Generating Specifications to Verify the Correctness of Sanitizers

Daniël Huisman
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
d.a.j.huisman@student.utwente.nl

ABSTRACT

Web applications widely use string sanitizers to prevent injection vulnerabilities, such as SQL injection and cross-site scripting (XSS). However, it is difficult to write correct sanitizers without introducing injection vulnerabilities. It is therefore important to verify the correctness of sanitizers. Previous work has presented an approach for verifying correctness by comparing learned models of sanitizers to specifications of the desired behaviour. It can be time-consuming to write these specifications by hand. Therefore, this paper presents an approach for automatically generating sanitizer specifications from minimal user input. Firstly, a classification of different types of sanitizer specifications was made. Secondly, automatic generation techniques have been conceived and implemented. Lastly, a domain-specific language is introduced which enables users to easily interact with the generation techniques. The domain-specific language also allows users to combine, export and test the generated sanitizer specifications.

Keywords

Sanitizers, Specifications, Symbolic Finite Automata, Symbolic Finite Transducers, Software Verification

1. INTRODUCTION

Sanitizers are programs or algorithms that take user input and replace or removed unwanted character sequences. This is most commonly done to attempt to prevent injection vulnerabilities. Examples of injection vulnerabilities are SQL injections, code injection, command injection and cross-site scripting (XSS). These injection vulnerabilities are a critical security risk for web applications and risk exposure of sensitive information [12]. Therefore, sanitizers play an important role in handling user input in web applications.

An example of a sanitizer is the `htmlspecialchars` function in PHP [13], which replaces all HTML tags with HTML entities to prevent them from being interpreted. For instance, the HTML tag `<script>` would be replaced with `&lt;script&gt;`.

Although sanitizers are widely used in web applications, they can be difficult to write. Generally, sanitizers operate on input of unlimited size UTF-8 strings, which makes it hard to account for all possible variations. A small mistake in the sanitizer can lead to a serious injection vulnerability.

To address this problem, research has been done into verifying the correctness of sanitizers. Various approaches for verifying the correctness of sanitizers have been proposed, for example studying a theoretical model of the sanitizer [2] or applying black-box learning algorithms to find such a model [11]. This paper focuses on the approach of verifying sanitizers by comparing a learned model to a specification of the desired behaviour. These models of desired sanitizer behaviour are known as sanitizer specifications.

Sanitizer specifications can be tedious to write by hand, as the number of states and transitions significantly increases for more complex sanitizers [10]. The goal of this paper is to automatically generate these sanitizer specifications from some minimal user input. Here minimal user input is defined as specifying one or more characteristics for which the user wants to generate a specification rather than providing an exact algorithm of how the sanitizer should work. For example, the user could provide a blacklist of certain words or a minimum and maximum word length. The ability to automatically generate sanitizer specifications reduces the amount of time and effort that has to be spent on manually writing specifications. Instead, this time and effort can be spent on analysing the sanitizer and its behaviour.

Firstly, this paper gives some background on the models used to present sanitizer specifications in Section 2. Secondly, related work that was used as a starting point is discussed in Section 3. Next, the methodology of this research is laid out in Section 4. Then, Section 5 presents a classification of sanitizer specifications and discusses the possibility of automatic generation for each type in the classification. Section 6 describes the automatic generation techniques that were conceived and implemented. Section 7 introduces a domain-specific language for interaction with the generation techniques. Next, Section 8 discusses benchmarking the generation techniques for scalability. Finally, Section 9 discusses the overall goal of automatically generating sanitizer specifications, its limitations and potenial future work.

2. BACKGROUND

This paper uses Symbolic Finite Automata and Symbolic Finite Transducers to represent sanitizer specifications. These two concepts and their application for sanitizer specifications are explained in this section.

2.1 Symbolic Finite Automata

"Symbolic Finite Automata (SFAs) are finite state automata in which the alphabet is given by a Boolean alge-

34th Twente Student Conference on IT Jan. 29th, 2021, Enschede, The Netherlands.

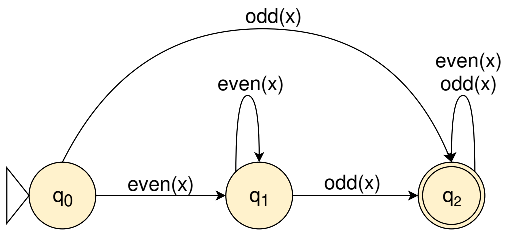

Figure 1. SFA that accepts all inputs which contain at least one odd number (Lathouwers [10]).

bra that may have an infinite domain, and transitions are labeled with first-order predicates over such algebra.” [4]

SFAs can be defined by a tuple (A, Q, q_0, F, Δ) [7]:

- A is an effective Boolean algebra
- Q is the finite set of states
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final states, also known as accepting states
- $\Delta \subseteq Q \times \Psi_A \times Q$ is the finite set of transitions, where Ψ_A is the set of predicates in A

An example of an SFA can be seen in Figure 1. Here transitions are labelled with *even* or *odd* predicates operating on the input number x. This SFA will for instance accept the number 316 and reject the number 226, because it contains no odd numbers. As seen in this example, SFAs allow the expression of wide range of input characters in a single transition by using predicates.

2.2 Symbolic Finite Transducers

Symbolic Finite Transducers (SFTs) extend Symbolic Finite Automata by introducing outputs. ”A Symbolic Finite Transducer also produces outputs when given an input. This output is computed using the given input character. Therefore all transitions in an SFT are labelled with an input condition as well as an output sequence.” [10]

SFTs can be defined by a tuple $(Q, Q_0, F, \iota, o, \Delta)$ [3, 11]:

- Q is the finite set of states
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final states, also known as accepting states
- ι is the input sort
- o is the output sort
- Δ is a function consisting of $\Delta^{\bar{\epsilon}} \cup \Delta^{\epsilon}$:
  - $\Delta^{\bar{\epsilon}}$ denotes all transitions labelled with a first-order predicate (which is the input condition) and the set of output functions. The output functions describe what output is generated when this transition is taken.
  - Δ^{ϵ} denotes all transitions labelled with ϵ as the input condition and the set of output functions. ϵ-transitions are transitions that can be taken without consuming an input at any point in time.

An example of an SFT can be seen in Figure 2. The input condition and output sequence are separated by a / sign. Note that a transition may produce multiple outputs. Taking the number 413 as input, the example SFT will end in an accepting state and output $[3, 5, 1, 3]$.

SFAs are a subset of SFTs, namely SFTs that produce only empty output [3]. Therefore, the SFAs can also be seen as SFTs without output.

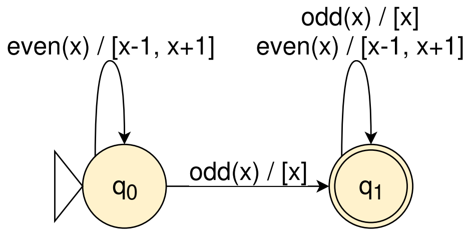

Figure 2. SFT that accepts all inputs which contain at least one odd number. For each even number it will output the two surrounding numbers and for each odd number it will output the odd number (Lathouwers [10]).

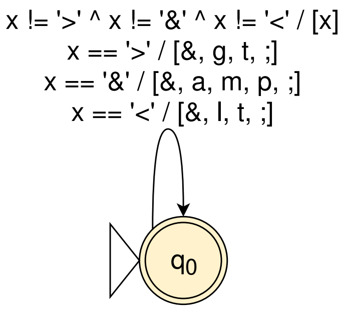
x != '>' ^ x != '&' ^ x != '<' / [x]
x == '>' / [&, g, t, ;]
x == '&' / [&, a, m, p, ;]
x == '<' / [&, l, t, ;]
q0

Figure 3. SFT which encodes $<$, $>$ and & into their HTML entities (Lathouwers [10]).

2.3 Sanitizer specifications

Sanitizers take some character sequence as input and remove or replace some of it. Symbolic Finite Transducers can be used to represent this behaviour. [2, 10]

An example of a sanitizer specification can be seen in Figure 3. This SFT replaces some characters, which are of special significance in HTML, with their respective HTML entities. The SFT represents a simple sanitizer that prevents a browser from interpreting HTML tags in the input.

Another example of a sanitizer specification is an SFA with two states, of which only the last is accepting, with a *true* transition between them. This SFA accepts all non-empty words. In other words, it is a sanitizer that requires a minimum length of one.

For both SFAs and SFTs equivalence can be checked. If one has an SFT representing the implementation of a sanitizer and an SFT representing the specification of the sanitizer, i.e. the desired behaviour, these two can be compared and checked for equivalence. This is one approach to verifying the correctness of sanitizers [10].

3. RELATED WORK

As mentioned in Section 1 and Section 2, there has been previous research on generating SFA or SFT representations of sanitizers. This research primarily focuses on generating an SFA or SFT from a given sanitizer, for example from its source code [2] or by using a black-box learning algorithm [11]. Another approach is introduced by Hooimeijer et al. [9] called BEK, which is a domain-specific language (DSL) for writing sanitizers and analysing their behavior. The BEK language can be compiled to other languages, such as JavaScript and C#. The BEK tools can also generate an SFA for a sanitizer written in BEK, which is then used in the analysis process. While this approach also generates an SFA from user input, the user input is not simple by our definition in Section 1. Effectively, BEK is just another way to write sanitizer source code. For a user to understand or investigate a sanitizer, they would first have to convert the source code of the original sanitizer to BEK before being able to reason about its SFA representation.

There has also been some previous research mentioning the generation of sanitizer specifications from user input. Lathouwers [10] proposes three techniques for automatic generation of certain types of sanitizer specifications, namely for whitelist, blacklist and length specifications. For the whitelist and blacklist specifications, the input is a list of acceptable or unacceptable words, respectively. Firstly, an SFA is constructed for each word on the list. Secondly, the union of these SFAs is computed. This results in a sanitizer specification for the entire whitelist or blacklist. For the length specification the inputs are an integer n and an operator, namely $=, \neq, <, >, \leq$ or $\geq$. From this input, an SFA can be constructed with n states with *true* transitions, which accept all input characters. Then, depending on the operator, certain states are marked as accepting states. This results in a sanitizer specification that is able to check certain requirements for the length of the input. Furthermore, Lathouwers [10] mentions exploring automatically generating specifications based on user input as future work to improve the user-friendliness of their SFTLearning tool. These proposed generation techniques were used as a starting point for this paper.

4. METHODOLOGY

To determine what types of sanitizer specification can be automatically generated, we started by classifying specifications into types and possibly subtypes. We used the classification of Lathouwers [10] as a starting point and expanded upon this classification by looking at properties that are checked in commonly used sanitizer implementations.

Afterwards, we iterated over the specification types as follows:

1. Select a (sub)type of specifications
2. Conceive a generation technique
3. Implement the generation technique
4. Test the generation technique to verify correctness
5. Benchmark the generation technique to determine scalability
6. Adjust type classification where necessary

4.1 Selecting a (sub)type

From the classification of sanitizer specifications, we picked one type or subtype for which no generation technique has previously been found. We started with simple specification types, e.g. only accept a certain length, and continued to more complex specification types. This way simpler techniques of previous iterations could be used as a basis for more complex techniques.

4.2 Conceiving a technique

We started by looking at examples of sanitizer implementations that belong to this classification type to determine similarities and differences. From the similarities, we create an initial technique. We then expanded upon this technique to address the differences until it was possible to generate all examples of this type. Furthermore, literature research was used to find solutions or inspiration in related work.

4.3 Implementing the technique

The generation technique were implemented in Java based on the work of Lathouwers et al. on SFTLearning [11]. Just like SFTLearning, these implementations use D'Antoni's Symbolic Automata library.

4.4 Testing the technique

Firstly, the generation technique were tested by hand by providing input and checking if the generated specification were correct. Secondly, test cases were written to automatically check a wide variety of inputs by comparing it to the output of the generated sanitizer specification. These automated tests were implemented using the JUnit 5 framework. The automated tests did not test all possible input combinations by rather focused on a diverse subset of input values.

4.5 Benchmarking the technique

To empirically determine the scalability of the generation technique, benchmarks were used. These benchmarks focus on two properties:

- Relation between input and the number of states and transitions in the specification
- Relation between input and the time complexity of the generation technique

These were both benchmarked by generating random input of increasing sizes and generating sanitizer specifications for them. For each input, the generated amount of states and transitions was counted and the execution time was measured.

4.6 Adjusting the classification

At any point in this process, it was possible that a generation technique did not work or only worked for a limited subtype of specifications. It was also possible that a generation technique could be extended to more types of specifications. These insights were noted and used to update the type classification.

5. CLASSIFICATION

This section presents the classification of sanitizer specifications. It is split into two categories, namely SFAs and SFTs. SFAs represent sanitizer specifications that only check whether input is valid or invalid. SFTs represent sanitizer specifications that also generate an output for the input.

5.1 Symbolic Finite Automata

Firstly, we separate two trivial SFAs: the SFA that accepts all input and the SFA that rejects all input. Secondly, the classification of Lathouwers [10] is added, see Section 3 for a description of these SFAs. Table 1 shows the initial SFA classification.

Specification type	User input
Trivial	Boolean b, whether to accept or not
Length	Integer n and operator $=, \neq, <, >, \leq$ or $\geq$
Blacklist	List of unacceptable words
Whitelist	List of acceptable words
Other	*Unknown*

Table 1. Initial classification of SFA specification types.

Next, we note that the blacklist and whitelist specifications are each other's complement. Therefore, these two specification types can be merged into a new classification called a word list. The word list requires a list of words and an operator, namely `equals` or `not equals`. The word list can match any input that is exactly on the list, but this can be extended by introducing two more operators, namely `contains` and `not contains`. The `contains` operator will match an input if the word occurs anywhere in the string at least once and the `not contains` operator behaves inversely.

It is now possible to match exact words, but this requires very specific and rigid user input. Another approach is matching a certain range of characters, for example the range `0-9` matches any digit. By extending the length specification with an additional character range parameter, is it possible to match inputs of a certain length and shape. The behaviour of the original length specification type can be recreated by providing the range of all possible characters (`\u0000-\uffff`) as parameter. Table 2 shows the updated classification.

Specification type	**User input**
Trivial	Boolean b, whether to accept or not
Range / length	Character range r, integer n and operator $=, \neq, <, >, \leq$ or $\geq$
Word list	List of words and operator `equals`, `not equals`, `contains` or `not contains`
Other	*Unknown*

Table 2. Classification of SFA specification types with word list and range added.

To extend the classification table, we can look at SFA operations for making combinations. There are at least three operations that can be performed to combine two SFAs, these are: concatenation, union and intersection [7]. By repeatedly performing these operations it is possible to combine multiple SFAs into one. First, these SFA operations allow us to simply the classification by only listing small building blocks for larger specifications. For example, the world list specification could also be formed by calculating the union of multiple word specifications. Note that for the negated operators the intersection operation has to be used to calculate the combined SFA. Secondly, these operations allow us to construct more complex specifications. Take, for example, the intersection of the length specifications `(5, >)`, `(10, <)` and the word specification `("cast", contains)`. The combined SFA would match any input between 6 and 9 characters containing the word `"cast"`, for instance `"multicast"` would match.

Looking at these combinations, the similarities with regular expressions (regex) become visible. In fact, it is possible to express a subset of regular expressions in regular SFAs [14]. We will refer to this subset as simple regular expressions and to all others as complex regular expressions. Simple regular expressions consist of the following features:

- Character classes
  - (Negated) character sets (`[abc]`, `[^abc]`)
  - Ranges (`[a-z]`)
  - Range shortcuts (`\w`, `\W`, `\d`, `\D`, `\s`, `\S`)
  - Wildcard (`.`)
  - Unicode categories (`\p{Z}`, `\P{Z}`)
  - Unicode scripts (`\p{Latin}`, `\P{Latin}`)
- (Non-)capturing groups (`(abc)`, `(?:abc)`)
- Quantifiers (`*`, `+`, `?`, `{1}`, `{1,}`, `{1,3}`)
- Alternation (`a|b`)
- String boundary anchors (`^`, `$`)

Complex regular expressions consist of, but are not limited to, the following features:

- Group references (`\1`)
- Lookaround (`(?=abc)`, `(?!abc)`, `(?<abc)`, `(?<!abc)`)
- Lazy quantifiers (`*?`, `+?`, `??`)
- Word boundary anchors (`\b`, `\B`)
- Flags (`i`, `g`, `m`, `u`, `y`)

Using regular expressions we can express a wide range of specification. Figure 4 shows an SFA for basic email address validation derived from the following regex [8]:
`/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/`

Furthermore, regular expressions can also express all the existing specification types in the classification. For instance, the range specification (`[a-z], 5,` $\neq$) is equal to the regex specification `/[a-z]{0,4}|[a-z]{6,}/`. However, the existing classification is kept, because there could be a performance benefit from directly generating these SFAs compared to generating them from regular expressions. Table 3 shows the updated classification, but without any of the combinations that could be made using the concatenation, union and intersection operations.

Specification type	**User input**
Trivial	Boolean b, whether to accept or not
Range / length	Character range r, integer n and operator $=, \neq, <, >, \leq$ or $\geq$
Word	Word w and operator `equals`, `not equals`, `contains` or `not contains`
Simple regex	Regular expression r
Complex regex	*Not possible, requires lookaround and/or registers*
Other	*Unknown*

Table 3. Classification of SFA specification types with regex added.

Finally, there are specifications that can not be represented by standard SFAs. For instance, validating if each HTML opening tag has a corresponding closing tag, which is not possible without registers [10].

5.2 Symbolic Finite Transducers

Firstly, we separate two trivial SFTs: the identity SFT, which accepts all input and produces output equal to the input, and the constant SFT, which accepts all input and always produces an output string s. Note that the trivial SFT that rejects all input does not produce any output and is therefore equivalent to the trivial SFA that rejects all output. SFTs that do not produce outputs are not listed in this classification, instead the SFA classification should be used. Table 4 shows the initial SFT classification.

Specification type	**User input**
Trivial, identity	-
Trivial, constant	Output string s
Other	*Unknown*

Table 4. Initial classification of SFT specification types.

Secondly, we introduce a simple SFT that accepts all input and replaces any occurrences of a specific character range with a certain output string. The character range input is the same as for the character range SFA specification. Next, we separate SFTs that operate on exact matches of a specific word, similar to the word list SFA

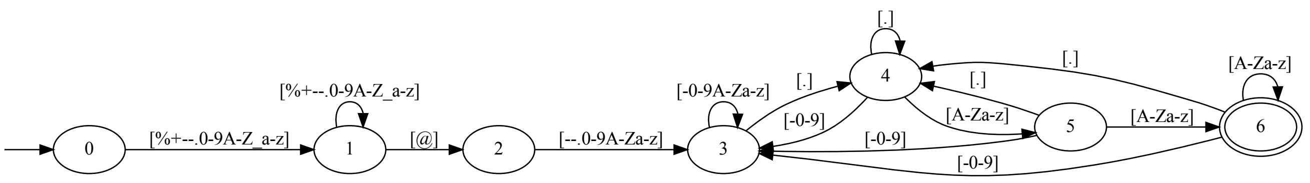

Figure 4. SFA generated from a regex which accepts valid email addresses.

specification. These SFTs will replace any occurrence of the specified word with a certain output string. However, there is a caveat, because these SFTs will not properly output strings that end in a partial match [10]. For example, an SFT that replaces `"abc"` with `"def"`, will output `"defdef"` for `"abcabc"`, but `"def"` for `"abcab"`. This can be resolved with ϵ-transitions, but that could make the SFT not single-valued and prevent equivalence checking [10]. Nonetheless, these SFTs are included in the classification.

Similarly to the SFA classification, we can define SFTs that operate on the length of the input string. The two most simple of these operations are trim and pad. Trim will reduce the length of the input string to a specified length and pad will extend the input string with a specified string to match a certain length. These operations can be performed on the start and the end of the string. The versions of trim and pad that operate on the start of the input string are not possible with regular SFTs. Instead, SFTs with registers are required to keep track of the start of the string after determining the entire length. The trim and pad versions that operate on the end of the input string are possible with regular SFTs. However, the end padding SFT requires ϵ-transitions, making it unsuitable for equivalence checking. Table 5 shows the updated classification.

Specification type	**User input**
Trivial, identity	-
Trivial, constant	Output string s
Replace range	Character range r and replacement string s
Replace word	Word w and replacement string s
Trim start	*Not possible, requires registers*
Trim end	Integer n
Pad start	*Not possible, requires registers*
Pad end	Integer n and padding string s
Other	*Unknown*

Table 5. Classification of SFT specification types with replace character and replace word added.

To extend the classification table, we can look at SFT operations for making combinations. There is at least one operation that can be performed to combine two SFTs, that is composition [7]. By repeatedly performing this operation it is possible to combine multiple SFTs into one. However, it is important to note that the composition operation is not commutative. Take, for example, the composition of the trim specification `(3)` and replace range specification `([&], "&amp;")`. The combined SFT would output `"&amp;ca"` for `"&card"`, because it trims the text before replacing characters. If the composition order of the two SFTs was reverse, it would output `"&am"` instead, because the trimming occurs after the characters have been replaced. Just like the SFA classification, the SFT classification also leaves out any combinations that could be made using the composition operation.

Finally, there are specifications that can not be represented by standard SFTs. For instance, replacing the match of a regular expression with a replacement string. Take the regular expression `/a*b/`. Without registers it is impossible to output the correct amount of `'a'` characters if the match fails.

6. GENERATION TECHNIQUES

This section presents the generation techniques that were developed for each sanitizer specification type ion the classification. The implementations, tests and benchmarks of all techniques can be found in the GitHub repository for this paper[1]. Each technique is implemented in a *generator*, which takes the required specification parameters as input and produces an automaton (SFA or SFT) as output.

6.1 SFA - Character range

The technique for generating character range specifications is based on the technique proposed by Lathouwers [10] for generating length specifications. The main difference is that this technique replaces the *true*-transitions with transitions that have the character range as a predicate (line 3). This technique is described in Algorithm 1.

Algorithm 1: Generate character range specification

```
Input: Character range $r$, length $n$, operator
       $o \in \{=, \neq, <, >, \leq, \geq\}$
1  Create empty SFA with initial state 0
2  for $i \in \{0..n-1\}$ do
3      Add transition $(i, i+1)$ with predicate $r$
4  end
5  if $o \in \{=, \leq, \geq\}$ then
6      Add $n$ as final state
7  end
8  if $o \in \{\neq, <, \leq\}$ then
9      for $i \in \{0..n-1\}$ do
10         Add $i$ as final state
11     end
12 end
13 if $o \in \{\neq, >, \geq\}$ then
14     Add transition $(n, n+1)$ with predicate $r$
15     Add transition $(n+1, n+1)$ with predicate $r$
16     Add $n+1$ as final state
17 end
18 return SFA
```

6.2 SFA - Word

The technique for generating word specifications starts of similar to the previous technique. The main difference is that this technique uses the characters of the word as separate predicates instead of using the same predicate for all transitions (line 3). Furthermore, it has some additional logic for handling the `contains` ($\supset$) and `not contains` ($\not\supset$) operators. This technique is described in Algorithm 2.

[1]https://github.com/DanielHuisman/sanitizer-specifications

Algorithm 2: Generate word specification

```
Input: Word $w$, operator $o \in \{=, \neq, \supset, \not\supset\}$
1  Create empty SFA with initial state 0
2  for $i \in \{0..|w| - 1\}$ do
3      Add transition $(i, i+1)$ with predicate $w_i$
4  end
5  if $o \in \{=, \supset\}$ then
6      Add $|w|$ as final state
7  end
8  if $o \in \{\neq, \not\supset\}$ then
9      for $i \in \{0..|w| - 1\}$ do
10         Add $i$ as final state
11     end
12 end
13 if $o \in \{\neq\}$ then
14     /* Add final state after word with self
          transition                              */
15     Add transition $(|w|, |w| + 1)$ with predicate $true$
16     Add transition $(|w| + 1, |w| + 1)$ with predicate
        $true$
17     Add $|w| + 1$ as final state
18     /* Add transitions from char to final state
          excluding said char                     */
19     for $i \in \{0..|w| - 1\}$ do
20         Add transition $(i, |w| + 1)$ with predicate $\overline{w_i}$
21     end
22 end
23 if $o \in \{\supset, \not\supset\}$ then
24     /* Add self transition for last char       */
25     Add transition $(|w|, |w|)$ with predicate $true$
26     /* Add transitions from char to initial
          state excluding said char               */
27     for $i \in \{0..|w| - 1\}$ do
28         Add transition $(i, 0)$ with predicate $\overline{w_i}$
29     end
30 end
31 return SFA
```

6.3 SFA - Simple regular expressions

The technique for generating simple regular expression specifications is based on techniques for constructing an NFA/DFA for a regular expression. To start off, the regular expressions string is first parsed to an abstract syntax tree (AST). In this implementation, ANTLR 4 was used to express the grammar and to generate a corresponding parser. The AST has three types of nodes with each their own properties:

- Operator
  - Operator: `CONCAT`, `OR`
  - Children: list of nodes
- Quantifier
  - Child: node
  - Minimum: integer
  - Maximum: integer, -1 for none
- Character class
  - Class: `CHARACTER`, `SET`, `NEGATED_SET`
  - Ranges: list of character ranges

Next, the SFA for the AST is generated recursively (line 3 and 14). For character class nodes, a simple SFA is generated consisting of two states and one transition with the character range(s) as predicate (lines 50-60). For operator nodes, the SFAs of all children are generated and combined (lines 2-12).

Algorithm 3: Generate regular expression specification

```
Input: AST node $n$
1  switch $n.type$ do
2      case $Operator$ do
3          Let $s = \{generate(c) \mid c \in n.children\}$
4          switch $n.operator$ do
5              case CONCAT do
6                  return concatenation of $s$
7              end
8              case OR do
9                  return intersection of $s$
10             end
11         end
12     end
13     case $Quantifier$ do
14         Let $s = generate(n.child)$
15         Let $c = n.max$
16         if $n.max < 0$ then $c = n.min$
17         Create empty SFA
18         Set $s.initialState$ as initial state
19         for $i \in \{0..c - 1\}$ do
20             Let $o_c = (1 + |s.transitions|) * i$
21             Let $o_n = (1 + |s.transitions|) * (i + 1)$
22             for $t \in s.transitions$ do
23                 Add copy of $t$ as transition
                    $(o_c + t.from, o_c + t.to)$
24             end
25             if $i < c$ - $1$ then
26                 for $f \in s.finalStates$ do
27                     Add $\epsilon$-transition
                        $(o_c + f, o_n + s.initialState)$
28                 end
29             end
30         end
31         Let $o_l = (1 + |s.transitions|) * (c - 1)$
32         if $n.min = 0$ then
33             for $f \in s.finalStates$ do
34                 Add $\epsilon$-transition $(s.initialState, o_l + f)$
35             end
36         end
37         if $n.max < 0$ then
38             for $f \in s.finalStates$ do
39                 Add $\epsilon$-transition
                    $(o_l + f, o_l + s.initialState)$
40             end
41         end
42         for $i \in \{max(n.min, 1)..c\}$ do
43             Let $o_c = (1 + |s.transitions| * (i - 1)$ for
                $f \in s.finalStates$ do
44                 Add $o_c + f$ as final state
45             end
46         end
47         if $n.parent = null$ then Minimize SFA
48         return SFA
49     end
50     case $Character\ class$ do
51         Let $p = null$
52         for $r \in n.ranges$ do
53             if $p = null$ then $p = r$ else $p = p \vee r$
54         end
55         if $n.class$ = NEGATED_SET then $p = \overline{p}$
56         Create empty SFA with initial state 0
57         Add transition $(0, 1)$ with predicate $p$
58         Add 1 as final state
59         return SFA
60     end
61 end
```

For quantifier nodes, first the SFA of the child is generated and then that SFA is copied and connected as many times as needed (lines 13-49). The copying of the child SFA occurs at an offset to prevent duplicate state numbers (lines 20-24). The final states of a copied SFA are connected to the next copied SFA using ϵ-transitions (lines 25-30). However, these copied final states are not actually marked as final in the new SFA. Additionally, ϵ-transitions are added to handle a minimum amount of zero occurrences (lines 31-36) and an unlimited maximum amount of occurrences (lines 37-41). The final states of copied SFAs are marked as actual final states at each relevant offset as specified by the minimum and maximum amount of occurrences (lines 42-46). Lastly, the SFA is minimized to remove the ϵ-transitions (line 47). The minimization algorithm used in the D'Antoni's automata library is an implementation of the minimization algorithm by D'Antoni and Veanes [6]. This full technique is described in Algorithm 3 and Figure 4 shows an example of a specification generated by this technique, as described in Section 5. Note that the generation technique can be adapted to similar ASTs of regular expressions.

6.4 SFT - Replace character range

The technique for generating replace character range specifications is straightforward. There is only one state with two self transitions. The first has the range as predicate and outputs the specified replacement. The second has the complement of the range as predicate and identity function as output.

6.5 SFT - Replace word

The technique for generating replace word specifications is an extension of the technique for generating SFA word specifications. The SFT has states for each character of the word. The transitions between them do have outputs, except for the last one, which outputs the replacement string. Furthermore, transitions are added back to the first and second states that output the partial matches. This technique is described in Algorithm 4.

Algorithm 4: Generate replace word specification

```
Input: Word $w$, replacement string $s$
1  Create empty SFT with initial state 0
2  for $i \in \{0..|w|-1\}$ do
3      if $i = |w| - 1$ then
4          Add transition $(i, i+1)$ with predicate $w_i$ and output $s$
5      else
6          Add transition $(i, i+1)$ with predicate $w_i$ and output $none$
7      end
8      Add transition $(i, 0)$ with predicate $\overline{w_0 \vee w_i}$ and output $w_{0..i}, identity$
9      if $i > 0$ then
10         Add transition $(i, 1)$ with predicate $w_0$ and output $w_{0..i}$
11     end
12     Add $i$ as final state
13 end
14 Add transition $(|w|, 0)$ with predicate $\overline{w_0}$ and output $identity$
15 Add transition $(|w|, 1)$ with predicate $w_0$ and output $none$
16 Add $|w|$ as final state
17 return SFT
```

6.6 SFT - Trim/pad end of string

The technique for generating trim end specifications is straightforward. Add n final states that output the input (identity function) and add one final state with a self transition that does not output anything.

The technique for generating pad end specifications is similar, but several ϵ-transitions are added for outputting the padding (lines 4-7). The padding is determined by repeating the specified padding string and then taking a substring. For example, take padding `"ab"`, desired length 10 and an input string of length 5. This string requires 5 characters of padding, so the padding string is repeated $\lceil 5/2 \rceil = 3$ times before taking the substring of the first 5 characters, which is `"ababa"`.

Both techniques are described in Algorithm 5.

Algorithm 5: Generate trim/pad end specification

```
Input: Trim/pad $o$, length $n$, padding string $s$
1  Create empty SFT with initial state 0
2  for $i \in \{0..n-1\}$ do
3      Add transition $(i, i+1)$ with predicate $true$ and output $identity$
4      if $o = Pad$ then
5          Let $a = n - i$, $b = \lceil a/|s| \rceil$, $p = w$ repeated $b$ times
6          Add $\epsilon$-transition $(i, n+1)$ with output $p_{0..a}$
7      end
8  end
9  if $o = Trim$ then
10     Add transition $(n, n)$ with predicate $true$ and output $none$
11     Add $n$ as final state
12 else if $o = Pad$ then
13     Add transition $(n, n)$ with predicate $true$ and output $identity$
14     Add $\epsilon$-transition $(n, n+1)$ with output $none$
15     Add $n+1$ as final state
16 end
17 return SFT
```

7. DOMAIN-SPECIFIC LANGUAGE

This section describes the domain-specific language (DSL) that was developed and implemented to allow users to interact with the generation techniques. During the development of the generation techniques, it became clear that calling the generators from Java source code added unnecessary complexity. The overall goal of this research was to allow users to generate sanitizer specifications without in-depth technical knowledge. To truly reach this goal, a simpler way to interact with the generators was desired. The two options were a graphical user interface or a text based interface. Due to time constraints, the latter was chosen and this DSL was implemented using ANTLR 4.

The DSL has the following features:

- Primitive types (integer, character, string, tuple, list, regex)
- Generators, these are implementation of the techniques that generate automata from some parameters.
- Variables, these store generated automata.
- Expressions (`not`, `plus`, `and`, `or`), these combine automata.
- Statements
    - `accepts/rejects`, test whether an automaton accepts or rejects some input string.

– `outputs`, test whether an automaton produces certain output for some input string.
– `import/export`, import/export an automaton from/to a DOT graph file.

An example of the DSL can be found in Listing 1. The character range and word SFA specifications are combined and tested in the first four lines. The replace character SFT specification is generated and tested in the last two lines.

Listing 1. Domain-specific language
```
var1 = range (">=", 10, ('a', 'z')) or
            word ("contains", "tuner")
export var1
accepts var1 "dvbtuner"
rejects var1 "cheese"

var2 = trim 3 and replace-char ('a', "")
outputs var2 "abcde" "bc"
```

8. BENCHMARKS

Some of the generation techniques were benchmarked to empirically determine the relation between the input size and the numbers of states, the number of transitions and the generation time. Not all generators had input that could easily be isolated and scaled. For example, regular expressions consist of multiple different combination operations which makes it difficult to determine the overall scalability.

8.1 States and transitions

The first part of the benchmarks relate the input to the numbers of states and transitions in the specification. For these benchmarks one input parameter was chosen for scaling up in size, but for some generators multiple rounds were benchmarked. For example, the character range specification has multiple operators, so these were all tested separately, but in all tests the length parameter was used for scaling. The character range parameter has no effect on the states or transitions, because it would only alter the predicates, so this parameter was left constant for all benchmarks. For the range parameter, `[a-z]` was chosen and for the word and padding parameters a repeating alphabetic string was used (e.g. `"abcd"` for size 4). The comparison of the number of states can be found in Table 6 and the comparison of the number of transitions can be found in Table 7. All these individual generators have linear states and transitions, but some have a different starting value or slope. These results are not surprising given the linear nature of the algorithms.

Generator	**Operator**	**0**	**1**	**10**	**100**	**1000**	**10000**
Range	$=, <, \leq$	1	2	11	101	1001	10001
Range	$\neq, >, \geq$	2	3	12	102	1002	10002
Word	$=, \supset, \not\supset$	1	2	11	101	1001	10001
Word	$\neq$	2	3	12	102	1002	10002
Rep. char		1	1	1	1	1	1
Rep. word		1	2	11	101	1001	10001
Trim		1	2	11	101	1001	10001
Pad		2	3	12	102	1002	10002

Table 6. Comparison of number of states

8.2 Time complexity

The second part of the benchmarks relate the input to the time spent generating the specification. The setup of these benchmarks is equal to the setup for benchmarking the amount of states and transitions. The generation time was measured in nanoseconds, but is displayed in milliseconds in the table. The generation time can be influenced by the JVM and the rest of the system, so the generation for each input size was performed multiple times and averaged. The differences between range/word operators was minimal, so the these times were also averaged to determine the overall performance of the generator rather than that of one operator. The comparison of generation times can be found in Table 8. Half of the generators show a roughly linear time complexity, which is confirmed by plotting the data in a graph with more data points. The replace character generator is constant, because there is no input size to vary. The replace word and pad generators quickly jump up in time, but still perform well for the large input sizes.

Generator	**Operator**	**0**	**1**	**10**	**100**	**1000**	**10000**
Range	$=, <, \leq$	1	2	11	101	1001	10001
Range	$\neq, >, \geq$	2	3	12	102	1002	10002
Word	$=, \supset, \not\supset$	1	2	11	101	1001	10001
Word	$\neq$	2	3	12	102	1002	10002
Rep. char		1	1	1	1	1	1
Rep. word		0	4	31	301	3001	30001
Trim		1	2	11	101	1001	10001
Pad		2	4	22	202	2002	20002

Table 7. Comparison of number of transitions

Generator	**1000**	**2000**	**3000**	**4000**	**5000**
Range	0.7	1.9	3.7	5.3	5.8
Word	1.6	4.2	7.0	8.6	9.6
Rep. char	0	0	0	0	0
Rep. word	28	96	196	374	559
Trim	0.3	0.9	1.6	3.2	3.6
Pad	11	30	52	85	123

Table 8. Comparison of average generation time in milliseconds

In general, the benchmarks show that the basic generation techniques scale well, even up to unrealistic input sizes, e.g. replacing words of 1000 character lengths.

9. CONCLUSION

As shown in this paper, it is possible to automatically generate sanitizer specifications. For SFAs we showed that a decent subset of regular expressions can be expressed. For SFTs we showed that only a few simple operations can be expressed. These limitations mainly come from the overall approach of representing sanitizer specifications in the limited feature set of SFAs and SFTs.

There is only a small selection of behaviour observed in real world sanitizers that can be represented by these automata. For example, the JavaScript sanitize-html library can remove script blocks, e.g. `<script>alert("<script> alert(1)</script>")</script>`. This would require the SFT to keep track of the HTML structure or it would only strip until the first closing tags. This is common and desired behaviour, as the library is downloaded roughly 900.000 times per week [1].

As future research, the classification of sanitizer specifications could be improved by looking at extensions of SFAs and SFTs. Specifically, by looking at Symbolic Register Automata (SRAs) [5] and Symbolic Register Transducers (SRTs) [3], which add registers to SFAs and SFTs respectively. The addition of registers could allow more sanitizers to be represented in automata, such as more complex regular expressions or the structure of HTML documents. D'Antoni et al. [5] show an example of how regular expressions with backreferences can be expressed using SRAs.

10. REFERENCES

[1] Apostrophe Technologies. sanitize-html. `https://www.npmjs.com/package/sanitize-html`. Accessed on 23 January 2021.

[2] N. Bjørner, B. Livshits, D. Molnar, and M. Veanes. Symbolic finite state transducers: Algorithms and applications. Technical Report MSR-TR-2011-85, Microsoft Research, July 2011.

[3] N. Bjørner and M. Veanes. Symbolic transducers. Technical Report MSR-TR-2011-3, Microsoft Research, January 2011.

[4] L. D'Antoni. Symbolic automata. `https://pages.cs.wisc.edu/~loris/symbolicautomata.html`. Accessed on 21 November 2020.

[5] L. D'Antoni, T. Ferreira, M. Sammartino, and A. Silva. Symbolic register automata. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2019.

[6] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 541–554. ACM, 2014.

[7] L. D'Antoni and M. Veanes. The power of symbolic automata and transducers. In R. Majumdar and V. Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 47–67. Springer, 2017.

[8] J. Goyvaerts. How to Find or Validate an Email Address. `https://www.regular-expressions.info/email.html`, 2019. Accessed on 14 January 2021.

[9] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *USENIX Security'11*, August 2011.

[10] S. Lathouwers. Reasoning about the correctness of sanitizers. Master's thesis, University of Twente, Enschede, the Netherlands, 2018.

[11] S. Lathouwers, M. Everts, and M. Huisman. Verifying sanitizer correctness through black-box learning: A symbolic finite transducer approach. In S. Furnell, P. Mori, E. Weippl, and O. Camp, editors, *Proceedings of the 6th International Conference on Information Systems Security and Privacy*, pages 784–795. SCITEPRESS Digital Library, 2020.

[12] OWASP Foundation. OWASP Top Ten Web Application Security Risks. `https://owasp.org/www-project-top-ten`, 2017. Accessed on 20 November 2020.

[13] PHP Group. PHP: htmlspecialchars - Manual. `https://www.php.net/manual/en/function.htmlspecialchars.php`. Accessed on 20 November 2020.

[14] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. Technical Report MSR-TR-2009-137, October 2009. ICST'10.