ABSTRACT
Securing web applications remains a pressing challenge. Unfortunately, the state of the art in web crawling and security scanning still falls short of deep crawling. A major roadblock is the crawlers’ limited ability to pass input validation checks when web applications require data of a certain format, such as email, phone number, or zip code. This paper develops Black Ostrich, a principled approach to deep web crawling and scanning. The key idea is to equip web crawling with string constraint solving capabilities to dynamically infer suitable inputs from regular expression patterns in web applications and thereby pass input validation checks. To enable this use of constraint solvers, we develop new automata-based techniques to process JavaScript regular expressions. We implement our approach extending and combining the Ostrich constraint solver with the Black Widow web crawler. We evaluate Black Ostrich on a set of 8,820 unique validation patterns gathered from over 21,667,978 forms from a combination of the July 2021 Common Crawl and Tranco top 100K. For these forms and reconstructions of input elements corresponding to the patterns, we demonstrate that Black Ostrich achieves a 99% coverage of the form validations compared to an average of 36% for the state-of-the-art scanners. Moreover, out of the 66,377 domains using these patterns, we solve all patterns on 66,309 (99%) while the combined efforts of the other scanners cover 52,632 (79%). We further show that our approach can boost coverage by evaluating it on three open-source applications. Our empirical studies include a study of email validation patterns, where we find that 213 (26%) out of the 825 found email validation patterns liberally admit XSS injection payloads.

CCS Concepts
• Security and privacy → Web application security; Formal methods and theory of security.

Keywords
web application scanning, string constraint solving
Black Ostrich to the rescue. This paper proposes Black Ostrich, a principled approach to deep web crawling and scanning. The key idea is to leverage string-based constraint solving, based on satisfiability modulo theories (SMT) [22], to infer suitable input from the analysis of forms in web applications, including both input types, such as email and URL, and support for regular expression (regex) patterns. While SMT is heavier than prepared inputs from a library, it trades more local computation for fewer network requests. Furthermore, our approach can be fruitfully combined with the traditional scanners by leveraging Black Ostrich to generate input libraries for traditional scanners, based on solving the patterns collected from the wild. SMT has been extensively used for web security for applications like finding SQL injections [43], analyzing and testing JavaScript [62], and detecting server-side parameter tampering [13]. However, these approaches focus on detecting particular vulnerabilities rather than the depth of web crawling. To the best of our knowledge, Black Ostrich is the first to leverage SMT technology for deep web crawling. As such, it requires addressing several research challenges.

1.2 Constraint Solving Challenges

One of the main challenges in deep crawling is to handle the ECMAScript regular expressions used as patterns for input validation [34], like the pattern `.*@.*\.[a-z]{2,3}`. Patterns in web applications can be thousands of characters long and frequently use features like anchors or look-arounds: in our experiments, we find 500 patterns using look-arounds and 4,044 using anchors. The longest pattern we find is a stunning 29,059 characters. Although several SMT solvers have been recently extended to string constraints, including the solvers Z3 [21, 51, 68], S3/p/# [65], cvc5 [5], Norn [1], Sloth [36], and Ostrich [16], up to now no SMT solver directly supports the much richer language of ECMA regular expressions. Developing an SMT solver capable of handling real-world regular expressions is a long-standing challenge [63].

Handling ECMA regular expressions using existing SMT technology is difficult. Anchors and look-arounds have non-compositional semantics, i.e., their effect is unbounded and can affect the complete string to be parsed, which prevents a direct translation to the supported textbook regular expressions [37] of today’s solvers. Look-arounds combined with capture groups and back-references even lead to undecidability of the language emptiness problem [17].

To our knowledge, the only translation of the ECMA regular expression language to SMT-LIB constraints was presented by Loring et al. [48] in the scope of symbolic execution by the ExpoSE tool, applying an abstraction-refinement loop to address the issue of undecidability. Their support of the very commonly used feature of look-arounds, however, is only partial (we provide a detailed comparison in Section 3.2), and our experiments show that the SMT-LIB encoding in ExpoSE turns is a less natural match for the intricate regexes on the web (Appendix G.2).

Solving ECMA regexes. Black Ostrich dynamically generates input data for web pages both for exploring and attacking. For this, we define a translation of the HTML5 validation constraints to logical formulas. We also present, to the best of our knowledge, the first sound and complete solver for ECMA regular expression including support for anchors, look-aheads and look-behinds, capture groups, and back-references. Depending on the phase of scanning, the validation constraints can be complemented by constraints that request the inclusion of payloads like `<script>` tags in the input.

Our starting point is the SMT solver Ostrich [16], an automata-based string solver for constraints in a rich language, including regular expressions, equations, and string functions like replace-all and let-to-letter transduction. Prior to our work, and in line with other SMT solvers, Ostrich could only process regexes in SMT-LIB notation, and did not support ECMA features such as anchors or look-arounds [34]. This paper extends Ostrich with a native parser for ECMAScript regexes, and presents a novel translation of ECMAScript regexes to two-way alternating automata, augmented with a refinement loop to support back-references. Completing the pipeline, we also present a new technique to simulate two-way alternating finite automata by non-deterministic finite automata that enables efficient implementation inside solvers.

1.3 Validation-aware Crawling and Fuzzing

Faced with a form, the crawler must decide what data to submit. The type of data expected by the server can range from numbers to strings to valid emails and URLs. Validation of such constraints can happen both client-side and server-side. To make progress in crawling, it is necessary to pass server-side checks since the provided input will otherwise be rejected by the web application. Traditional scanners source the input from a library with a diverse set of strings, hoping that one will be valid. This approach faces challenges because web applications can have arbitrarily complicated input validation.

Complementary to the traditional techniques, Black Ostrich applies a dynamic approach that takes all available information on the expected input data into account and systematically constructs input data through constraint solving. A key difficulty is that server-side validation is not visible to the crawler. On the other hand, client-side validation is fully under the crawler’s control. In fact, HTML5 provides several attributes for client-side input validation (Appendix A), including a pattern attribute to represent regular expressions, as specified by the ECMA [34], that user input must match for the form to be submitted, and are today commonly used in web applications.

Black Ostrich thus focuses on passing client-side validation constraints, with the hypothesis that successful inputs are likely to also satisfy server-side constraints. Our hypothesis is confirmed by evidence from web frameworks like Spring MVC [39] and ASP.NET [59] that are designed to reuse validation patterns for consistency between client-side and server-side validation. Further evidence from the open-source projects investigated in Section 8 indicates that client-side input validation is aligned with server-side input sanitization based on the same patterns.

Our main goal is to boost the code coverage of a web application thanks to the constraint solving capabilities of Black Ostrich. The improved code coverage enables deeper web crawling, which is important for finding new vulnerabilities. In addition, our technique can be also leveraged to detect immediate XSS vulnerabilities where input validation checks miss XSS payloads, like those exploiting email XSS payloads [38, 58]. Our focus on input validation is justified because according to the OWASP guidelines input validation...
We sample uniformly from the 64,000 archive parts, collecting forms We demonstrate that Black Ostrich boosts both code coverage and works well for custom regex-based JavaScript validations. Many validations used on popular websites, we combine this with a crawl validation patterns from the July 2021 Common Crawl archive [18]. vulnerability detection, compared to state-of-the-art crawlers/scan- ing the scanners on real websites. Indeed, even without the attack module, running a scanner can cause damage to the website in the form of forum posts, product reviews, purchases, etc. Instead, we create a testbed based on real-world validation.

The elegance of our approach is that we can extend it to handle JavaScript-based input validation. Indeed, we dynamically extract regex tests on our inputs and update the inputs accordingly. This works well for custom regex-based JavaScript validations. Many popular validation libraries, including jQuery Validate, rely on regex to validate predefined types such as email, allowing us to solve it. Clearly, JavaScript can use other methods for validation outside our coverage. Yet due to the ease of use of HTML5 patterns, they are likely to become increasingly common in the future, as they are indeed designed to replace JavaScript validation. Note that like most scanners, we consider out-of-band validation, e.g. 2FA and SMS validation, as out of scope.

1.4 Empirical Studies

We demonstrate that Black Ostrich boosts both code coverage and vulnerability detection, compared to state-of-the-art crawlers/scanners including Arachni, Enemy of the State, jÄk, ZAP, and Black Widow. The obvious ethical reasons prevent us from directly running the scanners on real websites. Indeed, even without the attack module, running a scanner can cause damage to the website in the form of forum posts, product reviews, purchases, etc. Instead, we create a testbed based on real-world validation.

To test Black Ostrich in a realistic environment, we harvest input validation patterns from the July 2021 Common Crawl archive [18]. We sample uniformly from the 64,000 archive parts, collecting forms from 8,266,577 URLs, in total 21,667,978 forms. To also capture validations used on popular websites, we combine this with a crawl of Tranco [46] top 100K.

Using the combined data from Common Crawl and Tranco we extract 881,329 HTML5 patterns which after de-duplication results in 9,805 patterns, all used in the wild. After removing broken and invalid patterns we have a total of 8,820. We create a testbed of mock websites using these patterns both on the client-side and server-side and evaluate the coverage for the state-of-the-art web crawlers. Our scanner shows a significant improvement by being able to solve 99% of the patterns compared to an average of 36% for the other scanners. As many websites share the same patterns [35], we also analyze how many domains we can improve coverage on. Comparing the number of domains using these patterns, we solve all patterns on 66,309 out of the total 66,377 domains. We subsume, i.e. solve everything the other scanners solve, and improve, i.e. solve something they miss, coverage on over 13,711 domains compared to the combined efforts of previous scanners.

We use the same testbed, which includes an input reflection if the server-side check is passed, to test vulnerable patterns that could allow for XSS. The results show an increase of 52% in vulnerability detection compared to the other scanners. We find 863 vulnerable patterns compared to an average of 594 vulnerable patterns for the other scanners.

We perform a manual analysis of the top 100 websites that use input validation and report on the input validation methods used. We demonstrate that our approach can handle 86% of these methods.

Open-source software. We explore the use of patterns in open-source web applications from GitHub. We download over 900 projects and analyze their use of patterns. We perform a case study analysis on three applications that use both client-side and server-side validation. Our head-to-head comparison of the scanners shows that we increase coverage by passing input validation.

Email pattern study. We report on an empirical study of 825 email patterns extracted from the Common Crawl dataset of real-world web pages. The study reveals remarkable inconsistencies in the current practices of email validation. We illustrate a significant diversity among the commonly used patterns, suggesting that many developers hand-craft email validating patterns. Further, we find that 213 (26%) out of the 825 found email validation patterns liberally admit XSS injection payloads that are exploited in the wild [38, 58]. These experiments illustrate how our regular expression semantics encoding is versatile and efficient enough to handle complex real-world regular expressions for practical applications.

The contributions of the paper are:

- We develop a novel platform for validation-aware web crawling and scanning (Section 2).
- We propose a new version of two-way alternating finite-state automata, 2AFA_SM (Section 4), and a simple yet efficient simulation of 2AFA_SM using standard non-deterministic finite-state automata, NFA (Section 5).
- Based on 2AFA_SM, we define the first sound and complete algorithm for computing solutions of ECMA regular expressions. This translation enables us to extend the state-of-the-art solver Ostrich with native support for ECMA regular expressions (Section 3 and Section 4).
- We evaluate the coverage and vulnerability detection (Section 6), showing that our scanner solves 99% of the patterns compared to the average of 36% for the other scanners. We improve the detection of vulnerable patterns by 52% (Section 7).
- We investigate the usage of HTML patterns in open-source web applications and demonstrate increased coverage thanks to string solving (Section 8).
- We present a case study of email validation patterns, pointing out common inconsistencies and vulnerabilities related to email patterns on the web (Section 9).

We open-source the code of our implementation and all gathered patterns [27].

2 VALIDATION-AWARE SCANNING

To improve the coverage and vulnerability detection we propose a design where the scanner uses a string solver to generate inputs. This empowers the scanner to submit the correct data type thus, potentially, improving coverage. The solver can also generate data matching both patterns and payloads. Figure 1 shows how to extend a scanner to interact with SMT solvers.
2.1 Motivating Example

This section walks through an example of where patterns are used. The scanner’s crawler requests a page, \( \circ \) in Figure 1. We mark the crawler as dashed in the figure to highlight that this can be any off-the-shelf crawler. The page it crawls uses \( .@.*\).. to validate emails. In step \( 2 \) the crawler sends the response and patterns to the validation controller, which extracts the patterns from the web page. This also includes dynamic interaction with the page to extract regex use in JavaScript. Before the scanner submits this form it picks a witness by calling the witness controller in step \( 3 \). It decides what type of data to send, e.g., a username, unique data token, XSS payload, etc. It looks up the elements it needs to submit in the validation controller in step \( 4 \). The validation controller returns the pattern, i.e., \( .@.*\).. . The next step depends on if the scanner is in the crawl phase or attack phase.

**Crawl Phase.** The witness controller sends the pattern directly to the SMT in step \( 5 \). The HTML5 pattern will then be parsed, translated to an automaton, and sent to the solver in step \( 6 \). The solver finds a string matching the pattern, e.g., \( \emptyset .\emptyset . \emptyset . \emptyset . \emptyset . \emptyset \). It returns the solution to the witness controller, step \( 7 \), which returns it to the crawler, step \( 8 \), where it is submitted to the application, step \( 9 \).

**Attack Phase.** The witness controller calls the payload generator to get a payload in step \( 10 \). The payload generator chooses a payload, commonly from a pre-defined list, e.g., \( \texttt{<script>alert(1)\</script>\text{y}} \). In step \( 11 \), the witness controller sends both the pattern and the payload to the SMT. Both are encoded and sent to the solver in step \( 12 \). The solver generates a valid solution to the pattern that also contains the payload and sends it back in \( 13 \). The solution \( \texttt{<script>alert(1)\</script>\emptyset .\emptyset .\emptyset .\emptyset .\emptyset .\emptyset .\emptyset .\emptyset .\emptyset \) matches the pattern and contains the payload. If no solution exists, we fall back to the payload. Finally, the witness controller sends it to the crawler, step \( 14 \), which submits it to the web page in step \( 15 \).

2.2 Scanning

To find vulnerabilities in a web application the scanner must be able to explore the application in a meaningful way and attack the application.

**Crawling.** As JavaScript is ubiquitous on the web, traditional crawling by statically parsing HTML is no longer enough. The modern scanners model and execute JavaScript and events, as pioneered by jÅk [57]. In addition to handling dynamic client-side interactions, server-side code must also be considered. The server-side code, which is not accessible to scanners, is responsible for authentication, posting comments, etc. This is important to handle as some actions, e.g., adding a comment can result in new parts of the application to explore. Therefore, modern scanners infer the server-side state and model actions and their effects, as pioneered by the Enemy of the State [23]. While our general method of combining a scanner and string solver works for any scanner, we choose to build on the Black Widow [26] scanner in this paper. Black Widow combines the advantages of jÅk, Enemy of the State, and other scanners. We improve on Black Widow by adding features that allow our scanner to interpret and solve input validation patterns.

**Input validation with patterns and JavaScript.** Web applications use input validation to ensure the correctness of users’ input. Many websites perform client-side validation, often using the HTML5 pattern attribute or regex-based JavaScript functions like \( \text{RegExp.test} \). Scanners can use this to infer the server-side validation. To find the client-side validation patterns we instrument the scanner to extract both the pattern attribute and other validation attributes (Appendix A) from input elements and add them to the navigation graph. In addition, we proxy JavaScript regex functions and dynamically interact with the page to extract the used expressions. We present more details about this in Section 6.3.1. Whenever the scanner needs a value for an input element, it will fetch the pattern for the navigation graph and use the SMT solver to find a matching string.

**Fuzzing.** An effective method for detecting XSS is executing JavaScript and searching for the expected runtime behavior of the
payload, for example, showing an alert with the text “XSS”. To further minimize false positives the payloads must be unique to each input parameter as stored payloads might be reflected in multiple places. The Black Widow scanner uses unique payloads and dynamic injection detection already minimizing the false positives. However, the payload is limited to unique numeric IDs. That is, the payloads execute $\text{xs}(123)$, where 123 will be changed for each payload. As some validation mechanisms might reject numbers, we extend Black Widow to also handle alphabetic IDs.

The generated payloads should also match any validation patterns. Recall the real-world pattern $^*\text{France}$, where the payload must end with France. To generate a payload, the payload generator will use an XSS payload with a unique ID. The string solver then creates a valid string with this payload. Using the pattern above, a possible solution is $\langle\text{script}\rangle\text{xs}(123)\rangle\langle\text{script}\rangle\text{France}$ While our focus is on XSS, the same method can be used to generate valid SQLi payloads, e.g. `DROP TABLE;--France`

### 3 HANDLING VALIDATION CONSTRAINTS USING SMT

The next sections introduce the SMT component of Black Ostrich in more detail, namely, the dashed box labeled as “SMT” in Figure 1. Black Ostrich builds on the existing state-of-the-art string solver Ostrich [16], but extends it for security scanning. Ostrich supports constraints formulated using the SMT-LIB theory of Unicode strings [7], in particular, regular expression membership assertions, and string functions including concatenation, substring, and replace. In addition, Ostrich accepts all functions that can be represented as finite-state transducers. Ostrich also has all the standard features of an SMT solver, for instance, handling of Boolean structures as well as support for other theories like integers and arrays. Given a set of assignments and assertions, Ostrich finds a model, that is, assignments of concrete strings to variables, or reports that the given formulas are inconsistent.

#### 3.1 ECMAScript Regular Expressions

For scanning and fuzzing, Black Ostrich translates the web application’s validation constraints into SMT-LIB constraints. A list of HTML5 validation attributes used by Black Ostrich is given in Appendix A; in this paper, we focus on fields with the pattern attribute, which enables web developers to specify further constraints on textual input using ECMAScript regular expressions [34]. Such regular expressions offer several features not present in traditional, textbook regular expressions: (i) anchors $^*$, $^+$ that check for the beginning or end of a string; (ii) look-aheads and look-behinds, which constrain accepted strings without consuming any characters; (iii) capture groups and back-references to the contents of those groups; (iv) greedy and lazy matching.

**Example 1.** A regular expression commonly used as a pattern for passwords is [66]:

$$^?(?:[\text{\^}\text{-}\text{@}\text{\$}\text{\{\text{|}\text{~}\text{\+}\text{-}\text{/}\text{=}\text{\^}\text{_}\text{\{}}\text{\}]}{1,63}])\text{France}$$

The assertions $^{\ldots}$ are positive look-aheads, and mandate that a password has to contain at least one digit, one lower-case letter, and one upper-case letter. The negative look-ahead $^{\ldots}$ forbids whitespace characters.

As a second real-world example, among the patterns considered in Section 6.1, we observed the following regular expression describing email addresses:

$$^?\text{[\text{\^}\text{-}\text{@}\text{\$}\text{\{}}{1,63}])\text{France}$$

The look-ahead is in this case used to restrict the local-part to at most 64 characters.

Note that, although present, the $^*$ (beginning) and $^+$ (end) anchors are not necessary in these regexes because the pattern attribute anyway requires full-string matching. This is in contrast to common server-side regex mechanisms that are based on substring matching. We will come back to this subtlety in Section 9.3.

We introduce our method to handle both anchors (i) and look-arounds (ii), based on two-way alternating automata, in Section 4 and Section 5. Back-references (iii), when combined with look-arounds (ii), lead to undecidability [17], but can be handled using an abstraction-refinement loop [48]. We discuss in Section 4.5 how such a refinement loop can be integrated into our framework. Greediness (iv) of matching is not relevant for HTML patterns, and therefore not considered in this paper: greediness affects the length of matched substrings, and the contents of capture groups, but it does not influence the overall language described by a regex.

#### 3.2 Previous Results for ECMAScript Regexes

Loring et al. [48] present a symbolic execution tool for JavaScript, ExpoSE, which can also handle ECMAScript regexes, excluding look-arounds. Since the language emptiness problem of this full language is undecidable [17], ExpoSE applies an abstraction refinement loop: initially, regexes are translated to SMT-LIB regular expressions (aka textbook regular expressions), which are supported by many SMT solvers. This translation is over-approximate, so the resulting constraints might have solutions even though the original regex described an empty language. Such spurious solutions are eliminated iteratively through refinement.

The ExpoSE translation of regexes [48] leads to complex formulas combining word equations, SMT-LIB regular expressions, and Boolean structure; in our experiments, we observed that the formulas are often taxing for SMT solvers. In addition, as defined in [48], the translation does not yield correct over-approximate constraints in some cases involving look-arounds. In particular the interaction of alternation and look-arounds, or of repetition (Kleene star) and look-arounds, is not correctly modelled, leading to an incorrect encoding of regular expressions like $(?=(a+x))x$. This regex is equivalent to a$x, but the translation defined in [48] interprets the regex as defining the language $x$. We conjecture that this issue is inherent in the strategy of directly translating ECMAScript regexes to SMT-LIB constraints, since a correct translation needs to handle the unboundedly many look-arounds $(?=a+x)$ caused by the outer Kleene star, which can most naturally be done in a finite-state automata setting.

Our approach has some similarities with recent work on translating regular expressions with look-arounds to Boolean automata, studying in particular the computational complexity [10]. In contrast to [10], this paper considers regexes with both look-arounds and look-behinds, as well as all other features of ECMA regular
expressions, and uses the formalism of two-way alternating automata [45].

4 TWO-WAY ALTERNATING AUTOMATA FOR REGEXES

We now introduce our approach to correctly handle ECMAScript regexes, implemented in exactly the same way in the symbolic setting, representing vals. From this symbolic NFA, candidate solution strings can be extracted. To compensate for over-approximation of back-references, we introduce a new variant of 2AFA, named 2AFA\textsubscript{SMT}, that is particularly suited for representing ECMAScript regexes. The encoding as 2AFA handles back-references by over-approximation (Section 4.5), and initially keeps character ranges symbolic. Character ranges are in the next step turned into concrete characters by applying the known Minterm transformation [20]. 2AFA\textsubscript{SMT} are translated further to NFA (Section 5), and then to a symbolic NFA by expanding Minterms to intervals. From this symbolic NFA, candidate solution strings can be extracted. To compensate for over-approximation of back-references, the correctness of the solution string has to be checked against the original regex; in case spurious solutions are detected, the 2AFA\textsubscript{SMT} is refined.

This overall algorithm is sound, in the sense that it will only compute genuine solutions of regular expressions, and complete in the sense that it will eventually find a solution whenever there is one. Unless a regular expression contains back-references, the algorithm is also guaranteed to terminate; with back-references, due to undecidability it is no longer possible to guarantee termination.

4.2 Basic Definitions

For ease of presentation, we adopt a mathematical notation and we focus on a core set of regular expression operators. We also present our translation of regexes to 2AFA\textsubscript{SMT} in the context of a finite alphabet \( \Sigma = \{ \sigma_1, \ldots, \sigma_n \} \); the translation to 2AFA\textsubscript{SMT} works in exactly the same way in the symbolic setting, representing character ranges using intervals. The set of textbook regexes \( R \) is then inductively defined as follows [37]:

\[
\rho ::= \emptyset | \varepsilon | \sigma | \rho^* | \rho_1 \cdot \rho_2 | \rho_1 + \rho_2 \]

where \( \sigma \in \Sigma \), \( \bar{\tau} \) is the complement of \( \tau \), \( * \) is the Kleene star operator, and \( \cdot \) and \( + \) are the usual concatenation and alternation operators, respectively. We also define syntactic shortcuts, namely \( \rho_1 \cap \rho_2 := \rho_1 + \rho_2 \) and, with slight notational abuse, \( \Sigma := \sigma_1 + \ldots + \sigma_n \).

On the other hand, the set of augmented regexes \( \mathcal{R} \) include the features (i) and (ii) from Section 3.1, but they lack complementation, and they are inductively defined as follows:

\[
\rho ::= \emptyset | \varepsilon | \sigma | \rho^* | \rho_1 \cdot \rho_2 | \rho_1 + \rho_2 | (\geq \rho) | (\leq \rho) | (\prec \rho) | (\succ \rho) | ^\dagger | | S | (\rho)_n | \text{n}
\]

where \((\geq \rho)\) and \((\leq \rho)\) are the positive and negative look-ahead operators, which check if \( \rho \) matches, resp., does not match, a prefix of the suffix of the string, without consuming any symbols. \((\prec \rho)\) and \((\succ \rho)\) are the positive and negative look-behind operators, which, analogously to the previous ones, check if \( \rho \) matches in the part of the string that has already been analyzed. Anchors \(^\dagger\) and \( S \) are true only at the beginning, resp., end of the string. Capture groups \((\rho)_n\) match the same strings as \( \rho \), but in addition record the matched sub-string, which can subsequently be back-referenced using \( \text{n} \). It is assumed that at most one capture group \((\rho)_n\) exists for each index \( n \). Appendix B formally defines the language \( L(\rho) \subseteq \Sigma^* \) described by an augmented regex \( \rho \).

4.3 Two-way Alternating Automata

2AFA are machines that read input words [45]. They are two-way in that they can scan the input both left-to-right and right-to-left, and alternating, meaning that they can take both existential (\( \exists \)) and universal (\( \forall \)) transitions. An \( \exists \)-transition corresponds to the transitions in a standard NFA: from some state, the automaton can transition to one out of multiple possible successor states. For the automaton to accept the word, it is enough if one such execution is successful. Conversely, \( \forall \)-transitions fork the execution to a set of paths that should all be successful. For both kinds of transitions, the automaton also specifies if it is moving forward or backward, with one exception: when a transition is an \( \varepsilon \)-transitions, no symbols are read/consumed and therefore the automaton does not move on the word.

It is well known that 2AFA have the same expressive power as standard NFA, although being exponentially more succinct, and indeed the former can be simulated by the latter [12, 31, 41]. These algorithms, however, besides having exponential complexity, are also quite intricate and have never been implemented in the context of SMT solvers, to the best of our knowledge. We, therefore, introduce a new version of 2AFA, which we call 2AFA\textsubscript{SMT} with the following features: (i) their semantics is closer to the semantics of ECMAScript regex, thus enabling a more direct representation of those and (ii) they allow for a simple and practically efficient translation to NFA. The main difference between traditional 2AFA and 2AFA\textsubscript{SMT} is on the way transitions are specified. The former
reads the character they are currently analyzing and then moves either forward or backward positioning themselves on the respective character, while the latter sits in-between characters, and they can either read the preceding one and move backward, or the succeeding one and move forward. This is obtained by having two different kinds of transitions, the backward $\delta^\prec$ transitions and the forward $\delta^\succ$ ones.

Definition 1 (2AFA$^\text{SMT}$). A two-way alternating automaton is a tuple $(\Sigma, S, s_0, F^\prec, F^\succ, \delta_1^\prec, \delta_2^\prec, \delta_1^\succ, \delta_2^\succ, \epsilon_3, \epsilon_4, \epsilon_\forall)$ where:
- $\Sigma$ is an alphabet of symbols;
- $S$ is a finite set of states;
- $s_0 \in S$ is an initial state;
- $F^\prec, F^\succ \subseteq S$ are disjoint sets of final states;
- $\delta_1^\prec, \delta_2^\prec, \delta_1^\succ, \delta_2^\succ : S \times \Sigma \to \phi(S)$ are partial existential ($\exists$) and universal ($\forall$) transition functions, respectively;
- $\epsilon_3, \epsilon_4 : S \to \phi(S)$ are partial $\epsilon$-existential ($\exists \epsilon$) and $\epsilon$-universal ($\forall \epsilon$) transition functions, respectively,
and $\phi(S)$ is the powerset of $S$. We require that for every state $s \in S$ and $\sigma \in \Sigma$ one of the $\delta^\prec$ or $\delta^\succ$ transitions is defined.

Next, we define the semantics of an automaton, namely the set of words it accepts.

Definition 2 (2AFA$^\text{SMT}$ run). Let $w = w_0 w_1 \ldots w_n$ be a word in $\Sigma^\star$ of length $\ell(w) = n + 1$, and $A$ be 2AFA$^\text{SMT}$. A run $\pi$ of $A$ on $w$ is a finite sequence of elements in $\phi(S \times \mathbb{N})$, called configurations, defined inductively: $\pi_0 := \{(s_0, 0)\}$ and for any $\pi_j$ we build the successor configuration $\pi_{j+1}$ as follows. Pick $(s, i) \in \pi_j$, then:

- $(s', i+1)$ if $s' = \delta_2^\prec(s, w_i)$ and $i < \ell(w)$;
- $(s', i-1)$ if $s' = \delta_2^\prec(s, w_{i-1})$ and $i > 0$;
- $(s', i+1)$ if $s' = \delta_2^\succ(s, w_i) = S'$ and $i < \ell(w)$;
- $(s', i-1)$ if $s' = \delta_2^\prec(s, w_{i-1}) = S'$ and $i > 0$;
- $(s', i)$ if $s' = \epsilon_3(s)$;
- $(s', i)$ if $s' = \epsilon_4(s)$.

Intuitively, a state/index pair $(s, i)$ expresses that the automaton is in state $s$ and in between the $i = 1$-th and $i$-th characters of $w$. Being alternating, we might have more than one pair at any moment, as the automaton is scanning multiple parts of the word at the same time. We start from the initial state $s_0$ at the beginning of the word, pair $(s_0, 0)$, and at each step a pair is picked and a transition is performed: if such transition is existential, then the current state is updated with one of the successor states; if it is universal, all the successor states are added to the current run. The index is updated depending on if the transition is moving backward < or forward >.

We say that automaton $A$ accepts word $w$ if there exists a run $\pi = \pi_0 \pi_1 \cdots \pi_k$ of $A$ over $w$ in which the last configuration is accepting, that is: for each $(s, i) \in \pi_k$ we have either $s \in F^\prec$ and $i = 0$ or $s \in F^\succ$ and $i = \ell(w)$.

4.4 Translation of Augmented Regexes

Our procedure recursively constructs a 2AFA$^\text{SMT}$ $A_\rho$ for each augmented regex $\rho \in \mathcal{R}$. Compared to the constructions for textbook regexes [37], ours adds cases for handling look-arounds and anchors. We notice that the latter can be seen as shortcuts: it is indeed easy to prove that $\wedge$ is equivalent to $(\exists \Sigma)$ and $S$ is equivalent to $(\exists \Sigma)$. We discuss the main cases of the translation in this section and refer the reader to Appendix B for further details.

Example 2. Consider the regex $(a+b)^+ \cdot (< b \cdot a)$. The regex is translated to the automaton in Figure 3, and illustrates the translation of concatenation, the $+$ operator, and a look-behind. When running on $w = abbab$, a successful execution sees the sub-automaton $A_{(a+b)^+}$ matching the whole word and ending in state $s_8$. From there, the execution forks: one path directly accepts in $s_{9k} \in F^\prec$, while the other goes through sub-automaton $A_{ab}$, which starts scanning backward. It first reads $b$ and then $a$, which indeed matches $(< a \cdot b)$ thus ending up in the final state $s_8 \in F^\prec$. Since both paths are in a final state, word $w$ is accepted.

Intuitively, the automaton translation works as follows (see Appendix B for more details and figures). The automata for atomic cases of $\epsilon, \sigma$ accept after seeing $\epsilon$ or $\sigma$, respectively. The automaton for $\emptyset$ never accepts. The automaton for alternation forks the execution with $\exists$-transition into two paths, each attempting to match a subexpression. The automaton for concatenation connects with an $\epsilon$-transition the automata for the sub-expressions. In the automaton for the Kleene star, an initial $\exists$-transition forks the execution into two paths, one directly accepting, the other matching one iteration of the sub-expression $\rho$, and then moving back to the initial state using an $\epsilon$-edge.
The novel case of lookahead (> \rho) builds the automaton schematized in Figure 4, in which the box is the automaton for \rho, double-circled states s_f, s_r \in F^\rho are final, and the outgoing transitions from s_\ell are V-transitions. This initial V-transition forks the execution in two paths that should both accept. The final state s_f, s_r will be recursively expanded into an automaton that recognizes the remaining regex.

When a look-behind is encountered, the same idea holds, but the automaton inside the box scans the word backward, hence the necessity of a two-way automaton. We reverse the regex inside the look-behind, hinging on the fact that scanning a word w backward from end to start is equivalent to scanning the reverse of w forward from the beginning. Negated look-arounds are handled by complementing the sub-automata in the boxes [30].

Capture groups (\rho)_n are translated like \rho, while there are two cases for back-references \setminus n: a back-reference occurring under an even number of negations \setminus or \exists is over-approximated by the regex \rho of the group (\rho)_n it references, while a back-reference under an odd number of negations is translated like the empty language \emptyset.

**Theorem 1.** Let \rho \in R be an augmented regex, and A_\rho be the two-way alternating automaton constructed from \rho. For every w \in \Sigma^\ast, if w \in L(\rho), then automaton A_\rho accepts w as well. If \rho does not contain back-references, then w \in \Sigma^\ast if and only if w \in L(\rho).

A proof is given in Appendix B. We also remark that:

**Lemma 1.** Building A_\rho for a regex \rho \in R without back-references takes linear time in the size of \rho.

Translating a regex \rho with back-references to an automaton A_\rho can in general be exponential in the nesting depth of the contained capture groups. In practice, nesting depth tends to be small.

### 4.5 Refinement Loop for Back-References

Theorem 1 guarantees the equivalence of an augmented regex \rho and its corresponding automaton A_\rho only if \rho does not contain back-references.

**Example 3.** Consider the regex \rho = (a+b|\emptyset), which describes the language L(\rho) = \{ab, bb\}. The automaton A_\rho recognizes the language \{aa, ab, ba, bb\}, and strictly over-approximates the regex.

It is possible to detect spurious words accepted by A_\rho, because although emptiness of L(\rho) is undecidable, the membership problem w \in L(\rho) is decidable for any concrete string w \in \Sigma^\ast. Any regex engine, for instance, the implementation in Nodejs, can be used to verify the correctness of solutions. This observation is also used in ExpoSE [48], and in our settings yields a complete refinement loop for computing solutions even in the presence of back-references.

In Figure 2, after computing a candidate solution w, its correctness is checked against the original regex \rho. In case w \notin L(\rho), i.e., w is a spurious word, the symbolic 2AFA\textsubscript{SMT} A has to be refined to an automaton A’ no longer accepting w, and afterwards new solution candidates can be computed. Different refinement methods are possible; the simplest approach is to derive A’ by intersecting A with an automaton recognizing \Sigma^\ast \setminus \{w\}. In the setting of 2AFA\textsubscript{SMT}, this intersection can be done in time linear in |w|. More sophisticated refinement, potentially eliminating multiple spurious solutions, can be achieved by extracting the substrings of w matched by the capture groups, and intersecting A with an automaton that ensures consistency of capture groups with back-references for those specific strings.

It is easy to ensure completeness of the overall algorithm, i.e., the ability to compute solutions whenever the considered language L(\rho) is non-empty. For this, it is only necessary to always compute shortest solution candidates w, which can be done by computing shortest accepting paths of the derived symbolic NFA. This implies fairness of the solution enumeration and guarantees that no solutions are missed.

### 5 SIMULATION OF 2AFA\textsubscript{SMT} BY NFA

We now define a translation from 2AFA\textsubscript{SMT} to a standard NFA.

#### 5.1 Overview

An NFA is an automaton scanning a word left-to-right, with possibly a non-deterministic transition function and \varepsilon-transitions. More precisely, an NFA is a tuple (\Sigma, Q, q_0, \delta, \varepsilon, \tau) in which q_0 \in Q is the initial state, Q_f \subseteq Q is the set of final states, and \delta : Q \times \Sigma \to \wp(Q) and \varepsilon : Q \to \wp(Q) are transition functions. The semantics of NFA is defined similarly to that of 2AFA\textsubscript{SMT}. A run \pi = \pi_0\pi_1\cdots\pi_\ell of an NFA over a word w is finite sequence of configurations from Q \times \Sigma^\ast defined inductively: \pi_0 = (q_0, 0), and for any \pi_j = (q, i) we have \pi_{j+1} = (q', i') with either (i) q' \in \delta(q, i) and i' = i; or (ii) i < (w, q' \in \delta(q, w(i)) and i' = i + 1. A run is accepting if the final configuration is (q, (\ell(w)) with q \in Q_f.

Similarly to other existing methods for transforming 2FA into an NFA [12, 31, 41], our approach is inspired by the original Shepherdson’s construction [64] for eliminating bidirectionality, and the powerset construction for removing the universal transitions. Our translation differs from the existing methods in that we apply a one-step powerset construction, which is intuitive yet efficient in practice. The intuition behind our approach is to categorize 2AFA\textsubscript{SMT} states based on the direction, left-to-right > or right-to-left <, from which they can be reached, and the direction they can be left. We denote the former with a superscript and the latter with a subscript. For example, a state belonging to S^\ast can be reached only with left-to-right transitions (> in the superscript) and can be left with right-to-left transitions (< in the subscript). Such a categorization is required to define the simulating NFA.

**Definition 3** (Simplified 2AFA\textsubscript{SMT}) A simplified 2AFA\textsubscript{SMT} or S-2AFA\textsubscript{SMT} is a tuple (\Sigma, S^\ast, S^\ast, S^\ast, S^\ast, S^\ast, s_\ast, F^\ast, \delta^\ast, \delta^\ast, \delta^\ast, \delta^\ast) in which:

- transition functions \delta^\ast, \delta^\ast, \delta^\ast, \delta^\ast are as in Definition 1;
- the sets of states S^\ast, S^\ast, S^\ast, S^\ast, S^\ast, S^\ast, (s_\ast) are pairwise disjoint, and we denote with S their union;
- s_\ast is the initial state, which does not have incoming transitions and only has outgoing left-to-right transitions: for each s' \in S, s \in S, * \in \{\exists, \forall\}, and * \in \{<, >\} we have that: s_\ast \notin \delta^\ast(s', s), and \delta^\ast(s, s') and \delta^\ast(s, s) are undefined;
- S^\ast is the set of sink states, which only have incoming left-to-right transitions: for each s \in S^\ast, s' \in S, \sigma \in \Sigma and * \in \{\exists, \forall\} we have that: s \notin \delta^\ast(s', \sigma), and \delta^\ast(s, \sigma) and \delta^\ast(s, \sigma) are undefined;
- F^\ast = S^\ast are final states;
for each state \( s, s', \sigma \in \Sigma \) and \( \epsilon \in \{\exists, \forall\} \) we have
\[ \delta_s^\epsilon(s', \sigma) \text{ and } \delta_s^\epsilon(s, \sigma) \text{ is undefined. Analogous} \]
definitions hold for \( S_3^\epsilon, S_4^\epsilon, S_5^\epsilon \),

We notice that any \( 2AFA_{SMT} \) can be transformed into a \( S-2AFA_{SMT} \),
and refer to the extended version of the paper Appendix B for the
procedure.

5.2 Simulation of \( S-2AFA_{SMT} \)
Next, we show how to build an NFA that is equivalent to a
\( 2AFA_{SMT} \). For our approach to work, we make a further, standard

assumption about the considered \( 2AFA_{SMT} \): we say that an automaton
\( A \) is non-cycling if, for every word \( w \), the set of (accepting or non-accepting) runs according to Definition 2 on \( w \) is finite.
This means that runs of \( A \) eventually either get stuck or terminate
in accepting configurations. It can be observed that any \( 2AFA_{SMT} \)
built from a regex in Section 4.4 is non-cycling.

States of the NFA simulating a \( S-2AFA_{SMT} \) are sets of \( S-2AFA_{SMT} \)
states, which we call macro-states henceforth, and transitions are
defined by suitably considering each category of states. The left-
hand side of Figure 5 pictures a run of a \( S-2AFA_{SMT} \) on word
\( w = \sigma_1 \sigma_2 \sigma_3 \sigma_4 \), where automaton states are in-between characters and on the dashed vertical lines. Starting from the state \( s_5 \), the
automaton reads three characters moving right (\( s_2, s_3 \in S_2^\epsilon \)) and lands in state \( s_4 \in S_2^\epsilon \); then it moves backward on \( s_3 \in S_2^\epsilon \) and ends up in \( s_6 \in S_3^\epsilon \), and then finally moves forward to the end of the word accepting in \( s_8 \in S^\epsilon \). The simulating NFA scans instead the word left-to-right only once, essentially guessing at each step the possible (forward and backward) computations of the \( S-2AFA_{SMT} \), as depicted on the right-hand side of Figure 5. Grey boxes represent the macro-states of the NFA.

Definition 4 (Simulating NFA). Let \( (\Sigma, S_2^\epsilon, S_3^\epsilon, S_4^\epsilon, S_5^\epsilon, S^\epsilon, s_\text{start}, P^\epsilon, \delta_2^\epsilon, \delta_3^\epsilon, \delta_4^\epsilon, \delta^\epsilon) \) be a \( S-2AFA_{SMT} \). Next, the equivalent NFA is
\( (\Sigma, \varphi(S), \{s_\text{start}\}, \{F \mid F \subseteq P^\epsilon\}, \delta, \epsilon) \), with \( \delta, \epsilon \) defined as follows. For every \( Q, Q' \in \varphi(S) \), we have
\( Q' \in \epsilon(Q) \) if and only if either:

1. there is \( s \in S_2^\epsilon \) such that \( Q' = Q \cup \{s\} \),
2. there is \( s \in S_3^\epsilon \) such that \( Q' = Q \setminus \{s\} \).

For each \( Q, Q' \in \varphi(S) \) and \( \sigma \in \Sigma \), we have
\( Q' \in \delta(Q, \sigma) \) if and only if all of the following conditions hold:

3. No sinks in \( Q \): \( Q \cap (P^\epsilon \cup S_2^\epsilon) = \emptyset \);
4. No sources in \( Q' \): \( Q' \cap (\{s_\text{start}\} \cup S_3^\epsilon) = \emptyset \);
5. Right-successors:
\[ \forall s \in Q \cap (S_2^\epsilon \cup S_3^\epsilon \cup \{s_\text{start}\}) \colon \]
\[ Q' \cap \delta_2^\epsilon(s, \sigma) \neq \emptyset \text{ or } \delta_3^\epsilon(s, \sigma) \subseteq Q' \; \]

6. Left-successors:
\[ \forall s' \in Q' \cap (S_3^\epsilon \cup S_4^\epsilon) \colon \]
\[ \delta_2^\epsilon(s', \sigma) \cap Q \neq \emptyset \text{ or } \delta_3^\epsilon(s', \sigma) \subseteq Q \; \]

7. Right-predecessors:
\[ \forall s \in Q \cap (S_2^\epsilon \cup S_3^\epsilon \cup \{s_\text{start}\}) \exists s' \in Q' \colon \]
\[ s \in \delta_2^\epsilon(s', \sigma) \text{ or } s \in \delta_3^\epsilon(s', \sigma) \subseteq Q \; \]

8. Left-predecessors:
\[ \forall s' \in Q' \cap (S_3^\epsilon \cup S_4^\epsilon \cup F^\epsilon) \exists s \in Q \colon \]
\[ s' \in \delta_3^\epsilon(s, \sigma) \text{ or } s \in \delta_2^\epsilon(s, \sigma) \subseteq Q' \; \]

The conditions on the transition functions follow from the shape of
the \( S-2AFA_{SMT} \) runs. For example, referring to Figure 5, we have
that states in \( S_3^\epsilon \), such as \( s_6 \), can “appear” in a macro-state, \( Q_2 \) in this case, thanks to \( \epsilon \) transitions as dictated by condition 1 in
Definition 4 (analogously, \( S_4^\epsilon \), such as \( s_4 \), can disappear). However, if they appear, then a state they come from should exist from the right (condition 7) as well as one where they go to, again to the right (condition 5). Similar conditions hold for \( S_3^\epsilon \) states, while for \( S_2^\epsilon \) and \( S_4^\epsilon \) states we simply require the existence of successor(s) and a predecessor on the right or on the left, respectively.

Theorem 2. For any word \( w \in \Sigma \), \( w \) is accepted by a non-cycling
\( S-2AFA_{SMT} \) if \( w \) is accepted by its simulating NFA.

The proof is provided in Appendix C.

6 COVERAGE AND VULNERABILITY STUDY

We evaluate our approach by performing a large-scale scan of patterns
used on the web. We explain how we gather the patterns in Section 6.1. We add these patterns to a testbed on which we compare our approach with other state-of-the-art scanners, Section 6.2. Design choices for the implementation of Black Ostrich are presented in Section 6.3. In Section 6.4 we manually compare our approach to validation methods used on popular websites. In
Appendix G, we include a performance comparison with ExpoSE.

6.1 Gathering Client Side Validation Regexes

To find real-world client-side validation, we use data from the Common Crawl data set [18]. From Common Crawl we extract all archives from (CC-MAIN–2021–31) and deduplicate incoming validation patterns per archive to avoid over-collecting. For each page, we extract all the HTML patterns along with their contexts such as other attributes of the element and the URL.

In addition to Common Crawl, we also crawl the top 100K domains from Tranco [46] to include patterns from popular websites. For each domain, we pick five random links and search all pages for forms with HTML5 patterns.

In total, we extract 9,805 unique patterns from 66,377 domains. Similar to previous work [35], we detect a high reuse of patterns.
across domains. We further remove any patterns that cause a syntax error in either Nodejs, Firefox, or PHP. First we, use Nodejs’ regex engine to filter out over 600 broken patterns. A majority of these are due to bad ranges, e.g. [05–09]. From there another 200 are removed for being invalid in Firefox, e.g. because of incorrect quantifiers (, 80) or trying to escape dash with a backslash. Additionally, patterns like [0–9]{1,10000000000} use too big quantifiers, causing both PHP and Ostrich to fail. While syntactically valid, we also remove unsatisfiable patterns such as /\d(5)/, where the slash before the caret makes it unsatisfiable (as an HTML pattern). In general, we believe that many of these problems stem from regular expressions being copied from other projects into HTML patterns without testing. This results in 8,820 valid patterns that we use for the testbed, and share publicly [27].

The most common patterns are for checking email addresses. This is interesting as type="email" already supports email validation. Other popular ones are the semantically equivalent patterns [0–9]* and \d*. Usually corresponding input elements for quantities. The complexity spans from simple and short to long and complex. The average length of the patterns is 39 characters but there are 453 longer than 100 characters and the longest pattern is 29,059. There are also 500 patterns using look-aheads and 4,044 patterns using anchors.

6.2 Testbed

To avoid damaging live websites we recreate the same input elements in a testbed. Using real-world patterns we create one page per unique pattern. Each page consists of a form with a single input element containing a pattern. We also include the most common name and type for each pattern as some scanners use this as a heuristic, as reflected in the code in Appendix D.

We ensure the same validation is applied on the server-side to stop scanners from simply ignoring the client-side check. We also check the input type validation for email and URL server-side. We show this server-side code in Appendix D. If the scanner sends a valid input the server will reflect this input, allowing for XSS. Finally, we run each scanner on the testbed and record both if it passes the pattern, and if it reports the XSS vulnerability.

6.3 Implementation

We implement our approach [27] by synergizing and improving the state-of-the-art web application scanner Black Widow [26] (Section 6.3.1) and solver Ostrich [16] (Section 6.3.2).

6.3.1 Scanner module. We make two major modifications, one to the data extraction and one to the witness selection. We update the navigation model in the crawler component to allow for modeling of the new pattern attributes and other validation attributes (Appendix A). During the crawling phase, we save all the patterns the scanner finds together with their respective input elements. To dynamically detect regex-based JavaScript validation our scanner injects JavaScript code before the page loads allowing us to proxy related functions including window.RegExp. test and String. match. Next, we input unique tokens on all input fields and trigger events such as onChange, onBlur, and onSubmit on the related form. Finally, we save any regex-input pair where validation is applied to our input.

6.3.2 Solver module. As SMT solver in Black Ostrich, we apply an extended version of the state-of-the-art solver Ostrich. The difference to the standard version of Ostrich is the handling of ECMA regular expressions (Section 4 and Section 5). This functionality was integrated by extending the Ostrich SMT-LIB interface [8], adding a function re. from_ecma2020 for converting a string in the ECMA regex format to a regular expression. The translation of 2AFASMT to NFA is implemented through the expansion in Definition 4. Our current implementation has good coverage of the ECMA regex features, but does not yet include the refinement loop from Section 4.5, and it only partially supports Unicode properties. The implementation includes several optimizations beyond what is described in Section 5, among others a direct translation (skipping 2AFASMT) from regexes without look-aheads to NFAs, and a refined version of the encoding in Definition 4 that only generates reachable NFA states to mitigate possible exponential growth.

6.4 Manual Inspection of Input Validation

Our focus is on HTML5 patterns and JavaScript regex functions. However, websites can use other methods for validation that we can not handle. To quantify this manually, we investigate the top sites from Tranco until we find 100 websites that use some input validation. We manually visit these websites and a maximum of 20 pages, searching for text inputs. Trying different values we test if validation is used and manually inspect the code both statically and dynamically to identify the validation method.

7 RESULTS

In this section, we present the results of our empirical study. Section 7.1 presents the results from our testbed. In Section 7.2 we analyze the results and present qualitative insights into the results. Finally, in Section 7.3 we report client-side validation methods used by popular websites.
7.1 Black-box Scanning

We divide the testbed results into pattern coverage and XSS vulnerability detection.

7.1.1 Coverage. In total, our scanner solves 8,782 patterns out of the total 8,820, resulting in a coverage of 99%. In comparison, the other scanners have an average coverage of 36%, ranging from jÄk solving 1,196 patterns (14%) to ZAP solving 4,638 patterns (53%).

The coverage results are presented in Figure 6, which shows that our method outperforms the combined efforts of previous approaches. A class of patterns only we find are patterns tightly bound in length, like \d\{16\} and \d\{6\}. Another case is patterns with complex use of multiple look-arounds like (?=.\d)(?=.*[a-z]) (?=.\d). We can also handle enumerations, e.g. (2018|2019|2020|2021|2022).

To understand the frequency of patterns and real-world effects, we report on the number of domains using these patterns. Figure 7 presents the number of domains, from both Common Crawl and Tranco, where the scanner can solve all patterns. Our method solves all patterns on 66,309 domains out of the total 66,377. We also subsample and improve coverage on 33,111 domains.

The heatmap, shown in Table 1, compares the solved patterns between the scanners. As is evident by the green Black Ostrich row, our approach has a strong matchup against the other scanners. In comparison, the Black Ostrich column shows that only a small number of patterns are solved by others that we miss. We discuss these in Section 7.2.

7.1.2 Vulnerabilities. Figure 8 shows the number of vulnerable patterns reported by the scanners. Our method outperforms the other scanners in terms of sending valid payloads. Compared to the average of the other scanners we improve detection of vulnerable patterns by 52%. The patterns passed by other scanners are usually simpler, like \d\{7\}, which allows any payload that is at least seven characters long. This explains the plateau at around 535 in Figure 8, which we discuss in Section 7.2. Our approach outperforms the others in cases with stricter formatting requirements. For example, email patterns that require the at-sign and period, like .+@.*, which requires a maximum of 20 characters, a normal XSS payload, e.g. <script>alert(1)</script>, is too long at 25 characters. In comparison, Enemy of the State uses the 19 characters long string "";|--""<Ocy1>=&{()}") and ZAP uses javascript: alert(1). We see these as false positives and therefore do not accept this in Black Ostrich. However, we still add support for detecting tag-injections, making it easy for developers to handle submission of unprintable values.

Vulnerabilities we miss. Both Enemy of the State and ZAP perform better than the other scanners we test. The reason for this is not that they use advanced string solving, but rather a different proof of XSS. This allows them to use shorter payloads. For example, for the pattern \d\{6\}, which allows a maximum of 20 characters, a normal XSS payload, e.g. <script>alert(1)</script>, is too long at 25 characters. In comparison, Enemy of the State uses the 19 characters long string "";|--""<Ocy1>=&{()} and ZAP uses javascript: alert(1). We see these as false positives and therefore do not accept this in Black Ostrich. However, we still add support for detecting tag-injections, making it easy for developers to handle submission of unprintable values.

Figure 7: Number of domains passed by scanners.

Figure 8: Number of vulnerable patterns found by scanners.

7.2 Analysis

In this section, we highlight patterns we miss and compare our scanner with the others.

Coverage we miss. As Table 1 shows, there are cases where other scanners solve patterns that we are not able to solve. In total, there are 15 cases where another scanner solves a pattern that we do not. These are complex patterns that have relatively easy solutions. A scanner-related problem is a pattern where the first solution is the DEL character (0x7F), which can not be typed into the text field by our scanner. To improve coverage in these cases we need to ensure the solutions are printable and improve the underlying scanner to handle submission of unprintable values.

Vulnerabilities we miss. Both Enemy of the State and ZAP perform better than the other scanners we test. The reason for this is not that they use advanced string solving, but rather a different proof of XSS. This allows them to use shorter payloads. For example, for the pattern \d\{6\}, which allows a maximum of 20 characters, a normal XSS payload, e.g. <script>alert(1)</script>, is too long at 25 characters. In comparison, Enemy of the State uses the 19 characters long string "";|--""<Ocy1>=&{()} and ZAP uses javascript: alert(1). We see these as false positives and therefore do not accept this in Black Ostrich. However, we still add support for detecting tag-injections, making it easy for developers to enable it.

jÄk’s coverage and vulnerability detection. jÄk’s performance is interesting as the coverage is significantly worse compared to the other scanners, yet the number of found vulnerabilities is on par with the others. This is because jÄk only sends attack payloads to the form. As such, jÄk’s coverage will match the vulnerabilities they find plus any pattern accepting empty strings. This differs from scanners that also try benign values for the input elements.

7.3 Results of Manual Inspection

Black Ostrich can handle a multitude of validation methods, including input types, patterns, and JavaScript regex functions. This again shows our method’s strong performance. The few we miss are analyzed in Section 7.2.
the top 212 websites, 100 use validation. The most popular methods rely on regex, test() was used in 56 cases and match() in seven. The pattern attribute was used on three websites. A common problem is exact length checks, e.g. input . length==10 for phone numbers. Some validations also split the input and check the parts individually, e.g. for email. In total, we support the methods used in 86%.

8 PATTERNS IN OPEN-SOURCE APPLICATIONS

To further explore the prevalence of patterns in the wild we perform a study on the use of patterns in GitHub projects. We download the 978 best-matching projects from GitHub’s “web-application” topic [32]. Next, we statically search for applications that use the pattern attribute and manually test that they are validated server-side. We acquire three usable projects with HTML patterns.

ALEX [47]. The ALEX project is a great example of a web application that validates the pattern both on the client-side and server-side. To create a new project in ALEX a URL is required, and the URL must match ^https?://.*? (ii) validation regexes would accept an XSS attack string? The login form is also dynamically generated making it impossible to specify the URL must match (iii) many validation regexes impose stricter constraints than MDN henceforth referred to as MDN [19].

\[
\text{[a-zA-Z0-9.!#$%&'\*+\-/\=\?^\_\`\{\}\]\}]+@\text{[a-zA-Z0-9.-]+(\.[a-zA-Z0-9.-]+)*\text{[a-zA-Z0-9.-]+}}
\]

The MDN validation strengthens the permissive requirements of RFC3696 [44], rejecting attack strings like <script>alert(1)</script>&example.com, at the same time disallowing some valid email addresses, like "<script"@example.com.

Users of the email input type can add additional validation using the pattern attribute. Alternatively, developers can forego the built-in email validation and only use a pattern attribute with their validation logic.

This section investigates how real-world regexes are used to validate email address inputs relate to the built-in validation of web browsers. We also investigate the security implications of sharing regexes for validation between the front-end and back-end of a web application without modification. This has implications for security, as the semantics of the pattern attribute are different from the ones of most regex engines.

In particular, we ask three research questions: (i) How many validation regexes would accept an XSS attack string? (ii) How many validation regexes impose stricter constraints than MDN, rejecting some string accepted by it? (iii) If the pattern validation regex is reused for validation in a back-end, how often would this let through an XSS attack string?
We investigate these questions on a collection of 825 unique email-validating regexes. We obtain this collection from the larger set of 9,805 unique patterns from Section 6.1. We narrow down our selection to patterns where the name or ID attributes of the input element contained the string “email”. This means that the data set contains both validation patterns used in addition to MDN and patterns used instead of it. In the experiments, we do not discriminate between these two cases, as it is hard to speculate how developers reflect the HTML5 email type on the server side. Finally, we remove 2 patterns that use anchors incorrectly, leaving us with a total of 825.

Table 3 summarizes the results of all three investigations. Recall that the acceptance of an email address containing a `<script>` tag by the client is neither necessarily in violation of the IETF standards nor is it a guaranteed vulnerability in the application. At the same time, accepting such an email server-side is a prerequisite of high-impact practical email vulnerabilities exploited in the wild [38, 58].

### 9.1 Vulnerable Patterns

To find potential sources of vulnerabilities, we instruct the solver to find a string `s` for each regex `ρ` out of the 825 collected ones such that `s` matches `ρ` contains a `<script>` tag. For an illustration, see the diagram on the left in Figure 9. This experiment finds 215 potentially vulnerable regexes (satisfiable). 38 regexes trigger syntax errors during parsing and had to be discarded from the study. All matching strings are validated against the RegExp class in Nodejs 15.14.0, and all but 2 (semantically invalid) are found to match. In total, we find 213 vulnerable regexes.

### 9.2 Strong Patterns vs MDN

We investigate if the patterns used are enforcing stronger constraints than MDN. To do this, we invoke the solver on each regex `ρ` to find a string not matching `ρ` but matching MDN, as illustrated on the right in Figure 9. This experiment yields a larger number of matching strings: 745, suggesting that these constraints are either typically used to narrow the set of allowed inputs or based on under-approximating expressions like `.+@+.+.`. The occurrence of negative look-aheads to eliminate some email hosts further suggests that the author intended to block these, a typical semantic validation not captured by the built-in syntactic email validation. One of the generated strings contained “me.com”, from a regex meant to block email addresses from common free email hosts. Other examples include exclamation points, ampersands, single quotes, pluses, or slashes, which are allowed before the @-sign by MDN but commonly disallowed by custom validation expressions.

### 9.3 Vulnerabilities When Sharing Code

While the HTML5 pattern must match the full input, most regex engines used in back-ends only need to match a substring. If the same regex is used for validation both at the front-end and at the back-end this would mean that the validation in the back-end is potentially weaker. Specifically, this would be the case for regexes without anchors matching the beginning and end of the string. We have found guides that incorrectly only check substrings [67].

To verify how common the use of such regexes is, we perform an experiment where we expand regexes not containing anchors (`^` or `$`) with catch-all expressions (`.+`), in an opposite fashion to the logic of Section 6.2. As the semantics of regexes are rather complicated, we expand them naively by simply replacing any expression beginning and ending with the expansion, if they did not contain either anchor, allowing post-solving validation to flag the edge cases where the regexes were more complicated. In other words, `.+@.+.` would become `.+@+.+.`. We could find an attack for 531 of the modified regexes, and could verify actual vulnerability for 502 of them; an increase of 289 from the 213 vulnerable ones we found in Section 9.1.

### 9.4 Summary

Email-validating HTML5 patterns are diverse. It is common for them to be both weak compared to the built-in validation, and to refine the built-in validation with additional constraints, as can be seen in Table 3. While the latter case does not affect the security of the application, the use of redundant validation expressions is also suggestive of code reuse. In which case, differences in semantics between the HTML pattern attribute and all common regex engines would make the validation at the back-end weaker than the front-end. This implies that security vulnerabilities will be present in many web applications if the strings are reused unsanitized.

Finally, these experiments illustrate that our encoding of the ECMA regex semantics is both versatile and performant enough to solve both substring matching and (non-) intersection for real-world regexes, many of them highly complex and all of them harvested from real websites. Only 38 (unable to parse) plus two (semantically invalid) out of the 825 regexes are untranslatable into our encoding.

### 10 RELATED WORK

SMT. String constraints solvers have flourished in recent years [3]. The two main paradigms for solving string constraints are SMT and constraint programming. Many SMT solvers have decision procedures for handling string constraints, for instance Z3 [21], Z3-smt/2/3/4 [68], S3/p/[#] [65], cvc5 [5], Norn [1], Noodler [14], Sloth [36], and Ostrich [16]. They rely on automata-based techniques or algebraic results for strings or reduce the problem to other well-known theories, such as integers or bit-vectors. To the
Web scanning is an actively explored topic with many open challenges, both for improving crawling and vulnerability detection. We compare our approach with other state-of-the-art scanners [23, 53, 57, 61]. There are also many other scanners and XSS detection methods [2, 4, 9, 24, 25, 29, 33, 40, 49, 55, 60] that made significant improvements in the field. Fonseca et al. [28] shows that many security patches in web applications update vulnerable regexes, further motivating the need for validation-aware web scanning. Barlas et al. [6] showed that the regex applied to input could itself be vulnerable to denial-of-service attacks.

While the goal of improving vulnerability detection has been common for previous approaches, the areas of scanning they improve vary. For example, jÄk [57], Enemy of the State [23], LigRE [24], and Black Widow [26] focus mainly on improving the crawling aspect of scanning, while using common payloads and fuzzing techniques. jÄk improved crawling by modeling JavaScript events in a novel way leading to deeper crawls and a higher detection rate of vulnerabilities. Enemy of the State achieved similar improvements by instead inferring the server-side state, thus being able to handle more complex workflows. Black Widow improves crawling by combining key features from previous methods, including navigation modeling, traversing, and inter-state dependency analysis. Although we build our scanner on top of Black Widow, neither of these approaches covers the orthogonal aspect of handling the validation patterns supplied by web applications.

In addition to improving crawling, the attack phase can also be improved to achieve better vulnerability detection rates. Both KameleonFuzz [25] and sqlmap [29] are examples of scanners that focus more on payload selection and fuzzing techniques to improve detection rate. KameleonFuzz dynamically mutates the XSS payloads based on the reflected value to iteratively update the payload until an attack is successful. While this has the potential of solving patterns, it is probabilistic and likely fails on very specific patterns. For example, one pattern only we could exploit was `*France`. Finding inputs with this specific string using mutations seems highly unlikely. FLAX [63] uses dynamic taint-tracking and mutation-based fuzzing to generate XSS payloads that can bypass client-side validation. However, their analysis requires a benign input that can already pass the validation. Finding this input is important for coverage and something our approach supports. While we focused on XSS in this study, solving patterns is important for finding other vulnerabilities such as SQL injections. sqlmap does not consider patterns when fuzzing, instead, they rely on a large table with payloads that use different escaping techniques. This too would fail on the vulnerable “France” example. To overcome this, Black Ostrich, also uses SMT to generate the payloads. This means that we can combine common attack payloads, like `<script>alert(1)</script>` with patterns like `.France` to generate successful attack inputs like `<script>alert(1)</script>France`.

11 CONCLUSIONS

We have presented Black Ostrich, a principled approach that leverages string-based constraint solving for deep crawling. We improve state-of-the-art string solving by extending the solver Ostrich with native support for ECMA regular expressions. To handle commonly occurring features like anchors and look-arounds in patterns on the web, we propose a new version of two-way alternating finite-state automata, named 2AFASM. Leveraging the observation that client-side validation, including HTML5 pattern attributes, custom JavaScript, and input types mirror the back-end validation of a web application, we illustrate how to integrate patterns like emails, zip codes, phone numbers, and maximum lengths into scanning and fuzzing. This increases our coverage of web applications, as we can pass form validation while still generating inputs containing XSS injections, tokens for taint tracking, or other side constraints required by the scanner. Our evaluation on 8,820 patterns extracted from popular websites demonstrates that Black Ostrich solves 99% of all patterns, yielding an improvement in coverage. This translates to us solving all patterns on 66,309 (99%) out of the 66,377 domains. We subsume and improve coverage on over 13,711 domains compared to the combined efforts of previous scanners. We also yield a 52% improvement in detecting vulnerable patterns compared to the average of the other scanners. In addition, we also manually inspect the validation methods (patterns, frameworks, custom JavaScript, etc.) used in the top 100 websites that use input validation and show that we can handle 86% of the validation methods. We analyze the use of patterns in open-source web applications from GitHub. We perform a case study on three of the projects and showcase improved coverage specifically thanks to our string solving capabilities. Finally, we have used our implementation of the ECMA Regular Expression standard of JavaScript to analyze a condensed set of harvested email validation patterns illustrating the correctness of our implementation, as we were able to find matching strings for the vast majority of the analyzed regular expressions. The study reveals remarkable inconsistencies in the current practices of email validation and shows that 213 (26%) out of the 825 found email validation patterns liberally admit XSS injection payloads.

Coordinated disclosure. Detecting if a vulnerable pattern leads to XSS is complex as the reflection can, for example, be in the admin panel. Therefore, we manually contact websites using vulnerable patterns and recommend improved patterns. So far, 26 have already updated their input validation.
ACKNOWLEDGMENTS

Thanks are due to Sebastian Lekies and Musard Balliu for the inspiring discussions and to Rustan Leino for his support. This work was partially supported by the Wallenberg AI. Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, the Wallenberg project UPDATE, the Swedish Foundation for Strategic Research (SSF) under the project WebSec (RIT17-0011), the Swedish Research Council (VR) under the grants 2018-04727 and 2021-06327, and an Amazon Research Award (AWS Automated Reasoning).

REFERENCES

### Table 4: HTML5 client-side form validation constraints

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General attributes</strong></td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>The type of the input, most relevant are: color, date, datetime-local, email, month, number, password, range, tel, text, time, url, week.</td>
</tr>
<tr>
<td>required</td>
<td>Field is non-empty</td>
</tr>
<tr>
<td><strong>Attributes for string inputs</strong></td>
<td></td>
</tr>
<tr>
<td>minlength</td>
<td>Minimum length of input string</td>
</tr>
<tr>
<td>maxlen</td>
<td>Maximum length of input string</td>
</tr>
<tr>
<td>pattern</td>
<td>A regular expression defining the expected textual input</td>
</tr>
<tr>
<td><strong>Attributes for numerical inputs</strong></td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>Minimum value of a numerical input, which can be of integer or fractional/decimal type</td>
</tr>
<tr>
<td>max</td>
<td>Maximum value of a numerical input</td>
</tr>
<tr>
<td>step</td>
<td>Step size of a numerical input: the input value has to be a multiple of the given (integer or fractional) number</td>
</tr>
</tbody>
</table>

For scanning and fuzzing, Black Ostrich has to generate input data page one by one, which is possible because HTML5 cannot express constraints relating multiple fields.

The type of a field specifies the general form of the type that is expected in different cases. For fields with numerical input types, we can directly compute suitable inputs and no constraint-solving is necessary. In particular, if the input type is number, data has to be an integer or fractional number in decimal notation, possibly subject to side-conditions min, max, and step. We can directly compute a suitable input as follows: (i) if min is present, this specified minimum value is chosen; (ii) otherwise, if max is present, then the biggest multiple of the step size that does not exceed max, namely \( n \cdot \text{step} \) where \( n \in \mathbb{Z} \), \( n \cdot \text{step} \leq \text{max} \); (iii) if neither min nor max is specified, the number 0 is taken.

### Table 5: Augmented RegEx Semantics

### Figure 10: Schematic representation of automaton construction recursive steps for the atomic cases: \( \epsilon, 0 \) and \( \sigma \in \Sigma \).

### Figure 11: Schematic representation of automaton construction recursive steps for: \( p_1, p_2, p_1 \cdot p_2 \) and \( p^n \).


### A CLIENT-SIDE FORM VALIDATION

For scanning and fuzzing, Black Ostrich has to generate input data satisfying the web application’s validation constraints. Table 4 contains a list of validation attributes in HTML5; some further form types, such as id, alt, title, etc., are less relevant from a security point of view and they are not considered here. Our general approach is to generate data for the individual input fields of a page one by one, which is possible because HTML5 cannot express constraints relating multiple fields.

The type of a field specifies the general form of the type that is expected and determines how exactly data generation works. For fields with numerical input types, we can directly compute suitable inputs and no constraint-solving is necessary. In particular, if the input type is number, data has to be an integer or fractional number in decimal notation, possibly subject to side-conditions min, max, and step. We can directly compute a suitable input number as follows: (i) if min is present, this specified minimum value is chosen; (ii) otherwise, if max is present, then the biggest multiple of the step size that does not exceed max, namely \( n \cdot \text{step} \) where \( n \in \mathbb{Z} \), \( n \cdot \text{step} \leq \text{max} \); (iii) if neither min nor max is specified, the number 0 is taken. The types color, range, date, datatime-local, month, time, and week can be handled similarly.

For fields \( f \) of other types (e.g., text), an SMT formula \( \phi_f[w] \) is constructed that represents all attributes that input \( w \) for \( f \) has.
to satisfy. This formula is sent to the SMT component, which will then check whether input data exists that fits the input field. The formula has the shape $\phi_f = \phi_{type} \land \phi_{len} \land \phi_{pat}$, and consists of a pre-defined regular expression constraint $\phi_{type} = (w \in \mathcal{L}_{type})$ that is specific to the field type, a numeric constraint $\phi_{len} = (|w| \in [l, u])$ that captures the attributes min\text{length} and max\text{length} (if present), and the HTML5 \texttt{pattern} constraint $\phi_{pat} = (w \in \mathcal{L}_{pat})$ discussed in the next section. The type-specific language $\mathcal{L}_{type}$ defines all inputs that are well-formed according to the HTML standard [66]; for instance, for an input field of type email, we can choose the language given in Section 9.

B DETAILS OF Section 3

We define the semantics of augmented regular expressions recursively, in the style of operational semantics:

Definition 5 (Augmented regex semantics). Let $w = w_0w_1 \ldots w_n$ be a string in $\Sigma^*$ of length $|w| = n+1$, and let $w(i, j)$ be the segment of the string starting at $i$ and ending at $j$ (excluded). A capture group mapping is a partial function $G: \mathbb{N} \rightarrow \Sigma^*$ mapping numbers to strings. We write $w, (i, G) \xrightarrow{\rho} (j, G')$ to state that regex $\rho$ can parse word $w$ starting at index $i$ and capture group mapping $G$ and finishing at index $j$ and mapping $G'$. The rules defining acceptance of a regex are given in Table 5. For a regex $\rho$ without back-references, we write $w, (i, j) \models \rho$ if $w, (i, G) \xrightarrow{\rho} (j, G')$ for any $G, G'$.

We now describe in detail the procedure for generating the 2AFA\text{SMT}.

Given $\rho \in \mathcal{R}$, we first transform the input in "normal form" (in linear time in the size of $\rho$) by calling TRFM($\rho$): this intuitively replaces anchors $\lambda$ and $\&$ with $\Sigma$ and $\Sigma$ respectively and it reverses subexpressions inside look-behinds. The transformation also replaces back-references $\\backref$ with the regex $\rho_{\text{cg}}$ of the corresponding capture group ($\rho_{\text{cg}}$).

1. Input: A regex $\rho \in \mathcal{R}$
2. Output: a regex $\rho' \in \mathcal{R}$
3. procedure TRFM($\rho$)
4. case $\rho$ of
5. $\lambda | \varepsilon | \emptyset$: return $\rho$
6. $\rho^*$: return TRFM($\rho$)$^*$
7. $\rho_1 \cdot \rho_2$: return TRFM($\rho_1$) . TRFM($\rho_2$)
8. $\rho_1 + \rho_2$: return TRFM($\rho_1$) + TRFM($\rho_2$)
9. $(\rho > \rho)$: return ($\rho >$ TRFM($\rho$))
10. $(\rho < \rho)$: return ($\rho <$ TRFM($\rho$))
11. $(\rho = \rho)$: return ($\rho = \rho$)
12. $\\backref$: return ($\backref$)
13. $\emptyset$: return $\emptyset$
14. $\\backref$: return $\\backref$
15. $(\rho)n$: return TRFM($\rho^n$) if positive context, otherwise $\emptyset$
16. end case
17. end procedure

Then, the automaton is built from the bottom up by calling AUT on TRFM($\rho$), which works recursively on the structure of the formula. The subroutine ATOMIC\text{AUT} returns an automaton as in Figure 10, while STAR\text{AUT}, ALT\text{AUT}, CONCAT\text{AUT} or LOOK\text{AUT} return automata as in the Figure 11 and Figure 4 where $s_f$ and $s_f'$ are the initial and final state of the automaton returned by the recursive call on $\rho_f$. The routine \text{NEG} takes an automaton and returns its complement in polynomial time, e.g., by following the technique in [30].

1. Input: A regex $\rho \in \mathcal{R}$
2. Output: 2AFA\text{SMT} $A_\rho$
3. procedure AUT($\emptyset$, $\rho$)
4. case $\rho$ of
5. $\lambda | \varepsilon | \emptyset$: return ATOMIC\text{AUT}($\emptyset$, $\rho$)$\times$ Transitions are $\delta < \dot\delta$ if $\emptyset$ is $\dot\delta$ otherwise
6. $\rho^*$: return STAR\text{AUT}($\emptyset$, $\rho$))
7. $\rho_1 \cdot \rho_2$: return CONCAT\text{AUT}($\emptyset$, $\rho_1$, $\rho_2$)
8. $\rho_1 + \rho_2$: return ALT\text{AUT}($\emptyset$, $\rho_1$, $\rho_2$)
9. $(\rho > \rho)$: return LOOK\text{AUT}($\emptyset$, $\rho$)
10. $(\bar{\rho})$: return LOOK\text{AUT}($\emptyset$, $\rho$)
11. $(\rho < \rho)$: return LOOK\text{AUT}($\emptyset$, $\rho$)
12. $(\rho = \rho)$: return LOOK\text{AUT}($\emptyset$, $\rho$)
13. end case
14. end procedure

Proof of Theorem 1. We focus on the case of a regex $\rho$ without back-references; the general case follows due to monotonicity. The proof is modular and the main part requires to prove the following statement: for each $w, i, j$ and $p, w, (i, j) \models p$ if AUT($\rho > \rho$) has an accepting run on $w$ starting from position $i$ in $w$ and AUT($\rho <$ $\rho$), AUT($\rho =$ $\rho$)) has an accepting run on $w$ starting from position $j$. From the semantics of regexes, $w, (i, j) \models p$ if $i = j$ and there exists $0 \leq k \leq i$ such that $w, (k, i) \models p$. The automaton LOOK\text{AUT}($\rho < \rho$)) is as in Figure 4 where $A_\rho$ is the (backward) automaton for $\rho$. Since the first transition is a $\epsilon$, the run expands into two pairs $((s_1^1, i), (s_f, i))$. The pair $(s_f, i)$ accepts iff $i = j$, namely there is nothing left to read to the right: indeed, by definition of 2AFA\text{SMT}, any symbol $\sigma$ read from $s_f, i$ leads to a non-accepting sink state (which we do not explicitly show in the figures for the sake of readability). Let us now consider the upper path: by inductive hypothesis, $A_{\rho_2}(\sigma)$ (without the $\Sigma$-loop on $s_f$) accepts iff $w, (k, i) \models p$ for some $0 \leq k \leq i$. This means that the run will eventually reach $s_f, i$ before, or at, index 0. Since $s_f, i$ is now a sink accepting state thanks to the $\Sigma$-loop, the run from $(s_1^1, i), (s_f, i)$ accepts iff there exists $0 \leq k \leq i$ such that $w, (k, i) \models p$. An analogous argument holds for AUT($\rho <$ $\rho$).

From 2AFA\text{SMT} to S-2AFA\text{SMT}. We transform the former into the latter by performing the following steps. First, we multiply the states: for each $s$ of the 2AFA\text{SMT}, which in general has $>$ and $<$ both incoming and outgoing transitions, we have four states of the S-2AFA\text{SMT} for each pairwise combinations of those ($>$ and $<$ in S-2AFA\text{SMT} are generated from the initial and sink states of the 2AFA\text{SMT}). Such states are connected with $\epsilon$ transitions and therefore the original semantics is preserved. In the second step, we remove the $\epsilon$ transitions by exploiting bidirectionality. Indeed, we can simulate a $\epsilon$ transition from, e.g., state $s$ to $s'$ by: adding a special state $s''$, having $>$ transitions from $s$ to $s''$ for every symbol $\sigma \in \Sigma$ and then having $<$ transitions from $s''$ to $s'$ again for every symbol. This easily generalizes to set of states and $\epsilon$ transitions as well, but special care is needed at the end of the word where $>$ are not allowed. To handle those cases, we actually introduce special markers at the beginning and end of the words, which we
do not explicitly include here for the sake of readability. Lastly, the S-2AFA\textsubscript{MT} only accepts at the end of the word, as it only has \(P^>\) state, and this can be obtained by moving to an \(P^<\) accepting state from former accepting states \(P^\leq\) by means of \(\delta^<_3\) transitions.

C DETAILS OF Section 5

Proof of Theorem 2. "e=I" Fix a word \(w = w_0w_1\ldots w_n\), and let \(\pi = ((Q_0, 0), (Q_1, i), (Q_2, i_2), \ldots, (Q_n, i+n))\) be the accepting run of the simulating NFA. Let \(P = \{(s, i) | (Q_i, i) \in \pi \text{ and } s \in Q_i\}\) be the set of S-2AFA\textsubscript{MT} state/index pairs occurring in the run. Consider then the graph \(G = (P, R)\) with the relation:

\[
R = \{(s, i, t, i+1) | t \in \delta^>_0(s, w_i) \cup \delta^<_0(s, w_i) \} \cup \{(s, i, t, i-1) | t \in \delta^>_0(s, w_i-1) \cup \delta^<_0(s, w_i-1) \}
\]

(1)

\(G\) is acyclic, because the considered S-2AFA\textsubscript{MT} is non-cycling. We prove that, for every \((s, i) \in P\), there is an accepting S-2AFA\textsubscript{MT} run \(\pi\) on \(w\) with the initial configuration \(p_0 = \{(s, i)\}\); this is proven by well-founded induction on the length of the longest path in \(G\) starting from \((s, i)\).

If \((s, i)\) has no outgoing edges, then \(i = n + 1\) and \(s\) must be accepting, and \(\pi = \{(s, i)\}\) is an accepting run. Suppose then \((s, i)\) has outgoing edges, and the first case of (1) applies and \(s \in S^<_0 \cup S^<_1 \cup (s_0)\) (the second case is similar). By Definition 4 condition 5, there is an \(s' \in S^<_0(s, w_i)\) such that \((s', i+1) \in P, or (s', i+1) \in P \) for all \(s' \in S^<_0(s, w_i)\). In both cases, by induction for each such \(s'\) an accepting run with initial configuration \(\{s', i+1\}\) exists, which can be combined and extended to obtain an accepting run with initial state \((s, i)\).


E ExpoSE JAVASCRIPT TEMPLATE

We use the following code to create JavaScript files for each pattern. The placeholder \([\text{[PATTERN]}]\) is replaced with the actual pattern. Running ExpoSE will tell us if \texttt{Reachable_regex_solved} is reachable and thus if the pattern is solvable.

```javascript
var $s = require('$$');
var a = $$\text{.symbol}('A', 'C');
var re = new RegExp('(?<\?:) + [\text{[PATTERN]}] + $' );
if (a.match(re)) {
  throw 'Reachable_regex_solved';
}
```

F ALGORITHM FOR VALIDATION-AWARE SCANNING

This algorithm shows the crawling phase of the scanner. For the attack phase line 6 is repeated for each payload the scanner uses. We assume a \texttt{node} to be a general object that can be scanned, like a URL or JavaScript event. For each input element, we check for a validation constraint, like a pattern attribute, and try to solve it. For payloads, we can add the payload as the second argument to \texttt{solveConstraint}, instructing it to solve the pattern and include the payload string.

```
Data: Target url
nodes = scanPage(url);
while node = nodes.pop() do
  if node.type == FORM then
    for element in node.elements do
      if validationConstraint(element) then
        element.value = solveConstraint(element, "");
    end
  end
  end

Algorithm 1: Crawling algorithm with pattern solver.
```

G PERFORMANCE COMPARISON WITH ExpoSE

G.1 Comparison of Ostrich and ExpoSE

Loring et al. [48] present ExpoSE, a symbolic execution tool for JavaScript. While ExpoSE is not a string solver, it contains one that could potentially be combined with a web scanner. To compare, we run both ExpoSE and Ostrich on all valid patterns and record their coverage and execution time. Since ExpoSE uses symboli
execution we need to create a small JavaScript file with the pattern. The exact JavaScript file is in Appendix E. To avoid getting stuck on any particular pattern and simulate a realistic timeout for a web scanner, we use a timeout of ten seconds. We did test allowing a higher timeout for ExpoSE, but it only had a marginal effect, allowing for 43 more patterns being solved with a one hour timeout.

G.2 Results of Black Ostrich vs. ExpoSE

From the 9,805 patterns, we remove broken, invalid, and unsatisfiable patterns to arrive at a total of 8,820. Our experiment shows that Black Ostrich can solve 8,795 while ExpoSE solves 7,188, an improvement of 22%. This is slightly better than the testbed results as it does not impose the extra constraint of printable characters. Comparing the time performance, Black Ostrich takes on average 1.67 seconds and ExpoSE takes 4.11. Also important to note is that ExpoSE time-outs on 158 patterns while Black Ostrich only time-outs on 2 patterns.

One example of where ExpoSE fails, but Black Ostrich does not, is a large group of 358 patterns using look-arounds; one such pattern is the recommended regex for passwords shown in Example 1. In general, the patterns ExpoSE fails on are also longer, with an average length of 62 characters compared to 26 for solved patterns. Similarly, for patterns that time out, the average length is 156 vs. 30 for patterns solved in time.