

# Coinductive Proofs of Regular Expression Equivalence in Zero Knowledge

JOHN C. KOLESAR, Yale University, USA

SHAN ALI, Yale University, USA

TIMOS ANTONOPOULOS, Yale University, USA

RUZICA PISKAC, Yale University, USA

Zero-knowledge (ZK) protocols enable software developers to provide proofs of their programs' correctness to other parties without revealing the programs themselves. Regular expressions are pervasive in real-world software, and zero-knowledge protocols have been developed in the past for the problem of checking whether an individual string appears in the language of a regular expression, but no existing protocol addresses the more complex PSPACE-complete problem of proving that two regular expressions are equivalent.

We introduce Crêpe, the first ZK protocol for encoding regular expression equivalence proofs and also the first ZK protocol to target a PSPACE-complete problem. Crêpe uses a custom calculus of proof rules based on regular expression derivatives and coinduction, and we introduce a sound and complete algorithm for generating proofs in our format. We test Crêpe on a suite of hundreds of regular expression equivalence proofs. Crêpe can validate large proofs in only a few seconds each.

CCS Concepts: • **Theory of computation** → **Automated reasoning**; **Program verification**; **Regular languages**; **Cryptographic protocols**.

Additional Key Words and Phrases: Regular Expressions, Zero-Knowledge, Language Equivalence

## ACM Reference Format:

John C. Kolesar, Shan Ali, Timos Antonopoulos, and Ruzica Piskac. 2025. Coinductive Proofs of Regular Expression Equivalence in Zero Knowledge. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 285 (October 2025), 33 pages. <https://doi.org/10.1145/3763063>

## 1 Introduction

Software verification is the process of translating a program into a mathematical formalism and then constructing a proof that the formalized program satisfies some specification. Traditional verification tools do not allow programmers to hide the implementation details of their software from other parties. The tools require all relevant information to be fully visible to all parties involved in the verification process, so they are inadequate for verification problems involving third-party libraries, cloud-based services, or other proprietary software: the owners of the software cannot share the evidence of the program's correctness with other parties without losing their privacy.

To make verification of private programs possible, recent research at the intersection of cryptography and formal verification has focused on the development of zero-knowledge (ZK) protocols for encoding proofs of programs' correctness [50, 54, 55]. A zero-knowledge protocol is a cryptographic method of communication that allows one party to demonstrate knowledge of a secret to another party without revealing the secret [40, 41]. The ZK program verification process involves two

---

Authors' Contact Information: John C. Kolesar, Yale University, New Haven, USA, john.kolesar@yale.edu; Shan Ali, Yale University, New Haven, USA, shan.ali@yale.edu; Timos Antonopoulos, Yale University, New Haven, USA, timos.antonopoulos@yale.edu; Ruzica Piskac, Yale University, New Haven, USA, ruzica.piskac@yale.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/10-ART285

<https://doi.org/10.1145/3763063>

parties: the *prover* who owns the private program and the *verifier* whom the prover wants to convince of the program’s correctness. (In the domain of ZK protocols, the terms “prover” and “verifier” have different meanings than they do in the domain of traditional formal verification.) The prover constructs a proof of the program’s correctness offline and then engages in an interactive zero-knowledge protocol with the verifier. During the protocol, the prover provides evidence that suffices to convince the verifier that the program is correct but does not reveal additional information about the program’s implementation.

To be efficient and scalable, ZK protocols must be tailored for specific domains. Existing ZK verification protocols have targeted Boolean logic [55], quantifier-free first-order logic with EUF and LIA [54], and Lean’s calculus of dependent types [50], but currently no ZK protocol exists for the problem of regular expression equivalence. Regular expressions are a common formalism for defining string patterns, and their uses in modern software are pervasive [21, 65, 68]. In particular, the applications of regular expression equivalence include correctness proofs for compiler optimizations [48], translation validation [36], and query optimization [23]. Moreover, ZK proofs about regular expression equivalence are valuable because regular expressions encode security and privacy constraints for software systems. Content filtering applications employ regular expressions extensively for tasks such as spam detection [1] and malware link blocking [62]. More recently, generative AI systems have used regular expressions to filter queries and outputs. For instance, Apple Intelligence reportedly relies on regular expressions as an initial layer of content filtering within its on-device models [17]. Additionally, regular expressions play a key role in intrusion detection and prevention for network packets [75].

Regular expressions for content filtering often contain sensitive information. In the case of Apple Intelligence, the regular expression filters used include the names of political figures and other politically charged terminology [16]. The specific choices of phrases to reject could harm Apple’s reputation if made public. Similarly, public disclosure of an intrusion detection system’s regular expressions can be harmful: if attackers know the filters, they can learn to circumvent them.

At the same time, content filters may be subject to regulations, and the systems’ developers may need to demonstrate to an auditor that their filters are at least as restrictive as a public standard specification [25, 35]. The developers cannot simply use membership proofs for individual strings in a private regular expression’s language: the public specification can require a filter to match entire classes of strings, and the classes may be infinite. Consequently, programmers can benefit from a ZK protocol for reasoning about the full languages of private regular expressions.

Importantly, we cannot reformulate the problem of regular expression equivalence in terms of problems that existing ZK regular expression protocols handle. Existing protocols [6, 56, 61, 74] reason only about the problem of matching individual strings against regular expressions. Regular expression equivalence is a PSPACE-complete problem [66, 67], and string matching is at most NP-complete if generalized [2], so we cannot reduce regular expression equivalence to string matching unless  $P=PSPACE$  or  $NP=PSPACE$ . In fact, no existing practical ZK protocol targets a PSPACE-complete problem at all, even though it has been known for decades that the complexity class IP (the set of all problems that can be solved with interactive ZK proofs) is equal to PSPACE [64].

In this paper, we introduce the first practical ZK protocol for the problem of regular expression equivalence. To implement our new ZK protocol, we developed a tool called Crêpe (Cryptographic Regex Equivalence Proof Engine) that validates regular expression equivalence proofs in ZK. When two parties communicate using Crêpe, the owner of a hidden proof that two regular expressions are equivalent can convince the other party that the proof is valid without leaking the regular expressions or the proof. For the proofs that Crêpe validates, we introduce a new custom calculus of proof rules. Instead of converting regular expressions into state machines or other objects, as standard solvers do [12, 76], our rules apply derivatives and coinduction to reason about regular

expressions directly, eliminating the need to validate a conversion between different formats. Also, we introduce a new decision procedure to generate the proofs that Crêpe validates. Given two equivalent regular expressions as input, our decision procedure constructs an equivalence proof for them in our custom calculus of proof rules. The decision procedure creates cleartext proofs, and Crêpe transmits the proofs from one party to the other in a ZK format. We cannot rely on an existing solver to generate proof certificates for us: although older tools can check whether two regular expressions are equivalent, they have only limited support for regular expression proof certificates. Modern SMT solvers can reason about other formalisms effectively, but there is no well-established format for providing proof certificates about regular expression equivalence analogous to the existing formats for Boolean logic, EUF, and LIA [26]. We prove that our rules are sound and complete for regular expression equivalence.

In summary, we make four main research contributions:

- (1) **ZK Protocol.** We introduce Crêpe, the first zero-knowledge protocol designed to validate regular expression equivalence proofs and also the first practical ZK protocol to target a PSPACE-complete problem.
- (2) **Proof Generator.** We develop a backend for Crêpe that generates regular expression equivalence proofs in a custom calculus, and we prove the calculus sound and complete, which is of independent interest.
- (3) **Proof Generator Evaluation.** We test our custom proof generator on a suite of commonly-used regular expression benchmarks from FlashRegex [53]. The regular expressions range from 2 to 60 characters in length. Our proof generator scales to handle large, complex inputs, and it generates equivalence proofs successfully for almost all of the equivalent pairs in the suite.
- (4) **ZK Protocol Evaluation.** We test Crêpe on a suite of hundreds of equivalence proofs from our custom proof generator. Crêpe can validate 84.39 percent of the proofs in 15 seconds or less. Also, we run Crêpe on the same suite with alternative configurations that change the amount of information that it leaks. Our default settings provide strong safety guarantees and incur only a 35% median slowdown relative to a version that leaks more information.

## 2 Motivating Example

As we mentioned in Section 1, the developers of a generative AI system may need to demonstrate to an auditor that their content filter is at least as restrictive as a public specification for filtering. Let  $p$  be a public regular expression for the specification, and let  $q$  be the private regular expression for the content filter. The developers can convince the auditor that  $q$  is at least as restrictive as the specification by proving in ZK that  $p|q = q$ . If  $p|q = q$ , then the language of  $q$  is a superset of the language of  $p$ . A regular expression for filtering accepts a string if that string is an unsafe input that the system should reject, so, in the context of content filtering,  $p|q = q$  means that the filter flags any input that the specification requires it to flag and can also flag other inputs as well.

To illustrate the technique, we will use the regular expressions  $aa^*$  and  $a^*$  as proxies for  $p$  and  $q$ . In practice,  $p$  and  $q$  would be much more complex, and Crêpe can handle complex inputs, as we demonstrate in our evaluation (Section 8). However, we choose to present a simple case for the sake of intelligibility. By using Crêpe and our sound and complete custom calculus of rules, we can show in ZK that the language of  $a^*$  contains the language of  $aa^*$ . The protocol involves two parties: the *prover* who owns both regular expressions and the *verifier* whom the prover wants to convince that the regular expressions are equivalent without revealing the regular expressions. First, the prover needs to create an equivalence proof offline. The proof relies on derivatives [22] and coinduction [63]. Informally, derivatives represent the changes that happen in a regular expression's behavior as it reads characters from an input string. For instance, the derivative of  $ab|ac$  with

StepID	RuleID	Premises	Result
#1	Derive		$\delta(a, aa^* a^*) = a^* a^*$
#2	Derive		$\delta(a, a^* a^*) = a^* a^*$
#3	Derive		$\delta(a, a^*) = a^*$
#4	Epsilon		$E(aa^* a^*) = \epsilon$
#5	Epsilon		$E(a^* a^*) = \epsilon$
#6	Epsilon		$E(a^*) = \epsilon$
#7	Eq	{#4, #6}	$E(aa^* a^*) = E(a^*)$
#8	Eq	{#5, #6}	$E(a^* a^*) = E(a^*)$
#9	SyncCycle	{#2, #3}	$\text{Sync}(a, a^* a^*, a^*)$
#10	Coinduction	{#8, #9}	$\text{Sync}(\epsilon, a^* a^*, a^*)$
#11	SyncEmpty	{#10}	$a^* a^* = a^*$
#12	Cong	{#1, #11}	$\delta(a, aa^* a^*) = a^*$
#13	Cong	{#3, #12}	$\delta(a, aa^* a^*) = \delta(a, a^*)$
#14	Match	{#7, #13}	$aa^* a^* = a^*$

Table 1. A proof of  $aa^*|a^* = a^*$  in Crêpe’s format. Some steps are omitted or simplified. We use # to denote proof step addresses.

respect to  $a$  is  $b|c$ . Coinduction is a proof technique that allows us to take advantage of cyclic patterns in the regular expressions’ derivatives. A proof of the equivalence of  $aa^*|a^*$  and  $a^*$  appears in Table 1. In the first few steps, we examine the derivatives of  $aa^*|a^*$  and  $a^*$ . Later, we find that  $a^*|a^*$  and  $a^*$  retain their behavior when they read an additional  $a$  from an input string, so we apply coinduction to the cycle. We know from our coinductive step that  $a^*|a^*$  and  $a^*$  agree on any number of repetitions of  $a$ , and neither regular expression accepts strings with characters other than  $a$ , so the two regular expressions must be equivalent. We then work backward from the equivalence of  $a^*|a^*$  and  $a^*$  to confirm that  $aa^*|a^* = a^*$ . We explain derivatives and coinduction in more detail in Section 3. A more detailed explanation of the proof appears in Section 4.

After generating an equivalence proof, the prover can allow the verifier to validate the proof in ZK. When the prover and verifier communicate using Crêpe, their interaction consists of a series of steps. In each step, the prover provides the verifier with evidence that an individual proof step is valid. The evidence corresponds to a single row of the proof step table: the verifier learns that the proof step is valid but cannot see the specific operations that the step performs. When the prover and verifier finish validating every individual step and also confirm some other necessary properties such as the absence of cyclic pointers (Section 4), the verifier knows that the proof as a whole is valid. We discuss our ZK operations in more detail in Section 6.

### 3 Preliminaries

#### 3.1 Grammar

The three main categories of entities in our formalism for Crêpe are strings ( $s$ ), terms ( $p$ ), and formulas ( $\varphi$ ). We give inductive definitions for all three:

$$\begin{aligned}
 s &::= \epsilon \mid cs_1 & p &::= \emptyset \mid \epsilon \mid c \mid p_1p_2 \mid p_1|p_2 \mid p_1^* \mid E(p_1) \mid \delta(c, p_1) \\
 & & \varphi &::= p_1 = p_2 \mid \text{Sync}(s_1, p_1, p_2) \mid p_1 \neq p_2 \mid \perp
 \end{aligned}$$

A string is simply a list of characters from an alphabet  $\Sigma$ . Starting from the empty string  $\epsilon$ , we build longer strings inductively by adding characters to the front of shorter ones. Sometimes we

write  $sc$  to denote the string  $s$  with  $c$  added to the back. Note that we use  $c$  to denote the individual character  $c$ , the one-character string whose only character is  $c$ , and also the regular expression that accepts only the string  $c$ . Additionally, we use  $\epsilon$  to denote both the empty string and the regular expression that accepts only the empty string.

Terms represent regular expressions and functions over regular expressions. Every term defines a *language*: the set of strings that it accepts.  $\emptyset$  is the regular expression whose language is empty: it accepts no strings.  $\epsilon$  accepts the empty string and nothing else. The regular expression  $c$  accepts the one-character string  $c$  and nothing else, where  $c$  is a character from  $\Sigma$ .  $p_1|p_2$  is the union of terms  $p_1$  and  $p_2$ : it accepts all strings accepted by either  $p_1$  or  $p_2$ .  $p_1p_2$  is the concatenation of  $p_1$  and  $p_2$ : it accepts all strings that consist of a part that  $p_1$  accepts followed by a part that  $p_2$  accepts. Lastly,  $p_1^*$  accepts zero or more repetitions of  $p_1$ .  $E(p_1)$  and  $\delta(c, p_1)$  are functions over terms, where  $c$  is a character. We say that a term is a *regular expression* if it does not contain any AST nodes of the form  $E(p)$  or  $\delta(c, p)$ .

A *formula*  $\varphi$  is an assertion about terms that has a truth value. Every proof step has a formula as its conclusion. We support four predicates for formulas. Equality ( $p = q$ ) means that two terms have the same language. The Sync predicate ( $\text{Sync}(s_1, p_1, p_2)$ ) serves as an indicator of incremental progress toward a proof that two terms are equivalent. Informally,  $\text{Sync}(s_1, p_1, p_2)$  means that, if  $p_1$  and  $p_2$  disagree on a string that starts with  $s_1$ , they also disagree on a strictly shorter string. We will explain the meaning of Sync in more depth in Section 5.1. Inequality ( $p_1 \neq p_2$ ) is simply the negation of equality. Lastly, bottom ( $\perp$ ) indicates that a proof has reached a contradiction. All of our proofs conclude  $\perp$  in the final step.

*Epsilon.* As part of our formalism, we include an *epsilon* function  $E$ . If  $p$  is a term, then  $E(p)$  indicates whether  $p$  accepts the empty string  $\epsilon$ . More specifically,  $E(p)$  returns  $\epsilon$  if  $p$  accepts  $\epsilon$  and returns  $\emptyset$  otherwise. Having  $E$  return a term rather than a Boolean value allows us to avoid introducing an extra type into our formalism.

*Derivatives.* If  $p$  is a term and  $c$  is a character, then  $\delta(c, p)$  is the *derivative* of  $p$  with respect to  $c$ : the term whose language is the set of strings  $s$  such that  $p$  accepts  $cs$  [22]. We can generalize the definition of derivatives to strings once we have a definition for individual characters. For the base case,  $\delta(\epsilon, p) = p$  for any  $p$ . For the inductive case,  $\delta(sc, p) = \delta(c, \delta(s, p))$  for a string  $s$  and character  $c$ . Note that the derivative nodes for individual characters are the only derivative nodes that exist within our abstract syntax trees (ASTs): we introduce this notation only for the sake of readability.

### 3.2 Regular Expression Equivalence

Two regular expressions are equivalent if and only if they have the same language. Regular expression equivalence is a decidable problem and is PSPACE-complete [66, 67]. Decision procedures for regular expression equivalence have existed for decades [7, 44], but the task of generating regular expression equivalence proofs that can be validated by parties other than the prover has received comparatively little attention. From a certain perspective, this is unsurprising: since the problem is PSPACE-complete, validating a proof of two regular expressions' equivalence is asymptotically just as costly as finding the proof in the first place. However, when privacy concerns arise, the ability to validate a regular expression equivalence proof constructed by another party becomes valuable.

*Custom Proof Format.* We use a custom backend to generate regular expression equivalence proofs for Crêpe. We do not use an existing solver to generate proofs because existing SMT solvers provide only limited support for reasoning about regular expressions. SMTInterpol [26], the solver used by the ZK protocol ZKSMT for proof generation [54], does not support regular expressions at all. CVC5 can check whether an individual string appears in the language of a regular expression,

but its rules for reasoning about regular expression equivalence are not complete [49]. Existing solvers that are complete for regular expression equivalence are not suitable backends for Crêpe either. The standard technique for comparing regular expressions to each other is to convert the regular expressions into state machines. The SMT solver Z3 uses this approach [12, 76], and its algorithm is complete for regular expression equivalence [27]. However, Z3 cannot always provide full evidence for the correctness of the SAT/UNSAT answers that it returns. Z3 does not support a full axiomatization of the theory of regular expressions in terms of rules for proof certificates [15]. When Z3 applies theory-specific reasoning for regular expression equivalence or other non-axiomatized theories, it represents the process in its proof certificates as a black-box term rewriting step [31]. All reasoning in a zero-knowledge proof needs to be explicit, so opaque term rewriting steps make Z3's proof certificates unusable in ZK. Moreover, Z3 gained the ability to provide certificates for any regular expression pair whose equivalence it can prove only recently, in parallel to the development of Crêpe. Before it received the extension, Z3's support for proof certificates about regular expression equivalence was even more limited. Previously, when the user requested a proof rather than only a SAT/UNSAT answer, Z3 lost completeness for regular expression equivalence, and it managed to provide certificates only for simple inputs [14].

Crêpe's regular expression equivalence proofs do not involve state machine conversion. Instead, they operate on regular expressions directly by using equations and algebraic data types. If we used proofs based on state machines, the proof would need to contain evidence of the correctness of our conversion of each starting regular expression into a corresponding state machine. Since our proofs deal with regular expressions directly, they eliminate the need to validate conversions between different formats.

State machine conversion is not the only existing alternative to direct reasoning about regular expressions. Quantified Boolean formula (QBF) satisfiability is a PSPACE-complete problem, just like regular expression equivalence is [39]. A quantified Boolean formula is a Boolean formula that can contain universal and existential quantifiers applied to Boolean variables. All problems in PSPACE are solvable in ZK by reduction to QBF solving, with a translation based on Turing machines [64]. QBF solving is a well-studied problem [13], but we choose not to translate regular expressions into quantified Boolean formulas for the same reason that we choose not to translate them into state machines: we would need to prove the correctness of the translation.

### 3.3 Coinduction

*Coinduction* serves as the backbone of Crêpe's custom proof format. Coinduction is a proof technique analogous to induction for reasoning about potentially infinite data structures. Instead of using base cases and inductive steps to prove that a property holds for all finite structures of a specific type, a coinductive proof constructs a *bisimulation* between two objects to demonstrate that they uphold a property and continue to uphold the property after any number of reductions. For our purposes, the two infinite objects being compared are the paths that two regular expressions' derivatives can take, and the property that they continue to hold after any number of reductions is the derivatives' equivalence. We use the Sync predicate to construct the bisimulation gradually.

Coinduction is a more suitable technique for regular expression equivalence proofs than induction is. Regular expressions are defined inductively, but the derivative function for regular expressions does not make gradual progress toward a base case. For a regular expression  $p$  and character  $c$ , there is no guarantee that  $\delta(c, p)$  will contain fewer AST nodes than  $p$  itself. For instance,  $\delta(a, (ab)^*)$  is  $b(ab)^*$ , which is larger than  $(ab)^*$ . Although the derivative function is not guaranteed to reach a terminating case, it is guaranteed to reach a fixed point or cycle eventually since the derivatives of a regular expression must fall into a finite number of equivalence classes [22]. We can utilize the cycles for our coinductive proofs. If the two regular expressions being compared behave equivalently

up to the point where they reach a cycle, further repetitions of the cycle will not cause them to behave differently. Consequently, when we find cycles in the derivatives of two regular expressions, we can work backward from the cycles to prune the search space until we know that the regular expressions align on all strings.

### 3.4 Zero-Knowledge Proofs

A zero-knowledge protocol is a method of communication between two parties, known as the *prover* and the *verifier*. Both parties know a public predicate  $P$ , and the prover possesses a private *witness*  $w$ . The prover's goal is to demonstrate to the verifier that  $w$  satisfies  $P$  without revealing the value of  $w$  [40, 41]. The protocol may leak some information about  $w$ , but the verifier should not be able to make incremental progress toward recovering the entirety of  $w$  by executing the protocol a large number of times. ZK protocols can be either interactive or non-interactive. In an interactive ZK protocol, the prover and verifier both send messages to each other iteratively, and the security of the protocol comes from the fact that the verifier makes random requests for information about  $w$  that the prover cannot anticipate [18]. In a non-interactive ZK protocol, a fixed certificate from the prover takes the place of the verifier's random requests and the prover's responses [32]. Crêpe is an interactive protocol, and so are multiple existing ZK protocols for validation of logical proofs [54, 55], but one is non-interactive [50].

In the context of Crêpe, the witness  $w$  is a logical deduction showing that two regular expressions are equivalent. The public predicate  $P$  is the assertion that the deduction is valid. Importantly, the word “proof” has two different senses in the domain of ZK verification. Logical proofs and ZK proofs are separate concepts. For our purposes, the prover wants to provide a zero-knowledge proof of the existence of a logical proof.

More specifically, Crêpe is a *commit-and-prove* ZK protocol. Commit-and-prove protocols use a technique known as *commitment* to conceal witnesses [24]. Commitment prevents the verifier from seeing the true underlying value of a witness, and it also binds the prover to a specific choice for what the underlying value is. This *binding* property distinguishes commitment from encryption. Like a secure hash function, a secure commitment scheme is not efficiently reversible and maps different witnesses to the same committed value only rarely. A cheating prover would need to identify collisions reliably and, consequently, has only a negligible chance of deceiving the verifier [30]. On the other hand, for encryption, collisions are desirable: in a perfect information-theoretically secure encryption scheme, any message can map to any ciphertext [57]. The feature that distinguishes a commit-and-prove protocol from other uses of commitment is that the prover never needs to reveal the committed values to the verifier. Instead, the prover proves that the committed values uphold specific properties [24]. Within a ZK proof, the verifier can see that the results of all operations on a committed object are consistent with each other. In our proofs for Crêpe, we perform addition, multiplication, and comparison operations on committed integers. Our formalism is not tied to any specific commit-and-prove system, but, for our implementation, we use a recently developed VOLE-ZK backend [70].

As is standard for two-party ZK protocols, our threat model does not require any assumption of limitations on the abilities of an adversarial prover [70]. The prover can attempt to lie, but our ZK backend on its own makes the prover's probability of successfully falsifying a witness negligible. Similarly, our backend makes the verifier's probability of successfully learning the underlying values of the prover's commitments negligible [30].

Addr.	NodeID	Imm	Pointers	Meaning
&0	Empty		{}	$\emptyset$
&1	Blank		{}	$\epsilon$
&2	Char	$a$	{}	$a$
&3	Star		{&2}	$a^*$
&4	Concat		{&2, &3}	$aa^*$
&5	Union		{&3, &3}	$a^* a^*$
&6	Union		{&4, &3}	$aa^* a^*$
&7	Epsilon		{&3}	$E(a^*)$
&8	Epsilon		{&5}	$E(a^* a^*)$
&9	Derivative	$a$	{&3}	$\delta(a, a^*)$
&10	Derivative	$a$	{&6}	$\delta(a, aa^* a^*)$

Table 2. Part of the term table  $M_t$  for the proof of the equivalence of  $aa^*|a^*$  and  $a^*$ . We use & to denote the addresses of terms.

## 4 Protocol Design

*Proof Structure.* Many aspects of Crêpe’s design resemble the designs of ZKSMT [54] and zkPi [50], two other ZK protocols for encoding proofs. An instance of Crêpe is a virtual machine with read-only memory, a set of checking instructions, and a list of proof steps. Our proofs involve three types of inductively-defined structures: terms, strings, and formulas. We represent all three as ASTs, where each inductive constructor is a single AST node. An instance of Crêpe includes three read-only tables for storing them. The *term table*  $M_t$  contains all of the terms in a proof. Each entry in the table represents an AST node, and nodes store pointers to their immediate children within the table. The *string table*  $M_s$  contains the strings used for Sync predicates. We represent strings as singly-linked lists. Each node stores a single character, namely the one at the front of the string, and also stores a pointer to the remainder of the list. One entry in the string table is a null terminator that represents  $\epsilon$ . All chains of linked list nodes in  $M_s$  ultimately lead to the null terminator. The *formula table*  $M_f$  contains the conclusions of all of the proof steps in the proof. In our setup, formulas do not contain pointers to other formulas, but they can build on top of terms and strings. In that event, a formula’s entry in  $M_f$  contains pointers to entries from  $M_t$  and  $M_s$ .

Portions of the term table, formula table, and string table for our example proof from Section 2 appear in Tables 2, 3, and 4, respectively. In our tables for AST nodes, the field NodeID indicates the specific constructor used in our grammar for an entry. In  $M_t$ , an entry’s NodeID denotes the kind of term node that it represents. In  $M_f$ , the NodeID is the predicate that a formula uses. Entries in  $M_s$  have no NodeID because they are all strings. Also, entries in  $M_t$  and  $M_s$  have a field Imm that represents the character stored at a node. Imm is an immediate value rather than a pointer to some other location. For all three AST tables, pointers are stored in the field Pointers, regardless of whether they point to other entries in the same table or to entries in different tables.

Along with the three tables for AST nodes, there is a fourth read-only table, the *step table*  $M_p$ . Table 1 from Section 2 is an example of a step table. Each entry in  $M_p$  represents a step in the proof. The steps in  $M_p$  do not necessarily appear in their underlying logical order. A step table entry includes an index indicating its logical order within the proof, an ID for the rule it uses, an ID for that rule’s multiplexing category (Section 6), a pointer to the step’s conclusion in  $M_f$ , and pointers to the step’s premises in  $M_f$ . The formula table and the step table always have the same number of entries. We keep the two tables distinct to make ZK execution of Crêpe more efficient (Section 6).

Addr.	NodeID	Pointers	Meaning
%0	Eq	{&9, &3}	$\delta(a, a^*) = a^*$
%1	Eq	{&7, &1}	$E(a^*) = \epsilon$
%2	Eq	{&5, &3}	$a^* a^* = a^*$
%3	Sync	{\$1, &5, &3}	$\text{Sync}(a, a^* a^*, a^*)$
%4	Sync	{\$0, &5, &3}	$\text{Sync}(\epsilon, a^* a^*, a^*)$

Table 3. Part of the formula table  $M_f$  for the proof of the equivalence of  $aa^*|a^*$  and  $a^*$ . We use % to denote the addresses of formulas.

Addr.	Imm	Pointers	Meaning
\$0	$\epsilon$	{}	$\epsilon$
\$1	$a$	{\$0}	$a$
\$2	$b$	{\$1}	$ba$

Table 4. The string table for the proof of the equivalence of  $aa^*|a^*$  and  $a^*$ . We use \$ to denote the addresses of strings. We include an extra row to show how we represent multi-character strings.

In the proof in Table 1, different steps perform different functions depending on their rules. Steps for unfolding derivatives and epsilon applications appear at the start of the proof. The step labeled SyncCycle takes advantage of the fact that  $a^*|a^*$  and  $a^*$  are both their own derivatives with respect to  $a$  to begin the creation of a bisimulation. The step labeled Coinduction continues the construction of the bisimulation, and the step labeled SyncEmpty converts the bisimulation into an equality predicate. At the end, we conclude that  $aa^*|a^* = a^*$ .

Proofs in Crêpe’s calculus start with a single assumption, but apart from that, there are no contexts, environments, or temporary assumptions for proof steps. Whenever we derive a conclusion, we know that it holds unconditionally under the starting assumption.

*Checking Instructions.* Crêpe relies on a calculus of more than forty distinct proof rules. Every proof rule has its own corresponding *checking instruction*. A checking instruction is a function that confirms that an individual application of a specific proof rule is valid. Most checking instructions only perform simple pattern matching on the premises and conclusion of a proof step. For instance, the checking instruction for Trans (transitivity of equality) starts by fetching the step’s premises  $\varphi_1$  and  $\varphi_2$  and its conclusion  $\varphi_0$  from  $M_f$ . It asserts that all three formulas have Eq as their NodeID. Next, the checking instruction asserts some pointer equalities:  $\varphi_1.\text{Pointers}[0] = \varphi_0.\text{Pointers}[0]$ ,  $\varphi_1.\text{Pointers}[1] = \varphi_2.\text{Pointers}[0]$ , and  $\varphi_2.\text{Pointers}[1] = \varphi_0.\text{Pointers}[1]$ . These assertions suffice to confirm that a proof step derives a conclusion of the form  $x = z$  from premises of the form  $x = y$  and  $y = z$ . Importantly, the checking instruction never needs to fetch  $x$ ,  $y$ , or  $z$  from  $M_f$ . To confirm that two occurrences of  $x$ ,  $y$ , or  $z$  are equal, it only needs to check that the same pointer appears in both formulas. The Pointers entries in all three formulas are pointers to the term table for  $x$ ,  $y$ , and  $z$ . In general, when we need to check for equality between two terms or strings, we only check pointer equality. Sometimes we perform more complex linear-time scans on strings or chains of derivatives, but not for mere equality checking.

*Size Parameters.* We use five numerical size parameters and one set to define the characteristics of an instance of Crêpe. All of them are public, and each one relates to a different component of the proof. (1)  $n$  is the number of distinct characters in the alphabet  $\Sigma$ . (2)  $\chi$  is the number of entries in  $M_t$ . (3)  $\xi$  is the number of entries in  $M_s$ . (4)  $\pi$  is the number of entries in  $M_f$ . (5)  $\nu$  is the

**Algorithm 1:** Crêpe $[\mathcal{T}](n, \chi, \xi, \pi, \nu)$ 


---

```

1 D ← [0, …, 0];
2 for pc = 0 to π - 1 do
3   Proof Step Fetch:
4   ψ0 ← Mp[pc];
5   T ←  $\mathcal{T}[\psi_0.\text{CatID}]$ ;
6   Conclusion Fetch:
7   φ0 = Mf[ψ0.Res];
8   Premise Fetch:
9   ψ1, ψ2 ← Mp[ψ0.Preams[0]], Mp[ψ0.Preams[1]];
10  φ1, φ2 ← Mf[ψ1.Res], Mf[ψ2.Res];
11  Rule Checking:
12  z ← false;
13  for τ ∈ T do
14    | z ← z ∨ CheckingInstrs[τ](φ0, {φ1, φ2});
15  assert(z);
16  Cycle Checking:
17  assert(ψ1.StepID < ψ0.StepID);
18  assert(ψ2.StepID < ψ0.StepID);
19  D[pc] ← ψ0.StepID;
20 PermuteCheck(D, [0, …, π - 1]);
21 ConsistencyCheck(Mt, Ms);

```

---

maximum length of any individual string stored in  $M_s$ . Lastly, (6)  $\mathcal{T}$  is a set of sets of checking instructions that Crêpe can use. Every set  $T$  within  $\mathcal{T}$  is a distinct category for multiplexing. We discuss multiplexing in more detail in Section 6.

*Execution.* Algorithm 1 shows how Crêpe validates equivalence proofs. Crêpe iterates over the entries in the step table, checking that each one is valid. At the end, we know that the whole proof is valid because every individual step is valid.

Crêpe confirms that proof steps are valid by applying checking instructions to them. Rather than applying only one checking instruction to each step, Crêpe multiplexes over a set of checking instructions (lines 5 and 11–15). Only one of the checking instructions needs to return a positive result for a given proof rule. Multiplexing allows us to hide the specific proof rule used for a step, which becomes important when we validate proofs in ZK (Section 6).

There is no need to perform type checking for our table of terms because all terms have the same type. Likewise, there is no need to perform type checking for our table of formulas  $M_f$  or our table of strings  $M_s$ . However, we do need to perform some consistency checks on the tables apart from the checks performed by checking instructions. We confirm that there are no more than  $n$  distinct characters, and we confirm that there are no cyclic pointers in any of the tables (lines 16–18 and 21). Also, at the end, we confirm that every proof step has been checked once (line 20).

## 5 Proof Rules

We will now explain the calculus of rules that Crêpe employs in its proofs. Most of the rules are based on existing work, but we introduce some custom rules for coinduction. Our proof rules are

sound and complete for regular expression equivalence. Our coinduction rules are the only ones whose soundness is non-trivial to establish, and we confirm their soundness as we introduce them. Our completeness proof appears in Appendix C.

Our work is not the first to apply coinduction to Kleene algebra. A decision procedure for NetKAT program equivalence based on coinduction appears in [36]. The procedure uses state machine conversion, and it produces only a Boolean result rather than a proof tree. Also, an axiomatization of regular expression equivalence that resembles our calculus of rules appears in [28]. Like our calculus, the axiomatization relies on derivatives and equations rather than state machine conversion. However, there is no accompanying decision procedure for generating proofs with the provided axioms.

*Rule Design.* We designed all of our proof rules to be checkable in either constant time or linear time relative to  $v$ . The proof rules for ZKSMT [54] and zkPi [50] follow a similar pattern. Keeping all of our proof rules simple allows us to check them easily in ZK without applying excessive padding. If we had individual proof steps that traversed ASTs of arbitrary depth, we would need to apply a significant amount of padding to hide the size and shape of the AST being traversed. Every application of a tree-traversing rule would need to incur the same cost, and that cost would scale relative to the maximum AST size.

For a similar reason, each proof rule takes at most two premises. In Table 5 we depict the rules Coinduction and Match as taking  $n + 1$  premises, but, in the underlying implementation, we split them into multiple rules, none of which take more than two premises.

*Simple Proof Rules.* Our rules other than the coinduction rules fall into five main categories. (1) Our *epsilon rules* unfold the definition of  $E$ . Instead of traversing a whole AST, each rule unfolds the definition of the epsilon function for one AST node. They follow the standard definition of the epsilon function. (2) Our *derivative rules* unfold the definition of  $\delta$ . They follow the standard definition of the derivative function, and their design is similar to the design of the epsilon rules. (3) Our *equality rules* are standard axioms of first-order logic. They allow us to use equality as an equivalence relation and to perform substitutions of equivalent terms. (4) Our *normalization rules* are standard axioms of Kleene algebra for manipulating unions and concatenations [46]. Kleene algebra is the mathematical formalism underlying regular expressions. Within our algorithm for proof generation, we use the normalization rules to convert regular expressions into normal form (Section 7, Algorithm 3). (5) Our *proof completion rules* bookend our proofs by introducing an assumption that two regular expressions are not equivalent at the start and deriving  $\perp$  from it at the end. A more detailed explanation of our simple rules appears in Appendix A.

## 5.1 Coinduction Rules

Our coinduction rules appear in Table 5. Before we explain our coinduction rules, we need to explain the meaning of the Sync predicate that they utilize. The formal meaning of Sync depends on the concept of *reducible counterexamples*. Let  $w$  be a string on which the terms  $p$  and  $q$  disagree. We say that  $w$  is a reducible counterexample if it can be expressed in the form  $w = stu$ , where  $t$  is non-empty, such that  $\delta(s, p) = \delta(st, p)$  and  $\delta(s, q) = \delta(st, q)$ . If  $w = stu$  is reducible, then  $p$  and  $q$  also disagree on  $su$ , which is a strictly shorter string. Note that it is possible for  $su$  to be reducible as well. Also, if  $p = \delta(w, p)$  and  $q = \delta(w, q)$ , then we can define  $t$  as  $w$  and have  $s$  and  $u$  be empty. An irreducible counterexample for  $p$  and  $q$  is simply a counterexample that is not reducible. Formally,  $\text{Sync}(s, p, q)$  means that  $p$  and  $q$  do not have any irreducible counterexamples that start with  $s$ .

The intuitive meaning of our coinduction rules is closely tied to the algorithm that we use to generate proofs in our calculus (Section 7). Our coinduction rules operate by searching the tree of two regular expressions' corresponding derivatives and identifying cycles in the tree. When we find

RuleID	Premises	Conclusion
Match	$E(p) = E(q),$ $\forall c. \delta(c, p) = \delta(c, q)$	$p = q$
Coinduction	$E(\delta(s, p)) = E(\delta(s, q)),$ $\forall c. \text{Sync}(sc, p, q)$	$\text{Sync}(s, p, q)$
SyncCycle	$\delta(cs, p) = p, \delta(cs, q) = q$	$\text{Sync}(cs, p, q)$
SyncFold	$\text{Sync}(s, \delta(c, p), \delta(c, q))$	$\text{Sync}(cs, p, q)$
EqualSync	$\delta(s, p) = \delta(s, q)$	$\text{Sync}(s, p, q)$
SyncEmpty	$\text{Sync}(\epsilon, p, q)$	$p = q$

Table 5. Coinduction Rules

a cycle on a path, we can ignore any further branches on that path because a branch with the same behavior exists before the cycle repeats. Consequently, with coinduction, we reduce the infinite search space of derivatives to a finite number of combinations. Applications of our coinduction rules within a proof certificate represent the execution trace of the search.

*Main Coinduction Rules.* Match and Coinduction are the two most important rules in our calculus. Match is simply the application of a key observation from prior work: if  $p$  and  $q$  are two regular expressions with the same alphabet  $\Sigma$ , then  $p$  and  $q$  are equivalent if and only if  $E(p) = E(q)$  and  $\delta(c, p) = \delta(c, q)$  for every  $c \in \Sigma$  [7]. Match takes  $n + 1$  premises, each of which corresponds to a different part of the observation. The premise  $E(p) = E(q)$  means that  $p$  and  $q$  agree on the empty string. Also, if  $\delta(c, p) = \delta(c, q)$  for some  $c$ , then  $p$  and  $q$  agree on all strings that start with  $c$ . If  $p$  and  $q$  are not equivalent, then there must exist some string that one accepts and the other rejects. The premises  $E(p) = E(q)$  and  $\forall c \in \Sigma. \delta(c, p) = \delta(c, q)$  eliminate the possibility of  $p$  and  $q$  disagreeing on any string, so  $p$  and  $q$  must be equivalent.

Coinduction also takes  $n + 1$  premises. The first of its premises,  $E(\delta(s, p)) = E(\delta(s, q))$ , establishes that  $p$  and  $q$  agree on  $s$ . For the other  $n$  premises, we have  $\text{Sync}(sc, p, q)$  for every  $c \in \Sigma$ . Each of those premises gives us that  $p$  and  $q$  have no irreducible counterexamples that start with  $sc$ . We know from the first premise that  $s$  itself is not a counterexample for  $p$  and  $q$ , so putting all of the premises together gives us that  $p$  and  $q$  have no irreducible counterexamples that start with  $s$ . This is precisely the meaning of  $\text{Sync}(s, p, q)$ , which is the conclusion of Coinduction.

*Auxiliary Coinduction Rules.* Along with Match and Coinduction, we have some additional rules for manipulating Sync formulas. The rule SyncCycle takes advantage of cycles in terms' derivatives. If  $\delta(cs, p) = p$  and  $\delta(cs, q) = q$  for a character  $c$  and string  $s$ , then we know that  $\text{Sync}(cs, p, q)$  holds. We can say in this situation that, if  $p$  and  $q$  have a counterexample that starts with  $cs$ , that counterexample is reducible. Suppose that  $p$  and  $q$  disagree on  $cst$  for some string  $t$ . We can express  $cst$  as  $s't'u$ , where  $s' = \epsilon$  and  $t' = cs$ . With this rephrasing, our premises become  $\delta(s't', p) = \delta(s', p)$  and  $\delta(s't', q) = \delta(s', q)$ . The string  $t' = cs$  is non-empty, so these are precisely the requirements for reducibility, and  $p$  and  $q$  have no irreducible counterexamples that start with  $cs$ . In other words,  $\text{Sync}(cs, p, q)$  holds, so SyncCycle is sound.

Importantly, SyncCycle cannot be applied with an empty string. Every term is its own derivative with respect to  $\epsilon$ . If we did not enforce non-emptiness, we could derive  $\text{Sync}(\epsilon, p, q)$  trivially for any  $p$  and  $q$  and then apply SyncEmpty to derive  $p = q$ , which would be unsound.

The rule SyncFold concludes  $\text{Sync}(cs, p, q)$  from the premise  $\text{Sync}(s, \delta(c, p), \delta(c, q))$ . The soundness of SyncFold follows from the definition of the derivative:  $\delta(c, p)$  accepts the set of strings  $s$  such that  $p$  accepts  $cs$ , and  $\delta(c, q)$  accepts the set of strings  $s$  such that  $q$  accepts  $cs$ . The conclusion

$\text{Sync}(cs, p, q)$  means that  $p$  and  $q$  have no irreducible counterexamples that start with  $cs$ . Suppose that  $p$  and  $q$  do have an irreducible counterexample  $cst$  for some string  $t$ . Since  $p$  and  $q$  disagree on  $cst$ , it must hold that  $\delta(c, p)$  and  $\delta(c, q)$  disagree on  $st$ . Moreover,  $st$  must be an irreducible counterexample for  $\delta(c, p)$  and  $\delta(c, q)$ . If  $st$  can be expressed as  $s't'u$  such that  $\delta(s', \delta(c, p)) = \delta(s't', \delta(c, p))$  and  $\delta(s', \delta(c, q)) = \delta(s't', \delta(c, q))$ , then that would violate our assumption that  $cst$  is irreducible because  $cst = cs't'u$ ,  $\delta(cs', p) = \delta(cs't', p)$ , and  $\delta(cs', q) = \delta(cs't', q)$ . The fact that  $\delta(c, p)$  and  $\delta(c, q)$  have an irreducible counterexample that starts with  $s$  contradicts  $\text{Sync}(s, \delta(c, p), \delta(c, q))$ , our premise from the start. Therefore,  $p$  and  $q$  cannot have any irreducible counterexamples that start with  $cs$ , and  $\text{SyncFold}$  is sound.

The rule  $\text{EqualSync}$  allows us to conclude  $\text{Sync}(s, p, q)$  from  $\delta(s, p) = \delta(s, q)$ . To see why  $\text{EqualSync}$  is sound, consider the string  $st$  for some  $t$ . By the definition of the derivative,  $p$  accepts  $st$  if and only if  $\delta(s, p)$  accepts  $t$ . Likewise,  $q$  accepts  $st$  if and only if  $\delta(s, q)$  accepts  $t$ . Therefore,  $p$  and  $q$  agree on  $st$  if and only if  $\delta(s, p)$  and  $\delta(s, q)$  agree on  $t$ . Our premise  $\delta(s, p) = \delta(s, q)$  gives us that  $\delta(s, p)$  and  $\delta(s, q)$  agree on all strings, so  $p$  and  $q$  must agree on all strings that start with  $s$ . If  $p$  and  $q$  have no counterexamples that start with  $s$ , they have no irreducible counterexamples that start with  $s$ , so  $\text{Sync}(s, p, q)$  holds, and  $\text{EqualSync}$  is sound.

The rule  $\text{SyncEmpty}$  takes  $\text{Sync}(\epsilon, p, q)$  as a premise and derives  $p = q$ . The premise  $\text{Sync}(\epsilon, p, q)$  means that  $p$  and  $q$  have no irreducible counterexamples that start with  $\epsilon$ . All strings start with  $\epsilon$ , so  $p$  and  $q$  have no irreducible counterexamples at all. This means that  $p$  and  $q$  must be equivalent: two regular expressions cannot have a counterexample unless they have an irreducible counterexample. By definition, every reducible counterexample has a strictly shorter counterexample that can be constructed from it. All strings are finite, so it cannot be the case that all counterexamples for a pair of inequivalent regular expressions are reducible. Since  $p$  and  $q$  do not have any counterexamples, they must be equivalent, which is what we wanted to show.

## 6 Zero-Knowledge Encoding

On its own, the basic structure of an instance of Crêpe does not hide the regular expressions being compared or the proof of their equivalence. We need to use multiple zero-knowledge proof techniques to hide the structure of our proofs. We use a commit-and-prove ZK protocol [24] to hide the terms and formulas manipulated by a proof, we apply padding when validating certain proof steps, and we multiplex over different checking instructions. Crêpe is not necessarily tied to any specific cryptographic backend. Our implementation uses many of the same ZK operations as ZKSMT [54], including VOLE-ZK commitment [9, 34, 70], optimizations for polynomials [71], and oblivious RAM [37]. There are resemblances to the ZK techniques used by zkPi [50] as well.

*Table Commitment.* Most of the proof structure in an instance of Crêpe appears in the tables  $M_t$ ,  $M_s$ ,  $M_f$ , and  $M_p$ , so we commit the entries of the tables. We can represent any term, string, formula, or proof step as a vector of bits because the entries in our tables are all tuples of numbers, where those numbers either represent values on their own or serve as pointers to other table entries. We can commit every bit in a table entry individually and then combine the committed bits for an entry into a single committed integer [37]. When we commit the rows in our tables, the verifier can check equality between table entries and perform arithmetic operations on them without learning their values.

To prevent information leakage, all rows in an individual table are padded to be the same size. Every row in a table has the maximum number of entries for its type, even if some of those entries are unused. For instance, not all terms have two children, and not all formulas have pointers to  $M_s$ , but, in our ZK proofs, the nodes that do not have those fields have padding in their place. There is

no need to pad rows with fields for other types (such as pointers to  $M_s$  in the term table) because all entries in a table are of the same type.

Additionally, we use oblivious indexing for  $M_t$ ,  $M_s$ , and  $M_f$ . Crêpe does not modify its tables during execution, so we can use a ZK protocol for read-only memory for the tables [33, 37, 43]. With oblivious indexing, the verifier can use a committed integer index to retrieve a row of a table without learning the value of the index or the entry that was retrieved. This means that the verifier cannot perform frequency analysis on the memory fetches that occur during proof validation to recover extra information.

Although we commit the rows in  $M_p$ , we do not support oblivious indexing for  $M_p$ . During proof validation, every entry of  $M_p$  is fetched exactly once, so there is no need to safeguard against frequency analysis for the step table.

The amortized time cost of fetching an entry from ZK memory is linear in the bit width of the entries and does not depend on the number of rows in the memory. Our memory entries are all of constant size: they do not vary with the size parameters for a proof. Consequently, memory fetches are effectively a constant-time operation for Crêpe. ZKSMT's read-only memory has the same amortized performance [54].

*Multiset Operations.* Unlike ZKSMT, Crêpe does not have a dedicated table for storing lists. Nevertheless, the protocol needs to reason about lists and multisets during validation. For multisets, the order of elements is irrelevant, but their multiplicities matter. Whenever we need to confirm that two lists are equivalent when viewed as multisets, we convert the lists into polynomials over a finite field and use a check similar to the checks used by ZKSMT [54].

*Multiplexing.* Just like ZKSMT [54], Crêpe shuffles the steps in its proofs to hide their logical ordering. As a further guard against information leakage, Crêpe uses *multiplexing*: when it validates a proof step, it runs multiple checking instructions on the step to hide the rule used for the step.

The simplest and most secure approach for multiplexing would be to run every checking instruction on every step. However, in Section 8.4, we observe that this approach causes Crêpe to run very slowly on large proofs. Instead, we group the rules in our calculus into three main categories for multiplexing. If the rule for a proof step  $\psi$  is in the category  $T$ , then Crêpe runs every checking instruction in  $T$  on  $\psi$ . Consequently, when Crêpe runs on a proof, it leaks the number of proof steps in each multiplexing category. This is a significant improvement over the information leakage of ZKSMT, which leaks the frequencies of all distinct proof rules [54].

The checking instructions that run in linear time relative to  $\nu$  have their own multiplexing category. If we placed linear-time rules in the same category as everything else, every proof step would take linear time to check. Also, we split the constant-time checking instructions into two categories: one for Symm, Trans, and FunCong2 (Appendix A, Table 7), and the other for everything else. We have a separate category for Symm, Trans, and FunCong2 because they are consistently the most commonly used rules in the regular expression equivalence proofs that our proof generation algorithm generates. The division of constant-time rules into two categories is an arbitrary choice for performance optimization, not a requirement for avoiding information leakage. It is similar to the approach taken by zkPi [50], which separates judgments into two categories for multiplexing for the sake of efficiency.

*Multiplexing Alternatives.* More efficient implementations of ZK multiplexing exist than the version that Crêpe uses. When Crêpe multiplexes over proof rules, it runs the rules' checking instructions sequentially. Consequently, the asymptotic cost of multiplexing for Crêpe is linear in the number of proof rules within a multiplexing category. When zkPi [50] multiplexes over proof rules, it pays the cost of the most complex individual rule within a multiplexing category.

Other existing techniques for ZK multiplexing achieve a similar result [72]. The Tight ZK CPU, a newer approach for ZK multiplexing, pays only the cost of the actual instruction used, not the most complex one [73]. Crêpe’s ZK backend does not support any of these improved versions of ZK multiplexing currently, but Crêpe does use some optimizations to eliminate redundant operations without increasing its information leakage. For instance, when two different checking instructions need to fetch the same entry from ZK memory for the same step, we cache the entry rather than retrieving it twice.

*Proof Step Ordering.* Like ZKSMT [54], Crêpe performs a permutation check to ensure that every proof step has been checked once. Every proof step has a committed ID, and a proof step cannot take other steps as premises unless those steps’ IDs are strictly lower than its own (Algorithm 1, lines 17–18). Every time Crêpe validates a proof step, it adds that step’s ID to the array D. At the end of validation, Crêpe confirms that every proof step’s ID appears in D exactly once. We can confirm this in linear time by performing a multiset equivalence check between D and the list of all step IDs from 0 to  $\pi - 1$  (Algorithm 1, line 20). This technique for checking that one list is a permutation of another comes from [19] originally, and ZKSMT uses it as well [54].

*Information Leakage.* All of Crêpe’s size parameters ( $n$ ,  $\chi$ ,  $\xi$ ,  $\pi$ , and  $\nu$ ) are public. Along with the size parameters, Crêpe leaks the number of proof steps in each multiplexing category, but it does not leak the specific rule used for each step. In contrast, ZKSMT leaks the number of uses of each proof rule [54]. The fact that we store terms, strings, and formulas in three different tables is not a source of information leakage. Even if we stored all three in the same table, the verifier would always know whether an entry being fetched at a given point during execution is a term, string, or formula since the verifier knows the definitions of the checking instructions.

On its own, Crêpe does not require the prover to reveal the regular expressions being proven equivalent. However, the prover can selectively leak information about one or both of them to ground the proof in a larger system. Depending on the use case, the verifier can demand information about the starting regular expressions to prevent the prover from providing an irrelevant proof. Also, as part of the task of grounding a ZK proof in a larger system, the prover may need to demonstrate that a compiled binary actually uses the regular expressions verified by Crêpe. The prover would not need to provide a new ZK proof every time the program runs, but the grounding would require different evidence than what Crêpe provides on its own. Tools such as translation validators [58, 60] and verified compilers [51] can help to provide the guarantees that a full ZK program verification pipeline would require. The incorporation of these tools into a ZK program verification pipeline is a direction for future work.

Side channel attacks, such as timing attacks, are out of scope for Crêpe. Nevertheless, the protocol is not tied to any specific cryptographic backend, so nothing prevents us from using a ZK backend that provides side channel protections.

*Knowledge Soundness and Zero-Knowledge.* The fact that Crêpe upholds knowledge soundness and zero-knowledge follows immediately from the fact that its cryptographic backend does [70]. The verifier validates every ZK operation in a proof individually, so there are no opportunities for the prover to falsify a ZK operation or for the verifier to learn anything about the ZK operations other than the number of operations performed. If we multiplex over all checking instructions for every step (Section 8.4), the verifier gains no information from the relative ordering of the ZK operations: the size parameters  $n$ ,  $\chi$ ,  $\xi$ ,  $\pi$ , and  $\nu$  are the only information leaked. This is analogous to the information leakage of ZKUNSAT [55] and zkPi [50]. At the opposite extreme, if every proof rule has its own multiplexing category, Crêpe leaks the frequencies of each distinct rule in a proof in the same way that ZKSMT does [54]. Also, for any configuration of the multiplexing categories,

**Algorithm 2:** Equivalence check  $\text{EQUIV}(H, p, q)$ 


---

```

1  $p' \leftarrow \text{normalize}(p)$ ;
2  $q' \leftarrow \text{normalize}(q)$ ;
3 if  $p' == q'$  then
4   |  $\text{return true}$ ;
5 if  $(p', q') \in H$  then
6   |  $\text{return true}$ ;
7  $e_1 \leftarrow \text{normalize}(\text{reduce}(E(p')))$ ;
8  $e_2 \leftarrow \text{normalize}(\text{reduce}(E(q')))$ ;
9 if not  $(e_1 == e_2)$  then
10  |  $\text{return false}$ ;
11  $v \leftarrow \text{true}$ ;
12 for  $c \in \Sigma$  do
13  |  $p_c \leftarrow \text{reduce}(\delta(c, p'))$ ;
14  |  $q_c \leftarrow \text{reduce}(\delta(c, q'))$ ;
15  |  $v \leftarrow v \wedge \text{EQUIV}(H \cup \{(p', q')\}, p_c, q_c)$ ;
16  $\text{return } v$ ;
```

---

the prover can always make the size parameters larger than they need to be to reduce information leakage even further. Moreover, the information that Crêpe leaks for a given execution depends only on the input proof and the multiplexing configuration. Since the information leakage does not vary randomly between executions, the verifier cannot gain extra information about a proof by executing the protocol more than once on the same input.

## 7 Proof Generation Process

Now we will explain our process for generating proofs that Crêpe can validate. We use two algorithms: the first checks whether two regular expressions are equivalent, and the second generates an equivalence proof for two equivalent regular expressions.

### 7.1 Equivalence Checking

Algorithm 2 checks whether two regular expressions have the same language. EQUIV is a reformulation of an algorithm from [4] in our syntax. The algorithm from [4] is itself based on decision procedures from [7] and [3]. At a high level, EQUIV is simply an exhaustive search over the derivatives of the two regular expressions being compared. It terminates its exploration of a path when it finds a cycle or a case where  $p$  and  $q$  disagree. EQUIV uses the accumulator set  $H$  to detect cycles:  $H$  stores the derivative pairs that it has encountered previously. Different recursive paths have their own versions of  $H$ , so derivative pairs are flagged as repeats only if they appeared previously in the path currently being explored. At the start of execution,  $H$  should be empty.

EQUIV is guaranteed to terminate because of a result from prior work [22]. For a regular expression  $r$ , the set of all unfolded derivatives  $\text{reduce}(\delta(s, r))$  for strings  $s$  can be infinite, but those derivatives must fall into a finite number of equivalence classes for similarity. Similarity is an equivalence relation: two regular expressions are similar if one can be converted into the other using our normalization and equality rules (Appendix A). The operator  $==$  checks whether two terms are syntactically identical. The function `reduce` unfolds all derivative and epsilon nodes in a term using our proof rules. The function `normalize` returns a normalized version of a regular expression: it

**Algorithm 3:** Proof generation PROOF( $H, s, p, q$ )

---

```

1 ( $p', \psi_p$ )  $\leftarrow$  normalize( $p$ );
2 ( $q', \psi_q$ )  $\leftarrow$  normalize( $q$ );
3 if  $p' == q'$  then
4   |  $\psi_r \leftarrow$  Refl( $p', q'$ );
5   |  $\psi'_r \leftarrow$  Subst( $\psi_r, \psi_p, \psi_q$ );
6   | return EqualSync( $\psi'_r$ );
7 if  $\exists s_0. (p', q', s_0) \in H$  then
8   |  $\psi_h \leftarrow$  SyncCycle(suffix( $s_0, s$ ),  $p', q'$ );
9   | return Subst( $\psi_h, \psi_p, \psi_q$ );
10 ( $e_1, \psi_1$ )  $\leftarrow$  normalize(reduce( $E(p')$ ));
11 ( $e_2, \psi_2$ )  $\leftarrow$  normalize(reduce( $E(q')$ ));
12  $\psi_e \leftarrow$  Refl( $e_1, e_2$ );
13  $\psi'_e \leftarrow$  Subst( $\psi_e, \psi_1, \psi_2$ );
14  $\Psi \leftarrow [\psi'_e]$ ;
15 for  $c \in \Sigma$  do
16   | ( $p_c, \psi_3$ )  $\leftarrow$  reduce( $\delta(c, p')$ );
17   | ( $q_c, \psi_4$ )  $\leftarrow$  reduce( $\delta(c, q')$ );
18   |  $\psi_c \leftarrow$  PROOF( $H \cup \{(p', q', s)\}, sc, p_c, q_c$ );
19   |  $\Psi \leftarrow \Psi :: \psi_c$ ;
20  $\psi_s =$  Coinduction( $\Psi$ );
21 return Subst( $\psi_s, \psi_p, \psi_q$ );

```

---

maps all regular expressions in a similarity class to the same unique normal form. Our version of similarity is more general than the version in [22], but making similarity more inclusive preserves the result. Because the sets of similarity classes for the derivatives of  $p$  and  $q$  are finite, the recursion in Algorithm 2 is guaranteed to find a cycle with  $H$  eventually on any path it takes.

## 7.2 Proof Generation

To generate proofs of two regular expressions' equivalence rather than simply finding a Boolean result, we extend Algorithm 2. Algorithm 3 follows the same strategy as EQUIV, but it records its path exploration in the form of a proof. PROOF does not check on its own that  $p$  and  $q$  are equivalent, but it produces a valid proof if they are. It has an accumulator set  $H$  that serves the same purpose as the one in EQUIV. Additionally, it uses an accumulator string  $s$  that should be  $\epsilon$  initially. The string records all of the derivative characters used so far to reach the current point in the algorithm's execution, and we use that information for our coinduction rules.

The versions of `normalize` and `reduce` used in Algorithm 3 return proofs that their main output is equivalent to their input. The function `suffix( $s_0, s$ )` returns the characters at the end of  $s$  that do not appear in  $s_0$ . We know that  $s_0$  must be a prefix of  $s$ , so the application of the function within Algorithm 3 is always safe. The function `Subst( $\psi, \psi_1, \psi_2$ )` creates a new proof tree where the equivalence proven by  $\psi_1$  is used as a substitution on the left-hand side of  $\psi$  and the equivalence proven by  $\psi_2$  is used as a substitution on the right-hand side of  $\psi$ . We can perform the substitution using our equality rules. Lastly, `Refl` is one of our equality rules.

Algorithm 3 is guaranteed to terminate if its two input regular expressions are equivalent. If  $p$  and  $q$  are equivalent, Algorithm 3 follows the same execution path as Algorithm 2 in terms of recursion

and branching. In the underlying implementation of PROOF, we use some performance optimizations not shown in Algorithm 3. For instance, we use Match instead of Coinduction whenever  $\Psi$  contains only equality formulas and no Sync formulas.

*Post-Processing.* Algorithm 3 often generates distinct proof steps that have the same conclusion. A proof that derives the same conclusion multiple times is redundant, so, after the algorithm finishes, we perform a second pass to eliminate redundancies. For any step  $\psi$  with  $\mu$  premises, let  $\text{size}(\psi) = 1 + \sum_{i=0}^{\mu-1} \text{size}(\psi.\text{Premis}[i])$ . (If  $\psi$  has no premises, then  $\text{size}(\psi) = 1$ .) If multiple distinct proof steps  $\psi_1, \dots, \psi_k$  within the proof returned by Algorithm 3 have the same conclusion, let  $\hat{\psi} = \text{argmin}_{\psi \in \{\psi_1, \dots, \psi_k\}} \text{size}(\psi)$ . Modify all of the proof steps that take  $\psi_1, \dots, \psi_k$  as premises to take  $\hat{\psi}$  as a premise instead. This transformation preserves the correctness of the proof because the validity of a proof step depends only on the conclusions of its immediate premises, not on any other structural features of its premises. Repeat this process until no redundant steps remain.

During the process of removing duplicates, it is possible for some steps to become disconnected from the main proof tree even if they are not duplicates. Consequently, once we have eliminated all redundant steps, we discard any remaining steps that are not in the tree of steps leading to the final conclusion. The end result is that we have a proof where every step has a distinct conclusion and no steps are unneeded. We do not formally guarantee that the reduced proof has the smallest possible number of steps needed to derive its conclusion, but in practice our post-processing makes proofs much smaller than they would be otherwise.

## 8 Evaluation

In our evaluation, we seek to answer four research questions. **(Q1)** How does PROOF scale relative to the size of the regular expressions it receives as input? **(Q2)** Can Crêpe validate equivalence proofs for complex regular expressions? **(Q3)** How does Crêpe scale relative to increases in proof size? **(Q4)** How does multiplexing affect the running time of Crêpe?

We report a positive answer for **Q2** and present the results for the other three questions in more detail later. For our evaluation, we used AWS instances of type r5b.4xlarge with 128 GB of memory and 16 vCPUs. For validation, we ran the prover and verifier on two separate instances with a 10 Gbps network connection between them.

### 8.1 Benchmarks

Our regular expression benchmarks come from the evaluation suite of FlashRegex, a tool for generating regular expressions that are immune to denial-of-service attacks [53]. Other existing regular expression benchmark suites, such as the suite used by ReDoSHunter [52], do not contain significant numbers of equivalent regular expression pairs. The FlashRegex suite does not group regular expressions into equivalent pairs on its own, so we follow a two-phase process to generate proofs for our evaluation. For the first phase, we use EQUIV to check whether two regular expressions are equivalent according to our formalization. We use 1,475 distinct regular expressions from FlashRegex for a total of  $\binom{1,475}{2} = 1,087,075$  pairs to check for equivalence. For the second phase, we run PROOF on the equivalent pairs that we find with EQUIV to generate our proofs.

Our time limit for EQUIV is ten minutes, and 21 pairs hit the time limit. For a small number of inputs, EQUIV takes a long time to terminate, with the slowest taking 594.62 seconds, but the average running time is only 1.12 seconds for confirmed equivalent pairs and 0.0025 seconds for confirmed inequivalent pairs. We plot the running times for EQUIV in Figure 1. We also impose a ten-minute time limit for PROOF. Of the 7,353 equivalent pairs that we find with EQUIV, 72 hit the time limit during proof generation. We exclude them from consideration for our evaluation.

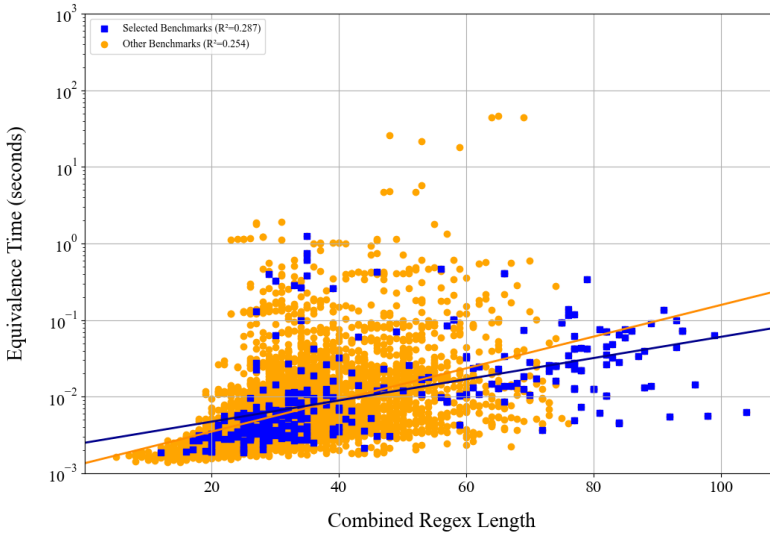


Fig. 1. Equivalence checking time relative to the combined size of the regular expressions being compared, plotted on a logarithmic scale.

Most of the regular expressions that we use for testing have two-character alphabets. Regular expressions with small alphabets are more likely to have semantic equivalents than regular expressions with large alphabets are. The benchmarks that hit the time limit for EQUIV and PROOF involve regular expressions with high numbers of unions and stars. All 21 timeouts for EQUIV and 13 of the 72 timeouts for PROOF involve this regular expression:

$$((aabb|abab|abba|bbaa|baba|baab)^*(aa|ab|ba|bb)^*)+((ab)?)$$

Moreover, all 15 of the non-timeout inputs that cause EQUIV to run for more than 61 seconds involve the same regular expression. Also, 33 of the PROOF timeouts involve this regular expression:

$$(bb|aa|((aabb|abab|abba|baba|bbaa|baab)^*)^*((aa)|(bb)|(ab)|(ba))^*$$

To keep our evaluation suite manageable, we select a sample of our proofs for validation instead of running Crêpe on all 7,281 of them. Our main test suite includes the 50 proofs whose starting regular expressions have the highest combined length, the 50 proofs with the highest value of  $\xi$  (string table length), the 50 proofs with the highest value of  $\nu$  (maximum Sync string length), the 50 proofs with the highest value of  $\pi$  (number of proof steps), and 200 randomly selected proofs. We have 301 main benchmarks in total rather than 400 because the categories overlap partially.

## 8.2 Proof Generation Time

To answer Q1, we time the proof generation process for all equivalent pairs except the ones that hit the time limit for EQUIV. We plot the running time of our proof generator against the combined size in characters of the two regular expressions being compared. The results appear in Figure 2. Our proof generation time includes the execution of PROOF and the post-processing for the removal of redundant steps that we describe in Section 7.2, but it does not include the execution of EQUIV.

Across all 7,281 equivalent pairs for which proof generation succeeds, the average running time for proof generation is 4.66 seconds, the median is 0.29 seconds, and the standard deviation is 26.04

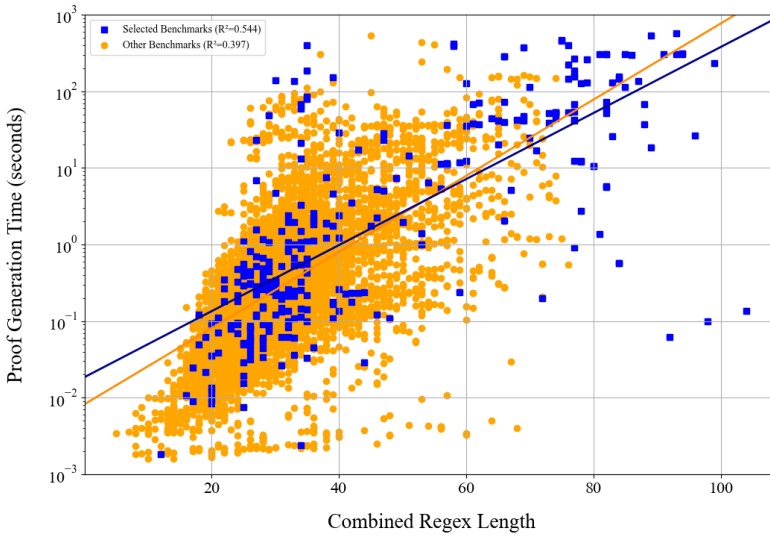


Fig. 2. Proof generation time relative to the combined size of the regular expressions being compared, plotted on a logarithmic scale.

seconds. There are a few very slow outliers, with the slowest taking 566.00 seconds, but proof generation is usually very quick. We do not consider the slow outliers a problem. Proof generation happens offline, before Crêpe runs. It involves no cryptographic operations, and the verifier does not observe it. Furthermore, because regular expression equivalence is PSPACE-complete, slow running times are inevitable for some large inputs.

The slowest non-timeout benchmark for PROOF has one of the highest combined regex lengths, but, in general, the correlation between combined regex length and proof generation time is not exact. Across all the benchmarks that do not hit the time limit, the  $R^2$  value for the correlation between the log of the running time of PROOF and the combined regex size of the input is approximately 0.63. The running time of PROOF grows at a roughly exponential rate relative to the combined size in characters of the regular expressions being compared, but factors other than regex length can affect the running time of PROOF significantly. If a regular expression from the starting pair has an AST that is far removed from normal form, PROOF will need to use a significant number of proof steps to normalize it. Also, a regular expression's length and the complexity of its AST do not correlate perfectly. For instance, parentheses do not appear in a regular expression's AST, but they still count toward its regex length.

We highlight the benchmarks that we select for validation in a different color in Figure 2. The proofs that we select for validation are generally larger and slower to generate than the rest of the proofs are. Among the 301 equivalent pairs that we use for validation, the average running time for proof generation is 43.89 seconds, the median is 0.98 seconds, and the standard deviation is 98.88 seconds. The maximum is 566.00 seconds, just as it is for the full suite. For proof step counts, the benchmarks that we selected for validation have an average of 1839.82, a median of 890, and a standard deviation of 1853.98. For comparison, the full collection of 7,281 equivalence proofs has an average proof step count of 807.79, a median of 652, and a standard deviation of 656.50.

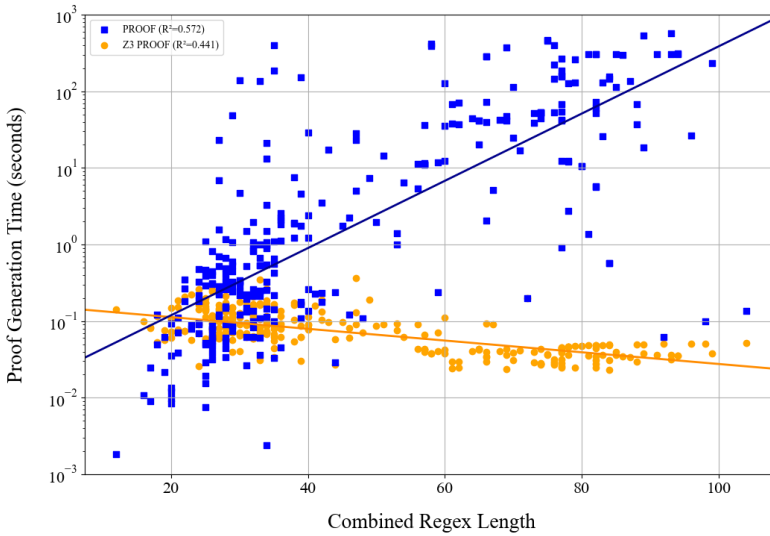


Fig. 3. Proof generation times for PROOF and Z3, plotted on a logarithmic scale.

### 8.3 Comparison to Z3

For a further assessment of **Q1**, we performed a baseline comparison of PROOF against Z3. As we mentioned in Section 3.2, Z3 can check equivalences between regular expressions and generate proof certificates for them. We ran Z3 on our 301 selected benchmarks with a time limit of ten minutes, both with and without proof generation enabled. We plot Z3’s running times with proof generation and without proof generation in Figures 3 and 4, respectively. With proof generation enabled, Z3 hits the time limit for 8 of the benchmarks. Among the 293 benchmarks for which Z3 does not hit the time limit, Z3 has an average running time of 0.088 seconds, a median of 0.070 seconds, and a standard deviation of 0.061 seconds. Its slowest non-timeout running time is 0.42 seconds. With proof generation disabled, Z3 is somewhat faster, but it hits the time limit on the same 8 benchmarks. Among the 293 successes, it has an average running time of 0.061 seconds, a median of 0.049 seconds, a standard deviation of 0.039 seconds, and a maximum of 0.26 seconds.

PROOF has no timeouts on the same 301 benchmarks, but overall it is slower than Z3 by a wide margin: we presented its statistics for our selected benchmarks in Section 8.2. On the other hand, EQUIV performs comparably to Z3 on our selected benchmarks: it has no timeouts, an average of 0.039 seconds, a median of 0.0074 seconds, a standard deviation of 0.11 seconds, and a maximum of 1.26 seconds. The fact that Z3 hits the time limit on 8 benchmarks for which EQUIV and PROOF do not hit the time limit does not necessarily imply that Z3 has worse coverage in general. We also ran Z3 on the 21 regular expression pairs for which EQUIV hits the time limit (Section 8.1). Z3 can prove all of them equivalent in less than two seconds each with proof generation enabled. With proof generation disabled, each one takes less than a second. Additionally, we ran Z3 on the 72 regular expression pairs for which PROOF hits the time limit but EQUIV does not. Z3 succeeds on each one in less than a second with proof generation enabled, and it succeeds on each one in less than half a second without proof generation enabled.

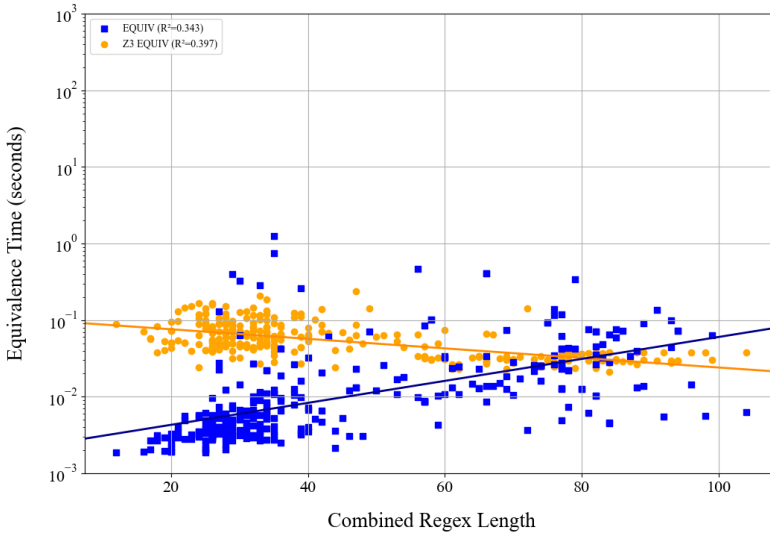


Fig. 4. Equivalence checking times for EQUIV and Z3, plotted on a logarithmic scale.

The fact that Z3 can generate proof certificates far more quickly than PROOF can does not make it a viable substitute for PROOF. As we explained in Section 3.2, the proof certificates that Z3 provides are not usable in ZK because they contain black-box term rewriting steps. If we could replace Z3’s black-box steps with explicit proofs, we could use Z3’s certificates in a ZK protocol. However, replacing those steps would require the creation of a new calculus of proof rules and a new algorithm for generating proofs in the calculus. This would be analogous to the rules and algorithm that we created for Crêpe.

#### 8.4 Proof Validation Time

To answer Q2, Q3, and Q4, we use Crêpe to validate the proofs that we generate for our chosen benchmarks. We run Crêpe with three different configurations for multiplexing:

- (1) **Default Settings.** By default, Crêpe uses the multiplexing categories that we define in Section 6.
- (2) **No Multiplexing.** The only checking instruction executed on a proof step is the one for that step’s real underlying proof rule. This causes Crêpe to leak information about the frequencies of different rules in the same way that ZKSMT does [54].
- (3) **Full Multiplexing.** All rules except Assume and Contra (Appendix A, Table 10) are grouped in a single category for multiplexing. With full multiplexing, Crêpe leaks less information than it does under the default settings.

We tried other multiplexing configurations, such as using two multiplexing categories where one category contains all constant-time rules and the other contains all linear-time rules. However, our preliminary results for them were inferior to the default settings.

*Results.* We ran each of the three versions of Crêpe on all 301 of our selected benchmarks. A scatter plot with our results for proof validation appears in Figure 5. We imposed a ten-minute timeout again for validation, but no benchmarks hit the time limit. In general, the running time of the default version of Crêpe increases at a roughly linear rate as the number of steps in the proof

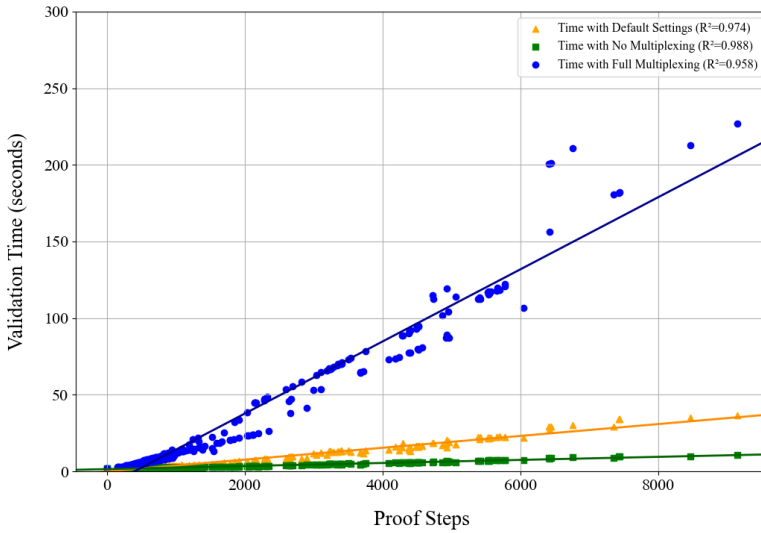


Fig. 5. Proof validation times for all configurations of Crêpe relative to proof step count.

increases. The version with no multiplexing follows a similar pattern but is faster in most cases. Full multiplexing is significantly slower than both of the other options. The difference between it and the other two options widens as the number of proof steps increases. The default version of Crêpe validates 84.39 percent of the benchmarks (254 out of 301) in 15 seconds or less.

With the default settings, the average validation time is 7.03 seconds, the median is 3.10 seconds, and the standard deviation is 7.27 seconds. With no multiplexing, the average is 3.30 seconds, the median is 2.27 seconds, and the standard deviation is 1.87 seconds. With full multiplexing, the average is 33.93 seconds, the median is 10.71 seconds, and the standard deviation is 44.58 seconds. The slowest individual validation takes 36.64 seconds with the default settings, 10.75 seconds with no multiplexing, and 226.85 seconds with full multiplexing. In the timing results for full multiplexing, there are two visibly distinct linear regressions. Having a higher value of  $\nu$  causes some benchmarks to be slower relative to their step count.

The median ratio of Crêpe’s running time with no multiplexing to its running time with the default settings is 0.74, so the added security that comes from multiplexing does not require a prohibitively large increase in running time. The median ratio of Crêpe’s running time with full multiplexing to its running time with the default settings is 3.42.

*Alphabet Sizes.* Most of the regular expressions that we use in our evaluation have two-character alphabets, but regular expressions used in real-world software usually have much larger alphabets. The number of premises taken by Coinduction and Match scales linearly relative to the alphabet size of the starting regular expressions, so large alphabets would cause Crêpe to run slowly. However, it is possible to refactor any regular expression to have a two-character alphabet: one can simply assign a binary representation to each distinct character in the original regular expression. This replacement increases the number of AST nodes in a regular expression by a factor of only  $O(\log n)$ , where  $n$  is the number of distinct characters in the original regular expression.

## 9 Related Work

*ZK Regular Expression Protocols.* Cryptographic protocols have been developed in the past for reasoning about regular expressions, but no existing protocol targets the problem that Crêpe does. Instead of checking whether two regular expressions are equivalent, they check whether an individual string appears in the language of a regular expression or whether the string contains a substring that matches the regular expression.

Zombie [74] can prove in zero knowledge that a packet satisfies network middlebox constraints involving Boolean combinations of regular expressions. The text of the packet is private, and the constraints are public. ZK-regex [56] also provides ZK proofs that a private string matches a public regular expression. Unlike Zombie, ZK-regex supports a two-party option for string matching where the string and regular expression are both private and belong to different parties. The protocol zkreg [61] provides succinct ZK proofs that a hidden string belongs to a public regular language. Reef [6] is another ZK protocol for matching private strings against public regular expressions that generates succinct proofs. Unlike Zombie, ZK-regex, and zkreg, Reef supports features that are not allowed in pure regular expressions, such as lookahead assertions.

Crêpe doubles as a protocol for encoding string matching proofs: to demonstrate that a private string  $s$  matches a public regular expression  $r$ , the prover can treat  $s$  as a regular expression, leak  $r$  to the verifier, and encode a proof that  $s|r = r$ . However, Crêpe would not be as efficient as Zombie, ZK-regex, zkreg, and Reef are because we designed it to handle a more general problem.

*ZK Proof Validation.* ZKSMT [54] is a ZK protocol for encoding SMT proofs. It could be extended to encode regular expression equivalence proofs, but it is not as well-suited for the task as Crêpe is. ZKSMT aims to provide a generalizable framework for proof validation in ZK. To support arbitrary first-order theories, ZKSMT relies on list-based operations and type checking, which are costly operations for the protocol. Neither one is necessary when we restrict ourselves to reasoning about only regular expressions. Furthermore, ZKSMT provides weaker security guarantees than Crêpe does. ZKSMT does not multiplex over rules: it leaks the frequencies of different proof rules but not their order. To our knowledge, there has been no prior work on reconstructing hidden proofs from the frequency distributions of their rules, but ZKSMT does not provide a formal guarantee of its own security in that regard.

The protocol zkPi encodes proofs written in Lean, an interactive theorem prover [50]. In principle, zkPi could encode proofs like the ones that PROOF creates for Crêpe. Lean allows users to create custom proof rules, and formalizations of regular expressions in Lean exist [11, 59], so we could have implemented a ZK regular expression equivalence protocol on top of Lean's regular expressions. However, Lean does not generate proofs on its own: for a new formalism, a user needs to create an algorithm for generating proofs or to write the proofs manually, and most programmers do not have the expertise necessary to do either. We would still need a custom proof generator, so the task of incorporating regular expression equivalence proofs into zkPi would require just as much effort as creating a new ZK protocol. Also, zkPi supports a full calculus of dependent types that is unnecessary for regular expression equivalence proofs. A specialized ZK protocol for validating only regular expression equivalence proofs is more efficient than zkPi would be.

ZKUNSAT [55] encodes proofs that pure Boolean formulas in conjunctive normal form are unsatisfiable. It is inadequate for regular expression equivalence proofs as well. Boolean satisfiability is in a lower complexity class than regular expression equivalence is, so the conversion from regular expressions to Boolean formulas would require a substantial increase in the input size.

*General-Purpose ZK Protocols.* Protocols such as Cheesecloth [29], TinyRAM [10], Pantry [20], and Buffet [69] can model the execution of arbitrary programs in zero knowledge. However, general-purpose ZK protocols are highly inefficient in practice: to validate a simple SMT proof, Cheesecloth needs to run for almost two hours [54]. Other general-purpose ZK protocols suffer from the same performance issues that Cheesecloth does, so they would be impractical for validation of regular expression equivalence proofs.

## 10 Conclusion

We have introduced Crêpe, the first ZK protocol designed to support regular expression equivalence proofs and the first ZK protocol to target a PSPACE-complete problem. Multiple potential directions exist for future work based on Crêpe and its custom calculus of proof rules. The integration of PROOF or a similar algorithm into mainstream SMT solvers would facilitate the use of regular expression equivalence proofs in practice.

Crêpe can be modified to support proofs about Kleene Algebra with Tests (KAT). KAT extends the structure of regular expressions with variables and logical connectives [46]. KAT itself supports further extensions with theories such as bit vectors [42], the behavior of packets in a network [5], and Hoare logic [8, 47]. For this paper, we choose to focus on ordinary regular expressions because they are the most common application of Kleene algebra in practice.

In Crêpe’s model, both private regular expressions belong to the same party. An additional direction for future work would be the development of a privacy-preserving regular expression equivalence protocol for the scenario where the regular expressions being compared belong to different parties. Unlike Crêpe, the multi-party protocol would need to prove the regular expressions’ equivalence in a cryptographic setting rather than relying on a proof constructed offline.

*Limitations.* We do not consider intersections, complements, backreferences, or lookahead assertions for Crêpe since they are not allowed in pure regular expressions. It is possible to extend Crêpe’s calculus of rules to support them, but we would lose decidability for equivalence if we incorporated backreferences or lookahead assertions [38].

*Acknowledgements.* We thank the anonymous reviewers of both the paper and the artifact. We thank Nikolaj Bjørner for answering our questions about Z3, and we thank Andrew Reynolds for answering our questions about CVC5. This work was supported by the National Science Foundation under Grant Numbers CCF-2131476, CCF-2318974, CCF-2219995, and CNS-2245344.

## 11 Data-Availability Statement

The artifact for Crêpe is available at <https://zenodo.org/records/16916368> [45]. The artifact includes our implementation of Crêpe, the benchmarks that we used for our evaluation, and scripts for running the evaluation.

## References

- [1] Omar Abahussain and Yousef Harrath. 2019. Detection of malicious emails through regular expressions and databases. In *2019 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT)*. IEEE, 1–5.
- [2] Alfred V Aho. 2014. Algorithms for finding patterns in strings. *Algorithms and Complexity* 1 (2014), 255.
- [3] Marco Almeida, Nelma Moreira, and Rogério Reis. 2008. Antimirov and Mosses’s rewrite system revisited. In *International Conference on Implementation and Application of Automata*. Springer, 46–56.
- [4] Marco Almeida, Nelma Moreira, and Rogério Reis. 2009. Testing the equivalence of regular languages. *arXiv preprint arXiv:0907.5058* (2009).
- [5] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. *Acm sigplan notices* 49, 1 (2014), 113–126.

- [6] Sebastian Angel, Eleftherios Ioannidis, Elizabeth Margolin, Srinath Setty, and Jess Woods. 2023. Reef: Fast Succinct Non-Interactive Zero-Knowledge Regex Proofs. Cryptology ePrint Archive (2023).
- [7] Valentin M Antimirov and Peter D Mosses. 1995. Rewriting extended regular expressions. Theoretical Computer Science 143, 1 (1995), 51–72.
- [8] Timos Antonopoulos, Eric Koskinen, Ton Chanh Le, Ramana Nagasamudram, David A Naumann, and Minh Ngo. 2023. An algebra of alignment for relational verification. Proceedings of the ACM on Programming Languages 7, POPL (2023), 573–603.
- [9] Carsten Baum, Alex J Malozemoff, Marc B Rosen, and Peter Scholl. 2021. Mac’ n’ Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions. In Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part IV 41. Springer, 92–122.
- [10] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In CRYPTO 2013, Part II (LNCS, Vol. 8043), Ran Canetti and Juan A. Garay (Eds.). Springer, 90–108.
- [11] @bergmannjg. [n. d.]. Regex | Reservoir. <https://reservoir.lean-lang.org/@bergmannjg/Regex>. Accessed: 2025-07-08.
- [12] Murphy Berzish, Mitja Kulczynski, Federico Mora, Florin Manea, Joel D Day, Dirk Nowotka, and Vijay Ganesh. 2021. An SMT solver for regular expressions and linear arithmetic over string length. In International Conference on Computer Aided Verification. Springer, 289–312.
- [13] Olaf Beyersdorff, Mikołaj Janota, Florian Lonsing, and Martina Seidl. 2021. Quantified boolean formulas. In Handbook of Satisfiability. IOS Press, 1177–1221.
- [14] Nikolaj Bjørner. 2024. Private communication.
- [15] Nikolaj Bjørner. 2025. Private communication.
- [16] BlueFalconHD. [n. d.]. `apple_generative_model_safety_decrypted/decrypted_overrides/com.apple.gm.safety_deny.output.mail_reply.long_form_rewrite.generic/AssetData/locale/es_US/metadata.json` at main · BlueFalconHD/apple\_generative\_model\_safety\_decrypted · GitHub. [https://github.com/BlueFalconHD/apple\\_generative\\_model\\_safety\\_decrypted/blob/main/decrypted\\_overrides/com.apple.gm.safety\\_deny.output.mail\\_reply.long\\_form\\_rewrite.generic/AssetData/locale/es\\_US/metadata.json](https://github.com/BlueFalconHD/apple_generative_model_safety_decrypted/blob/main/decrypted_overrides/com.apple.gm.safety_deny.output.mail_reply.long_form_rewrite.generic/AssetData/locale/es_US/metadata.json). Accessed: 2025-07-06.
- [17] BlueFalconHD. [n. d.]. GitHub - BlueFalconHD/apple\_generative\_model\_safety\_decrypted: Decrypted Generative Model safety files for Apple Intelligence containing filters. [https://github.com/BlueFalconHD/apple\\_generative\\_model\\_safety\\_decrypted/tree/main](https://github.com/BlueFalconHD/apple_generative_model_safety_decrypted/tree/main). Accessed: 2025-07-06.
- [18] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. 1991. Noninteractive zero-knowledge. SIAM J. Comput. 20, 6 (1991), 1084–1118.
- [19] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. 1991. Checking the correctness of memories. In 32nd FOCS. IEEE Computer Society Press, 90–99.
- [20] Benjamin Braun, Ariel J Feldman, Zuo Cheng Ren, Srinath Setty, Andrew J Blumberg, and Michael Walfish. 2013. Verifying computations with state. In ACM SOS.
- [21] Janusz A Brzozowski. 1962. A Survey of Regular Expressions and Their Applications. IRE Transactions on Electronic Computers 3, EC-11 (1962), 324–335.
- [22] Janusz A Brzozowski. 1964. Derivatives of regular expressions. Journal of the ACM (JACM) 11, 4 (1964), 481–494.
- [23] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. 1999. Rewriting of regular expressions and regular path queries. In Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. 194–204.
- [24] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. 2002. Universally composable two-party and multi-party secure computation. In Proceedings of the thirty-fourth annual ACM symposium on Theory of computing. ACM Press, 494–503.
- [25] Sarah Cen and Devavrat Shah. 2021. Regulating algorithmic filtering on social media. Advances in Neural Information Processing Systems 34 (2021), 6997–7011.
- [26] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. 2012. SMTInterpol: An interpolating SMT solver. In International SPIN Workshop on Model Checking of Software. Springer, 248–254.
- [27] Microsoft Corporation. [n. d.]. Regular Expressions | Online Z3 Guide. <https://microsoft.github.io/z3guide/docs/theories/Regular%20Expressions/>. Accessed: 2024-09-02.
- [28] Flavio Corradini, Rocco De Nicola, and Anna Labella. 2002. An equational axiomatization of bisimulation over regular expressions. Journal of Logic and Computation 12, 2 (2002), 301–320.
- [29] Santiago Cuéllar, Bill Harris, James Parker, Stuart Pernsteiner, and Eran Tromer. 2023. Cheesecloth: Zero-Knowledge Proofs of Real World Vulnerabilities. In 32nd USENIX Security Symposium (USENIX Security 23). USENIX Association, Anaheim, CA, 6525–6540. <https://www.usenix.org/conference/usenixsecurity23/presentation/cuellar>

- [30] Ivan Damgård. 1998. Commitment schemes and zero-knowledge protocols. In School organized by the European Educational Forum. Springer, 63–86.
- [31] Leonardo Mendonça de Moura and Nikolaj S Bjørner. 2008. Proofs and Refutations, and Z3. In LPAR Workshops, Vol. 418. Doha, Qatar, 123–132.
- [32] Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. 1988. Non-interactive zero-knowledge proof systems. In Advances in Cryptology—CRYPTO’87: Proceedings 7. Springer, 52–72.
- [33] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. 2022. Efficient Proof of RAM Programs from Any Public-Coin Zero-Knowledge System. In International Conference on Security and Cryptography for Networks. Springer, 615–638.
- [34] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. 2021. Line-Point Zero Knowledge and Its Applications. In 2nd Conference on Information-Theoretic Cryptography (ITC 2021) (Leibniz International Proceedings in Informatics (LIPIcs)). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany.
- [35] Chiara Patricia Drolsbach and Nicolas Pröllochs. 2024. Content moderation on social media in the EU: insights from the DSA transparency database. In Companion Proceedings of the ACM Web Conference 2024. 939–942.
- [36] Nate Foster, Dexter Kozen, Mae Milano, Alexandra Silva, and Laure Thompson. 2015. A coalgebraic decision procedure for NetKAT. In Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 343–355.
- [37] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. 2021. Constant-overhead zero-knowledge for RAM programs. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 178–191.
- [38] Dominik D Freydenberger. 2013. Extended regular expressions: Succinctness and decidability. Theory of Computing Systems 53 (2013), 159–193.
- [39] Enrico Giunchiglia, Paolo Marin, and Massimo Narizzano. 2009. Reasoning with quantified boolean formulas. In Handbook of satisfiability. IOS Press, 761–780.
- [40] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1991. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. Journal of the ACM (JACM) 38, 3 (1991), 690–728.
- [41] Shafi Goldwasser, Silvio Micali, and Chales Rackoff. 1985. The knowledge complexity of interactive proof-systems (extended abstract). In 17th ACM STOC. ACM Press, 291–304.
- [42] Michael Greenberg, Ryan Beckett, and Eric Campbell. 2022. Kleene algebra modulo theories: a framework for concrete KATs. In Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 594–608.
- [43] David Heath and Vladimir Kolesnikov. 2020. A 2.1 KHz zero-knowledge processor with BubbleRAM. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, 2055–2074.
- [44] Donald M Kaplan. 1969. Regular expressions and the equivalence of programs. J. Comput. System Sci. 3, 4 (1969), 361–386.
- [45] John C. Kolesar, Shan Ali, Timos Antonopoulos, and Ruzica Piskac. 2025. Coinductive Proofs of Regular Expression Equivalence in Zero Knowledge: Artifact. <https://doi.org/10.5281/zenodo.16916368>
- [46] Dexter Kozen. 1997. Kleene algebra with tests. ACM Transactions on Programming Languages and Systems (TOPLAS) 19, 3 (1997), 427–443.
- [47] Dexter Kozen. 2002. On Hoare logic, Kleene algebra, and types. SYNTHESE LIBRARY (2002), 119–136.
- [48] Dexter Kozen and Maria-Christina Patron. 2000. Certification of Compiler Optimizations Using Kleene Algebra with Tests. In Computational Logic. <https://api.semanticscholar.org/CorpusID:17109057>
- [49] Sebastian Kühne. 2021. Automatically Testing Solvers for String and Regular Expressions Constraints. Bachelor’s Thesis. Swiss Federal Institute of Technology Zurich.
- [50] Evan Laufer, Alex Ozdemir, and Dan Boneh. 2024. zkpi: Proving lean theorems in zero-knowledge. In Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. 4301–4315.
- [51] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert—a formally verified optimizing compiler. In ERTS.
- [52] Yeting Li, Zixuan Chen, Jialun Cao, Zhiwu Xu, Qiancheng Peng, Haiming Chen, Liyuan Chen, and Shing-Chi Cheung. 2021. {ReDoSHunter}: A combined static and dynamic approach for regular expression {DoS} detection. In 30th USENIX Security Symposium (USENIX Security 21). 3847–3864.
- [53] Yeting Li, Zhiwu Xu, Jialun Cao, Haiming Chen, Tingjian Ge, Shing-Chi Cheung, and Haoren Zhao. 2020. FlashRegex: deducing anti-ReDoS regexes from examples. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. 659–671.

- [54] Daniel Luick, John C Kolesar, Timos Antonopoulos, William R Harris, James Parker, Ruzica Piskac, Eran Tromer, Xiao Wang, and Ning Luo. 2024. ZKSMT: A VM for Proving SMT Theorems in Zero Knowledge. In 33rd USENIX Security Symposium (USENIX Security 24). 3837–3845.
- [55] Ning Luo, Timos Antonopoulos, William R Harris, Ruzica Piskac, Eran Tromer, and Xiao Wang. 2022. Proving UNSAT in zero knowledge. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM Press, 2203–2217.
- [56] Ning Luo, Chenkai Weng, Jaspal Singh, Gefei Tan, Ruzica Piskac, and Mariana Raykova. 2023. Privacy-Preserving Regular Expression Matching using Nondeterministic Finite Automata. Cryptology ePrint Archive (2023).
- [57] Christian Matt and Ueli Maurer. 2013. The one-time pad revisited. In 2013 IEEE international Symposium on information Theory. IEEE, 2706–2710.
- [58] George C Necula. 2000. Translation validation for an optimizing compiler. In ACM SIGPLAN.
- [59] @pandaman64. [n. d.]. lean-regex | Reservoir. <https://reservoir.lean-lang.org/@pandaman64/lean-regex>. Accessed: 2025-07-08.
- [60] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In TACAS.
- [61] Michael Raymond, Gillian Evers, Jan Ponti, Diya Krishnan, and Xiang Fu. 2023. Efficient Zero Knowledge for Regular Language. Cryptology ePrint Archive (2023).
- [62] Dominika Regéciová and Dušan Kolář. 2023. GenRex: Leveraging Regular Expressions for Dynamic Malware Detection. In 2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). IEEE, 857–864.
- [63] Davide Sangiorgi. 2009. On the origins of bisimulation and coinduction. ACM Trans. Program. Lang. Syst. 31, 4, Article 15 (may 2009), 41 pages. <https://doi.org/10.1145/1516507.1516510>
- [64] Adi Shamir. 1992. Ip= pspace. Journal of the ACM (JACM) 39, 4 (1992), 869–877.
- [65] Robin Sommer and Vern Paxson. 2003. Enhancing byte-level network intrusion detection signatures with context. In Proceedings of the 10th ACM conference on Computer and communications security. 262–271.
- [66] Richard Edwin Stearns and Harry B Hunt III. 1985. On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. SIAM J. Comput. 14, 3 (1985), 598–611.
- [67] Larry J Stockmeyer and Albert R Meyer. 1973. Word problems requiring exponential time (preliminary report). In Proceedings of the fifth annual ACM symposium on Theory of computing. 1–9.
- [68] Jan Van Lunteren. 2006. High-performance pattern-matching for intrusion detection. In Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications. Citeseer, 1–13.
- [69] Riad S. Wahby, Srinath T. V. Setty, Zuo Cheng Ren, Andrew J Blumberg, and Michael Walfish. 2015. Efficient RAM and control flow in verifiable outsourced computation. In NDSS 2015. The Internet Society.
- [70] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. 2021. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In 2021 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society Press, 1074–1091.
- [71] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. 2021. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 2986–3001.
- [72] Yibin Yang, David Heath, Carmit Hazay, Vladimir Kolesnikov, and Muthuramakrishnan Venkitasubramaniam. 2023. Batchman and robin: Batched and non-batched branching for interactive zk. In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. 1452–1466.
- [73] Yibin Yang, David Heath, Carmit Hazay, Vladimir Kolesnikov, and Muthuramakrishnan Venkitasubramaniam. 2024. Tight ZK CPU: Batched ZK Branching with Cost Proportional to Evaluated Instruction. Cryptology ePrint Archive (2024).
- [74] Collin Zhang, Zachary DeStefano, Arasu Arun, Joseph Bonneau, Paul Grubbs, and Michael Walfish. 2024. Zombie: Middleboxes that {Don't} Snoop. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). 1917–1936.
- [75] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100gbps intrusion prevention on a single server. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). 1083–1100.
- [76] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. 2017. Z3str2: an efficient solver for strings, regular expressions, and length constraints. Formal Methods in System Design 50 (2017), 249–288.

RuleID	Conclusion
UnionAssoc	$p (q r) = (p q) r$
UnionComm	$p q = q p$
UnionEmpty	$p \emptyset = p$
UnionSelf	$p p = p$
ConcatAssoc	$p(qr) = (pq)r$
ConcatBlankL	$\epsilon p = p$
ConcatBlankR	$p\epsilon = p$
ConcatEmptyL	$\emptyset p = \emptyset$
ConcatEmptyR	$p\emptyset = \emptyset$

Table 6. Normalization Rules

RuleID	Premises	Conclusion
Refl		$x = x$
Symm	$x = y$	$y = x$
Trans	$x = y, y = z$	$x = z$
PredCongL	$P(x_1, y), x_1 = x_2$	$P(x_2, y)$
PredCongR	$P(x, y_1), y_1 = y_2$	$P(x, y_2)$
FunCong1	$x_1 = x_2$	$f(x_1) = f(x_2)$
FunCong2	$x_1 = x_2, y_1 = y_2$	$f(x_1, y_1) = f(x_2, y_2)$

Table 7. Equality Rules

## A Simple Proof Rules

*Normalization Rules.* The normalization rules appear in Table 6. None of the normalization rules have premises. Note that we do not include all of the standard axioms of Kleene algebra as proof rules [46]. There are no rules for manipulating stars or for distributing concatenations over unions. We can still uphold completeness with the axioms that we have, so we omit the others to avoid unnecessary complexity.

In the ZK proofs that PROOF produces as output, we need to confirm that our transformations for normalization are valid. However, we never need to prove that the end result of normalization is a regular expression in normal form. Whether a regular expression is in normal form does not have any bearing on which proof rules can be applied to it: its only importance is that it forces PROOF to terminate.

*Equality Rules.* The equality rules appear in Table 7. All of our equality rules are standard axioms of first-order logic. The rules Refl, Symm, and Trans simply allow us to use equality as an equivalence relation. Our congruence rules only need to support binary predicates because all predicates in our formalism take two terms as arguments except  $\perp$ , which takes none. Sync also takes a string as an argument, but we never need to perform substitutions for strings.

*Epsilon Rules.* Our epsilon rules appear in Table 8. Instead of having a single rule for the epsilon function, we have a separate rule for each regular expression node. The definition for each case is standard [22].

*Derivative Rules.* The derivative rules appear in Table 9. They follow the standard definition of the derivative function [22]. DeriveCharDifferent is the only rule in our calculus with a *side*

RuleID	Premises	Conclusion
EpsilonEmpty		$E(\emptyset) = \emptyset$
EpsilonBlank		$E(\epsilon) = \epsilon$
EpsilonChar		$E(c) = \emptyset$
EpsilonUnionPos1	$E(p) = \epsilon$	$E(p q) = \epsilon$
EpsilonUnionPos2	$E(q) = \epsilon$	$E(p q) = \epsilon$
EpsilonUnionNeg	$E(p) = \emptyset, E(q) = \emptyset$	$E(p q) = \emptyset$
EpsilonConcatPos	$E(p) = \epsilon, E(q) = \epsilon$	$E(pq) = \epsilon$
EpsilonConcatNeg1	$E(p) = \emptyset$	$E(pq) = \emptyset$
EpsilonConcatNeg2	$E(q) = \emptyset$	$E(pq) = \emptyset$
EpsilonStar		$E(p^*) = \epsilon$

Table 8. Epsilon Rules

RuleID	Conclusion
DeriveEmpty	$\delta(c, \emptyset) = \emptyset$
DeriveBlank	$\delta(c, \epsilon) = \emptyset$
DeriveCharSame	$\delta(c, c) = \epsilon$
DeriveCharDifferent	$\delta(c_1, c_2) = \emptyset$ if $c_1$ is not $c_2$
DeriveUnion	$\delta(c, p q) = \delta(c, p)   \delta(c, q)$
DeriveConcat	$\delta(c, pq) = \delta(c, p)q   E(p)\delta(c, q)$
DeriveStar	$\delta(c, p^*) = \delta(c, p)p^*$

Table 9. Derivative Rules

RuleID	Premises	Conclusion
Assume		$p \neq q$
Contra	$p \neq q, p = q$	$\perp$

Table 10. Proof Completion Rules

*condition.* The side condition is not a premise, nor is it part of the conclusion. It does not appear in any table at all. The checking instruction for DeriveCharDifferent merely checks at runtime that the AST nodes for  $c_1$  and  $c_2$  have different characters. The conclusion of DeriveCharDifferent that appears in  $M_f$  is simply  $\delta(c_1, c_2) = \emptyset$ .

*Proof Completion Rules.* We need two additional rules to bookend our proofs: Assume and Contra. They appear in Table 10. A proof contains one use of Assume as its starting point. Assume can introduce only formulas of the form  $p \neq q$ , so it cannot trivialize a proof. Contra concludes a proof: it derives  $\perp$  from  $p \neq q$  and  $p = q$ . Assume is the only rule that can introduce inequality formulas, so the first premise of Contra must come from Assume.

For multiplexing, Assume and Contra each have a category to themselves. Proofs always contain exactly one occurrence of each, so this is not a source of information leakage.

## B Rule Redundancy

Match is redundant with Coinduction: any equivalence that can be proven with Match can also be proven less directly with Coinduction and our auxiliary coinduction rules (Table 5). We choose to

include Match in our calculus of rules anyway because applications of Match are more efficient to validate in ZK than applications of Coinduction are. When we split Coinduction into  $n$  steps, some of the steps take linear time in  $\nu$  to check. When we split Match into  $n$  steps, all of the steps take constant time to check.

## C Completeness

Let  $p$  and  $q$  be two equivalent regular expressions. We want to show that there exists a proof that  $p = q$  in our calculus of rules. We will give an algorithm for finding the equivalence proof. Before we begin the completeness proof, we will state a number of preliminary findings whose proofs we omit.

Our preliminary findings depend on the definition of similarity from Section 7: two regular expressions are similar if one can be converted into the other using only our normalization rules and equality rules. We write  $p \equiv q$  to denote that  $p$  and  $q$  are similar. We also rely on the functions `reduce` and `normalize` from the same section.

**THEOREM 1 (TERM CONVERSION).** *Any term  $r$  can be converted into an equivalent regular expression using our normalization, equality, epsilon, and derivative rules. In particular, if  $r$  is  $E(r')$ , then  $r$  can be converted into either  $\emptyset$  or  $\epsilon$ , but not both.*

*Reduction Function.* Building on Theorem 1, let `reduce` be a function that takes a term  $r$  as input and produces an equivalent regular expression by unfolding any  $E$  and  $\delta$  nodes. If  $r$  is already a regular expression, `reduce( $r$ )` returns  $r$  itself.

*Normal Form.* Let  $R$  be an arbitrary total order over terms. We say that a term is in *normal form* if it satisfies the following conditions according to its `NodeID`:

- (1)  $\emptyset$ ,  $\epsilon$ , and individual characters are all in normal form.
- (2)  $p|q$  is in normal form if it satisfies one of two combinations of conditions:
  - (a)  $q$  is a union  $q_1|q_2$ ,  $p$  and  $q$  are in normal form,  $p \neq q_1$ ,  $p$  is not a union, neither  $p$  nor  $q_1$  is  $\emptyset$ , and  $p < q_1$  according to  $R$ .
  - (b)  $q$  is not a union,  $p$  and  $q$  are in normal form,  $p \neq q$ ,  $p$  is not a union, neither  $p$  nor  $q$  is  $\emptyset$ , and  $p < q$  according to  $R$ .
- (3)  $pq$  is in normal form if  $p$  and  $q$  are in normal form,  $p$  is not a concatenation, and neither  $p$  nor  $q$  is  $\emptyset$  or  $\epsilon$ .
- (4)  $p^*$  is in normal form if  $p$  is.

Note that terms in normal form must be regular expressions: our requirements forbid derivative and epsilon nodes from appearing.

**THEOREM 2 (NORMAL FORM CONVERSION).** *There exists a function `normalize` such that, for any regular expression  $r$ , the output of `normalize( $r$ )` is a regular expression in normal form that is similar to  $r$ .*

**THEOREM 3 (UNIQUENESS OF NORMAL FORM).** *A regular expression in normal form is not similar to any normal-form regular expression other than itself.*

### C.1 Main Completeness Proof

Let  $p$  and  $q$  be two equivalent regular expressions. We want to construct a proof of  $p = q$  using our calculus of rules.

To start, let  $P$  be the set of all regular expressions of the form `normalize(reduce( $\delta(s, p)$ ))` for strings  $s$ . Since  $s$  can be the empty string,  $P$  contains the normalized version of  $p$  itself. Following the same pattern, let  $Q$  be the set of all regular expressions of the form `normalize(reduce( $\delta(s, q)$ ))`.

We know from [22] that the set of all unfolded derivatives for  $p$  and  $q$  must fall into a finite number of equivalence classes for similarity. Because the regular expressions in  $P$  and  $Q$  are all normalized, combining that result with Theorem 3 gives us that  $P$  and  $Q$  must be finite sets.

Let  $k = |P| \cdot |Q|$ , and consider the set  $K$  of all strings in  $\Sigma$  of length  $k$ . A string  $s \in K$  has  $k + 1$  distinct prefixes, including both the empty string and  $s$  itself. If we view similarity as a relation over pairs of regular expressions, the set  $P \times Q$  has at most  $k$  distinct similarity classes, so there must be at least two prefixes of  $s$  whose derivatives for  $p$  and  $q$  are similar. Let  $\text{head}(s)$  and  $\text{foot}(s)$  be the shortest two distinct prefixes of  $s$  such that  $\delta(\text{head}(s), p) \equiv \delta(\text{foot}(s), p)$  and  $\delta(\text{head}(s), q) \equiv \delta(\text{foot}(s), q)$ , where  $\text{head}(s)$  is shorter than  $\text{foot}(s)$ . Also, let  $\text{body}(s)$  be the non-empty string such that  $\text{foot}(s) = \text{head}(s)\text{body}(s)$ . We can use SyncCycle to derive  $\text{Sync}(\text{body}(s), \delta(\text{head}(s), p), \delta(\text{head}(s), q))$ . Next, we can apply SyncFold repeatedly to convert that formula into  $\text{Sync}(\text{foot}(s), p, q)$ .

For the next step, let  $F_0$  be the set of all formulas of the form  $\text{Sync}(\text{foot}(s), p, q)$  that can be constructed from strings in  $K$  using this procedure. Note that two different strings in  $K$  may map to the same element of  $F_0$ .  $F_0$  is the initial *frontier* for our proof generation algorithm. A frontier  $F$  must uphold a number of invariants:

- (1)  $F$  is a finite set.
- (2) Every formula in  $F$  is of the form  $\text{Sync}(u, p, q)$  for some string  $u$ .
- (3) If  $s_1$  and  $s_2$  are strings such that the formulas  $\text{Sync}(s_1, p, q)$  and  $\text{Sync}(s_2, p, q)$  are both in  $F$ , then  $s_1$  is not a strict prefix of  $s_2$ .
- (4) For every string  $s$  of length  $k$ ,  $F$  contains  $\text{Sync}(t, p, q)$  for some string  $t$  that is a non-strict prefix of  $s$ .
- (5) Every formula in  $F$  has a proof in our calculus of rules.

All of these hold trivially for our initial frontier except invariant 3. We can prove invariant 3 for  $F_0$  by contradiction. Assume that we have both  $\text{Sync}(\text{foot}(s_1), p, q)$  and  $\text{Sync}(\text{foot}(s_2), p, q)$  in  $F_0$ , and, without loss of generality, assume that  $\text{foot}(s_1)$  is a strict prefix of  $\text{foot}(s_2)$ . According to our definition from before,  $\text{foot}(s_1)$  is the shortest prefix of  $s_1$  that hits a cycle for similarity. If  $\text{foot}(s_1)$  is a prefix of  $\text{foot}(s_2)$ , then  $\text{foot}(s_1)$  is a prefix of  $s_2$  as well. This would make  $\text{foot}(s_1)$  a prefix of  $s_2$  that hits a cycle for similarity.  $\text{foot}(s_1)$  is strictly shorter than  $\text{foot}(s_2)$ , so this contradicts the definition of  $\text{foot}(s_2)$  as the shortest prefix of  $s_2$  that hits a cycle for similarity. Therefore, this situation is impossible, and  $F_0$  must uphold invariant 3.

Now we need to define an algorithm for constructing a proof. We will define a function  $\text{update}(F)$  that takes a frontier  $F$  and produces a new frontier. To start, let  $m$  be the maximum depth of any entry of  $F$ . We define the depth of a formula  $\text{Sync}(u, p, q)$  as the length of the string  $u$  that it contains. If  $m$  is 0, return  $F$ . If  $m$  is positive, select an element  $\text{Sync}(w, p, q)$  from  $F$  such that the string  $w$  is of length  $m$ . We know that  $w$  is non-empty, so let  $c$  be the final character of  $w$ , and let  $w'$  be all of the characters before it. Let  $B$  be the set of all formulas of the form  $\text{Sync}(wb, p, q)$  for characters  $b \in \Sigma$ . Let  $F' = (F \cup \{\text{Sync}(w, p, q)\}) \setminus B$ . Return  $F'$ .

If  $F$  is a valid frontier, then so is  $F'$ . Assume that  $F$  upholds all of the required invariants. Invariants 1 and 2 are simple to prove.  $F'$  must be finite because only one formula appears in  $F'$  that does not also appear in  $F$ .  $F$  is finite, so  $F'$  must be finite as well. Also, every formula in  $F'$  is of the correct form because the only formula in  $F'$  that does not also appear in  $F$  is  $\text{Sync}(w, p, q)$ .

$F'$  upholds invariant 3 because  $F$  does. The only new entry in  $F'$  is  $\text{Sync}(w, p, q)$ . Suppose that we have strings  $s_1$  and  $s_2$ , where  $s_1$  is a strict prefix of  $s_2$ , such that  $\text{Sync}(s_1, p, q)$  and  $\text{Sync}(s_2, p, q)$  are in  $F'$ . If neither  $s_1$  nor  $s_2$  is  $w$ , then the pair is also present in  $F$ , which is impossible.

If  $s_1$  is  $w$ , then  $s_2$  must be more than one character longer than  $w$  because  $\text{Sync}(wb, p, q)$  does not appear in  $F'$  for any character  $b$ . Besides those formulas and  $\text{Sync}(w, p, q)$ , every formula in

$F'$  also appears in  $F$ , so  $\text{Sync}(s_2, p, q)$  must appear in  $F$  as well. Let  $c$  be a character and  $t$  be a non-empty string such that  $s_2 = wct$ .  $F$  contains the formula  $\text{Sync}(wc, p, q)$ , so we have a prefix conflict between the formulas  $\text{Sync}(wc, p, q)$  and  $\text{Sync}(wct, p, q)$  in  $F$ , which is impossible.

If  $s_2$  is  $w$ , then  $\text{Sync}(s_1, p, q)$  must appear in  $F$ . Since  $s_1$  is a strict prefix of  $w$ , it must be a strict prefix of  $wc$  as well for any character  $c$ . We know that  $\text{Sync}(wc, p, q)$  appears in  $F$ , so we have a prefix conflict between  $\text{Sync}(s_1, p, q)$  and  $\text{Sync}(wc, p, q)$  in  $F$ , which is impossible. This eliminates all of the possible ways that  $F'$  could violate invariant 3.

$F'$  upholds invariant 4 because, for every  $b \in \Sigma$ , the formula  $\text{Sync}(w, p, q)$  functions as a replacement for  $\text{Sync}(wb, p, q)$ . The string  $w$  is a prefix of  $wb$ , so any string  $s$  of length  $k$  that was covered by  $\text{Sync}(wb, p, q)$  in  $F$  is covered by  $\text{Sync}(w, p, q)$  in  $F'$ .

For invariant 5, we need to confirm that there exists a valid proof of  $\text{Sync}(w, p, q)$ , given that, for every  $b \in \Sigma$ , there exists a valid proof of  $\text{Sync}(wb, p, q)$ . To get a proof of  $\text{Sync}(w, p, q)$ , we can apply our Coinduction rule, taking the proofs of  $\text{Sync}(wb, p, q)$  for every  $b \in \Sigma$  as premises. As an additional premise, we also need a proof that  $E(\delta(w, p)) = E(\delta(w, q))$ . The proof must exist because of Theorem 1.

Now that we have defined  $\text{update}(F)$ , we need to define our algorithm for using it. We start from the initial frontier  $F_0$ . Apply the update function to  $F_0$  repeatedly until the maximum depth of a formula in the frontier is 0. We know that this loop will terminate eventually. Let  $F' = \text{update}(F)$  for some frontier  $F$ . We know from the definition of  $\text{update}$  that  $F'$  is created from  $F$  by adding  $\text{Sync}(w, p, q)$  and removing all formulas of the form  $\text{Sync}(wb, p, q)$  for some string  $w$  and all characters  $b \in \Sigma$ . The string  $w$  is shorter than  $wb$  for any  $b$ , so adding  $w$  to the set does not increase the maximum depth. Also, note that  $F'$  removes  $n = |\Sigma|$  formulas of some depth  $m$  and adds a single formula of depth  $m - 1$ . The frontier must be a finite set, so the formulas of depth  $m$  must be depleted eventually. Once the formulas of depth  $m$  are eliminated, the loop moves to eliminating the formulas of depth  $m - 1$ ,  $m - 2$ , and so on until the formulas in the frontier are all of depth 0.

There is only one possible  $\text{Sync}$  formula of depth 0 that a frontier can contain, namely  $\text{Sync}(\epsilon, p, q)$ . Once we have  $\text{Sync}(\epsilon, p, q)$  in the frontier, we can apply  $\text{SyncEmpty}$  to reach the conclusion that  $p = q$ . When we reach the end, we know that a valid proof exists for  $\text{Sync}(\epsilon, p, q)$ , so adding an application of  $\text{SyncEmpty}$  gives us a valid proof of  $p = q$ .

Since we can always prove  $p = q$  for a pair of equivalent regular expressions using our calculus of rules, we can always make a proof that Crêpe accepts. If we have a proof of  $p = q$ , we can apply  $\text{Assume}$  to derive  $p \neq q$  and then apply  $\text{Contra}$  to derive  $\perp$ .

## D Plaintext Validation

For all three validation configurations that we examine, the time cost of running Crêpe comes almost entirely from our ZK operations rather than the structure of the algorithm itself. If we execute Crêpe on our 301 selected benchmarks with no ZK operations, it runs much more quickly. With all ZK operations disabled, the benchmarks have an average running time of 0.0049 seconds and a median running time of 0.0033 seconds. The standard deviation is 0.0030 seconds. The slowest individual benchmark takes only 0.017 seconds to validate. For comparison, the fastest individual ZK benchmark takes 1.74 seconds.

Received 2025-03-25; accepted 2025-08-12