

Verifying Sanitizer Correctness through Black-Box Learning: A Symbolic Finite Transducer Approach

Sophie Lathouwers¹ ^a, Maarten Everts² ^b and Marieke Huisman¹ ^c

¹Formal Methods and Tools, University of Twente, Enschede, The Netherlands

²Services and Cybersecurity, University of Twente & TNO, Enschede, The Netherlands

Keywords: Automata Learning, Sanitizers, Symbolic Finite Transducers, Injection Attacks, Software Verification.

Abstract: String sanitizers are widely used functions for preventing injection attacks such as SQL injections and cross-site scripting (XSS). It is therefore crucial that the implementations of such string sanitizers are correct. We present a novel approach to reason about a sanitizer's correctness by automatically generating a model of the implementation and comparing it to a model of the expected behaviour. To automatically derive a model of the implementation of the sanitizer, this paper introduces a black-box learning algorithm that derives a Symbolic Finite Transducer (SFT). This black-box algorithm uses membership and equivalence oracles to derive such a model. In contrast to earlier research, SFTs not only describe the input or output language of a sanitizer but also how a sanitizer transforms the input into the output. As a result, we can reason about the transformations from input into output that are performed by the sanitizer. We have implemented this algorithm in an open-source tool of which we show that it can reason about the correctness of non-trivial sanitizers within a couple of minutes without any adjustments to the existing sanitizers.


1 INTRODUCTION


Injection flaws have been identified as the most serious web application security risk by the OWASP Top Ten project (OWASP Foundation, 2017). Some examples of injection vulnerabilities include cross-site scripting (XSS), code injection, command injection and SQL injection. Injection vulnerabilities occur when untrusted data is interpreted by the system, which can result in the execution of a user-given command or the injection of malicious data into the system. This may have consequences such as disclosure of personal information, modification of data and even deletion of data. To prevent exploitation of injection vulnerabilities one can aim to detect vulnerabilities on time and repair them, reject malicious input, limit the privileges of users or modify the given input. This research focuses on the approach where input, given by the user to the system, is modified such that dangerous characters are removed.


Sanitizers, also called string manipulating programs, are programs that remove or replace

such unwanted characters. For example, the `FILTER_SANITIZE_EMAIL` function in PHP (The PHP Group, 2018b) removes all characters that are not allowed in email addresses. Sanitizers are widely used in practice, however, writing them is quite difficult. This is because sanitizers are used in a security-sensitive environment where a small mistake can already introduce a vulnerability in the application. To address this problem, we investigate how we can easily *verify the correctness of existing sanitizers*.

Figure 1 gives an overview of the methodology that we use to reason about the correctness of sanitizers. We developed a black-box learning algorithm that can automatically deduce a model, called a Symbolic Finite Transducer (SFT), from a sanitizer by inspecting the input and output. We compare the learned model to a specification written by the user to reason about the correctness of a sanitizer. As far as we are aware, this is the first approach to automatically derive a model that reasons about the input-output behaviour in a black-box manner. Moreover, aside from writing the specification, the approach is fully automatic and can be applied to existing sanitizers written in any language. We have evaluated this approach by analysing the correctness of existing real-world sanitizers. We identify what types of sanitizers can be

^a  <https://orcid.org/0000-0002-7544-447X>

^b  <https://orcid.org/0000-0002-5302-8985>

^c  <https://orcid.org/0000-0003-4467-072X>

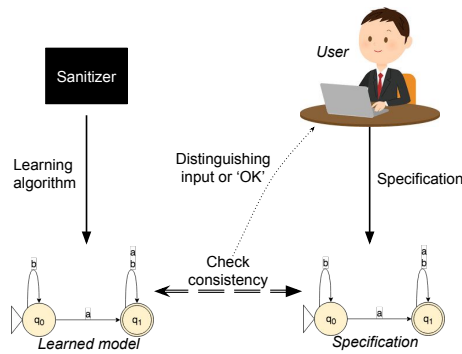


Figure 1: An overview of the methodology that we use to reason about the correctness of sanitizers. We use a black-box learning algorithm to derive a model from the sanitizer which is compared to a specification written by the user. Afterward, any discrepancies between the models are reported to the user.

learned using this approach and what the bottlenecks of the current implementation are.

Contributions:

1. A novel approach to study the correctness of sanitizers by comparing learned models to a specification.
2. A *black-box SFT learning algorithm* that uses equivalence and membership queries.
3. *Implementation* of the black-box SFT learning algorithm and *evaluation* of its performance and applicability.

2 SYMBOLIC FINITE AUTOMATA AND TRANSDUCERS

For this research we used automata to represent sanitizers. Mealy machines are unfortunately not suitable to represent sanitizers because it would need one transition per input character per state in the automaton. If this is applied in a setting with strings, with many possible input characters, this would result in very large and cluttered automata. For example, if we only reason about the generic alphabet (a-z), this would already lead to 26 transitions per state. To make the automata more concise, we turn to symbolic finite automata (SFAs) and symbolic finite transducers (SFTs) which can concisely represent similar transitions for many different input characters. For example, all characters that are not changed by a sanitizer can be represented by a single transition per state in an SFT. SFAs can be used to reason about behaviour that is only related to the input or the output language.

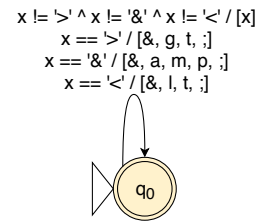


Figure 2: An SFT that encodes `<`, `>` and `&` into their HTML references `<`, `>` and `&`. This describes the behaviour of the python method `escape` with the optional flag set to `False` (Python Software Foundation, n.d.).

SFTs can be used to reason about the relation between the input and the output.

A *Symbolic Finite Transducer* (SFT) can be described using the tuple $(Q, q_0, F, \iota, \sigma, \Delta)$ (Bjørner and Veanes, 2011) where:

- Q is the finite set of states
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of accepting states
- ι is the input sort
- σ is the output sort
- Δ is a function consisting of $\Delta^\varepsilon \cup \Delta^\varepsilon$:
 - Δ^ε denotes all transitions in the automaton 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 in the automaton 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. Each transition is labelled \bar{y}/z where \bar{y} denotes the input condition and z denotes the set of output functions. The initial state is indicated by an arrow and accepting states are encircled twice in the figure. To check whether an SFT accepts a certain input, we start in the initial state. For each character of the input, denoted by x in the input condition, we evaluate the predicates of the transitions starting in the current state. We will then follow the transition whose predicate evaluates to true for this input character and generate the corresponding output. If we end in an accepting state, then the input is accepted.

An SFA is an SFT that produces empty outputs (Bjørner and Veanes, 2011). Thus, an SFA looks similar to an SFT, the only difference is that there are no output functions for transitions.

3 SPECIFICATIONS

To reason about the correctness of sanitizers, we establish what behaviour is considered correct by writing a specification which describes how the sanitizer is supposed to behave. In this section, we therefore first discuss what types of specifications can be checked with our approach (see Section 3.1). Followed by Section 3.2 which describes how the correctness of the specifications can be checked.

Instead of comparing the specification and implementation, it is also possible for the user to inspect the learned model itself without writing a specification. However, we think that writing a specification is important since it forces the user to think about what correct behaviour would be. Moreover, it is easy to overlook mistakes in a model. Aside from that, specifications can also be reused for similar sanitizers whereas manual inspection would be required for each new implementation.

3.1 Type of Specifications

To discover the type of behaviours that users are interested in, we looked at the literature (Hooimeijer et al., 2011) and organised a brainstorming session with employees of a security company called Northwave (Northwave, n.d.), that specialises in, among other things, security testing.

The following type of specifications can be checked with our approach:

- **Blacklisting:** Specify which (sequences of) characters are not allowed in the input or output.
- **Whitelisting:** Specify which (sequences of) characters are allowed in the input or output.
- **If z then $x \rightarrow y$:** If condition z is satisfied, then all occurrences of x are replaced by y . Note that this can also be used to specify that something must not change, in that case, you specify “if z then $x \rightarrow x$ ”. If z is replaced by *True* then the specification means that x should always be changed into y .
- **Length:** Specify the allowed length of the input or output.
- **Equivalence, Idempotency and Commutativity:** Check whether two sanitizers behave the same, whether a sanitizer is idempotent or whether a sanitizer commutes with another sanitizer.
- **Bad Output:** Given a bad output, search for an input that leads to this output.

We can divide these types of specifications into two categories: input/output-only and input-output related.

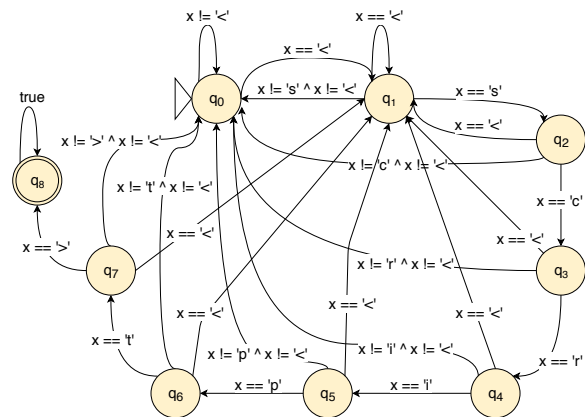


Figure 3: An SFA that accepts all inputs that contain `<script>`.

Input/output-only Specifications are specifications that only reason about the input language or the output language. Specifications that are input/output-only, and should be expressed with SFAs, include: blacklisting, whitelisting and length specifications.

Input-output Related Specifications reason about the transformation from the input into the output. Input-output related specifications, that should be expressed with SFTs, include: If z then $x \rightarrow y$, equivalence, idempotency, commutativity and bad output specifications.

3.2 Checking Specifications

Next, we explore what users need to specify to check each type of specification mentioned in the previous section. It also explains how each specification is compared to the learned model using the information provided by the user. It is important that if a specification is in the form of an SFT, then the SFT needs to be single-valued, i.e., it always produces the same output upon a given input. This is necessary in order to determine equivalence. While determinism of the SFT implies that it is single-valued, nondeterministic SFTs can also be single-valued (Veanes et al., 2012). In case the specification is in the form of an SFA, we can compare this to an SFA of the input or output language which can be derived from an SFT.

Blacklisting: The user needs to construct an SFA that accepts all unacceptable inputs or outputs as specified on the blacklist. To check the specification, compute the union of the specified SFA with the SFA that represents the input or output language. If the union is equal to the empty automaton, then no disallowed input or output is accepted. For example, if the text “`<script>`” is disallowed, then we specify an automaton that accepts everything containing this text (see

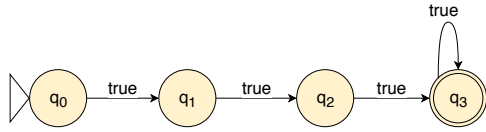


Figure 4: An SFA that accepts all inputs that consist of at least three characters.

Figure 3).

Whitelisting: The user needs to construct an SFA that accepts all acceptable inputs or outputs as specified in the whitelist. The specification can be checked in one of the following two ways: (1) check if the two SFAs are equivalent, if so, then they accept the same inputs or outputs, or (2) check if the learned SFA is a subset of the specified SFA, if so, then the sanitizer accepts some, perhaps all, inputs or outputs from the whitelist.

Length: The user needs to specify an SFA that accepts all words of length x . To check the specification, first, compute the complement of the specified automaton. Next, compute the union of the complement and the SFA that represents the input or output language. If the union is non-empty, then there exists a word with a length that is not x . Otherwise, all words have length x . The user can also specify an SFA that accepts all words of length $< x$, $\leq x$, $> x$, $\geq x$ to check whether all words have the corresponding length. For example, if we only allow text which consists of at least three characters, then we specify the automaton in Figure 4.

If z then $x \rightarrow y$: For this, the user needs to specify the complete system in the form of an SFT. To check the specification, we check for equivalence between the learned model and the specification. For example, if the characters $<$, $>$ and $\&$ are translated into their HTML references, we write a specification as in Figure 2.

Equivalence, Idempotency and Commutativity: For equivalence, the user needs to specify which sanitizers should be compared. To check equivalence, we check whether domain and partial equivalence hold (Hooimeijer et al., 2011). To check for idempotency, we compose the learned SFT that represents the sanitizer with itself. If the composed SFT is equivalent to the SFT that represents the sanitizer, then the sanitizer is idempotent. When checking commutativity, the user needs to specify between which two sanitizers, A and B , (s)he wants to check for commutativity. Then, we compute the composition of A with B as well as the composition of B with A . If the compositions are equal, then the sanitizers A and B commute.

Bad Output: The user needs to specify the bad output in the form of a string. For example, the user might be looking for an input that leads to the out-

Table 1: Example of an SOT. *ID* is an abbreviation of the *IDENTITY* function type.

Input		W	
		ϵ	ϵ
S	ϵ	$f = \epsilon$	$T = [ID]$
	\backslash	$f = \backslash \backslash$	$T = [ID, ID]$
	$\backslash \backslash$	$f =$	$T = []$
Λ	a	$f = a$	$T = [ID]$
	$\backslash a$	$f = a$	$T = [ID]$
	$\backslash \backslash a$	$f = \backslash \backslash$	$T = [ID, ID]$
	$\backslash \backslash a$	$f = a$	$T = [ID]$

put “`<script>`”. To find such an input, we use a pre-image computation over the SFT that represents the sanitizer. This can be implemented with a backward breadth-first search.

4 LEARNING ALGORITHM FOR SFTs

In this section, the black-box learning algorithm for SFTs is explained. This algorithm allows us to derive a model of the sanitizer’s implementation to which the specification can be compared.

Firstly, some necessary background information is introduced in Section 4.1. Secondly, the main algorithm for learning SFTs is shown in Section 4.2. Thirdly, details about the hypothesis generation in the main algorithm are discussed in Section 4.3. Finally, the implementation of the equivalence oracle that is used in the algorithm is discussed in Section 4.4.

4.1 Preliminaries

The automata-based string analysis technique that we have developed is a black-box learning algorithm inspired by Angluin’s L^* algorithm (Angluin, 1987). Our algorithm learns the behaviour of the System Under Learning (SUL) without needing access to the implementation. In order to do this, the algorithm can ask two types of questions:

- *Membership Queries:* What is the output of the SUL when it is given the string s ?
- *Equivalence Queries:* Given a hypothesis automaton, either obtain a confirmation that it is a correct hypothesis automaton or obtain a counterexample that distinguishes the hypothesis and the SUL.

We store the results of these queries in a Symbolic Observation Table (SOT). The SOT is represented by the tuple (S, W, Λ, f, T) . In this definition we also use

Σ and Γ which represent the input and output alphabet respectively:

- $S \subseteq \Sigma^*$ is a set of access strings (Argyros et al., 2016).
- $W \subseteq \Sigma^*$ is a set of distinguishing strings (Argyros et al., 2016).
- $\Lambda \subseteq S \cdot \Sigma$ is a set of one-step extensions of S , i.e., this is a (sub)set of access strings which are concatenated with a character from the input alphabet.
- $f : \Sigma^* \times \Sigma^* \rightarrow \Gamma^*$ is a partial function that results in the suffix of the output. The suffix is equal to the output corresponding to the input string sd when we have removed the largest common prefix of the output corresponding to the input string s . sd consists of some input string s followed by a single character d from the input alphabet.
- $T : \Sigma^* \times \Sigma^* \rightarrow \{IDENTITY, CONSTANT\}^*$ is a partial function that results in a set of types of output functions. It corresponds to an encoding of the output found in f . For each character in the output, T specifies whether it corresponds to an identity function or a constant (compared to the input character).

Note that this is a *different* SOT as used by Argyros et al. (Argyros et al., 2016). The difference is that we store the output in f and its encoding in T whereas Argyros et al. only store the output for their SFA learning algorithm. This encoding is necessary to ensure that similar inputs are represented by one state in the SFT.

Next, we discuss an example of an SOT for a sanitizer that escapes all (unescaped) backslashes with another backslash. In Table 1, you can see the final SOT that was generated when learning. Consider the input “\a”. When giving this to the sanitizer, this should result in “\\a” as output. In the SOT we store the output that was generated for the last character, in this case “a”, in f . As output we generated the same character as the input character, therefore the stored encoding is ID (which represents the identity function) in T . If you want to deduce the complete output, we would need to have a look at all prefixes of the input “\a”: $\{\epsilon, \backslash, \backslash a\}$ and their corresponding generated output functions. For ϵ the generated output is the identity function, which is equal to the empty string. This is then followed by twice the identity function for \backslash which results in the output $\\$. And finally, the identity function corresponding to the last character a generates a as output. If we concatenate all these outputs, then we get that the final output is “\\a”.

Function Types: Note that an important design choice for this algorithm is to use the function types

$\{IDENTITY, CONSTANT\}$ to identify different output characters. The minimal set of function types which can represent all outputs would consist of only $CONSTANT$. This would, however, not allow us to effectively group transitions since each different input character would need a different output function. Therefore, we have chosen to add the type $IDENTITY$ which represents all characters that are not modified. Using many different function types would result in SFTs with more states therefore we limited our number of function types to two for our experiments. It is, however, possible to define other function types. For example, if you want to learn a model of a sanitizer that shifts all letters with an offset of 1, then one could define the function type $OFFSET + 1$.

The black-box algorithm also uses the concept of closedness of the observation table. Let OT be an observation table. Then OT is *closed* if, for all $t \in S \cdot \Sigma$, there exists an $s \in S$ such that all entries in the rows of s and t in the OT are equal (Angluin, 1987).

4.2 The SFT Learning Algorithm

The SFT learning algorithm is described in Algorithm 1. More details on this algorithm, including an example of how the algorithm works and more specification examples, can be found in the Master’s thesis on which this paper is based (Lathouwers, 2018).

4.3 From SOT to Hypothesis Automaton

Line 12 of the SFT learning algorithm calls an algorithm to generate a hypothesis automaton from the SOT. This algorithm is described in Algorithm 2.

Final States: We observe that different sanitizer implementations can handle rejecting an input differently, e.g., returning “null” or returning an empty string. Considering that we are using a black-box algorithm, the user would not know how the program acts upon an unacceptable input. Therefore, as in other research (Botinčan and Babić, 2013; Shahbaz and Groz, 2009), we assume that all states of the SFT are final.

Generating Guards: On line 9 of the hypothesis generation algorithm, a guard generating algorithm is used. This algorithm generates guards, also called input conditions, for all transitions that start in the same state q_s . To generate the guards, it uses a set of evidence and the corresponding outputs. The evidence is the input character upon which a transition is taken to move to a next state in the automaton. The guard generator works as follows:

Algorithm 1: The SFT Learning algorithm.

Data: SUL to which we can pose membership and equivalence queries

Result: SFT that represents the SUL

- 1 SOT = $(S = \{\varepsilon\}, W = \{\varepsilon\}, \Lambda = \emptyset, T = \emptyset, f = \emptyset)$
- 2 Fill the SOT with entries by posing membership queries to the SUL.
- 3 **while** *no equivalent hypothesis automaton has been found* **do**
- 4 **while** *SOT is not closed* **do**
- 5 Find the shortest $t \in \Lambda$ such that for all $s \in S$ it holds that $row(s) \neq row(t)$.
- 6 Let $S = S \cup \{t\}$
- 7 **if** *there is no $b \in \Sigma$ such that $t \cdot b \in S \cup \Lambda$* **then**
- 8 | add $t \cdot b$ to Λ
- 9 **end**
- 10 Fill the missing entries in T and f by posing membership queries.
- 11 **end**
- 12 Create hypothesis automaton from the SOT.
- 13 Pose equivalence query for the generated hypothesis automaton.
- 14 **if** *there exists a counterexample z* **then**
- 15 Let $i_0 \in \{0, 1, \dots, |z| - 1\}$ such that the response of the target machine is different for the strings $s_{i_0}z_{>i_0}$ and $s_{i_0+1}z_{>i_0+1}$.
- 16 Define the distinguishing string d as $z_{>i_0+1}$.
- 17 Let b be an arbitrary character from the input language
- 18 **if** $row(s_{i_0}b) = row(s_j)$ *when d is added to W for some $j \neq i_0 + 1$* **then**
- 19 | Add $s_{i_0}b$ to Λ
- 20 **else**
- 21 | Add d to W .
- 22 **end**
- 23 Update the missing entries in T and f .
- 24 **end**
- 25 **end**
- 26 Return the hypothesis automaton

- If the set of evidence is empty, generate one transition with the guard *True*. The set of output functions of this transition will consist only of the identity function.
- If the set contains one piece of evidence, then one transition will be generated with the guard *True*. The set of output functions will be generated based on the output associated with the evidence. It will generate either the identity function

Algorithm 2: Algorithm that describes how a hypothesis automaton is derived from a closed SOT.

Data: Closed SOT

Result: Hypothesis automaton

- 1 **for** $s \in S$ **do**
- 2 | Create a *final* state q_s
- 3 **end**
- 4 Set the initial state to q_ε , which is the state corresponding to the empty string
- 5 **foreach** q_s **do**
- 6 | Find its one-step extensions in Λ in the rows of the SOT
- 7 **end**
- 8 **foreach** q_s **do**
- 9 $(\phi, \psi, q) = \text{guardGeneratingAlgorithm}(\dots)$
 /* Call the guard generating algorithm with all one-step extensions of q_s */
- 10 Add transition $q_s \xrightarrow{\phi/\psi} q$ to the set of transitions
- 11 **end**

or a constant for each character in the output. The identity function will be generated if the character is the same as the evidence, otherwise a constant with the value of the output character is generated.

- Otherwise, *the set contains multiple pieces of evidence*. Pieces of evidence are grouped together if they lead to the same state. The largest group of evidence is chosen to act as a sink transition. This means that all characters for which no evidence exists will be grouped into this sink transition. For each generated guard, the term generator is called.

Generating Terms: The term generator is a novel addition to the algorithm that generates terms, i.e., output functions, for all transitions. It takes a predicate, i.e., a guard, and the starting state of the transition as an argument (see Algorithm 3).

4.4 Equivalence Oracle

In practice, there is no all-knowing entity that can check whether a hypothesis automaton is equivalent to the specified black-box system because we assume that the user cannot access the implementation of the system. Therefore, an equivalence oracle, as used in step 5 of the learning algorithm, is approximated using membership queries. If no counterexample is found, then we assume that the hypothesis automaton is a correct description of the program's behaviour. Some ways in which an equivalence oracle can be im-

Algorithm 3: Algorithm that describes how terms are generated for a specific guard and state.

Data: State q_s and a guard g
Result: (Set of) guards with corresponding terms.

```

1 for all one-step extensions of  $q_s$  that satisfy
  guard  $g$  do
2   Let  $s$  be the string that represents state  $q_s$ 
3   Let  $s \cdot b$  the string that represents state
   $q_s \cdot b$ .
4   Compute the suffix of the output such that
  it is equal to  $o_{s \cdot b} - o_s$  where  $o_s$  denotes
  the output of the automaton upon
  input  $s$ . /* The suffix
  represents the output that is
  generated for character  $b$ . */
5   Let  $T = \{\}$ 
6   foreach  $c \in \text{suffix}$  do
7     if  $c == b$  then
8       Extend  $T$  with the identity
       function
9     else
10      Extend  $T$  with the constant
      function  $c$ 
11    end
12  end
13 end
14 if  $T$  is the same for all one-step extensions
  then Return  $T$ 
15
  /* There exist two one-step
  extensions,  $q_s \cdot b$  and  $q_s \cdot c$ , of  $q_s$  for
  which the set of term functions
  differ, therefore the predicate
  needs to be split. */
16
17 Split the predicate into two predicates such
  that  $q_s \cdot b$  satisfies only one of the two
  predicates and  $q_s \cdot c$  satisfies only the other
  predicate.
18 foreach generated predicate do
19   termGenerator( $q_s$ , generated predicate)
20 end

```

plemented include the following (which are all implemented in our tool):

- Random testing (Hamlet, 2002): Generate strings of a specified length randomly.
- Random prefix selection (Smeenk, 2012): Take the access string of a randomly chosen state as a prefix and append a suffix of randomly generated characters.

- State coverage (Simao et al., 2009): Generate a set of strings such that each state in the automaton is visited at least once.
- Transition coverage (Simao et al., 2009): Generate a set of strings such that each transition is taken at least once.
- Predicate coverage (Offutt et al., 2003): Generate a set of strings such that each predicate, including sub-predicates, is satisfied at least once.

5 RESULTS

5.1 Validation of SFT Learning Algorithm

To validate our approach, we have tried to learn models of existing real-world sanitizers. The following sanitizers have been chosen to evaluate our approach:

1. Encode (from the he project (Bynens, 2018))
2. Escape (from the cgi python module (Python Software Foundation, 2018))
3. Escape (from the escape-string-regexp project (Sorhus, 2016))
4. Escape (from the escape-goat project (Sorhus, 2017))
5. Unescape (from the escape-goat project (Sorhus, 2017))
6. Unescape (from the postrank-uri project (PostRank Labs, 2017))
7. To Lowercase (from the CyberChef project (GCHQ (Government Communications Headquarters), n.d.))
8. htmlspecialchars (built-in PHP function (The PHP Group, 2018a))
9. filter_sanitise_email (built-in PHP function (The PHP Group, 2018b))
10. Remove tags (from the govalidator project (Saskevich, 2018))

These specific sanitizer implementations have been found by searching on GitHub among all projects for the keywords “escape”, “encode”, and “sanitize”, which are keywords often used to describe sanitizers. The results on GitHub were sorted by “Most stars” after which the top 20 repositories have been chosen. Then, the repositories have been filtered so that only string sanitizers were left which had clear documentation on how they should work, which could be used and for which we could write a specification in

at most 10 minutes. Aside from sanitizers that have been found this way, three other sanitizers have also been tested of which two have been written in PHP and one in Python. These have been added so that we tested sanitizer implementations written in different languages.

An implementation of the algorithm, as well as all tested programs and corresponding specifications, are available at: <https://github.com/Sophietje/SFTLearning>.

The tests have been run on a MacBook Pro, running Mojave 10.14.2 with a 2.3GHz Intel Core i5 (4 cores) and 16 GB of memory available. We have chosen the input alphabet consisting of the Unicode characters represented by the decimals 32 up to 400 (unless stated otherwise). This includes the Basic Latin alphabet, the Latin-1 Supplement, Latin Extended-A and part of Latin Extended-B. We used an equivalence oracle that guarantees predicate coverage; it generates 3000 tests per (sub-)predicate. Also, a time-out was set that interrupts the process if it did not deduce a model within 10 minutes. See Table 2 for the results of the SFT learning algorithm on existing sanitizers.

The learned models have all been compared to a specification of the program to check whether they are correct. No mistakes were found in these implementations.

Most of the sanitizers could be automatically inferred with our SFT learning algorithm within two minutes. Overall, the sanitizers can be divided into two main categories:

- Sanitizers that act based on the occurrence of a *single* character (sanitizers 1, 2, 3, 4, 7, 8, 9).
- Sanitizers that act based on the occurrence of *multiple* characters (sanitizers 5, 6, 10).

From the results we conclude that we can fully automatically learn models of existing sanitizers that act based on occurrences of *single* characters within a couple of minutes. Learning models of sanitizers that act based on the occurrence of multiple characters is not yet feasible with this approach. This is what we expected since the underlying model, SFT, cannot represent these.

To explain why SFTs cannot represent sanitizers that act based on the occurrence of multiple characters, we have a look at an example. Consider a specification for a sanitizer that encodes the character &, if it is not yet encoded, into its HTML reference "&" (see Figure 5). This specification does *not* perfectly model the behaviour of the sanitizers. If a string ends with "&a", "&am" or "&" then it will only output the encoding of & and ignore the characters after this. We need to recognize the end of the input to be able to

Table 2: Results of the SFT learning algorithm on existing sanitizers. *a* means that the model has been correctly derived for the Basic Latin alphabet. *b* means that we are only able to learn this model when acting as an equivalence oracle ourselves. When automatically learning this automaton, the counterexamples are not minimal, thus resulting in a timeout because the automaton becomes too complex.

Sanitizer	Total running time(s)	Time spent in mem. oracle (ms)	Time spent in eq. oracle (ms)	# mem. queries	# eq. queries	# states learned	# states specified	# transitions learned	# transitions specified	Can be learned?
1. Encode (he)	7	3919	4870	129733	336	3	1	21	7	Yes ^a
2. Escape (cgi)	4	1971	2635	88442	241	3	1	12	4	Yes ^a
3. Escape (escape-string+regexp)	75	18945	27137	680560	1328	3	1	14	2	Yes
4. Escape (escape-goat)	40	13882	21138	536243	1121	5	1	30	6	Yes
5. Unescape (escape-goat)	-	-	-	-	-	14	-	22	-	No
6. Unescape (postmark-urt)	-	-	-	-	-	7	-	40	-	No
7. To Lowercase (CyberChef)	23	8087	9079	192865	919	2	1	54	27	Yes ^a
8. htmlspecialchars (php)	20	8581	16392	484277	316	4	1	20	5	Yes
9. filter_var (php)	-	-	-	-	-	1	-	12	-	No ^b
10. Remove tags (gobalidator)	-	-	-	-	-	8	-	18	-	No

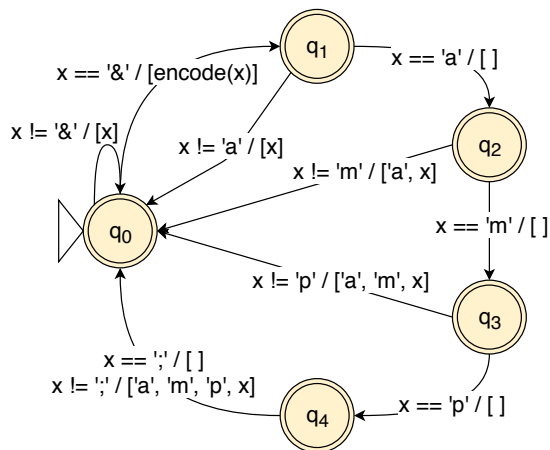


Figure 5: An SFT which encodes & into its HTML reference "&"; unless it is part of an encoded &. Note that this SFT does *not* precisely model the sanitizer.

solve this problem. This can be modelled by adding ϵ -transitions to the automaton, which are currently not inferred. This would, however, result in an automaton that is not single-valued. Therefore, we would be unable to check whether the automaton is equivalent to another automaton which is necessary to compare it to specifications. Another solution would be to extend this algorithm to deduce SFTs with lookback, lookahead or registers, which is an extension we leave for future work.

5.2 Finding Bugs in Sanitizers

We also wanted to evaluate whether our method can be used to automatically find errors in the implementations of sanitizers (since no bugs were found in previous experiments). To do this, we have asked an individual with a security and programming background to implement the sanitizers that we were able to automatically learn from the previous section. He provided us with two implementations for each sanitizer, one that was intended to be correct, and one in which he (might have) introduced mistakes on purpose. He wrote down which errors have been introduced such that we could, after learning and (possibly) identifying errors, check whether we found all mistakes.

We learned models from his provided implementations¹ using our SFT learning algorithm. This learning process has been done with the same settings, namely input alphabet, equivalence oracle implementation, and time-out, as were used for the previous experiments. Then, we checked for any mistakes in these implementations by comparing the learned

¹Available at <https://github.com/Sophietje/SFTLearning/tree/master/Sanitizers/Implementations>

models to the specification that we had already written for the previous experiment. If these were not equal then a counterexample was produced for which the two models behave differently. Such a counterexample gives an idea of what the error in the program is, which can then be used to perform a detailed manual inspection of the learned model to identify the problem.

We identified the following errors in the implementations that were supposed to be correct:

- *Encode (he)*: Wrong first output function for several characters and incorrect encodings of all characters.
- *Escape (Escape-string-regexp)*: Two characters (\backslash , $\$$) should have been escaped and one character ($-$) should not have been escaped.
- *Escape (escape-goat)*: Wrong order in applying the encodings.
- *htmlspecialchars (php)*: Character encoding of $''$ misses the character $;$.

Next, we tried to find errors in the programs in which mistakes were introduced on purpose. The following errors were found:

- *Encode (he)*: Wrong first output function for several characters and a wrong encoding of $\&$ and $<$.
- *Escape (cgi)*: Double substitution of $\&$.
- *Escape (escape-string-regexp)*: Two characters (\backslash , $\$$) should have been escaped and one character ($-$) should not have been escaped.
- *Escape (escape-goat)*: Wrong order in applying encodings.
- *htmlspecialchars (php)*: Two single quotes are wrongly encoded instead of encoding a double quote.

We compared the errors found with the list describing the intended errors. We correctly identified the cause of all mistakes.

There were two cases in which the error that we identified was not completely accurate. Firstly, in the case of *escape (cgi)*, we identified that the character $\&$ was substituted twice. In the actual implementation the characters $<$ and $>$ were also substituted twice. However, these second substitutions do not change the string because the first substitution already removed all $<$ and $>$ characters. So, while the double substitution of the $\&$ character results in incorrect output, which we identified correctly, the double substitution of $<$ and $>$ does not result in incorrect output. Secondly, the model could not accurately represent the error that was introduced in *htmlspecialchars* because the sanitizer's behaviour for the character $''$

depended on the character that followed. Therefore, although we misinterpreted the error, we were still able to identify the cause of the problem.

6 DISCUSSION

As shown previously, our approach for reasoning about the correctness of sanitizers can effectively be used to find errors in the implementations of sanitizers. There are, however, a number of limitations one should be aware of.

For instance, no black-box learning algorithm has access to an exact equivalence oracle, i.e., we cannot precisely determine equivalence between the implementation and the model. Such an equivalence oracle is therefore simulated by testing a large number of test cases. If all test cases succeed, we assume that the model correctly represents the implementation. However, if the number of test cases is too small, then the model will not accurately represent the sanitizer. As a result, any analysis done on such a model may also not accurately reflect the behaviour of the sanitizer. We have shown that we can deduce correct models when using enough test cases. Users of the tool should, however, be aware that if it is used to analyse more complex sanitizers, then a larger number of test cases is likely required to deduce correct models.

Also, when learning a sanitizer, the user needs to specify which input alphabet (s)he considers. This means that if an error occurs outside of the specified alphabet, then this cannot be found using our method. Fortunately, our approach can handle large input alphabets.

Additionally, the user is asked to write a specification that accurately describes the sanitizer's behaviour. If the user makes any mistakes in this specification, then the corresponding errors in the sanitizer may not be uncovered. Moreover, writing such a specification can take a lot of time and may not be feasible for larger models. It is, however, also possible for the user to inspect a graphical representation of the learned model to find errors. We leave it as future work to minimise these graphical representations.

Finally, we note that the proposed SFT learning algorithm cannot accurately represent all sanitizers. Specifically, it is unable to precisely represent sanitizers whose behaviour depends on multiple characters. Thus, if our approach is used to reason about sanitizers whose behaviour depends on multiple characters, then the results will be inaccurate. One could reason about such sanitizers by extending the current algorithm to SFTs with lookahead.

7 RELATED WORK

Black-box automata learning was first introduced by Angluin with the L* algorithm (Angluin, 1987). L* derives deterministic finite automata using equivalence and membership queries. A similar approach, using such queries, has been developed for many other types of automata such as: Mealy machines (Shahbaz and Groz, 2009), register automata (Cassel et al., 2014), and SFAs (Argyros et al., 2016; Drews and D'Antoni, 2017). We extend this list by developing a learning algorithm for SFTs. Moreover, automata learning has shown to be a valuable technique to derive models from large real-world systems by several *case studies* (Smeenk et al., 2015; Bohlin and Jonsson, 2008).

Automata learning has also been used to detect vulnerabilities in TLS implementations (De Ruiter and Poll, 2015). Similar to how we reason about input-output behaviour, De Ruiter and Poll reason about TLS implementations using messages between a client and server. They use Mealy machines to represent the implementations which use one transition per input per state. As the input and output alphabet they use an abstraction of the possible messages, which amounts to 12 messages for servers and 13 messages for clients. After they have inferred a Mealy machine, they minimise the representation by combining similar transitions and then the model is inspected manually to find errors. In the case of sanitizers, we are interested in much larger alphabets; for our experiments we reasoned about ± 370 possible input characters. Due to the size of the input alphabet, Mealy machines are not an ideal representation because the automata would be very large. Therefore, rather than minimising afterwards, we try to learn a symbolic finite transducer immediately. And while manual inspection of the model is possible, we advocate writing specifications which can be reused and automatically checked.

This work is an extension of the work by Argyros et al. (Argyros et al., 2016) who presented a black-box learning algorithm that infers SFAs from sanitizers. Unfortunately, SFAs can only describe the input or output language and not the relation between the input and the output language. Argyros et al. mention that their SFA learning algorithm can be adapted to learn SFTs. In our research, we have developed and implemented such an algorithm for deducing SFTs which allows us to reason about the correctness of transformations between the input and the output language. We allow the user to check types of specifications such as blacklisting or length whereas Argyros et al. only allow the checks that are provided

for BEK (Hooimeijer et al., 2011) programs such as equivalence, idempotency and commutativity, which are included in our tool as well.

BEK (Hooimeijer et al., 2011) is a language that can be used to develop sanitizers and analyse their correctness. However, this cannot be used to reason about the correctness of *existing* sanitizers without re-implementing them.

Botinčan and Babić (Botinčan and Babić, 2013) present a technique Sigma* that learns symbolic look-back transducers from programs. This model can represent more sanitizers than the SFTs that we use. However, they use a white-box learning technique, meaning that they need access to the source code whereas we only need to be able to observe the input and output of the program. Extending the algorithm that we present in this paper to symbolic lookback transducers that Botinčan and Babić use is a topic for future work.

There exist several other methods to reason about sanitizers' correctness most of which focus on detecting vulnerabilities (Balzarotti et al., 2008; Mohammadi et al., 2015; Shar and Tan, 2012). Our approach can be used to detect vulnerabilities similar to these methods. However, we are also able to reason about their input-output behaviour in terms of, e.g. idempotency and commutativity.

Aside from correct implementation of sanitizers, the *placement of sanitizers* also influences the correctness of an application. If sanitizers are not placed correctly then applications may still be vulnerable. Several researchers have therefore focused on either repairing the placement of sanitizers, or automatically placing sanitizers (Saxena et al., 2011; Welearegai and Hammer, 2017; Yu et al., 2011). These approaches are considered complementary research to the ideas discussed in this paper.

Aside from sanitization, there are also *sanitization-free defences*. For example, Scholte et al. (Scholte et al., 2012) show that automatically validating input can be a good alternative to output sanitization for preventing XSS and SQL injection vulnerabilities. Similarly, Costa et al. (Costa et al., 2007) have presented the tool Bouncer which prevents exploitation of software by generating input filters that drop dangerous inputs.

8 CONCLUSION AND FUTURE WORK

To conclude, we have presented a new approach to reason about the correctness of sanitizers. First of all, we developed a new learning algorithm, which

uses equivalence and membership queries, to automatically derive SFTs of existing sanitizers. This automaton describes how the sanitizer transforms an input into its corresponding output. Then, we wrote a specification of the sanitizer, in the form of an SFA or SFT. This specification is compared to the learned model of the sanitizer in order to find any discrepancies between the models. With a case study, we have shown that we can use our approach to automatically reason about real-world existing sanitizers within a few minutes.

As future research, we think that extending the learning algorithm to support epsilon transitions and SFTs with lookahead, lookback or registers is most important. This would allow us to reason about more complex sanitizers. One can also look into improving the user experience of the approach by letting users write the specifications in ways that are more familiar to them such that they do not need to understand how SFTs work. Another option is to present users with a minimised graphical representation of the learned models for manual correctness inspection.

REFERENCES

- Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106.
- Argyros, G., Stais, I., Kiayias, A., and Keromytis, A. D. (2016). Back in black: towards formal, black box analysis of sanitizers and filters. In *2016 IEEE Symposium on Security and Privacy*, pages 91–109. IEEE.
- Balzarotti, D., Cova, M., Felmetzger, V., Jovanovic, N., Kirida, E., Kruegel, C., and Vigna, G. (2008). Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy*, pages 387–401. IEEE.
- Bjørner, N. and Veanes, M. (2011). Symbolic transducers. Technical Report MSR-TR-2011-3, Microsoft Research.
- Bohlin, T. and Jonsson, B. (2008). Regular inference for communication protocol entities. Technical report, Technical Report 2008-024, Uppsala University, Computer Systems.
- Botinčan, M. and Babić, D. (2013). Sigma*: symbolic learning of input-output specifications. In *ACM SIGPLAN Notices*, volume 48, pages 443–456. ACM.
- Bynens, M. (2018). he. <https://github.com/mathiasbynens/he>. Accessed on: 19-12-2019.
- Cassel, S., Howar, F., Jonsson, B., and Steffen, B. (2014). Learning extended finite state machines. In *International Conference on Software Engineering and Formal Methods*, pages 250–264. Springer.
- Costa, M., Castro, M., Zhou, L., Zhang, L., and Peinado, M. (2007). Bouncer: Securing software by blocking bad

- input. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 117–130. ACM.
- De Ruiter, J. and Poll, E. (2015). Protocol state fuzzing of tls implementations. In *Proceedings of the 24th USENIX Security Symposium*, pages 193–206. USENIX Association.
- Drews, S. and D’Antoni, L. (2017). Learning symbolic automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–189. Springer.
- GCHQ (Government Communications Headquarters) (n.d.). Cyberchef. <https://github.com/gchq/CyberChef>. Accessed on: 19-12-2019.
- Hamlet, R. (2002). Random testing. *Encyclopedia of software Engineering*.
- Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., and Veanes, M. (2011). Fast and precise sanitizer analysis with BEK. In *Proceedings of the 20th USENIX Security Symposium*. USENIX Association.
- Lathouwers, S. (2018). Reasoning about the correctness of sanitizers. Master’s thesis, University of Twente, Enschede, the Netherlands.
- Mohammadi, M., Chu, B., and Ritcher Lipford, H. (2015). POSTER: Using unit testing to detect sanitization flaws. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1659–1661. ACM.
- Northwave (n.d.). <https://northwave-security.com/>. Accessed on 19-12-2019.
- Offutt, J., Liu, S., Abdurazik, A., and Ammann, P. (2003). Generating test data from state-based specifications. *Software testing, verification and reliability*, 13(1):25–53.
- OWASP Foundation (2017). Owasp top 10 application security risks - 2017. https://www.owasp.org/index.php/Top_10-2017_Top_10. Accessed on 19-12-2019.
- PostRank Labs (2017). PostRank URI. <https://github.com/posrank-labs/posrank-uri>. Accessed on: 19-12-2019.
- Python Software Foundation (2018). 20.2. cgi — common gateway interface support. <https://docs.python.org/2/library/cgi.html>. Accessed on: 19-12-2019.
- Python Software Foundation (n.d.). 20.1. html — HyperText Markup Language support. <https://docs.python.org/3/library/html.html#html.escape>. Accessed on 19-12-2019.
- Saskevich, A. (2018). govalidator. <https://github.com/asaskevich/govalidator/>. Accessed on: 19-12-2019.
- Saxena, P., Molnar, D., and Livshits, B. (2011). SCRIPT-GARD: automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 601–614. ACM.
- Scholte, T., Robertson, W., Balzarotti, D., and Kirda, E. (2012). Preventing input validation vulnerabilities in web applications through automated type analysis. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pages 233–243. IEEE.
- Shahbaz, M. and Groz, R. (2009). Inferring mealy machines. In *International Symposium on Formal Methods*, pages 207–222. Springer.
- Shar, L. K. and Tan, H. B. K. (2012). Mining input sanitization patterns for predicting sql injection and cross site scripting vulnerabilities. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1293–1296. IEEE Press.
- Simao, A., Petrenko, A., and Maldonado, J. C. (2009). Comparing finite state machine test coverage criteria. *IET software*, 3(2):91–105.
- Smeenk, W. (2012). Applying automata learning to complex industrial software. *Master’s thesis, Radboud University Nijmegen*.
- Smeenk, W., Moerman, J., Vaandrager, F., and Jansen, D. N. (2015). Applying automata learning to embedded control software. In *International Conference on Formal Engineering Methods*, pages 67–83. Springer.
- Sorhus, S. (2016). escape-string-regexp. <https://github.com/sindresorhus/escape-string-regexp>. Accessed on: 19-12-2018.
- Sorhus, S. (2017). escape-goat. <https://github.com/sindresorhus/escape-goat>. Accessed on: 19-12-2019.
- The PHP Group (2018a). htmlspecialchars. <http://php.net/manual/en/function htmlspecialchars.php>. Accessed on: 19-12-2019.
- The PHP Group (2018b). Sanitize filters. <http://php.net/manual/en/filter.filters.sanitize.php>. Accessed on: 19-12-2019.
- Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., and Bjorner, N. (2012). Symbolic finite state transducers: Algorithms and applications. In *ACM SIGPLAN Notices*, volume 47, pages 137–150. ACM.
- Welearegai, G. B. and Hammer, C. (2017). Idea: Optimized automatic sanitizer placement. In *International Symposium on Engineering Secure Software and Systems*, pages 87–96. Springer.
- Yu, F., Alkhalaf, M., and Bultan, T. (2011). Patching vulnerabilities with sanitization synthesis. In *Proceedings of the 33rd International Conference on software engineering*, pages 251–260. ACM.