UNIVERSITY OF TWENTE

MASTER'S THESIS

Reasoning About the Correctness of Sanitizers

Author:
Sophie LATHOUWERS

Supervisors:
Prof. Dr. Marieke HUISMAN
Dr. Maarten EVERTS
Martijn HOOGESTEGER, MSc
(Northwave)

September 3, 2018

Abstract

Nowadays, most applications are developed to be deployed to the web. These web applications can be accessed by many users, including malicious users who try to attack them. In order to defend against attacks such as SQL injections and XSS, many web applications use sanitizers. Sanitizers will modify a given input so as to remove the presence of dangerous characters. It is, however, very difficult to write correct sanitizers. And reasoning about the correctness of any program tends to be very difficult, especially if the program is quite large. Fortunately, sanitizers are generally small pieces of code due to which it is possible to reason about their correctness. In this research, we present a method to reason about the correctness of sanitizer implementations by comparing them to specifications. This method learns models, symbolic finite transducers, in a black-box manner, which describe the behaviour of the sanitizers. These models are compared to specified models in order to find erroneous behaviour of sanitizer implementations. The learning algorithm and ability to compare models have all been implemented in a tool called SFTLearning. Using SFTLearning, we were able to derive models from real-world sanitizers and reason about their correctness.

Contents

List of Abbreviations

DFA	**D**eterministic **F**inite **A**utomaton
FST	**F**inite **S**tate **T**ransducer
OT	**O**bservation **T**able
SFA	**S**ymbolic **F**inite **A**utomaton
SFT	**S**ymbolic **F**inite **T**ransducer
SOT	**S**ymbolic **O**bservation **T**able
SQL	**S**tructured **Q**uery **L**anguage
SRS	**S**tring **R**ewriting **S**ystem
SUL	**S**ystem **U**nder **L**earning
XSS	**C**ross-**s**ite **S**cripting

Chapter 1

Vulnerabilities

1.1 Introduction

Nowadays more and more daily tasks are completed through the use of digital systems. People rely heavily on the digital systems due to which these systems become targets for criminals. It is therefore important that these systems are secure. For example, in online banking only you and authorized bank employees should be able to access your records. The systems that we use are becoming increasingly complex and therefore writing correct and secure programs becomes more difficult.

The term *flaw* is used to denote problems in a system. When a flaw can be exploited by a user, then we call it a *vulnerability*. Exploiting means that the user is able to do something which (s)he should not be able to by abusing a system's flaw. There can be many types of vulnerabilities in a system. As of 2017, the most serious web application security risk is injection flaws [1]. Such an injection flaw occurs when untrusted data is interpreted. As a result, a user-given command may be executed or malicious data may be injected into the system.

The following example of SQL injection illustrates the danger of such vulnerabilities. Consider the database table called "customers" which contains all personal information of the customers of a web application which can be seen in Table 1.1. Assume that the web application uses the following SQL query to select a customer's data where *user_email* and *user_password* are given by the user, and added to the query using string concatenation:

```
1 "SELECT name, address
  FROM customers
3 WHERE email='" + user_email + "' AND password='" +
    user_password+ "';"
```

If a non-malicious user inputs his email and password, then this query will simply return his name and address as it should. However, consider what a malicious user might give as input for these variables. For example, if a user inputs the following:
user_email = abc@google.com
user_password = 12345' OR '1'='1

customer_id	name	address	email	password
1	Andrew Smith	Main Road 123	a.smith@google.com	9e209040...
2	Olivia Jones	High Street 81	o.jones@google.com	d1d3ec2e...
...	...	...	...	...

TABLE 1.1: Database representing customers for a web application

Then the query will look as follows:

```
" SELECT name, address
FROM customers
WHERE email='abc@google.com' AND password='12345' OR '1'='1';
"
```

Since the constraint "1=1" is always true, this query will return the name and address of *all customers*. As a result, the malicious user now has the personal information of all the users of the web application.

This is only one example of an injection vulnerability, namely SQL injection, but there are many more such as Cross-site scripting (XSS), code injection, command injection, XPATH injection, etcetera. These attacks can potentially lead to the disclosure of sensitive information, modification of data or even deletion of data. Therefore, it is important to find ways to protect against such vulnerabilities.

On the 25th of May in 2018, the European Union introduced the General Data Protection Regulation (GDPR). This law requires data processors to use appropriate defences to ensure data integrity and security. It is therefore import to develop tools that can (mostly) automatically give us insight into the correctness of programs, discover vulnerabilities and protect against vulnerabilities.

Depending on the kind of application and the associated security risks, one can use different approaches to find the vulnerabilities. First of all, one can try to find errors when the application is being built. Then one tries to find the problems before they can be exploited, e.g. by using static code analysis. Secondly, one can try to detect when vulnerabilities can be exploited by users and take action when it is being exploited. For example, if you detect an ongoing attack then start blocking connections originating from the offender's IP address. Finally, one can detect vulnerabilities by examining logs and taking action when the application has already been exploited. This research focuses on the first kind; detecting errors in applications when they have not yet been exploited. If these errors are eliminated, then the number of attack possibilites is reduced.

1.2 Preventing exploitation of injection vulnerabilities

Injection vulnerabilities occur when attackers are able to "relay malicious code through an application to another system" [2]. In other words this means that to exploit injection vulnerabilities, attackers need to be able to give some input to the system. There are several approaches that one can use to eliminate the exploitation of an injection vulnerability.

First of all, one can use detection mechanisms to find vulnerabilities and then fix these vulnerabilities. As a result, since the vulnerability no longer exists, it can also no longer be exploited. The detection of this vulnerability can happen either before or after deployment of the application. It is, however, very difficult to detect all vulnerabilities. An example of a vulnerability detection mechanism is fuzzing. Fuzzing is a testing technique that generates input for a computer program. The generated input is then given to the computer after which the program is monitored to detect any unexpected behaviour. It is important to note that fuzzing can never prove the correctness of a program. After all, there are infinitely many inputs thus it is always possible that the program would not work as expected when it is given a yet untried input. You will need to try many different inputs to get the confidence that a program works correctly. Thus, fuzzing a program takes quite some time. The

advantage of using fuzzing as a detection mechanism is that it is a fully automatic approach.

Secondly, one can give the application only the privileges that are necessary to complete the required tasks. For example, one should not run commands as root. This can, however, not prevent all exploitations of vulnerabilities. For example, if an application can insert data, then the user may be able to insert malicious data. In this case, the user only needs the same priviliges as the application.

Thirdly, one can filter the input that is given to the system. When filtering input, the user can use either whitelisting or blacklisting. *Whitelisting* limits what input is considered acceptable by comparing all inputs to a list of acceptable inputs. If the given input in on the list, then the input is accepted. *Blacklisting* limits what input is considered acceptable by comparing all inputs to a list of unacceptable inputs. If the given input is on the list, then the input is rejected. Adapting a whitelisting or blacklisting approach is quite difficult as the danger is that the user forgets to specify a good or dangerous input respectively. If a good input was not specified, then non-malicious users are likely hindered. For example, some websites do not accept + in an email address even though it is allowed according to the specification [3]. If a bad input was not specified, then a malicious user can misuse this to trigger a vulnerability thus the system is no longer secure.

Dangerous inputs could also be accepted. However, then the developers need to be aware that any user input can be malicious. This likely leads to the need of more validation steps throughout the code base. Also, it is then crucial that no validation step is missed as this may introduce a vulnerability. Moreover, in general it is easier to prevent dangerous input from entering your system rather than allowing it inside and making sure it cannot abuse anything.

Aside from filtering, accepting bad inputs and adapting the code, one can also choose to modify the given input. By modifying the input, you can eliminate any possible dangerous sequences of characters within the given input. Modification of input to eliminate dangerous (sequences of) characters is what *sanitizers* do.

This research focuses on sanitizers, because they are widely used in practice. Moreover, since sanitizers are typically small pieces of code, it allows us to reason about the program without running into problems that occur when reasoning about large systems. For example, specifying a complete banking application is very difficult whereas specifying one method can be done quite easily. Also, sanitizers are often reused since writing them is very difficult. As a result, if we reason about one instance of a sanitizer, then these results will also apply to the other instances. The results can only be reused if the analysis is dependent solely on the sanitizer code, and if the sanitizer's code has not been modified.

1.2.1 Research Questions

It is important to be able to reason about the correctness of sanitizers as this may be able to prove the absence, or presence, of vulnerabilities. Writing sanitizers is prone to errors therefore people are often encouraged to reuse existing sanitizers. Therefore this research will focus on verifying existing sanitizers. In this research, we focus on the following research question:

- **RQ 1:** How can we reason about the correctness of existing sanitizers?
- **RQ 2:** Can the approach, which was found in RQ 1, be fully automatised such that it can be used in someone's workflow?

To answer the first research question, we will focus on the following sub-questions:

- **RQ 1.1:** What would users want to specify when reasoning about the security of sanitizers?
- **RQ 1.2:** What can already be proven about sanitizers using previous research?
- **RQ 1.3:** How can the specifications, found in RQ 1.1, be proven?

These research questions led us to develop a new algorithm to learn a model, symbolic finite transducers, from a program and compare this to specifications. We have implemented this in SFTLearning which can derive two types of models, Symbolic Finite Automata or a Symbolic Finite Transducer, from a program.

1.2.2 Overview of Thesis

Chapter 2 introduces the methodology of this research. Next, Chapter 3 explains sanitizers and their challenges. Chapter 4 presents related work and its relation to this research. Then, Chapter 5 discusses models that can be used to reason about existing sanitizers. Next, Chapter 6 discusses what one would want to specify about sanitizers and how these specifications can be proven. Chapter 7 explains how we can deduce models from existing sanitizers. Then, Chapter 8 presents details on how the approach has been implemented. In Chapter 9, the results of the performance tests and case study are presented. Next, the results are discussed in Chapter 10. Finally, the research is concluded in Chapter 11.

Chapter 2

Methodology

2.1 Reasoning about sanitizers

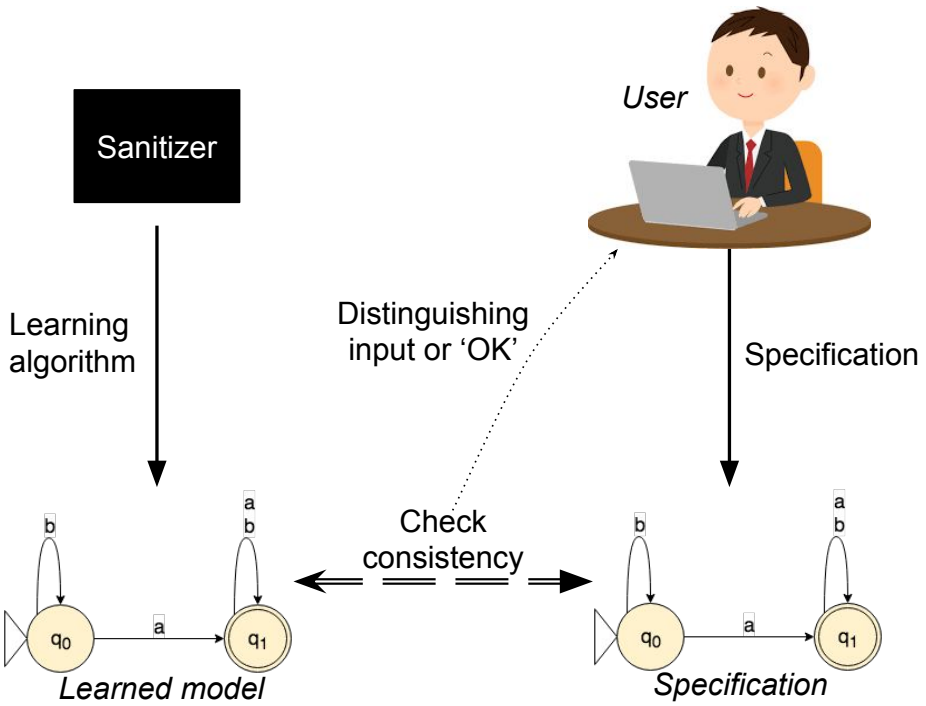


FIGURE 2.1: Overview of the approach that is used to reason about sanitizers

Figure 2.1 gives an overview of our approach. Firstly, we develop a tool that will derive an SFT from a sanitization function. To derive this SFT, we will use a symbolic learning algorithm which is a black-box approach. Secondly, the user needs to write a specification of the sanitization function (see Section 6.3). Finally, we can compute whether the learned model conforms to the specification. If they are not consistent, we plan to present an input upon which the two models act differently. This allows the user to find a discrepancy in the specification or in the existing sanitizer. If there is a discrepancy in the specification, then the user can modify this specification and check for consistency again.

Contributions. In summary, the contributions of this research are:

- A black-box SFT learning algorithm based on equivalence and membership oracles.

- An implementation of the SFA learning algorithm described in [4] in Java.
- An implementation of the black-box SFT learning algorithm in Java.
- A new approach to study the correctness of sanitizers by writing specifications as SFAs and SFTs and comparing these to a learned model.
- Evaluation of the performance and applicability of this approach.
- A case study that demonstrates how this approach can be used to reason about the correctness of sanitizers.

2.2 Evaluation

To evaluate the approach, we want to answer the following questions:

- What programs can, and what programs cannot, be learned using the learning algorithm? This question aims to uncover the limitations of the algorithm which was developed as an answer to the first research question.
- Is this approach feasible to use in someone's workflow? The tool is attractive to users if it can reason about the correctness of string manipulating programs within a couple of minutes. Therefore we want to know whether this approach, which includes learning a model and comparing it to the specification, takes *at most 5 minutes*. Note that it does *not* include the time needed to write the specification(s). This will provide the answer to the second research question.

2.2.1 Learning algorithm

To test the limitations of the SFTLearning algorithm, we will collect a sample of sanitizer functions and specify corresponding models. SFTLearning is used to derive a model from the existing sanitizer. This learned model is then compared to the specification to check whether they are consistent.

During this process we will measure the following:

- Time needed to derive an SFA/SFT from an existing sanitizer
- Time needed to compare model to specification
- Number of states of learned model
- Number of states of specification
- Number of transitions of learned model
- Number of transitions of specification

Note that we measure the number of states from a model as this influences how long the algorithm takes to find a model. Moreover, it allows us to take into account memory usage.

We do not measure the time it takes to write a specification as this is highly dependent on the individual who writes the specifications.

2.2.2 Usability

Whether SFTLearning is feasible to use in someone's workflow, will be assessed based on the following:

- Can we derive models from systems completely automatically within 5 minutes?
- How does the SFTLearning algorithm scale when using larger alphabets?

The first question can be answered using the results from the previous experiments. To answer the second question, we will learn one specific model multiple times. Each time the alphabet from which counterexamples are generated becomes larger.

2.3 Case study on correctness of sanitizers

To test the correctness of sanitizers, we will write specifications for several existing sanitizers. This specification will then be compared to learned models of the sanitizers. When writing the specification, we make sure that each type of specification is used at least once. When evaluating the correctness of sanitizers, we will take the following into account:

- Is the function idempotent?
- If there are sanitizers that should behave the same, is their behaviour the same upon all inputs, i.e. are the sanitizers equivalent?
- Do the tested sanitizers commute with each other?
- Are there any discrepancies compared to the user's specification of the sanitizer?

The first three questions can indicate whether a sanitizer is secure or not. For example, a sanitizer that is not idempotent may be vulnerable to double encoding attacks [5]. If there is an input upon which two sanitizers, which should act the same, behave differently, then this indicates that at least one of the two is not correct. Moreover, sanitizers are preferably commutative because this means that the sanitizer will always behave the same, independently of the order in which it is used. The last question allows us to further investigate the correctness of the sanitizer according to its specification.

Chapter 3

Sanitization

This chapter introduces sanitizers and background information concerning the behaviour of sanitizers.

3.1 Sanitizers

The term *sanitizers* is used to refer to code that sanitizes some piece of input. This input may be given by the user of the system but can also originate from somewhere else such as a database. *Sanitizing* is the act of adapting a string so as to remove or change unwanted characters. This can be done through the removal of characters, escaping characters or by encoding it, e.g. replacing "<" by its HTML entity encoding "<". The characters that we wish to change or remove are those that can be used to exploit a vulnerability. Sanitizers are sometimes also referred to as sanitization functions or sanitization procedures. No clear distinction in the different terms has been found in the literature. We will define the different terms as follows:

- *Sanitization function* refers to a single function that performs sanitization.
- *Sanitization procedure* refers to a complete procedure, which may consist of multiple sanitization functions as well as other steps, such as retrieving the input, that are used to sanitize a piece of text.
- *Sanitizers* is used to refer to the set of one or more sanitization functions used to sanitize a string. Unlike a sanitization procedure, this may not include steps other than sanitization functions.

Sanitization functions are functions that receive one or more string input parameters. These functions will then (possibly) adapt these input parameters and finally return the adapted result. Most sanitization functions need only one string input parameter which is then sanitized. Assuming that sanitization functions are correct and safe, their behaviour should only depend on their input. It is, however, important to note that how an input should be sanitized is strongly related to the context in which it will be used.

For each sanitizer, you will need to specify what you want the sanitizer to do. This behaviour is captured in a *policy*. The policy contains rules which describe how the sanitizer should behave when encountering a specified string or character. You can write rules that state what input is allowed (whitelisting), what input is not allowed (blacklisting), and how specific input should be modified. Note that not all policies are clearly specified. Some policies are implied through the sanitization code or documentation.

3.1.1 Examples

We will now discuss some examples of sanitization functions to give the reader better insight into what sanitization functions are. The first sanitizer's policy is to remove all characters from the given string that are not letters, digits, or special characters (!#$%&'*+-=?^_`{\}~@.[]) [6].

An example of this is the `FILTER_SANITIZE_EMAIL` sanitizer in PHP.

```
1 $sanitized_email = filter_var($users_email,
    FILTER_SANITIZE_EMAIL);
```

LISTING 3.1: Example of applying an email sanitizer filter in PHP

In Listing 3.1 you can see how the `FILTER_SANITIZE_EMAIL` can be applied to some variable. `filter_var` is a function that applies the given filter to the given variable. In this case, the filter `FILTER_SANITIZE_EMAIL` is applied to the variable `users_email`. This function will return the sanitized version of the string that was given.

```
1 void php_filter_email(PHP_INPUT_FILTER_PARAM_DECL)
  {
3         /* Check section 6 of rfc 822 http://www.faqs.org/
              rfcs/rfc822.html */
          const unsigned char allowed_list[] = LOWALPHA HIALPHA
               DIGIT "!#$%&'*+-=?^_`{|}~@.[]";
5         filter_map      map;

7         filter_map_init(&map);
          filter_map_update(&map, 1, allowed_list);
9         filter_map_apply(value, &map);
  }
```

LISTING 3.2: Code that is called when applying the email sanitizer filter in PHP

In Listing 3.2 you can see how the sanitization actually works. First, a list is defined which contains all allowed characters, which in this case consists of all letters, digits and some special characters. Then a map is initialized which contains which characters are allowed. Finally, in the call to `filter_map_apply`, we copy each character in the given string `value` if it is in the map with all allowed characters. This sanitizer would transform "(foo;@bar.com)" into "foo@bar.com" since ";" and brackets are disallowed characters.

Secondly, we consider an HTML sanitizer which allows you to specify a policy which dictates what is allowed in the HTML.

An example of an HTML sanitizer can be seen in Listing 3.3

```
  private static String sanitize(@Nullable String html) {
2     StringBuilder sb = new StringBuilder();
      HtmlStreamRenderer renderer = HtmlStreamRenderer.create(
4         sb,
          new Handler<String>() {
6           public void handle(String errorMessage) {
              fail(errorMessage);
8           }
          });
10
      HtmlSanitizer.Policy policy = new HtmlPolicyBuilder()
```

```
12          // Allow these tags.
        .allowElements(
14              "a", "b", "br", "div", "i", "iframe", "img", "
                    input", "li",
                "ol", "p", "span", "ul", "noscript", "noframes", "
                    noembed", "noxss")
16      // And these attributes.
        .allowAttributes(
18              "dir", "checked", "class", "href", "id", "target",
                    "title", "type")
        .globally()
20      // Cleanup IDs and CLASSes and prefix them with p- to
            move to a separate
        // name-space.
22      .allowAttributes("id", "class")
        .matching(
24          new AttributePolicy() {
            public String apply(
26              String elementName, String attributeName,
                    String value) {
                return value.replaceAll("(?:^|\\s)([a-zA-Z])",
                    " p-$1")
28                  .replaceAll("\\s+", " ")
                    .trim();
30          }
            })
32      .globally()
        .allowStyling()
34      // Don't throw out useless <img> and <input> elements
            to ease debugging.
        .allowWithoutAttributes("img", "input")
36      .build(renderer);

38  HtmlSanitizer.sanitize(html, policy);

40  return sb.toString();
}
```

LISTING 3.3: Example of a HTML Sanitizer from the OWASP Java HTML Sanitizer project[7]

This sanitizer would transform the input "<iframe><script>alert(Hi!)</script></iframe>" into "<iframe></iframe> since script tags are not allowed in this policy.

Thirdly, we consider a sanitizer whose policy is to encode the characters "<", ">" and "&" into their corresponding HTML reference. It also encodes quotes if the optional flag is set to true. For example, the method `escape` in Python module `html` [8] (See Listing 3.4).

```
1 def escape(s, quote=True):
    """
3   Replace special characters "&", "<" and ">" to HTML-safe
        sequences.
```

```
    If the optional flag quote is true (the default), the
       quotation mark
5   characters, both double quote (") and single quote (')
       characters are also
    translated.
7   """
    s = s.replace("&", "&amp;") # Must be done first!
9   s = s.replace("<", "&lt;")
    s = s.replace(">", "&gt;")
11  if quote:
        s = s.replace('"', "&quot;")
13      s = s.replace('\'', "&#x27;")
return s
```

LISTING 3.4: A sanitizer that encodes several special characters to make a string HTML-safe.

3.2 Challenges of sanitization

Sanitizers are very critical pieces of code as they can prevent exploitation of certain vulnerabilities if used correctly. However, to use sanitizers correctly, there are a few things that one should keep in mind which will be discussed in this section.

3.2.1 Placement of sanitizers

The first thing to keep in mind is that sanitizers will need to be placed correctly within the code. If some piece of code is not sanitized, then the sanitization will not be able to achieve the desired effect.

There has been some research into detecting (in)correct placement of sanitizers. First of all, a system ScriptGard has been proposed by Saxena et al. [9] which "can detect and repair the incorrect placement of sanitizers". Welearegai et al. [10] have proposed an idea to optimize the placement of sanitizers to reduce the amount of sanitization required. B. Livshits and S. Chong [11] have proposed the idea of fully automatic sanitizer placement by statically analysing the code and using taint tracking. And Yu et al. [12] have shown that automatically generating sanitization statements can be used to patch vulnerable web applications.

Aside from research into correct placement of sanitizers, there is also research into finding missing sanitizers. For example, the tool CAT.NET [13] tracks the data flow to find vulnerabilities in binary code.

3.2.2 Usage order of sanitizers

Aside from the correct placement of sanitizers, one also needs to take into account the order in which multiple sanitizers may be applied. For example consider a function `remove_backslash` that removes all "\" and a function `escape_and` that escapes all "&" by prefixing it with "\". We are given the piece of text "Slashes & \'s" and we want to remove all "\" and escape all "&". If we first apply `remove_backslash` and then `escape_and` then the result would be "Slashes \& 's". However, if we first apply `escape_and` and then `remove_backslash` then the result would be "Slashes &

's". Therefore we conclude that the usage order of sanitizers can influence the final result. One should therefore be very careful when using multiple sanitization functions on one string and make sure that it will give them the desired effect.

3.2.3 Context sensitivity

Finally, the last thing to keep in mind is that sanitizers are context sensitive. Depending on what kind of text you are sanitizing, for example a piece of HTML, Javascript or XSS, you will need to sanitize the string in different ways. For example, "escape, which does an HTML entity encode, is safe for use in HTML tag context but unsafe for other contexts" [14]. It is therefore important to know in what kind of context you are applying a sanitizer and whether the sanitizer is safe for that context.

3.3 Properties of sanitizers

There are three properties of sanitization functions that are important to discuss namely equivalence, idempotency and commutativity.

3.3.1 Equivalence

First of all, two sanitization functions A and B are *equivalent* if applying A to any input string always result in the same output as applying B. $A(s) = B(s)$ for all possible strings s.

Checking equivalence is especially useful to find deviations between sanitizers. Moreover, we need to be able to check equivalence to decide whether sanitizers are idempotent or commutative (see 3.3.4).

3.3.2 Idempotency

Secondly, a sanitization function A is *idempotent* if after applying A x times (where $x > 1$), the result will always be equal to applying the function only once. $A(s) = A(A(s)) = A(A(A(s))) = ...$ for all possible strings s.

If sanitizers are not idempotent then this can indicate a possible vulnerability. Imagine a sanitizers whose goal is to remove script-tags from a given input. It will search for the text "<script>" and "</script>" within the given input and if found, it will replace it by the empty string. However, it will traverse the string only once from beginning to end. As a result, if we input the string "<scr<script>ipt>alert(Hi!)</scr</script>ipt>", the result will be "<script>alert(Hi!)</script>". This result still contains a script-tag thus we consider it to be improperly sanitized.

3.3.3 Commutativity

Finally, two sanitization functions A and B are *commutative* if first applying A and then B to the input, is always equal to first applying B and then A to the input. $A(B(s)) = B(A(s))$ for all possible strings s.

As could be seen in the example discussed in 3.2.2, the order in which sanitizers are applied can influence the final result. It is even possible that one sanitizer reverses the changes made by another sanitizer.

3.3.4 Proving idempotency and commutativity

We are able to check idempotency and commutativity of sanitization functions when we can compose sanitization functions and check equivalence of them.

To prove that a sanitization function is idempotent, we consider the original sanitization function f and a second sanitization function $f \circ f$ which consists of two original functions composed. If we are able to decide equivalence, then we can prove that $f = f \circ f$. Assume that for some n, $n >= 2$, that $f(s) = f^n(s)$. We then need to prove that $f^{n+1}(s) = f(s)$. We can rewrite $f^{n+1}(s)$ as $f(f^n(s))$. We know that $f^n(s) = f(s)$ and therefore $f^{n+1}(s) = f(f^n(s)) = f(f(s)) = f(s)$.

To prove that two sanitization functions A and B are commutative, we consider two sanitizers. The first sanitizer S_1 consists of first applying A and then B ($B \circ A$), the second sanitizer S_2 consists of first applying B and then A ($A \circ B$). The sanitization functions are commutative when $S_1 = B \circ A = A \circ B = S_2$. To determine this we need to compose the two sanitization functions and check the equivalence of the resulting sanitizers.

Chapter 4

Related Work

4.1 Correctness of sanitizers

Weinberger et al. [14] have studied XSS sanitization in web application frameworks. In this study, they have found that "there is a wide gap between the abstractions provided by frameworks and the requirements of applications." [14]. They indicate that implementing sanitizers is very error-prone. Therefore, being able to reason about the correctness of sanitizers is important. Some research on the correctness of sanitizers has been done and will now be discussed.

First of all, Balzarotti et al. [15] try to identify vulnerabilities in sanitization procedures used in web applications. To achieve this, they use static analysis to find potentially vulnerable sanitization procedures. Afterwards, they use dynamic analysis to confirm that this is an actual vulnerability. This approach has been implemented in the tool Saner.

Unlike Balazarotti et al., who studied existing web applications, Hooimeijer et al. [16] take another approach. They introduce the language BEK for writing new sanitizers. This language allows the user to perform analysis of the sanitizer's behaviour, including checking equivalence, idempotency and commutativity. These properties can be used to study the correctness of sanitizers. Our research focuses on performing similar analysis on *existing* sanitizers such that no rewriting of the sanitizer in the BEK language is necessary.

Multiple approaches already exist which can be used to reason about existing sanitizers. For example, Argyros et al. [4] present an algorithm that learns symbolic finite automata from sanitizers. These models can then be converted into BEK programs which allows further analysis of the models. They briefly sketch how the learning algorithm for symbolic finite automata can be adapted to learn symbolic finite transducers. In this research, we have invented and implemented an algorithm for deducing symbolic finite transducers and use it to reason about the correctness of sanitizers by comparing it to specifications.

Botinčan and Babić [17] have presented the technique Sigma* which learns symbolic lookback transducers from programs. Unlike Argyros et al., this technique is a white-box technique. Sigma* uses dynamic symbolic execution to discover the transitions in the automaton. This is followed by counterexample guided abstraction refinement to refine the automaton. They show that their technique can be used to reason about the correctness of web sanitizers. Unlike this research, we aim to implement a black-box algorithm which enables us to reason about the correctness of sanitizers.

Alkhalaf [18] presents another automata-based symbolic string analysis approach which is used to over-approximate values for the input or output of a function. Validation and sanitization functions are extracted using formal specifications for different functions, after which the automata-based symbolic string analysis is performed.

The result is then used to detect and repair bugs in the function. The detection and repairing of bugs is done by either comparing the automata to a given policy or by comparing two functions. Note that in our research we reason about symbolic finite transducers instead of symbolic finite automata due to which we can do more extensive analyses.

Finally, Yu [19] presents another automata-based symbolic string analysis approach which can be used to automatically verify string manipulating programs. This technique relies on given attack patterns to find vulnerabilities in functions. They specify a vulnerability signature which indicates all inputs that may exploit a vulnerability. Then they modify a given input, if it matches the vulnerability signature, to prevent exploitation. The modification of a given input can be seen as automatically sanitizing the input. This technique has been implemented in a tool called Stranger. This tool has been evaluated and has been shown to detect vulnerabilities when presented with correctly specified attack patterns [20]. The main focus of Yu is to find vulnerabilities in sanitization routines whereas our research focuses on reasoning about all behaviour of sanitizers, which can be used to, but is not limited to, finding vulnerabilities.

4.2 Placing sanitizers

Although a correct implementation of sanitizers is necessary, the placement of sanitizers also influences the correctness of an application. Saxena et al. [9] present a tool which detects and repairs incorrect placement of sanitizers.

Aside from detecting incorrect placement, there are also techniques to automatically place all sanitizers in an application which eliminates possible wrong placement of sanitizers [10, 11].

Yu et al. [12] have shown that automatically placing sanitization statements can also be used to patch web applications. And, Long et al. [21] propose a technique that automatically modifies input to eliminate any dangerous input that may exploit a vulnerability. This technique not only places, but also generates, the sanitizers automatically. It does, however, rely on a learning phase, thus it is unlikely to detect new vulnerabilities. These different approaches all focus on the correct placement of sanitizers and are therefore complementary research to the ideas discussed in this thesis.

4.3 Vulnerability Detection

Using the approach in this research, we are able to reason about the correctness of string manipulating programs. We can reason about many correctness properties of such programs, including the presence or absence of specific vulnerabilities. Finding vulnerabilities has extensively been researched and some researchers decided to focus on vulnerabilities in sanitization functions. For example, Mohammadi et al. [22] automatically generate security test cases in order to detect injection vulnerabilities in sanitization functions. And Shar et al. [23] have data mined sanitization methods in order to predict XSS and SQL injection attacks.

4.4 Sanitization-free defenses

Sanitization is a very well known approach to prevent vulnerabilities. There are, however, also defenses that do not use sanitization. Scholte et al. [24] show that automated input validation can be a good alternative to output sanitization to prevent XSS and SQL injection vulnerabilities. Input validation checks whether a given input is of an allowed form or type. For example, one may check whether a given value is a valid email address or whether a given value is an integer. The difference with sanitization is that validation only checks for the validity of the data. It does not modify the input in any way. Scholte et al. use machine learning and static analysis to determine what the type of an input parameter should be. Then, validators for the learned type are applied to each parameter. They have developed an approach which automatically validates input parameters at runtime.

Another sanitization-free defense was presented by Costa et al. [25]. Costa et al. have made a tool Bouncer that prevents exploitation of software by generating input filters for a program. If the input is considered dangerous then it will be dropped, i.e. it will not be passed to the program. As a result, exploitation cannot take place if all malicious messages are correctly identified as dangerous.

4.5 Automata learning

Finally, we discuss research that has been done on deriving models from existing programs, also called "automata learning". Black-box automata learning has first been introduced by D. Angluin with the L* algorithm [26]. This algorithm learns deterministic finite automata using equivalence and membership queries. A similar learning approach, using equivalence and membership queries, has been developed for many other types of automata such as:

- Mealy machines [27]
- Moore machines [28]
- Extended mealy machines [29]
- Register automata [30, 31]
- Hybrid automata [32]
- Symbolic finite automata [4, 33]

In this research, we extend this list by developing a learning algorithm for symbolic finite transducers.

Automata learning can be quite time consuming and the implementation of the oracles (see Chapter 7) can greatly influence this. Henrix [34] has investigated several ways which could be used to improve the performance of a learning algorithm. He has found that parallelization and saving and restoring software states can be used to improve the performance of learning Mealy machines. Similar approaches could be used to improve the performance for the symbolic finite transducer learning algorithm (see Section 7.2) that is presented in this thesis.

Automata learning has also been evaluated through multiple case studies in which it has been used to learn systems such as an Engine Status Manager from Océ [35], communication protocol entities [36] and a computer-telephone integrated (CTI) application [37]. This has shown that automata learning can be used to derive models from large real-world systems.

Chapter 5

Verifying Sanitizers

In this chapter, we look into several ways which could be used to reason about sanitizers. Firstly, we have a look at existing string verification research where we focus on automata-based string analysis which has been proven useful in earlier sanitization analysis. Secondly, we investigate whether sanitizers can be represented by string rewriting systems. This chapter introduces theoretical concepts such as symbolic automata, transducers, and string rewriting systems. The information on symbolic automata and transducers is considered essential background information for this research.

5.1 String Verification

To be able to reason about sanitizers and their results, we need to be able to reason about strings. String verification is the research topic that focuses on techniques to reason about the value of a string at a specific point in program execution. This is especially difficult since a string can have infinitely many values. As a result, verifying string manipulating programs, such as sanitizers, is undecidable [38]. Instead of finding an exact solution, the string analysis techniques focus on finding either an over- or under-approximation of possible string values. Examples of such techniques are:

- Grammar-based string analysis
- String constraint solving
- Automata-based symbolic string analysis

In this research, we focus on automata-based symbolic string analysis techniques as this has proven to be effective in sanitizer analysis [4, 15, 16]. Automata-based symbolic string analysis uses abstract representations of a program called automata to reason about the value of a string. In the following sections, we introduce several types of automata which could be used to reason about strings.

5.1.1 Deterministic Finite Automata

A *deterministic finite-state automata* (DFA) is represented by the quintuple $(Q, \Sigma, \delta, q_0, F)$ [39]:

- Q is a finite set of states
- Σ is the input alphabet
- δ is a function from $Q \times \Sigma$ to Q. It represents all transitions in the automaton.

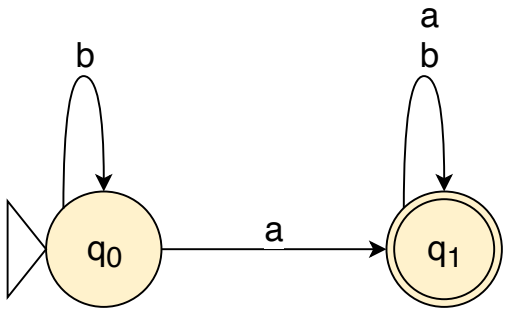


FIGURE 5.1: DFA that accepts all words which contain at least one *'a'* character

- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of accepting states

An example of a DFA can be seen in Figure 5.1. This DFA will only accept a word, which consists of a combination of the characters `a` and `b`, if it contains at least one `a`. Q, which is the set of states consists of $\{q_0, q_1\}$. The input alphabet Σ consists of $\{a, b\}$. The accepting state, q_1, is encircled twice in the figure. Finally, the lines, annotated with characters from the input alphabet, represent all transitions in the automaton. To check whether a DFA accepts a certain word, you start in the initial state (which is indicated with an arrow). Then you look at the first character of the word and follow the transition which starts in your current state and is labelled with that character. Repeat this process until all characters of the word have been processed. If you end in an accepting state, then the word is accepted by the automaton.

DFAs accept some set of words from the given input language. Therefore, they can represent a set of acceptable string values at a specific point in a program. However, they cannot represent how strings are changed by a program. There is no way to represent the relation between two strings in a DFA.

5.1.2 Finite State Transducers

To reason about sanitizers, we need to be able to reason about the relation between the input and output string, therefore DFAs are not sufficient. Instead of DFAs, we therefore look at finite state transducers. Transducers are somewhat similar to DFAs, however they generate output based on the given input. A *finite state transducer* (FST) is represented by the septuple $(Q, \Sigma, \Gamma, \delta, \omega, q_0, F)$ [40]:

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the output alphabet
- δ is a function from $Q \times \Sigma \times \Gamma$ to Q that represents all transitions in the automaton.
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of accepting states

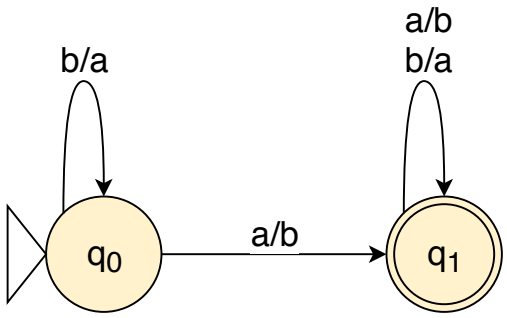


FIGURE 5.2: FST that accepts all inputs which contain at least one *'a'* character. It outputs the original word with all b's replaced with a's and vice versa.

See Figure 5.2 for an example of a finite state transducer. The transitions are labelled with a character from the input alphabet, followed by a "/", and then a character from the output alphabet. A transition labelled `x/y` means that upon detecting the character `x`, we will take the transition to the next state, and output `y`. This specific transducer outputs the inverse of the given input, replacing all a's by b's and vice versa. Given the input "abba", it will compute the output "baab". If one wants to represent deletion of a character, then this can be done by outputting the empty string upon detecting the character.

Finite state transducers allow us to represent the relation between two strings. Thus we can use this to represent the relation between the input and output string in a sanitizer. One can however imagine that such an automaton will get many transitions if we have an input alphabet that is larger than two characters since we require one transition per input character per state. Thus if we only use the generic alphabet (a-z), this would already lead to 52 transitions in the case of two states as in the example. To make the automaton more concise, we turn to symbolic automata.

5.1.3 Symbolic Finite Automata

"*Symbolic Finite Automata* (SFAs) are finite state automata in which the alphabet is given by a Boolean algebra that may have an infinite domain, and transitions are labelled with first-order predicates over such algebra." [41] Thus the most important difference between SFAs and DFAs is that transitions are not labelled with specific characters from the input alphabet, but instead they are labelled with first-order predicates. The second important difference is that SFAs are not necessarily deterministic whereas DFAs are deterministic by definition. An SFA is deterministic if, for each state in the SFA, the transitions starting in that state are labelled such that when multiple predicates are true at the same time, then these transitions will lead to the same state. If an SFA is non-deterministic, then there is at least one state where multiple transitions can be taken which lead to different states.

An *effective Boolean algebra* can be described as the following tuple $(\mathcal{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ [42]:

- $\mathcal{D}$ is the set of domain elements
- Ψ is a set of predicates closed under the Boolean connectives, with $\bot, \top \in \Psi$
- The component $[\![\_]\!] : \Psi \to 2^{\mathcal{D}}$ is a denotation function such that:

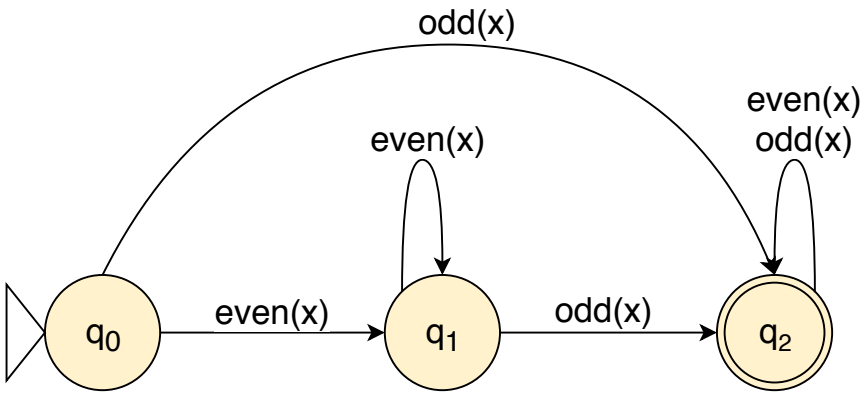


FIGURE 5.3: SFA that accepts all inputs which contain at least one odd number.

- $[\![\bot]\!] = \emptyset$
- $[\![\top]\!] = \mathcal{D}$
- For all $\varphi, \psi \in \Psi$,
  - $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$
  - $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$
  - $[\![\neg\varphi]\!] = \mathcal{D} \backslash [\![\varphi]\!]$

An example of a Boolean algebra could be the following. We have the domain consisting of the numbers 0 up to including 9. The set of predicates contains `odd(x)` and `even(x)`. These predicates will return whether a given number x is odd or even, respectively. For example, `odd(1)` will evaluate to true whereas `even(3)` will evaluate to false.

Symbolic Finite Automata can be defined using the tuple (A, Q, q_0, F, δ) [42]:

- 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 accepting states
- δ is a function from $Q \times \Psi_A$ to Q that represents all transitions in the automaton.

See Figure 5.3 for an example of a Symbolic Finite Automata. The transitions are labelled with the predicates `even(x)` and `odd(x)`. They return true if x is an even number, or odd number, respectively These predicates correspond to the predicates for the Boolean algebra described earlier. Instead of comparing your first character to the labels on the transition, as was done in DFAs, we now need to compute `even(x)` and `odd(x)`. You will then follow the transition whose predicate evaluates to true. For example, take the input 261. We start in q_0 and compute `odd(2)` and `even(2)`. Since 2 is an even number, the predicate `even(2)` evaluates to true and thus we follow this transition and end up in q_1. The second number, 6, is also even, thus the predicate `even(6)` will evaluate to true. We follow the transition which leads us to the same state q_1. Finally, we compute `odd(1)` and `even(1)`. This time `odd(1)`

evaluates to true, thus we end up in the state q_2. This is an accepting state thus the input 261 is accepted.

In order to reason about the correctness of SFAs, we also introduce several operators namely intersection, complement and equivalence.

Intersection of SFAs

The following notation will be used in the following sections:

- q_i^M denotes state q_i in automaton M.
- $\delta_M(q_i)$ denotes the set of transitions in automaton M which start in state q_i.
- F_M denotes the set of final states of automaton M.
- $Guard(t)$ denotes the guard of transition t.
- $Target(t)$ denotes the target to which transition t leads.

The intersection of two SFAs can be computed using the following algorithm [43]:

1. Let $S = (\langle q_0^A, q_0^B \rangle)$, $V = \{\langle q_0^A, q_0^B \rangle\}$ and $T = \emptyset$.
2. If S is empty, go to step 4 else pop $\langle q_1, q_2 \rangle$ from S.
3. Iterate for each $t_1 \in \delta_A(q_1)$ and $t_2 \in \delta_B(q_2)$, let $\varphi = Guard(t_1) \wedge Guard(t_2)$, let $p_1 = Target(t_1)$ and let $p_2 = Target(t_2)$. If φ is satisfiable then
   (a) Add the transition $(\langle q_1, q_2 \rangle, \varphi, \langle p_1, p_2 \rangle)$ to T
   (b) If $\langle p_1, p_2 \rangle$ is not in V then add $\langle p_1, p_2 \rangle$ to V and push $\langle p_1, p_2 \rangle$ to S
   (c) Proceed to step 2.
4. Let $C = (\langle q_0^A, q_0^B \rangle, V, \{q \in V | q \in F_A \times F_B\}, T)$ where V represents the set of states and T the transition relation.
5. Eliminate states in C from which no final state can be reached.

Note that we need to be able to check whether a certain guard is satisfiable. Therefore the SFAs, of which we compute the intersection, should operate over an effective Boolean algebra.

Complement of an SFA

When computing the complement of an SFA, the SFA first needs to be made complete. An SFA is incomplete if there is a state for which there exists an input i for which the guards of all transitions starting in s evaluate to false.

The complement $\overline{A}$ of SFA A can be computed as follows [42]:

1. First add a non-final state s to the automaton.
2. Add a transition from s to itself with the condition "True".
3. For each non-complete state p in A, add the following transition: $p \xrightarrow{\neg(Guard(t_1) \vee Guard(t_2) \vee ... \vee Guard(t_n))} s$ where $t_1, t_2, ..., t_n$ denote all transitions starting in p.
4. Finally, make all final states non-final and vice versa.

Equivalence of SFAs

Note that equivalence of SFAs is only decidable "if the alphabet theory forms a decidable Boolean algebra" [44]. Two SFAs A and B are equivalent if they accept the same language ($\mathcal{L}(A) = \mathcal{L}(B)$ [42]). This can be computed as follows:

1. Compute $C = \overline{A} \cap B$
2. Compute $D = A \cap \overline{B}$
3. If C and D are empty, then there is no word that is accepted by one automaton but not by the other. Therefore $\mathcal{L}(A) = \mathcal{L}(B)$, i.e. the SFAs are equivalent.

5.1.4 Symbolic Finite Transducers

Now that some basics of symbolic automata have been explained, we introduce the *Symbolic Finite Transducer* (SFT). Similar to a Finite State Transducer, 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 [45]. Note that it is possible to compute multiple outputs per input character. A *Symbolic Finite Transducer* can be described using the tuple $(Q, q_0, F, \iota, o, \Delta)$ [45]:

- 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
- o is the output sort
- Δ is a function consisting of $\Delta^{\epsilon} \cup \Delta^{\bar{\epsilon}}$:
  - Δ^{ϵ} is a function from $Q \times F(c_\iota) \times (T^o(c_\iota))^*$ to Q. It denotes all transitions in the automaton labelled with a first-order predicate and the set of output functions.
  - $\Delta^{\bar{\epsilon}}$ is a function from $Q \times \{\epsilon\} \times (T^o)^*$ to Q. It denotes all transitions in the automaton labelled with ϵ and the set of output functions. ϵ-transitions are found in non-deterministic automata and represent transitions that can be taken without consuming an input at any point in time.

Note that a Symbolic Finite Automaton is a Symbolic Finite Transducer that produces empty outputs [45].

An example of a Symbolic Finite Transducer can be seen in Figure 5.4. Each transition is now labelled x/y as in the transducer example. x denotes the input condition which is expressed using a predicate over some Boolean algebra. y denotes the output sequence which is one or more functions, expressed using the input symbol, which computes the output. To see how such an automaton works, we will have a look at the number 216 as an input for the SFT illustrated in Figure 5.4. The first number is even, thus we will take the transition from state q_0 to q_0. Upon taking this transition, we compute the following output: $[x-1, x+1] = [1, 3]$. The second number is odd thus we will take the transition to the state q_1. Upon taking this transition we compute the output $[x] = [1]$. Thus our output, up to this point, is $[1, 3, 1]$.

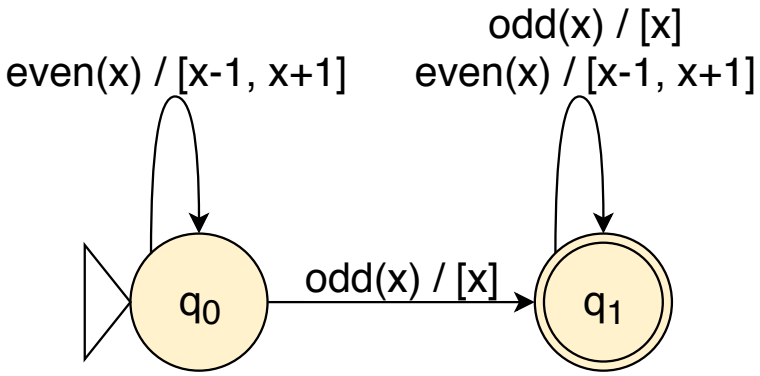


FIGURE 5.4: SFT that accepts all inputs which contain at least one odd number. For each even number it will output the two surrounding numbers and for each odd number it will output the odd number.

Finally, the last number is even, thus we take the transition labelled with `even(x) / [x-1, x+1]` to the state q_1. The output for this step will be $[x-1, x+1] = [5, 7]$. We are now done processing the input symbols. We end up in an accepting state thus the input is accepted. The complete output that has been computed is $[1, 3, 1, 5, 7]$.

Aside from presenting all possible string values of a specific variable at a program point, an SFT also shows the relation between the input and output. This is very useful to reason about sanitizers since it allows us to reason, not only about the input or output language, but also about the relation between the input and output language. As explained in Section 3.3.4, if we can prove equivalence of two sanitizers, then we can also prove commutativity and idempotency. Sanitizers can be represented by SFTs. Therefore, we examine what is needed to prove the equivalence of SFTs.

Equivalence of SFTs

In general, equivalence of SFTs is undecidable due to the unbound number of outputs that they may produce upon a given input [46]. However, sanitizers generally always produce the same single output upon a given input. In other words, if represented as an SFT, they always take one specific path to reach a final state. We call an SFT *single-valued* if there is at most one path starting in the initial state and ending in some final state, such that the input has been completely consumed, for all possible inputs. An SFT will be single-valued if it is deterministic. Note that there are also single-valued SFTs that are not deterministic. An SFT is *deterministic* if for all states p, if the input condition on two transitions starting in p is true, then these transitions will lead to the same state and the produced output will be the same. Equivalence of SFTs can be determined through two steps:

- *Domain equivalence*: $\mathscr{D}(A) = \mathscr{D}(B)$
- *Partial equivalence*: $\forall a \in \mathscr{D}(A) \cap \mathscr{D}(B), \mathscr{T}_A(a) = \mathscr{T}_B(a)$ where $\mathscr{T}_A(a) = \{b \in \Gamma^* | \exists q \in F_A(q_A^0 \overset{a/b}{\twoheadrightarrow} q)\}$. Γ^* denotes the possible outputs of automaton A.

Domain equivalence states that the SFTs A and B must accept the same inputs. Partial equivalence states that for each possible input, the two automata should produce the same output(s). These two statements combined mean that the SFTs A and B must accept the same inputs and produce the same outputs for each input.

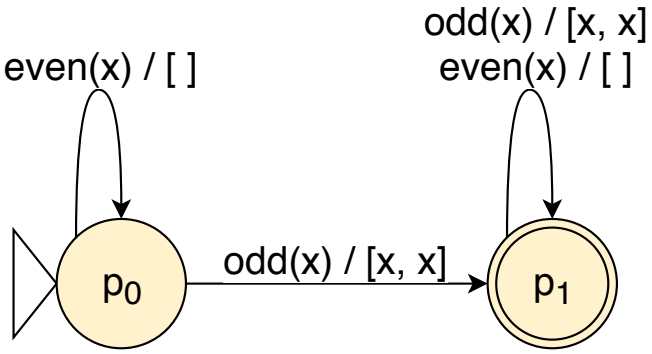


FIGURE 5.5: SFT T_1 which accepts all numbers with at least one odd number. It produces every odd number twice as output.

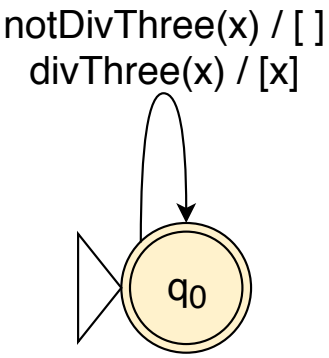


FIGURE 5.6: SFT T_2 which accepts all numbers. The output is the original input with all numbers not divisible by 3 removed.

Composition of SFTs

Function composition is the act of combining multiple functions to create a more complex function. In terms of sanitizers this means that we have an input which is given to some sanitization function whose output is passed to the next sanitization function and so on. The final result is equal to the result of the last sanitization function. Since each SFT represents one sanitization function, the composing of SFTs is relatively simple. We need to pass the result from the first SFT to the next and so on until the last sanitizer is reached. Take for example an SFT with the transition $p \xrightarrow{\varphi/[f_1,f_2]} p'$ and an SFT with the transitions $q \xrightarrow{\psi/[g]} q' \xrightarrow{\gamma/[h]} q''$. The composition of these transitions will give you the following: $(p,q) \xrightarrow{\varphi \wedge \psi(f_1) \wedge \gamma(f_2)/[g(f_1),h(f_2)]} (p',q'')$ [42]. It is important to note that the outputs of the first transition, f_1 and f_2, are given as arguments to the next transitions. This means that the input language of the second SFT must at least accept the output language of the first SFT.

An SFT composed from SFT A and SFT B is described by the tuple $(Q_A \times Q_B, (q_A^0, q_B^0), F_A \times F_B, \iota_A, o_B, \Delta_{A \circ B})$ [16]:

- $Q_A \times Q_B = \{(a,b) | a \in Q_A \text{ and } b \in Q_B\}$ is the cross-product of all states. All states that cannot be reached can be eliminated.
- $(q_A^0, q_B^0) \in Q_A \times Q_B$ is the initial state
- $F_A \times F_B \subseteq Q_A \times Q_B$ are the accepting states. A word is accepted if it is accepted by SFT A, and if the output of SFT A is accepted by SFT B.
- ι_A, the input language of SFT A, is the input language of the composed SFT.
- o_B, the output language of SFT B, is the output language of the composed SFT.
- $\Delta_{A \circ B}$ is the set of all transitions which are composed as explained earlier. All transitions with an unsatisfiable condition can be omitted.

Now an example of the composition of SFTs will be discussed. We consider the SFTs T_1 and T_2 which can be seen in Figure 5.5 and 5.6 respectively. Both SFTs take the input alphabet consisting of all natural numbers $\mathbb{N}$. The predicates in T_1 are the same as used in earlier examples of automata where `even(x)` is true when

the number is even, and `odd(x)` is true when the number is odd. The predicate `divThree(x)` in T_2 checks whether you can divide the given number by 3 without any remainder (x % 3 = 0). `notDivThree(x)` is the negated predicate `divThree(x)`. We start by computing the cross-product of all states: $\{(p,q)|p \in T_1 \text{ and } q \in T_2\} = \{(p0,q0),(p1,q0)\}$. The initial state is $(p0,q0)$. The accepting states are $\{(p1,q0)\}$. The input language, which is the input language of T_1, is the set of all natural numbers. The output language, which is the output language of T_2, is a subset of all natural numbers. Namely, all natural numbers that consist of digits that are divisible by three (3,6,9). Note that T_2 also accepts the empty word ε. Finally, the set of all transitions needs to be computed:

- $(p0,q0) \xrightarrow{even(x)/[]} (p0,q0)$
- $(p0,q0) \xrightarrow{odd(x) \wedge divThree(x) \wedge divThree(x)/[x,x]} (p1,q0)$ which can be simplified to $(p0,q0) \xrightarrow{odd(x) \wedge divThree(x)/[x,x]} (p1,q0)$. Note that this is only possible because the output of automaton T_1 is twice the same number and because the predicate is the same.
- $(p0,q0) \xrightarrow{odd(x) \wedge divThree(x) \wedge notDivThree(x)/[x]} (p1,q0)$. This transition can be omitted since the predicate is unsatisfiable due to the clauses $divThree(x)$ and $notDivThree(x)$.
- $(p0,q0) \xrightarrow{odd(x) \wedge notDivThree(x) \wedge divThree(x)/[x]} (p1,q0)$. This transition can also be omitted since the predicate is unsatisfiable.
- $(p0,q0) \xrightarrow{odd(x) \wedge notDivThree(x) \wedge notDivThree(x)/[]} (p1,q0)$ which can be simplified to $(p0,q0) \xrightarrow{odd(x) \wedge notDivThree(x)/[]} (p1,q0)$
- $(p1,q0) \xrightarrow{even(x)/[]} (p1,q0)$
- $(p1,q0) \xrightarrow{odd(x) \wedge divThree(x) \wedge divThree(x)/[x,x]} (p1,q0)$ which can be simplified to $(p1,q0) \xrightarrow{odd(x) \wedge divThree(x)/[x,x]} (p1,q0)$
- $(p1,q0) \xrightarrow{odd(x) \wedge divThree(x) \wedge notDivThree(x)/[x]} (p1,q0)$ whose predicate is unsatisfiable thus it can be omitted.
- $(p1,q0) \xrightarrow{odd(x) \wedge notDivThree(x) \wedge divThree(x)/[x]} (p1,q0)$ whose predicate is unsatisfiable thus it can be omitted.
- $(p1,q0) \xrightarrow{odd(x) \wedge notDivThree(x) \wedge notDivThree(x)/[]} (p1,q0)$ which can be simplified to $(p1,q0) \xrightarrow{odd(x) \wedge notDivThree(x)/[]} (p1,q0)$

The final automaton $T_2 \circ T_1$ can be seen in Figure 5.7. It will output all numbers that are odd and divisible by three twice. All other numbers will not occur in the output. Take for example the input 23569. The output produced by $T_2 \circ T_1$ will be $[3,3,9,9]$.

5.2 String Rewriting Systems

Sanitizers essentially rewrite a given input. They rewrite this input either by adding, replacing or removing characters. Therefore we investigate whether we can use rewrite systems to reason about the correctness of sanitizers.

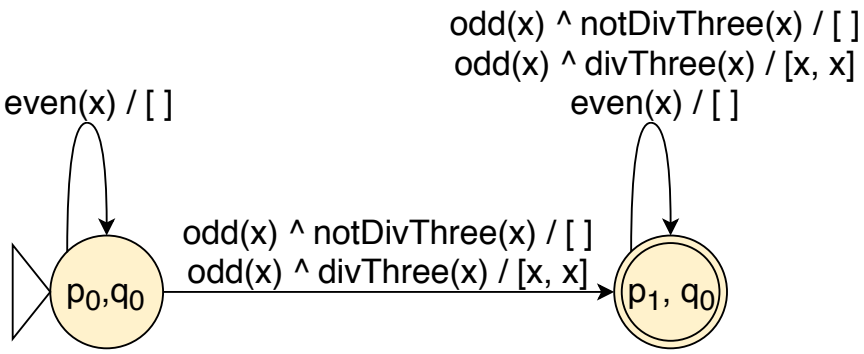


FIGURE 5.7: SFT $T_2 \circ T_1$ which represents the composition of SFTs T_1 and T_2

String Rewriting Systems (SRS), also called Semi-Thue systems, are a type of rewriting system for strings. It can be described using an alphabet Σ and a binary relation R which describes the rewriting rules. These rewriting rules have a form similar to "$usv \rightarrow utv$" where s, t, u, v are in the alphabet of the system. If the current input matches the pattern on the left-hand side, then it will be changed into the pattern on the right-hand side.

As an example consider an alphabet containing all unicode characters. The relation R contains two rewriting rules:

1. $< script > \rightarrow \varepsilon$
2. $< /script > \rightarrow \varepsilon$

ε denotes the empty string in these rules. Note that in these rules we omit the u and v which are used to show that it may also be a substring of the original string. We start with the string "<scr<script>ipt>alert(hi!)</scr</script>ipt>". In the beginning, the first rule is applied, thus removing the nested <script> tag in the string. This results in the string "<script>alert(hi!)</scr</script>ipt>". We continue rewriting the string until no rules are applicable anymore. The final string will be "alert(hi!)". In Figure 5.8 you can see a representation of how applicable rules can be applied in different orders.

Although it seems like a natural way of reasoning about sanitizers, there are some issues to take into account. First of all, some sanitizers may stop after traversing the string only once. This may lead to cases where some rules would still be applicable. In the case of a rewriting system, one typically tries to apply rules until no more rules are applicable. Thus a sanitizer would then be represented by only a subset of the applied rules.

Secondly, rewriting systems may not terminate whereas sanitizers should. This behaviour can occur when adding or replacing characters which enables the applicability of a rule. Such behaviour is however not possible when removing characters from the string. Reasoning about removing characters in a string using a rewrite system will always terminate since the length of the string will strictly decrease. As a result, you will either end with the empty string or no rules will be applicable anymore.

5.2.1 Properties of Rewriting Systems

This section will describe some properties of rewriting systems and how they may be related to properties of sanitizers and the security of sanitizers.

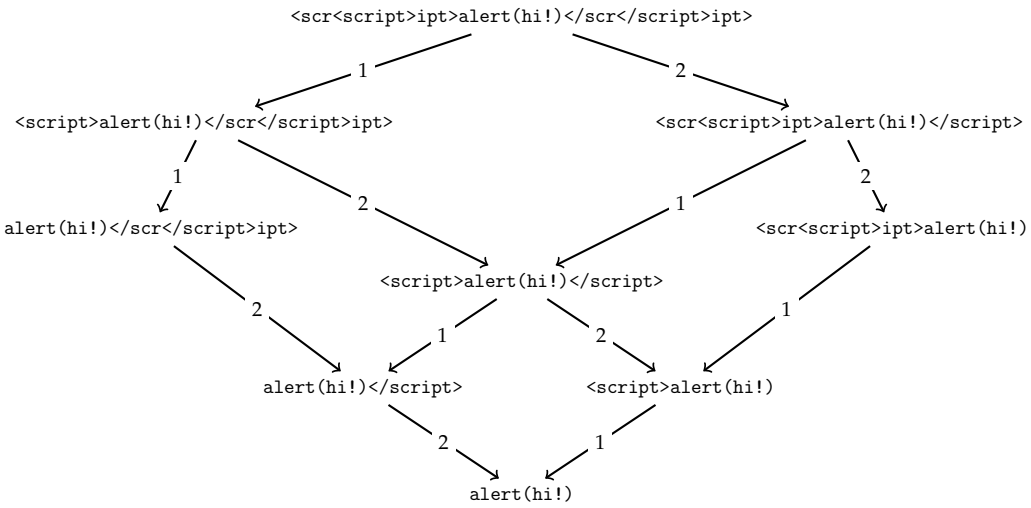


FIGURE 5.8: Diagram illustrating all possible paths of applying the rewriting rules. It should be read from top to bottom. The numbers on the edges denote which rewriting rule is applied.

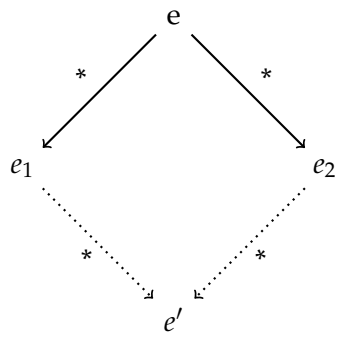


FIGURE 5.9: Diagram illustrating the Church-Rosser property.

The *Church-Rosser* property states that two diverged paths are joinable (see Figure 5.9). Assume that we have some input e. We have a rewriting system consisting of multiple rules of which two are applicable to e. Applying either the first or second applicable rule results in e_1 and e_2 respectively. There are now two diverged paths: $e \to e_1$ and $e \to e_2$. Joining these two diverged paths means that we apply (other) rewriting rules until we end up with the same result e'. Thus, the Church-Rosser property states that, if we apply two different rewriting rules to the same input, then they can lead to the same output. This property is similar to the notion of commutativity of sanitizers. The difference, however, is that commutativity reasons about the result in two rewriting steps whereas the Church-Rosser property allows taking multiple steps to reach the common successor.

Termination of a rewriting system states whether at some point in time, no more rules will be applicable. Termination is guaranteed when reasoning about rules which remove characters from the input. It is, however, not guaranteed when reasoning about rules that add or replace characters in the input. If a sanitizer is idempotent, then its corresponding rewriting system terminates. However, if a rewriting system is non-terminating, then this can indicate that the corresponding sanitizer might be vulnerable since it must be non-idempotent (see Section 3.3.2).

5.2.2 Word problem

Finally, we investigate a well known problem of string rewriting systems called the word problem, and its relation to sanitizers. The word problem for String Rewriting Systems can explained as follows: given some SRS (Σ, R) and $u, v \in \Sigma^*$, can u be transformed into v using the rules in R. This problem is undecidable. It is however important to note the following. If we can describe a sanitizer as an SRS, then the following question is undecidable: given an input u, can we transform it into (malicious) output v using the rules in R.

5.3 Automata-based analysis or String Rewriting Systems?

In this chapter, we have investigated two possible approaches to reason about the behaviour of sanitizers. String Rewriting Systems seem a very logical model to reason about sanitizers, however we do not know how we could derive a String Rewriting System from a sanitizer. Moreover, there are still some theoretical issues which would need to be investigated further such as how to determine when a string rewriting system should terminate. Automata-based analysis of sanitizers has already been used to analyse sanitizers' behaviour and has been proven to be effective. Therefore we will use automata-based analysis to model sanitizers in our research.

Chapter 6

Specifying Sanitizers

This chapter will discuss what specifications one would want to write for sanitizers and what is needed to prove these specifications.

6.1 Specifications

This section will give an overview of specifications that may be written to specify the behaviour of a sanitizer. Note that the main focus of this research is to reason about the security of sanitizers thus the specifications will also focus on properties related to this.

To be able to reason about whether sanitizers work correctly, we need to know what the correct behaviour is. To achieve this, one can write a specification: an abstract view of how the sanitizer should act. The existing sanitizer is then compared to the specification to check whether the sanitizer adheres to the specifications.

Below you can find a list of possible behaviours that might be specified for a sanitizer. The options on this list have been discovered through a brainstorming session with employees of the company Northwave [47] which specializes in, among other things, security testing. The complete list of suggested specifications can be found in Appendix A.

- *Blacklisting*: You may want to specify which characters are disallowed in the output. It allows reasoning about some of the simplest sanitizers such as for an email address sanitizer which removes disallowed characters.
- *Whitelisting*: You may want to specify which characters are strictly allowed in the output.
- *Structure of output*: If you can specify the structure of the output, one can make sure that the output of a sanitizer will always adhere to a certain structure. For example, a sanitizer might always output a valid HTML document. Note that this is related to validation where you check whether input adheres to a specific structure. One might also want to check whether the structure of the input is the same as the structure of the output. Note that structure, in this context, refers to the way that documents' contents are ordered. For example, the structure of an HTML document is described using HTML elements which are denoted using tags such as " , <body></body>".
- *if* z *then* $x \rightarrow y$: If condition z is true, then everything that matches x should change into y. This allows you to specify changes such as delete a whitespace if it is a trailing whitespace. Note that it can also be used to denote that something must *not* change by using *if* z *then* $x \rightarrow x$.

  - There is also a special case of this specification namely $x \rightarrow y$ which means that x *always* changes into y.

- *Length*: You may want to specify the allowed length of the output, which may be related to the length of the input.
- *Equivalence, idempotency and commutativity*: You might want to know whether a sanitizer is equal to another sanitizer, whether a sanitizer is idempotent and whether it is commutative in regards to another sanitizer.
- *Bad output specification*: Given a bad output, you want to know what inputs can lead to that bad output. If you have a set of inputs that lead to the bad output, then one likely wants to specify that this set is empty or perhaps get an example of a bad input.

6.2 Necessary models

This section examines what type of model each specification needs to be compared to. A specification can generally be divided into two categories namely input-/output-only or input-output relation. The first category includes specifications that only reason about either the input or the output language. The second category includes specifications that reason about the relation between the input and the output language.

6.2.1 Input-only or output-only

The following specifications only reason about the input or output of a sanitizer:

- Blacklisting
- Whitelisting
- Structure of output
- Length

To check these specifications, one needs a model that represents either the input or the output of the sanitizer, for example an SFA.

The structure of the output is only related to the output language. This can, however, not be represented by an SFA. Instead, a model such as Symbolic Tree Automata[48] can be used. In case one wants to reason about the way the structure is changed, then a model such as a Symbolic Tree Transducer [49] can be used. Note that tree automata and tree transducers are out of scope for this research. Therefore, we will not be able to reason about the structure of the input or output.

6.2.2 Input-output relation

The following specifications reason about the relation from input to output of a sanitizer:

- if z then $x \rightarrow y$
- Equivalence, idempotency and commutativity
- Bad output specification

To prove these specifications, one needs a model that represents the relation between the input and output of a sanitizer such as an SFT.

Idempotency and commutativity only reason about the output. However, equivalence checking is needed to determine these properties. Since equivalence checking depends on the input-output relation, idempotency and commutativity checking also depend on the input-output relation.

6.3 How to write specifications

In this section, we explore what the user needs to specify for each individual type of specification mentioned earlier. Moreover, we discuss how each of these specifications can then be checked using the specification provided by the user. We assume that specifications are written in the form of an SFA or SFT by the user. Note that an SFT specification needs to be single-valued. If this is not the case, then we cannot determine equivalence, thus checking the specifications becomes impossible. Since SFAs are closed under product [42], we can write multiple specifications and compute the product of these to check all specifications at once. This is, however, not possible with SFTs since SFTs are not closed under product [46].

Below are some notations that will be used in SFA and SFT examples in the upcoming sections:

- *true*: This matches any possible input character.
- x == y: Instead of using the notation `isY(x)`, we use `x==y` to denote that x is equal to character y.
- x != y: Instead of using the notation `isNotY(x)`, we use `x!=y` to denote that x is not equal to character y.
- ^: This is an operator between predicates which denotes a conjunction.

6.3.1 Blacklisting

Blacklisting limits what input is considered acceptable by comparing all inputs to a list of unacceptable inputs. If you want to write a blacklist specification, then you need to construct an SFA that accepts all unacceptable inputs as specified by the blacklist. If this has been specified, then you can compute the union with the SFA representing the accepted input language. If the union is equal to the empty automaton, it means that no unacceptable input is accepted.

Formally, this process can be described as follows. We need to specify an SFA A such that: $\forall s \in \Sigma^*.\neg isAcceptable(s) \iff A.accepts(s)$.
isAcceptable(s) denotes whether input s is acceptable according to the blacklist. *A.accepts(s)* denotes whether SFA A accepts the input s. Note that we reason about the input language Σ^* which is the input language of the *sanitizer*.

Assume that you have another SFA B, which is a model of the sanitizer. SFA B should have been made such that: $\forall s \in \Sigma^*.isAccepted(s) \iff B.accepts(s)$.
isAccepted(s) denotes whether the input s is accepted by the sanitizer. We then compute the union of these two automaton, which means that we construct the automaton $C = A \cap B$, for which the following is true by construction:

$$\begin{aligned} \forall s \in \Sigma^*.C.accepts(s) &= (A \cap B).accepts(s) \\ &= A.accepts(s) \wedge B.accepts(s) \\ &= \neg isAcceptable(s) \wedge isAccepted(s) \end{aligned} \tag{6.1}$$

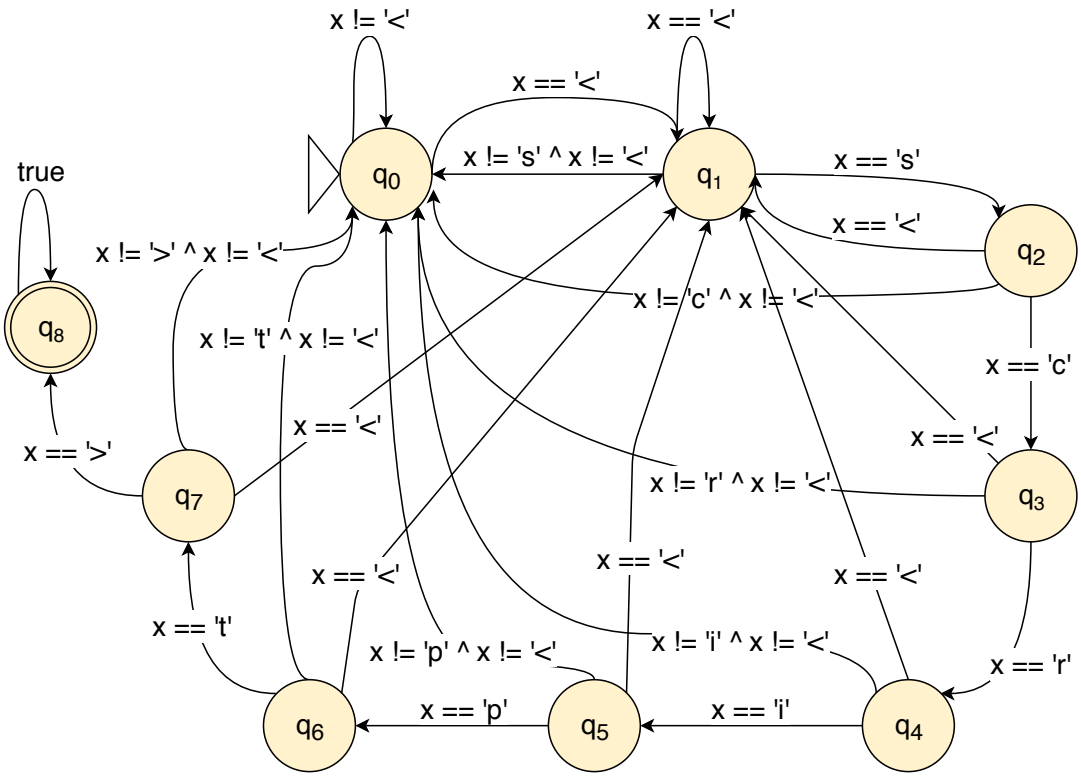


FIGURE 6.1: Example of a blacklisting specification where all words containing "<script>" are unacceptable.

Thus C accepts all words that are not acceptable according to the blacklist and are accepted by the sanitizer. Therefore if $\mathcal{L}(C) = \emptyset$ then there does not exist any input such that it is accepted by the sanitizer but not acceptable according to the blacklist.

An example could be that you want to reject all inputs that contain "<script>". A possible specification for this example can be seen in Figure 6.1. Note that this approach can also be used to reason about all unacceptable outputs, in which case the specified SFA is compared to the SFA that represents the output language.

6.3.2 Whitelisting

Whitelisting limits what input is considered acceptable by comparing all inputs to a list of acceptable inputs. The user can specify an SFA that accepts all acceptable inputs. Let SFA A be the specified SFA such that: $\forall s \in \Sigma^*.isAcceptable(s) \iff A.accepts(s)$.
In this case, *isAcceptable(s)* denotes whether s is acceptable according to the whitelist. Also, assume that we have an SFA B that represents the sanitizer such that: $\forall s \in \Sigma^*.isAccepted(s) \iff B.accepts(s)$ where *isAccepted(s)* denotes whether an input is accepted by the sanitizer.

In this case, there are two ways in which you can check the specification, based on the user's preference. Firstly, one can check whether SFA A is equal to SFA B. If the automaton are equivalent, then they accepts the same inputs. Secondly, one can check whether $B \subseteq A$. This means that the sanitizer accepts some, perhaps all, inputs from the whitelist. Note that there is *not* a third option where one can check whether $A \subseteq B$ since this would mean that the sanitizer may accept some word that is not allowed according to the whitelist.

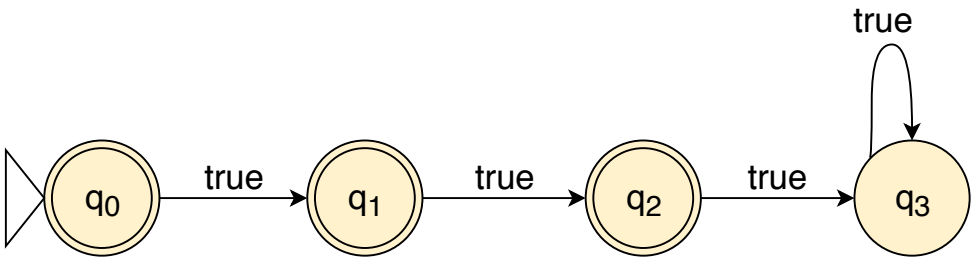


FIGURE 6.2: SFA which accepts all words with a length of at most 2.

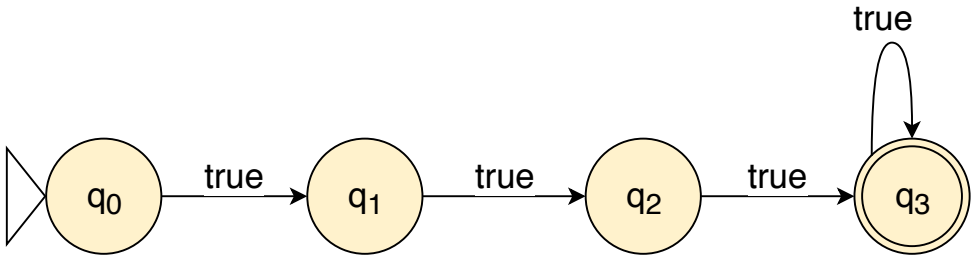


FIGURE 6.3: SFA which accepts all words with a length of at least 3.
Note that this SFA is the complement of the SFA in Figure 6.2.

Note that this approach can also be used to reason about all acceptable outputs, in which case the specified SFA is compared to the SFA representing the output language.

6.3.3 Length

The user may be interested in checking the length of all possible outputs of the sanitizer. If so, then the user can check whether all strings have the length $x \in \mathbb{N}$ by specifying an SFA that accepts all words of length x. We need to compute the union of the complement of this SFA with the SFA representing the input or output language. If this union is empty, then all words have the length x. If this union is non-empty, then there exists a word with a length that is not equal to x.

Formally, we can describe this process as follows. Construct an SFA A such that: $\forall s \in \Sigma^*.s.length() == x \iff A.accepts(s)$
where $x \in \mathbb{N}$ denotes the length for which we want to construct the SFA. Assume that you have an SFA B that represents either the input or output language of the sanitizer such that: $\forall s \in \Sigma^*.isAccepted(s) \iff B.accepts(s)$
where *isAccepted(s)* denotes whether the sanitizer accepts input s. We then construct the union, $C = \overline{A} \cap B$, for which the following is true by construction:

$$\begin{aligned} \forall s \in \Sigma^*.C.accepts(s) &= (\overline{A} \cap B).accepts(s) \\ &= \overline{A}.accepts(s) \wedge B.accepts(s) \\ &= (s.length() \neq x) \wedge isAccepted(s) \end{aligned} \tag{6.2}$$

If $\mathcal{L}(C) = \emptyset$ then all words that are accepted by the sanitizer have length x.

Note that instead of computing the complement of the specification, you can also specify an SFA which accepts all words which do not have length x. This approach can be used to check whether there exist input or output strings with a length x, not equal to x, smaller than x or greater than x (See Figures 6.2 and 6.3).

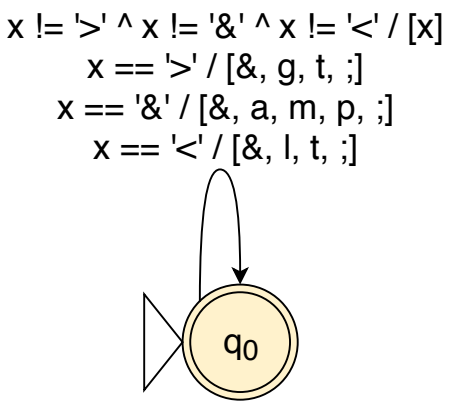


FIGURE 6.4: SFT which encodes <, > and & into their HTML reference. This SFT describes the behaviour of the Python `encode` method with the optional flag set to False (see Listing 3.4).

6.3.4 If z then $x \rightarrow y$

Next, we consider the "if z then $x \rightarrow y$" specification. In this case the user needs to specify the complete system. Remember that all arrows in an SFT are annotated with $pred(x)/[f(x)]$ where $pred(x)$ (=z) denotes a predicate over input x, and $f(x)$ (=y) denotes how the input x is changed, which results in y. Therefore the condition z is limited to what can be expressed using predicates over x in the boolean algebra. Below you can find a list of examples that are possible to express:

- Capitalize the letters a to z.
- Escape all quotes using a backslash, unless it is already escaped.
- Encode >, < and `&` into their HTML reference (see Figure 6.4).

And a list of examples that are *impossible* to express:

- Escape the last character in the input
- If the input word has an odd length then add a character.
- Remove a script tag and its corresponding closing tag.
- Encode `&` into its HTML reference, unless it is part of an HTML encoded `&`.

Note that at this point, it is not possible to express specifications over multiple input characters. This would require an automaton with lookahead, lookback or registers. As a result, we are unable to express something such as "Remove all HTML tags".

To check whether this specification is true, we compare the constructed SFT A to the SFT B which represents the sanitizer. Therefore, the specification needs to include all behaviour of the sanitizer.

An example specification for a sanitizer that encodes the character `&`, if it is not yet encoded, into its HTML reference can be seen in Figure 6.5. However, this specification does *not* perfectly model the behaviour of the sanitizer. If a string ends with `&a`, `&am` or `&amp` then it will only output `&` and ignore the following characters. We need to recognize the end of the input to be able to solve this problem. This can be modelled by adding ε-transitions to the automaton. 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.

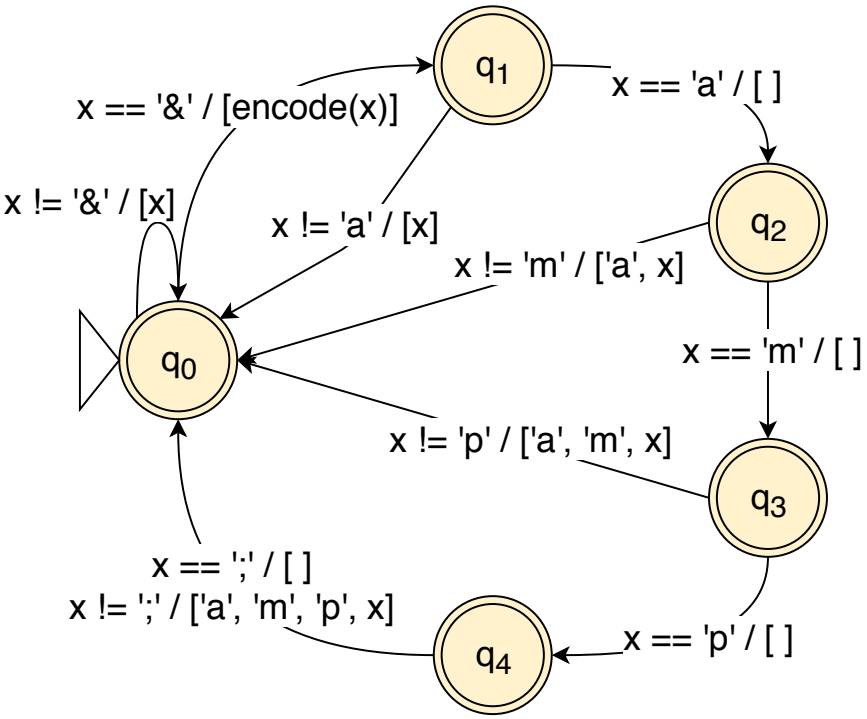


FIGURE 6.5: SFT which encodes & into its HTML reference, unless it is part of an encoded &. Note that this SFT does *not* precisely model the sanitizer.

6.3.5 Equivalence, idempotency and commutativity

Equivalence checking has already been explained in 5.1.4, therefore only idempotency and commutativity are discussed in this section.

The user does not need to specify anything to check idempotency. We will compute the composition of the SFT that represents the sanitizer with itself. This composed SFT is then equivalence-checked with the SFT that represents the sanitizer. Thus formally we compute whether $A \circ A = A$. If these are equivalent then we can conclude that the sanitizer is idempotent.

To check commutativity, the user needs to indicate with for which two sanitizers, A and B, commutativity needs to be checked. We then compute the compositions $A \circ B$ and $B \circ A$. If these compositions are equivalent, then we conclude that sanitizers A and B commute.

6.3.6 Bad output specification

To check what inputs can lead to a bad output, the user only needs to specify the bad output which can be done in the form of a string. Then, using a pre-image computation over the SFT representing the sanitizer, we can compute the set of inputs that lead to the bad output. Formally, we need to compute $Pre(y) = \{x | f(x) = y\}$ where $f(x)$ denotes the function which describes how the sanitizer behaves and y is the bad output that the user has specified. It is, unfortunately, impossible to compute the complete set of inputs that lead to a certain output. This is due to the fact that an infinite amount of inputs may lead to one specific output. You can compute a possible input by using a backwards breadth-first search (BFS) or depth-first search (DFS) where the output of the transitions is matched to the specified bad output. Note that the input that is found should start in an initial state and end in an accepting state.

Type of specification	What user needs to specify
Blacklisting	List of unacceptable words
Whitelisting	List of acceptable words
Length	Integer x and $=, \neq, <, >, \leq$ or $\geq$

TABLE 6.1: Table which shows per specification type what the user needs to specify in order to be able to automatically generate the specification.

6.4 Automatic generation of specifications

Writing specifications requires significant effort and knowledge. To simplify the user's tasks, we can automatically generate some specifications. The specifications mentioned in Table 6.1 can be automatically generated. The user will still need to specify what needs to be checked. However, it does require less work and less specific knowledge from the user. Thus, it makes the approach more user-friendly,

For the specification of a blacklist, we need an SFA that accepts all unacceptable inputs. When given a list of unacceptable words, i.e. the blacklist, such an automata can be automatically constructed as follows. Firstly, for each unacceptable input, construct the automaton that accepts only this input. Then, compute the union of all constructed automata. This results in the automaton which accepts all unacceptable inputs.

For the specification of a whitelist, we need an SFA that accepts all acceptable inputs. When given a list of acceptable words, i.e. the whitelist, then such an automaton can be automatically constructed in a similar manner as the specification for a blacklist. Firstly, for each acceptable input, construct the automaton that accepts only this input. Then, compute the union of all constructed automata. This results in an automaton which accepts all acceptable inputs.

Finally, we discuss how to automatically construct an automaton for a length specification. For each specification, we need to construct an automaton with $x + 2$ states where x denotes the specified length. Let the states be labelled $s_0, ..., s_{x+1}$. For each state s_i in $\{s_0, ..., s_x\}$, construct a transition from s_i to s_{i+1} labelled with *true*. Next, construct a self-loop, labelled *true*, for the state s_{x+1}. Finally, we need to define the accepting state(s) as follows:

- In case of $=$, make the state s_x accepting.
- In case of $\neq$, make all states except s_x accepting.
- In case of $<$, make all states s_i for which $i < x$ accepting.
- In case of $>$, make all states s_i for which $i > x$ accepting.
- In case of $\leq$, make all states s_i for which $i \leq x$ accepting.
- In case of $\geq$, make all states s_i for which $i \geq x$ accepting.

Note that if the length automaton are constructed this way, the automaton for $<$ and $\geq$ can be minimized by removing the $(x+1)^{th}$ state and adding a self-loop to the x^{th} state.

One can also construct automata where these conditions are combined. For example, the user might want to specify an automaton where all lengths between l and m are acceptable. In this case, all states s_i for which $i > l$ and $i < m$ should be made accepting. The user can also construct this automaton by specifying two

automaton, one which accepts all lengths greater than l and one which accepts all lengths smaller than m. Then we can compute the union of these two automaton. The resulting automaton will accept all lengths which are greater than l and smaller than m.

Chapter 7

Learning Algorithms

This chapter introduces learning algorithms and how they can be used to deduce models from code in a black-box manner. Black-box means that we are only able to observe the inputs and outputs of a system, but not the inner workings. Learning algorithms construct models from existing programs which describe how these programs works. We investigate the use of learning algorithms because it can provide us with a model that can be compared to specifications. Firstly, the original L* algorithm is explained in Section 7.1. Then, in Section 7.2, our SFT learning algorithm is discussed.

7.1 L* algorithm

The L* algorithm has been introduced by Angluin[26]. The algorithm can be used to deduce a DFA that represents the output of a program.

Firstly, some background information is introduced in Section 7.1.1. Then, in Sections 7.1.2 and 7.1.3, the algorithm is explained. Finally, we show an example of the L* algorithm in Section 7.1.4.

7.1.1 Preliminaries

In the L* algorithm we construct a hypothesis automaton. This hypothesis automaton should describe the behaviour of the program which we want to learn, also called the System Under Learning (SUL). To learn the behaviour of the SUL, the algorithm can execute two types of queries:

- *Membership queries*: the algorithm submits a string s to the SUL and obtains the output of the SUL.
- *Equivalence queries*: the algorithm can submit a hypothesis automaton to the 'teacher'. The teacher will then respond with a confirmation that the hypothesis is the same as the SUL or it will return a counterexample that distinguishes the hypothesis and the SUL.

To keep track of the results of all queries, and to construct a hypothesis, the algorithm will keep track of the results in an observation table (OT).

An observation table OT with respect to an automaton M is a tuple $OT = (S, W, T)$ where

- $S \subseteq \Sigma^*$ is a set of access strings
- $W \subseteq \Sigma^*$ is a set of distinguishing strings
- T is a partial function $T : \Sigma^* \times \Sigma^* \to \{0, 1\}$

In an observation table the set S forms the rows, W forms the columns and T represents all entries in the table. For example if we have a row "ab" and column "a", then the entry in the table denotes whether "aba" is accepted by the SUL or not.

For some upcoming definitions we introduce the syntax $M_q[s]$ which denotes the state that is reached on input s in automaton M when starting in state q. If the q is omitted then we start in the initial state of automaton M.

The set of *access strings* A is defined as follows for automaton M with the set of states Q_M: "For every state $q \in Q_M$, there is a string $s_q \in A$ such that $M[s_q] = q$" [4]. Thus the set of accepting strings contains all strings such that each state in the automaton is reachable by at least one string.

The set of *distinguishing strings* D is defined as follows for automaton M with the states Q_M: "For any pair of states $q_i, q_j \in Q_M$, there exists a string $d_{i,j} \in D$ such that exactly one state of $M_{q_i}[d_{i,j}]$ and $M_{q_j}[d_{i,j}]$ is accepting" [4]. Thus the set of distinguishing strings contains all strings such that, for each pair of states, there is at least one string that is accepted in one state but not by the other.

Let OT be an observation table. OT is *closed* if, for all $t \in S \cdot \Sigma$, there exists $s \in S$ such that all entries in the rows of s and t in the OT are equal.

7.1.2 The algorithm

The *L* algorithm* works as follows [4]:

1. Start with $OT = (S = \{\epsilon\}, W = \{\epsilon\}, T)$
2. Fill the table with entries by posing membership queries to the SUL.
3. While the table is not closed, repeat the following:
   (a) Let $t \in S \cdot \Sigma$ be a string such that for all $s \in S$ it holds that $row(s) \neq row(t)$ in the OT.
   (b) Let $S = S \cup \{t\}$.
   (c) Fill the missing entries in the table by posing membership queries to the SUL.
4. Create hypothesis automaton from OT.
5. Pose equivalence query with hypothesis automaton.
6. If there was no counterexample, then the algorithm is finished. Otherwise, if there was a counterexample z, process the counterexample as follows:
   (a) Let a_i be an element in $\{0, 1\}$ that is produced by processing the first i symbols of z with the hypothesis automaton and the remaining with the SUL.
   (b) Let $s_i \in S$ be the state reached when processing the first i symbols of z with the hypothesis automaton.
   (c) Let $z_{>i}$ be the suffix of z that is not processed yet.
   (d) Find the $i_0 \in \{0, 1, ..., |z|\}$ for which $a_{i_0} \neq a_{i_0+1}$.
   (e) Define the new distinguishing string d as $z_{>i_0+1}$.
   (f) Add d to the set of distinguishing strings W.
   (g) Update the missing entries in the OT.
   (h) Return to step 3.

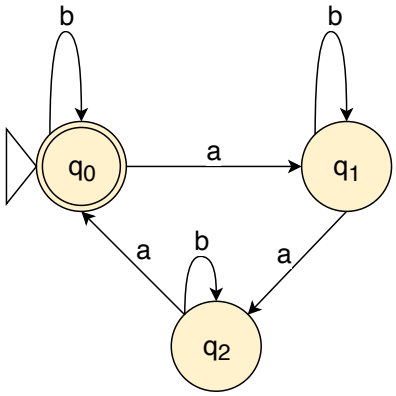


FIGURE 7.1: System Under Learning which accepts all words such that $\{w \in \{a,b\}^* | w.count(a) = 3 * i, i \in \{0,1,...\}\}$ where $w.count(a)$ counts the number of times that a occurs in word w.

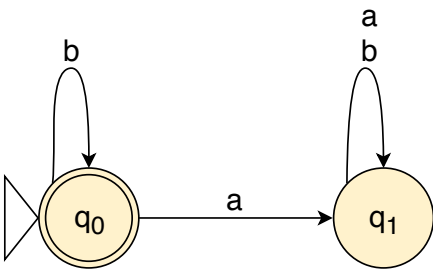


FIGURE 7.2: First hypothesis automaton.

7.1.3 From observation table to automaton

You can construct an automaton from an observation table as follows [4]:

1. For each string $s \in S$, create a state q_s.
2. Set the initial state to q_ε which is the state corresponding to the empty string.
3. For a state q_s and symbol $b \in \Sigma$, add the transition $q_s \xrightarrow{b} q_t$ if and only if $s \cdot b$ is equal to t. More concretely, this means the following: take a string $s \in S$ from the OT. $row(s)$ corresponds to a state q_s. Then consider the one-step extensions of s. Then add a transition with symbol b (the one-step extension) from state q_s to the state q_t, the state that corresponds to $row(s \cdot b)$. Repeat this for all states and their one-step-extensions.

7.1.4 Example of L* algorithm

Finally, we discuss an example to show how the L* algorithm works. For this example, we consider the automaton in Figure 7.1 to be our SUL which has the input language consisting of {a,b}.

	ε
ε	1
a	0
b	1

(A) First OT

	ε
ε	1
a	0
b	1
aa	0
ab	0

(B) Second OT

	ε	a
ε	1	0
a	0	0
b	1	0
aa	0	1
ab	0	0

(C) Third OT

	ε	a
ε	1	0
a	0	0
aa	0	1
b	1	0
aa	0	1
ab	0	0
aaa	1	0
aab	0	1

(D) Fourth OT

TABLE 7.1: Observation tables made when executing the L* algorithm for the SUL (See Figure 7.1)

We start with the following observation table where ε denotes the empty string (See Table 7.1a). All rows beneath the horizontal line in the table are one-step extensions of words in S. All rows above the horizontal line are words in S. The columns denote the suffix-closed distinguishing set W.

The table is not closed since the word "a" is not accepted by the automaton but there is no $s \in S$ such that the row is equal, which would be a 0. Thus we add "a" to S which results in the OT that can be seen in Table 7.1b.

The table is now closed thus we create an hypothesis automaton (See Figure 7.2) from the OT and pose the equivalence query.

The equivalence query will return with a counterexample such as "aaa". "aaa" is accepted by the SUL but not by the hypothesis automaton. a_i is therefore 1. s_i is equal to q_1 in the hypothesis automaton, which is reached when processing "aa". The suffix, that is not yet processed is therefore "a". This is also the distinguishing string, thus we add "a" to W. The resulting OT can be seen in Table 7.1c.

We return to step 3. The table is not closed because there is no $s \in S$ such that $row(aa) = row(s)$. Thus, we add "aa" to S and fill in the missing entries in the OT (see Table 7.1d).

The table is now closed thus we create a new hypothesis automaton, which is the same as the automaton in Figure 7.1. We pose an equivalence query, which does not return a counterexample thus we are done.

7.2 SFT learning algorithm

In this section, we will introduce the SFT learning algorithm that we have developed. We start by introducing some background information. Then, in Sections 7.2.1 and 7.2.2, the algorithm is explained. Finally, we show an example of the SFT learning algorithm in Section 7.2.3.

The SFT learning algorithm, which is an extension of the algorithm described for SFA learning in [4], is based on the L* algorithm. It also uses membership and equivalence queries to ask the SUL questions. The symbolic observation table (SOT) differs compared to the observation table in the original algorithm. The SOT is described by the tuple (S, W, Λ, T, f) where

- $S \subseteq \Sigma^*$ is a set of access strings.
- $W \subseteq \Sigma^*$ is a set of distinguishing strings.
- $\Lambda \subseteq S \cdot \Sigma$ is a set of one-step extensions of S.
- $f : \Sigma^* \times \Sigma^* \rightarrow \Gamma^*$ is a partial function that results in the suffix of the output. The suffix is equal to the output corresponding to sd when we have subtracted the largest common prefix of the output corresponding to s, i.e. $\mathcal{T}_M(sd) - \mathcal{T}_M(s)$ [4] where $\mathcal{T}_M(s)$ represents the transduction function which returns the output of automaton M upon input s.
- $T : \Sigma^* \times \Sigma^* \rightarrow \{IDENTITY, CONSTANT\}^*$ 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 will contain whether it corresponds to an identity function or a constant.

In the symbolic observation table, the set S forms the rows, W forms the columns and T represents all entries in the table. f is used to store the results of all membership queries and to generate the final output functions, also called term functions.

An important design choice in this algorithm is to use the function types $\{IDENTITY, CONSTANT\}$ to identify different outputs. The minimal set of function types that is needed to identify all outputs consists of only the $CONSTANT$ function type. However, in that case, one would need a different transition for each different input. This does not allow us to use predicates effectively to group transitions. Therefore the $IDENTITY$ function type has been added, since this function is used to represent all characters that are not modified. Aside from these two functions, one can also choose to recognize other types such as a type for a function with an offset of one. This would, however, lead to a greater amount of different rows causing the SFT to become larger since each unique row leads to a unique state. Therefore, we have chosen the set consisting of $\{IDENTITY, CONSTANT\}$. When using this set, it is possible that transitions are grouped together in case they output a constant. Therefore, it is important to check whether the grouped transitions actually output a different constant. If so, then this transition will need to be split into two transitions such that both transitions will output a different constant.

Note that "inferring the minimal automaton which is consistent with a set of strings is NP-Hard to approximate even within any polynomial factor."[4]. In practice this means that we will not infer the minimal automaton but simply a correct automaton based on the set of strings given as evidence. This automaton can be incorrect if the set of strings does not contain a counterexample which invalidates this automaton.

7.2.1 The algorithm

1. Start with SOT = $(S = \{\epsilon\}, W\{\epsilon\}, \Lambda = \emptyset, T)$.
2. Fill the table and f with entries by posing membership queries to SUL.
3. While the table is not closed, repeat the following:
   (a) Find the shortest $t \in \Lambda$ such that for all $s \in S$ it holds that $row(s) \neq row(t)$.
   (b) Let $S = S \cup \{t\}$
   (c) If there does not exist a $b \in \Sigma$ such that $t \cdot b \in S \cup \Lambda$, then add $t \cdot b$ to Λ.
   (d) Fill the missing entries in the table and f by posing membership queries to the SUL.
4. Create hypothesis automaton from the SOT.
5. Pose equivalence query with hypothesis automaton.
6. If there was no counterexample, then the algorithm is finished. Otherwise, if there was a counterexample z, process the counterexample as follows:
   (a) Let $i_0 \in \{0, 1, ..., |z| - 1\}$ such that the response of the target machine is different for the strings $s_{i_0} z_{>i_0}$ and $s_{i_0+1} z_{>i_0+1}$.
   (b) Define the distinguishing string d as $z_{>i_0+1}$.
   (c) If $row(s_{i_0} b) = row(s_j)$ when d is added to W for some $j \neq i_0 + 1$ then add $s_{i_0} b$ to Λ. b denotes an arbitrary character in the input language. Otherwise add d to W.
   (d) Update the missing entries in the SOT and f.
   (e) Return to step 3.

7.2.2 From symbolic observation table to automaton

You can construct an automaton from a symbolic observation table as follows:

1. For each string $s \in S$, create a *final* state q_s
2. Set the initial state to q_ϵ, which is the state corresponding to the empty string.
3. For each q_s find its one-step extensions in Λ in the rows of the SOT.
4. For all q_s, call a guard generating algorithm with all one-step extensions of q_s. The guard generating algorithm returns a set of tuples (ϕ, ψ, q) for each q_s.
5. Add transition $q_s \xrightarrow{\phi/\psi} q$ to Δ for all (ϕ, ψ, q).

Final states

In the original SFA learning algorithm, a state was final if the word representing the state, when extended with the empty word, was accepted. In the case of SFTs, we no longer reason about accepting words or not. Thus we cannot use the same reasoning to find the final states. Botinčan and Babić noted the following: "It is unclear how to learn transducers with non-final states, as such transducers allow inherent ambiguity in where the output is produced. Indeed, existing algorithms for learning concrete transducers require all states to be final." [17] We observe that different sanitizer implementations may handle rejecting an input differently, e.g. returning "null" or returning an empty string. There is no straightforward way to discover how a sanitizer acts on an unaccepted string unless it is defined by the user. However, since we are looking into black-box testing, the user would not know how the program acts upon an unacceptable word. Therefore, as in other research [27, 50], we assume that all states of the SFT are final.

Guard generator

The guard generating algorithm will generate the guards for all transitions starting in a state q_s. To generate these guards, it needs a set of evidence and the corresponding output. Evidence is the input character upon which a transition is taken to move to a next state in the automaton.

The algorithm starts with the set of evidence. Then one of the following cases will be followed:

- If *the set of evidence is empty*, generate one transition with the guard *True*. The set of term functions of this transition will consist of only the identity function.
- If *the set contains one piece of evidence*, then one transition will be generated with the guard *True*. The set of term functions will be generated based on the output associated with the evidence. It will generate either the identity function 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 large (sink) transition. This means that all undefined pieces of evidence will be grouped into this sink transition. Finally, for each generated guard, the term generator is called.

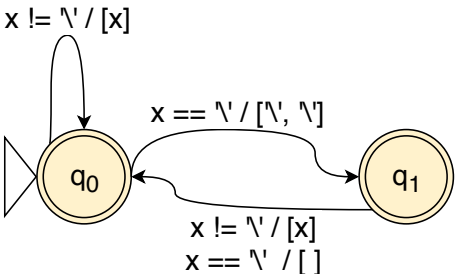


FIGURE 7.3: System Under Learning which escapes all (unescaped) backslashes with a backslash.

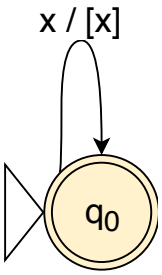


FIGURE 7.4: First hypothesis automaton.

Term generator

The term generator takes a predicate and the starting state as an argument. This predicate is the predicate for which the set of term functions needs to be generated. The starting state q_s is the state where the transition starts. The term generator then works as follows:

1. For all one-step extensions of q_s, calculate the set of term functions as follows:
   (a) Let s be the string that represents the state q_s and $s \cdot b$ represents the state $q_s \cdot b$.
   (b) Compute the suffix of the output such that it is equal to $o_s - o_{s \cdot b}$ where o_s denotes the output of the automaton upon input s.
   (c) For each character in the output, if it is equal to the one-step extension b, then extend the set of term functions with the identity function. Otherwise extend the set of term functions with the constant function b.
2. If the set of term functions are the same for all one-step extensions, then return this set. The algorithm terminates. Otherwise, proceed to the next step.
3. 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. 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.
4. Call the term generator for both generated predicates.

7.2.3 Example of SFT learning algorithm

Finally, we show an example of how the SFT learning algorithm works. For this example, we consider the automaton in Figure 7.3 to be our SUL which has the input language consisting of all unicode characters.

We start with the observation table where ε denotes the empty string. All rows beneath the horizontal line in the table are one-step extensions of words in S. This set is also referred to as Λ. All rows above the horizontal line are words in S. The columns denote the suffix-closed distinguishing set W. The entries in the table are those found in T. Note that $IDENTITY$ is abbreviated to ID and $CONSTANT$ is abbreviated to C. The algorithm proceeds as follows:

- Iteration 1:
  1. The table is closed (see Table 7.2a) therefore we move on to the next step.

	ε
ε	[ID]

(A) First OT

	ε
ε	[ID]
\	[ID, ID]

(B) Second OT

	ε
ε	[ID]
\	[ID, ID]
a	[ID]

(C) Third OT

	ε
ε	[ID]
\	[ID, ID]
\\	[]
a	[ID]

(D) Fourth OT

	ε
ε	[ID]
\	[ID, ID]
\\	[]
a	[ID]
\a	[ID]

(E) Fifth OT

	ε
ε	[ID]
\	[ID, ID]
\\	[]
a	[ID]
\a	[ID]
\\\	[ID, ID]

(F) Sixth OT

	ε
ε	[ID]
\	[ID, ID]
\\	[]
a	[ID]
\a	[ID]
\\\	[ID, ID]
\\a	[ID]

(G) Final OT

TABLE 7.2: Observation tables made when executing the SFT learning algorithm for the SUL (See Figure 7.3)

2. Create the hypothesis automaton (see Figure 7.4).
3. Execute the equivalence query. The equivalence query returns with a counterexample such as "\".
4. Process the counterexample "\" which produces "\\" as output:
   (a) i_0 is 0, s_{i_0} is ε and $s_{i_0}b$ is "\".
   (b) The counterexample will be added to Λ with $T(''\backslash'') = [ID, ID]$.

- Iteration 2:

1. The table is not closed. To close the table, "\" will be moved from Λ to S (see Table 7.2b).
2. Create the hypothesis automaton (see Figure 7.5).
3. Execute the equivalence query. The equivalence query returns with a counterexample such as "a". "a" returns "aa"in the hypothesis automaton whereas the SUL returns "a".
4. Process the counterexample "a" which produces "a" as output:
   (a) i_0 is 0, s_{i_0} is ε and $s_{i_0}b$ is "a".
   (b) The counterexample is added to Λ with $T(''a'') = [ID]$.

- Iteration 3:

1. The table is closed (see Table 7.2c).
2. Create the hypothesis automaton (see Figure 7.6).
3. Execute the equivalence query. The equivalence query returns with a counterexample such as "\\". This counterexample produces "\\\" in the hypothesis automaton whereas the SUL produces "\\".
4. Process the counterexample "\\" which produces the output "\\":
   (a) i_0 is 1, s_{i_0} is "\" and $s_{i_0}b$ is "\\".
   (b) The counterexample is added to Λ with $T(''\backslash\backslash'') = []$.

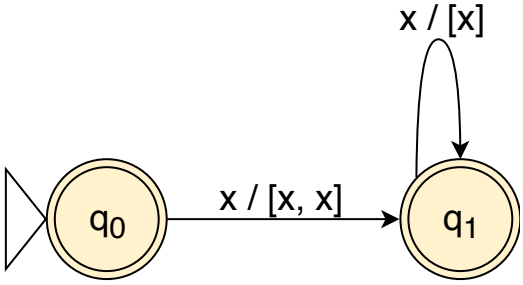


FIGURE 7.5: Second hypothesis automaton.

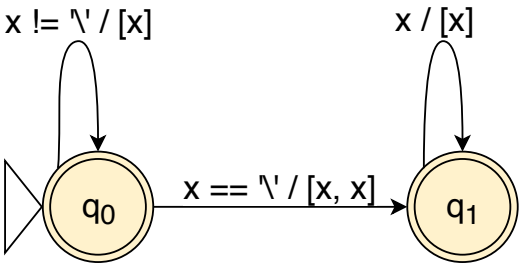
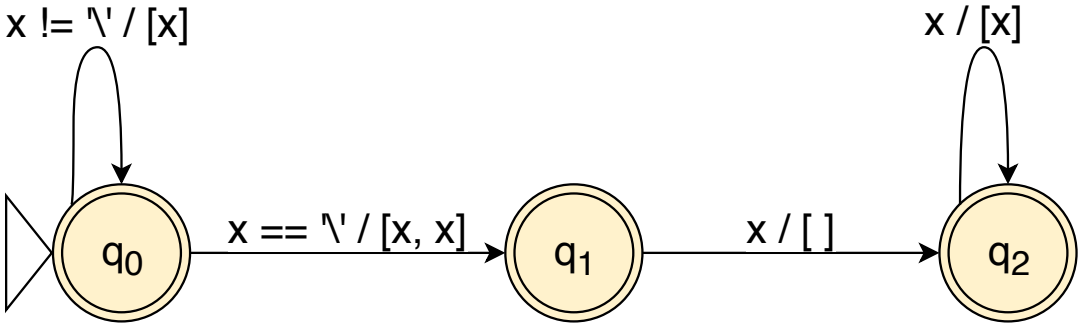
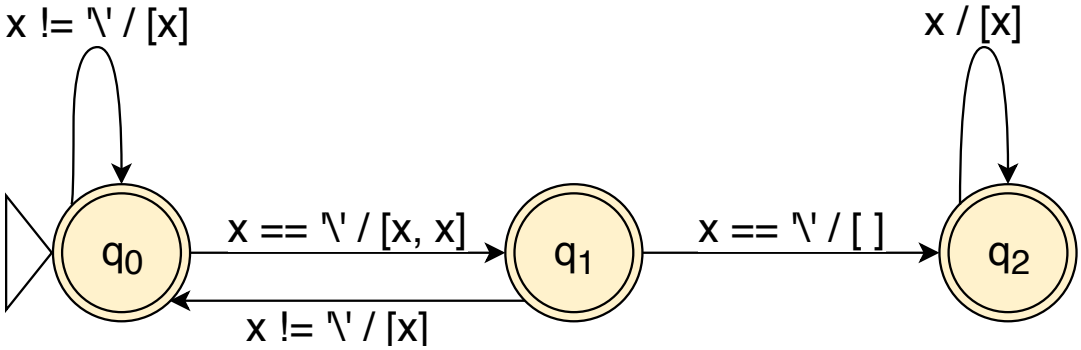


FIGURE 7.6: Third hypothesis automaton.

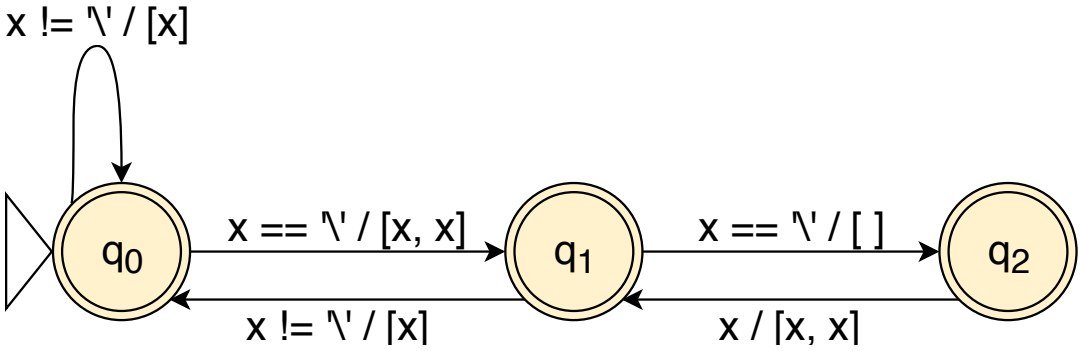
- Iteration 4:
  1. The table is not closed. To close the table, "\\" is moved from Λ to S (see Table 7.2d).
  2. Create the hypothesis automaton (see Figure 7.7a).
  3. Execute the equivalence query. The equivalence query returns with a counterexample such as "\a" which should result in "\\a". The hypothesis automaton, however, returns "\\".
  4. Process the counterexample "\a" which produces the output "\\a":
     (a) i_0 is 1, s_{i_0} is "\" and $s_{i_0}b$ is "\a".
     (b) The counterexample is added to Λ.
- Iteration 5:
  1. The table is closed (see Table 7.2e), thus move to the next step.
  2. Create the hypothesis automaton (see Figure 7.7b).
  3. Execute the equivalence query. The equivalence query will return a counterexample such as "\\\". Upon this input, the hypothesis automaton produces "\\\" whereas the SUL produces "\\\\".
  4. Process the counterexample "\\\" which produces the output "\\\\":
     (a) $i_0 = 2$, s_{i_0} is "\\" and $s_{i_0}b$ is "\\\".
     (b) The counterexample is added to Λ.
- Iteration 6:
  1. The table is closed (see Table 7.2f), thus move to the next step.
  2. Create the hypothesis automaton (see Figure 7.7c).
  3. Execute the equivalence query. The equivalence query will return with a counterexample such as "\\a".
  4. Process the counterexample "\\a" which produces the output "\\a":
     (a) i_0 is 2, s_{i_0} is "\\" and $s_{i_0}b$ is "\\a".
     (b) The counterexample is then added to Λ.
- Iteration 7:
  1. The table is closed (see Table 7.2g). Move to the next step.
  2. Create the hypothesis automaton (see Figure 7.7d).
  3. Execute the equivalence query. No counterexample is returned.
  4. The algorithm terminates.



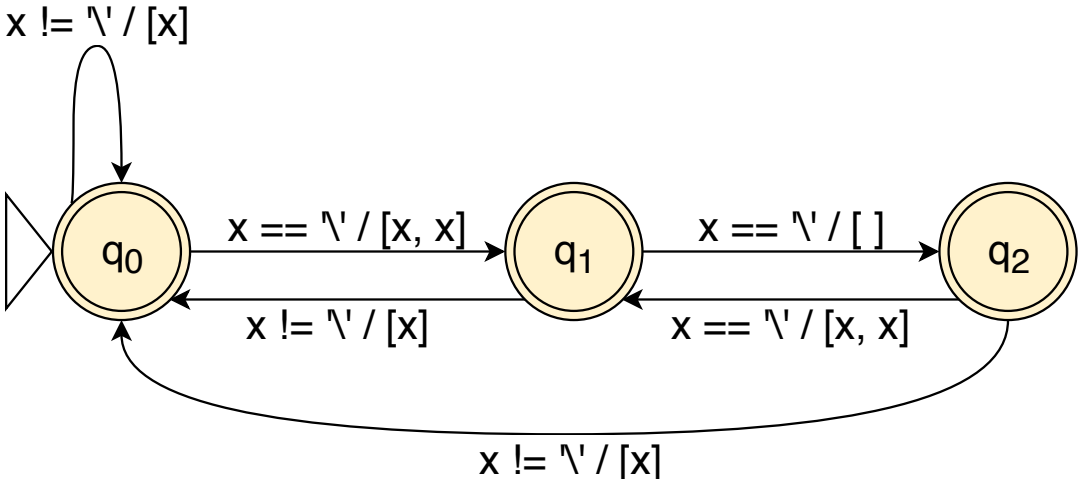
(A) Fourth hypothesis automaton.



(B) Fifth hypothesis automaton.



(C) Sixth hypothesis automaton.



(D) Final automaton.

FIGURE 7.7: Automata generated by the learning algorithm.

7.3 Equivalence Oracle

To conclude this chapter, we discuss how the equivalence oracle can be implemented. There are many different ways in which an equivalence oracle can be implemented. We consider random testing, random prefix selection, history-based testing and automaton-based testing. Random testing has been chosen as it is an approach that is easy to understand and implement. Automaton-based testing has been chosen because it allows us to choose tests cleverly based on the structure of the hypothesis automaton. Finally, random prefix selection has been chosen because random testing sometimes has difficulties finding transitions in some states [51].

7.3.1 Random testing

The first and easiest approach is random testing[52]. In case of random testing, we need to generate random strings of a specified length consisting of characters in the alphabet. This can either be done completely random or one can choose random transitions in the learned automaton and generate a character that satisfies the guard of the chosen transition.

7.3.2 Random prefix selection

The second approach is called random prefix selection. When generating test cases completely at random, states with a shorter access string are visited more frequently [51]. To make sure that each state has the same chance of being visited, we randomly choose a state from the automaton. The access string of this state will form the prefix of the test case. The rest of the test case is generated randomly and appended to the prefix.

7.3.3 Automaton-based testing

Another approach is testing based on the structure of the hypothesis automaton. You can generate test cases such that you achieve one of the following conditions [53]:

- State coverage which means that each state must be visited at least once.
- Transition (or branch) coverage which means that each transition must be taken at least once.
- Predicate (or condition) coverage which means that each predicate, also sub-predicates, must be satisfied at least once.

Note that there are more possible conditions, such as multiple-condition coverage. However, we only consider these options due to the scope of this research. Test cases are generated such that the chosen coverage condition will be satisfied. The user needs to specify how many test cases should be generated per state, transition or predicate.

Chapter 8

Results

This chapter presents the results of experiments with the learning algorithm as well as some details of the implementation. We test whether correct automata are learned, whether we can check specifications and the performance of the algorithm. Any limitations or threats to validity of these tests are discussed in the next chapter.

8.1 SFTLearning

This section discusses some details on how the oracles for the learning algorithm have been implemented. The implementation of the algorithm is based on `symbolicautomata` [54] which is a Java library for symbolic automata. We refer to the developed tool as `SFTLearning`. SFTLearning includes a learning algorithm for SFAs, a learning algorithm for SFTs, implementations for equivalence oracles and the ability to read and check specifications. SFTLearning can be found at:

`https://github.com/Sophietje/SFTLearning`

8.1.1 Oracles

Membership oracle

Since we are using a black-box approach, meaning that we are unable to see the inner workings of the sanitizer, we need to be able to give an input to the sanitizer and read the sanitizer's output. To be able to use the program as a membership oracle, there needs to be a connection between the algorithm and the membership oracle. Therefore the user is required to write a program which will execute the sanitizer with a given commandline argument. To execute these programs, we use a ProcessBuilder in Java. This allows us to execute a process and read the output from the standard output. Moreover, it is independent of which programming language is used.

Equivalence oracle

The equivalence oracle is crucial for the learning algorithm. If it does not provide the correct counterexamples, then the algorithm will be unable to learn the correct model of the program. The user needs to specify a range from which input characters are generated for the equivalence oracle. If this range is too small, then an equivalence oracle may not be able to discover some counterexamples. Note that it is impossible to test all possible strings. Therefore the amount of generated test cases is limited by a maximum which the user can define. There are many different ways in which an equivalence oracle can be implemented. The following approaches (see Section 7.3 for details) have been implemented in SFTLearning:

- Random testing
- Random prefix selection
- State coverage
- Transition (or branch) coverage
- Predicate (or condition) coverage
- History-based testing

8.2 Performance

8.2.1 Equivalence Oracle

To test the performance of the SFTLearning algorithm, we needed to decide which equivalence oracle to use. To test which one derived a correct model of the sanitizer in the least amount of time, we compare the following equivalence oracles:

- Random testing with 30,000 tests
- Random transition testing with 10,000 tests
- Random prefix selection with 10,000 tests
- State coverage with 3,000 tests per state
- Transition coverage with 2,000 tests per transition
- Predicate coverage with 2,000 tests per (sub-)predicate

All processes had a time-out set to 30 minutes. If no model was found within this time, then the learning process was terminated. The input characters have been chosen from the range 1 to 400 which refer to the decimal representations of Unicode characters. This range includes all control characters (except null), Basic Latin, the Latin-1 supplement, Latin Extended-A and part of Latin Extended-B [55]. The number of tests for random testing has been chosen such that it resulted in mostly correct models for the encode function. The other variables, such as tests per state, tests per transitions, etcetera, have been deduced from that number. The encode sanitizer is represented by three states when learned. Therefore random transition testing and random prefix selection use $10,000$ tests which leads to roughly the same amount of tests since $10,000 \times 3 = 30,000$. Next, assuming that an automaton is typically represented by 10 states or less, we use $3,000$ tests for each state when using state coverage. Finally, the encode function has 3 states with each 4 transitions when it is correctly learned. We round this number such that there are $3 \times 4 = 12 \approx 15$ transitions in the automaton. Combined with the assumption that each transition typically has a single predicate, this leads to $2,000 \times 15 \approx 30,000$ tests in total.

We test these oracles on two different sanitizers namely:

- *Encode*: encodes <, > and & into their HTML entities
- *Escape*: escapes backslashes with another backslash unless it was already escaped

Sanitizer	Random		Random transition		Random prefix selection	
	# Correct	Time (s)	# Correct	Time (s)	# Correct	Time (s)
Encode	7/10	60.4	10/10	109.9	10/10	571.5
Escape	0/10	25.4	0/10	11.1	1/10	1351.2

Sanitizer	State coverage		Transition coverage		Predicate coverage	
	# Correct	Time (s)	# Correct	Time (s)	# Correct	Time (s)
Encode	10/10	67.3	8/10	60.8	10/10	119.4
Escape	0/10	1038.6	2/10	221.2	1/10	329

TABLE 8.1: Results of comparing different implementations of the equivalence oracle.

Each oracle is used to learn each sanitizer ten times. We then calculate the average running time which we will use to compare different approaches. The final equivalence oracle which is used to judge the performance of the learning algorithm, will be the one who produces the most correct models in the least amount of time.

The tests have been performed within a VM running Debian 7 on a Dell PowerEdge R420 with two Intel Xeon E5-2420 CPUs and 4GB of memory available. The results of the tests can be seen in Table 8.1. From this we conclude that predicate coverage is the best approach since this results in the most correct models, 11 of 20 models were correct, in the least amount of time.

8.2.2 SFTLearning Algorithm

We want to discover the limitations of the SFTLearning algorithm. Therefore, we have learned models for the following methods, in order to discover which types of models can be learned using the SFTLearning algorithm:

1. Encode (from the he project [56])
2. Escape (from cgi python module [57])
3. Escape (from the escape-string-regexp project [58])
4. Escape (from the escape-goat project [59])
5. Unescape (from the escape-goat project [59])
6. Unescape (from the postrank-uri project [60])
7. To Lowercase (from the CyberChef project [61])
8. htmlspecialchars (built-in PHP function [62])
9. filter_sanitize_email (built-in PHP filter [63])
10. Remove tags (from the govalidator project [64])

These implementations have been found by searching on GitHub for the keywords "escape", "encode" and "sanitize". Then the top 20 results, sorted by "Most stars", have been chosen for each search term. Next, the repostories have been filtered so that there were only string sanitizers 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.

In Appendix B, a complete list of the repositories can be found as well as the reason why some have not been used. Aside from the above mentioned sanitizers, three other sanitizers have also been tested of which two have been written in PHP and one in Python. This has been done so that we test sanitizers that have been written in different languages. Including these, the sanitizers are either written in JavaScript, Ruby, Go, PHP or Python.

We have performed some test runs using a fully automatic approach, however all test runs did not finish within three hours. Note that although it was not completely finished, the SFTLearning algorithm was able to derive an almost completely correct model within three hours. Therefore we conclude that the fully automatic approach is not yet viable in its current state. As a result, we have chosen to serve as an equivalence oracle ourselves instead of using an automatic equivalence oracle. While this may influence the quality of the model that is learned, it does not affect what kinds of models we can learn. These tests have been run on a MacBook Air running Darwin 17.7.0 with an Intel Core i5 (2 cores) and 4GB of memory available. The results can be found in Table 8.2.

Overall we can divide sanitizers into two main categories:

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

From the results, we can conclude that we can learn models of all sanitizers which act based on occurrences of *single* characters. Sanitizers which act based on sequences of characters cannot yet be learned. This is also what we expected based on the theory since we would need lookback, lookahead or registers to be able to learn those models. Note that if an input would always end with a specific character, then we should be able to learn some sanitizers whose behaviour depends on a sequence of characters. Adapting the SFTLearning algorithm such that it can derive models with lookback, lookahead or registers is left as future work.

8.3 Scalability

The SFTLearning algorithm is able to represent models of sanitizers with very large input alphabets. In order to find out how the SFTLearning algorithm scales, multiple tests have been conducted where the size of the input alphabet becomes larger. This will show whether the SFTLearning algorithm is still viable to use when reasoning about larger input alphabets.

To test how the algorithm scales, we learn the model of a sanitizer that removes the character "<" from any input. As an equivalence oracle, we use the predicate coverage oracle which seemed to be the best performing oracle as found in Section 8.1.1.

Two variables have been changed during these tests. The first variable is the size of the input alphabet which is used to derive counterexamples from. The second variable is the number of tests per predicate. When increasing the size of the input alphabet, the chance to find the one character which is modified by the sanitizer becomes smaller. And, when increasing the number of tests per predicate, the chance to find the one character which is modified by the sanitizer becomes larger.

These tests have been run on a MacBook Air running Darwin 17.7.0 with an Intel Core i5 (2 cores) and 4GB of memory available. The result has been measured in terms of total running time (in seconds) (see Figure 8.1), number of membership

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?
Encode (he)	612	6206	464293	51	38	3	1	22	7	Yes [a]
Escape (cgi)	199	1958	160907	16	16	3	1	13	4	Yes[a]
Escape (escape-string-regexp)	1379	19358	1233774	113	87	3	1	14	2	Yes
Escape (escape-goat)	415	6724	388324	36	36	31	1	5	1	Yes
Unescape (escape-goat)	521	25478	482746	202	23	14	-	22	-	No
Unescape (postrank-uri)	381	52114	311917	178	47	7	-	40	-	No
To Lowercase (CyberChef)	1109	185645	691777	130	105	2	1	54	27	Yes[a]
htmlspecialchars (php)	414	4708	354000	27	27	4	1	20	5	Yes
filter_sanitize_email (php)	776	10340	731765	69	69	3	1	13	2	Yes[a]
Remove tags (govalidator)	723	131102	587348	184	40	8	-	18	-	No

TABLE 8.2: Results of the tests for the learning algorithm. From this we can derive what kind of models can be learned using the SFTLearning algorithm.

[a] This model has been correctly derived for the Basic Latin alphabet.

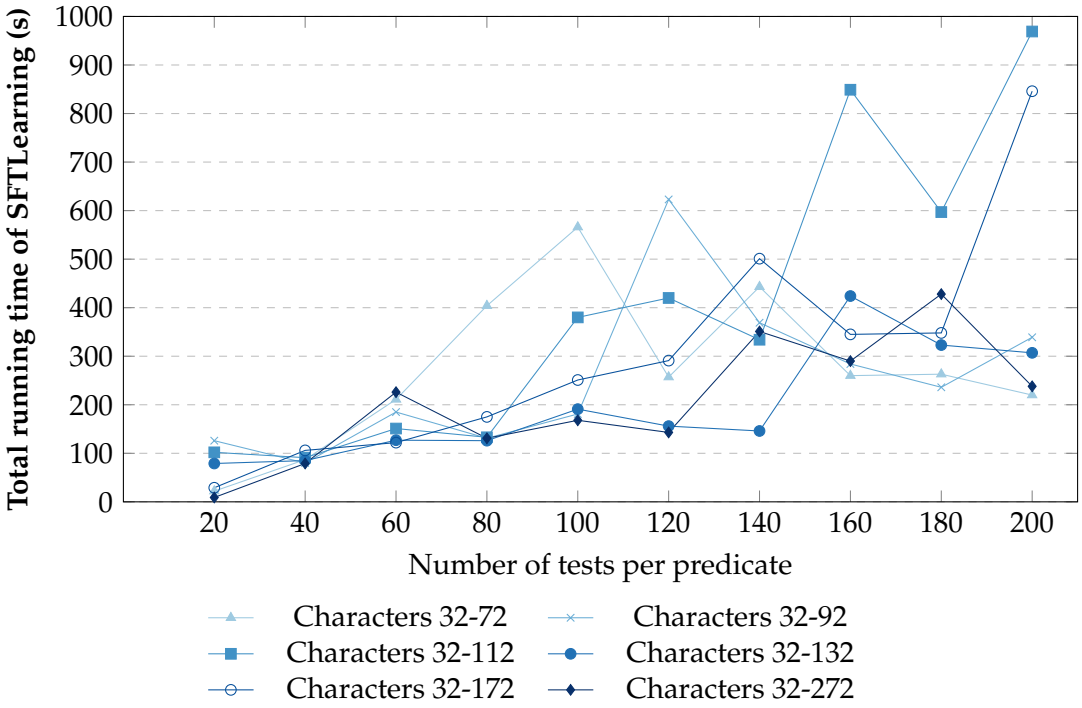


FIGURE 8.1: This graph shows the effect of the number of tests per predicate on the total running time of the SFTLearning algorithm.

queries posed (see Figure 8.2) and number of equivalence queries posed (see Figure 8.3). Note that many of the derived models were not correct. Also, the equivalence oracle uses membership queries to check whether something is a counterexample. Therefore, there will always be an equal or larger amount of membership queries than equivalence queries. This was due to the limited number of counterexamples which were considered. Overall, the approach seems to scale linearly.

In Figure 8.4, you can see how much of the total running time is spent in the equivalence oracle. This shows that a large amount of time is spent in the equivalence oracle. Therefore, we can conclude that implementing a more efficient equivalence oracle can greatly reduce the total running time of the SFTLearning algorithm. The equivalence oracle could be improved in two main ways:

- *Improve the efficiency of the equivalence oracle.* For example by presenting minimal counterexamples or using another approach to find counterexamples.
- *Improve the efficiency of the membership oracle.* This can be done by implementing a membership oracle which uses only one process to which all membership queries are posed instead of using a single process for each membership query.

8.4 Case study

For the final tests, we show how our approach can be used. We have reused the models that have been correctly learned in the tests for the learning algorithm. This includes the following methods:

1. Encode (from the he project)
2. Escape (from the cgi package in Python)

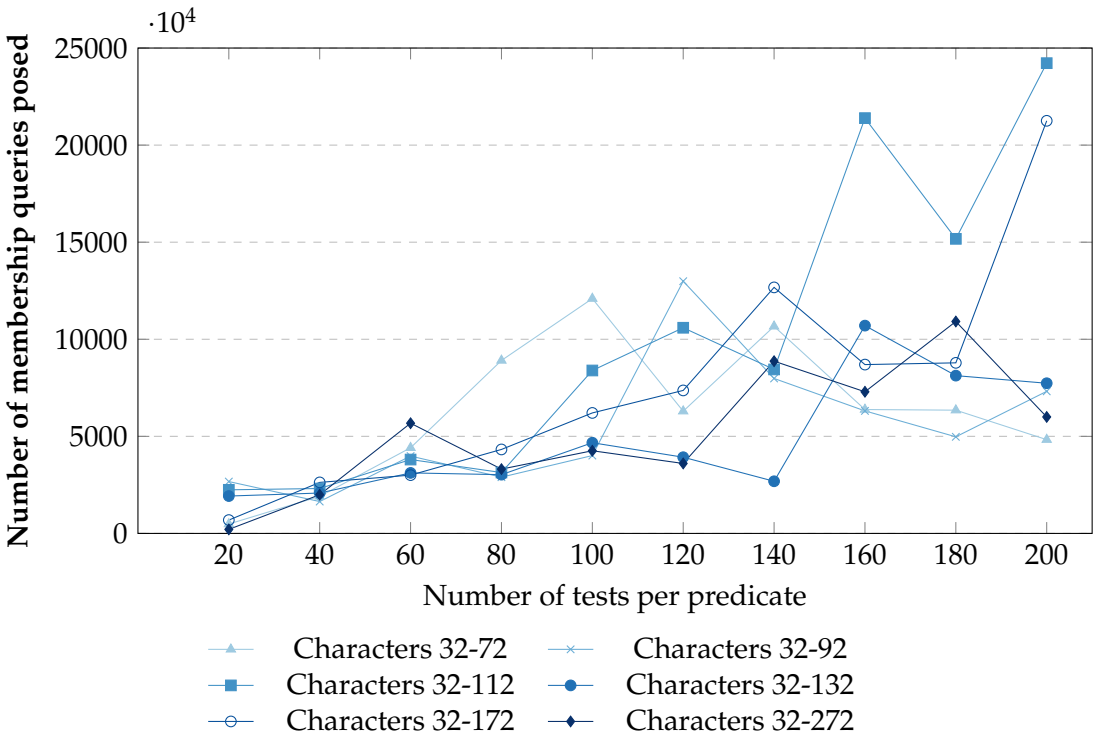


FIGURE 8.2: The effect of the number of tests per predicate on the number of membership queries posed when using the SFTLearning algorithm.

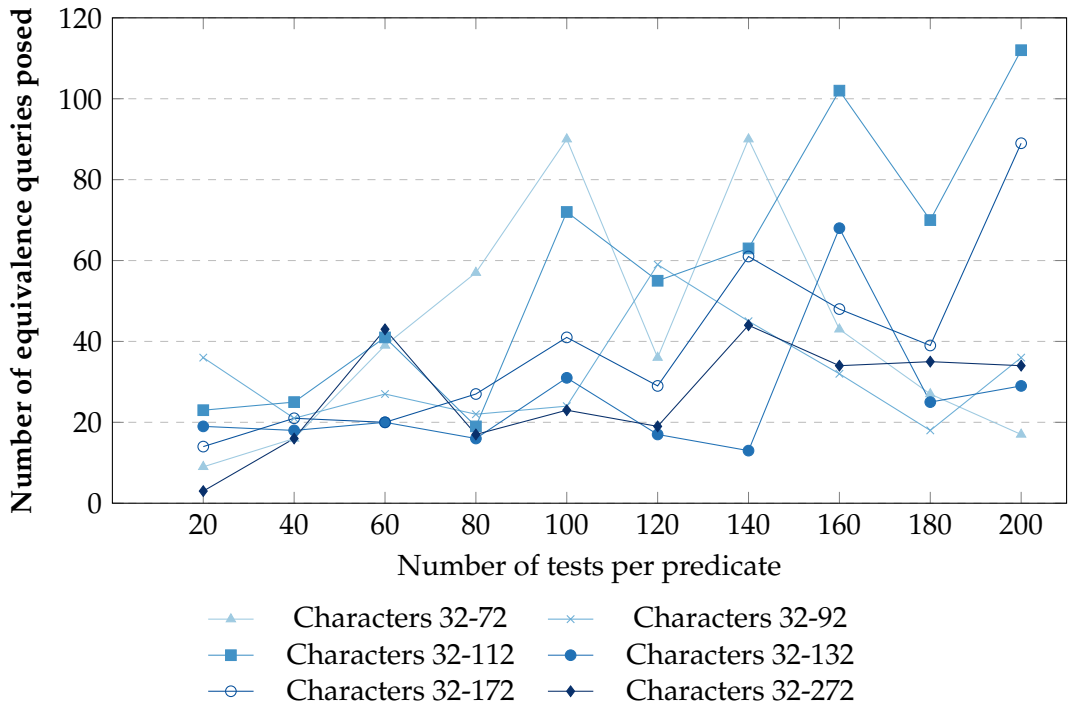


FIGURE 8.3: The effect of the number of tests per predicate on the number of equivalence queries posed when using the SFTLearning algorithm.

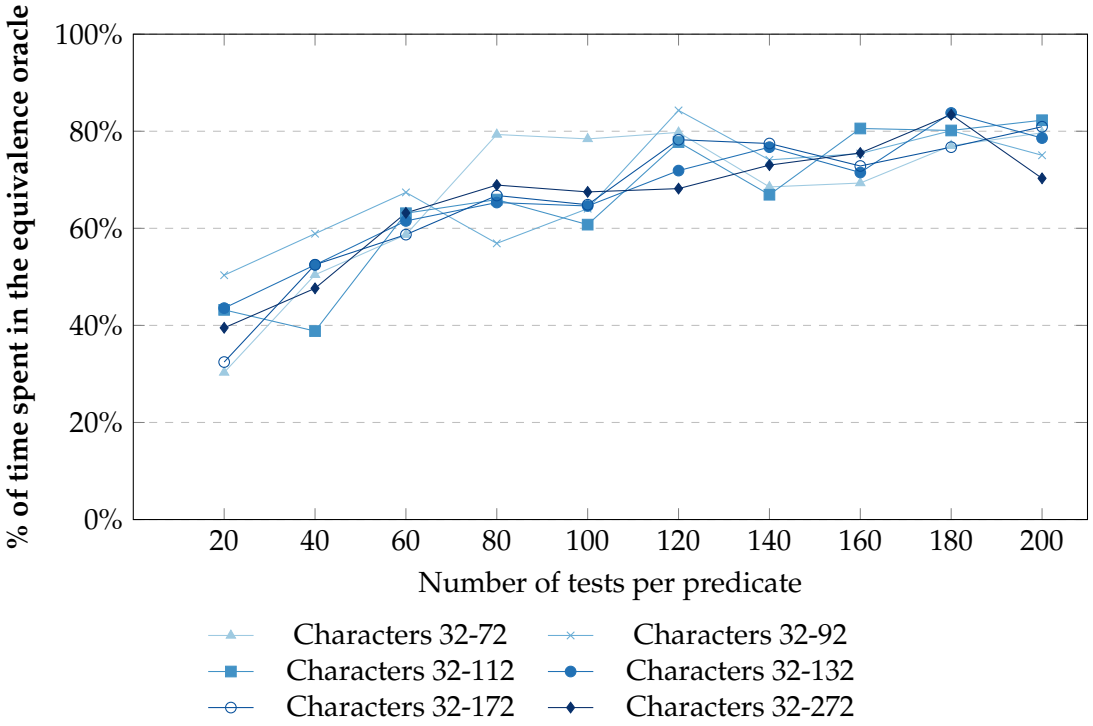


FIGURE 8.4: The effect of the number of tests per predicate on the time spent in the equivalence queries posed when using the SFTLearning algorithm.

3. Escape (from the escape-string-regexp project)
4. Escape (from the escape-goat project)
5. toLowercase (from the CyberChef project)
6. htmlspecialchars (a built-in function in PHP)
7. filter_sanitize_email (a built-in function in PHP)

For each of these methods we have asked the following questions:

- Is it idempotent?
- Are there any sanitizers that should behave the same, and if so, is their behaviour the same on all inputs?
- Does it commute with the other learned models?
- Are there any discrepancies compared to the user's specification of the sanitizer?

In Table 8.3, you can find which learned models were expected to be idempotent as well as the corresponding computed result which indicates whether the learned model was idempotent. The results matched the expected values. In Table 8.4, the computed results can be found which indicate whether the learned models commute with each other. Some of the commutativity results could not be computed. This was because the garbage collector exceeded the overhead limit when computing the composition of two sanitizer functions. This may be prevented by using a computer

Sanitizer	Expectation	Computed
encode (he)	×	×
escape (cgi)	×	×
escape (escape-string-regexp)	×	×
escape (escape-goat)	×	×
toLowercase (CyberChef)	✓	✓
htmlspecialchars (php)	×	×
filter_sanitize_email (php)	✓	✓

TABLE 8.3: This table shows whether a sanitizer was expected to be idempotent and the computed results.

	1	2	3	4	5	6	7
1	✓	×	✓	×		×	×
2	×	✓	✓	×		×	×
3	✓	✓	✓	✓	✓	✓	×
4	×	×	✓	✓		×	×
5			✓		✓		✓
6	×	×	✓	×		✓	×
7	×	×	×	×	✓	×	✓

TABLE 8.4: This table shows which learned models from sanitizers commute with each other. The numbers correspond to sanitizers as mentioned in the beginning of Section 8.4.

with more memory available. Another possible solution would be to minimize the learned model before computing the composition.

None of the learned models should behave exactly the same. Therefore we cannot compare the behaviour of similar models.

Finally, we have written a specification for each learned model. Each specification precisely describes the behaviour of the sanitizer (see Appendix C for all specifications). These specifications have been written before the model was derived to make sure that these specifications were not influenced by the structure of the models that were derived. At first, the escape model from the escape-string-regexp project was not correct. After some investigation, an error in one of the guards on a transition in the specification was found and fixed. After this, all specifications were equivalent to the learned model.

Chapter 9

Discussion

In this chapter, we discuss the results which have been presented in the previous chapter, as well as the limitations of the tests discussed in the previous chapter.

9.1 Implementation of Membership Oracle

The membership oracle is implemented by executing a user given command. This enables the user to apply fine-grained configuration. However, it also has some drawbacks. Firstly, when reading output from the standard output, we read one line. As a result, newlines and carriage returns are not dealt with properly. This is a possible point of improvement for further research. Secondly, it is impossible to give a null character as an argument. Therefore, we are unable to reason about the behaviour of a sanitizer given a null character.

Also, currently a new process is started each time a membership query needs to be executed. This could be optimised if we start the process once and keep executing new queries. This can reduce the overhead of starting a new process each time. Note that this would require a program that keeps reading new input from the standard input and sanitizing this instead of a program that takes one argument which it will sanitize.

9.2 Performance of Equivalence Oracle

We have chosen a best performing equivalence oracle based on the performance of the oracle on two sanitizers. This is prone to bias since one approach for the equivalence oracle may work very well on these two sanitizers whereas it may perform very poorly on others. Moreover, the number of tests for each approach has been derived based on several assumptions as well as the assumption of 30,000 tests for random testing. If the assumption for random testing or any of the other assumptions are incorrect, then we have chosen a number of tests which is likely too large or too little. This would influence whether the approach can derive a correct model and how much time this may take. Therefore we stress that we have compared the different approaches with the specified number of tests. When using different numbers of tests for the approaches, then one may conclude that another approach is actually the best. We are aware of these discrepancies and therefore we encourage further research into what approach for an equivalence oracle performs best in practice on many sanitizers.

9.3 SFTLearning Algorithm

The SFTLearning algorithm experiment has shown what type of models can be learned using the SFTLearning algorithm. These results were as expected namely we can currently only learn models which reason about single character occurrences. This can be improved by extending the algorithm such that we can learn SFTs with registers, lookback or lookahead. If such an extension is made, then we will also be able to reason about sanitizer whose behaviour depends on the occurrence of multiple characters.

The implementation of the equivalence oracle strongly influences the time needed to derive a model. The SFTLearning algorithm works best when presented with a minimal counterexample. This has been achieved by acting as an equivalence oracle ourselves. Before deriving these models, we had already written specifications which describe how the sanitizers acts. As a result, it is possible that we try to derive this model that we have in mind. However, this does not influence what kind of types of models we can derive.

Next, it should be noted that for some models we only considered the Basic Latin alphabet as the input alphabet. Since we used a manual equivalence oracle, deriving models from very large input alphabets becomes difficult and time-consuming. Therefore, we have chosen to derive some of the larger models only for a limited input alphabet. As a result, the behaviour of some sanitizers is only partly described. For example, the "toLowercase" method only modifies the characters A-Z in the learned model. Whereas this method also modifies characters such as Á.

Finally, one should take into account that we only considered sanitizers for which we could write the specification in at most 10 minutes. As a result, these tests may not accurately show the ability of the algorithm to reason about more complex sanitizers.

9.4 Scalability

The scalability tests seem to indicate a linear approach when performing more tests per predicate. When performing more tests per predicate, the confidence in the correctness of a model increases. This is because performing more tests per predicate, increases the chance of finding a counterexample. Thus the cost, in this case time, seems to steadily increase when performing more tests per predicate. This indicates that deriving a model, of which we are reasonably confident that it is correct, is possible in a limited amount of time. Note that, our tests may be biased since it is possible that the algorithm starts showing exponential behaviour when reasoning about more tests per predicate. Also, the graphs show the results of one run due to which outliers strongly influence the results. It would have been better to run the algorithm multiple times and take the average of these results, however this was not possible due to the limited scope of this research.

Finally, we note tht the amount of time needed to derive a model, when reasoning about larger input alphabets, is similar to the time that is needed when reasoning about smaller alphabets. This is promising since it indicates that we can use SFTLearning to reason about large input alphabets.

9.5 Case study

The case study has been done to show how our approach can be applied. This case study has been done on the sanitizers which were derived using a manual equivalence oracle. As said before, this may have introduced a bias since we then tend to derive the model that we have in mind.

Moreover, one should note that some compositions could not be computed when reasoning about a larger model. This indicates that some specification checks may not be feasible when reasoning about more complex sanitizers, regardless of whether the sanitizer's model is minimized.

This test has shown that we are able to derive correct models from real-world sanitizers and that we can use these models to reason about the correctness of the sanitizer's implementation.

9.6 Overall limitations

Overall, the SFTLearning algorithm has two main limitations:

- Implementation of the equivalence oracle
- Execution of a membership query

Firstly, as stated before, the equivalence oracle strongly influences what model is derived from the sanitizer. The algorithm works best when it is presented with minimal counterexamples. At this point, we are unsure of what a best implementation for an equivalence oracle would be. However, when presented with enough time and if the performance of the SFTLearning algorithm is improved, then we are able to reason about real-world sanitizers using the SFTLearning algorithm.

Secondly, a new process is started each time a membership query needs to be executed. This introduces overhead which can be avoided by constructing a program which continuously observes input and outputs corresponding results. Since many equivalence oracles use a significant amount of membership queries to find counterexamples, this will very likely improve the performance of the algorithm.

It is important to note that an automatic equivalence oracle is not yet a viable approach depending on the time limit. It took more than 3 hours to derive a model. One should, however, know that after 3 hours, the algorithm had derived an almost fully correct model of the sanitizer. Therefore, if time is not a limiting constraint, then the SFTLearning algorithm is a viable approach.

Chapter 10

Conclusion

In this research, we wanted to find a way to reason about the correctness of sanitizers in a black-box manner. Firstly, we gathered different types of specifications that one might want to write for a sanitizer by discussing this with security experts.

Next, we determined what would be necessary to prove these specifications. Before this research, we were only able to derive models which can reason about the correctness of the input or output language of a sanitizer. We noticed that many specifications require us to reason about the relation between the input and the output language. Therefore, we have presented a new algorithm, SFTLearning, which derives Symbolic Finite Transducers of existing sanitizers. These models allow us to reason about the relation between the input and the output language. The models are derived in a black-box manner and can be used to reason about the correctness of sanitizers by comparing them to specifications. The user only needs to write specifications which describe how the sanitizer should (not) work. No work is needed in terms of rewriting or annotating the sanitizer in order to reason about its correctness. The SFTLearning algorithm has been implemented and is freely available.

Then, we evaluated the SFTLearning algorithm on real-world sanitizers. Our tests have shown that we can learn many models within 15 minutes when using a near-perfect (human) equivalence oracle. The models can be derived completely automatically with SFTLearning when given enough time and an efficient equivalence oracle. The SFTLearning algorithm can derive correct models of all sanitizers whose behaviour depends on the occurrence of single characters. Finally, our tests have also shown that it is able to reason about large input alphabets (up to 240 characters) without incurring high costs.

10.1 Future Work

For those who have an interest in pursuing this research, below you can find several options to consider.

10.1.1 Additional types of sanitizers

In this section, you can find possible future work which will extend what types of models can be learned.

- *Develop an SFT learning algorithm with lookahead, lookback or registers*. This would allow the algorithm to learn more types of sanitizers. For example, a sanitizer that removes a script tag and its corresponding closing tag, or a sanitizer that encodes characters into their HTML entities unless they are already part of an HTML encoded entity.

- *Add the learning of epsilon transitions.* Currently no epsilon transitions can be derived. As a result, some sanitizers cannot be learned. Epsilon transitions are already supported by the library which is used to represent the models. One would need to discover when an epsilon transition should be added to the automaton.

- *Implement the algorithm for automata that reason about other alphabets*, such as integers. The current implementation is only able to reason about character intervals.

10.1.2 Improving SFTLearning

In this section, you can find future work which aims to improve the existing approach that is presented in this paper.

- *Use length encoded input/output for membership oracle.* This will allow us to start a program only once and keep asking queries instead of restarting the membership oracle each time a query is sent. This should reduce the overhead of the membership oracle thus improving the performance of the learning algorithm.

- *Evaluate equivalence oracles for learning sanitizers.* Finding an optimal equivalence oracle can significantly improve the performance of the learning approach. One promising option is to look into the work of Aichernig and Tappler [65] who developed an equivalence oracle based on model-based mutation testing. They try to minimise the number of queries needed for equivalence queries. Instead of considering different ways to generate test cases, it can also be profitable to look into ways to select test cases. If we can efficiently select test cases, then this can also reduce the number of test cases that are needed.

- *Develop equivalence oracle optimized for learning sanitizers.* Instead of evaluating existing equivalence oracles, one can choose to develop an equivalence oracle which is optimized for learning sanitizers. For example, we propose a "history-based" approach which chooses characters based on behaviour in previous states. The intuition behind the approach is that a character is likely to produce unique behaviour if it did so earlier on in the automaton. For the initial state, there is no previous state to use as a basis to choose input, therefore only random inputs are tested. For all other states, we find all states which have transitions that lead to the current state, i.e. the neighbouring states. Then for each transition starting in a neighbouring state, randomly generate a test case such that this transition is taken. This randomly generated test case is appended to a prefix which is the access string to the current state. Each transition is tested a maximum number of times which is specified by the user. The user can also specify the maximum length of all randomly generated test cases. This approach has already been implemented in SFTLearning, however it has not yet been evaluated.

- *Optimize the learning algorithm.* Optimize how many membership and equivalence queries are posed in order to improve the performance of the learning algorithm.

- *Look into generating other types of functions* from the symbolic observation table, not only the identity or constant functions. One possible approach would be

to fill the table based on offsets and constants. Then select groups based on the most common function in the table. This would allow the algorithm to deduce smaller automaton from some programs.

- *Write a correctness proof for the algorithm.* Due to the limited scope of this research, we were unable to write a correctness proof of the algorithm. Although it seems to work, it would be better to support this claim by writing a correctness proof of the SFT learning algorithm.

10.1.3 User-friendliness

Finally, in this section, we show several options for future work which aim to improve the usability of the SFTLearning algorithm.

- *Automatically generate specifications based on user input.* This will require less user input and thus make the approach more suitable for automatization.
- *Find more types of specifications which users would want to test.* The tool can then be extended to include the possibility to check these specifications.
- *Let users evaluate learned models.* Instead of asking the user to write a specification, one can choose to let the user act as a final equivalence query for learned models. You show the user the learned model and ask whether it is correct. In this case, it would be best to show the user a minimized SFT which is easier for a user to check. Saarikivi and Veanes have already developed an algorithm for minimizing deterministic symbolic finite transducers [66].

Appendix A

List of Specifications

Below you can find a list of all types of specifications that have been suggested during a brainstorming session with employees of Northwave [47].

"I want to be able to know...

- whether a specific input (character) can occur in the output.
- whether a specific input (character) cannot occur in the output.
- whether a sanitizer uses blacklisting or whitelisting.
- how a sanitizer does (not) change a specified character set.
- what happens to input that is not specified as part of the input language.
- whether a specified encoding is accepted by a sanitizer.
- whether a validly structured input document also results in a validly structured output document.
- whether the structure of the input document is (not) changed.
- what happens to an invalidly structured document that is given to a sanitizer as input.
- whether (a part of) the input is *not* modified.
- which interesting (such as specific HTML tags) things can occur in the output.
- whether the sanitizer also removes the values of removed HTML tags.
- what happens to special characters such as invisible characters, left-to-right mark, control characters, emoticons, etcetera.
- the possible (minimal and maximal) lengths of the ouput.
- whether a known vulnerability (specific sequence of characters) can occur in the output.
- what happens to input that is very similar to bad input.
- which inputs lead to a specific bad output.
- whether two sanitizers are the same.
- whether it matters in which order two sanitizers are called.
- whether it matters if a sanitizer is called multiple times.

Appendix B

List of repositories

This appendix describes all repositories on GitHub that have been considered for learning models. These repositories have been found using the global search for repositories on GitHub with the keywords "encode", "escape" and "sanitize". Below you can find a table which describes the top 20 results for each keyword sorted by "Most stars". In each table you can find the link to the repository as well as the reason why it was not used (if this is the case).

B.1 Encode

This table reflects the order of the results as it was on the 6th of July 2018.

Repository link	Reason for rejection
https://github.com/Blankj/AndroidUtilCode	Could not get the Android SDK to work
https://github.com/google/guetzli	Not a string sanitizer
https://github.com/leandromoreira/digital_video_introduction	Not a string sanitizer
https://github.com/google/seq2seq	Not a string sanitizer
https://github.com/gchq/CyberChef	-
https://github.com/topojson/topojson	Not a string sanitizer
https://github.com/json-iterator/go	Not a string sanitizer
https://github.com/mozilla/mozjpeg	Not a string sanitizer
https://github.com/jnordberg/gif.js	Not a string sanitizer
https://github.com/monochromegane/the_platinum_searcher	Not a string sanitizer
https://github.com/esnme/ultrajson	Not a string sanitizer
https://github.com/ashtuchkin/iconv-lite	Not a string sanitizer
https://github.com/akheron/jansson	Not a string sanitizer
https://github.com/ryankiros/skip-thoughts	Not a string sanitizer
https://github.com/mathiasbynens/he	-
https://github.com/mikechambers/as3corelib	Not a string sanitizer
https://github.com/knrz/CSV.js	Not a string sanitizer
https://github.com/sublimehq/anim_encoder	Not a string sanitizer
https://github.com/real-logic/simple-binary-encoding	Not a string sanitizer
https://github.com/mpdf/mpdf	Not a string sanitizer

B.2 Escape

This table reflects the order of the results as it was on the 9th of July.

Repository link	Reason for rejection
https://github.com/mozilla/bleach	Too complicated to write specification[a]
https://github.com/fazibear/colorize	Not a string sanitizer
https://github.com/klange/nyancat	Not a string sanitizer
https://github.com/adoxa/ansicon	Not a string sanitizer
https://github.com/unamer/vmware_escape	Not a string sanitizer
https://github.com/jorgebucaran/turbocolor	Not a string sanitizer
https://github.com/fusesource/jansi	Not a string sanitizer
https://github.com/brianmario/escape_utils	Has similar methods to other repositories
https://github.com/atdt/escapes.js	Not a string sanitizer
https://github.com/mathiasbynens/jsesc	Unclear documentation
https://github.com/postrank-labs/postrank-uri	-
https://github.com/sindresorhus/escape-goat	-
https://github.com/sindresorhus/escape-string-regexp	-
https://github.com/brianmichel/ESCapey	Not a string sanitizer
https://github.com/susam/uncap	Not a string sanitizer
https://github.com/NZKoz/rails_xss	Not a string sanitizer
https://github.com/mathiasbynens/CSS.escape	Unclear documentation
https://github.com/justsml/escape-from-callback-mountain	Not a string sanitizer
https://github.com/sindresorhus/ansi-escapes	Not a string sanitizer
https://github.com/chalk/ansi-styles	Not a string sanitizer

[a] The sanitizer is too complex to be able to write a specification for it in the limited time frame that was available.

B.3 Sanitize

This table reflects the order of the results as it was on the 9th of July.

Repository link	Reason for rejection
https://github.com/csstools/sanitize.css	Not a string sanitizer
https://github.com/asaskevich/govalidator	-
https://github.com/google/sanitizers	Not a string sanitizer
https://github.com/cure53/DOMPurify	Unclear documentation
https://github.com/leizongmin/js-xss	Too complicated to write specification[a]
https://github.com/rgrove/sanitize	Too complicated to write specification[a]
https://github.com/mozilla/bleach	Too complicated to write specification[a]
https://github.com/cypriss/mutations	Not a string sanitizer
https://github.com/punkave/sanitize-html	Too complicated to write specification[a]
https://github.com/microcosm-cc/bluemonday	Too complicated to write specification[a]
https://github.com/flavorjones/loofah	Unclear documentation
https://github.com/mganss/HtmlSanitizer	Too complicated to write specification[a]
https://github.com/ruslo/polly	Not a string sanitizer
https://github.com/theSmaw/Caja-HTML-Sanitizer	Too complicated to write specification[a]
https://github.com/jaywcjlove/validator.js	Unable to read Chinese documentation
https://github.com/OWASP/java-html-sanitizer	Too complicated to write specification[a]
https://github.com/gbirke/Sanitize.js	Not a string sanitizer
https://github.com/hmans/slodown	Uses another repository (rgrove/sanitize) for sanitization
https://github.com/bevacqua/insane	Too complicated to write specification[a]
https://github.com/Microsoft/JSanity	Unclear documentation

Appendix C

Specified models for case study

This appendix includes all specifications that have been written for the case study on sanitizers. Each specification precisely describes how the sanitizer should behave.

C.1 Encode (he)

```
digraph spec{
 rankdir=LR;
0[label=0,peripheries=2]
XX0 [color=white, label=""]XX0 -> 0
XX0 -> 0
0 -> 0 [label="[\u0000-!#-%\(-;=?-_a-\uffff]/x+0"]
0 -> 0 [label="[&]/x+0 # x 2 6 ;"]
0 -> 0 [label="[<]/& # x 3 C ;"]
0 -> 0 [label="[>]/& # x 3 E ;"]
0 -> 0 [label="[\"]/& # x 2 2 ;"]
0 -> 0 [label="[\']/& # x 2 7 ;"]
0 -> 0 [label="[`]/& # x 6 0 ;"]
}
```

C.2 Escape (cgi)

```
digraph spec{
 rankdir=LR;
0[label=0,peripheries=2]
XX0 [color=white, label=""]XX0 -> 0
XX0 -> 0
0 -> 0 [label="[\u0000-%\'-;=?-\uffff]/x+0"]
0 -> 0 [label="[&]/& a m p ;"]
0 -> 0 [label="[<]/& l t ;"]
0 -> 0 [label="[>]/& g t ;"]
}
```

C.3 Escape (escape-string-regexp)

```
digraph spec{
 rankdir=LR;
0[label=0,peripheries=2]
XX0 [color=white, label=""]XX0 -> 0
XX0 -> 0
```

```
6  0 -> 0 [label="[$\(-+.?\[-^{-}]/\\ x+0"]
   0 -> 0 [label="[\u0000-#%-\',-\-/->@-Z_-z~-\uffff]/x+0"]
8  }
```

C.4 Escape (escape-goat)

```
   digraph spec{
2   rankdir=LR;
   0[label=0,peripheries=2]
4  XX0 [color=white, label=""]XX0 -> 0
   XX0 -> 0
6  0 -> 0 [label="[\u0000-!#-%\(-;=?-\uffff]/x+0"]
   0 -> 0 [label="[&]/& a m p ;"]
8  0 -> 0 [label="[<]/& l t ;"]
   0 -> 0 [label="[>]/& g t ;"]
10 0 -> 0 [label="[\"]/& q u o t ;"]
   0 -> 0 [label="[\']/& # 3 9 ;"]
12 }
```

C.5 toLowercase (CyberChef)

```
   digraph spec{
2   rankdir=LR;
   0[label=0,peripheries=2]
4  XX0 [color=white, label=""]XX0 -> 0
   XX0 -> 0
6  0 -> 0 [label="[\u0000-@\[-\uffff]/x+0"]
   0 -> 0 [label="[A]/a"]
8  0 -> 0 [label="[B]/b"]
   0 -> 0 [label="[C]/c"]
10 0 -> 0 [label="[D]/d"]
   0 -> 0 [label="[E]/e"]
12 0 -> 0 [label="[F]/f"]
   0 -> 0 [label="[G]/g"]
14 0 -> 0 [label="[H]/h"]
   0 -> 0 [label="[I]/i"]
16 0 -> 0 [label="[J]/j"]
   0 -> 0 [label="[K]/k"]
18 0 -> 0 [label="[L]/l"]
   0 -> 0 [label="[M]/m"]
20 0 -> 0 [label="[N]/n"]
   0 -> 0 [label="[O]/o"]
22 0 -> 0 [label="[P]/p"]
   0 -> 0 [label="[Q]/q"]
24 0 -> 0 [label="[R]/r"]
   0 -> 0 [label="[S]/s"]
26 0 -> 0 [label="[T]/t"]
   0 -> 0 [label="[U]/u"]
28 0 -> 0 [label="[V]/v"]
   0 -> 0 [label="[W]/w"]
30 0 -> 0 [label="[X]/x"]
```

```
    0 -> 0 [label="[Y]/y"]
32  0 -> 0 [label="[Z]/z"]
    }
```

C.6 htmlspecialchars (php)

```
 1  digraph spec{
     rankdir=LR;
 3   0[label=0,peripheries=2]
    XX0 [color=white, label=""]XX0 -> 0
 5  XX0 -> 0
    0 -> 0 [label="[\u0000-!#-%\'-;=?-\ufffd]/x+0"]
 7  0 -> 0 [label="[&]/x+0 a m p ;"]
    0 -> 0 [label="[<]/& l t ;"]
 9  0 -> 0 [label="[>]/& g t ;"]
    0 -> 0 [label="[\"]/& q u o t ;"]
11  }
```

C.7 filter_sanitize_email (php)

```
 1  digraph spec{
     rankdir=LR;
 3   0[label=0,peripheries=2]
    XX0 [color=white, label=""]XX0 -> 0
 5  XX0 -> 0
    0 -> 0 [label="[\u0000-\u001f!#-\'*-+\--.0-9=?-\[\]-\ufffd]/x
       +0"]
 7  0 -> 0 [label="[ ]/"]
    0 -> 0 [label="[\"]/"]
 9  0 -> 0 [label="[\(]/"]
    0 -> 0 [label="[\)]/"]
11  0 -> 0 [label="[,]/"]
    0 -> 0 [label="[/]/"]
13  0 -> 0 [label="[:]/"]
    0 -> 0 [label="[;]/"]
15  0 -> 0 [label="[<]/"]
    0 -> 0 [label="[>]/"]
17  0 -> 0 [label="[\\]/"]
    }
```

Bibliography

[1] OWASP Foundation, "Top 10-2017 application security risks." https://www.owasp.org/index.php/Top_10-2017_Application_Security_Risks, 2017. Accessed on 26-03-2018.

[2] OWASP Foundation, "Injection flaws." https://www.owasp.org/index.php/Injection_Flaws, 2015. Accessed on 06-04-2018.

[3] P. W. Resnick, "Internet message format," RFC 5322, RFC Editor, October 2008. http://www.rfc-editor.org/rfc/rfc5322.txt Accessed on 22-05-2018.

[4] G. Argyros, I. Stais, A. Kiayias, and A. D. Keromytis, "Back in black: towards formal, black box analysis of sanitizers and filters," in *2016 IEEE Symposium on Security and Privacy*, pp. 91–109, IEEE, 2016.

[5] OWASP Foundation, "Double encoding." https://www.owasp.org/index.php/Double_Encoding, 2014. Accessed on 06-06-2018.

[6] The PHP Group, "Sanitize filters." https://secure.php.net/manual/en/filter.filters.sanitize.php, n.d. Accessed on 28-03-2018.

[7] OWASP Foundation, "OWASP Java HTML Sanitizer." https://github.com/OWASP/java-html-sanitizer, n.d. Accessed on 28-03-2018.

[8] Python Software Foundation, "20.1. html — HyperText Markup Language support." https://docs.python.org/3/library/html.html#html.escape, n.d. Accessed on 28-03-2018.

[9] P. Saxena, D. Molnar, and B. Livshits, "SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 601–614, ACM, 2011.

[10] G. B. Welearegai and C. Hammer, "Idea: Optimized automatic sanitizer placement," in *International Symposium on Engineering Secure Software and Systems*, pp. 87–96, Springer, 2017.

[11] B. Livshits and S. Chong, "Towards fully automatic placement of security sanitizers and declassifiers," in *ACM Sigplan Notices*, vol. 48, pp. 385–398, ACM, 2013.

[12] F. Yu, M. Alkhalaf, and T. Bultan, "Patching vulnerabilities with sanitization synthesis," in *Proceedings of the 33rd International Conference on software engineering*, pp. 251–260, ACM, 2011.

[13] Microsoft, "Microsoft code analysis tool .NET (CAT.NET)." https://www.microsoft.com/en-us/download/details.aspx?id=19968, 2009. Accessed on 28-03-2018.

[14] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A systematic analysis of XSS sanitization in web application frameworks," in *European Symposium on Research in Computer Security*, pp. 150–171, Springer, 2011.

[15] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *2008 IEEE Symposium on Security and Privacy*, pp. 387–401, IEEE, 2008.

[16] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and precise sanitizer analysis with BEK," in *Proceedings of the 20th USENIX Security Symposium*, USENIX Association, 2011.

[17] M. Botinčan and D. Babić, "Sigma*: symbolic learning of input-output specifications," in *ACM SIGPLAN Notices*, vol. 48, pp. 443–456, ACM, 2013.

[18] M. A. Alkhalaf, *Automatic Detection and Repair of Input Validation and Sanitization Bugs*. University of California, Santa Barbara, 2014.

[19] F. Yu, *Automatic verification of string manipulating programs*. PhD thesis, University of California, Santa Barbara, 2010.

[20] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra, "Automata-based symbolic string analysis for vulnerability detection," *Formal Methods in System Design*, vol. 44, no. 1, pp. 44–70, 2014.

[21] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard, "Automatic input rectification," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 80–90, IEEE Press, 2012.

[22] M. Mohammadi, B. Chu, and H. Ritcher Lipford, "POSTER: Using unit testing to detect sanitization flaws," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 1659–1661, ACM, 2015.

[23] L. K. Shar and H. B. K. Tan, "Mining input sanitization patterns for predicting sql injection and cross site scripting vulnerabilities," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 1293–1296, IEEE Press, 2012.

[24] T. Scholte, W. Robertson, D. Balzarotti, and E. Kirda, "Preventing input validation vulnerabilities in web applications through automated type analysis," in *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pp. 233–243, IEEE, 2012.

[25] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, "Bouncer: Securing software by blocking bad input," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 117–130, ACM, 2007.

[26] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.

[27] M. Shahbaz and R. Groz, "Inferring mealy machines," in *International Symposium on Formal Methods*, pp. 207–222, Springer, 2009.

[28] M. Spichakova, "An approach to the inference of finite state machines based on a gravitationally-inspired search algorithm.," *Proceedings of the Estonian Academy of Sciences*, vol. 62, no. 1, 2013.

[29] B. Jonsson, "Learning of automata models extended with data," in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pp. 327–349, Springer, 2011.

[30] S. Cassel, F. Howar, B. Jonsson, and B. Steffen, "Learning extended finite state machines," in *International Conference on Software Engineering and Formal Methods*, pp. 250–264, Springer, 2014.

[31] F. Howar, B. Steffen, B. Jonsson, and S. Cassel, "Inferring canonical register automata," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pp. 251–266, Springer, 2012.

[32] R. Medhat, S. Ramesh, B. Bonakdarpour, and S. Fischmeister, "A framework for mining hybrid automata from input/output traces," in *Proceedings of the 12th International Conference on Embedded Software*, pp. 177–186, IEEE Press, 2015.

[33] S. Drews and L. D'Antoni, "Learning symbolic automata," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 173–189, Springer, 2017.

[34] M. Henrix, "Performance improvement in automata learning," *Master's thesis, Radboud University Nijmegen*, 2015.

[35] W. Smeenk, J. Moerman, F. Vaandrager, and D. N. Jansen, "Applying automata learning to embedded control software," in *International Conference on Formal Engineering Methods*, pp. 67–83, Springer, 2015.

[36] T. Bohlin and B. Jonsson, "Regular inference for communication protocol entities," tech. rep., Technical Report 2008-024, Uppsala University, Computer Systems, 2008.

[37] H. Hungar, T. Margaria, and B. Steffen, "Test-based model generation for legacy systems," in *Test Conference, 2003. Proceedings. ITC 2003. International*, vol. 2, pp. 150–159, IEEE, 2003.

[38] T. Bultan, F. Yu, M. Alkhalaf, and A. Aydin, *String Analysis for Software Verification and Security*. Springer, 2018.

[39] T. A. Sudkamp, *Languages and Machines : An Introduction to the Theory of Computer Science*. Addison-Wesley Longman Publishing Co., Inc., 1988.

[40] E. Roche and Y. Schabes, *Finite-state language processing*. MIT press, 1997.

[41] L. D'Antoni, "Symbolic automata." http://pages.cs.wisc.edu/~loris/symbolicautomata.html, n.d. Accessed on 09-04-2018.

[42] L. D'Antoni and M. Veanes, "The power of symbolic automata and transducers," in *Proceedings Part I of 29th International Conference on Computer Aided Verification, CAV 2017*, pp. 47–67, Springer, 2017.

[43] M. Veanes, P. De Halleux, and N. Tillmann, "Rex: Symbolic regular expression explorer," in *Third International Conference on Software Testing, Verification and Validation (ICST)*, pp. 498–507, IEEE, 2010.

[44] L. D'Antoni and M. Veanes, "Forward bisimulations for nondeterministic symbolic finite automata," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 518–534, Springer, 2017.

[45] N. Bjørner and M. Veanes, "Symbolic transducers," Tech. Rep. MSR-TR-2011-3, Microsoft Research, 2011.

[46] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner, "Symbolic finite state transducers: Algorithms and applications," in *ACM SIGPLAN Notices*, vol. 47, pp. 137–150, ACM, 2012.

[47] Northwave. https://www.northwave.nl/. Accessed on 31-07-2018.

[48] M. Veanes and N. Bjørner, "Symbolic tree automata," *Information Processing Letters*, vol. 115, no. 3, pp. 418–424, 2015.

[49] M. Veanes and N. Bjørner, "Symbolic tree transducers," in *Proceedings of the 8th International Conference on Perspectives of System Informatics*, PSI'11, (Berlin, Heidelberg), pp. 377–393, Springer-Verlag, 2012.

[50] J. M. Vilar, "Query learning of subsequential transducers," in *International Colloquium on Grammatical Inference*, pp. 72–83, Springer, 1996.

[51] W. Smeenk, "Applying automata learning to complex industrial software," *Master's thesis, Radboud University Nijmegen*, 2012.

[52] R. Hamlet, "Random testing," *Encyclopedia of software Engineering*, 2002.

[53] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[54] L. D'Antoni, "symbolicautomata." https://github.com/lorisdanto/symbolicautomata. Accessed on: 01-07-2018.

[55] Unicode Inc., "Unicode 11.0 character code charts." https://www.unicode.org/charts/, 2018. Accessed on: 01-08-2018.

[56] M. Bynens, "he." https://github.com/mathiasbynens/he, 2018. Accessed on: 15-08-2018.

[57] Python Software Foundation, "20.2. cgi — common gateway interface support." https://docs.python.org/2/library/cgi.html, 2018. Accessed on: 15-08-2018.

[58] S. Sorhus, "escape-string-regexp." https://github.com/sindresorhus/escape-string-regexp, 2016. Accessed on: 15-08-2018.

[59] S. Sorhus, "escape-goat." https://github.com/sindresorhus/escape-goat, 2017. Accessed on: 15-08-2018.

[60] PostRank Labs, "PostRank URI." https://github.com/postrank-labs/postrank-uri, 2017. Accessed on: 15-08-2018.

[61] GCHQ (Government Communications Headquarters), "Cyberchef." https://github.com/gchq/CyberChef. Accessed on: 23-07-2018.

[62] The PHP Group, "htmlspecialchars." http://php.net/manual/en/function.htmlspecialchars.php, 2018. Accessed on: 15-08-2018.

[63] The PHP Group, "Sanitize filters." http://php.net/manual/en/filter.filters.sanitize.php, 2018. Accessed on: 15-08-2018.

[64] A. Saskevich, "govalidator." https://github.com/asaskevich/govalidator/, 2018. Accessed on: 15-08-2018.

[65] B. K. Aichernig and M. Tappler, "Learning from faults: Mutation testing in active automata learning," in *NASA Formal Methods Symposium*, pp. 19–34, Springer, 2017.

[66] O. Saarikivi and M. Veanes, "Minimization of symbolic transducers," in *International Conference on Computer Aided Verification*, pp. 176–196, Springer, 2017.