Computers & Security 114 (2022) 102575

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

TC 11 Briefing Papers

A Two-Step TLS-Based Browser fingerprinting approach using combinatorial sequences☆,☆☆

Bernhard Garn[a], Stefan Zauner[a], Dimitris E. Simos[a,*], Manuel Leithner[a], Richard Kuhn[b], Raghu Kacker[b]

[a] SBA Research, Floragasse 7, 1040 Vienna, Austria
[b] NIST, 100 Bureau Drive, Gaithersburg, MD 20899, USA

ARTICLE INFO

Article history:
Received 16 July 2021
Revised 17 November 2021
Accepted 7 December 2021
Available online 28 December 2021

Keywords:
Browser fingerprinting
TLS Protocol
Combinatorial sequences
Anonymity set
Fraud detection

ABSTRACT

We propose a two-step TLS-based fingerprinting approach using combinatorial sequences and properties of TLS handshake messages. Our approach combines fingerprinting based on attributes of the initial ClientHello message with the observed behavior of TLS clients when presented with permuted handshake messages in order to enhance the granularity of the derived fingerprints without increasing the required number of exchanged messages. We conduct a detailed evaluation against 21 browsers and TLS clients on two operating systems. The results show a significant increase in the entropy of the achieved splittings, allowing for a more precise identification of the TLS client than permitted by either of the underlying approaches in isolation.

© 2021 Elsevier Ltd. All rights reserved.

1. Introduction

The term *browser fingerprinting* describes the process and corresponding methods for collecting data about a user's browser and system (Laperdrix et al., 2020), often with the goal of uniquely identifying a browser. In 2010, the results of one of the first large-scale empirical studies on browser fingerprinting reported in Eckersley (2010) pointed towards serious resulting privacy issues. Ensuing research has uncovered a variety of different ways in which browsers – voluntarily or involuntarily – may leak information (Laperdrix et al., 2016).

Fingerprinting finds applications in multiple domains; for example, it can be used to augment the authentication processes of web services (Alaca and Van Oorschot, 2016; Andriamilanto et al., 2021; 2020) or detect likely cases of fraud. More controversial use cases include covert tracking for the purposes of targeted advertising, where its main advantage compared to traditional methods is the absence of identifiers that must be stored in the user's browser (e.g. through the use of cookies).

While early works already identified a plethora of features that can be used to generate a browser fingerprint(Eckersley, 2010), modern web browsers have further expanded this surface by offering additional Application Programming Interfaces (APIs) that expose details of the underlying operating system and hardware. This is especially true for browsers running on mobile devices, where sensors that are typically not found on desktop hardware have been shown to be usable for fingerprinting (e.g., the audio API (Queiroz and Feitosa, 2019), the accelerometer (Bojinov et al., 2014) or the battery API (Olejnik et al., 2017)).

Most of these approaches share a common weakness: They require the use of JavaScript, which can be limited or disabled completely via browser add-ons such as NoScript (Maone, 2012). Similarly, fingerprinting methods that build upon external plugins such as Flash or Java can generally be defeated by disabling these components.

In contrast, the approach presented in this work does not rely on any external components, but merely requires the use of TLS (used in conjunction with HTTP as HTTPS (Rescorla et al., 2000)), a protocol that is widely deployed on the internet today (Felt et al., 2017) and enabled in all modern browsers.

The general domain of browser fingerprinting can be approached from two different and complementary sides: performing empirical observation-based studies via the collection of browser fingerprints in the real world or evaluating fingerprinting ap-

☆ 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

* Corresponding author.
E-mail addresses: BGarn@sba-research.org (B. Garn), SZauner@sba-research.org (S. Zauner), DSimos@sba-research.org (D.E. Simos), MLeithner@sba-research.org (M. Leithner), d.kuhn@nist.gov (R. Kuhn), raghu.kacker@nist.gov (R. Kacker).

https://doi.org/10.1016/j.cose.2021.102575
0167-4048/© 2021 Elsevier Ltd. All rights reserved.

proaches in controlled lab experiments. Related work following the first approach includes (Eckersley, 2010; Gómez-Boix et al., 2018; Laperdrix et al., 2016), while the work presented herein uses the latter paradigm. While observation-based studies generally have access to a large corpus of empirical data, the quality thereof is unknown; in contrast, deriving fingerprints in a lab environment offers less variety, but allows full control over the process of generating fingerprints. It is thus possible to analyze the resulting anonymity sets qualitatively, since it is exactly known which browsers correspond unambiguously to the observed fingerprints. This enables the explicit linking of browsers to their respective fingerprint and also to make fine-grained, dedicated qualitative evaluations for special characteristics or properties of certain browsers only, for example focusing on the observed behavioral differences of browsers from the same browser vendor on different OSs.

In this work, we take the viewpoint of a website operator that has control over one or more servers (see Section 4.1 for a more detailed description of our system and adversary model). A server receives a connection from an unknown browser and the operator aims to identify the corresponding browser vendor, version and platform the connecting browser is running on, based only on the observed behavior of this connecting browser during one or more (possibly incomplete) crafted TLS handshake sequences. Given a sequence of client responses to these handshakes, the goal is to uniquely identify the browser (i.e., client in the TLS handshake) in the sense that there is only one record of a browser in a database that matches the behavior (i.e., emitted error messages).

In Garn et al. (2019), a *behavior-based* approach for browser fingerprinting using only the observed behavior of a browser (i.e., received response messages from a browser by a server) during (possibly) malformed TLS handshakes that might fail to complete was initially proposed. The *goal of the research presented in this work* is to demonstrate how the *combinatorial browser fingerprinting* approach from Garn et al. (2019) can be combined with a state-of-the-art TLS-based browser fingerprinting approach presented in Husák et al. (2016) into a **two-step browser fingerprinting approach** with increased distinguishing capabilities compared to both of its two individual building blocks. We showcase the increased distinguishing capabilities of our proposed two-step approach in an *extended case study* compared to Garn et al. (2019), containing multiple browser families, versions and underlying operating systems (OSs). The case study presented in this work comprises of 21TLS client, mostly browsers, including versions of the well known browsers Firefox and Chromium and 2 OSs (Debian GNU/Linux 10 and Windows 10). It also includes non-browser TLS clients such as wget. An overview of our proposed two-step approach for browser fingerprinting is depicted in Fig. 1. We show that our two-step approach uniquely identifies almost all investigated browsers and TLS clients.

Contribution This work includes the following contributions:

- A two-step browser fingerprinting approach combining a combinatorial behavior-based and a property-based TLS fingerprinting technique with the goal of enhancing the granularity and efficiency;
- Evaluation of a case study consisting of 21 browsers and TLS clients on 2 operating systems using both the approach presented in this work and the two techniques it is based on in isolation.

Our evaluation shows an increase in entropy of the achieved splitting (approximately 9% and 16% increase over the underlying techniques) and a corresponding increase in the number (and decrease in the size) of obtained anonymity sets. At the same time, it is guaranteed not to exceed the number of handshakes required for the underlying approach presented in Garn et al. (2019).

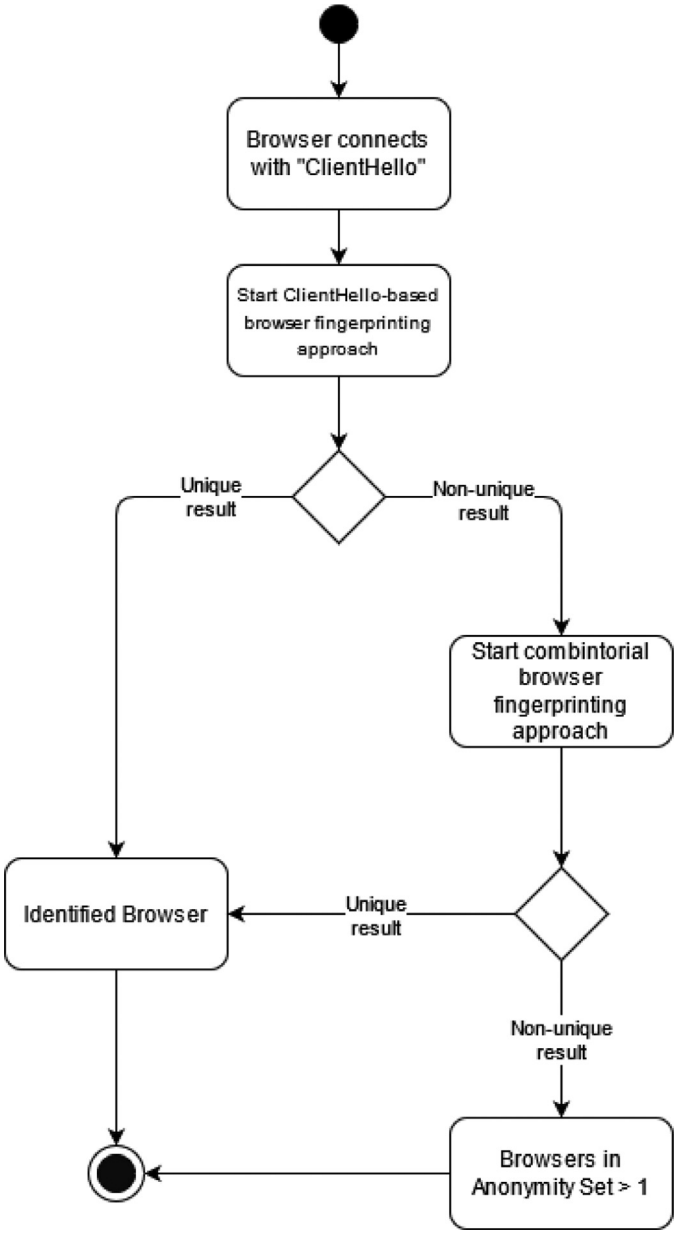

Fig. 1. Schematic overview of the proposed two-step approach.

This paper is structured as follows. In Section 2, we discuss related work, paying particular attention to approaches used later in this work. After some preliminary notions detailed in Section 3, we describe our two-step approach in Section 4. This is followed by an explanation of the implementation of this approach in Section 5. In Section 6, we give the details of the setup of our conducted case study, the results of which are evaluated in Section 7. In Section 8, we comment on potential threats to validity regarding our work, followed by the conclusion in Section 9.

2. Related work

For a recent survey providing a comprehensive treatment of the topic of browser fingerprinting we point to Laperdrix et al. (2020), where different methods of fingerprinting as well as evasion mechanisms are discussed in detail. At the other end of the timescale, one of the first large-scale case studies that examined browser fingerprinting was conducted by Peter Eckersley in 2010 Eckersley (2010). Eckersley showed that 94.2% of browsers

that use Flash or Java could be identified uniquely. However, support for Flash as well as for Java was mostly dropped in favour of JS.

Other large scale case studies were performed in Gómez-Boix et al. (2018) and Laperdrix et al. (2016). In Gómez-Boix et al. (2018), over 2 million unique fingerprints from one of the top 15 French websites were collected and subsequentially processed. The authors showed that there is a difference in browser fingerprints between mobile phones and desktop computers and came to the conclusion that large-scale browser fingerprinting is not as effective as shown in previous studies, where the fingerprints were created using purpose-built websites.

A work in which a purpose-built website was used to collect fingerprints is Laperdrix et al. (2016). Using a domain called *AmIUnique.org*, 118,934 fingerprints were created and analysed and it was shown that HTTP headers and HTML5 canvas fingerprinting play a major role when identifying browers.

2.1. Javascript approaches

JavaScript offers a wide variety of options to fingerprint a browser. Mulazzani et al. used differences in the outcome of JavaScript conformance tests to differentiate between browsers (Mulazzani, Martin and Reschl, Philipp and Huber, Markus and Leithner, Manuel and Schrittwieser, Sebastian and Weippl, Edgar and Wien, FC, 2013), while Mowery et al. found the performance of JavaScript engines suitable for this task (Mowery et al., 2011). Another possibility to leverage JavaScript for browser fingerprinting is the JavaScript *math*-object, as shown in Saito et al. (2018a). Here, differences in the implementation of mathematical operations were used to distinguish between browsers.

In Queiroz and Feitosa (2019), the Audio API is used to distinguish browsers by creating different kinds of periodic waves. Small differences among the same type of wave can then subsequently be used to differentiate between different browsers.

In Olejnik et al. (2017), the authors explain how the battery API, initially designed to expose power management functions of a device, was misused to track users. This API has been removed or limited to internal use by major browser vendors, since it was heavily misused to track users.

The functionality of browsers can be extended through various *extensions*. However, some of these components have been shown to be usable for fingerprinting (Karami et al., 2020; Laperdrix et al., 2021).

In Bursztein et al. (2016), HTML5 canvas functions like *arc(), strokeText(), bezierCurveTo(), quadraticCuve()* were used to find differences in rendering when executed on different browsers, making it possible to distinguish between them.

2.2. Approaches without javascript

The aforementioned techniques rely on the use of JavaScript. However, a handful of approaches that can derive a fingerprint by other means exist.

For instance, a tracking mechanism misusing the ubiquitous browser feature of favicons has been presented in Solomos et al. (2021). Another technique presented by Takei et al. uses CSS features to differentiate between browsers, leaking information regarding the support of specific features by causing background images from known locations to be loaded in case a feature is supported (Takei et al., 2015).

In Walz and Sikora (2018), an evolutionary approach to generate ClientHello messages was used in order to evoke more behavioral discrepancies in TLS implementations. In particular, a behavioral approach is used, where the goal is to reveal implementation flaws and not to identify a certain entity.

Other browser fingerprinting techniques involve the network characteristics while loading a website (Yen et al., 2009).

The browser fingerprinting method given in Husák et al. (2016) relies on TLS handshake characteristics and serves as a building block for the approach presented in this work.

2.3. Evasion

While there are legitimate uses for browser fingerprinting – and this work exclusively focuses on its use for enhancing the security of a web service – the same approaches can be used for more nefarious purposes. Therefore, a variety of methods that aim to allow users to evade fingerprinting and online tracking have been devised throughout recent years. For a systematic literature review of anonymous communication systems, we point towards (Alidoost Nia and Ruiz-Martínez, 2018). In particular, Tor Browser, a modified version of Mozilla Firefox with first-class support for browsing via the Tor low-latency anonymity network, has been suggested (and subsequently investigated) as a means of protecting the privacy of users (Arshad et al., 2021; Cao et al., 2017; Horsman et al., 2019; Jadoon et al., 2019; Saito et al., 2018b).

3. Preliminaries

In this section, we provide some background on several concepts and definitions used throughout this work. Specifically, we give a high-level overview of the TLS protocol in Section 3.1, state the abstract mathematical definition of certain sequence structures arising in discrete mathematics and contextualize their usage in the domain of software testing to this work in Section 3.2. Pertaining to the evaluation of fingerprinting approaches, Section 3.3 details how we use the notion of entropy to assess and compare the results of fingerprinting experiments.

3.1. Transport layer security

The term Transport Layer Security (TLS) describes a set of protocols that are designed to secure the communication between two applications. The most current version of TLS at the time of writing is 1.3 (Rescorla and Dierks, 2018); however, TLS 1.2 is still widely used and also employed in this work.

TLS is used to provide transport security for the communication between browsers and servers on the Internet. For example, TLS is often used in conjunction with HTTP for securely transmitting the webpages a user wants to visit as well as to protect subsequent data exchanges. HTTPS is used by an ever-increasing amount of websites (Felt et al., 2017) and offers confidentiality, integrity and authentication to the participating parties in the communication.

Internally, TLS 1.2 consists of four protocols that work closely together. Three of these protocols form the Handshaking Protocols, while the Record Protocol has no subordinate protocols (Dierks and Rescorla, 2008):

- Handshaking Protocols
  - Handshake Protocol
    The Handshake Protocol is used to negotiate the security parameters of a session and establish a shared secret that is subsequently used to encrypt traffic. In most cases, the server's identity is verified using public key cryptography. This procedure is shown in Fig. 4.
  - Change Cipher Spec Protocol
    This protocol signals that from this point on, another ciphering strategy previously established through the Handshake Protocolis used.
  - Alert Protocol
    The Alert Protocol signals errors between the two peers.

- Record Protocol
  The Record Protocol is responsible for transporting payload data. It fragments the data, optionally compresses it, and applies a Message Authentication Code (MAC) before encrypting and transmitting the result. The same steps are applied in reverse upon receiving record protocol messages. The Handshaking Protocols use this protocol to transmit messages.

3.2. Sequence covering arrays

Sequence Covering Arrays are a class of mathematical artefacts originating in the field of combinatorics. Their fitness for the purpose of testing event-driven software was recognized in Kuhn et al. (2012) and further investigated in Chee et al. (2013). Following the notion commonly used in combinatorial software testing and related approaches (including the browser fingerprinting method proposed in Garn et al. (2019), which is used as a building block for the work presented herein), we will refer to the actual browsers that are being fingerprinted in our case study as *systems under test* (SUTs). The underlying combinatorial structure used for deriving (sequence) test cases can be defined as follows Kuhn et al. (2012):

Definition 1. Let[1] $N, s, t \in \mathbb{N}^{\times}$. A *sequence covering array* (SCA), denoted as $\mathcal{SCA}(N, s, t)$, is an $N \times s$ matrix, where the entries are from a finite set S of s symbols with $t \leq s$, 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. The t symbols in the permutation are not required to be adjacent. That is, for every t-way arrangement[2] $\langle x_1, \ldots, x_t \rangle$ of pairwise different symbols $x_1, \ldots, x_t$, the regular expression $.*x_1.*x_2 \cdots .*x_t.*$ matches at least one row in the array. The parameter t is also called the *strength* of the SCA.

The elements of the set S are often referred to as *events*, due to the temporal interpretation of their appearances in relation to each other in the individual sequences (i.e., rows) of a SCA. Note that SCAs always exist for any given nonempty finite set S of cardinality s and strength $t \in \mathbb{N}^{\times}$ with $t \leq s$, which can be seen by considering an array with rows consisting of all $s!$[3] s-way permutations of all s elements from the set S; however, they are not unique. Hence, one is interested in constructing SCAs with as few rows as possible while still maintaining full t-way permutation coverage.

The problem of constructing SCAs with desirable properties has been approached using various techniques from different fields, including heuristics and (constructive) combinatorial optimization (Banbara et al., 2012; Feng et al., 2012; Kuhn et al., 2012; Margalit, 2013; Mayo et al., 2014; Rahman et al., 2014). Note that for any nonempty finite set S with cardinality $s \geq 2$, a $\mathcal{SCA}(2, s, 2)$ exists: taking any enumeration of the s elements in the set S and then this enumeration with its order reversed and using these two sequences of length s to form a $2 \times s$ array in any order between those two sequences yields an array in which every permutation of every two distinct symbols (i.e., elements of the set S) occurs at least once within at least one row. In this specific construction for strength $t = 2$, every permutation of every two distinct elements appears exactly once in exactly one row of the constructed array.

An example of a SCA with six events and strength $t = 3$ is given in Table 1.

[1] We denote the set of positive integers with $\mathbb{N}^{\times}$.

[2] We denote finite, nonempty sequences in ascending order between angle brackets, e.g. $\langle 0, 1, 2 \rangle$, highlighting that the *order* of the appearing elements is of uppermost importance. We also employ this notation to encode a single permutation of pairwise different elements.

[3] For a positive integer i, we denote with $i!$ the factorial of i.

Table 1
Full 3-way permutation coverage for six events.

Test	Sequences					
1	1	2	3	4	5	6
2	6	5	4	3	2	1
3	4	5	6	1	2	3
4	3	2	1	6	5	4
5	2	6	1	4	3	5
6	5	3	4	1	6	2
7	1	5	6	3	2	4
8	4	2	3	6	5	1
9	3	5	1	4	2	6
10	6	2	4	1	5	3

3.3. Quantifying uniqueness of fingerprints by entropy

We use the notion of *entropy* (Laperdrix et al., 2020) as a quantitative way to assess and compare the overall *distinguishing quality* of the obtained splitting into anonymity sets by derived fingerprints of all browsers for any fingerprinting process. Let $P = (p_i)$ be the probability mass function on the anonymity sets induced by a fingerprinting process based on the individual fingerprints of the considered entities, then its resulting (Shannon) *entropy* is calculated as

$$H = H(P) = -\sum_i p_i \log_2(p_i), \tag{1}$$

where the index of the above sum ranges over all anonymity sets.

To counteract the effects of data sets of different sizes, we also rely on a commonly used *normalized* version of (Shannon) entropy (Laperdrix et al., 2020), which is calculated as

$$\frac{H}{H_{max}}, \tag{2}$$

where $H_{\max}$ equals the binary logarithm of the total number of fingerprinted entities of a dataset and corresponds to the best possibly achievable splitting obtained when each fingerprint is unique among the considered entities.

4. Methodology

In this Section, we present the methodology of our two-step fingerprinting approach. We first elaborate on the underlying system model and adversary model in Section 4.1 before presenting the details of our proposed two-step TLS handshake-based methodology for browser fingerprinting in the following sections. Section 4.2 describes how we combine two individual fingerprinting methods into a two-step approach for browser fingerprinting. Next, in Section 4.3, we give the details of the first fingerprinting step, which uses properties of the ClientHello TLS handshake message to build property-based fingerprints. Finally, in Section 4.4, we explain how certain t-way permutation covering structures arising in discrete mathematics are employed to derive nonempty finite sequences of server-side TLS handshake messages that are subsequently used in the second fingerprinting step to additionally derive behavior-based fingerprints.

4.1. System and adversary model

Our system and adversary model focuses on a website operator aiming to strengthen the security of their authentication mechanism, specifically the use of session identifiers, against an attacker trying to impersonate a logged in user. Note that this is not the sole use case of our approach; as hinted at throughout the introduction of this work, fingerprinting has multiple applications, from

strengthening session security to fraud detection to targeted advertising. While our approach is well-suited to identify individual browsers, their version, and the underlying operating system, it is intended to be used in conjunction with additional identifying factors (e.g. those referenced in Section 2).

In this scenario, three roles exist:

- The **website operator**; this entity controls a web service that is (exclusively) accessible via HTTPS. The website operator can observe all incoming and outgoing traffic on servers used to host the service. They can also deploy additional services and alter the response of the service to a request based on incoming traffic (see Section 5.5 for a practical example of how this might be used when applying our approach). We assume that a common session identifier mechanism is employed on the website; that is, a randomly generated ephemeral secret is stored in the user's browser and submitted along with HTTP(S) requests (e.g. by using cookies).
- The **victim user** is a legitimate user of the web service, who is currently logged in.
- The **adversary** has obtained the victim user's session identifier (e.g. by exploiting a vulnerability in the web service or via social engineering) and is now attempting to impersonate the user by replaying this information. Their goal is to perform privileged actions on behalf of the victim user on the target website.

The adversary is assumed to be an external attacker. No assumptions are made about their network location; any location that could plausibly serve as the origin of a legitimate user (including the victim user's local network) is suitable.

While the adversary knows the user's session identifier, they do not have any further control over the victim's browser. Instead, they add the session identifier to their own browser, which they control to the extent commonly afforded to users. Amongst others, this gives them the ability to adjust any browser preferences they want, including settings that disable JavaScript (which disables most commonly deployed fingerprinting mechanisms) and modify the reported user agent string. The adversary may use any browser, or indeed any client capable of communicating over HTTPS; however, for the approach presented in this work to function properly without modifications, either the victim user, the adversary, or both must use a client/browser that the website operator has information on in their database constructed using the steps described later in this section.

The goal of the website operator is to derive a fingerprint of each user's browser, enabling them to detect that an adversary has hijacked the user's session. Since there exists no widely deployed mechanism for different browsers to share session identifiers, a user changing browser within the same session is considered abnormal behavior and should be reacted to accordingly (e.g. by logging out this session and requiring the user to log in again).

4.2. Two-step approach overview

The idea underlying our proposed two-step browser fingerprinting approach is to apply two TLS handshake-based browser fingerprinting approaches – the property-based approach given in Husák et al. (2016) and the behavior-based method presented in Garn et al. (2019) – consecutively with the goal of achieving better splitting results than either technique in isolation. Each approach independently produces a splitting of the considered browser SUTs into anonymity sets, which can be used together to derive refined anonymity sets. A graphical procedural overview of our proposed two-step approach is depicted in Fig. 1.

The order of how the two approaches from Husák et al. (2016) and Garn et al. (2019) are executed sequentially arises naturally from the data they use for deriving fingerprints:

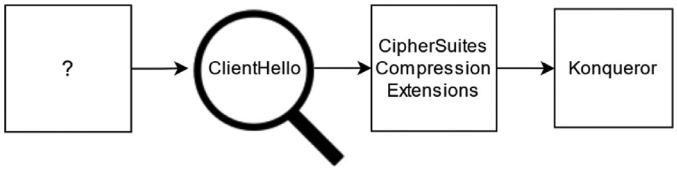

Fig. 2. Example of identification of Konqueror using ClientHello-based browser fingerprinting.

- The approach from Husák et al. (2016) uses only properties of the initial ClientHello message sent by a browser to a server right at the beginning of the TLS handshake. Since this message must be sent in order to initiate an encrypted connection and does not require – or allow for – any intervention by the server, this is considered a passive fingerprinting technique.
- The combinatorial browser fingerprinting approach from Garn et al. (2019) uses messages sent from a browser to a server during one or more neither necessarily complete nor necessarily correct TLS handshakes. In contrast to the passive technique mentioned before, this step involves the server actively selecting sequences of TLS messages to send to the client.

As a TLS handshake always begins with a ClientHello message sent from a browser to the server, we first derive a fingerprint and its associated anonymity set according to the approach given in Husák et al. (2016). Then, depending on whether the resulting anonymity set is a singleton or not, the combinatorial browser fingerprinting approach according to Garn et al. (2019) is executed as a second step with the goal of further narrowing down the anonymity set.

4.3. Clienthello-based browser fingerprinting

In Husák et al. (2016), Husk et al. report that some properties in a conducted TLS handshake differ between HTTPS clients (e.g., the list of supported cipher suites).

A ClientHello TLS handshake message contains the following fields Dierks and Rescorla:

1. ProtocolVersion,
2. Random,
3. SessionID,
4. CipherSuites,
5. CompressionMethods,
6. Extensions.

The ClientHello-based HTTPS client fingerprinting approach described in Husák et al. (2016) uses the three fields CipherSuites, CompressionMethods and Extensions from a ClientHello TLS handshake message to create the fingerprint of a client. The authors found that the CipherSuites list provided the strongest distinguishing capabilities between clients.

In the experiment reported in Husák et al. (2016), sslhaf Ristic (2009), a module for the popular Apache webserver, was utilized to obtain these values. The case where the ClientHello-based browser fingerprinting process leads to a unique browser is visualized in Fig. 2.

Like the aforementioned work, we also make use of a dictionary (i.e., table in a relational database) for storing the derived fingerprints of browsers based on the properties of their initial ClientHello TLS handshake message. Due to the experimental lab setting of this work, we are able to populate this dictionary with

the ClientHello-based fingerprints of all considered browsers, since they are fully known a priori.

4.4. Combinatorial browser fingerprinting

We now detail the combinatorial browser fingerprinting approach proposed in Garn et al. (2019), which uses an abstract model of the TLS handshake to model sequences of messages.

4.4.1. Abstract modeling

To cast the TLS handshake into a *software testing* setting within which the fingerprints for the combinatorial browser fingerprinting approach are created, we regard the exchanged messages during a TLS handshake as abstract events. A TLS handshake conforming to the specification is depicted in Fig. 4. Since we are only interested in fingerprinting browsers (i.e., TLS clients), we exclusively concern ourselves with the modelling of server-side TLS handshake messages as abstract events in this work.

We regard the six TLS messages sent by a server to a browser as part of a correct TLS handshake as six abstract events and collect them into a set $\mathcal{E}$:

$$\mathcal{E} = \{\text{ServerHello}, \text{Certificate}, \tag{3a}$$

$$\text{ECDHEServerKeyExchange}, \text{ServerHelloDone}, \tag{3b}$$

$$\text{ChangeCipherSpec}, \text{Finished}\}. \tag{3c}$$

These events are mapped to the following non-negative integers:

0. ServerHello,
1. Certificate,
2. ECDHEServerKeyExchange,
3. ServerHelloDone,
4. ChangeCipherSpec,
5. Finished.

The TLS specification defines only one way for a *correct* TLS handshake to be conducted, therefore any divergence during the actual execution of a TLS handshake should be recognized as such and reacted to accordingly.

There are many conceivable ways on how to create TLS handshake message sequences that do not conform to the specification; in this work, we leverage SCAs for different numbers of events and strengths.

To this end, we define a nonempty finite set $\mathcal{S}$ of permutation sequences of different lengths of events from the set $\mathcal{E}$. Formally, for all $\kappa \in \{1, 2, 3, 4, 5, 6\}$, for all subsets Δ of $\mathcal{E}$ of cardinality κ (i.e., $\Delta \subseteq \mathcal{E}$ and $|\Delta| = \kappa$), consider the image $Sym_\kappa(\Delta)$ of Δ under the full symmetric group of κ elements and collect all of these permutations in a set $\mathcal{S}$, i.e., we have

$$\mathcal{S} = \bigcup_{\substack{\kappa \in \{1,2,\ldots,6\} \\ \Delta \subseteq \mathcal{E} \wedge |\Delta| = \kappa}} Sym_\kappa(\Delta). \tag{4}$$

In other words, the set $\mathcal{S}$ exactly consists of all permutations of any combination (i.e., unordered selection) of one to six events from the set $\mathcal{E}$. It follows that there are

$$\sum_{i=1}^{6} \binom{6}{i} \cdot i! = 1956 \tag{5}$$

generated individual sequences in total; i.e., $|\mathcal{S}| = 1956$. All of these sequences determine a specific TLS handshake by containing server-side TLS handshake messages in some order and most of these generated sequences of server-side TLS handshake messages will be invalid according to the TLS specification.

We remark that for fixed $\ell \in \{1, 2, 3, 4, 5, 6\}$ and fixed unordered ℓ-combination $C = \{e_1, \ldots, e_\ell\} \subseteq \mathcal{E}$, all SCAs of strength $t \in \mathbb{N}^\times$ for all $t \leq \ell$ with symbol set C appear – when considered as a set of sequences instead of an array – as a subset of the set $\mathcal{S}$.

4.4.2. Browser fingerprints via combinatorial sequences

In combinatorial browser fingerprinting, the TLS messages transmitted from the server to the client as part of a handshake as well as their order are determined by the individual sequences (i.e., rows) in an instantiated SCA of some strength t that uses a nonempty subset of the set $\mathcal{E}$ of events as symbols. A fingerprint of a browser is constructed based on the entire observed behavior of a browser (i.e., client-side TLS handshake messages sent from the browser during a handshake). The experimental lab setting of this work makes it possible to precisely control the execution of these TLS handshakes inside the testing infrastructure. When a browser is instrumented to connect to the server via TLS, the server replies with the messages contained in a single sequence $\sigma \in \mathcal{S}$ selected by the testing infrastructure. A conceptual overview of the combinatorial browser fingerprinting methodology is visualized in Fig. 3.

Since the sequences $\sigma \in \mathcal{S}$ might not be complete and their events might not be in the correct order according to the TLS specification, any such TLS handshake with a browser might fail at any point during their execution. This outcome is expected and intended, because the responses of the browser are used to build its resulting fingerprint. In particular, the messages sent from a browser to the server during the execution of a potentially *malformed* TLS handshake may contain error or alert messages that are specific to this browser alone, i.e., in the best case these response messages uniquely identify the emitting browser. The complete fingerprint of a browser consists of the vectorized interpretation of the concatenation of all responses received from the client during the attempted execution of one or more potentially malformed TLS handshakes determined by selected sequences in the set $\mathcal{S}$.

Precomputing the behavior of all browsers against all individual $\sigma \in \mathcal{S}$ (i.e., attempting the TLS handshake encoded by a sequence in $\mathcal{S}$ once) in a database has the advantage that complete fingerprints of browsers based on SCAs can then be constructed with a database lookup, since all SCAs with server-side TLS handshake messages as symbol set used for combinatorial browser fingerprinting appear as a subset of the set $\mathcal{S}$ and fingerprints of browsers for nonempty non-singleton subsets of the set $\mathcal{S}$ are composed of the recorded behavior of browsers against their individual members (i.e., sequences).

The *combinatorial browser fingerprinting* approach uses the previously described abstract modelling and constructed set $\mathcal{S}$ of server-side TLS handshake message sequences and works as follows: For a chosen nonempty subset $\mathcal{F}$ of the set $\mathcal{S}$ of all generated server-side TLS handshake message sequences (i.e., $\emptyset \neq \mathcal{F} \subseteq \mathcal{S}$) and given browser, a fingerprint of that individual browser with respect to $\mathcal{F}$ is derived. Each individual element in $\mathcal{F}$ (i.e., nonempty sequence of server-side TLS handshake messages) encodes a TLS handshake attempt to be performed, which will give rise to a part of the overall derived fingerprint for the browser as outlined above. Specifically, for every sequence σ in the set $\mathcal{F}$, the TLS handshake determined by σ has to be executed and the resulting behavior of the browser recorded. Note that the existence of the pre-populated database described in the previous paragraph facilitates this step, since all needed behaviors of browsers against all sequencen in the set $\mathcal{S}$ can be obtained from this database, which increases the efficiency of the combinatorial browser fingerprinting approach. The overall fingerprint of a browser based on $\mathcal{F}$ contains $|\mathcal{F}|$ parts, which are the individual behaviors of that browser during the attempted TLS handshakes encoded in the elements in $\mathcal{F}$. Note that

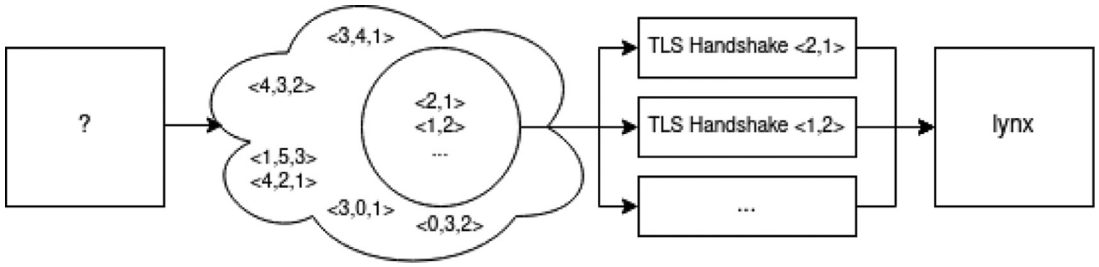

Fig. 3. Example of identification of lynx using combinatorial browser fingerprinting.

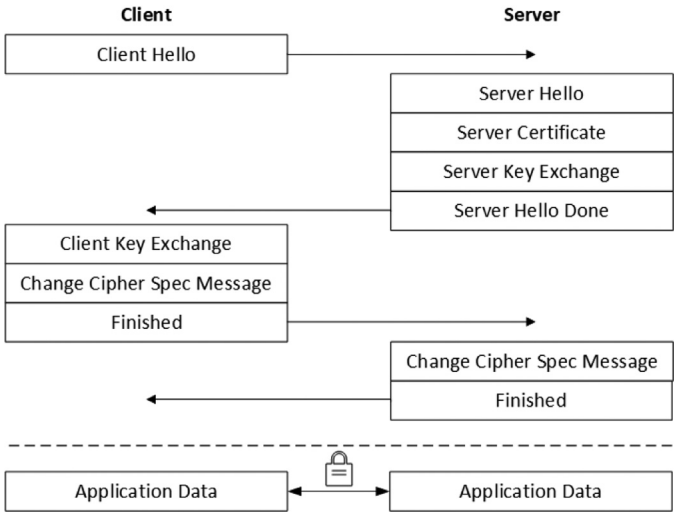

Fig. 4. A TLS 1.2 handshake conforming to the specification.

for all sets $\emptyset \neq \mathcal{F} \subseteq \mathcal{S}$, the order of concatenation of the responses of the individual test sequences (i.e., parts of the overall fingerprint) has to be uniform for all tested browsers to be able to compare the resulting overall fingerprints. Note further that for every nonempty subset C of the set $\mathcal{E}$ of events (i.e., $\emptyset \neq C \subseteq \mathcal{E}$) and every $t \in \mathbb{N}^{\times}$ with $t \leq |C|$, the set $\mathcal{F}$ can be chosen in such a way that it constitutes a SCA of strength t for the set C of symbols. In particular, for given browser and SCA, the prepopulated database containing all behaviors of all browsers against all potentially malformed TLS handshakes can be queried to generate the fingerprint for the given browser with respect to the SCA.

The reason why SCAs are appealing for use as underlying structure in behavioral TLS-based fingerprinting is twofold: First, their guaranteed coverage of t-way permutations leads to higher diversity amongst the constructed rows, which correlates with a greater ability to distinguish between browsers. Second, algorithms geared towards generating these structures are optimized for minimizing the size (i.e., number of rows) of their result, leading a the efficient inducement of diverse error-reporting behavior.

4.5. Approximate costs

We state below approximations of the cost in terms of space and time for our proposed two-step approach as well as for its two constituting individual parts.

4.5.1. Clienthello-based fingerprinting

Space. In the ClientHello-based fingerprinting approach, the fingerprint of a browser is derived from values reported in the ClientHello TLS-handshake message. Therefore, under the assumption that space requirements for storing a pair consisting of a browser and its fingerprint can be uniformly (i.e., independent from the browser) bounded, then the overall space needed for the ClientHello-based fingerprinting approach scales linearly in the number of browsers. These costs correspond to the built-up of the dictionary.

Time. The ClientHello-based fingerprint of a browser can be derived from one conducted TLS handshake. Therefore, under the assumption that the time needed to execute one TLS handshake can be uniformly (i.e., independent from the browser) bounded, the overall time needed in the ClientHello-based fingerprinting approach to derive fingerprints scales linearly in the number of browsers. These costs accrue during the creation of the dictionary. When the ClientHello-based fingerprinting approach is executed in practice, the time to query the dictionary for a derived fingerprinting has to be considered additionally. Assuming uniformly bounded time costs for database queries implies linear scaling time costs for conducting the fingerprinting process.

4.5.2. Browser fingerprints via combinatorial sequences

Space. Given the abstract model of the TLS handshake considered in this work, it follows that the number of rows in any reasonably optimized (i.e. no duplicate rows) SCA used for deriving fingerprints based on combinatorial sequences is bounded by $6! = 720$. Under the assumption that the responses received from a browser by a server during any TLS handshake is bounded uniformly (i.e., independent from the browser), any fingerprint created by combinatorial sequences is also bounded. After the creation of this fingerprint, it could further be hashed to save space in the dictionary.

Time. Every derivation of the fingerprint of any browser via a SCA involves not strictly more than 720 TLS handshakes. Therefore, under the assumption that the execution of a TLS handshake can be bounded in time, the time cost for creating the fingerprint of a browser via a SCA scales linearly in the number of browsers.

4.5.3. Two-step approach

Space. In addition to the space requirements of the dictionaries of the ClientHello-based and the combinatorial sequences-based browser fingerprinting approaches, in our proposed two-step approach it is necessary to also store the respective anonymity sets.

Time. Given the dictionaries of both constituting approaches, it is essential to compute the actual anonymity sets. This computation is bounded quadratically in the number of browsers.

Given the splitting that is achieved by the ClientHello-based fingerprinting approach, it is then necessary, for every non-singleton anonymity set, to precompute a strategy (optimized in the number of iterations) with which SCAs to continue the fingerprinting. There are at most 192 SCAs available, each with a size less than or equal to 720. Therefore, under the assumption that the execution time of a TLS handshake can be uniformly bounded, the time costs of the second fingerprinting step can also be bounded.

Practical experiments. According to our experiments, executing a single TLS handshake requires a mean time of 1506 ms, excluding

any additional latency introduced by the network. While modern browsers make multiple requests in parallel, there is a limit to this concurrency (e.g. Mozilla Firefox will perform a maximum of six requests in parallel by default at the time of writing). In contrast to the original behavioral approach presented in Garn et al. (2019), the technique detailed in this work effectively narrows down the required TLS handshakes based on the properties of the initial ClientHello message. This potentially reduces (and is guaranteed not to increase) the amount of network round trips and exchanged messages required to execute the overall process.

5. Implementation & data processing

This section details the implementation of the proposed two-step approach, beginning with the ClientHello-based approach in Section 5.1, followed by the combinatorial fingerprinting step in Section 5.2. In Section 5.3, we expand on the steps used to process the data obtained thus far. We give a walkthrough example for the database generation and processing necessary for our proposed two-step approach in Section 5.4, followed by an example of how the approach can be deployed on real-world sites (Section 5.5).

5.1. Implementation of clienthello-based browser fingerprinting approach

We implemented the ClientHello-based browser fingerprinting approach in the same way as originally described in Husák et al. (2016) in order to recreate their browser fingerprinting approach as closely as possible. To this end, we also use the Apache web server together with the plugin sslhaf. In our implementation, this plugin parses and stores the values of the client-side ClientHello TLS handshake message received by the server from a connecting browser on the web server in a log file. The data in these log files is subsequently extracted by our testing infrastructure for storing the hash of the derived ClientHello-based fingerprints in a database.

The process for creating the ClientHello-based fingerprint of a browser in our testing infrastructure works in detail as follows:

1. A browser is selected for ClientHello-based fingerprinting.
2. The browser is started manually and connects via a script to the Apache webserver running sslhaf.
3. sslhaf parses the values from the ClientHello-message sent by the browser to the server and writes the obtained data into a log file.
4. The testing infrastructure extracts the necessary data for creating the ClientHello-based fingerprint for that browser from this log file, creates the ClientHello-based fingerprint for that browser and then a hash of the resulting fingerprint is stored manually in the database.

5.2. Implementation of combinatorial browser fingerprinting approach

This section describes the research prototype tool developed to perform the combinatorial browser fingerprinting experiments in this work. It is based on the techniques described in Garn et al. (2019) and was implemented in the Java programming language. Our tool COMFIT (COMbinatorial FIngerprinting Tool) is based on TLS attacker Somorovsky (2016), which was designed to test TLS libraries and is capable of sending *TLS messages with arbitrarily modified message contents in custom TLS handshake message flows* (i.e., TLS handshake message sequences). It can be configured using an XML file, which contains the order in which the specified server-side TLS handshake messages shall be sent to the TLS client. Note that in this work we only manipulate the order of sent server-side TLS handshake messages via combinatorial sequence methods and use the default values provided by TLS attacker for the actual message content creation.

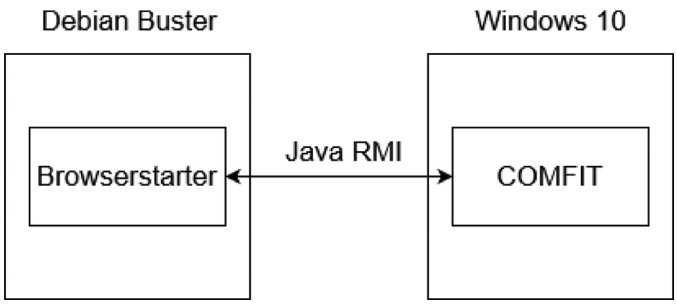

Fig. 5. BROWSERSTARTER and COMFIT.

In Garn et al. (2019), the fingerprinting of browsers was done on a single virtual machine (VM) running Windows 10. All browsers considered in Garn et al. (2019) were installed in this VM and connected to a local instance of TLS attacker in server mode inside the VM. This approach was initially also used for this work; however, when analyzing the results from some initial fingerprinting experiments with the combinatorial approach on Linux and Windows, we recognized that some exceptions thrown by the used Java version 8u191 only occurred on Linux but never on Windows. Therefore, to be sure to obtain consistent and comparable results, the operating system on which COMFIT is executed has to be fixed, while it still needs to be able to start a browser on another machine running a different operating system. This was achieved by developing an additional piece of software called BROWSERSTARTER, which starts and stops browser processes on remote machines. It is controlled by COMFIT using Java Remote Method Invocation (RMI) Corporation (2020c) as depicted in Fig. 5.

For a given TLS handshake and any considered browser, the following procedure is executed by COMFIT to derive the behavior of this browser against the TLS handshake determined by the sequence σ:

1. COMFIT starts TLS attacker in server mode within a new process and attaches to its *stdout* and *stderr*.
2. TLS attacker reads the XML file containing the sequence σ, from which the events are determined and the order in which they shall be sent during the attempted TLS handshake. Afterwards, COMFIT waits for an incoming connection (i.e., `ClientHello` TLS handshake message).
3. COMFIT instruments BROWSERSTARTER to start the browser and connect to TLS attacker.
4. TLS attacker receives a ClientHello from the browser and responds with the messages and the order in which they are given in the XML file corresponding to the sequence σ.
5. The complete output from TLS attacker is recorded and parsed by COMFIT during the execution of the TLS handshake until all messages from the attempted handshake have been exchanged or the browser closes the connection.
6. COMFIT stores the obtained information in the database.

Recall that fingerprints of browsers for nonempty non-singleton subsets of the set $\mathcal{S}$ – this includes SCAs with at least two symbols and at least strength two – are composed of the observed behavior of browsers against their members, therefore all these fingerprints can be constructed according to Section 4.4.2 based on data in the database as described above.

5.3. Data processing

In this section, we describe how the arising data in both the ClientHello- and the combinatorial-based fingerprinting approaches is processed.

Table 2
First step: Using the ClientHello message values to distinguish browsers .

Browser	SHA-1 Hash
Firefox 68.4.1esr (64-Bit)	e99e561504b91b ...
Firefox 70.0.1	e99e561504b91b ...
NetSurf 3.6-3.2	ff8cb795592e4e ...

5.3.1. Clienthello based fingerprinting

Modern browser use GREASE (Benjamin, 2020), a mechanism to prevent extensibility failures in the TLS ecosystem, which introduces a certain randomness to the ClientHello TLS handshake message. Therefore, GREASE-values have to be removed from the ClientHello message when deriving fingerprints in the ClientHello-based fingerprinting approach.

The actual computation of the anonymity sets for the ClientHello-based browser fingerprinting approach was done in a.net core program, which queries the database for the fingerprints of browsers, hashes them with SHA256, and subsequently compares and groups them to obtain the resulting splitting.

5.3.2. Combinatorial browser fingerprinting

Recall that the database contains the precomputed behavior of all browsers against all sequences in the set $\mathcal{S}$. To derive fingerprints of browsers for nonempty non-singleton subsets of the set $\mathcal{S}$, the data from the database was processed using Pandas, an open source data analysis tool for the Python programming language. The processing performed in this work is similar to the one performed in Garn et al. (2019), but had to be enhanced to cope with the greater number of browsers in the case study of this work.

For any $\emptyset \neq \mathcal{F} \subseteq \mathcal{S}$, the process of creating the respective fingerprint for all $\mathbb{N}^{\times} \ni \beta$ considered browsers and the corresponding splitting into the resulting anonymity sets works as follows:

- The fingerprint of each considered browser is created by concatenating the values obtained from a lookup to the database for the behavior of each browser against the elements of $\mathcal{F}$ as described in Section 4.4.2. These β fingerprints are then processed further in Python, where a Pandas `DataFrame`-object (pandas development team, 2020) (a two-dimensional labelled data structure) is used to hold the built fingerprint with respect to the set $\mathcal{F}$ of every considered browser in a dedicated row. Thus, it follows that this `DataFrame` consists of β rows and $5 \cdot |\mathcal{F}|$ columns and contains string values.
- The `DataFrame` is then grouped by the constructed fingerprint. Each group constitutes an *anonymity set*.

5.4. Walkthrough example

In this section, we provide a complete step-by-step walkthrough example for the proposed two-step approach for browser fingerprinting.

The example discussed here is based on the following three browsers:

- Firefox 68.4.1esr (64-Bit) running on Linux,
- Firefox 70.0.1 running on Windows,
- NetSurf 3.6-3.2 running on Linux.

Clearly, the best outcome would be if our proposed two-step approach was able to split these three browsers into three singleton anonymity sets.

In the first step, our testing infrastructure controls and instruments the three browsers to start a TLS handshake with a dedicated server, which is responsible for deriving their fingerprints based upon the ClientHello approach. Table 2 shows the result of this first step.

Table 3
SCA of strength two, comprising of only two event sequences σ_1 and σ_2.

Test ID	Custom TLS Handshake Sequence
σ_1	⟨ ServerHello, ServerHelloDone, ECDHEServerKeyExchange, Certificate ⟩
σ_2	⟨ Certificate, ECDHEServerKeyExchange, ServerHelloDone, ServerHello ⟩

Table 4
Responses of the browser Firefox 68.4.1esr (64-Bit) against the TLS handshake sequence σ_1.

Name	Value
RM1	'CLIENT_HELLO'
RM2	'AlertMessage'
AM	'UNEXPECTED_Message'
EH	' '
EPH	'Connection reset by peer: socket write error'

Table 5
Responses of the browser Firefox 68.4.1esr (64-Bit) for the TLS handshake sequence σ_2.

Name	Value
RM1	'CLIENT_HELLO'
RM2	' '
AM	' '
EH	' '
EPH	'Connection reset by peer: socket write error'

As can be seen from Table 2, NetSurf 3.6-3.2 was uniquely identified based on the value of its hashed fingerprint. However, it was not possible to distinguish between the two Firefox browsers. Therefore, the second step employs the combinatorial browser fingerprinting approach with the goal of splitting this anonymity set. Next, we consider the derivation of the fingerprint for the browser Firefox 68.4.1esr (64-Bit) for a SCA of strength two for the following set of four server-side TLS handshake messages:

$$C = \{\text{ServerHello}, \text{ServerHelloDone}, \tag{6a}$$

$$\text{ECDHEServerKeyExchange}, \text{Certificate}\} \subseteq \mathcal{E}. \tag{6b}$$

As explained in Section 3.2, there always exists a SCA of size (i.e., number of rows) two for any nonempty set of events and one such SCA can be constructed by taking as rows any enumeration of these events as one row and then this enumeration with its order reversed as the other row. We make use of this construction in this example and hence obtain the two sequences listed in Table 3 as a SCA for the symbol set C of strength two. Note that $\sigma_1, \sigma_2 \in \mathcal{S}$ holds for the two sequences given in Table 3.

Each of the two sequences listed in Table 3 encodes the messages transmitted from the server to the client as part of a TLS handshake. Upon receiving a response from the client, COMFIT records the five strings reported from TLS attacker and stores them in the database. For the sequence σ_1, these five values are listed in Table 4 and for the sequence σ_2 in Table 5.

Finally, the complete fingerprint derived for the SCA given in Table 3 is generated by concatenating the two captured behaviors (i.e., groups of five strings each) from the two sequences σ_1 and σ_2 from Table 4 and Table 5 in this order, producing the fingerprint given in Table 6.

The same process is exercised for the browser Firefox 70.0.1 and yields the fingerprint given in Table 7.

Since the two derived fingerprints in Table 7 and Table 6 differ, the two-step approach was successful in splitting the set of three considered browsers into three singleton anonymity sets.

Table 6
Complete feature vector for Firefox 68.4.1esr (64-Bit) on Debian.

Name	Value
RM1	'CLIENT_HELLO'
RM2	'AlertMessage'
AM	'UNEXPECTED_Message'
EH	''
EPH	'Connection reset by peer: socket write error'
RM1	'CLIENT_HELLO'
RM2	''
AM	''
EH	''
EPH	'Connection reset by peer: socket write error'

Table 7
Complete feature vector for Firefox 70.0.1 on Microsoft Windows 10 .

Name	Value
RM1	'CLIENT_HELLO'
RM2	''
AM	''
EH	''
EPH	'Connection reset by peer: socket write error'
RM1	'CLIENT_HELLO'
RM2	''
AM	''
EH	''
EPH	'Connection reset by peer: socket write error'

5.5. Real-world deployment

In this section, we describe how a real-world server operator might implement our approach in order to identify specific clients. Note that significant variations are permitted, depending on the environment, needs and preferences of the implementing entity. Following the execution of the steps in Section 5.4 using all browsers to be identified, we assume that the following components are already in place:

- A *ClientHello dictionary*, mapping a ClientHello-based fingerprint to the corresponding anonymity set of browsers.
- A *SCA dictionary*, mapping each of the non-singleton anonymity sets to a vector of identifiers of TLS handshake sequences (i.e. an instantiated SCA) suitable to further split these anonymity sets using the combinatorial browser fingerprinting approach.
- A *handshake resource dictionary* that maps a specific TLS handshake sequence identifier to a URL that serves this handshake sequence.
- A *fingerprint dictionary* that maps the combination of a ClientHello-based fingerprint and the feature vectors derived from the executed TLS handshake sequences to an anonymity set of browsers. The key could be created by e.g. starting with the ClientHello-based fingerprint, concatenating the names and values of all elements in each feature vector (in order), and taking a hash of the resulting string.

5.5.1. Handshake resource setup

For each entry in the handshake resource dictionary, the server must set up a TLS endpoint (e.g. an instance of COMFIT with the appropriate configuration) reachable under the URL given in the dictionary entry. In real-world scenarios, these endpoints would likely run on the same host, but differ in their port; for instance, all requests to `https://example.com:8081` would result in the server replying with σ_1, while `https://example.com:8082` would respond with the messages in σ_2, etc.

5.5.2. Process

We assume that a client is requesting a web site `https://example.com` served via TLS under the control of the server operator. This initial request contains a ClientHello message, which is used to retrieve the corresponding anonymity set from the ClientHello dictionary.

If the anonymity set is a singleton, the fingerprinting process is complete and no further steps are necessary. Otherwise, the server selects the corresponding list of TLS handshakes from the SCA dictionary, which is then used to retrieve a list of URLs from the handshake resource dictionary.

The server then modifies the HTML code returned in response to the original request to `https://example.com`, embedding a HTML <img>tag for each of the resource URLs, with the `src` attribute set to the respective URL.

Upon parsing the returned page, the client will proceed to try to load each of those images, resulting in one TLS handshake for each image, allowing the respective COMFIT instances to collect the feature vectors for this client. The feature vectors can then be combined with the ClientHello property hash in order to look up the minimal anonymity set from the fingerprint dictionary.

5.5.3. Example

For instance, assume that the ClientHello property hash is `b8473b86d4c2072ca9b08bd28e373e8253e865c4`, and assume that the corresponding anonymity set is not a singleton.

The server retrieves the corresponding TLS handshake sequences from the SCA dictionary (we will call them σ_{123} and σ_{200} in this example) and maps them to their resource URLs using the handshake resource dictionary; here, we assume that σ_{123} maps to `https://example.com:8203` and σ_{200} corresponds to `https://example.com:8280`.

Subsequently, the server replies to the original request to `https://example.com` with a page that includes the following content:

```
<img src=''https://example.com:8203/img.jpg''>
<img src=''https://example.com:8280/img.jpg''>
```

This causes the client to make additional requests to port 8203 and 8280, each of which is served by a COMFIT instance that will execute σ_{123} and σ_{200}, respectively. The COMFIT instance then stores the resulting feature vector. Once all requests have been made, the stored feature vectors can be combined with the ClientHello property hash, enabling the server to look up the minimal anonymity set in the fingerprint dictionary.

6. Case study

In this section, we provide details on the case study and its underlying infrastructure. While the ClientHello-based fingerprinting that constitutes the first step of our approach permits no additional choices by the server, the combinatorial fingerprinting approach requires us to choose a set of SCAs, which we describe in Section 6.1. Next, we list the considered browsers for fingerprinting in Section 6.2. Details on the virtualized infrastructure used for this case study are provided in Section 6.3.

6.1. SCAs For combinatorial fingerprinting

In total, we performed 192 independent fingerprinting experiments with the combinatorial browser fingerprinting approach based on SCAs with each one consisting of all 21 browsers considered in this work.

Specifically, for every $\kappa \in \{1, 2, 3, 4, 5, 6\}$, for every $\tau \in \mathbb{N}^{\times}$ with $2 \leq \tau \leq \kappa$, we generated a SCA for the symbol set $\{0, \ldots, \kappa - 1\}$ of

Table 8
Number of rows in the generated SCAs.

t	1	2	3	4	5	6
s						
1	1					
2	1	2				
3	1	2	6			
4	1	2	8	24		
5	1	2	10	28	120	
6	1	2	10	36	156	720

cardinality κ of strength τ using a Python implementation of the algorithm given in Kuhn et al. (2012). For SCAs of strength one, we simply used the ascending enumeration of the set $\{0, \ldots, \kappa - 1\}$. The sizes of these SCAs are listed in Table 8. Based on these abstract SCAs, we generated *instantiated SCAs* used in the combinatorial browser fingerprinting experiments as follows: for all $\ell \in \{1, 2, 3, 4, 5, 6\}$, for all subsets $C = \{e_1, \ldots, e_\ell\} \subseteq \mathcal{E}$ of events of cardinality ℓ, for all $t \in \mathbb{N}^\times$ with $1 \leq t \leq \ell$, replace the entries in the generated SCA for ℓ symbols of strength t with the events in the set C according some fixed bijection. It follows that the resulting *instantiated array* is a SCA for the symbol set C of strength t. Thus, it follows that in this work we consider in total

$$192 = \sum_{i=1}^{6} \binom{6}{i} \cdot i \tag{7}$$

fingerprinting experiments based on SCAs.

6.2. Tested browsers

The case study presented in this work comprises multiple browsers and other TLS clients from various vendors for two operating systems. Besides widely known browsers, this includes Konqueror, epiphany, qutebrowser, NetSurf and surf running on Linux, the text-based browser lynx running on Linux, the programs wget and curl running on Linux and the dedicated TLS clients provided by the TLS libraries gnutls and openssl. The case study was designed to include a wide range of diverse types of software that can act as a TLS client and to cover both Linux and Windows OS. The complete list of tested browsers and their versions is given in Table 9.

6.3. Setup

All experiments were run in a virtualized environment using (Corporation, 2020a) in version 6.1.10_Ubuntu_r138449 on a Ubuntu 20.04.1 LTS host. In total, three different virtual machines were used, as listed in Table 10.

The VM running Microsoft Windows was used to execute COMFIT, which ultimately created the fingerprints of all browsers that connected to it. COMFIT used BROWSERSTARTER to control the browsers installed on the Debian-based VMs. Browsers that were installed on the Microsoft-based VM were started and stopped directly by COMFIT.

As NetSurf could only be started on Debian when using the GNOME-Classic desktop environment, two otherwise identical VMs were created, one using the more modern GNOME interface and one using its legacy counterpart. The latter VM was also used to fingerprint the 21 browsers using the ClientHello-based approach.

Regarding the overall setup of the case study, we have made use of some auxiliary software, e.g. scripting languages and libraries, for various tasks (see Table 11). Since COMFIT and BROWSERSTARTER are based on Java, it had to be installed on every VM. SQLite was used on the windows-based-, and on the Debian based GNOME-Classic machine to store the results generated by COMFIT and from the ClientHello fingerprinting from our case study; while Perl, Python,.net core and Pandas (a data analysis tool based on Python) were used to process the collected data.

The overall setup of our testbed infrastructure is depicted in Fig. 6. All VMs were connected using the *Internal Network* option of VirtualBox, which allows communication between VMs on the same host system, connected to the same internal network (Corporation, 2020b). Consequently, the three VMs were able to communicate which each other, but had no connectivity to other hosts.

The previously explained setup enabled COMFIT to test each one of the 21 browser listed in Table 9 against each one of the 1956 individual test sequences independently (i.e., TLS handshakes determined by the members of the set $\mathcal{S}$) and record their behavior, regardless of the VM and OS each browser was installed in and running on. Similarly, the setup also enabled the seamless creation of the ClientHello-based fingerprints for all 21 browsers.

7. Evaluation

In this section, we present an evaluation of the browser fingerprinting experiments performed in the case study. We begin with the individual results of the ClientHello approach in Section 7.1 and turn to the individual results of the combinatorial browser fingerprinting approach in Section 7.2. Then, in Section 7.3, we highlight the results of our proposed two-step approach, where we also compare its achieved results with those of its two building blocks. For all three approaches, we report qualitative (in terms of members of anonymity sets) and quantitative (in terms of individual cardinalities and total number of anonymity sets in partitions of browsers as well as the entropy and normalized entropy of the partitions) measures about the achieved splitting results.

7.1. Results of clienthello-based browser fingerprinting

The ClientHello-based browser fingerprinting approach yielded 16 anonymity sets, which are listed in Table 12. This splitting consists of 13 singletons, two anonymity sets of cardinality two and one anonymity set with cardinality four and achieved a value of 3.8208 for entropy and 0.8698 for normalized entropy.

7.2. Results of combinatorial browser fingerprinting

The 192 independent fingerprinting experiments with the combinatorial browser fingerprinting approach yielded different splitting results (i.e., different partitions of all 21 browsers into anonymity sets) depending on the instantiated SCA used for encoding the TLS handshakes upon which the classification of browsers into anonymity sets is being done.

Due to the high number of experiments, we only summarize the most important conclusions from the combinatorial browser fingerprinting approach here in this section and present the detailed results in Appendix A.

The three primary properties of the instantiated SCAs, i.e., the chosen positive cardinality of defining symbol set, the chosen corresponding nonempty event subset selection as symbols from the set $\mathcal{E}$ of events and the chosen strength of the SCA, all have an impact on the resulting splitting.

Overall, the obtained results for different instantiated SCAs exhibit the behavior that one would expect, e.g., that performing combinatorial browser fingerprinting with higher strengths or more events would generally lead to a better splitting. Therefore, based on the observed results in these 192 combinatorial browser fingerprinting experiments, we formulate several hypothesis and interpretations about the impact of various combinatorial properties of the instantiated SCAs used to determine the experiments on

Table 9
Tested browsers in the extended case study compared to Garn et al. (2019).

Browser	Version	Operating System	resource
surf	0.7-2	Debian 10 GNOME	debian packages
Chromium	78.0.3904.108	Debian 10 GNOME	debian packages
lynx	2.8.9rel.1–3	Debian 10 GNOME	debian packages
Mozilla Firefox	68.4.1esr (64-Bit)	Debian 10 GNOME	debian packages
Konqueror	4:18.12.0–1	Debian 10 GNOME	debian packages
epiphany	3.32.1.2-3 deb10u1	Debian 10 GNOME	debian packages
qutebrowser	1.6.1–2	Debian 10 GNOME	debian packages
Chromium	38.0.2125.0	Debian 10 GNOME	Projects (2014)
Chromium	44.0.2403.0	Debian 10 GNOME	Projects (2015)
Chromium	53.0.2785.0	Debian 10 GNOME	Projects (2016)
Chromium	64.0.3282.0	Debian 10 GNOME	Projects (2017)
openssl_sclient	1.1.1d	Debian 10 GNOME	debian packages
gnutls-cli	3.6.7	Debian 10 GNOME	debian packages
wget	1.20.1	Debian 10 GNOME	debian packages
curl	7.64.0	Debian 10 GNOME	debian packages
NetSurf	3.6-3.2	Debian 10 GNOME-Classic	debian packages
Firefox	70.0.1	Microsoft Windows 10	Mozilla (2019)
Chromium	72.0.3582.0	Microsoft Windows 10	Projects (2019a)
Chromium	78.0.3904.108	Microsoft Windows 10	Projects (2019b)
Edge	44.17763.831.0	Microsoft Windows 10	MicrosoftEdge (2019)
Edge	79.0.309.40	Microsoft Windows 10	MicrosoftEdge (2019)

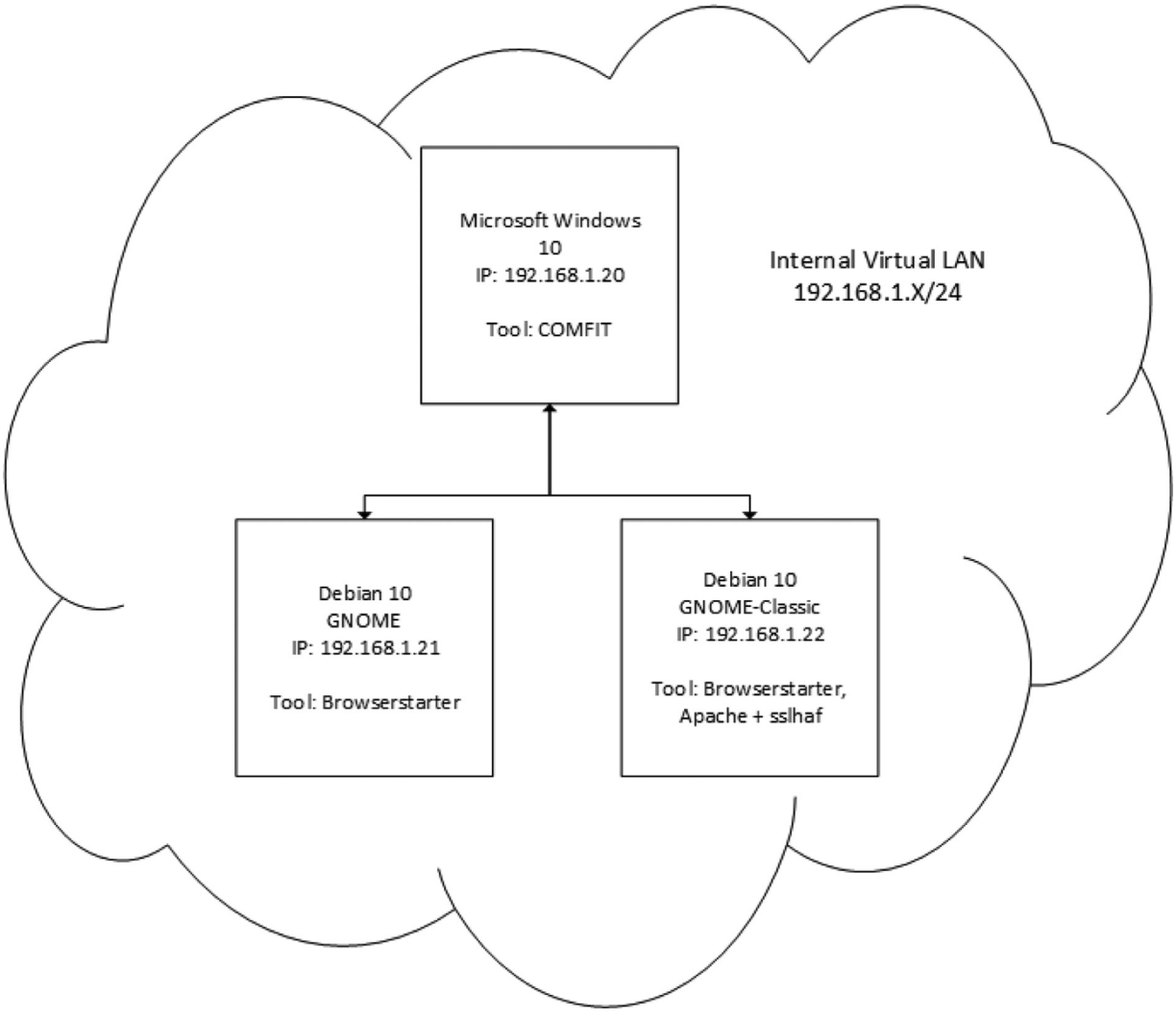

Fig. 6. Testing infrastructure setup.

Table 10
Used operating systems.

OS	Version
MS Windows 10 Pro	Build: 17763.864
Debian 10 (GNOME)	Kernel: 4.19.0.6-amd64
Debian 10 (GNOME-Classic)	Kernel: 4.19.0.6-amd64

Table 11
Software used for data processing and evaluation.

Name	Version
Java	Java 8u191
SQLite	3.11.2
Apache	2.4.38-3+deb10u3
sslhaf	7f8702 (SHA1 commit)
Perl	v5.28.1 (used for evaluation)
Python	2.7.16 (used for evaluation)
Pandas	0.24.2 (used for evaluation)
.net core	3.1.201 (used for evaluation)

Table 12
Anonymity sets resulting from ClientHello-based browser fingerprinting approach.

Anonymity Set	Browsers
1	Chromium 72 Windows, Chromium 78 Windows, Edge 79 Windows, Chromium 78 Debian
2	Firefox Windows, Firefox Debian
3	NetSurf, curl
4	Konqueror
5	epiphany
6	Chromium 64
7	qutebrowser
8	surf
9	Edge 44 Windows
10	lynx
11	Chromium 44
12	Chromium 38
13	Chromium 53
14	wget
15	openssl_sclient
16	gnutls-cli

the resulting splittings into anonymity sets. We observe the following from the results:

- *There is a clear behavioral dependency during potentially malformed TLS handshakes on the individual browser.*
- *For the some fixed strength, running combinatorial browser fingerprinting experiments with a SCA of the same strength, but with symbol set of greater cardinality, will lead to a better splitting.* This hypothesis is aligned with the intuition that malformed TLS handshakes with more events will execute more error logic in the browser and hence have a greater chance of revealing differences in their implementations.
- *Running combinatorial browser fingerprinting experiments for the same symbol set of events with higher strength will lead to better splitting results.* This hypothesis is aligned with the intuition that a higher strength leads to higher structural complexity in the SCA which increases the chances to reveal differences in the behavior of different browsers.
- The results obtained from our case study with combinatorial browser fingerprinting in this work are in accordance with what has been reported in Garn et al. (2019).

Consistent with the reasoning outlined above, the best possible splitting in all performed combinatorial browser fingerprinting experiments was achieved with the SCA of strength $t = 6$ for the complete symbol set $\mathcal{E}$. In this splitting, the 21 browsers were split into 14 anonymity sets given in Table 13. These are ten sigletons, two anonymity sets of cardinality two, one anonymity set of cardinality three and one anonymity set of cardinality four. This splitting has values 3.5944 and 0.8183 as entropy and normalized entropy, respectively.

Table 13
Best possible splitting achieved by the combinatorial browser fingerprinting approach for the SCA with symbol set $\mathcal{E}$ of strength six.

Anonymity Set	Browsers
1	lynx
2	Konqueror, Edge 44 Windows
3	openssl_sclient
4	Chromium 38 Debian, Chromium 44 Debian
5	Chromium 78 Debian
6	qutebrowser
7	Chromium 53 Debian
8	Chromium 64 Debian, Chromium 72 Windows, Chromium 78 Windows, Edge 79 Windows
9	surf, epiphany, wget
10	NetSurf
11	curl
12	Firefox Windows
13	Firefox Debian
14	gnutls-cli

Table 14
Anonymity sets resulting from proposed two-step approach.

Anonymity Set	Browsers
1	Chromium 78 Windows, Chromium 72 Windows, Edge 79 Windows
2	Firefox Windows
3	Firefox Debian
4	Chromium 78 Debian
5	NetSurf
6	curl
7	Konqueror
8	epiphany
9	Chromium 64
10	qutebrowser
11	Edge 44 Windows
12	surf
13	lynx
14	Chromium 44
15	Chromium 38
16	Chromium 53
17	wget
18	gnutls-cli
19	openssl_sclient

7.3. Results of two-Step approach

Our proposed two-step approach yielded 19 anonymity sets, which are listed in Table 14. This splitting consists of 18 singletons and one anonymity set of cardinality three and achieved the values 4.1658 for entropy and 0.9484 for normalized entropy.

Comparing the results of the two-step approach to the ClientHello- and combinatorial fingerprinting-based approaches on their own, we observe the following:

- The results of the two-step approach improve over the ClientHello-based approach alone, i.e., employing the combinatorial browser fingerprinting approach as a second step leads to better results, both in terms of the number of anonymity sets in

the respective partitions, which increases from 16 to 19, as well as in terms of entropy, which increases from 3.8208 to 4.1658.

- All three non-singleton anonymity sets produced in the first step (i.e., ClientHello-based approach) of our proposed two-step approach (see Table 12) were split up further in the second step via the combinatorial browser fingerprinting approach. We postpone a detailed evaluation of the combinatorial browser fingerprinting approach from a (purely) *combinatorial perspective* to Appendix A and present herein only the details of the improved splitting and how these improvements can actually be achieved (i.e., give SCAs with which these improvements can actually be achieved).
  - The first step of our proposed two-step approach put the two browser "Firefox Windows" and "Firefox Debian" together into the same anonymity set of cardinality two, but the second step of our proposed two-step approach was able to split them into two singleton anonymity sets. This further splitting is not achievable by a single TLS handshake consisting of only one single TLS handshake server-side message, but with certain single TLS handshakes consisting of two TLS handshake server-side messages, for example with the sequence $\langle 1, 2\rangle = \langle \text{Certificate}, \text{ECDHEServerKeyExchange}\rangle$, which is a SCA with two symbols of strength one.
  - Similarly, the anonymity set of cardinality two consisting of the two browsers "NetSurf" and "curl" produced in the first step was split into two singleton anonymity sets in the second step. This further splitting is not achievable by a single TLS handshake consisting of only one single TLS handshake server-side message, but with certain single TLS handshakes consisting of two TLS handshake server-side messages, for example with the sequence $\langle 1, 4\rangle = \langle \text{Certificate}, \text{ChangeCipherSpec}\rangle$, which is a SCA with two symbols of strength one.
  - Likewise, the anonymity set of cardinality four consisting of the browsers "Chromium 72 Windows", "Chromium 78 Windows", "Edge 79 Windows", "Chromium 78 Debian" produced in the first step is split up in the second step into two anonymity sets, namely one of cardinality three consisting of the browsers "Chromium 78 Windows", "Chromium 72 Windows" and "Edge 79 Windows" and a singleton anonymity set consisting only of the browser "Chromium 78 Debian". The lowest strength of a SCA that achieves this splitting is three, for example for the subset $\{0, 1, 3, 4, 5\} = \{\text{ServerHello}, \text{Certificate}, \text{ServerHelloDone}, \text{ChangeCipherSpec}, \text{Finished}\}$ of the set $\mathcal{E}$ of server-side TLS handshake messages.

8. Threats to validity

In this section, we comment on possible threats to validity of this work.

First, with regard to internal validity, we recognize the threat posed by using TLS attacker as the only tool for sending and extracting data from TLS messages. The same holds for the other used software in our data processing pipeline. However, all of the used third party software is considered to be state-of-the-art.

Second, with regard to external validity, it is evident that although some separation of the browsers and TLS clients was achieved in the case study, it cannot be concluded that the used properties for distinguishing browsers will lead to non-trivial results for any combination of operating system and browser. Nonetheless, we designed the case study to be diverse by including browsers and TLS clients built using a variety of different technologies (text and graphical user interface, rendering engine and underlying TLS library).

Table 15
Best results for SCAs for one event.

t	max num anon sets	max entropy	max norm. entropy
1	2	0.9587	0.2182

9. Conclusion

In this work, we have shown how combining a property-based fingerprinting approach with a behavior-based one results in stronger splitting capabilities than can be achieved by each underlying approach in isolation. The approach only uses properties of the TLS handshakes and the behavior induced by combinatorial sequences exhibiting a specific structure. The presented two-step approach decreases the size of the anonymity sets, increases the entropy of the splitting, and allows an almost unique identification of browsers and TLS clients.

Disclaimer: Any mention of commercial products in this paper is for information only; it does not imply recommendation or endorsement by NIST.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A

This appendix provides a detailed evaluation of the resulting splittings of the set of all 21 browsers given in Section 6.2 for the conducted experiments with the combinatorial browser fingerprinting approach alone from a *combinatorial perspective*. In particular, we focus on the impact of the cardinality of the symbol set, the subset of correspondingly selected TLS handshake server-side messages from the set $\mathcal{E}$ of events and the strength t of the instantiated SCAs on the resulting splittings into anonymity sets of the set of all browsers in terms of various quantitative measures.

In the following, we detail the obtained results grouped by the cardinality of the selected nonempty subset from the set of all events. For each such group, we consider results from SCAs for all event subsets of that cardinality of all applicable strengths. Within each group, for each applicable strength, we report the greatest number of anonymity sets for all considered splittings, the maximal entropy as well as maximal normalized entropy in a table. Note that these three values might be achieved by SCAs for different event selections, but which nevertheless have the same strength.

A1. Results for SCAs of singleton symbol sets

For each singleton subset of the set $\mathcal{E}$ of events (i.e., $n = 1$), we regard this singleton as a SCA of strength one encoding a single TLS handshake. The best achieved values of these SCAs are listed in Table 15.

A visualization of the distribution of the achieved relative entropy for $n = 1$ is depicted in Fig. 7.

A2. Results for SCAs for symbol sets of cardinality two

For all subsets C of the symbol set $\mathcal{E}$ of cardinality two (i.e., $n = 2$), for all $t \in \{1, 2\}$, we consider the SCAs of strength t for the set C constructed in Section 6.1. The best achieved values of these SCAs are given in Table 16 and also visualized in Fig. 8. A visualization of the distribution of the achieved relative entropy for $n = 2$ is depicted in Fig. 9.

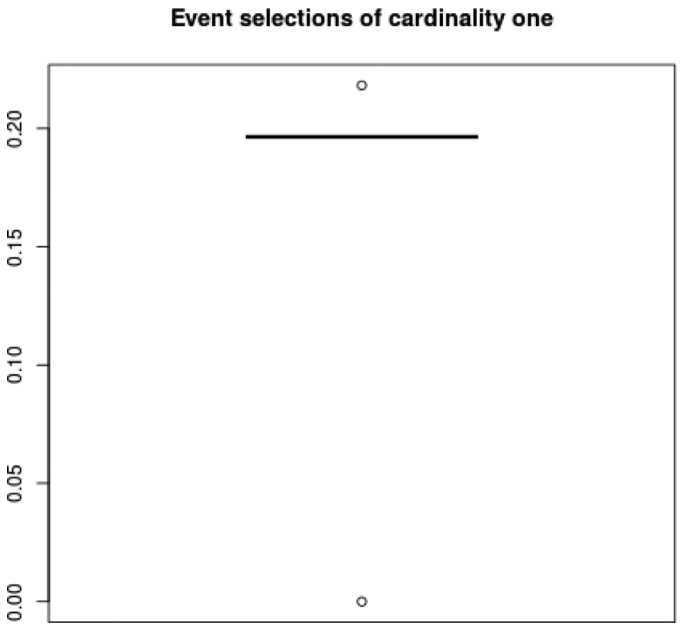

Fig. 7. Visualization of distribution of relative entropy for $n = 1$.

Table 16
Best results for SCAs for symbol set of two events for strength one and two.

t	max num anon sets	max entropy	max norm. entropy
1	7	2.2623	0.5150
2	9	2.7013	0.6150

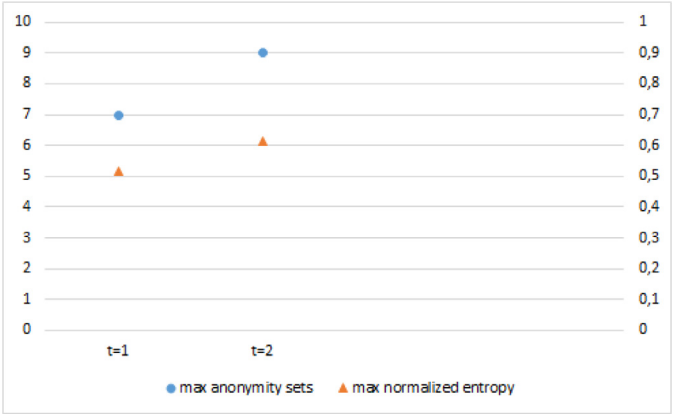

Fig. 8. Visualization best achieved values for $n = 2$.

Table 17
Best results for SCAs for symbol set of three events for strength one to three.

t	max num anon sets	max entropy	max norm. entropy
1	7	2.2437	0.5108
2	8	2.5468	0.5798
3	9	2.7013	0.6150

A3. Results for SCAs for symbol sets of cardinality three

For all subsets C of the symbol set $\mathcal{E}$ of cardinality three (i.e., $n = 3$), for all $t \in \{1, 2, 3\}$, we consider the SCAs of strength t for the set C constructed in Section 6.1. The best achieved values of these SCAs are given in Table 17 and also visualized in Fig. 10. A visualization of the distribution of the achieved relative entropy for $n = 3$ is depicted in Fig. 11.

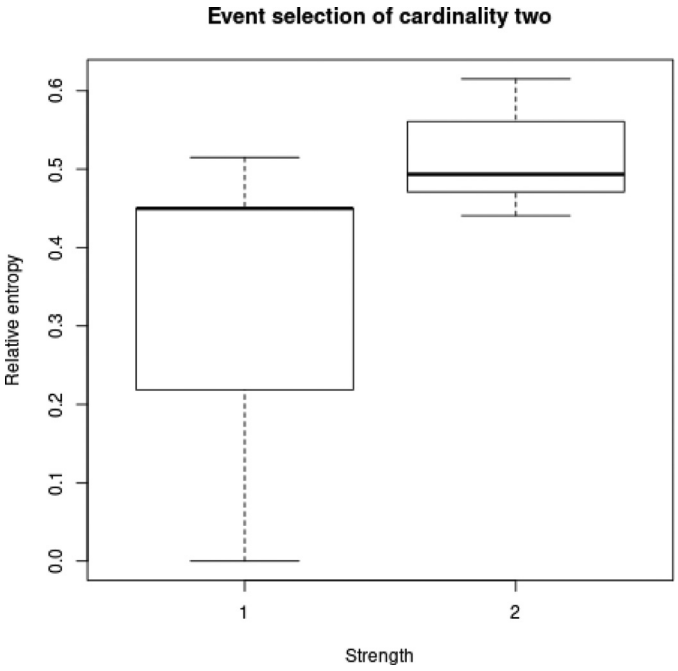

Fig. 9. Visualization of distribution of relative entropy for $n = 2$.

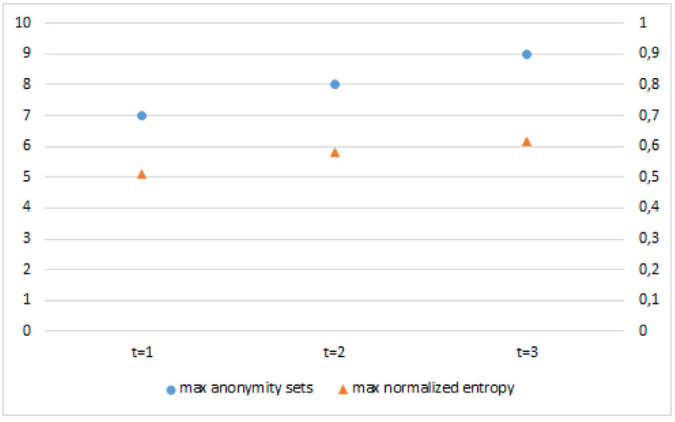

Fig. 10. Visualization best achieved values for $n = 3$.

Table 18
Best results for SCAs for symbol set of four events for strength one to four.

t	max num anon sets	max entropy	max norm. entropy
1	7	2.2623	0.5150
2	8	2.6061	0.5933
3	9	2.7013	0.6150
4	9	2.7013	0.6150

A4. Results for SCAs for symbol sets of cardinality four

For all subsets C of the symbol set $\mathcal{E}$ of cardinality four (i.e., $n = 4$), for all $t \in \{1, 2, 3, 4\}$, we consider the SCAs of strength t for the set C constructed in Section 6.1. The best achieved values of these SCAs are given in Table 18 and also visualized in Fig. 13. A visualization of the distribution of the achieved relative entropy for $n = 4$ is depicted in Fig. 14.

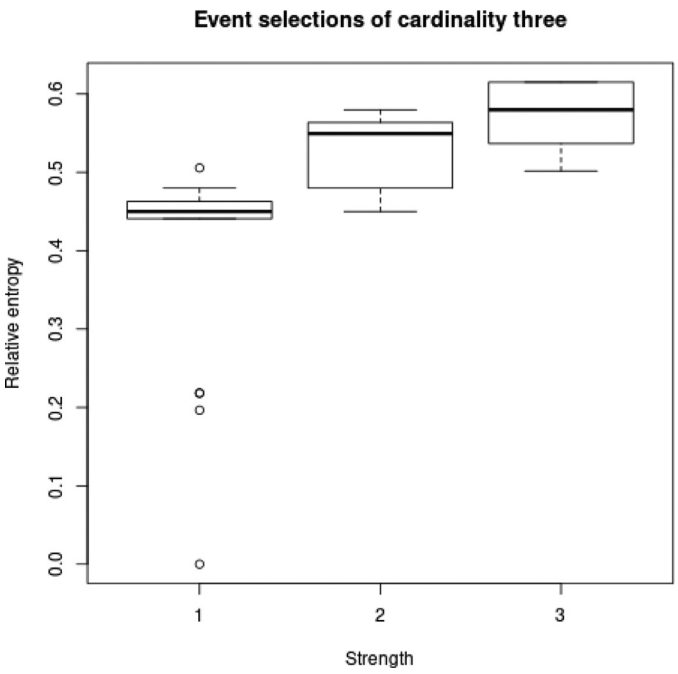

Fig. 11. Visualization of distribution of relative entropy for $n = 3$.

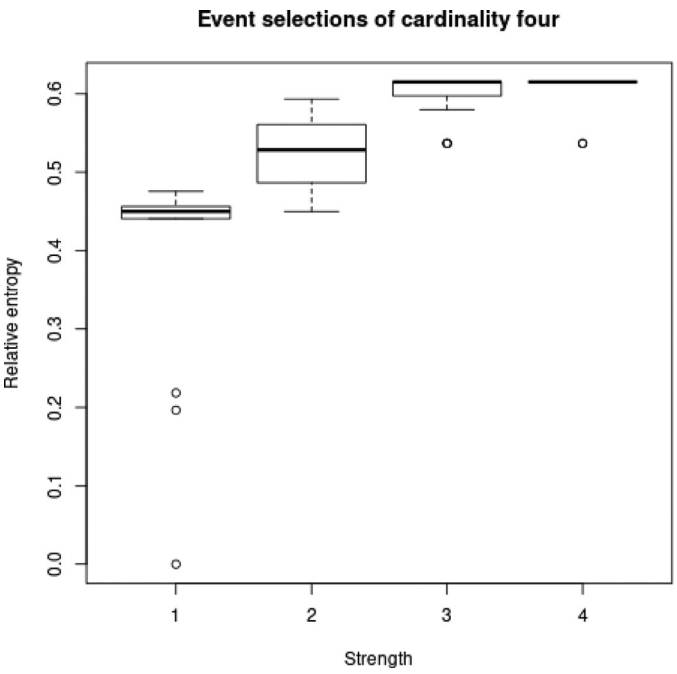

Fig. 13. Visualization of distribution of relative entropy for $n = 4$.

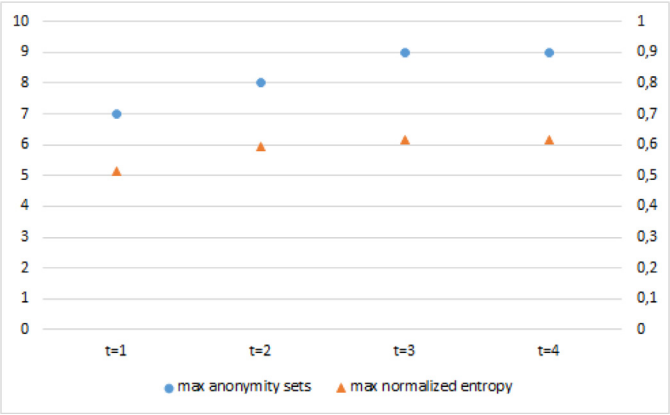

Fig. 12. Visualization best achieved values for $n = 4$.

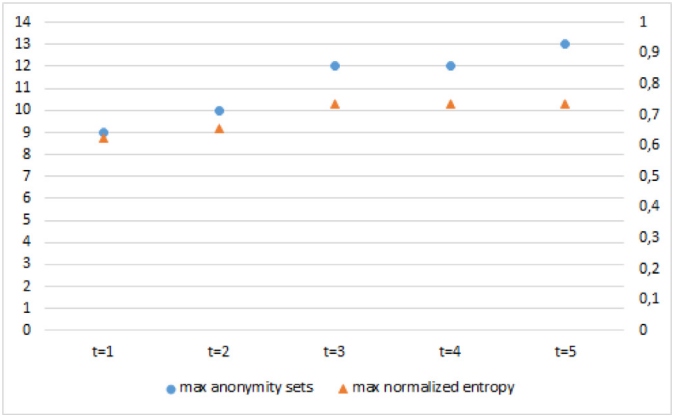

Fig. 14. Visualization best achieved values for $n = 5$.

Table 19
Best results for SCAs for symbol set of five events for strength one to five.

t	max num anon sets	max entropy	max norm. entropy
1	9	2.7539	0.6269
2	10	2.8851	0.6568
3	12	3.2368	0.7369
4	12	3.2368	0.7369
5	13	3.3320	0.7585

Table 20
Best results for SCAs for symbol set of six events for strength one to six.

t	max num anon sets	max entropy	max norm. entropy
1	9	2.7539	0.6269
2	8	2.521	0.5739
3	11	3.0396	0.6920
4	12	3.2368	0.7369
5	12	3.2368	0.7369
6	14	3.5944	0.8183

A5. Results for SCAs for symbol sets of cardinality five

For all subsets C of the symbol set $\mathcal{E}$ of cardinality five (i.e., $n = 5$), for all $t \in \{1, 2, 3, 4, 5\}$, we consider the SCAs of strength t for the set C constructed in Section 6.1. The best achieved values of these SCAs are given in Table 19 and also visualized in Fig. 15. A visualization of the distribution of the achieved relative entropy for $n = 5$ is depicted in Fig. 16.

A6. Results for SCAs for symbol sets of cardinality six

For all subsets C of the symbol set $\mathcal{E}$ of cardinality six (i.e., $n = 6$), for all $t \in \{1, 2, 3, 4, 5, 6\}$, we consider the SCAs of strength t for the set C constructed in Section 6.1. The best achieved values of these SCAs are given in Table 20 and also visualized in Fig. 16. A visualization of the distribution of the achieved relative entropy for $n = 6$ is depicted in Fig. 17.

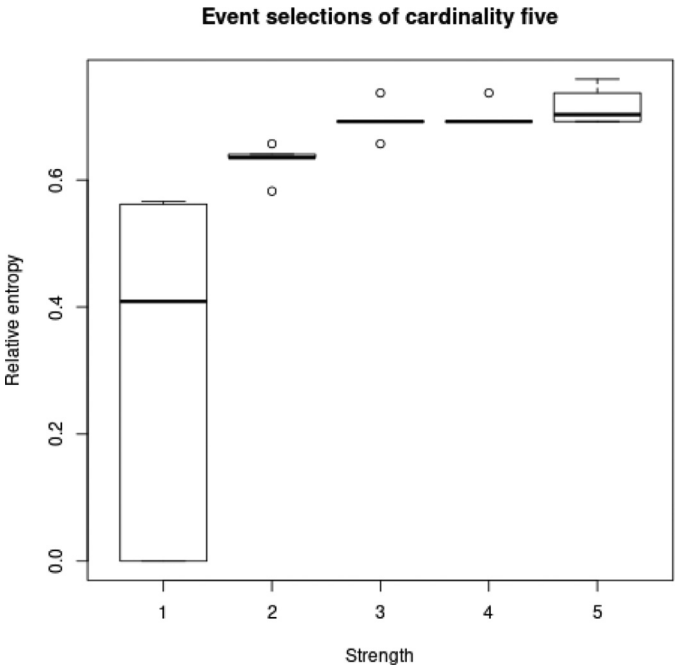

Fig. 15. Visualization of distribution of relative entropy $n = 5$.

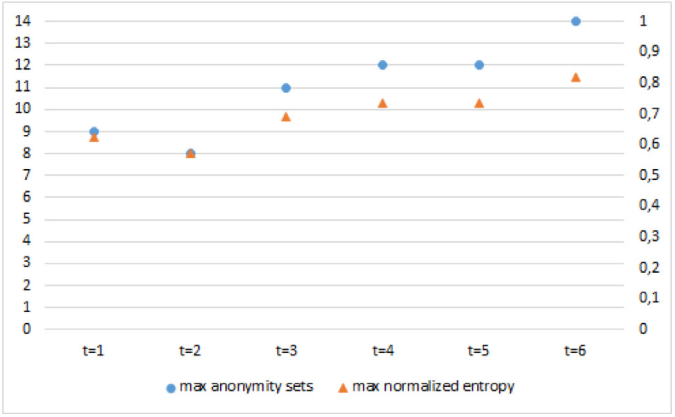

Fig. 16. Visualization best achieved values for $n = 6$.

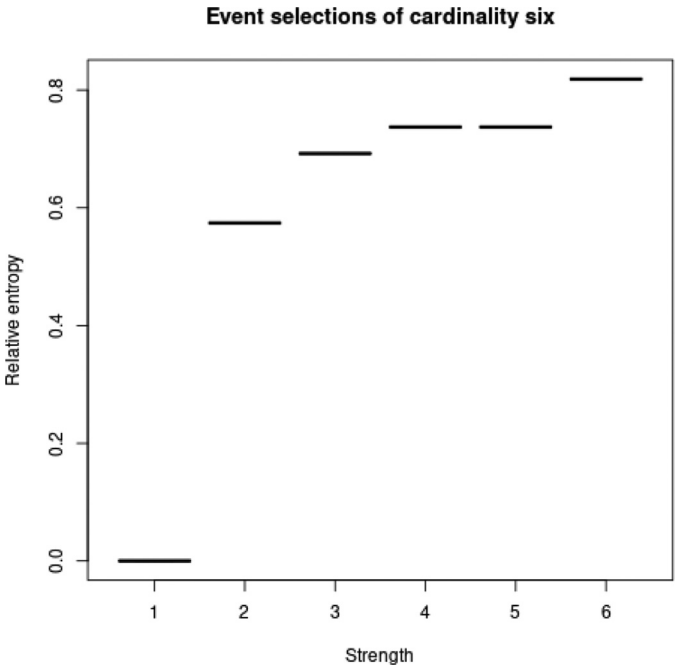

Fig. 17. Visualization of distribution of relative entropy $n = 6$.

Algorithm 1 Two-step approach for browser fingerprinting.

```
procedure 2StepFingerprinting(browser)
    AnonSet_1 ← ClientHelloFP(browser)
    if AnonSet_1.size == 1 then
        return Identified browser
    else
        AnonSet_2 ← CombinatorialFP(AnonSet_1)
        if AnonSet_2.size == 1 then
            return Identified browser
        else
            return browser in group of AnonSet_2.size
        end if
    end if
end procedure
```

A7. Complete results for all SCAs

In Table A.21, we provide the results for each one of the considered 192 fingerprinting experiments with combinatorial browser fingerprinting via SCAs. For each SCA, we give its event subset, the number of anonymity sets in its resulting splitting, the corresponding entropy and normalized entropy.

Table A.21
Results for all considered SCAs.

n	t	event selection	groups	entropy	norm. entropy	n	t	event selection	groups	entropy	norm. entropy
1	1	{0}	1	0.000000	0.000000	4	1	{0, 3, 4, 5}	5	2.048542	0.466392
1	1	{1}	2	0.863121	0.196506	4	1	{1, 3, 4, 5}	5	1.976648	0.450023
1	1	{2}	2	0.863121	0.196506	4	1	{0, 2, 3, 5}	5	1.976648	0.450023
1	1	{3}	2	0.863121	0.196506	4	1	{2, 3, 4, 5}	5	1.976648	0.450023
1	1	{4}	2	0.863121	0.196506	4	1	{0, 1, 3, 5}	5	2.048542	0.466392
1	1	{5}	2	0.958712	0.218270	4	1	{0, 1, 2, 4}	2	0.863121	0.196506
2	1	{1, 5}	6	2.167124	0.493389	4	1	{0, 2, 4, 5}	5	1.976648	0.450023
2	1	{3, 5}	7	2.262362	0.515072	4	1	{0, 2, 3, 4}	6	2.031184	0.462440
2	1	{2, 3}	5	1.976648	0.450023	4	1	{0, 1, 3, 4}	5	2.089245	0.475658
2	1	{3, 4}	5	1.976648	0.450023	4	1	{1, 2, 3, 4}	5	1.976648	0.450023
2	1	{0, 5}	2	0.958712	0.218270	4	1	{0, 1, 2, 5}	2	0.958712	0.218270
2	1	{4, 5}	5	1.935946	0.440757	4	1	{1, 2, 3, 5}	5	1.976648	0.450023
2	1	{0, 4}	2	0.863121	0.196506	4	1	{0, 1, 2, 3}	1	0.000000	0.000000
2	1	{1, 2}	5	1.976648	0.450023	4	1	{1, 2, 4, 5}	5	1.935946	0.440757
2	1	{0, 1}	1	0.000000	0.000000	4	1	{0, 1, 4, 5}	5	1.935946	0.440757
2	1	{2, 5}	6	2.131177	0.485205	4	2	{0, 3, 4, 5}	8	2.606138	0.593340
2	1	{1, 3}	5	1.976648	0.450023	4	2	{, 3, 4, 5}	7	2.203071	0.501573
2	1	{0, 2}	2	0.863121	0.196506	4	2	{0, 2, 3, 5}	7	2.451608	0.558158
2	1	{0, 3}	2	0.958712	0.218270	4	2	{2, 3, 4, 5}	5	1.976648	0.450023
2	1	{2, 4}	5	1.976648	0.450023	4	2	{0, 1, 3, 5}	7	2.451608	0.558158
2	1	{1, 4}	5	1.935946	0.440757	4	2	{0, 1, 2, 4}	6	2.107833	0.479890
2	2	{1, 5}	8	2.606138	0.593340	4	2	{0, 2, 4, 5}	8	2.546847	0.579841
2	2	{3, 5}	9	2.701376	0.615022	4	2	{0, 2, 3, 4}	7	2.262362	0.515072
2	2	{2, 3}	5	1.976648	0.450023	4	2	{0, 1, 3, 4}	8	2.357600	0.536755
2	2	{3, 4}	6	2.131177	0.485205	4	2	{1, 2, 3, 4}	6	2.031184	0.462440
2	2	{0, 5}	6	2.243774	0.510840	4	2	{0, 1, 2, 5}	7	2.474953	0.563473
2	2	{4, 5}	7	2.451608	0.558158	4	2	{1, 2, 3, 5}	6	2.107833	0.479890
2	2	{0, 4}	6	2.167124	0.493389	4	2	{0, 1, 2, 3}	8	2.546847	0.579841
2	2	{1, 2}	7	2.203071	0.501573	4	2	{1, 2, 4, 5}	6	2.167124	0.493389
2	2	{0, 1}	5	1.935946	0.440757	4	2	{0, 1, 4, 5}	6	2.320423	0.528291
2	2	{2, 5}	8	2.606138	0.593340	4	3	{0, 3, 4, 5}	9	2.701376	0.615022
2	2	{1, 3}	6	2.107833	0.479890	4	3	{1, 3, 4, 5}	9	2.701376	0.615022
2	2	{0, 2}	6	2.131177	0.485205	4	3	{0, 2, 3, 5}	9	2.701376	0.615022
2	2	{0, 3}	7	2.474953	0.563473	4	3	{2, 3, 4, 5}	9	2.701376	0.615022
2	2	{2, 4}	5	1.976648	0.450023	4	3	{0, 1, 3, 5}	9	2.701376	0.615022
2	2	{1, 4}	6	2.031184	0.462440	4	3	{0, 1, 2, 4}	8	2.357600	0.536755
3	1	{0, 3, 4}	5	2.089245	0.475658	4	3	{0, 2, 4, 5}	9	2.701376	0.615022
3	1	{1, 4, 5}	5	1.935946	0.440757	4	3	{0, 2, 3, 4}	8	2.357600	0.536755
3	1	{2, 3, 4}	5	1.976648	0.450023	4	3	{0, 1, 3, 4}	9	2.701376	0.615022
3	1	{0, 1, 4}	2	0.863121	0.196506	4	3	{1, 2, 3, 4}	8	2.357600	0.536755
3	1	{0, 3, 5}	6	2.220430	0.505525	4	3	{0, 1, 2, 5}	9	2.701376	0.615022
3	1	{2, 3, 5}	5	1.976648	0.450023	4	3	{1, 2, 3, 5}	9	2.701376	0.615022
3	1	{1, 2, 5}	6	2.107833	0.479890	4	3	{0, 1, 2, 3}	8	2.546847	0.579841
3	1	{0, 1, 2}	1	0.000000	0.000000	4	3	{1, 2, 4, 5}	9	2.701376	0.615022
3	1	{0, 2, 4}	5	1.976648	0.450023	4	3	{0, 1, 4, 5}	9	2.701376	0.615022
3	1	{1, 2, 4}	5	1.976648	0.450023	4	4	{0, 3, 4, 5}	9	2.701376	0.615022
3	1	{0, 2, 3}	6	2.107833	0.479890	4	4	{1, 3, 4, 5}	9	2.701376	0.615022
3	1	{1, 3, 5}	5	1.935946	0.440757	4	4	{0, 2, 3, 5}	9	2.701376	0.615022
3	1	{1, 2, 3}	5	1.976648	0.450023	4	4	{2, 3, 4, 5}	9	2.701376	0.615022
3	1	{0, 2, 5}	5	1.976648	0.450023	4	4	{0, 1, 3, 5}	9	2.701376	0.615022
3	1	{1, 3, 4}	5	1.976648	0.450023	4	4	{0, 1, 2, 4}	8	2.357600	0.536755
3	1	{0, 1, 5}	2	0.958712	0.218270	4	4	{0, 2, 4, 5}	9	2.701376	0.615022
3	1	{3, 4, 5}	5	1.976648	0.450023	4	4	{0, 2, 3, 4}	9	2.701376	0.615022
3	1	{2, 4, 5}	6	2.107833	0.479890	4	4	{0, 1, 3, 4}	9	2.701376	0.615022
3	1	{0, 1, 3}	2	0.958712	0.218270	4	4	{1, 2, 3, 4}	8	2.357600	0.536755
3	1	{0, 4, 5}	5	1.976648	0.450023	4	4	{0, 1, 2, 5}	9	2.701376	0.615022
3	2	{0, 3, 4}	7	2.451608	0.558158	4	4	{1, 2, 3, 5}	9	2.701376	0.615022
3	2	{1, 4, 5}	8	2.546847	0.579841	4	4	{0, 1, 2, 3}	9	2.701376	0.615022
3	2	{2, 3, 4}	6	2.107833	0.479890	4	4	{1, 2, 4, 5}	9	2.701376	0.615022
3	2	{0, 1, 4}	6	2.131177	0.485205	4	4	{0, 1, 4, 5}	9	2.701376	0.615022
3	2	{0, 3, 5}	7	2.374959	0.540707	5	1	{1, 2, 3, 4, 5}	8	2.485694	0.565918
3	2	{2, 3, 5}	7	2.451608	0.558158	5	1	{0, 1, 3, 4, 5}	7	2.468204	0.561936
3	2	{1, 2, 5}	8	2.546847	0.579841	5	1	{0, 2, 3, 4, 5}	7	2.290462	0.521470
3	2	{0, 1, 2}	6	2.107833	0.479890	5	1	{0, 1, 2, 3, 4}	1	0.000000	0.000000
3	2	{0, 2, 4}	6	2.107833	0.479890	5	1	{0, 1, 2, 4, 5}	4	1.30124	0.296253
3	2	{1, 2, 4}	6	2.107833	0.479890	5	1	{0, 1, 2, 3, 5}	1	0.000000	0.000000
3	2	{0, 2, 3}	7	2.262362	0.515072	5	2	{1, 2, 3, 4, 5}	10	2.885104	0.656852
3	2	{1, 3, 5}	7	2.451608	0.558158	5	2	{0, 1, 3, 4, 5}	9	2.789866	0.635169
3	2	{1, 2, 3}	5	1.976648	0.450023	5	2	{0, 2, 3, 4, 5}	9	2.789866	0.635169
3	2	{0, 2, 5}	8	2.546847	0.579841	5	2	{0, 1, 2, 3, 4}	9	2.557588	0.582286
3	2	{1, 3, 4}	6	2.107833	0.479890	5	2	{0, 1, 2, 4, 5}	9	2.813210	0.640484
3	2	{0, 1, 5}	6	2.243774	0.510840	5	2	{0, 1, 2, 3, 5}	9	2.789866	0.635169
3	2	{3, 4, 5}	7	2.474953	0.563473	5	3	{1, 2, 3, 4, 5}	10	2.885104	0.656852
3	2	{2, 4, 5}	8	2.546847	0.579841	5	3	{0, 1, 3, 4, 5}	12	3.236857	0.736936
3	2	{0, 1, 3}	7	2.474953	0.563473	5	3	{0, 2, 3, 4, 5}	11	3.039633	0.692034
3	2	{0, 4, 5}	7	2.451608	0.558158	5	3	{0, 1, 2, 3, 4}	11	3.039633	0.692034

(*continued on next page*)

Table A.21 (*continued*)

3	3	{0, 3, 4}	7	2.451608	0.558158	5	3	{0, 1, 2, 4, 5}	11	3.039633	0.692034
3	3	{1, 4, 5}	8	2.546847	0.579841	5	3	{0, 1, 2, 3, 5}	11	3.039633	0.692034
3	3	{2, 3, 4}	7	2.203071	0.501573	5	4	{1, 2, 3, 4, 5}	11	3.039633	0.692034
3	3	{0, 1, 4}	8	2.357600	0.536755	5	4	{0, 1, 3, 4, 5}	11	3.039633	0.692034
3	3	{0, 3, 5}	8	2.606138	0.593340	5	4	{0, 2, 3, 4, 5}	12	3.236857	0.736936
3	3	{2, 3, 5}	8	2.546847	0.579841	5	4	{0, 1, 2, 3, 4}	11	3.039633	0.692034
3	3	{1, 2, 5}	9	2.701376	0.615022	5	4	{0, 1, 2, 4, 5}	11	3.039633	0.692034
3	3	{0, 1, 2}	7	2.203071	0.501573	5	4	{0, 1, 2, 3, 5}	11	3.039633	0.692034
3	3	{0, 2, 4}	8	2.357600	0.536755	5	5	{1, 2, 3, 4, 5}	11	3.039633	0.692034
3	3	{1, 2, 4}	8	2.357600	0.536755	5	5	{0, 1, 3, 4, 5}	12	3.236857	0.736936
3	3	{0, 2, 3}	9	2.701376	0.615022	5	5	{0, 2, 3, 4, 5}	13	3.332095	0.758618
3	3	{1, 3, 5}	9	2.701376	0.615022	5	5	{0, 1, 2, 3, 4}	11	3.039633	0.692034
3	3	{1, 2, 3}	8	2.357600	0.536755	5	5	{0, 1, 2, 4, 5}	11	3.039633	0.692034
3	3	{0, 2, 5}	8	2.546847	0.579841	5	5	{0, 1, 2, 3, 5}	12	3.134871	0.713716
3	3	{1, 3, 4}	7	2.203071	0.501573	6	1	{0, 1, 2, 3, 4, 5}	1	0.000000	0.000000
3	3	{0, 1, 5}	9	2.701376	0.615022	6	2	{0, 1, 2, 3, 4, 5}	8	2.521641	0.574102
3	3	{3, 4, 5}	9	2.701376	0.615022	6	3	{0, 1, 2, 3, 4, 5}	11	3.039633	0.692034
3	3	{2, 4, 5}	8	2.546847	0.579841	6	4	{0, 1, 2, 3, 4, 5}	12	3.236857	0.736936
3	3	{0, 1, 3}	8	2.606138	0.593340	6	5	{0, 1, 2, 3, 4, 5}	12	3.236857	0.736936
3	3	{0, 4, 5}	9	2.701376	0.615022	6	6	{0, 1, 2, 3, 4, 5}	14	3.594466	0.818352

CRediT authorship contribution statement

Bernhard Garn: Conceptualization, Methodology, Formal analysis, Writing – original draft, Writing – review & editing. **Stefan Zauner:** Methodology, Software, Investigation, Writing – original draft, Writing – review & editing, Visualization. **Dimitris E. Simos:** Supervision, Conceptualization, Methodology, Validation, Project administration, Funding acquisition. **Manuel Leithner:** Validation, Visualization, Writing – review & editing, Resources. **Richard Kuhn:** Conceptualization, Writing – original draft. **Raghu Kacker:** Conceptualization, Writing – original draft.

References

Alaca, F., Van Oorschot, P.C., 2016. Device fingerprinting for augmenting web authentication: classification and analysis of methods. In: Proceedings of the 32nd annual conference on computer security applications, pp. 289–301.

Alidoost Nia, M., Ruiz-Martínez, A., 2018. Systematic literature review on the state of the art and future research work in anonymous communications systems. Computers & Electrical Engineering 69, 497–520. doi:10.1016/j.compeleceng.2017.11.027. https://www.sciencedirect.com/science/article/pii/S0045790617302586

Andriamilanto, N., Allard, T., Guelvouit, G.L., 2021. "Guess Who?" Large-Scale Data–Centric Study of the Adequacy of Browser Fingerprints for Web Authentication. In: Barolli, L., Poniszewska-Maranda, A., Park, H. (Eds.), Innovative Mobile and Internet Services in Ubiquitous Computing. Springer International Publishing, Cham, pp. 161–172.

Andriamilanto, N., Allard, T., Guelvouit, G. L., Garel, A., 2020. A Large-scale Empirical Analysis of Browser Fingerprints Properties for Web Authentication. 2006.09511.

Arshad, M.R., Hussain, M., Tahir, H., Qadir, S., Ahmed Memon, F.I., Javed, Y., 2021. Forensic analysis of tor browser on windows 10 and android 10 operating systems. IEEE Access 9, 141273–141294. doi:10.1109/ACCESS.2021.3119724.

Banbara, M., Tamura, N., Inoue, K., 2012. Generating Event-Sequence Test Cases by Answer Set Programming with the Incidence Matrix. In: Dovier, A., Costa, V.S. (Eds.), Technical Communications of the 28th International Conference on Logic Programming (ICLP'12). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 86–97.

Benjamin, D., 2020. Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility. RFC 8701. RFC Editor. https://tools.ietf.org/rfc/rfc8701.pdf

Bojinov, H., Michalevsky, Y., Nakibly, G., Boneh, D., 2014. Mobile device identification via sensor fingerprinting. arXiv preprint arXiv:1408.1416.

Bursztein, E., Malyshev, A., Pietraszek, T., Thomas, K., 2016. Picasso: Lightweight device class fingerprinting for web clients. In: Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 93–102.

Cao, Y., Chen, Z., Li, S., Wu, S., 2017. Deterministic Browser. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA, pp. 163–178. doi:10.1145/3133956.3133996.

Chee, Y., Colbourn, C., Horsley, D., Zhou, J., 2013. Sequence covering arrays. SIAM J. Discrete Math. 27 (4), 1844–1861.

Corporation, O., 2020a. Oracle VM VirtualBox. https://www.virtualbox.org/.

Corporation, O., 2020b. Oracle VM VirtualBox User Manual. https://www.virtualbox.org/manual/.

Corporation, O., 2020c. Package java.rmi. https://docs.oracle.com/javase/8/docs/api/java/rmi/package-summary.html.

Dierks, T., Rescorla, E.,. RFC 5246. https://tools.ietf.org/rfc/rfc5246.txt. Accessed: 2021-06-31.

Dierks, T., Rescorla, E., 2008. The transport layer security (tls) protocol version 1.2.

Eckersley, P., 2010. How unique is your web browser? In: Atallah, M.J., Hopper, N.J. (Eds.) Privacy Enhancing Technologies. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–18.

Felt, A.P., Barnes, R., King, A., Palmer, C., Bentzel, C., Tabriz, P., 2017. Measuring HTTPS adoption on the Web. In: 26th USENIX Security Symposium (USENIX Security 17), pp. 1323–1338.

Feng, J., Yin, B., Cai, K., Yu, Z., 2012. 3-way gui test cases generation based on event-wise partitioning. In: 2012 12th International Conference on Quality Software, pp. 89–97.

Garn, B., Simos, D., Zauner, S., Kuhn, R., Kacker, R., 2019. Browser fingerprinting using combinatorial sequence testing. 6th Symposium and Bootcamp on the Science of Security (HotSoS).

Gómez-Boix, A., Laperdrix, P., Baudry, B., 2018. Hiding in the crowd: An analysis of the effectiveness of browser fingerprinting at large scale. In: Proceedings of the 2018 World Wide Web Conference. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, pp. 309–318. doi:10.1145/3178876.3186097.

Horsman, G., Findlay, B., Edwick, J., Asquith, A., Swannell, K., Fisher, D., Grieves, A., Guthrie, J., Stobbs, D., McKain, P., 2019. A forensic examination of web browser privacy-modes. Forensic Science International: Reports 1, 100036. doi:10.1016/j.fsir.2019.100036. https://www.sciencedirect.com/science/article/pii/S2665910719300362

Husák, M., Čermák, M., Jirsík, T., Čeleda, P., 2016. Https traffic analysis and client identification using passive ssl/tls fingerprinting. EURASIP Journal on Information Security 2016 (1), 6.

Jadoon, A.K., Iqbal, W., Amjad, M.F., Afzal, H., Bangash, Y.A., 2019. Forensic analysis of tor browser: A Case study for privacy and anonymity on the web. Forensic Sci. Int. 299, 59–73. doi:10.1016/j.forsciint.2019.03.030. https://www.sciencedirect.com/science/article/pii/S0379073819301082

Karami, S., Ilia, P., Solomos, K., Polakis, J., 2020. Carnus: Exploring the Privacy Threats of Browser Extension Fingerprinting. In: In Proceedings of the 27th Network and Distributed System Security Symposium (NDSS).

Kuhn, D.R., Higdon, J.M., Lawrence, J.F., Kacker, R.N., Lei, Y., 2012. Combinatorial methods for event sequence testing. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE, pp. 601–609.

Laperdrix, P., Bielova, N., Baudry, B., Avoine, G., 2020. Browser fingerprinting: a survey. ACM Trans. Web 14 (2). doi:10.1145/3386040.

Laperdrix, P., Rudametkin, W., Baudry, B., 2016. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In: 2016 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 878–894.

Laperdrix, P., Starov, O., Chen, Q., Kapravelos, A., Nikiforakis, N., 2021. Fingerprinting in Style: Detecting Browser Extensions via Injected Style Sheets. In: 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, pp. 2507–2524. https://www.usenix.org/conference/usenixsecurity21/presentation/laperdrix

Maone, G., 2012. Noscript-javascript/java/flash blocker for a safer firefox experience.

Margalit, O., 2013. Better bounds for event sequencing testing. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, pp. 281–284.

Mayo, Q., Michaels, R., Bryce, R., 2014. Test suite reduction by combinatorial-based coverage of event sequences. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops, pp. 128–132.

MicrosoftEdge, 2019. Edge. https://www.microsoft.com/en-us/edge.

Mowery, K., Bogenreif, D., Yilek, S., Shacham, H., 2011. Fingerprinting information in javascript implementations. Proceedings of W2SP 2 (11).

Mozilla, 2019. Firefox. https://ftp.mozilla.org/pub/firefox/releases/70.0.1/win64/en-US/.

MulazzaniMartin and Reschl, Philipp and Huber, Markus and Leithner, Manuel and Schrittwieser, Sebastian and Weippl, Edgar and Wien, FC, 2013. Fast and reliable

browser identification with javascript engine fingerprinting. Web 2.0 Workshop on Security and Privacy (W2SP), Vol. 5. Citeseer
Olejnik, L., Englehardt, S., Narayanan, A., 2017. Battery status not included: Assessing privacy in web standards. In: IWPE@ SP, pp. 17–24.
Projects, T. C., 2014. chromium 38. https://commondatastorage.googleapis.com/chromium-browser-snapshots/index.html?prefix=Linux_x64/290041/.
Projects, T. C., 2015. chromium 44. https://commondatastorage.googleapis.com/chromium-browser-snapshots/index.html?prefix=Linux_x64/330230/.
Projects, T. C., 2016. chromium 53. https://commondatastorage.googleapis.com/chromium-browser-snapshots/index.html?prefix=Linux_x64/403380/.
Projects, T. C., 2017. chromium 64. https://commondatastorage.googleapis.com/chromium-browser-snapshots/index.html?prefix=Linux_x64/520842/.
Projects, T. C., 2019a. chromium. https://chromium.woolyss.com/.
Projects, T. C., 2019b. chromium. https://chromium.woolyss.com/.
Queiroz, J.S., Feitosa, E.L., 2019. A web browser fingerprinting method based on the web audio api. Comput J 62 (8), 1106–1120.
Rahman, M., Othman, R.R., Ahmad, R.B., Rahman, M.M., et al., 2014. A meta heuristic search based t-way event driven input sequence test case generator. Int. J. Simul. Syst. Sci. Technol 15 (3), 65–71.
Rescorla, E., Dierks, T., 2018. The transport layer security (tls) protocol version 1.3.
Rescorla, E., et al., 2000. Http over tls.
Ristic, I., 2009. sslhaf: Passive ssl client fingerprinting using handshake analysis.
Saito, T., Noda, T., Hosoya, R., Tanabe, K., Saito, Y., 2018. On estimating platforms of web user with javascript math object. In: International Conference on Network-Based Information Systems. Springer, pp. 407–418.
Saito, T., Takahashi, K., Yasuda, K., Tanabe, K., Taneoka, M., Hosoya, R., 2018. Tor Fingerprinting: Tor Browser Can Mitigate Browser Fingerprinting? In: Barolli, L., Enokido, T., Takizawa, M. (Eds.) Advances in Network-Based Information Systems. Springer International Publishing, Cham, pp. 504–517.
Solomos, K., Kristoff, J., Kanich, C., Polakis, J., 2021. Tales of favicons and caches: Persistent tracking in modern browsers. Network and Distributed System Security Symposium (NDSS 2021).
Somorovsky, J., 2016. Systematic fuzzing and testing of tls libraries. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1492–1504.
Takei, N., Saito, T., Takasu, K., Yamada, T., 2015. Web browser fingerprinting using only cascading style sheets. In: 2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA). IEEE, pp. 57–63.
pandas development team, T., 2020. pandas.dataframe. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html.
Walz, A., Sikora, A., 2018. Maximizing and leveraging behavioral discrepancies in tls implementations using response-guided differential fuzzing. In: 2018 International Carnahan Conference on Security Technology (ICCST). IEEE, pp. 1–5.
Yen, T.-F., Huang, X., Monrose, F., Reiter, M.K., 2009. Browser fingerprinting from coarse traffic summaries: Techniques and implications. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 157–175.

Bernhard Garn received a bachelor's and master's degree (DI) in applied mathematics from TU Wien. Currently, he is a PhD student of informatics at TU Wien. Bernhard's research focuses on design theory and discrete mathematics as well as applications thereof. With his background in mathematics, he is especially interested in the application of theoretical results to practical problems, effectively bridging the gap between discrete mathematics and application domains of information security. He has developed an algebraic modelling for a special class of combinatorial designs and has also developed combinatorial testing for software methodologies for security targeting operating systems and web technologies. His publications include papers in applied discrete mathematics, software testing (combinatorial testing) and security testing.

Stefan Zauner received a master's degree in IT Security from FH Campus Wien. Besides his research activities, he works as a software engineer. He is interested in privacy-related topics with an emphasis on browsers and browser fingerprinting using combinatorial methods.

Dr. Dimitris E. Simos is a Key Researcher with SBA Research, Austria, working on mathematical aspects of information security. He is also an Adjunct Lecturer with Vienna University of Technology and a Distinguished Guest Lecturer with Graz University of Technology. His research interests include combinatorial designs and their applications to software testing, combinatorial testing in particular, applied cryptography and optimization algorithms, and information security. He is a Fellow of the Institute of Combinatorics and its Applications (FTICA) since 2012 and has been in the organizing or program committee of many international scientific conferences and workshops (ESORICS, ARES, CAI, QRS, IWCT, MoCrySEn etc). Dimitris has been the Co-Chair of IWCT in the years 2017–2019. He also served as the General Chair for the 7th International conference on Mathematical Aspects of Computer and Information Sciences (MACIS 2017).

Manuel Leithner has previously conducted research on cloud storage security, mobile messenger applications, and combinatorial security testing. He is currently authoring a master thesis on input model inference via dynamic analysis. His research interests include combinatorial testing, dynamic analysis, reverse engineering, anonymity, and web vulnerabilities.

Rick Kuhn is a computer scientist in the Computer Security Division of the National Institute of Standards and Technology. He has authored two books and more than 100 conference or journal publications on information security, empirical studies of software failure, and software assurance, and is a Fellow of the Institute of Electrical and Electronics Engineers (IEEE). He co-developed the role- based access control model (RBAC) used throughout industry and led the effort that established RBAC as an ANSI standard. Previously he served as Program Manager for the Committee on Applications and Technology of the President's Information Infrastructure Task Force and as manager of the Software Quality Group at NIST.

Raghu Kacker is a mathematical statistician in the Applied and Computational Mathematics Division (ACMD) of the Information Technology Laboratory (ITL) of the US National Institute of Standards and Technology (NIST). His current interests include combinatorial testing of software and systems and evaluation of uncertainty in outputs of computational models and physical measurements. Advancing the methods and tools for combinational testing is a mission of his. He has authored or coauthored over 100 papers. He has a Ph.D. in statistics. He is a Fellow of the American Statistical Association and a Fellow of the American Society for Quality.