Browser Fingerprinting using Combinatorial Sequence Testing

Bernhard Garn
SBA Research
Vienna, Austria
bgarn@sba-research.org

Dimitris E. Simos
SBA Research
Vienna, Austria
dsimos@sba-research.org

Stefan Zauner
FH Campus Wien
Vienna, Austria
stefan.zauner@stud.fh-campuswien.ac.at

Rick Kuhn
NIST
Gaithersburg, MD, USA
d.kuhn@nist.gov

Raghu Kacker
NIST
Gaithersburg, MD, USA
raghu.kacker@nist.gov

ABSTRACT

In this paper, we report on the applicability of combinatorial sequence testing methods to the problem of fingerprinting browsers based on their behavior during a TLS handshake. We created an appropriate abstract model of the TLS handshake protocol and used it to map browser behavior to a feature vector and use them to derive a distinguisher. Using combinatorial methods, we created test sets consisting of TLS server-side messages as sequences that are sent to the client as server responses during the TLS handshake. Further, we evaluate our approach with a case study showing that combinatorial properties have an impact on browsers' behavior.

KEYWORDS

combinatorial testing, security testing, browser fingerprinting

ACM Reference Format:
Bernhard Garn, Dimitris E. Simos, Stefan Zauner, Rick Kuhn, and Raghu Kacker. 2019. Browser Fingerprinting using Combinatorial Sequence Testing. In *Hot Topics in the Science of Security Symposium (HotSoS), April 1–3, 2019, Nashville, TN, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3314058.3314062

1 INTRODUCTION

The security and threat landscape of the computer systems of today can be viewed from a macroscopic and a microscopic point of view. Both views are essential, independent and influence each other. In this work, we focus on fingerprinting Internet browsers by analyzing their behavior when they are processing non-standard response messages from a server during a TLS handshake. These non-standard messages are derived using combinatorial methods and no other means than the resulting behavior are used in the fingerprinting approach. Browsers, a type of end-user software which is paramount, constitute the central piece of software by which computing power and the Internet are consumed on a variety of mobile and classic devices. A lot of effort is spent to annotate vulnerabilities found not only with the specific software where the vulnerability was discovered, but to also determine all other software that is also affected. In a *defensive* position, these efforts usually lead to security advisories where users are then asked to update the affected software to a version for which this vulnerability has been resolved. From an *offensive* or penetration testing position, a list of pairs consisting of a software-vulnerability combination provides the means for planning concrete penetration tests on systems. A prerequisite to this step is, however, that sufficient knowledge is available about the target system so that a detailed software version-vulnerabilities list can be used effectively. In such a hypothetical scenario, the process likely begins with a reconnaissance phase to determine the exact versions of the software that is running on the target system. To this end, in this paper, we employ a combinatorial sequence testing technique in a black-box testing approach tailored to fingerprint software in use. Specifically, we assume the role of a classical web server where users of that service want to connect with a secure connection (HTTP over TLS 1.2 [4]). Our idea is that by responding with certain TLS handshake messages (on some connection attempts), the resulting error messages (if any are sent) can be used to uniquely identify the used browser. After a successful fingerprinting of the browser, the next attack steps can be planned with the precise knowledge of the browser version in use on the target (i.e., exploits specifically developed for this target). We would like to note that it is possible to create a database of browser behavior depending on TLS sequences independent from any specific target and most importantly, in advance, and that it can be continuously updated to have it available for later usage.

Over the last couple of years, there has been a trend to offer more and more content over HTTPS, for a variety of reasons. Apart from the previously described use case, browser fingerprinting might be used in a variety of scenarios, like identifying users or obtaining information about the type and version of a browser in order to deliver suitable malware.

Our approach maps the behavior of browsers to *feature vectors*, upon which we base our classification. The resulting analysis on the obtained partitions of all considered browsers can be regarded as a way to *quantify* and *reason* about the *similarities* in observed browser behavior.

The goal of this work is *to investigate the applicability of combinatorial methods to generate artifacts upon which the behavior of browsers will be evaluated and recorded to be used as underlying means to build a fingerprinting approach based on it.* Specifically, the *research question* for this work reads as follows:

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only.
HotSoS, April 1–3, 2019, Nashville, TN, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7147-6/19/04.
https://doi.org/10.1145/3314058.3314062

RQ: *Can the behavior observed from test sets created by sequence covering arrays containing TLS server-side messages be used to fingerprint browsers?*

The methods proposed in this paper can be used independently or in conjunction with existing methods and techniques for browser fingerprinting. Moreover, the research objective of this paper is not to increase the efficiency or to address specific limitations of existing approaches, but to evaluate the applicability of a new approach based on combinatorial methods to the problem of browser fingerprinting and as a result to extend the tools and techniques available in general in the field of fingerprinting.

We would like to note that some properties reported by a browser to a server (for example, its user agent string) are inherently not trustworthy (since they can be set easily arbitrarily), whereas the methods proposed in this paper is solely based on the observed behavior of browsers (i.e., the message-exchange occurring during a TLS handshake attempt). We expect it to be more difficult to defend against this type of fingerprinting or to fake the behavior on errors to imitate the error-behavior of another browser, as mitigation strategies would have to operate on the level of the underlying TLS implementation.

Contribution. In particular, this paper makes the following contributions:

- Proposes certain combinatorial sequences as test cases for fingerprinting browser behavior;
- Presents experimental results of a case study demonstrating our approach;

This paper is structured as follows. In Section 2 we discuss related work. We present our approach to fingerprinting of browsers using combinatorial methods in Section 3 and describe our developed testing framework in Section 4. In Section 5 we describe the setup of our case study and analyze the obtained results in Section 6. Last, Section 8 concludes the paper.

2 RELATED WORK

Approaches for browser fingerprinting often rely on properties exposed by the browser about itself and the underlying operating system. The authors of [5] followed this approach, and in [16] this methodology was strengthened by additionally considering browser plug-ins. In [1], the authors focused on fingerprinting scripts.

Another line of research uses not only properties, but also capabilities of browsers for developing fingerprinting approaches. A fingerprinting technique based on the onscreen dimension of font glyphs was presented in [7]. The rendering of 3D scenes together with text rendering in a web page on an HTML5 <canvas> element was used in [12] to base a fingerprinting method upon it. Other browser capabilities have also been used to create fingerprinting approaches [22].

In [13], the underlying JavaScript engine was used to devise a fingerprinting approach. The correlation between feature combinations and identification accuracy has been considered in [20].

In [17], the authors used planning with combinatorial methods for providing test cases for testing different TLS implementations.

3 COMBINATORIAL METHODS FOR FINGERPRINTING

In this section, we detail how combinatorial methods arising in the field of discrete mathematics in conjunction with an abstract modelling methodology can be used to create test sequences that enable fingerprinting of browsers. First, we describe the combinatorial structures employed in this paper, and then we present our modelling methodology and how the constructed sequences can be used for testing. While combinatorial methods have been used in the past in the context of combinatorial security testing [18], in this paper combinatorial methods are used for the first time as the underlying means to create a fingerprinting approach.

3.1 Sequence Covering Arrays

Sequence covering arrays (SCAs) [9],[2] are matrices designed to test software behavior that depends on the order of events, by ensuring that any t events will occur in every possible t-way order (allowing interleaving events among each subset of t events). A sequence covering array, SCA (N, S, t), is defined as an $N \times S$ matrix where entries are from a finite set S of s symbols, such that every t-way permutation of symbols from S occurs in at least one row and each row is a permutation of the s symbols [9]. The t symbols in the permutation are not required to be adjacent. That is, for every t-way arrangement of symbols $x_1, x_2, \ldots, x_t$, the regular expression $.*x_1.*x_2 \cdots .*x_t.*$ matches at least one row in the array.

For example, with six events, a, b, c, d, e, f, one subset of three events is $\{a, c, e\}$, which can be arranged in six possible permutations. There are[1] $\binom{6}{3} = 20$ sets of three events, and each can be arranged in $3! = 6$ orders. Using only 10 tests, it is possible to include all 3-way orders of these six events, as shown in Table 1. It can be shown that, for a given value of t, the number of tests required to cover all t-way orders grows with $\log S$. For the case where $t = 2, N = 2$ for all values of S, i.e., testing 2-way sequences never requires more than two tests, regardless of the number of events.

Sequence covering arrays have been used in a variety of testing applications, including industrial control systems [3], web applications [10], cryptographic hash functions [11], and laptop utilities [9]. They can be constructed using a simple greedy algorithm approach [9], although search and logic based strategies have been used as well [6],[8]. Greedy algorithms are generally faster, while other approaches may produce slightly smaller test arrays. To our knowledge, they have not been used for device fingerprinting or other applications outside of conventional software testing.

SCA generation. We implemented the algorithm given in [9] in the Python 2 programming language [15]. It takes as input the number of events n and the desired interaction strength t and returns a SCA for the specified configuration over the alphabet $\{0, \ldots, n-1\}$. The algorithm follows a greedy strategy, where after empty initialization of the test sequence set, in each iteration, a number of test sequences are created randomly, individually scored by the number of previously uncovered t-way sequences they cover, and then the highest scoring test sequence is added to the test set

[1] For two nonnegative integers n and k with $k \leq n$ we denote with $\binom{n}{k}$ the binomial coefficient. For a positive integer i we denote with $i!$ the factorial of i.

Table 1: All 3-event sequences of 6 events.

Test	Sequence
1	a b c d e f
2	f e d c b a
3	d e f a b c
4	c b a f e d
5	b f a d c e
6	e c d a f b
7	a e f c b d
8	d b c f e a
9	c e a d b f
10	f b d a e c

Table 2: Number of tests for generated SCAs.

n	t	#tests
2	2	2
3	2	2
3	3	6
4	2	2
4	3	8
4	4	24
5	2	2
5	3	10
5	4	28
5	5	120
6	2	2
6	3	10
6	4	36
6	5	156
6	6	720

until all required t-way sequences are covered. The sizes of the SCAs that this program generated are given in Table 2.

SCA verification. Test sequences used in this work were verified for coverage using the Combinatorial Sequence Coverage Measurement (CSCM) tool [23],[24]. CSCM was designed to allow testers to evaluate the sequence coverage of test sets that have not necessarily been generated to cover sequences. Event sequence testing has long been known to be important in fields such as communication protocols, and many methods exist to generate sequences that cover the state transition graphs for protocol testing. Many consumer-level applications, particularly for web sites and smartphone apps, also have the potential for complex interactions that are difficult to test fully. CSCM allows testers to determine the extent to which t-way sequences are covered in a test set, and to supplement existing tests as needed to enhance test thoroughness.

3.2 Application to fingerprinting

Now, we explain how to use SCAs in the context of browser fingerprinting. First, we briefly summarize some facts about the TLS protocol, which is used to establish a secure connection between two parties.

Transport Layer Security. The *transport layer security protocol* (TLS) can be used by browsers to establish a secure connection with a web server. The setup of this secure channel is achieved via the exchange of a sequence of messages, where the two parties negotiate some parameter values for later use. This procedure is encoded in the handshake protocol within the TLS specification, and both client and server should follow it. We regard all TLS messages that are specified for the server as a set of six abstract events $\mathcal{E}$, consisting of:

$$\mathcal{E} = \{\text{ServerHello}, \text{Certificate}, \text{ECDHEServerKeyExchange}, \quad (1a)$$
$$\text{ServerHelloDone}, \text{ChangeCipherSpec}, \text{Finished}\}. \quad (1b)$$

If we use the ordering derived from the message exchange in the TLS specification, then the following mapping of TLS events to numbers is canonical:

(0) ServerHello
(1) Certificate
(2) ECDHEServerKeyExchange
(3) ServerHelloDone
(4) ChangeCipherSpec
(5) Finished

We denote finite, nonempty sequence over the alphabet $\mathcal{E}$ is ascending order between angle brackets, for example $\langle 0, 3, 5\rangle$.

Combinatorial Sequence Model and derived test cases. Given a nonempty subset of cardinality κ of abstract TLS events[2] of the set $\mathcal{E}$, it is possible to construct a SCA for any strength $t \in \{1, \ldots, \kappa\}$. For later evaluation purposes, for any $\emptyset \neq E \subseteq \mathcal{E}$ of cardinality[3] κ we created and stored the image of E under the symmetric group of κ elements in the database. In other words, for any nonempty subset of sequences, we created all of its permutations and stored them in the database in the `Sequence` table and we denote the corresponding set of all test sequences with $\mathcal{S}$. It follows that we have in total

$$\sum_{i=1}^{6} \binom{6}{i} \cdot i! = 1956 \quad (2)$$

test sequences in the database.

Since we store, for any nonempty subset E of $\mathcal{E}$, all of its permutations in the database, it follows that we can find any SCA defined over the elements of E in the database (i.e., fetching exactly those rows from the database which correspond to the rows of the SCA).

For later use, we also fix an enumeration of the set $\mathcal{S}$, that is we fix a bijection[4] from the set $\{1, \ldots, |\mathcal{S}|\} \longrightarrow \mathcal{S}$.

Given a SCA, we refer to a sequence as a test sequence (i.e., row in the array) and to the array in its entirety as a test set. Such an abstract test sequence can now be translated into a concrete TLS message that will be sent to the client *over the network* by instantiating its values with default values taken from TLS attacker. We

[2]It would also be possible to consider not only subsets, but also sub-multisets, i.e., by allowing at least one event to appear strictly more than once in the considered selection. We leave this as future work.
[3]For a set S, we denote its cardinality with $|S|$.
[4]We obtained such an enumeration from the sequence identifiers in the database.

would like to emphasize that while we are considering permutations of subsets of the set $\mathcal{E}$, the concrete message values are taken from TLS attacker[5].

4 TESTING FRAMEWORK

In this section, we describe our testing setup of our automated framework. It is composed of several components and their general interactions are depicted in Figure 1. The frameworks' test execution component uses TLS attacker [19] for sending and receiving of TLS messages of clients (i.e., browsers). TLS test sequences are taken from a specified list, created using combinatorial methods. Subsequently, the logging component of the framework stores annotated results for each browser in a dedicated relational database. In other words, for any given browser, the framework executes the following steps:

(1) TLS attacker is started as a server and given as input the XML translation of a test sequence from a set test.
(2) The browser is instrumented to connect via HTTPS to a specific local url of TLS attacker.
(3) The framework (i.e., the server side in this connection attempt) will respond according to the encoded messages of the test sequence given in XML format.
(4) The exchange of TLS messages between the client (browser) and server will continue as long as possible until all messages from the test case have been sent from the server. The complete message exchange is recorded.
(5) The recorded message exchange is annotated and stored in a relational database.

Next, we describe each component in detail.

Test cases. In our experiments we considered all possible injective ordered sequences $\mathcal{S}$ over the alphabet $\mathcal{E}$ of length one to six. This means that, in particular, the set $\mathcal{S}$ has as a subset all t-SCAs for all $t \in \{1, 2, 3, 4, 5, 6\}$. In Section 6, during the evaluation, we will make use of this fact and compare the *distinguishing* capabilities of various subsets of $\mathcal{S}$, in particular those of SCAs for different event selections and different strengths. Due to the size advantage of SCAs compared to the set of all permutations of some fixed set of elements, it is clear that while for our initial experiments we were able to work the complete set $\mathcal{S}$, in future extension steps of an existing analysis database it would be more desirable to follow a test set augmentation strategy based on higher strength of SCAs instead of adding all permutations.

Test execution. The test execution component uses TLS attacker[6] [19], which is an open source framework that allows the creation of custom TLS message flows, both from a client and server side. Since we want to fingerprint clients (i.e., browsers), we use TLS attacker in the server role. For each abstract test sequence, an XML encoded TLS handshake sequence is created, which is one of the input formats of TLS attacker. If available, we start the browsers in headless mode. The browsers connect to a locally running TLS attacker. A self-signed root certificate was created and added to the list of trusted certificates for each browser and the certificate that is sent by TLS attacker was signed with the private key of the root certificate (no intermediate certificate). Also, the certificate sent by our server has a V3-Extension called `subjectaltname`. This was necessary since Google Chrome uses this extension to validate the certificate and otherwise it would not accept it.

During the handshake, the framework will reply with exactly the messages in the order specified in the current test sequence with the concrete message values, except for the certificate, instantiated to default values provided by TLS attacker[7]. The ciphersuite was fixed to `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA`, as all tested browsers considered it secure.

The browser process is terminated as soon as the stream (from TLS attacker) is closed and all the results are stored. This is the same process for both headless and not headless.

Logging and database. While instrumenting TLS attacker, we parse its comprehensive output. This output includes detailed information about sent and received TLS messages. It also outputs TLS `alert messages` and prints out exceptions that happened during execution (stored in the field `ExceptionHandshake`), for example if the browser closed the socket or if TLS attacker failed to parse a received message. The tests also send an HTTPS-response to the browser, so it is possible to record exceptions that happened after the handshake (stored in the field `ExceptionPostHandshake`), when data should be sent over the now secure channel. We added this information to make it possible to have more data in the feature vectors used as distinguishers (e.g., no exception is thrown during the handshake, but the socket is closed as soon as TLS attacker tries to send data).

This logging information, together with the tested client (i.e., browser) is then stored in a SQLite database [14]. Moreover, the database also contains tables with

- the list of browsers, their paths and command line options for how to start them, respectively;
- the set of TLS events $\mathcal{E}$;
- a workflow skeleton that is populated with the saved TLS messages;
- the list of all considered test sequences $\mathcal{S}$.

We give some examples for results obtained in Table 3, as they are stored in the database.

Walk through example. We illustrate our approach and the test case execution with an example. Consider a five event selection out of the set $\mathcal{E}$ where the event Certificate is omitted, i.e. we choose the following sequence

$$\langle \text{ServerHello}, \text{Finished}, \text{ServerHelloDone}, \tag{3a}$$
$$\text{ChangeCipherSpec}, \text{ECDHEServerKeyExchange}\rangle \tag{3b}$$
$$= \langle 0, 5, 3, 4, 2\rangle \tag{3c}$$

built from these events. The corresponding XML encoded sequence is depicted in Listing 1.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<workflowTrace>
    <Receive>
        <expectedMessages>
```

[5]We leave their additional manipulation as future work.
[6]Release v2.6.
[7]https://github.com/RUB-NDS/TLS-Attacker/blob/master/TLS-Core/src/main/resources/default_config.xml

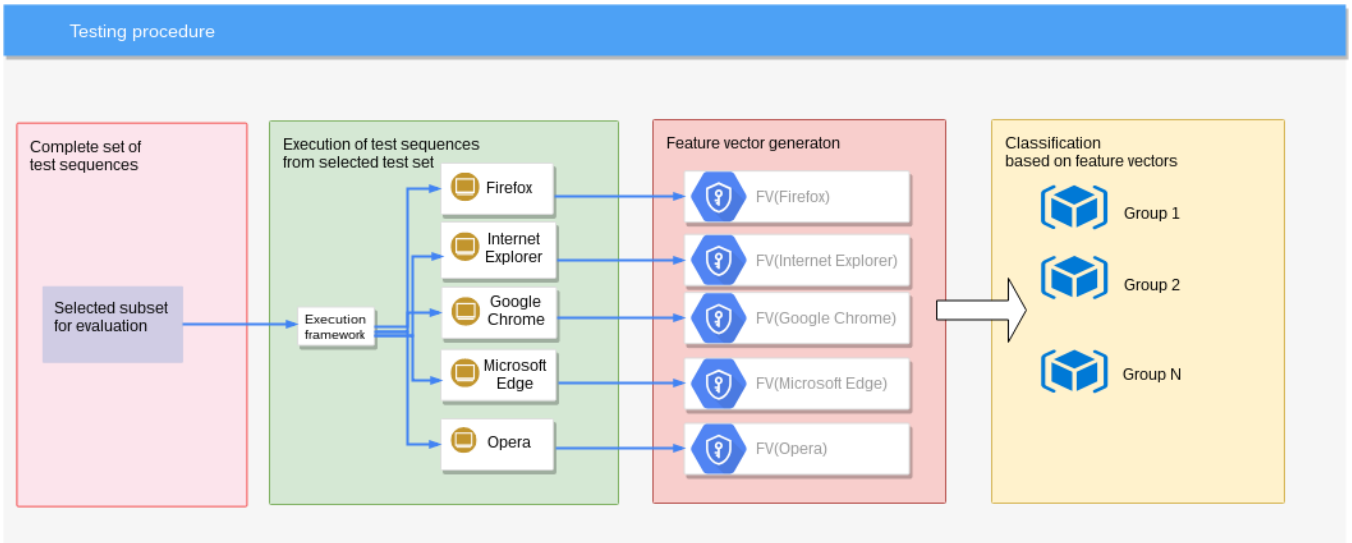

Figure 1: Overview of the fingerprinting process.

Table 3: Excerpt from the `Results` table.

ID	ReceivedMsgs1	ReceivedMsgs2	AlertMessage	ExceptionHandshake	ExceptionPostHandshake	br_id	Seq_id
6	CLIENT_HELLO	AlertMessage	UNEXPECTED_MESSAGE			1	2
761	CLIENT_HELLO	AlertMessage	UNEXPECTED_MESSAGE		java.net.SocketException: Connection reset by peer: socket write error	1	153
3881	CLIENT_HELLO			java.net.SocketException: Connection reset by peer: socket write error	java.net.SocketException: Connection reset by peer: socket write error	1	777
3882	CLIENT_HELLO				java.net.SocketException: Software caused connection abort: socket write error	2	777
3883	CLIENT_HELLO	AlertMessage	UNEXPECTED_MESSAGE		java.net.SocketException: Software caused connection abort: socket write error	3	777
3884	CLIENT_HELLO				java.net.SocketException: Software caused connection abort: socket write error	4	777
3885	CLIENT_HELLO	AlertMessage	UNEXPECTED_MESSAGE		java.net.SocketException: Software caused connection abort: socket write error	5	777

```
            <ClientHello/>
        </expectedMessages>
    </Receive>
    <Send>
        <messages>
            <ServerHello/>
            <Finished/>
            <ServerHelloDone/>
            <ChangeCipherSpec/>
        </messages>
    </Send>
    <Receive>
        <expectedMessages>
            <ECDHClientKeyExchange/>
            <ChangeCipherSpec/>
        <Finished/>
    </expectedMessages>
    </Receive>
    <Send>
        <messages>
            <ECDHEServerKeyExchange/>
        </messages>
    </Send>
    <Send>
    <messages>
        <HttpsResponse>
        </HttpsResponse>
    </messages>
    </Send>
</workflowTrace>
```

Listing 1: TLS 1.2 altered handshake workflow.

For this example, we consider the case of testing the behavior of the browser Firefox. It is started in `headless` mode pointing to a local resource from the execution framework using the following command:

```
C:\Program Files\Mozilla Firefox\firefox.exe
        -headless -url https://localhost:5555
```

Upon execution, this test sequence led to the following exchange of TLS messages:

(1) `ClientHello` sent from Firefox.
(2) The framework sends the following messages:

$$\langle \text{ServerHello, Finished, ServerHelloDone,} \tag{4a}$$
$$\text{ChangeCipherSpec} \rangle \tag{4b}$$

(3) The client (Firefox in this case) responds and the response is identified by the execution framework as a TLS alert message of type `UNEXPECTED_MESSAGE`. Furthermore, we obtain

from TLS attacker an exception of type `java.net.SocketException: Connection reset by peer: socket write error`.

(4) Finally, an annotated version of the above test execution result is stored in the database.

5 CASE STUDY

In this section, we describe in detail the sequences used as test sets and the tested browsers and their version.

The objective of this case study is to provide initial experimental results about the capabilities of the executed methods for fingerprinting browsers. To this end, we ran the tests on a prevalent operating system for major browser vendors.

Test sets. As already mentioned in Section 3.2, our automated approach made it possible to work with the complete set $\mathcal{S}$ containing all permutations for all nonempty subset selections of TLS messages.

Independently, we used the generated SCAs (see Section 3.1) to obtain for every compatible selection of events a list of test sequence IDs. These IDs were used to obtain a subset of rows from the `Sequence` table in the database corresponding to the instantiated abstract SCA with the TLS messages selection.

Since we can find these SCAs as subsets of $\mathcal{S}$, by executing all test sequences of $\mathcal{S}$ against all browsers, we have also tested all SCAs of interest.

Browsers. In total, we tested five browsers for a case study, consisting of the following:

(1) Mozilla Firefox, version 64.0.0.6914;
(2) Opera, version 57.0.3098.106;
(3) Google Chrome, version 71.0.3578.98;
(4) Microsoft Internet Explorer, version 11.0.17134.1;
(5) Microsoft Edge, version 11.00.17134.471.

Framework setup. All experiments were performed on Windows 10 Pro, 64-bit Build 17134.472 Version 1803 running inside a virtual machine created with VMware Workstation 12 Pro 12.5.9 build-7535481. During the testing, all browsers connected to a locally running instance of TLS attacker.

6 EVALUATION

In this section, we present our results from running the experiments described in Section 5 and the theoretical criteria upon which we base our analysis. We explain how we instantiate feature vectors for abstract analysis in Section 6.1 and subsequently remark on how we compare them in Section 6.2. Afterwards, we elaborate on the results obtained in Section 6.3. The evaluation was performed with a program written in Perl v5.24.1 [21], which queried the results database and carried out the necessary steps for the analysis of our results.

Results show that the methods presented in this paper are effective for distinguishing between browser classes. That is, a very large number of the SCA tests were able to determine the browser type as belonging to one of the three categories: {Firefox}, {Google Chrome, Opera}, {Microsoft Internet Explorer, Microsoft Edge}.

6.1 Feature vector definition

For given browser and nonempty set of test sequences $S \subseteq \mathcal{S}$, we define a *feature vector* as follows:

- For each $s \in S$, let r_s be the result of executing test sequence s against the given browser (queried from the results database), stored as an array of strings. Note that the length of this array is uniform for all browsers and all test sequences in the set $\mathcal{S}$.
- Let fv denote an array of length $|S|$, where each entry is a reference to the array r_s, in ascending order according to the chosen enumeration of the set $\mathcal{S}$. The result can be interpreted as a two-dimensional array where the the first position of a two-dimensional index pair corresponds to the enumeration identifier of a test sequence and the second position to a column in the `Result` table schema.
- We define the array fv as feature vector for the given browser and set S of sequences.

We exemplify our definition of feature vectors with an example. Consider the browser Mozilla Firefox together with the following set of $S \subseteq \mathcal{S}$ of test sequences:

$$S = \{\langle \text{Certificate} \rangle, \langle \text{Finished}, \text{ChangeCipherSpec}, \text{ServerHello} \rangle\} \tag{5}$$

The result of these two test sequences executed against Firefox (which has `browser_id` equal to one) are depicted in Table 3, where

- the sequence ⟨Certificate⟩ has `Sequence_id` equal to two and the corresponding result is stored in the `Result` table with `ID` equal to six.
- the sequence ⟨Finished, ChangeCipherSpec, ServerHello⟩ has `Sequence_id` equal to 153 and the corresponding result is stored in the `Result` table with `ID` equal to 761.

The resulting feature vector fv has length two. The first entry points to an array of strings with the respective values shown in Table 3 for `ID` equal to six[8]:

```
("CLIENT_HELLO"|"AlertMessage"|"UNEXPECTED_MESSAGE"|""|"")
```
(6)

The second entry points to an array of strings with the respective values shown in Table 3 for `ID` equal to 761[9]:

```
("CLIENT_HELLO"|"AlertMessage"|"UNEXPECTED_MESSAGE"|""|   (7a)
                                "java.net.SocketException:   (7b)
          Connection reset by peer: socket write error")   (7c)
```

6.2 Classification of feature vectors

Since we are interested in finding nonempty subsets of the set $\mathcal{S}$ where we can observe *different behavior* of the tested browsers, we now give a precise definition of how the term *different behavior* is to be understood in this paper. Suppose we are given a nonempty subset $S \subseteq \mathcal{S}$ and two different browsers, then we say that they

[8] The characters (and) denote the start and end of the array, respectively; the double high quotes delimit strings; and the character | is used as array element separator.
[9] See Footnote 8.

exhibit different behavior with respect to S, if and only if, their respective feature vectors differ[10].

For five browsers, there are ten possible pairwise comparisons between their behaviors (i.e., pairwise comparisons between their respective feature vectors). We use these pairwise comparisons to define an equivalence relation on the set of all browsers. For fixed nonempty set $S \subseteq \mathcal{S}$, two browsers are equivalent, if and only if, they exhibit the same behavior. In other words, two browsers are members of the same equivalence class, if and only if, they exhibit the same behavior for the set S. Due to our interest in fingerprinting browsers according to their behavior using test cases created with combinatorial methods, we are especially interested in analyzing the resulting partitions for different nonempty sets of test sequences.

6.3 Analysis of results

Firstly, we make some general observations on our results in Section 6.3.1. Then, we proceed with our analysis for groups of sequences of the same length for lengths from 1 up to 6 in Section 6.3.2 until Section 6.3.7. For $i \in \{1, 2, 3, 4, 5, 6\}$ we denote with S_i the subset of $\mathcal{S}$ consisting of exactly those sequences of length i. Note that in the case of $n = t$, the notions of the image of a set of cardinality n under the full permutation group and a *SCA* of strength n result in the same set of sequences.

6.3.1 General observations. For each individual test sequence in the set $\mathcal{S}$, we have seen at least the following pairwise equalities between behavior of two specific selections of two browsers:

- The browsers Microsoft Internet Explorer and Microsoft Edge exhibit the same behavior.
- The browsers Google Chrome and Opera exhibit the same behavior.

It follows that:

- Both of these selections of two browsers will have the same behavior for any nonempty subset of the set $\mathcal{S}$.
- For both of these selections of two browsers, for any nonempty set of test sequences, the resulting partition will have at most three classes.
- The approach for fingerprinting presented in this paper is currently not able to distinguish browsers within these two browser pairs.
- The result that those two browser-pair selections always exhibit the same behavior is not surprising, since they internally use closely related libraries for handling TLS handshakes.

It is possible to make the above statements more precise, which we state in the form of an explicit description of the appearing partitions. For each test sequence, the number of equivalence classes in the corresponding partition is an element of the set

$$C = \{1, 2, 3\}, \tag{8}$$

and for each number in the set C there is at least one test sequence where the partition corresponding to this sequence (i.e., singleton of test sequence selection) has exactly this number of equivalence classes.

For each number in the set C, we give now a more detailed description for the occurring partitions.

- Number of equivalence classes equal to one: All browsers have the same behavior and the corresponding partitions are equal. This partition occurs seven times.
- Number of equivalence classes equal to two: There are two different partitions:
  - One partition consisting of the two classes:
    (1) {Firefox, Google Chrome, Opera},
    (2) {Microsoft Internet Explorer, Microsoft Edge}
    occurring 22 times.
  - The other partition consisting of the two classes:
    (1) {Firefox},
    (2) {Microsoft Internet Explorer, Microsoft Edge, Google Chrome, Opera},
    occurring only once.
- Number of equivalence classes equal to three: We denote this unique class as $\mathcal{P}_3$ and the classes for the respective partitions are equal to:
  (1) {Firefox},
  (2) {Google Chrome, Opera},
  (3) {Microsoft Internet Explorer, Microsoft Edge}.
  This case occurs 1926 times.

After this analysis of all possible singletons of test sequence selections, next we analyze test sets for the same length and of cardinality at least two, in particular SCAs for different strengths.

6.3.2 Sequences of length 1. There are six selections of one event.

⟨*ServerHello*⟩*:* In this case, the resulting partition contains only one class with five elements, i.e., all browsers behave the same way.

⟨*Certificate*⟩, ⟨*ECDHEServerKeyExchange*⟩, ⟨*ServerHelloDone*⟩, ⟨*ChangeCipherSpec*⟩, ⟨*Finished*⟩*:* All of these cases resulted in the same partition of the set of all browsers, which has the following structure:

- Class I: {Microsoft Internet Explorer, Microsoft Edge}
- Class II: {Firefox, Google Chrome, Opera}

In the case of singleton selections of test sequences of length one, the concepts of test sequence, SCA of strength one and image under the full permutation group all coincide. We conclude that different singleton selections (i.e., different selections of one event), have different differentiation capabilities.

6.3.3 Sequences of length 2. For all subset selections of cardinality two, for our generated SCAs of strength two, the result is the partition $\mathcal{P}_3$.

6.3.4 Sequences of length 3. For all subset selections of cardinality three, for all our generated SCAs of strengths $t \in \{2, 3\}$, the result is the partition $\mathcal{P}_3$.

6.3.5 Sequences of length 4. For all subset selections of cardinality four, for all our generated SCAs of strength $t \in \{2, 3, 4\}$, the result is the partition $\mathcal{P}_3$.

[10] Equality of feature vectors is to be understood as canonical equality between two-dimensional arrays with the same dimensions; i.e., in each position equality for the respective strings holds.

6.3.6 Sequences of length 5. For all subset selections of cardinality five, for all our generated SCAs of strength $t \in \{2, 3, 4, 5\}$, the result is the partition $\mathcal{P}_3$.

6.3.7 Sequences of length 6. For all subset selections of cardinality six, for all our generated SCAs of strength $t \in \{2, 3, 4, 5, 6\}$, the result is the partition $\mathcal{P}_3$.

6.4 Interpretation of results

An analysis of the results in the previous evaluation shows that testing with the selection of only one test sequence of only one event leads to the weakest results in terms of differentiation. It is interesting to note that for some selections of only individual test sequences consisting of at least two events, the resulting partition is equal to $\mathcal{P}_3$. This case even occurs for about 98% of all individual event sequence selections. When considering test sequence selections of cardinality at least two, the resulting partition is also always equal to $\mathcal{P}_3$ and the best distinguishing capabilities obtained in this paper are reached. In particular, whenever there are at least two events appearing in the test sequence and a SCA is chosen as test set, then the best possible partition $\mathcal{P}_3$ is obtained.

Based on these results, we can answer the *research question* regarding the applicability of combinatorial methods to the problem of fingerprinting browsers in the affirmative.

7 THREATS TO VALIDITY

In this section, we comment on possible threats to validity of this work. It is clear that the performed case study is limited, since it only contains five browsers all running on Windows 10 Pro. Another threat stems from the fact that we relied on TLS attacker to act as the server side when the clients (i.e., browsers) attempted the TLS handshake. Likewise, although the goal of the paper is to investigate the applicability of combinatorial methods for fingerprinting, it is clear that a comparison of the proposed methods in this paper with existing approaches for browser fingerprinting would help to properly classify and position this work into the existing literature and methodologies for browser fingerprinting. We plan to conduct such a comparison in future work.

8 CONCLUSION AND FUTURE WORK

In this paper, we used combinatorial methods to create finite non-empty sequences of TLS messages to be used as the underlying means for a method of browser fingerprinting. An experimental evaluation shows that test sets of TLS sequences where the defined order of events compared to the specification is changed, lead to differentiation capabilities.

However, our results also showcase that a refined modelling is needed to strengthen the approach. This model development could be done in parallel to the extension of our set of tested browsers. As briefly mentioned before, the additional manipulation of TLS message contents and the extension to also consider multi-sets of events are directions for future research. Finally, any difference in observed behavior could be analyzed from the point of view of conformance testing, linking browser fingerprinting to problems in the field of conformance testing like undefined behavior or conformance quantification.

ACKNOWLEDGMENTS

The research presented in this paper was carried out in the context of the Austrian COMET K1 program and partly publicly funded by the Austrian Research Promotion Agency (FFG) and the Vienna Business Agency (WAW).

Moreover, this work was performed partly under the following financial assistance award 70NANB18H207 from U.S. Department of Commerce, National Institute of Standards and Technology.

Disclaimer: *Products may be identified in this document, but identification does not imply recommendation or endorsement by NIST, nor that the products identified are necessarily the best available for the purpose.*

REFERENCES

[1] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. 2013. FPDetective: Dusting the Web for Fingerprinters. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 1129–1140. DOI: http://dx.doi.org/10.1145/2508859.2516674

[2] Y. Chee, C. Colbourn, D. Horsley, and J. Zhou. 2013. Sequence Covering Arrays. *SIAM Journal on Discrete Mathematics* 27, 4 (2013), 1844–1861. DOI: http://dx.doi.org/10.1137/120894099 arXiv:https://doi.org/10.1137/120894099

[3] G. Dhadyalla, N. Kumari, and T. Snell. 2014. Combinatorial Testing for an Automotive Hybrid Electric Vehicle Control System: A Case Study. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. 51–57. DOI: http://dx.doi.org/10.1109/ICSTW.2014.6

[4] Tim Dierks and Eric Rescorla. 2008. RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2. https://tools.ietf.org/html/rfc5246. (2008). Accessed: 2019-01-07.

[5] Peter Eckersley. 2010. How Unique Is Your Web Browser?. In *Privacy Enhancing Technologies*, Mikhail J. Atallah and Nicholas J. Hopper (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–18.

[6] Esra Erdem, Katsumi Inoue, Johannes Oetsch, Jörg Pührer, Hans Tompits, and Cemal Yılmaz. 2011. Answer-set programming as a new approach to event-sequence testing. (2011).

[7] David Fifield and Serge Egelman. 2015. Fingerprinting Web Users Through Font Metrics. In *Financial Cryptography and Data Security*, Rainer Böhme and Tatsuaki Okamoto (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 107–124.

[8] M. Z. Mohd Hazli, Z. Kamal Z., and O. Rozmie R. 2012. Sequence-based interaction testing implementation using Bees Algorithm. In *2012 IEEE Symposium on Computers Informatics (ISCI)*. 81–85. DOI: http://dx.doi.org/10.1109/ISCI.2012.6222671

[9] D. R. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker, and Y. Lei. 2012. Combinatorial Methods for Event Sequence Testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 601–609. DOI: http://dx.doi.org/10.1109/ICST.2012.147

[10] H. Mercan and C. Yilmaz. 2014. Pinpointing Failure Inducing Event Orderings. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. 232–237. DOI: http://dx.doi.org/10.1109/ISSREW.2014.34

[11] N. Mouha, M. S. Raunak, D. R. Kuhn, and R. Kacker. 2018. Finding Bugs in Cryptographic Hash Function Implementations. *IEEE Transactions on Reliability* 67, 3 (Sep. 2018), 870–884. DOI: http://dx.doi.org/10.1109/TR.2018.2847247

[12] Keaton Mowery and Hovav Shacham. 2012. Pixel perfect: Fingerprinting canvas in HTML5. *Proceedings of W2SP* (2012), 1–12.

[13] Martin Mulazzani, Philipp Reschl, Markus Huber, Manuel Leithner, Sebastian Schrittwieser, Edgar Weippl, and FC Wien. 2013. Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)*, Vol. 5. Citeseer.

[14] SQLite project. 2019. SQLite. https://www.sqlite.org/index.html. (2019). Accessed: 2019-01-07.

[15] Python Software Foundation. 2019. Python. https://www.python.org/. (2019). Accessed: 2019-01-07.

[16] Research project of the Electronic Frontier Foundation. 2019. Panopticlick. https://panopticlick.eff.org/. (2019). Accessed: 2019-01-07.

[17] Dimitris E Simos, Josip Bozic, Bernhard Garn, Manuel Leithner, Feng Duan, Kristoffer Kleine, Yu Lei, and Franz Wotawa. 2018. Testing TLS using planning-based combinatorial methods and execution framework. *Software Quality Journal* (2018), 1–27.

[18] D. E. Simos, R. Kuhn, A. G. Voyiatzis, and R. Kacker. 2016. Combinatorial methods in security testing. *IEEE Computer* 49 (2016), 40–43.

[19] Juraj Somorovsky. 2016. Systematic Fuzzing and Testing of TLS Libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1492–1504.

[20] Kazuhisa Tanabe, Ryohei Hosoya, and Takamichi Saito. 2019. Combining Features in Browser Fingerprinting. In *Advances on Broadband and Wireless Computing, Communication and Applications*, Leonard Barolli, Fang-Yie Leu, Tomoya Enokido, and Hsing-Chung Chen (Eds.). Springer International Publishing, Cham, 671–681.

[21] The Perl Foundation. 2019. Perl. https://www.perl.org/. (2019). Accessed: 2019-01-07.

[22] Thomas Unger, Martin Mulazzani, Dominik Fruhwirt, Markus Huber, Sebastian Schrittwieser, and Edgar Weippl. 2013. Shpf: Enhancing http (s) session security with browser fingerprinting. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*. IEEE, 255–261.

[23] Z. B. Ratliff. 2018. Black-box Testing Mobile Applications Using Sequence Covering Arrays. (2018). undergraduate thesis, Texas A&M University.

[24] Zachary Ratliff. 2019. CSCM-Tool. https://github.com/zachratliff22/CSCM-Tool. (2019). Accessed: 2019-01-07.