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 web crawling and validation. The key idea is to equip web crawlers 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

ACM Reference Format:


1 INTRODUCTION

As the modern digitalized society increasingly relies on web applications, securing them remains an important challenge. Web security scanners like Arachni and ZAP play an important role, focusing on crawling and scanning for vulnerabilities. Recent efforts by the research community have focused on moving away from traditional static crawling techniques based on link discovery and URL traversal. As JavaScript enables increasingly dynamic web pages, new approaches incorporate dynamic behaviors as in jÄk [57], and asynchronous HTTP requests as in CrawlJAX [11, 50]. Other approaches address the complexity of the server-side application by reverse engineering, as in LigRE [24] and KameleonFuzz [25], or inferring the state of the server-side as in Enemy of the State [23]. Black Widow [26] demonstrates how to fruitfully combine navigation modeling, traversing, and tracking inter-state dependencies for black-box web application scanning.

1.1 Web Scanning Challenges

While this progress is encouraging, 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 for data such as email, phone number, or zip code. Consider, for instance, the pattern .*@.*[a-z]{2,3} for emails, which allows any string followed by an `@` sign followed by any string then a period and two or three lowercase letters. Although seemingly sound, the pattern allows for malicious inputs such as <script>alert(1)</script>@mail.com, which are exactly the types of email XSS payloads commonly exploited in the wild [38, 58]. Regular expressions of this kind are commonly associated with input fields of web applications, and a crawler will only be able to proceed to the next page by submitting a string matching the pattern. Resorting to brute force would be intractable, and using a library of prepared payloads (as many scanners do) is infeasible when websites use specialized validation. As an example, we did not find any scanner with payloads matching the real-world pattern .*France. A more refined approach is thus needed.
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 ECMA Script 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 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 letter-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 [27], 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 Widow. The obvious ethical reasons prevent us from directly run-
We demonstrate that Black Ostrich boosts both code coverage and we also analyze how many domains we can improve coverage on.
validations used on popular websites, we combine this with a crawl on three applications that use both client-side and server-side vali-
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. As many websites share the same 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, (1) 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 to 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., @@@. It returns the solution to the witness controller, step 5, which returns it to the crawler, step 9, where it is submitted to the application, step 10.

Attack Phase. The witness controller calls the payload generator to get a payload in step 7. The payload generator chooses a payload, commonly from a pre-defined list, e.g., <script>alert(1)</script>. In step 3, the witness controller sends both the pattern and the payload to the SMT. Both are encoded and sent to the solver in step 6. The solver generates a valid solution to the pattern that also contains the payload and sends it back in 9. The solution <script>alert(1)</script>@@@ 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 5, which submits it to the web page in step 10.

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 jÄk [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 jÄk, 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 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 [27] 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 `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 `.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 `<script>xxs(123)</script>France` While our focus is on XSS, the same method can be used to generate valid SQLi payloads, e.g. `DROP TABLE;`.

### 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 [27]; 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 `^` and `$` 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]:

```
^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).*$
```

The assertions `(?=.*\d)` 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 `(?![a-z])` 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:

```
^\(?-\.(\d{1,64})\)?(\[(a-zA-Z0-9-]{1,62}.)+(a-zA-Z0-9-]{1,62}.)?[a-zA-Z]{1,63}@\.(a-zA-Z0-9-]{1,62}.)+(a-zA-Z0-9-]{1,62}.)+$
```

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 (Kleenestar) and look-arounds, is not correctly modelled, leading to an incorrect encoding of regular expressions like `a*ax`. This regex is equivalent to `a*xa`, 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*xa)` 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 Ostrich. It is well known that textbook regexes and finite-state automata are equivalent, in the sense that they can express the same set of languages [37]. Reasoning on regexes, e.g., checking emptiness or inclusion, can therefore be performed using automata techniques.

4.1 Overview

Our approach translates ECMAScript regexes in several steps to non-deterministic finite-state automata (NFA, used both in singular and plural), as illustrated in Figure 2. Regexes are first parsed and simplified, resulting in a term representation of the regex. This term is then encoded as a two-way alternating automata (2AFA) [45], as described in Section 4.4; for this, we introduce a new variant of 2AFA, named 2AFASM, 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]. 2AFASM 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 2AFASM 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 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 2AFASM in the context of a finite alphabet \( \Sigma = \{\sigma_1, \ldots, \sigma_n\} \); the translation to 2AFASM works in exactly the same way in the symbolic setting, representing character ranges using intervals. The set of textbook regexes \( \mathcal{R} \) is then inductively defined as follows [37]:

\[
\mathcal{R} := \emptyset \mid \varepsilon \mid \sigma \mid r^* \mid \overline{r} \mid r_1 \cdot r_2 \mid r_1 + r_2
\]

where \( \sigma \in \Sigma \), \( \overline{\cdot} \) is the complement of \( \cdot \), \( ^* \) is the Kleene star operator, and \( \cdot \) and \( + \) are the usual concatenation and alternation operators, respectively. We also define syntactic shortcuts, namely \( r_1 \cap r_2 := \overline{r_1 + r_2} \) and, with slight notational abuse, \( \mathcal{A} := \sigma_1 + \cdots + \sigma_n \).

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:

\[
\rho := \emptyset \mid \varepsilon \mid \sigma \mid \rho^* \mid \rho_1 \cdot \rho_1 \mid \rho_1 + \rho_1 \mid (\rho') \mid (\leq \rho) \mid (\geq \rho) \mid (\geq \rho) \mid ^{\dagger} \mid ^{\star} \mid (\rho)n \mid \n
\]

where \((\rho')\) and \((\geq \rho)\) 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. \((\leq \rho)\) and \((\leq \rho)\) 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 \(^{\star}\) 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 \( n \). It is assumed that at most one capture group \((\rho)n\) exists for each index \( n \). [27] 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 \( \varepsilon \)-transition, 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 2AFASM 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 2AFASM is on the way transitions are specified. The former
4.4 Translation of Augmented Regexes

Our procedure recursively constructs a 2AFA_{SMT} \( A_p \) 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 \( ^\ast \) is equivalent to \((\varepsilon \Sigma)\) and \( $\) is equivalent to \((\varepsilon \Sigma)\).

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 \( \delta^< \) transitions and the forward \( \delta^\geq \) ones.

**Definition 1 (2AFA_{SMT}).** A two-way alternating automaton is a tuple \((\Sigma, S, s_0, F^<, F^\geq, \delta^<, \delta^\geq, \delta^\psi, \delta^\varepsilon, \epsilon_\exists, \epsilon_\forall)\) where:

- \( \Sigma \) is an alphabet of symbols;
- \( S \) is a finite set of states;
- \( s_0 \in S \) is an initial state;
- \( F^<, F^\geq \subseteq S \) are disjoint sets of final states;
- \( \delta^< : S \times \Sigma \rightarrow \phi(S) \) are partial existential (\( \exists \)) and universal (\( \forall \)) transition functions, respectively;
- \( \epsilon_\exists, \epsilon_\forall : S \rightarrow \phi(S) \) are partial \( \epsilon \)-existential (\( \exists \)) and \( \epsilon \)-universal (\( \forall \)) transition functions, respectively;
- \( \phi(S) \) is the powerset of \( S \). We require that for every state \( s \in S \) and \( \sigma \in \Sigma \) one of the \( \delta^< \) or \( \delta^\geq \) transitions is defined.

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

**Definition 2 (2AFA_{SMT} run).** Let \( w = w_0 w_1 \ldots w_n \) be a word in \( \Sigma^* \) of length \( \ell(w) = n + 1 \), and \( A \) be 2AFA_{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:

\[
\pi_{j+1} := \pi_j \setminus \{(s, i)\} \cup T \text{ where } T \text{ is one of the following:}
\]

- \( \{(s', i+1)\} \) if \( s' \in \delta^<_{\exists}(s, w_i) \) and \( i < \ell(w) \);
- \( \{(s', i-1)\} \) if \( s' \in \delta^\geq_{\forall}(s, w_{i-1}) \) and \( i > 0 \);
- \( \{(s', i+1) \mid s' \in S'\} \) if \( s' \in \delta^<_{\exists}(s, w_i) = S' \) and \( i < \ell(w) \);
- \( \{(s', i) \mid s' \in S'\} \) if \( s' \in \delta^\geq_{\forall}(s, w_{i-1}) = S' \) and \( i > 0 \);
- \( \{(s', i)\} \) if \( s' \in \epsilon_\exists(s) \);
- \( \{(s', i)\} \) if \( \epsilon_\forall(s') = S' \).

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 \( s \in F^< \) and \( i = 0 \) or \( s \in F^\geq \) and \( i = \ell(w) \).

### Example 2
Consider the regex \((a+b)^\ast(\varepsilon \cdot (<a \cdot b))\). 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 \( w = abbbab \), a successful execution sees the sub-automaton \( A_{(a+b)^\ast} \) matching the whole word and ending in state \( s_8 \). From there, the execution forks: one path directly accepts in \( s_{12} \in F^\geq \), while the other goes through sub-automaton \( A_{a \cdot b} \), which starts scanning backward. It first reads \( b \) and then \( a \), which indeed matches \((<a \cdot b))\) thus ending up in the final state \( s_9 \in F^\geq \). Since both paths are in a final state, word \( w \) is accepted.

Intuitively, the automaton translation works as follows (see [27] for more details and figures). The automata for atomic cases of \( \varepsilon, \sigma \) accept after seeing \( \varepsilon \) 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 \( \varepsilon \)-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 \( \varepsilon \)-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 \(s_f, t \in F^p\) are final, and the outgoing transitions from \(s_t\) are \(\vee\)-transitions. This initial \(\vee\)-transition forks the execution in two paths that should both accept. The final state \(s_{f,\bar{\Sigma}}\) 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 \((\rho)\) are translated like \(\rho\), while there are two cases for back-references \(\\backslash n\): a back-reference occurring under an even number of negations \(\exists\) or \(\forall\) is over-approximated by the regex \(\rho\) of the group \((\rho)\) 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 \(w \in \Sigma^*\), if \(w \in L(\rho)\), then automaton \(A_\rho\) accepts \(w\) as well. If \(\rho\) does not contain back-references, then \(w \in \Sigma^*\) if and only if \(w \in L(\rho)\).

A proof is given in [27]. 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 \(\rho = (a + b)\%).\) describes the language \(L(\rho) = \{aa, bb\}\). 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 \(w \in \Sigma^*\). 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 solution, the symbolic 2AFA\_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 \(\Sigma^* \backslash \{w\}\). In the setting of 2AFA\_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\_SMT BY NFA

We now define a translation from 2AFA\_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 \((\Sigma, Q, q_0, Q_f, \delta, \epsilon)\) in which \(q_0 \in Q\) is the initial state, \(Q_f \subseteq Q\) is the set of final states, and \(\delta : \Sigma \times Q \rightarrow \wp(Q)\) and \(\epsilon : Q \rightarrow \wp(Q)\) are transition functions. The semantics of NFA is defined similarly to that of 2AFA\_SMT. A run \(\pi = \pi_0 \pi_1 \cdots \pi_n\) of an NFA over a word \(w\) is finite sequence of configurations from \(Q \times \Sigma\) defined inductively: \(\pi_0 = (q_0, 0)\), and for any \(\pi_j = (q, i)\) we have \(\pi_{j+1} = (q', i')\) with either (i) \(q' \in \epsilon(q)\) and \(i' = i\); or (ii) \(i < \ell(w), q' \in \delta(q, w(i))\) 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 2AFA 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\_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 set \(S^+\) 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\_SMT).** A simplified 2AFA\_SMT (or S-2AFA\_SMT) is a tuple \((\Sigma, S^+, S^-\subseteq S^+; S^0, S^\max; S^0, S^\max; \Sigma^+; S^0, S^\max; \Sigma^+; S^0, S^\max; \Sigma^+; S^0, S^\max; \Sigma^+; S^0, S^\max)\) in which:

- transition functions \(\delta, \delta^+, \delta^-, \delta^0\) are as defined in Definition 1;
- the sets of states \(S^+, S^-, S^0, S^\max\) are pairwise disjoint, and we denote with \(S\) their union;
- \(s_0\) is the initial state, which does not have incoming transitions and only has outgoing left-to-right transitions: for each \(s' \in S, s \in S_\Sigma, s' \in \{\exists, \forall\}, \text{ and } s \in \{<, >\}\) we have that: \(s_0 \notin \delta^s(s', \sigma)\), and \(\delta^0(s, \sigma)\) and \(\delta^s(s, \sigma)\) are undefined;
- \(S^+\) is the set of sink states, which only have incoming left-to-right transitions: for each \(s \in S^+, s' \in S, s \in S_\Sigma\) and \(\sigma \in \{\exists, \forall\}\) we have that: \(s \notin \delta^s(s', \sigma)\), and \(\delta^s(s, \sigma)\) and \(\delta^0(s, \sigma)\) are undefined;
- \(F^+ = S^0\) are final states;
for each state $s \in S_\prec$, $s' \in S$, $\sigma \in \Sigma$ and $* \in \{\exists, \forall\}$ we have that: $s \notin \delta_\prec^*(s', \sigma)$ and $\delta_\prec^*(s, \sigma)$ is undefined. Analogous definitions hold for $S_\succ$, $S_\succ\prec$, $S_\prec\succ$, $S_\prec\succ\succ$.

We notice that any 2AFA$_{SMT}$ can be transformed into a S-2AFA$_{SMT}$, and refer to the extended version of the paper [27] for the procedure.

### 5.2 Simulation of S-2AFA$_{SMT}$

Next, we show how to build an NFA that is equivalent to a S-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 $A$ cannot 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 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 $s_0$, the automaton reads three characters moving right ($s_2, s_3 \in S_\prec$) and lands in state $s_4 \in S_\succ$; then it moves backward on $s' \in S_\prec\succ$ and ends up in $s_6 \in S_\succ\prec\succ$, and finally moves forward to the end of the word accepting in $s_8 \in S_\succ\succ\succ$. 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. Gray boxes represent the macro-states of the NFA.

**Definition 4** (Simulating NFA). Let $(\Sigma, S_\prec, S_\succ, S_\succ\prec, S_\prec\succ, S_\prec\succ\succ, s_0, F^\prec, \delta_\prec, \delta_\succ, \delta_\succ\prec, \delta_\prec\succ, \delta_\prec\succ\succ)$ be a S-2AFA$_{SMT}$. Next, the equivalent NFA is $(\Sigma, \phi(S), \{s_0\}, \{F\} \subseteq F^\prec, \delta, \epsilon)$, with $\delta, \epsilon$ defined as follows. For every $\forall Q' \in \phi(S)$, we have $Q' \in \epsilon(Q)$ if and only if either:

1. there is $s \in S_\prec\succ\succ$ such that $Q' = Q \cup \{s\}$, or
2. there is $s \in S_\succ\prec\succ$ such that $Q' = Q \cup \{s\}$.

For each $Q, Q' \in \phi(S)$ 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 $Q$: $Q \cap (F^\prec \cup S_\prec\succ\succ) = \emptyset$;
4. No sources in $Q'$: $Q' \cap (\{s_0\} \cup S_\prec\succ\prec\succ) = \emptyset$;
5. Right-successors: $\forall s \in Q \cap (S_\prec\succ \cup S_\prec\succ\succ \cup \{s_0\})$:

\[
Q' \cap \delta_\succ\prec\succ(s, \sigma) \neq \emptyset \text{ or } \delta_\prec\succ\succ(s, \sigma) \subseteq Q';
\]
6. Left-successors: $\forall s' \in Q' \cap (S_\prec\succ\succ \cup S_\prec\succ\succ\prec)$:

\[
\delta_\prec\succ\succ(s', \sigma) \cap Q \neq \emptyset \text{ or } \delta_\succ\prec\succ(s', \sigma) \subseteq Q';
\]

7. Right-predecessor: $\exists s \in Q \cap (S_\prec\succ\prec \cup S_\prec\succ\prec\succ) \exists s' \in Q'$:

\[
s \notin \delta_\prec\succ\prec(s', \sigma) \text{ or } s \notin \delta_\prec\succ\prec\succ(s', \sigma) \subseteq Q;
\]
8. Left-predecessor: $\exists s' \in Q' \cap (S_\prec\succ\succ \cup S_\prec\succ\succ\prec \cup F^\prec) \exists s \in Q$:

\[
s' \notin \delta_\prec\succ\succ\prec(s, \sigma) \text{ or } s' \notin \delta_\prec\succ\succ\prec\prec(s, \sigma) \subseteq Q'.
\]

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 $S_\succ\succ\succ$ such as $s_6$ can "appear" in a macro-state, $Q_2$ in this case, thanks to $s$ transitions as dictated by condition 1 in Definition 4 (analogously, $S_\prec\succ\succ$ 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 $S_\succ\prec\succ$ states, while for $S_\prec\succ\succ$ and $S_\prec\succ\prec\succ$ 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$_{SMT}$ iff $w$ is accepted by its simulating NFA.

The proof is provided in [27].

### 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 [27], 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 syntactic 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 (, 0) 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 */^<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-arounds 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 [27].

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 [27]. 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 [27]. 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, onBlurr, and onSubmitt on the related form. Finally, we save any regex-input pair where validation is applied to our input.

Once the scanner is ready to submit a value to a form we retrieve the corresponding pattern from the navigation model and send it to the solver, which generates a matching value for the pattern. When the scanner is in attack mode then the payload is also sent to the solver as an additional constraint.

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 2AFA 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 2AFA21) from regexes without look-arounds 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.

Figure 6: Number of patterns passed by scanners.
7.1 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 \(1\d(16)\) and \((.)(6,6)\). Another case is patterns with complex use of multiple look-arounds like \((?=\.*\d)(?=\.*[a-z])(?=\.*[A-Z]).{6,}\). 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 subsume and improve coverage on 13,711 domains.

The heatmap, shown in Table 2, 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 \((7,7)\), 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 \(.*@.\{\}.\). Or requirements of specific strings, like "France" in the pattern \.*France. The Black Ostrich row in the heatmap in Table 2, once again shows our method’s strong performance. The few we miss are analyzed in Section 7.2.

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 test 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 \((0,20)\), 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 `'';|--"<Gcyt>"&{()}` 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. Of 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
To further explore the prevalence of patterns in the wild we perform application that validates the pattern both on the client-side and server-side. Here we do not provide is registering a user. The registration form uses patterns to validate phone numbers, among other things. We acquire three usable projects with HTML patterns.

<table>
<thead>
<tr>
<th></th>
<th>Arachni</th>
<th>Black Ostrich</th>
<th>Black Widow</th>
<th>Enemy</th>
<th>JÄk</th>
<th>ZAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arachni</td>
<td>6,387</td>
<td>4,891</td>
<td>4,726</td>
<td>7,584</td>
<td>4,355</td>
<td></td>
</tr>
<tr>
<td>Black Ostrich</td>
<td>1,462</td>
<td>899</td>
<td>79</td>
<td>37</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>Black Widow</td>
<td>1,888</td>
<td>385</td>
<td>101</td>
<td>86</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>Enemy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JÄk</td>
<td>6,2</td>
<td>2</td>
<td>3</td>
<td></td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>ZAP</td>
<td>1,888</td>
<td>104</td>
<td>125</td>
<td>122</td>
<td>228</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Unique coverage found between scanners

<table>
<thead>
<tr>
<th></th>
<th>Arachni</th>
<th>Black Ostrich</th>
<th>Black Widow</th>
<th>Enemy</th>
<th>JÄk</th>
<th>ZAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arachni</td>
<td>11</td>
<td>101</td>
<td>101</td>
<td>86</td>
<td>113</td>
<td>59</td>
</tr>
<tr>
<td>Black Ostrich</td>
<td>359</td>
<td>379</td>
<td>309</td>
<td>345</td>
<td>230</td>
<td></td>
</tr>
<tr>
<td>Black Widow</td>
<td>104</td>
<td>8</td>
<td>4</td>
<td>97</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Enemy</td>
<td>182</td>
<td>31</td>
<td>142</td>
<td>152</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>JÄk</td>
<td>125</td>
<td>3</td>
<td>106</td>
<td>68</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>ZAP</td>
<td>271</td>
<td>68</td>
<td>221</td>
<td>132</td>
<td>228</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Unique vulnerabilities found between scanners

In this case, all scanners except Enemy of the State found the registration form. From here, only Black Ostrich was able to correctly solve all the patterns needed for registration and authentication.

Opera DNS UI [52]. This application uses many challenging patterns in its forms relating to DNS records, like the pattern for IPv4 addresses:

\[(25[0-5]|2[0-4]\.|0[0-9]\.)\{3\},3\|(25[0-5]\.|2[0-4]\.|0[0-9]\.)\{0,1,0-9\}\] 

As these are checked server-side too, only a valid IP address will be accepted in this case. As Enemy of the State does not support basic access authentication it could not log in. It would still fail to submit the form as the submission is triggered by JavaScript. The other scanners can log in and find the form. As the form’s method is post, JÄk does not interact with it. Both ZAP and Arachni manages to submit, but not valid data, and thus rejected by the server. Only Black Ostrich can submit a valid record.

9 CASE STUDY: EMAIL REGEXES

This section illustrates the viability of our implementation of ECMA regexes in a string solver by a case study of email regexes found in the wild. Browsers implementing the HTML5 input type also implement syntactic email validation using the following regex, henceforth referred to as MDN [19].

\([a-zA-Z0-9-_.]*\|[a-zA-Z0-9-_.]+@\[a-zA-Z0-9-_.]+\] ?\-9\-\]{0,61}\[a-zA-Z0-9-_.]+\] ?\-9\-\]{0,61}\[a-zA-Z0-9-_.]+\] ?\-9\-\]{0,61}\[a-zA-Z0-9-_.]+\] ?

The MDN validation strengthens the permissive requirements of RFC3696 [44], rejecting attack strings like `<script>alert(1)</script>@mail.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 $\rho$ out of the 825 collected ones such that $s$ matches $\rho$ contains a `<script>` tag by the client. 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 $\rho$ to find a string not matching $\rho$ 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 exclusion 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 (^$) 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 naïvely 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-str/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 best of our knowledge, our solver is the first to directly handle ECMA/Script regular expressions. A comparison with the tool ExpoSE, which includes support for ECMAScript regexes, is in Section 3.2.
2AFA. The equivalence between two-way and one-way automata has been originally proven by Rabin, but a most modern straightforward proof is given by Shepherdson in [64]. Alternation has been introduced only later, in the seminal work in the '70s [15]. Since then, the study of the combination of the two has been scattered, and we refer to [42] for a thorough survey. Although a first proof of equivalence between 2AFA and NFA appeared in [45], Birget [12] is the first to provide a comprehensive study of different kinds of finite automata, as well as a translation from (non-cycling) 2AFA to NFA which is done in several steps and make use of homomorphisms between alphabets. More recent translations appear in [31], which also works for cycling 2AFA. The above are theoretical construction, usually oriented to complexity theory, and to the best of our knowledge, they have not been implemented in practice.

Web Scanning. 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. Badras 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 [25], Li-gRE [24], and Black Widow [26] focuses 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 2AFA\textsubscript{SMT}. 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 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 (RT17-0011), the Swedish Research Council (VR) under the grants 2018-04727 and 2021-06327, and an Amazon Research Award (AWS Automated Reasoning).

REFERENCES

[23] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the state: A state-aware 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 (rt17-0011), the swedish research council (vr) under the grants 2018-04727 and 2021-06327, and an amazon research award (aws automated reasoning).