s_2} r'}\ \textsc{AppR}$$

$$\frac{r_1 \xrightarrow{s} r_1'}{r_1 r_2 \xrightarrow{s} r_1' r_2}\ \textsc{AppL}$$

FIGURE 13—Inference rules for a partial, recursive function $r \xrightarrow{s} r'$ extracting skip s from the head position of a regex r and returning the tail r'

$$\frac{}{\emptyset \preceq r}\ \textsc{Bot} \qquad \frac{}{r \preceq r}\ \textsc{Refl} \qquad \frac{r \preceq s \qquad s \preceq u}{r \preceq u}\ \textsc{Trans}$$

$$\frac{\alpha \in \Sigma}{\alpha \preceq .}\ \textsc{Wild} \qquad \frac{v(r) = true}{\epsilon \preceq r}\ \textsc{Nil} \qquad \frac{}{r \preceq r^*}\ \textsc{Star}$$

$$\frac{r \preceq s}{r^* \preceq s^*}\ \textsc{StarIn} \qquad \frac{j \in \mathbb{N}}{r\{0,j\} \preceq r^*}\ \textsc{Rep*}$$

$$\frac{r \preceq s \qquad i_2 \leq i_1 \qquad j_1 \leq j_2}{r\{i_1,j_1\} \preceq s\{i_2,j_2\}}\ \textsc{Rep}$$

$$\frac{r \simeq s \qquad r' \preceq s'}{rr' \preceq ss'}\ \textsc{App} \qquad \frac{r_1 \preceq r \qquad r_2 \preceq r}{r_1 \mid r_2 \preceq r}\ \textsc{AltOpp}$$

$$\frac{r \preceq r_1}{r \preceq r_1 \mid r_2}\ \textsc{AltR} \qquad \frac{r \preceq r_2}{r \preceq r_1 \mid r_2}\ \textsc{AltR}$$

$$\frac{r \preceq u}{r \,\&\, s \preceq u}\ \textsc{AndL} \qquad \frac{s \preceq u}{r \,\&\, s \preceq u}\ \textsc{AndR}$$

$$\frac{u \preceq r \qquad u \preceq s}{u \preceq r \,\&\, s}\ \textsc{AndOpp}$$

FIGURE 11—Inference rules for a partial ordering on regex, when $r \preceq s$ then r matches a subset of the language of s.

$$\frac{r \preceq s \qquad s \preceq r}{r \simeq s}\ \textsc{Eq} \qquad \frac{r \simeq s}{s \simeq r}\ \textsc{Symm}$$

$$\frac{r \preceq s \qquad s \preceq r}{r \simeq s}\ \textsc{AntiSym} \qquad \frac{}{r \simeq \epsilon r}\ \textsc{AppNilL}$$

$$\frac{}{r \simeq r\epsilon}\ \textsc{AppNilR} \qquad \frac{}{\emptyset \simeq \emptyset r}\ \textsc{AppZL} \qquad \frac{}{\emptyset \simeq r\emptyset}\ \textsc{AppZR}$$

$$\frac{}{r\{i,j\}r\{i',j'\} \simeq r\{i+i',j+j'\}}\ \textsc{AppRep}$$

$$\frac{s \preceq r}{r^* s\{0,j\} \simeq r^*}\ \textsc{StarRep} \qquad \frac{s \preceq r}{s\{0,j\} r^* \simeq r^*}\ \textsc{RepStar}$$

$$\frac{s \preceq r}{r^* s^* \simeq r^*}\ \textsc{StarStarL} \qquad \frac{r \preceq s}{r^* s^* \simeq s^*}\ \textsc{StarStarR}$$

$$\frac{r \preceq s}{r \mid s \simeq s}\ \textsc{AltR} \qquad \frac{r \preceq s}{r \mid s \simeq r}\ \textsc{AltL} \qquad \frac{}{r \simeq r\{1,1\}}\ \textsc{One}$$

$$\frac{r \simeq s \qquad i' \leq j+1}{r\{i,j\} \mid s\{i',j'\} \simeq r\{min(i,i'), max(j,j')\}}\ \textsc{AltRep}$$

FIGURE 12—Inference rules for weak regex equivalence $\simeq$.

2. On match$_\exists$ we parallelize solving across edges $e \in E$, but instead of join we race the threads. The first thread to find a solution returns and the rest are killed.

3. On match$_s$ we parallelize our search for different values of $n \in s$ and race the threads again. Even though skips s are unbounded, the document is bounded so we limit our solution search from $min(s)$ to $max(max(s), |D|)$.

The benefits of parallelization in the solver are concrete and is a contribution outside the cryptographic benefits of SAFA. Using SAFA we improve the performance of regex matching by taking advantage of non-determinism in branches and wildcards.

E Matrix Representation of R1CS

We repeat our running example of constraints over $\mathbb{F}$:

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

We would like to convert these constraints to matrices A, B, and C 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. There should only exist a solution vector $z = (io, 1, w)$, with witness $w \in \mathbb{F}^{cols-|io|-1}$ when this set of constraints is satisfiable.

In the example from Section 2.3, y is the only public variable in io. The variables $x_0, x_1, guard, tmp, prod$, and inv are known only to $\mathcal{P}$, so they make up our witness w. So $z = (y, 1, x_0, x_1, guard, tmp, prod, inv)$.

First, we shuffle some of the constraints so that each is of the form *(addition term) * (addition term) = (addition term)*:

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

We create the corresponding R1CS matrices A, B, C:

$$A = \begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0\\
0 & 1 & 0 & 0 & -1 & 0 & 0 & 0\\
0 & 1 & 0 & 0 & -1 & 0 & 0 & 0\\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}$$

$$B = \begin{bmatrix}
0 & -30 & 1 & 0 & 0 & 0 & 0 & 0\\
1 & 0 & 0 & -1 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 1 & 0 & -1 & 0 & 0\\
1 & 0 & 0 & 0 & 0 & 0 & -1 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1\\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}$$

$$C = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0\\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

Notice all of the matrices have 6 rows, since there are 6 multiplication constraints, and 8 columns, since the length of z is 8. If $\mathcal{P}$ has (for example) $x_0 = 10, x_1 = 5$, and wants to prove that $y = 2$, a vector $z = (2, 1, 10, 5, 0, 5, 2, 5^{-1})$ satisfies this R1CS instance. Note that we use 5^{-1} as the inverse of 5 in $\mathbb{F}$.

It is easy to see that this z only satisfies $(A \cdot z) \circ (B \cdot z) = (C \cdot z)$ when the assignments $y = 2$, $x_0 = 10$, $x_1 = 5$, $guard = 0$, $tmp = 5$, $prod = 2$, $inv = 5^{-1}$ satisfy our original constraints.

F Low cost padding

The obvious way to hide the size of a document D is by constructing a document D' which is equal to D but padded with dummy characters to some suitable upper bound. If the padding is chosen to be $0 \in \mathbb{F}$, then the committer $\mathcal{G}$ has to do no extra work, since $g^0 = 1$ for all generators g of the polynomial commitment scheme. However, the `nlookup` prover and the `nlookup` verifier (which is embedded within Reef's step function) still need to do work proportional to $|\mathsf{D}'|$ for each step: linear for the prover and a logarithmic number of constraints to express the verifier, plus $\mathcal{O}(|\mathsf{D}'|)$ operations at the end for *ProveEval* and $\mathcal{O}(\sqrt{|\mathsf{D}'|})$ operations for *VerifyEval* when we use the Hyrax (§6.3). If the upper bound is chosen to be large (e.g., $|\mathsf{D}'| = 2^{30}$), the cost to the prover would be prohibitive.

We observe that the same ideas in table projections that allow the prover to do less work can be used here: the prover's work during each step can be made linear in $|\mathsf{D}|$ (the unpadded document). The key observation is that given that D is a subset of D', and that padding is just 0s, it is possible for the prover to project the entries in the table corresponding to D, *without having to reveal to the verifier the selector s*. Consequently, the verifier learns nothing about D except for $|\mathsf{D}'|$, and the prover is able to save considerable costs.

The basic idea is to pad out the multilinear extension of the document with 0, commit to the multilinear extension of the larger document, $\widetilde{\mathsf{D}'}$, and run a slightly larger `nlookup` in the step function that looks to be operating over a large document, so the size of the real document is hidden. But we leverage the structure of our multilinear polynomial so that work to generate the commitment and the work of $\mathcal{P}$ to generate `nlookup` witnesses is closer to the work done in the case of the original, smaller document.

Given a multilinear extension to the original document, $\widetilde{D}(x_0, ..., x_{\ell-1})$, of length $|D| = 2^{\ell}$, we generate a multilinear extension to a larger document, of length $|D'| = 2^{\ell'}$:

$$\widetilde{D'}(p_0, ..., p_{\ell'-\ell-1}, x_0, ..., x_{\ell-1}) = \widetilde{D}(x_0, ..., x_{\ell-1})$$

$\widetilde{D'}$, will evaluate the same way $\widetilde{D}$ does on any point, "throwing away" its padding variables, $p_0, ..., p_{\ell'-\ell}$. It is committed to by inserting zeros into the multilinear extension's coefficient commitment vector for every term that includes any padding variable p_i. (This will be a predictable pattern.) Although the literal document D' (i.e. the evaluations of $\widetilde{D'}$ over the boolean hypercube) never has to be materialized, it may be helpful to visualize it. For each padding variable added to the input's of $\widetilde{D'}$, the document size doubles, and the document repeats itself.

For example, if $\widetilde{D}(x_0,x_1) = 7 + 5x_0 + 3x_1 + 2x_0x_1$, the commitment to a extension with one padding variable, $\widetilde{D'}(p_0,x_0,x_1)$ will be to the vector $[7,0,5,3,0,0,2,0]$. The final check of that $\widetilde{D'}(q_0,q_1,q_2) = v$ will be done with an inner product argument that proves $\langle[7,0,5,3,0,0,2,0],[1,q_0,q_1,q_2,q_0q_1,q_0q_2,q_1q_2,q_0q_1q_2]\rangle = v$. The actual document D is $[7,12,10,17]$, and if materialized, D' would be $[7,12,10,17,7,12,10,17]$.

This varies from typical projections in that our padding variables are not known to $\mathcal{V}$, since knowing the length would leak things about the length of the document.

The work to generate the commitment is the same as if we did not have padding—any generator exponentiated by 0 is 1. So $\mathcal{G}$ does not have to do extra exponentiations or multiplications for this larger commitment.

When producing sumcheck witnesses (as part of producing nlookup witnesses), $\mathcal{P}$ has to calculate evaluations over $\widetilde{D'}$ of the form:

$$\begin{aligned}&\widetilde{D'}(r_0,...,r_{i-1},x,b_{i+1},...,x_{\ell'-1})\\ &r_i \in \mathbb{F}\\ &x \in \{0,1\}\\ &b_i \in \{0,1\}\end{aligned}$$

Instead, it can calculate evaluations over $\widetilde{D}$. Because of the structure of $\widetilde{D'}$, the evaluations over $\widetilde{D}(x_0,...,x_{\ell-1})$ can be calculated once, and reused to mimic evaluations over $\widetilde{D'}(p_0,...,p_{\ell'-\ell-1},x_0,...,x_{\ell-1})$, no matter what the values of $p_0,...,p_{\ell'-\ell-1}$. This ends up being $\mathcal{O}(|D| + \log(\frac{|D'|}{|D|}))$ work, instead of $\mathcal{O}(|D'|)$. The log factor covers any doubling of the precalculated $\widetilde{D}$ evaluations that have to be done to pad "extra" nlookup rounds (since there are now $\log(|D'|)$ rounds in the step function).

At the end of our protocol, $\mathcal{V}$ must verify a claim of the form $\widetilde{D'}(q_r) = v_r$, where $q_r \in \mathbb{F}^{\ell'}$. This is done in the usual way using an inner product argument and our commitment to $\widetilde{D'}$, and implies consistency of all of our lookups with the original $\widetilde{D}$.

G Implementation Optimizations

To leverage the power of nlookup, Reef processes b-sized *batches* of multiple characters within each step function. This results in having to perform $\frac{|D|}{b}$ folds, as opposed to $|D|$ folds.

Readers might question the utility of batching, since the size of the step function would obviously increase with the batch size. This increases the work done by both $\mathcal{P}$ and $\mathcal{V}$. However, as the batch size increases, the number of steps decreases. The relationship between the size of the final compressed zkSNARK, the batch size, and the number of steps is not linear and requires careful tuning since its dependent on the evaluation method used in the step function as well as the complexity of the regex itself.

Rather than using the hash chain stack construction, Reef represents a stack using a vector of field elements and a stack pointer field element:

```
field[stack_size+1] push(
  field[2*stack_size] stack,
  field stack_ptr, (field child, field cursor)) {
    for i in stack_size {
    if i == stack_ptr {
      stack[i] = (child, cursor);
      stack_ptr += 1;
    }
  }
  return {stack, stack_ptr};
}

field[stack_size+3] pop(field[2*stack_size] stack,
  field stack_ptr) {

  for i in stack_size {
    if i == stack_ptr {
      (popped_child, popped_cursor) = stack[i];
      stack_ptr -= 1;
    }
  }
  return {stack, stack_ptr, popped_child,
    popped_cursor};
}
```

The stack needs to be big enough to accommodate all of the children for all of the nested forall nodes on any particular path. This number, stack_size is calculated during the step function compilation. This is usually more efficient than a hash chain stack.

Additionally, since pushes to and pops from the stack only need to happen under certain conditions (encountering a forall state or finishing transversal of a branch), it is a waste of constraints to include pop constraints and multiple sets of push constraints for every lookup in the batch. Reef instead uses constraints that may perform a single pop or several pushes during only the first lookup. If during witness generation, $\mathcal{P}$ needs to perform a pop/push and does not currently have access to the correct (first) lookup, it is allowed to "loop" on the current state, consuming ϵ characters, until the lookup constraints are available. Obviously, if the batch size is set badly, this could become inefficient. We choose batch size carefully; during table generation, Reef walks over the SAFA in a depth-first search, and takes note of the length of paths between forall nodes and accepting states. The batch size is the average length of these paths. Users of Reef can also override this mechanism and set the batch size themselves.

Reef also optimizes the solving/proving pipeline; $\mathcal{P}$'s solver runs in parallel with the thread that produces the folded cryptographic proof for $\mathcal{P}$. The solver thread calculates a witness for step i and hands it off to the prover, which is able to focus on proving step i while the solver moves on to generating witnesses for step $i+1$.

H Applications

Here we recount the full results from our experimental evaluation of Reef for our motivating applications. We start by discussing the origin and rationale behind our test data.

Password Strength We randomly generated our good password set. Our bad password set was selected at random from the NordPass list of the top 200 most common passwords [11], which is a list of weak passwords. Our regex indicates strong passwords, of a certain length, with required characters from several different fields (uppercase and lowercase alphabet characters, numbers, and special characters).

Email Redactions For our redactions, we use the Enron email dataset [12]. Our small instance is their smallest instance, and our large instance was randomly selected. Our regexes indicate redacted versions of both.

ODoH Blocklisting We use a regex filter for Pi-hole [9], which is a DNS sinkhole, for our oblivious DNS over HTTPS regexes. While blocklisting would traditionally prove non-matching, to better compare to existing work we instead prove matching. Our queries are randomly generated.

Genetic Matching For our evaluation we consider three common mutations of the BRCAI and BRCAII genes. Mutations in these genes are commonly linked to most forms of breast cancer. The base pairs for these genes, as well as for common mutations are all publicly available from the US National Institutes of Health [13, 14].

Full results. The results are in Figures 14–16.

Application	Document ID	Regex ID	R1CS Constraints	Document Length	# Steps	Compile Time (s)	Solving Time (s)	Proving Time (s)	Verifying Time (s)	Proof Size (KB)
Redactions										
	Small Email	r1	52,631	415	4	340.953	1.243	3.580	0.521	33.665
	Large Email	r2	54,636	1,000	10	550.010	5.005	7.012	0.534	33.793
ODoH										
	5f558	r3	18,437	128	3	75.168	0.106	2.084	0.378	32.401
	25424	r4	22,692	128	2	58.029	0.215	1.912	0.420	34.193
	55824	r5	23,148	128	1	132.777	0.050	1.533	0.402	33.105
	21d97	r6	18,409	128	2	34.247	0.044	1.785	0.377	32.273
	49b9a	r7	18,433	128	2	34.688	0.045	1.783	0.376	32.273
	b8f74	r8	18,263	128	2	57.819	0.042	1.774	0.368	32.273
	3b4ed	r9	17,177	128	2	26.310	0.033	1.753	0.369	32.273
	24448	r10	18,865	128	2	18.105	0.045	1.777	0.382	32.273
	b329c	r11	18,241	128	2	29.502	0.040	1.770	0.373	32.273
	6f74a	r12	18,241	128	2	37.131	0.041	1.769	0.371	32.273
	83a9c	r13	17,785	128	2	63.528	0.037	1.761	0.368	32.273
	5410f	r14	17,617	128	1	32.646	0.020	1.478	0.366	32.273
	4ad7a	r15	17,365	128	1	26.982	0.019	1.442	0.370	31.953
	b4ebd	r16	17,617	128	3	43.684	0.049	2.083	0.370	32.273
Passwords										
Match	dcdc9	r17	19,982	12	5	67.999	0.096	2.868	0.401	32.433
	43db4	r17	19,982	12	5	46.988	0.096	2.870	0.386	32.433
	91edc	r17	19,982	12	5	99.937	0.100	2.855	0.386	32.433
	2bcf2	r17	19,982	12	5	69.578	0.096	2.866	0.387	32.433
	10bf0	r17	19,982	12	5	41.517	0.097	2.884	0.387	32.433
	aff42	r17	19,982	12	5	56.667	0.094	2.874	0.382	32.433
	edde7	r17	19,982	12	5	89.487	0.099	2.883	0.389	32.433
	1539c	r17	19,982	12	5	43.381	0.100	2.869	0.384	32.433
	7bfcc	r17	19,982	12	5	75.974	0.096	2.836	0.383	32.433
	dfa02	r17	19,982	12	5	89.154	0.103	2.870	0.382	32.433
Non-Match	e73ee	r17	20,728	8	7	49.026	0.432	3.594	0.404	33.265
	b5f3a	r17	20,728	8	6	48.109	0.373	3.228	0.397	33.265
	fd1e7	r17	20,725	6	6	31.847	0.365	3.238	0.404	33.265
	db267	r17	20,725	3	5	25.403	0.305	2.902	0.394	33.265
	40867	r17	20,728	8	6	54.598	0.365	3.205	0.398	33.265
	f4a98	r17	20,725	6	6	73.322	0.365	3.170	0.400	33.265
	7474f	r17	20,728	8	6	44.740	0.385	3.241	0.406	33.265
	b20ef	r17	20,725	6	6	131.589	0.370	3.228	0.397	33.265
	27ba9	r17	20,728	7	6	40.408	0.369	3.202	0.399	33.265
	304b5	r17	20,728	9	6	31.937	0.377	3.252	0.407	33.265
DNA										
Match	BRCA1 Var1	r18	37,182	43,054,295	2	152.790	1.823	20.952	0.829	296.033
	BRCA1 Var2	r19	62,296	43,054,295	4	306.740	3.420	23.633	0.884	296.129
	BRCA2 Var1	r20	85,352	32,325,508	8	252.596	8.037	18.254	0.886	165.745
Non-Match	BRCA1 Var1	r19	38,885	43,054,295	1	261.704	1.726	20.454	0.835	296.097
	BRCA1 Var2	r18	62,821	43,054,295	1	297.474	2.354	22.475	0.887	296.161
	BRCA2 Primary	r20	95,916	32,325,508	1	509.691	3.111	12.449	0.931	165.809

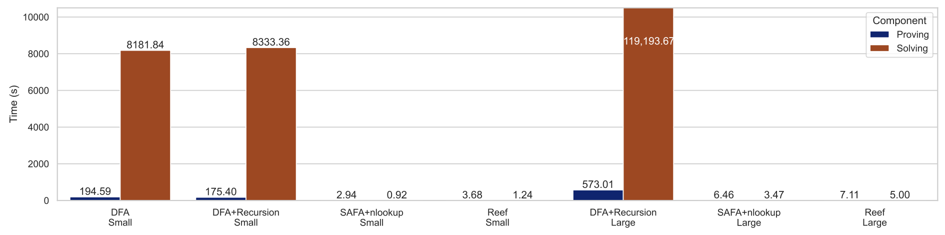
FIGURE 14—Summary of all costs for all applications evaluated in Reef. R1CS Constraints are for the step function in Nova. Times are averaged across 10 runs, standard deviation was less than 5% for all components and applications.

Application	Regex ID	SAFA States	SAFA Transitions	DFA States	DFA Transitions
Redactions					
	r1	331	42,318	433	55,424
	r2	908	116,751	1,013	129,664
ODoH					
	r3	28	3,232	94	12,032
	r4	36	4,012	33	4,224
	r5	30	3,238	25	3,200
	r6	12	1,421	11	1,408
	r7	15	1,808	14	1,792
	r8	20	2,453	19	2,432
	r9	16	1,937	15	1,920
	r10	13	1,550	12	1,536
	r11	11	1,292	10	1,280
	r12	11	1,292	10	1,280
	r13	12	1,421	11	1,408
	r14	8	905	7	896
	r15	10	1,163	9	1,152
	r16	14	1,679	13	1,664
Passwords					
	r17	21	1,188	—	—
DNA					
	r18	331	42,318	43,052,484*	172,209,936*
	r19	331	42,318	43,050,383*	172,201,532*
	r20	976	4,861	32,318,453*	129,273,812*

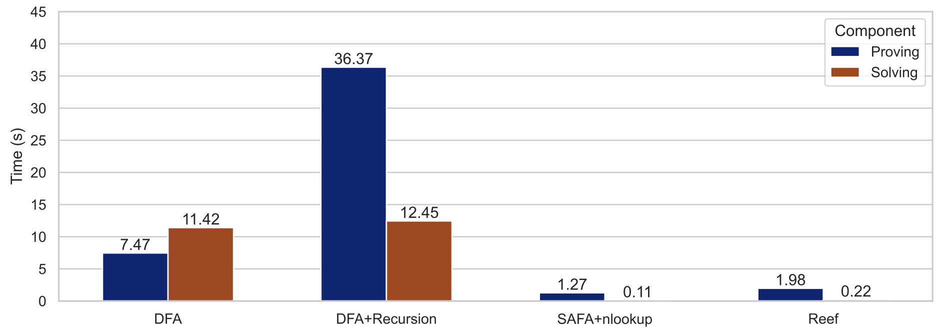
FIGURE 15—SAFA size vs DFA size for all evaluated regex
* are estimates

Application	Document ID	Regex ID	DFA	DFA # Foldings	DFA + Recursion	DFA + Recursion # Foldings	SAFA + nlookup	SAFA + nlookup # Foldings	Reef	Reef # Foldings
Redactions										
	Small Email	r1	23,041,771	1	67,472	415	54,679	4	52,631	4
	Large Email	r2	—	—	141,712	1,000	57,268	10	54,636	10
ODoH										
	5f558	r3	1,552,754	1	24,131	128	22,573	3	18,437	3
	25424	r4	553,295	1	16,288	128	25,129	2	22,692	2
	55824	r5	422,219	1	15,260	128	25,576	1	23,148	1
	21d97	r6	192,831	1	13,456	128	22,193	2	18,409	2
	49b9a	r7	241,983	1	13,840	128	22,217	2	18,433	2
	b8f74	r8	323,903	1	14,480	128	22,094	2	18,263	2
	3b4ed	r9	258,367	1	13,968	128	21,009	2	17,177	2
	24448	r10	209,215	1	13,584	128	22,020	2	18,865	2
	b329c	r11	176,447	1	13,328	128	21,138	2	18,241	2
	6f74a	r12	176,447	1	13,328	128	21,749	2	18,241	2
	83a9c	r13	192,831	1	13,456	128	21,305	2	17,785	2
	5410f	r14	127,295	1	12,944	128	20,515	1	17,617	1
	4ad7a	r15	155,141	1	13,200	124	20,589	1	17,365	1
	b4ebd	r16	225,599	1	13,712	128	21,149	3	17,617	3
Passwords										
Match	dcdc9	r17	—	—	—	—	21,002	5	19,982	5
	43db4	r17	—	—	—	—	21,002	5	19,982	5
	91edc	r17	—	—	—	—	21,002	5	19,982	5
	2bcf2	r17	—	—	—	—	21,002	5	19,982	5
	10bf0	r17	—	—	—	—	21,002	5	19,982	5
	aff42	r17	—	—	—	—	21,002	5	19,982	5
	edde7	r17	—	—	—	—	21,002	5	19,982	5
	1539c	r17	—	—	—	—	21,002	5	19,982	5
	7bfcc	r17	—	—	—	—	21,002	5	19,982	5
	dfa02	r17	—	—	—	—	21,002	5	19,982	5
Non-Match	e73ee	r17	—	—	—	—	21,721	7	20,728	7
	b5f3a	r17	—	—	—	—	21,721	6	20,728	6
	fd1e7	r17	—	—	—	—	21,401	6	20,725	6
	db267	r17	—	—	—	—	21,401	5	20,725	5
	40867	r17	—	—	—	—	21,721	6	20,728	6
	f4a98	r17	—	—	—	—	21,401	6	20,725	6
	7474f	r17	—	—	—	—	21,721	6	20,728	6
	b20ef	r17	—	—	—	—	21,401	6	20,725	6
	27ba9	r17	—	—	—	—	21,721	6	20,728	6
	304b5	r17	—	—	—	—	21,721	6	20,728	6
DNA										
Match	BRCA1 Var1	r18	—	—	—	—	44,698	2	37,182	2
	BRCA1 Var2	r19	—	—	—	—	71,818	4	62,296	4
	BRCA2 Var1	r20	—	—	—	—	96,296	8	85,352	8
Non-Match	BRCA1 Var1	r19	—	—	—	—	46,650	1	38,885	1
	BRCA1 Var2	r18	—	—	—	—	72,343	1	62,821	1
	BRCA2 Primary	r20	—	—	—	—	107,184	1	95,916	1

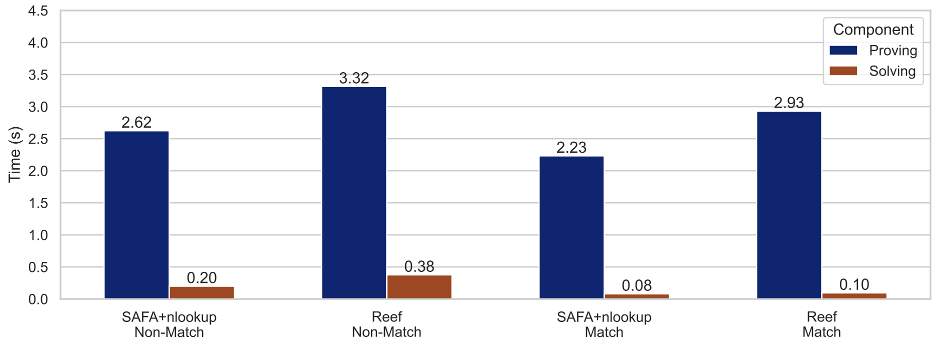
FIGURE 16—Total number of R1CS constraints for DFA, number for step function for DFA+Recursion, SAFA+nlookup, and Reef



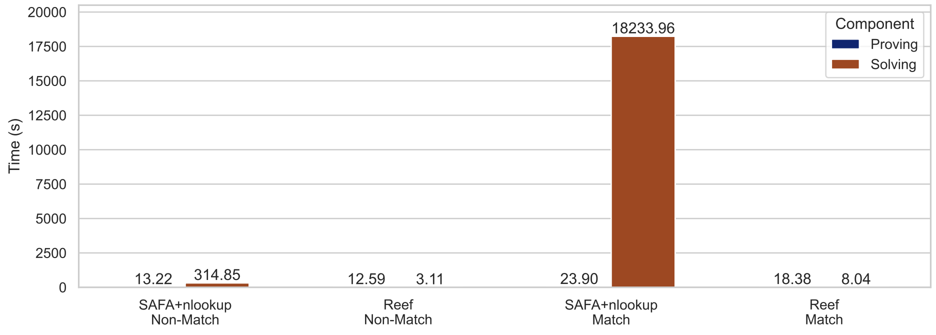
(a) Proof that a (small / large) committed email matches a redaction regex. DFA and DFA+recursion were unable to finish within 12 hours for the large email.



(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 17—Mean proving and solving time 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 5.

Application	Regex ID	Regex
Redactions		
	r1	∧ Message-ID: .*[[:space:]] Date: Tue, 8 May 2001 09:16:00 -0700 (PDT) [[:space:]] From: .* [[:space:]] To: .* [[:space:]] Subject: Re: [[:space:]] Mime-Version: 1.0 [[:space:]] Content-Type: text/plain; charset=us-ascii [[:space:]] Content-Transfer-Encoding:7bit [[:space:]] X-From: Mike Maggi [[:space:]] X-To: Amanda Huble[[:space:]]X-cc: [[:space:]]X-bcc: [[:space:]] X-Folder:\\ Michael_Maggi_Jun2001 \\ Notes Folders \\ Sent[[:space:]]X-Origin: Maggi-M[[:space:]]X-FileName: mmaggi.nsf[[:space:]]*at 5:00$
	r2	∧ Message-ID: .*[[:space:]]Date: Tue, 11 Jul 2000 11:11:00 -0700 (PDT)[[:space:]]From: .*[[:space:]]To: .*[[:space:]] Subject: Reimbursement of Individually Billed Items[[:space:]]Mime-Version: 1.0[[:space:]]Content-Type: text/plain; charset=us-ascii[[:space:]]Content-Transfer-Encoding: 7bit[[:space:]]X-From: Enron Announcements[[:space:]]X-To: All Enron Employees North America[[:space:]]X-cc: [[:space:]]X-bcc: [[:space:]]X-Folder: \\Michelle_Lokay_Dec2000_June2001_1\\Notes Folders\\Corporate[[:space:]]X-Origin: LOKAY-M[[:space:]]X-FileName: mlokay\.nsf[[:space:]]*The memo distributed on June 27 on Reimbursement of Individually Billed Items [[:space:]]requires[[:space:]]clarification\. The intent of the memo was to give employees an alternate [[:space:]]method[[:space:]]of paying for pagers, cell phones, etc\. Employees can continue to submit[[:space:]]these[[:space:]]invoices to Accounts Payable for processing or pay these items with their [[:space:]]corporate[[:space:]]American Express card and request reimbursement through an expense report\. [[:space:]]Either[[:space:]]way is an acceptable way to process these small dollar high volume invoices\.$
ODoH		
	r3	∧ad([sxv]?[0 − 9] ∗ |system)[_.−]([∧.[[: space :]] + .){1, }|[_.−]ad([sxv]?[0 − 9] ∗ |system)[_.−]|$
	r4	∧(. + [_.−])?adse?rv(er?|ice)?s?[0 − 9] ∗ [_.−]
	r5	∧(. + [_.−])?telemetry[_.−]
	r6	∧(adim(age|g)s?[0 − 9] ∗ [_.−]
	r7	∧(adtrack(er|ing)?[0 − 9] ∗ [_.−]
	r8	∧(advert(s|is(ing|ements?))?[0 − 9] ∗ [_.−]
	r9	∧(aff(iliat(es?|ion))?[_.−]
	r10	∧(analytics?[_.−]
	r11	∧(banners?[_.−]
	r12	∧(beacons?[0 − 9] ∗ [_.−]
	r13	∧(ount(ers?)?[0 − 9] ∗ [_.−]
	r14	∧(mads.
	r15	∧(pixels?[−.]
	r16	∧(stat(s|istics)?[0 − 9] ∗ [_.−]
Passwords		
	r17	∧(? = . ∗ [A − Z]. ∗ [A − Z])(? = . ∗ [!%∧@#$&∗])(? = . ∗ [0 − 9]. ∗ [0 − 9])(? = . ∗ [a − z]. ∗ [a − z]. ∗ [a − z]).{12}$
DNA		
	r18	∧.{43052424}ATGGGCTACAGAAACCGTGCCAAAAGACTTCTACAGAGTGAACCCGAAAATCCTTCCTTG
	r19	∧.{43050079}ATGCTGAAACTTCTCAACCAGAAGAAAGGGCCTTCACAGTGTCCTTTATGTAAGAATGATATAACCAAAAG. ∗ AGCCTACAAGAAAGTACGAGATTTAGTCAACTTGTTGAAGAGCTATTGAAAATCATTTGTGCTTTTCAGCTTGACACAGGTTTGGAGT. ∗ ATGCAAACAGCTATAATTTTGCAAAAAAGGAAAATAACTCTCCTGAACATCTAAAAGATGAAGTTTCTATCATCCAAAGTATGGGCTACAGAAACCGTGCCAAAAGACTTCTACAGAGTGAACCCGAAAATCCTTCCTTG
	r20	∧.{32317478}CACAACTAAGGAACGTCAAGAGATACAGAATCCAAATTTTACCGCACCTGGTCAAGAATTTCTGTCTAAATCTCATTTGTATGAACATCTGACTTTGGAAAAATCTTCAAGCAATTTAGCAGTTTCAGGACATCCATTTTATCAAGTTTCTGCTACAAGAAATGAAAAAATGAGACACTTGATTACTACAGGCAGACCAACCAAAGTCTTTGTTCCACCTTTTAAAACTAAATCACATTTTCACAGAGTTGAACAGTGTGTTAGGAATATTAACTTGGAGGAAAACAGACAAAAGCAAAACATTGATGGACATGGCTCTGATGATAGTAAAAATAAGATTAATGACAATGAGATTCATCAGTTTAACAAAAACAACTCCAATCAAGCAGTAGCTGTAACTTTCACAAAGTGTGAAGAAGAACCTTTAG. ∗ ATTTAATTACAAGTCTTCAGAATGCCAGAGATATACAGGATATGCGAATTAAGAAGAAACAAAGGCAACGCGTCTTTCCACAGCCAGGCAGTCTGTATCTTGCAAAAACATCCACTCTGCCTCGAATCTCTCTGAAAGCAGCAGTAGGAGGCCAAGTTCCCTCTGCGTGTTCTCATAAACAG. ∗ CTGTATACGTATGGCGTTTCTAAACATTGCATAAAAATTAACAGCAAAAATGCAGAGTCTTTTCAGTTTCACACTGAAGATTATTTGGTAAGGAAAGTTTATGGACTGGAAAAGGAATACAGTTGGCTGATGGTGGATGGCTCATACCCTCCAATGATGGAAAGGCTGGAAAAGAAGAATTTTATAG. ∗ GGCTCTGTGTGACACTCCAGGTGTGGATCCAAAGCTTATTTCTAGAATTTGGGTTTATAATCACTATAGATGGATCATATGGAAACTGGCAGCTATGGAATGTGCCTTTCCTAAGGAATTTGCTAATAGATGCCTAAGCCCAGAAAGGGTGCTTCTTCAACTAAAATACAG

FIGURE 18—Regexs with ID