RADBOUD UNIVERSITY

Faculty of Computer Science

Master Specialisation Cyber Security

Measuring the Adoption of Device Class Fingerprinting

Master Thesis

Lorenzo Casini (s1062069)

Supervised by:

Dr. Gunes Acar, Radboud University

Dr. Fabian van de Broek, Open University Netherlands & Radboud University

June 2024

Contents

List of Tables

List of Figures

Abstract

When navigating the web, users' actions and preferences are constantly observed. First and third party services on web pages collects user data to provide tailored services or to increase the security of navigation. Different techniques can be used to reach these scopes, however, involved parties do not always respect regulatory principles or value users' privacy.

Device or browser fingerprinting is a well-known technique, that is also very hard to counter. It relies on browser supported APIs to collect different and apparently harmless information on a user's device. This information is then aggregated to create an almost unique identifier of that user's device, which allows re-identification across different sessions.

However, another technique known as device class fingerprinting allows to identify only the *class* of a device. This approach relies on less information, protecting users' privacy without sacrificing usability and reliability for many business scenarios. In this Thesis, we investigated how popular this technique is in the wild. We selected the *Picasso* [4] implementation proposed by *Google* and analysed its presence in the top 40K URLs of the Tranco list [17]. In addition to this, we tested if navigating as a mobile device or giving consent for data processing could trigger a more intense use of the technique, especially for abuse fighting.

To reach our goal, we first defined a heuristic to identify an implementation of device class fingerprinting in the wild. Among other distinguishing features, *Picasso* strongly relies on randomness at each execution to protect itself from various replay and dictionary attacks. This characteristics added complexity in the definition of our heuristic. To verify its presence in the wild, we run a data collection campaign on the defined URLs range multiple times to allow us to compare different executions over time. Lastly, we implemented a static analysis detection algorithm based on our heuristic and executed it against our data to retrieve results.

Although we observed only one implementation of *Picasso*, we identified other scripts relying on randomness, probably to obfuscate their behaviour by adding noise. Additionally, we identified scripts generating *Picasso*-like canvase images that did not rely on randomness. Overall, the results showed that this technique is not particularly popular. However, as *Picasso* relies on Canvas APIs, its affinity with canvas fingerprinting allowed us to observe the behaviour of other more popular canvas fingerprinting scripts.

Acknowledgements

Thanks to Aristea, my girlfriend. Thank you for being the support I needed and for taking care of me in my darkest moments. You are the greatest thing that happened in my life.

Thanks to my parents, Elisabetta and Marco. Thank you for being always understanding and patient. Thank you for being always there. I am who I am thanks to your teachings.

Thanks to my sister Ilaria. As different and as far away as we are, you always support me.

Thanks to my whole larger family, Flavia, Giuliano, Federica, Paolo, Irene and Niccolò. Thanks for always rooting for me unconditionally. Thanks to Iolanda, Siro, Annamaria, Enrico, Flora and Nilo, that are no more with us to see this accomplishment. Thank you for your love. I am happy you did not see me struggle. I wish you could see me succeed. I miss you all dearly.

Thanks to my friends Julen, Daniele, Cesare, Patrick, Alessandro M., Martina B., Maurizio, Martina P., Ilaria B., Ilaria T., Fjorela and Alessandro T. and all the others waiting for this result. Thank you for bringing me laugh and joy. Thank you for the time spent together. Thank you for both the kind words and the suggestions, without ever judging me.

Thanks to professor Acar, for guiding me in this way too long journey. Thank you for your patience, understanding and precious suggestions. Thanks to Antoine Vastel for sharing precious feedback, insights and experience about device class fingerprinting and *Picasso*.

Thanks to my employers Reply and Elca, for offering me a working opportunity and understanding the struggles and needs of working and studying.

Chapter 1

Introduction

1.1 Motivation

During web navigation, both first and third parties collect user data for various purposes. Tracking techniques are often used to provide personalised content, such as advertisements, but they are also used in security to detect bots or crawlers. Among the employed techniques, *Device Fingerprinting* relies on creating a unique identifier out of the differences in a device's software and hardware. By transitivity, the fingerprinted device exposes its user as well.

Differently from other fingerprinting approaches, *device class fingerprinting* tries to identify a device's class from fewer characteristics, such as the operating system and browser used. Results are then grouped and aggregated into different *classes*. This technique proved to be able to detect and distinguish even devices that try to spoof these characteristics or behave as something they are not. *Device class fingerprinting* techniques are then useful for different business scenarios by collecting minimum data and preserving a user's privacy.

This research aims at investigating how spread and popular *device class fingerprinting* is. For this purpose, we identified *Picasso* [4] as the *device class fingerprinting* technique of choice to investigate. As this approach needs some initial preparation and lots of storage, it is expected to be applied by large companies that can withstand the effort and continuous support. It is also supposed that these organisations can eventually provide *device class fingerprinting* as a Service, similar to what ReCaptcha[1] does. However, references to this technique in literature are scarce, which pose a challenge in defining solid detection rules.

1.2 Problem Statement

The starting point of this research was *Picasso: Lightweight device class fingerprinting for Web Clients* [4] by *Bursztein et al.*. The proposed solution is a variation of the technique known as canvas fingerprinting [22]. However, it is currently unknown how widely this technique is used. This research aims to measure how prevalent *device class fingerprinting* is across different websites and explore whether its usage is related to website popularity or their categories. By

[1] https://developers.google.com/recaptcha

investigating these aspects, we can gain a better understanding of how much this technique is used and its impact on web privacy and security.

1.3 Objectives and Contribution

Despite the time that has elapsed since *Bursztein et al.*'s publication, there has been little research specific to *device class fingerprinting*. This Thesis develops a heuristic to detect *device class fingerprinting* as an implementation of *Picasso*. We collected data through a crawler and implemented a custom analysis tool to detect websites and third parties that use *device class fingerprinting*. The main objective is to measure how popular the technique is today, if there are any differences in its presence between mobile and desktop form factors and if the detection is owned or provided by a third party.

Collected data might also show an updated and upgraded version of the technique. We will also look for possible differences in implementation with respect to *Picasso*. As robust as the proposed technique claims to be, different implementations might contain vulnerabilities. Based on observed data, we might find indicators of possible implementation flaws.

1.4 Thesis Outline

This Chapter introduced our research's motivation and its goals. In Chapter 2 we give an overview of the online tracking ecosystem, looking at its history, business models, techniques and legal basis. We then present canvas fingerprinting and how Canvas APIs are used as the backbone for *Picasso* to work. Chapter 3 describes the characteristics and the selection of *Picasso*'s distinguishing features. Then, it describes how data collection was approached and how the defined heuristic was implemented. In Chapter 4 we describe how we modified the crawler for a large-scale data collection campaign. We also describe how we selected the ideal analysis strategy and take advantage of the flexibility of the developed analysis tool. Chapter 5 presents the analysis results and how these relate to our research goals. We also highlight some interesting techniques and the probability of these to be an implementation of *Picasso*. Lastly, in Chapter 6 we discuss the achieved goals, the encountered limitations as well as potential future work.

Chapter 2

Background

This Chapter introduces the reader to the background concepts on which this research is based. First, it introduces how online tracking works, how it is used for benign and malicious purposes and the ethical concerns that come with it (2.1). Then, it presents the concept of canvas fingerprinting (2.2), which is the backbone of this research. Lastly, it introduces the concept of device class fingerprinting (2.3), a more privacy-friendly fingerprinting technique for abuse fighting.

2.1 Online Tracking

2.1.1 Overview

At the dawn of the internet era, the web seemed to allow anonymous communication just because "nobody knows who is behind the keyboard" [11], as long as one of the parties would not share such information willingly. Simultaneously, web content was intended to be administered by a single entity, be it a person or an organisation [20], generally called *first party*. With the evolution of the web, first parties found themselves in need of more flexibility and information to grow and increase revenue. As such needs emerged, the number of entities on each website increased dramatically.

Third party services allow the integration of valuable functionalities, such as website analytics, social networking and advertising (among others). However, the value they add to first parties and users comes at the price of potential security threats for the first and privacy concerns for the latter [20]. Nevertheless, to sustain their business model and continuously improve the offered services, both first and third parties monitor user activity on the website they manage or are hosted on.

Online tracking (or web tracking) is the technique in which first or third parties collect, store and connect users' browsing data both to offer an improved online service or for personal gains. The technique relies on identifying single users across different websites, online sessions and devices [9]. However, the web is not only populated by real users. It is estimated that bots and crawlers compose almost 50% of online traffic and are used for both benign and malicious intents [13]. In this landscape, third parties offer *bot detection* services, which rely on tracking techniques to detect suspicious behaviour [2]. Tracking techniques are extremely diverse and their usage depends on the first and third parties purposes. Depending on the final goal, these techniques rely on different mechanisms, which are either *stateful* or *stateless* [20].

Stateful tracking relies on the browser's ability to generate and store an identifier on a given device. HTTP cookies are an example of stateful tracking and still play a big role on the web and are probably the technique most known to the general public. They allow to save key-value pair information on the client device and its identification across multiple sessions over time. However, cookies are not only used for session handling but also for actively tracking users across domains. Due to an increased interest and attention to privacy concerns, most browsers today allow cookie blocking [24]. For instance, Safari and Firefox already block third party cookies by default[1] [14], and recently also Chrome announced to phaseout third party cookies [21]. This is slowly leading trackers to rely less on them and look for other approaches.

Stateless tracking, on the other hand, does not store any client-side information. It relies on obtaining information about a user's session or device by looking at HTTP Headers or through client-side JavaScript API calls in scripts embedded on the page. These information are generally harmless when considered alone, but altogether create an almost unique identifier of the device, called a *fingerprint* [14].

Modern browsers have become extremely complex, providing a set of functionalities originally more tied to the operating system side [22]. Fingerprinting relies on collecting measurable characteristics, such as screen resolution and installed fonts, which are then exploited and combined to create an identifier. This approach reduces the entropy[2] of a browser's identity and results in an (almost) unique identifier for the device, which is then shared in each communication with the party's server to allow recognition.

Fingerprinting is a stateless technique and can be either active or passive [20]. The first happens through scripts that actively request specific device characteristics (such as user agent or screen size). Conversely, passive fingerprinting is particularly difficult to detect and counteract, as it relies on information present in HTTP request packets or server logs.

Compared to cookies, fingerprinting is *opaque* to users, as it does not leave persistent evidence. Additionally, users do not have any control over their fingerprints and cannot "delete" them [14]. Only major changes in the browser's and client's configuration can result in a different fingerprint, making the previous one unlikable. However, client re-identification is generally possible and was proved to be relatively easy [7]. In Section 2.2 we will look at *canvas fingerprinting* in detail.

2.1.2 Business Models and Ethical Concerns

Online tracking comes with its advantages and concerns, due to the business models it is used for, as well as the data it uses. For instance, information is used to create users' profiles to then offer tailored *advertising*. This approach optimises advertising investments and allows companies to better reach potential customers [9][20]. *Analytics* services offer first parties the

[1]For detailed information on a specific browser, see https://www.cookiestatus.com/

[2]For a definition of entropy in the context of browser identification, refer to: https://www.eff.org/deeplinks/2010/01/primer-information-theory-and-privacy

possibility to better understand their visitors and their website usage. This allows for refinement of the user experience and implementation of additional services [20].

Online tracking is used in the *security* field as well. Financial institutions use online tracking techniques to prevent fraud and detect suspicious behaviours [14][22]. A similar approach is used also by online service providers during login procedures to further validate users. In both cases, it is checked whether requests come from a "known" and previously seen device [14]. User tracking can also happen for *testing* purposes or through other means, like integration with *social networks* or *content providers* [20].

Each mentioned scenario comes with its unique privacy concerns. *Advertising* collects and consumes data specifically aimed at identifying each single user and their interests. Collected data is extremely rich and specific [9] and although it might be pseudonymised, the quantity of information is often enough to uniquely identify a user. Data can be used to provide targeted advertising based on specific personal information, like their health situation, to leverage purchases of products based on their fears and insecurities.

Data collected for *security* reasons is again very user-specific. The extent and legitimacy of the data collected are bound only by the ethics of the party handling such information. For instance, a government tracking people without evidence or legitimate suspicion is easily addressed as "mass surveillance". In authoritarian regimes, these methods are used to control and repress dissent or to track down activists.

Similarly, *testing* data can be used against the user, through deceptive or manipulative design practices (i.e. dark patterns) to prevent the user from performing certain actions or push them into clicking something out of instinct, instead of using data to ease and improve the navigation experience [12][19].

In addition to this, one should consider that no data or company is immune to data breaches: there has been a notable increase in this kind of incidents in the past years [16][26]. Often the aim of these cyber-attacks is not only havoc or immediate economic profit, but also stealing consumer data from companies [16][26]. Once data is outside the control of the original owner (may it be sold or stolen), it can be read, copied, transferred, altered and misused.

2.1.3 Legal Basis

The right to privacy is recognised to some extent by almost every country in the world [6] and is also recognised as a fundamental right by the Universal Declaration of Human Rights [27]. However, data protection is not yet recognised as a human right [6]. Online tracking has been a technique widely controversial for its implications concerning data protection. The European Union's law on data protection, the GDPR, came into force in 2018 and brought a new definition of "consent": it had to be freely given, specific and unambiguous. This helped in reducing third party tracking although probably not for vocation but just to be compliant [18][23]. However, the conformity to GDPR remains somewhat lacking [23]. Legislators and tech companies have not yet found a consensus on how to guarantee and enforce the right to privacy of users online [23]. It would be much more suitable to have the user's consent being given as a browser configuration, like the do-not-track header, but tech companies are strongly against having such settings as legally binding [23].

2.2 Canvas Fingerprinting

In Subsection 2.1.1 we presented the concept of *browser fingerprinting*, a stateless tracking mechanism that generates a unique identifier for the device based on multiple unrelated information. This information are openly available and needed for various reasons, such as obtaining a device screen resolution to display a suitable layout. A unique identifier can then be obtained through specific techniques relying on different types of these information, such as canvas, font, AudioContext or battery API fingerprinting [8][24].

The possibility of using HTML Canvases as a tracking mechanism was first introduced by *Mowery and Shacham* [22] in 2012. Their intuition was that modern browsers strongly rely on the client's operating system functionalities and hardware resources through APIs and that their behaviour varies depending on the client's capabilities. The proposed canvas fingerprinting technique renders both text and WebGL scenes to a `<canvas>` element to then examine produced pixels. With this technique, they proved that rendering the same image across systems produces surprising variations in the rendered result, whereas the result remains unchanged on the same machine. *Fifield and Egelman* [10] came to the same conclusions in their work on Font fingerprinting.

The approach proposed by *Mowery and Shacham* [22] is quite simple and straightforward: once HTML "`2d`" image context is acquired, the Canvas API provides basic drawing primitives, such as `fillRect` and `arc`, as well as more complex functions, like drawing Bézier curves and defining color gradients. The "`2d`" image context can also draw text directly to the canvas, and it allows CSS-like text styling to change size and fonts (Figure 2.1). Lastly, to extract the result with pixel accuracy, the API provides two methods: `getImageData()` and `toDataURL()`. The first returns an `ImageData` object which contains the RGBA values for every pixel, the second returns a data URL consisting of the Base64 encoding of an image containing the content of the canvas. `toDataURL()` is especially interesting as one can perform the hash of the generated URL and use this value as the fingerprint. This makes it very easy to share and compare its value without the need to upload or send the full image.

To summarize, canvas fingerprinting is a method that draws an invisible image and extracts a persistent, long-term fingerprint without the user's knowledge [1]. As we will see throughout Chapters 3 and 4, it is extremely hard to precisely detect canvas fingerprinting due to its benign uses.

2.3 Device Class Fingerprinting and the *Picasso* Approach

Until now we have discussed tracking mechanisms that aim at identifying a single user or their browsers (2.1). However, the specificity of the collected information poses a risk to a user's privacy. For instance, a request for the user's location will return a very precise and specific

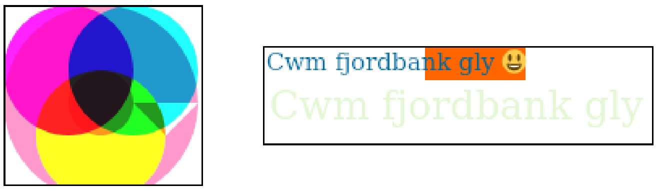

Figure 2.1: Example of canvas images generated by a script for canvas fingerprinting.

information. Targeted advertisements based on location might not need such a specific high level of accuracy and can rely on a broader location information, like city or region to be effective. This level of relaxation can be applied to Device Fingerprinting as well.

Device class fingerprinting (or *Device Type Fingerprinting*) tries to address this concern by relaxing the level of identification performed in online tracking. Information is aggregated to identify only the class of a client device. For instance, instead of tracking and recognising the exact device used by a user during a login procedure, it might be enough to identify the *class* of the device (an Android phone), to distinguish it from others (an iPhone, a PC or a cloud machine). Google's *Picasso* approach proposed by *Bursztein et al.* [4] relies on canvas fingerprinting and builds a protocol to perform device class fingerprinting.

2.3.1 Picasso

Picasso is a device class fingerprinting protocol that allows to verify if a specific client is running the hardware and software stack it claims and not spoofing or emulating any of these. The protocol can be used in many abuse fighting scenarios to verify the authenticity of clients. As canvas fingerprinting produces enough entropy to distinguish single devices, their protocol is based on the fact that specific Canvas API primitives can produce a *unique*, yet *stable* output across devices of the same class. In this way, the scheme minimises entropy between devices of the same class, while maximising it across distinct device classes. In fact, the scheme proved to be able to distinguish real hardware-software configurations from emulated ones. The scheme is intended to identify the device class of the client, which in *Picasso* is defined as the unique combination of *browser*, *operating system* and *graphics hardware*. The protocol is designed as a challenge-response scheme (Figure 2.2):

1. the server sends the challenge to the client;
2. the client performs the required computation and returns a response;
3. the response is validated by the server against the expected correct one.

The challenge (1) sent by the server contains indication on the canvas' size `A`, a random seed `s` and an integer number `N`, representative of the number of rounds to perform. The seed and the integer are used for randomisation of the image's content.The response (2) is the hash of the generated image. The generation algorithm executes a selected number of steps for `N` times. The seed `s` is used both to randomly select a primitive function to draw on the canvas, as well

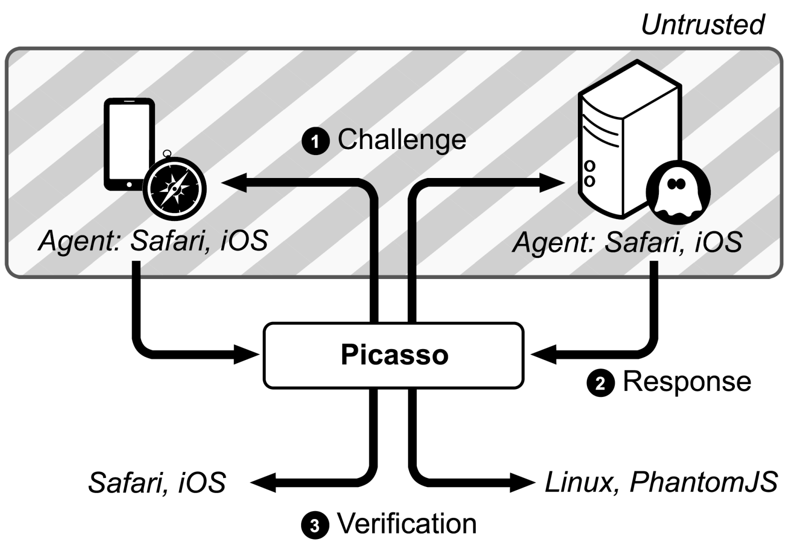

Figure 2.2: Image courtesy of *Picasso: Lightweight Device Class Fingerprinting for Web Clients* by *Burzstein et al.* [4]. High-level overview of device class fingerprinting: a challenge is sent to an untrusted client to prove its purported device class.

as randomise the arguments passed to the selected primitive. At the end of each round, the image is extracted and hashed with the previous hash value, if present. At the end, the final hash is sent as a response. The verification step (3) compares the received response against a pre-built knowledge base. As the client has already sent information about its nature through its `userAgent` or `navigator`, the server can compare the response with the expected response hash for that given device class.

For the scheme to work, a large initial knowledge base needs to be computed from a variety of trusted devices running different configurations. The protocol is also structured in a way that the server can increase its knowledge base, by sending multiple challenges, of which at least one response is known. Once all the challenges are received, if the known ones are correct, all the others can be safely added to the knowledge base. Otherwise, the challenge failed and all can be discarded. This possibility was already anticipated by *Mowery and Shacham* [22], as canvas fingerprinting supported a white box approach to prove that a machine is running a specific configuration.

The scheme is also designed to be robust against different kind of attacks. For instance, the seed and the integer used for randomisation help in protecting the scheme from replay attacks. Similarly, the knowledge base increase is inspired by how reCAPTCHA works. This is necessary to protect the scheme against dictionary attacks, i.e. from an attacker that was able to create a dictionary of known challenge-response pairs. Lastly, other mechanisms are in place to protect the scheme from pollution attacks from clients trying to inject bogus responses. Due to the costly preparation requirements, it is automatically assumed that smaller services do not have the computing capabilities to deploy *Picasso*. The authors envision again a system similar to reCAPTCHA, which is offered “as a Service” to third parties.

2.3.2 Prevalence of Device Class Fingerprinting

Since *Picasso*'s publication, there has not been much research on device class fingerprinting. The only study that openly mentions it was presented in 2020 by *Bird et al.* [3]. They tested a semi-supervised machine learning approach to identify regular browser fingerprinting but managed to detect device class fingerprinting as well. They managed to identify a Facebook's alternative implementation of *Picasso* with different graphical primitives, plus a never before seen device class fingerprinting approach through deep feature inspection.

This research is then built on the question "How popular is device class fingerprinting?" in its *Picasso* implementation and derivatives. To verify this, there is the need to (1) define what are the distinguishing features of device class fingerprinting (Chapter 3), (2) collect browsing data on a large set of web pages (Chapter 4) and (3) analyse such data based on the defined heuristic (Chapter 5).

# Chapter 3

Methodology

In the previous Chapter, we introduced how online device tracking works and more specifically how Canvas API could be used for device class fingerprinting. Next, we describe how we measured the prevalence of device class fingerprinting in its *Picasso* implementation [4] by answering some other intermediate research questions, such as:

- How did other researchers implement canvas fingerprinting detection?
- What are the distinguishing features of canvas device class fingerprinting?
- What kind of function calls are used? How can we capture their calls?
- Are there notable differences when accessing a page from a desktop or from a mobile device?
- Will giving consent for processing of personal data when accessing a page trigger additional behaviour?

In this Chapter, we present our research methodology, which was divided into three steps. First, we define possible distinguishing features of *Picasso*'s device class fingerprinting scheme and how they were identified (3.1). As second step, we look into possibilities for a large scale data collection of scripts on web pages and select OpenWPM as our data collection tool of choice (3.2). Lastly, we define how the heuristic defined in step one is implemented in a custom analysis tool (3.3).

3.1 Distinguishing Features of *Picasso*

As a first step, we tried to formalise and define *Picasso*'s distinguishing features. These features, also referred to as *indicators* throughout the paper, were inferred from both the original *Picasso* paper and other sources that investigated canvas fingerprinting.

3.1.1 Reviewing *Picasso*'s Scheme

To first identify the distinguishing features of *Picasso*'s scheme, we review its protocol and its steps, which were presented in Subsection 2.3.1. As described, the challenge contains an indication of the canvas' size `A`, a random seed `s` and an integer number `N`, representative of the number of rounds to perform.

One of the strengths of the solution proposed with *Picasso* lays in its variability, making the algorithm unpredictable. The `s` (seed) argument affects the random decision-making process of the algorithm, which influences how the graphical primitives will be selected, be it shapes or text. Assuming also that the seed will be different when the same script is executed again, the chosen graphical primitives across different executions will be different as well. Additionally, across different executions, the number of rounds `N` might change. This feature is formalised as "*variability of primitives*" (3.1.3). The seed `s` is also used in the randomisation of the arguments passed to the selected primitive. These arguments will define values like colors, size and position of the primitive, but also variation in its shape and orientation. This feature is formalised as "*variability of arguments*" (3.1.3). During the algorithm execution, the canvas image is extracted to compute the actual fingerprint. We expect to find calls to `getImageData()` or `toDataURL()` methods and therefore we formalise this feature as "*image extraction*" (3.1.3).

Until now, we did not list which Canvas API methods represent the graphical primitives for the implementation of *Picasso*. *Bursztein et al.* [4] do not openly state what methods were used in the paper, although some indication can be inferred from their charts in Figure 4, where pixel differences are plotted. The chart shows results for `circle`, `font`, `bezier` and `quadratic` shapes, which all have their respective method in the Canvas API. As these methods will be used to implement the indicator for either "*variability of primitives*", "*variability of arguments*" or both, we keep track of them in Table 3.1.

3.1.2 Relevant Literature Review

Burzstein et al. [4] do not provide clear distinguishing features. For instance, the algorithm is just presented with pseudo-code, with no additional details for implementation. Nevertheless, some features we expect to find for any *Picasso* implementation can be searched in related relevant literature.

Looking at other references for possible graphical primitives, *Bird et al.* [3] discovered the usage of emoji as primitive in the Facebook implementation of *Picasso*, which can be achieved by writing to canvas the emoji's unicode representation through `fillText()` or `strokeText()`. Outside of the literature, the best indication for valid primitives can be found in the demo implementation of *Picasso* by Vastel [28]. The provided source code[1] reveals the usage of gradients and randomly selected colors, as well as the primitives he used and their correlated methods. These findings are shown in Table 3.1. Figure 3.1 shows an example of a *Picasso* generated canvas image.

[1] Github: https://github.com/antoinevastel/picasso-like-canvas-fingerprinting

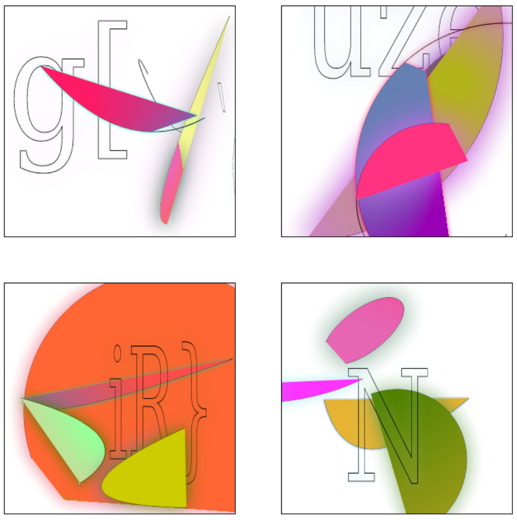

Figure 3.1: Examples of canvas images generated by the *Picasso* implementation proposed by Antoine Vastel [28].

Acar et al. [1] defined the criteria to detect canvas fingerprinting and to reduce the number of false positives. First, they expect canvas fingerprinting scripts to always have calls to Canvas API methods that write text to the canvas, namely `fillText()` and `strokeText()`, as well as calls to extract the final image, specifically `toDataUrl()`. *Englehardt and Narayanan* [8] expanded the reasoning behind how the image should be extracted by also including `getImageData()`. These findings are already covered by the list of expected graphical primitives in Table 3.1 and by the "*image extraction*" feature (3.1.3).

Acar et al. also defined that the extracted image should contain more than one color and its size should be greater than 16×16 pixels. This rule was adjusted by *Englehardt and Narayanan*, as they extended the size boundary by including the cited values. They claimed that the size should be at least 16×16 pixels. Finally, the extracted image should not be in a lossy compression format, such as JPEG. These three concepts are respectively formalised as "color constraint" (3.1.3), "size constraint" (3.1.3) and "no lossy compression" (3.1.3).

Papadogiannakis et al. [24] claim that canvas fingerprinting using text usually contains a pangram[2] in order to increase the number of entropy bits. To identify this technique, they required the text to be longer than five characters, otherwise "they do not contain enough bits of entropy to uniquely identify a user". Similarly, also *Englehardt and Narayanan* [8] expected the text to display "at least 10 distinct characters". This indicator is formalised as "*text length constraint*" in Subsection 3.1.3. Still, we would like to point out that *Englehardt and Narayanan* [8] introduced the concept and requirement of "*distinct*" characters, which is different from the already defined concept of "*variability of arguments*". We decided not to formalise this concept, assuming that if a tracker's goal is to get high entropy features, then it will not use same characters strings. Here we assumed all text will be composed of mostly different characters.

[2]A set of characters that contain all the letters of the English alphabet

3.1.3 Identified Distinguishing Features

We will now enumerate all the identified features and related considerations. For relative details about their implementation for static analysis, please refer to Appendix C.

Canvas API

As *Picasso* is based on Canvas API [22], we can limit the scope of research to scripts that make use of it.

Variability of Primitives

As *Picasso* strongly relies on randomness, we will look at anything that is not constant across different executions. Then, given a set of possible graphical primitives, this indicator searches for variations in their call order for a given script across multiple executions. If the call order and used methods are constant when comparing two different executions, then the script is excluded from the analysis. For a list of the chosen graphical primitives check Table 3.1.

Variability of Arguments

Similarly to "*variability of primitives*", we will look for variations across different executions. Then, given a set of possible graphical primitives, this indicator searches for variations in passed arguments for the same primitive for a given script across multiple executions. If the arguments passed to a given method do not change when comparing two different executions, then the script is excluded from the analysis. For a list of the chosen graphical primitives check Table 3.1.

API method/attribute	Primitive †	Var. of Primitives	Var. of Args	Text Length	Source
arc()	arc	✓	✓		[4] [28]
font()	text	✓	✓		[1] [3] [4] [28]
strokeText()	text	✓	✓	✓	[1] [3] [28]
fillText()	text	✓	✓	✓	[3]
bezierCurveTo()	bezierCurve	✓	✓		[4] [28]
quadraticCurveTo()	quadraticCurve	✓	✓		[4] [28]
moveTo()	bezierCurve, quadraticCurve		✓		[4] [28]
ellipse()	ellipse	✓	✓		[28]
createRadialGradient()	all		✓		[28]
shadowBlur	all		✓		[28]
shadowColor	all		✓		[28]

Table 3.1: Considered API methods and attributes.
"✓" marks when each one is relevant for a specific feature.
†: "Primitive" is used as a grouping name to point out in which Primitive-call it can be used.

Image Extraction

To compute the hash of the generated canvas image, we need to look for methods that return this information. This indicator looks for scripts that call either `getImageData()` or `toDataURL()` methods. If neither of these methods is present, then the script is excluded from the analysis.

Color Constraint

The generated canvas image needs to have enough entropy to be used effectively. When a canvas image is extracted, this indicator checks for its colors. If the image consists of a single color only, then the script is excluded from the analysis.

Size Constraint

The generated canvas image needs to have enough entory to be used effectively. When a canvas image is extracted, this indicator checks for its size. The size should be at least 16×16 pixels. Otherwise, the script is excluded from the analysis.

No Lossy Compression

The generated canvas image needs to have enough entropy to be used effectively. When a canvas image is extracted, this indicator checks for its compression type. If the image is extracted in a lossy compression format, then the script is excluded from the analysis.

Text Length Constraint

Generally, text is a strong indicator for canvas fingerprinting. However, *Picasso* only requires few characters to be useful, setting it apart from other techniques. If any text is written to a canvas, this indicator checks for its length. *Papadogiannakis et al.* [24] and *Englehardt and Narayanan* [8], that mention length checks, do not agree on the value. Of the two proposed values (5 vs. 10 characters), we can use the lowest value as a delimiter for a stronger indication of device class fingerprinting and the highest as an upper bound, above which results are excluded from the analysis, as it is inserting a lot of entropy for identification. The in-between values can be used as a softer, possible indicator, still worth keeping track of. The methods that allow writing text to canvas are tracked in Table 3.1.

3.2 Data collection

As second step of our research methodology, we need to collect execution data of scripts embedded in web pages. Due to the variability of the scheme, it is impossible to detect *Picasso*-based device class fingerprinting run-time while navigating a website. As mentioned in Subsection 3.1.3, we can detect device class fingerprinting by looking for differences when comparing two separate executions of the same script.

Therefore, for this study, we aimed at crawling Tranco[3] [17] top 100K URLs. Unfortunately, due to technical issues and limitations (4.2), we had to scale down to the top 40K URLs of said list. Additionally, to check for randomness, we aimed at crawling each website 5 times.

3.2.1 OpenWPM

In order to navigate a website, to capture calls to specific APIs (e.g. Canvas API) and to collect other relevant data, there was the need to develop a crawler as done by *Acar et al.* [1] or *Papadogiannakis et al.* [24] in their studies. Trying to address similar researchers' needs, *Englehardt and Narayanan* [8] implemented OpenWPM[4], an open source Selenium-based[5] web crawler to automatically navigate websites, intercept JavaScript function calls and much more.

We decided to use OpenWPM as our crawler, as it already offered enough flexibility and customisation options that fit our research goals. Additionally, it is a well known automated crawler designed for research purposes already used by other researchers, such as *Bird et al.* [3] and *Nayanamana and Mohammad* [25]. We relied on OpenWPM v0.21.1 of October 13th, 2022, which was the latest stable release prior to our data collection campaign (4.2).

OpenWPM offers a plethora of extensive configuration possibilities, which allow capturing HTTP requests and responses, DNS requests, cookies, any JavaScript API calls (defaulting the configurable file to capturing known APIs used in fingerprinting) and more. It also allows to easily extend and add functionalities on top of its existing infrastructure. Lastly, being built on top of a Selenium web driver, it allows to navigate web pages on consumer browsers such as Firefox or Chrome.

OpenWPM also offers the possibility to run multiple browsers in parallel to reduce crawling times and the instrumentation possibilities allowed us to easily capture Canvas API calls as desired. Finally, all data collected by OpenWPM is persisted in SQLite[6], making it also easy to perform analysis on it.

As mentioned at the beginning of this Chapter, among our research goals there are the questions if crawling as desktop or mobile or giving consent for personal data processing brings notable differences in scripts behaviour. This question was inspired by *Papadogiannakis et al.* [24], who identified stronger use of fingerprinting upon data processing rejection. Unfortunately, OpenWPM does not offer the possibility to crawl emulating a mobile device, nor it allows to give (or reject) consent when prompted. Lastly, although it can capture JavaScript function calls and their arguments, it does not capture values returned by said functions. Implementation details on these gaps are further discussed in Subsection 4.1.3.

[3] List generated on April 1st 2023. Available at `https://tranco-list.eu/list/4K83X`. For more information see Appendix A

[4] `https://github.com/openwpm/OpenWPM`

[5] `https://www.selenium.dev/documentation/`

[6] `https://www.sqlite.org/index.html`

3.2.2 Cloud Crawling

OpenWPM allows crawling multiple URLs in parallel by spawning a configurable amount of browsers. However, from our tests, the more browsers were spawned, the more the number of crawl failures increased. We noticed that in order to have a stable reliability, the number of browsers should have been at most one less than the number of available CPU cores. Nevertheless, we still had to expand our crawling capabilities and run our crawling campaign on parallel cloud machines. We selected Digital Ocean[7] as cloud provider, as it offered an easy way to set up Linux instances at affordable prices[8]. The best machine Digital Ocean could provide at the time, was a 4 CPU, 8 GB RAM, 160 GB NVMe disk and we could have at most 10 parallel instances at once.

3.3 Applying the heuristic

After defining the distinguishing features of *Picasso* implementations in Section 3.1, we designed an additional tool to apply this heuristic and analyse the data produced by OpenWPM's crawls. Due to the variable nature of *Picasso*, we needed to compare data from at least two different crawls. We then defined different aggregation possibilities: the first compared the full URL of scripts, the second aggregated scripts by their content hash as provided by OpenWPM ("aggregation by hash"), and the third aggregated them by comparing a portion of their URLs ('fuzzy aggregation"). The tool was designed to perform the analysis as a series of subsequent steps:

- *preparation* step, where relevant data is selected and grouped according to parameters. Data in raw format is extracted and grouped first by "top-level domain" and then by its relative full "JavaScript URL" (JS-URL). During this grouping, "aggregation by hash" could apply and JS-URL could be substituted by its script content hash. Right after, an additional attempt for grouping JS-URLs is performed and "fuzzy aggregation" could apply. Due to different *aggregation strategy* possibilities (C.2), we refer to JS-URL or its eventual aggregated substitute as `js_key`. Lastly, the grouped data of each `js_key` is evaluated for *Picasso*'s features;

- *analysis* step, where each feature indicator is computed. It elaborates the data processed in the previous step and performs the computational analysis for each *Picasso* feature. For each `js_key`, we return the probability of that script of being an implementation of *Picasso* (C.1);

- *output* step, where identified scripts are returned. This step returns information on evaluated crawls, such as failure percentage, as well as the list of `js_key` scripts that have a chance of being *Picasso* above a user-defined threshold. The canvas images generated by these scripts are also extracted as a visual proof;

[7]https://www.digitalocean.com/

[8]When comparing same capabilities machines offered by other providers, such as AWS or Microsoft, also considering the respective program for student or free credits

- *generate canvas* step, which extracts all crawls' canvas images. This step is for verification purposes only: it generates all other encountered canvas images, whose script did not reach the desired threshold. This step is meant for additional proof of work and is disabled by default.

Chapter 4

Data Collection

In the previous Chapter, we enumerated the expected distinguishing features of *Picasso* implementations. Then, we defined which tools were needed in our research, selecting OpenWPM as web crawler to collect browsing data, Digital Ocean as cloud provider to set up a crawling campaign at scale, and lastly, we designed a custom tool for the data analysis.

Nevertheless, although the selected tools covered the majority of our needs, there were still some gaps to fill. In this Chapter, we addressed these concerns by extending the capabilities offered by existing tools (4.1). Then, Section 4.2 presents how we conducted the data collection campaign.

4.1 Data Collection Preparation

To crawl the anticipated 100K URLs at scale, there was the need to address some gaps in the selected tools. In this Section, we review the decision-making process in configuring and extending OpenWPM, as well as verifying requirements for crawling in a cloud instance.

4.1.1 Device Class Fingerprint Example

To test the functionalities offered by OpenWPM and to verify the validity of the distinguishing features presented in Section 3.1, we required a sound example of *Picasso*'s device class fingerprinting. Similarly to what *Eckersley* [7], *Mowery and Shacham* [22] and *Fifield and Egelman* [10] did to test their approaches, we developed a static, local HTML page, which embedded Vastel's *Picasso* implementation [28]. The embedded script simply generates a canvas upon which some of the defined primitives are drawn (Table 3.1). The selection and position of the primitives on the canvas, as well as their colors and gradients, are dependent on the randomly chosen *seed* parameter. The final image is then extracted with the JavaScript method `toDataURL()` and the result hashed and returned.

Running OpenWPM against this page proved that it captured method calls and property accesses necessary to detect Device Class Fingerprint. For convenience, this HTML page was kept as reference also when crawling desired web pages. It was later decided to keep it as a “control” website during the data collection campaign, to show that the crawl worked as expected and that the collected data was sound.

4.1.2 Configuring OpenWPM

Among the data collection options offered by OpenWPM, there is the instrument to capture JavaScript calls. This instrument just needs to have its relative parameter `js_instrument` active and it will capture function calls as configured in an additional separate file. The `fingerprinting.json` file is pre-configured to capture JavaScript function calls known to be used in fingerprinting techniques.

OpenWPM also offers the possibility to log the “content hash” of HTTP responses. To get this information, OpenWPM needs its relative `save_content` parameter to be active, the `http_instrument` and an additional “unstructured” file storage to be configured (different from the pre-configured SQLite database). A LevelDB[1] storage served this purpose, which allowed to log *key-value* pairs, with the *value* being the full content of the response and the *key* being its “content hash”. This value is necessary to perform the “aggregate by hash” approach introduced in Section 3.3.

4.1.3 Extending OpenWPM

Following the results of the fit-gap analysis presented in Subsection 3.2.1, OpenWPM was extended with additional custom functionalities needed to fulfil our research goals.

Capturing Return Values

Capturing the return value of intercepted function calls is not offered out of the box by OpenWPM. The return value is especially needed for implementing the distinguishing features related to the final canvas image. It is impossible to evaluate size, colors and format of an image without getting the return value from `toDataURL()` and `getImageData()`.

Mobile Crawling

When crawling a website, OpenWPM inherits its browsing identity and “`User Agent`” from the machine it is running. One of our research goals is about whether there are any notable differences between desktop and mobile crawling, but OpenWPM does not offer such navigation possibility.

We relied on the possibility of OpenWPM to define custom parameters, to define an “`isMobile`” boolean check, which controls the screen resolution and “`User Agent`” of the Selenium driver. When active, this parameter changes these values to those of a Pixel 2 XL. We took inspiration from the work of *Das et al.* [5] and their OpenWPM-mobile[2]. Although their implementation

[1] https://github.com/google/leveldb
[2] https://github.com/sensor-js/OpenWPM-mobile

was based on an older and slightly different version of OpenWPM, it still offered valuable insight on how to achieve it.

Giving Consent

OpenWPM does not offer the possibility to identify a data processing banner and interact with it. Nevertheless, it offers the possibility to define a custom "`Command`" on top of its extensible framework, which can then be queued after the main crawling command. Details on how we implemented the "give consent" functionality can be found in Appendix B.

4.1.4 Cloud Crawling Requirements

Digital Ocean's cloud instances are accessible through a command-line only interface, which upon closure would terminate the running session and any processes spawned from it. To allow OpenWPM to keep running in the background, we relied on Tmux[3], a terminal multiplexer. Moreover, as this solution does not provide a screen, we could not rely either on the native or the headless browser's mode. Luckily, OpenWPM offers the possibility to run the browser through Xvfb[4], a virtual display that performs all graphical operations in virtual memory. Lastly, as one of our approach is to try to give consent for personal data processing, we deployed all cloud instances on servers within the EU. This gives us enough confidence that a request for consent should be prompted.

4.2 Data Collection Campaign

The data collection campaign was conducted between April and May 2023. We run the same crawl strategy[5] in groups of 5 instances, aiming at having at least 3 complete crawls for the same set of URLs. We decided to run our crawl strategies with 3 parallel browsers as a result of the hardware capabilities of a cloud instance reviewed in Subsection 3.2.2. Additionally, we defined these crawl strategies to simply crawl a range of URLs as intended by OpenWPM, or by either trying to give consent for data processing or emulate a mobile device to better compare crawl data. All these configuration combinations are shown in Table 4.1.

Due to long crawling times and frequent crashes of the cloud instances, it was decided to lower our target of URLs to crawl from 100K to 40K. We believe it is still a relevant number to discuss the distribution of the researched technique.

[3]`https://github.com/tmux/tmux/wiki`

[4]`https://x.org/releases/X11R7.7/doc/man/man1/Xvfb.1.xhtml`

[5]A crawl strategy is the combination of parameters used to perform the crawl. It is the combination of range of URLs, number of browsers, try to give consent for data processing and emulate mobile.

	Crawl Strategy		
Tranco Ranks	**Num. Browsers**	**Give Consent**	**Mobile**
1 - 20K	3	-	-
1 - 20K	3	✓	-
1 - 20K	3	-	✓
20K - 40K	3	-	-
20K - 40K	3	✓	-
20K - 40K	3	-	✓

Table 4.1: List of crawl strategies for data campaign grouped by Tranco rank, and their distinctive parameters.

Chapter 5

Data Analysis and Results

In the previous Chapter, we presented our contributions and implementation details. First, we selected OpenWPM as our crawling and data collection tool and augmented its capabilities to meet our needs. We then described how our heuristic was implemented and finally, we performed the data collection at scale on 40K URLs multiple times, to have enough comparable data. Finally, we designed and tested our analysis tool, as well as defined how to approach the analysis of the full dataset.

In this Chapter, we tested our analysis tool with part of the crawled data to define how to approach the analysis of the full dataset (5.1). Once defined the desired strategy, we performed an offline analysis of the full collected dataset. First, in Section 5.2 we review the collected crawled data in relation to the defined *Analysis Strategies*. Then, in Section 5.3 we review the results of crawling with giving consent for data processing. In Sections 5.4 through 5.7 we investigate the information that can be inferred from the scripts generating *Picasso*, *Picasso*-like, "shapes" and "numbered" canvas images.

5.1 Defining an *Analysis Strategy*

Although the heuristic was developed before the data collection campaign, a definite analysis strategy was not yet defined. As scripts generating canvas images could behave very differently, we assumed that even those performing *Picasso* could be slightly different than anticipated. As a stricter approach could lead to overfitting for a specific canvas image, we worked in the direction of defining multiple *analysis strategies*. In this way we managed to gradually relax the search for features in order to detect eventual false negatives from a stricter approach. This should mitigate the specificity of the final outputs if only one analysis strategy were applied.

Due to the many distinguishing features and aggregation possibilities defined by the analysis tool, we needed a way to identify which combination of these would yield the best results. An *analysis strategy* is then a possible combination of evaluated features, aggregation strategy and other detection considerations. This selection process was carried out through various steps and tests and is documented in Appendix E. During the evaluation, we not only identified implementation of *Picasso*, but also a set of scripts which generate *Picasso*-like canvas images.

We also identified a set of scripts with a degree of randomness in the generated canvas images. These images contain either random numbers or various shapes in variable positions, which we called “numbered” and “shapes” canvas images, respectively (Figure 5.1). However, these last were not an implementation of *Picasso*, as the scripts generating these canvas images also produce canvases known to be used in device fingerprinting.

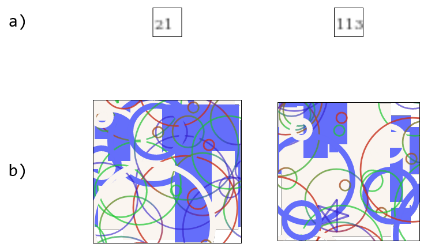

Figure 5.1: Examples of detected “numbered” (a) and “shapes” (b) canvas images.

We then selected three analysis approaches, namely *Analysis Strategy 1*, *2* and *3* and their selection is explained in detail in Appendix E.3. The first aims at detecting only implementations of *Picasso*, whereas the other two were selected as an attempt to investigate “shapes” and “numbered” canvas images. Due to the static and constant nature of scripts generating *Picaso*-like canvas images, our analysis tool is unable to detect them. No *analysis strategy* could be defined to identify them. As these scripts do not contain any randomness, we do not believe they are an implementation of *Picasso*. We discuss this further in Section 5.5.

In the following sections, we observe the detection capabilities of selected analysis strategies, from the very strict *Analysis Strategy 1*, which considers all the defined features, to *Analysis Strategy 3*, the most relaxed one, which looks only at a relaxed version of the “*text length constraint*” feature (3.1.3).

5.2 Crawls Overview

As introduced in the previous Section, the analysis possibilities offered by the developed tool were tested and evaluated against a small sample of the collected data. From these tests, we selected three *Analysis Strategies* with low false positive rates and good accuracy in detecting targeted canvas images.

Additionally, as introduced in Section 4.2, websites were crawled through OpenWPM alone, or with our extensions trying to give consent for data processing or emulating a mobile device. In this Section, we will refer to this crawling concept as *crawl type*, which will be either No-Action, Give-Consent or Mobile. Table 5.1 shows the overall results for all the crawls grouped first by Tranco rank and then by crawl type.

On average, all crawls performed the same, with an average failure rate of around 17%. There are although two exceptions in the 20K-40K rank for No-Action and Give-Consent crawl types. Due to the frequent interruption in crawls presented in Section 4.2, these results were expected considering the rate at which the relative crawl strategies failed.

Tranco ranks	Crawl Type	Total Websites Crawled †	Avg. Crawl Failure	Unique Websites Successfully Crawled	Websites using Canvas API	Total Scripts
1-20K	No-Action	80004	15%	17727	5767	9752
	Give-Consent	80004	17%	17698	5915	10262
	Mobile	120006	17%	17789	5367	10478
20K-40K	No-Action	100020	25%	17559	5581	8694
	Give-Consent	60012	24%	17340	5614	8499
	Mobile	80006	18%	17630	5228	8336

Table 5.1: Overall results of data collection campaign.
†: This value is the range of crawled websites × the number of complete crawls for this rank and crawl type. The few additional numbers above the total are how many times the "control" website was crawled in that dataset.

We also looked at how many websites contained scripts that use the Canvas APIs, before performing any additional check through our heuristic. The Canvas APIs seem to be used by slightly more than a fourth of the total crawled websites, and each of these contains on average between 1 and 2 scripts that make use of these APIs.

5.2.1 Analysis Strategy 1

In this Subsection, we evaluate the output for applying *Analysis Strategy 1* to the collected data. This strategy aims specifically at detecting implementations of *Picasso* only. It relies on *Approach 1* in its "toDataURL" variant with the "Within-FuzzyAgg" *Navigation* and *Aggregation* strategy combination as defined in Appendix E.3. Table 5.2 shows the results for *Analysis Strategy 1*.

Tranco ranks	Crawl Type	Scripts with DCF †	Scripts with DCF † (with agg. strategy)	Websites with DCF †	False Positives (websites)	True Positives (websites)	Accuracy
1-20K	No-Action	42	14	13	2	11	85%
	Give-Consent	39	17	16	8	8	50%
	Mobile	56	18	17	6	11	65%
20K-40K	No-Action	48	18	17	7	10	59%
	Give-Consent	28	16	15	5	10	67%
	Mobile	86	59	57	45	12	21%

Table 5.2: Overall results for *Analysis Strategy 1*.
†: DCF stands for Device Class Fingerprint.
Detection of pure *Picasso* implementations according to our heuristic. True positives are consistent regardless of tranco rank or crawl type.

We can see that the detection rate is consistent across both the Tranco ranks and crawl types, with the sole exception in the 20K-40K rank for Mobile crawl type. Here we can see a spike in both the detection rate and in the number of verified false positives. Nevertheless, the final output of encountered true positives is stable with no evident increase in the use of the technique at the variations of the crawl type or across the two inspected Tranco ranks.

We can also see how the aggregation strategy significantly helped in detecting scripts implementing *Picasso*. For instance, the number of websites that contain a script performing the technique is almost the same number of detected aggregated scripts, whereas the number of distinct scripts is much higher and oscillates between 2 and 4 times the number of aggregated scripts. This shows how these scripts performing *Picasso* have variations in their URL's parameters, as pointed out in Appendix C.2.

	Crawl Type		
URL	**No-Action**	**Give-Consent**	**Mobile**
bitcoinmagazine.com	✓		✓
blablacar.fr			✓
carsales.com.au			✓
cma-cgm.com			✓
commercialtrucktrader.com		✓	
cycletrader.com	✓	✓	✓
delishably.com	✓		✓
govx.com	✓		✓
ha.com		✓	
hellyhansen.com		✓	
hermes.cn			✓
hermes.com			✓
hubpages.com	✓	✓	✓
idealista.pt	✓	✓	✓
inc.com	✓	✓	✓
londondrugs.com			✓
mensjournal.com	✓		
organicauthority.com	✓	✓	✓
parade.com	✓		✓
petco.com	✓	✓	
seloger.com	✓	✓	✓
rvtrader.com	✓	✓	
thefork.com	✓	✓	✓
thefreethoughtproject.com	✓	✓	✓
thehockeynews.com	✓	✓	✓
thestreet.com	✓	✓	✓
turbofuture.com	✓	✓	✓
zocdoc.com	✓		

Table 5.3: Websites with a script performing *Picasso* and the crawl type in which they were detected.

Considerations on true positives

As the *Analysis Strategy 1* aims at detecting only implementations of *Picasso*, our results show that no first party makes use of such technique. According to our findings, true positives were all generated by a third party script provided by the same service provider. We inspect and analyse the canvas images generated by this script in Section 5.4.

Table 5.3 shows the complete list of websites that hosted this *Picasso* implementation. In total we detected 28 websites. The table also shows that not all websites were always detected across all crawl types. This might be due to the script being triggered only as a stronger verification measure against bots.

Finally, we inspected how these websites were ranked in the used Tranco list. The box plots in Figure 5.2 show the distribution of the found true positives divided by crawl type and then observed as a whole group. Here we can see how this implementation of *Picasso* is evenly distributed across the whole crawled websites dataset. However, the distribution of this single implementation simply reflects the range of this single provider's customers, from very popular websites to least known ones.

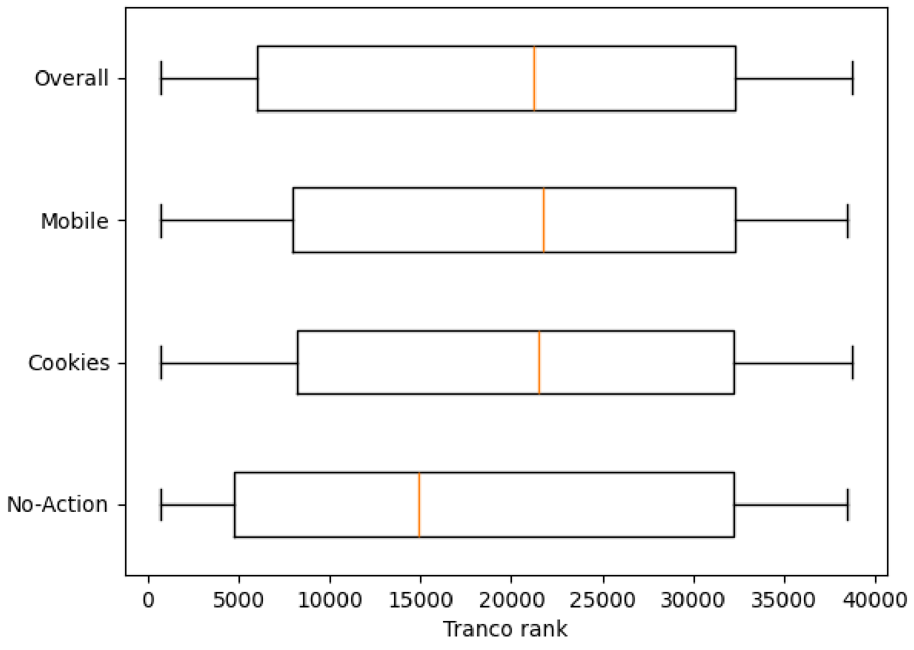

Figure 5.2: Distribution of true positives across the Tranco ranks for *Analysis Strategy 1*. Diagram shows distribution for each crawl type and overall.

Through this *Analysis Strategy*, we managed to answer some of our research questions. First, we showed that a third party is implementing this technique at this point in time and providing it "as a service". Second, according to these results, we detected 28 websites out of 40K crawled that host an implementation of *Picasso*. Its distribution stands at around 0.07% of the original crawled websites dataset, making this technique not very widespread. Lastly, we showed that the technique is evenly distributed across the considered websites dataset, with no notable shift towards more popular websites on the Tranco rank.

5.2.2 Analysis Strategy 2

In this Subsection, we evaluate the output for applying *Analysis Strategy 2* to the collected data. This strategy aims at detecting implementations of *Picasso* together with "numbered" canvas images. It relies on *Approach 2* in its "toDataURL" variant with the "Within-FuzzyAgg" *Navigation* and *Aggregation* strategy combination as defined in Appendix E.3. Table 5.4 shows the results for *Analysis Strategy 2*.

Tranco ranks	**Crawl Type**	**Scripts with DCF †**	**Scripts with DCF † (with agg. strategy)**	**Websites with DCF †**	**False Positives (websites)**	**True Positives (websites)**	**Accuracy**
1-20K	No-Action	339	311	302	2	300	99%
	Give-Consent	357	331	309	10	299	97%
	Mobile	461	423	351	33	318	91%
20K-40K	No-Action	198	171	154	12	142	92%
	Give-Consent	150	137	134	9	125	93%
	Mobile	252	223	176	48	128	73%

Table 5.4: Overall results for *Analysis Strategy 2*.
†: DCF stands for Device Class Fingerprint.
Detection of relaxed *Picasso* implementations according to our heuristic. *Variability of primitives* is ignored and *Text length constraint* is relaxed. True positives are consistent within tranco rank, but more present in the higher range of domains.

Differently from *Analysis Strategy 1*, this strategy shows a clear difference in the detection rate between the two Tranco ranks. The top 20K rank has at least twice the detection hit of the 40K one. Otherwise, comparing the true positive results for the two Tranco ranks returns us a consistent presence of these canvas images across all crawl types.

It is worth noticing how the accuracy greatly improved in this *Strategy* when compared to *Analysis Strategy 1*, although the detection threshold is still at `0.5`. This is obviously due to the popularity of scripts generating "numbered" canvas images when compared to *Picasso* implementation scripts, which are still part of the true positives here. *Analysis Strategy 2* then proves to be effective in detecting these "numbered" canvas images, which floods the detection algorithm's result and improves the overall accuracy at the cost of very few additional false positives in respect to *Analysis Strategy 1*.

Considerations on true positives

Due to the relaxed approach of this *Analysis Strategy*, results include not only "numbered" canvas images, but also a valid implementation of *Picasso*. However, the magnitude of their presence allows us to consider the presence of *Picasso* implementations as negligible for this analysis. In addition to this, by inspecting the results we noticed that no domain is hosting scripts that perform both techniques for *Picasso* or related to "numbered" canvas images.

Our results show that scripts generating "numbered" canvas images are quite popular. In particular, the scripts are mostly first party, with them being around 10 times more than when provided by a third party. Figure 5.3 shows the popularity of the technique. It also shows how this technique is more popular in the top 20K rank. We further analysed these scripts and generated "numbered" canvas images in Section 5.7.

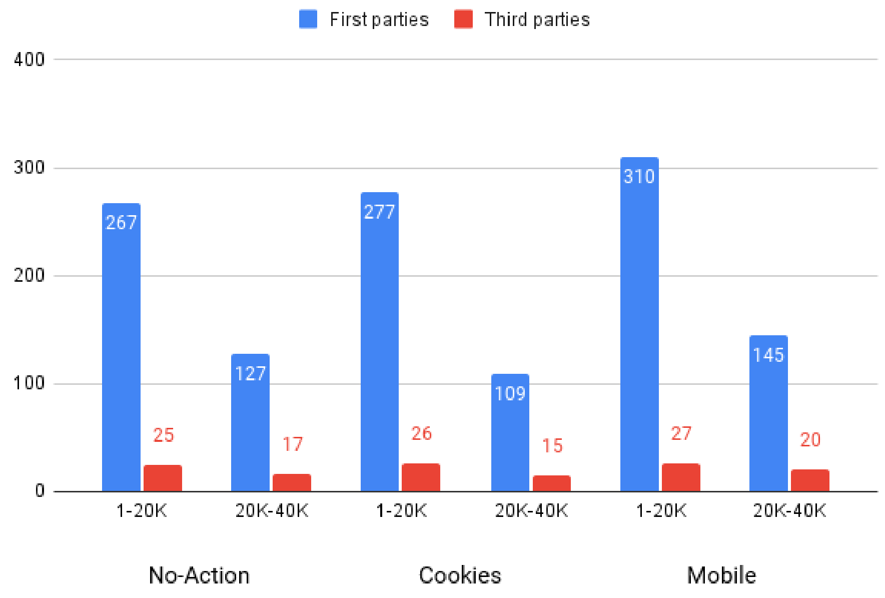

Figure 5.3: First vs. Third parties scripts distribution count detected with *Analysis Strategy 2*. First party scripts are around 10x more popular than Third party scripts.

Finally, we inspected how the detected websites were ranked in the used Tranco list. The box plot in Figure 5.4 shows the distribution of the found true positives as an aggregated result of all crawl types. As already noted in Figure 5.3, scripts generating "numbered" canvas images are most popular in the top 20K rank, which means are mostly used by popular websites.

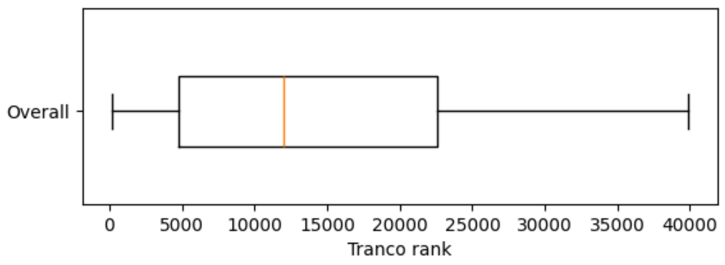

Figure 5.4: Distribution of true positives across the Tranco ranks for *Analysis Strategy 2*.

5.2.3 Analysis Strategy 3

In this Subsection, we evaluate the output for applying *Analysis Strategy 3* to the collected data. This strategy aims at detecting a potential implementation of *Picasso* in the form of "shapes" canvas images. Additional details on *Analysis Strategy 3* can be found in Appendix E.3. Table 5.5 shows the results for *Analysis Strategy 3*.

Tranco ranks	Crawl Type	Scripts with DCF †	Scripts with DCF † (with agg. strategy)	Websites with DCF †	False Positives (websites)	True Positives (websites)	Accuracy
1-20K	No-Action	18	13	13	8	5	38%
	Give-Consent	20	13	13	8	5	62%
	Mobile	23	14	13	9	4	31%
20K-40K	No-Action	10	10	10	7	3	30%
	Give-Consent	13	12	12	9	3	25%
	Mobile	11	8	7	5	2	29%

Table 5.5: Overall results for *Analysis Strategy 3*.
†: DCF stands for Device Class Fingerprint.
Detection of relaxed *Picasso* implementations according to our heuristic. Only relaxed *Text length constraint* is considered. True positives are consistent within tranco rank, but slightly more present in the higher range of domains.

Due to the low true positive results and low accuracy, definitive conclusions are hard to draw. Nevertheless, we can see an overall consistency in the true positives. It appears that this technique is more used in the top 20K Tranco rank, similarly to scripts generating “numbered” canvas images in *Analysis Strategy 2*.

	Crawl Type		
URL	**No-Action**	**Give-Consent**	**Mobile**
becu.org		✓	
bell.ca	✓	✓	✓
capitalone.com	✓		
choicehotels.com	✓	✓	
ebates.com	✓	✓	✓
healthybenefitsplus.com	✓		
ncsecu.org	✓	✓	✓
newbalance.com		✓	
optimum.net		✓	✓
rakuten.com	✓	✓	✓
target.com	✓		✓

Table 5.6: Websites with a script generating “shapes” canvas images and the crawl type in which they were detected.

Considerations on true positives

As already presented, scripts generating “shapes” canvas images are not particularly popular. Table 5.6 shows the complete list of websites that hosted this technique. In total we detected 11 websites. Similarly to what we observed in *Analysis Strategy 1*, not all websites were always detected across all crawl types. In addition to this, by inspecting the results we noticed that no domain is hosting scripts that perform both techniques for *Picasso* or related to “shapes” canvas images. We further analysed these scripts and generated “shapes” canvas images in Section 5.6.

Upon inspection, we noticed that all scripts are first party implementations with the sole exception of one of the websites providing its solution to another domain. Finally, we inspected how these websites were ranked in the used Tranco list. The box plot in Figure 5.5 shows the distribution of the found true positives as an aggregated result of all crawl types. Here we can see how scripts generating “shapes” canvas images are strongly shifted to the top 20K range of the crawled data.

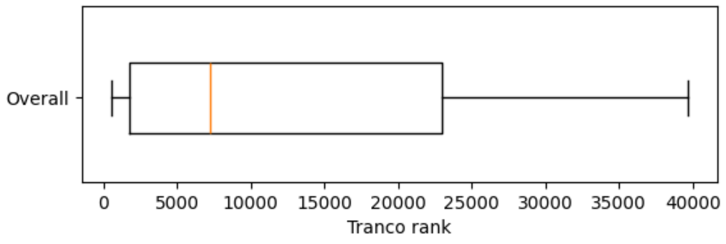

Figure 5.5: Distribution of true positives across the Tranco ranks for *Analysis Strategy 3*.

5.3 Give-Consent Algorithm Review

As part of our research questions, we were curious to see if trying to give consent for data processing on the websites we crawled would bring any variation in the behaviour of scripts implementing *Picasso*. As we have seen from the review of the three *Analysis Strategies*, it appears that giving consent for data processing does not have a notable impact on the usage of this fingerprinting technique.

In Figure 5.6, we can see how our algorithm did not detect any possible “give consent” button in around 75% of the time. The algorithm managed to detect a possible button and successfully clicked it at around 18% of the times, whereas the remaining 7% resulted in an error, which probably propagated through OpenWPM and corrupted the navigation for that URL.

5.4 *Picasso* Canvas Images Review

All true positives identified in Subsection 5.2.1 were generated by the same third party script. The script is provided by a cybersecurity company that offers a range of bot and fraud protection services. Among their services, their Captcha’s protection is claimed to preserve users’ privacy and to be 10x faster than regular Captchas.

Nevertheless, we tried to infer some characteristics of their implementation by inspecting the generated canvas images. In Figure 5.7 we can observe that this implementation is performing a series of rounds where each time a primitive or more are added to the canvas, as by definition

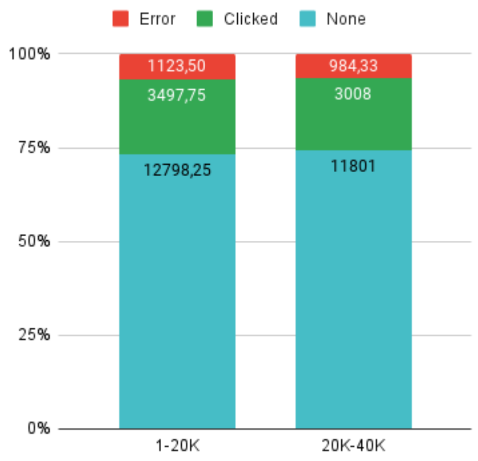

Figure 5.6: Success rate results of our custom give-consent algorithm. No "give consent" button was detected around 75% of the time, whereas it managed to detect and click a button around 18% of times. The remaining 8% of occasions resulted in an error generated by our algorithm's attempts.

of *Picasso*'s scheme from Subsection 2.3.1. Moreover, the number of possible sequences of generated canvas images is fixed. Figure F.3 shows all identified canvas images generated by this implementation. According to *Picasso*'s scheme from Subsection 2.3.1, this could be the service provider's knowledge base against which *Picasso*'s verification step is performed.

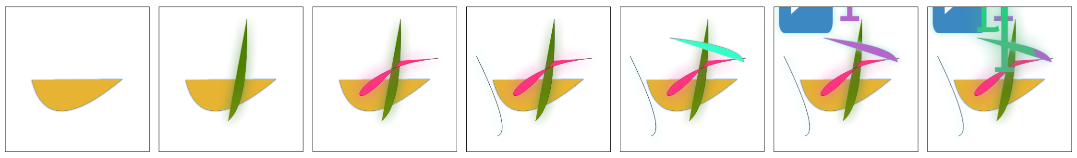

Figure 5.7: Full sequence of generated canvas images for the detected *Picasso* implementation. It is clearly visible the fact that the underlying technique is performing rounds, adding each time a different primitive with variating position, color and gradient.

From Figures F.2 and F.3, we can infer also some additional information. All the identified sequences shown in Figure F.3 perform 7 rounds of primitives with a single exception, which performs 6 rounds. Although *Picasso* was designed to be resource scalable with variating number of rounds, we were surprised to find all identified executions to perform always the same number of rounds and not, for instance, occasionally a portion of them.

We also noticed two additional differences from the original *Picasso* scheme. First, it seems that at some point some of the already drawn primitives have their color changed in the following rounds. Second, from Figure F.2 it appears that at the end of each round, after the primitive is written, the canvas image is extracted twice. We believe that both of these choices are techniques to protect the scheme against possible attacks.

However, the presented characteristics make this scheme quite static and limited. One of the strengths of *Picasso*'s scheme is its possibility to challenge a client with variable complexity. In addition to this, the scheme is robust against spoofing attacks as it expects the provider to send multiple requests, some of which are part of a known knowledge base of challenge-response pairs and the others allow to increase it if all are valid. Here, the number of identified sequences is just 9. The limited number of challenges and the fixed number of rounds discovered in this provider's script made us assume that we are looking at a simplified technique inspired by *Picasso* which still serves its purposes.

To be sure we were not looking at an implementation flaw, we disclosed our findings about limited device class fingerprinting challenges to the service provider. They confirmed that they maintain and rotate a limited number of challenges to remain in control of their knowledge base. This is because in order to remain undetected, bots implement different strategies to randomise their canvas image, which then leads to the number of canvas hashes to explode. For this reason, they stepped back on a variation of the original algorithm. However, following our disclosure, they shared their plan to reinforce their Captcha challenges with other techniques. This disclosure from them could be the reason why we found a limited usage of *Picasso*.

5.5 *Picasso*-like Canvas Images Review

In Section 5.1, we identified a set of *Picasso*-like canvas images. The scripts that generated these canvases always returned the same output at each execution. As our heuristic relied on variation of primitives and variation of their arguments, these scripts were always marked as true negatives. From the 6 *Picasso*-like detected in our dataset (Figure F.4), we observed the following characteristics:

- `Number 1` was found in three flavours. Considering `1a` as reference, `1b` differs in the font it uses for the text, whereas `1c` is identical, but smaller in size;
- `Number 4` is the only one that seems to perform 7 rounds, but again, these canvas images are constant in each execution. It is used on Japanese domains only;
- `Number 5` is the only one that is extracted through `getImageData()`;
- `Number 6` generates two different canvas images which do not appear like the script is performing any round;
- `Numbers 4` and `6` are the only ones that generate more than one canvas image.

We then noticed that the scripts that generated these canvas images are used on few websites, with the sole exception of `Number 1`. Although the `1b` variant is only used by *researchgate.net*, `1a` and `1c` were detected on 545 and 312 websites respectively, for a total of 630 distinct websites and with 72% of `1c`'s websites being shared with `1a`. This implies, and it was verified, that the scripts that share the same domain, generate both canvas images i.e. the same canvas in two different sizes. Nevertheless, as shown in Figure 5.8, it appears that the `1c` variant is predominant in the lower part of the inspected Tranco range.

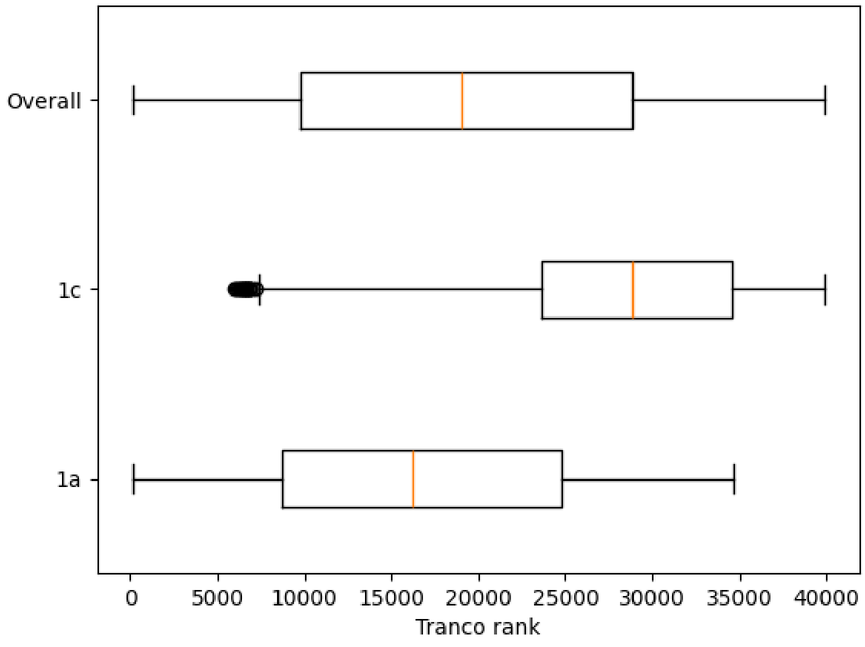

Figure 5.8: Distribution of variants `1a` and `1c` of *Picasso*-like canvas images and their overall distribution, when considered together.

We then evaluated their balance between first and third party scripts. We found out that 92% of the time, both `1a` and `1c` canvas images are generated by a first party script[1]. However, all the scripts' URLs have in their hosting path minimal variations, namely:

$$<domain>/(b|g)/orchestrate/(managed|jsch|captcha)/v1$$

Based on this and on the canvas match, we believe they all share the same technique, although most are first party hosted.

Lastly, we searched for the most popular domain categories among the detected 630 websites that hosted a script that generated either `1a` or `1c` canvas images. We relied on Webshrinker's Domain API Feed[2] to extract IAB categories[3]. Table 5.7 shows the top 10 found IAB categories and their total occurrence.

Regardless of their similarities with the original algorithm, the scripts that generate these canvas images are definitely not an implementation of *Picasso*. We believe that there are two possibilities why these scripts generate *Picasso*-like canvases. The first is that the provider of these scripts implemented a full *Picasso* algorithm in the past and then decided to abandon it, but kept their fingerprinting technique based on the primitives used by *Picasso*. The second possibility is that the provider of the scripts never intended to use *Picasso* in the first place, but

[1]Detected third party scripts were: 'vlxx.cc', 'empire-streaming.app', 'dramacool9.pw', 'animevietsub.in', 'zgogc.com', 'dsd10.lol', 'upwork.com', 'wgetcloud.org', 'buffer.com', 'sextop1.vin', 'animehay.live', 'rita99.com'
[2]`https://www.webshrinker.com/`
[3]IAB Tech Lab Content Taxonomy: `https://iabtechlab.com/standards/content-taxonomy/`

IAB Id	IAB Label	Total Occurence
IAB19	Technology & Computing	127
IAB25	Non-Standard Content	72
IAB9	Hobbies & Interests	50
IAB24	Uncategorized	49
IAB3	Business	39
IAB5	Education	36
IAB22	Shopping	36
IAB12	News / Weather / Information	36
IAB1	Arts & Entertainment	34
IAB11	Law, Government, & Politics	20

Table 5.7: Top 10 IAB categories and their total occurrence for websites hosting a script generating either `1a` or `1c` *Picasso*-like canvas images.

based their fingerprinting technique on a *Picasso*-like canvas to distinguish themselves from other fingerprinting canvases, for whatever reasons.

5.6 "Shapes" Canvas Images Review

All the "shapes" canvas images from the true positives identified in Subsection 5.2.3 are extracted by the Canvas API `getImageData()` method. As mentioned, almost all scripts are self-hosted by each domain but they all behave the same way. In full, these scripts also generate another canvas image known to be used in fingerprinting, together with other very small canvas images. We will ignore the smaller ones as they are of negligible size and are excluded by the analysis tool as their size is less than 16×16 pixels.

Figure 5.9 shows the two main canvas images drawn during these scripts executions. What we first noticed is that the first canvas image is extracted through *toDataURL()* and was often detected as a false positive during the evaluation of *Analysis Strategy 1* of Subsection 5.2.1. However, it is interesting to notice how the "shapes" canvas image is not always generated. In addition to this, while crawling a specific domain multiple times, often the "shapes" canvas image did not change. However, across all the detected "shapes" canvas images from all the websites, none was repeated in a different domain other than itself.

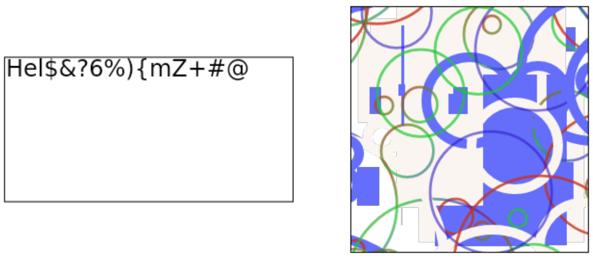

Figure 5.9: Main canvas images generated from a "shapes" script.

As we have mentioned in Subsection 5.2.3, the low popularity of this scheme does not allow us to draw definitive conclusions. Moreover, we were not originally investigating this specific canvas image and we are definitely overlooking some of its characteristics. However, we simply believe that this canvas images are generated as a means to create noise against possible detection mechanisms for classic fingerprinting, possibly, after some suspicious behaviour from our crawler being tagged as a bot.

We have already shown how hard it is to draw a line to exclude noise and false positives for our analysis. Although classic canvas fingerprinting relies on a fixed and very specific canvas image, detection algorithms against this technique would look for patterns. Assuming that it is already hard to detect and prevent canvas fingerprinting without disabling the full capabilities of the Canvas API, it is not a surprise to see fingerprinters hardening their resilience against identification by injecting random noise. Appendix F.5 shows all the different "shapes" canvas images detected by our research.

5.7 "Numbered" Canvas Images Review

Almost all the "numbered" canvas images' scripts from the true positives identified in Subsection 5.2.2 are self-hosted. In full, these scripts also generate another canvas image known to be used in fingerprinting. Figure 5.10 shows the main canvas images drawn during these scripts executions.

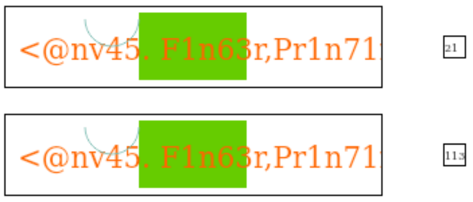

Figure 5.10: Main canvas images generated from a "numbered" script.

These scripts apparently perform two rounds, in which they generate a known fingerprinting canvas image with a pangram, followed by a separate and very small canvas image containing always a different random number. These "numbered" canvas images are always of the same 16×16 pixel size, making them a very edge case when considering the image constraints we defined in 3.1.2. As defined by *Acar et al.* [1], canvas images smaller than this specific size might not have anti-aliasing applied and lose an important source of diversity at the pixel level for fingerprinting. We are unsure of the purpose of this random number, but we could speculate that these canvas images are generated as a mean to create noise against possible detection mechanisms in a similar way as seen in Section 5.6 for "shapes" canvas images.

Chapter 6

Conclusions

In this Chapter we review how this research performed in terms of achievements. In Section 6.1 we review our contributions and the goals achieved by this research. In Section 6.2 we review the limitations we had to face. Lastly, Section 6.3 presents unanswered research questions and possible future implementations based on this research's results.

6.1 Achieved Research Goals

This research aimed to identify the popularity and distribution of device class fingerprinting in its *Picasso* implementation. Our heuristic was able to detect only one implementation of it. Its distribution stands at around 0.07% of the original crawled URLs dataset resulting in this technique being not particularly popular. This result was nevertheless possible thanks to the intermediate results obtained in the research sub-goals.

As a first step, it was vital to identify which are the distinguishing features of *Picasso*. Through the definition of its scheme and relevant literature review, the defined heuristic managed to successfully identify one implementation of it in the wild. The analysis tool was designed to offer the flexibility to configure which features or constraints to consider during the analysis. This allowed us to successfully search for other scripts implementing other canvas fingerprinting even though they were not an implementation of *Picasso*.

Moreover, due to the variable nature of *Picasso*, we needed to compare data from at least two different crawls. To then put data together, we defined different aggregation strategies: the first compared the full URL of scripts, the second aggregated scripts by their content hash as provided by OpenWPM, and the third aggregated them by comparing a portion of their URLs. Although OpenWPM was discovered to be flawed in providing the script hashes, it was also evident how the variation in URLs' arguments did not allow us to compare them by the full URL. In addition, these variations probably have a role in servicing a different script to execute.

Lastly, we detected an implementation of *Picasso* which was provided "as a service" to first party domains. Its usage was evenly distributed across the considered URLs dataset, with no particular notable shift towards more popular websites on the Tranco rank. In parallel, we identified a more widespread usage of *Picasso*-like canvas images: they are visually similar to

Picasso-generated ones but with no variations whatsoever upon multiple executions of the same script. Differently, these were mostly first party hosted.

Although these results do not allow us to draw definitive conclusions on *Picasso*'s prevalence, to the best of our knowledge there is no first party currently using this technique as device class fingerprinting. Not even Google, which originally presented the solution.

6.2 Limitations

To the best of our knowledge, we are the first to investigate the diffusion of device class fingerprinting as a technique. We concentrated and looked specifically for implementations of *Picasso* because is the only publicly available scheme building this kind of fingerprinting. Although an implementation of *Picasso* and an additional device class fingerprinting technique were discovered by *Bird et al.* [3], their research mainly focused on detecting fingerprinting in general. Due to the scarcity of literature on the topic, it was really hard to have a clear view of the steps to perform. This limitation impacted our methodology in defining and detecting possible distinguishing features.

The defined *text length* feature was a particularly relevant one, as it was the only feature that remained present throughout all the *Analysis Strategies* reviewed in Chapter 5. We based our check on the length mentioned by *Papadogiannakis et al.* [24] and by *Englehardt and Narayanan* [8]. In particular, *Englehardt and Narayanan* expected the text to display "at least 10 distinct characters", but we did not implement this feature as described. Our feature only checked the length of the text written to a canvas. We then evaluated any text variation through the *variability of arguments* feature, although is not exactly a check for "distinct" characters. We did not actively check as such as we did not expect to find any "same character" string. We add this detail as a limitation, although we do not believe that this omission added noise to our analysis.

Moreover, we implemented the *text length constraint* feature with the possibility to relax it. However, when used, this relaxation introduced a lot of false positives as the length evaluation does not filter for encoded emojis. This was intended as *Bird et al.* [3] discovered that emojis are a possible valid primitive in Facebook's implementation of *Picasso*. We did not invest time in identifying a way to discriminate and accepted the limitation of the relaxed approach.

Lastly, we encountered additional limitation with the used tools. Although OpenWPM is a powerful and very flexible tool, it comes with its known limitations and issues. First, it is built on Selenium which was not intended to be used as a crawler for intensive and scaled data collection campaigns. The authors did a great job in making their infrastructure as robust and reliable as possible, but as acknowledged by themselves, one "disadvantage of Selenium is that it frequently hangs indefinitely due to its blocking API" [8]. We encountered some of these issues related to its instability during the data collection campaign (4.2). The running crawler would often not properly close its running sessions or create too many temporary files which caused it to stop for unavailable disk space. This instability lead us to scale down our original crawling goals from the original 100K down to 40K domains.

Moreover, although OpenWPM offered the possibility to capture the “content hash” of HTTP response content, it resulted in same scripts hashing to different values, making this information not fully reliable for our aggregation strategy. This is a known issue[1] as this feature was not intended to be used as an actual valuable data point in research. We accepted this limitation of OpenWPM knowing it might not be 100% reliable.

Another limitation of our approach is related to mobile crawling. Although it would have been ideal to browse with a real mobile device, it was easier and more practical to implement an emulation of it within OpenWPM.

6.3 Future Work

As anticipated in Section 6.2, once we realised how little the literature on the topic was, it became clear that our research needed to be relatively simple in concept in order not to risk to go too much astray. In addition to this, some of the already presented limitations were as such because of the lack of ground information from other sources. Therefore we had to build either on assumptions or totally set aside possible interesting research questions which were going to take too much effort to include.

For instance, considering the capabilities of OpenWPM of both logging HTTP requests and responses, we envisioned the possibility of detecting HTTP traffic containing the actual shared fingerprint. It could be interesting to verify if it is possible as it might give insight into possible new fingerprinting techniques or even reveal a new trend not yet described.

As a possible additional investigation, it might be worth selecting a set of very well-known domains and implementing a custom OpenWPM command to try login on them. This action might trigger additional scripts for bot detection including other implementations of *Picasso*. It should be relatively easy to implement and might give back interesting results.

Lastly, another possible future implementation is related to our analysis tool. As it was designed with flexibility in mind, it allowed us to tweak our research approach and look for the best strategy. However, it proved to be useful in searching also other canvas fingerprinting scripts. This analysis tool could be extended to include additional features and possibly detect other techniques relying on Canvas API, making it useful as a generic analysis tool, rather than one for *Picasso* implementations only.

[1] https://github.com/openwpm/OpenWPM/issues/711#issuecomment-656859147

Bibliography

[1] Gunes Acar et al. “The Web Never Forgets: Persistent Tracking Mechanisms in the Wild”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 674–689. ISBN: 9781450329576. DOI: 10.1145/2660267.2660347.

[2] Babak Amin Azad et al. “Web Runner 2049: Evaluating Third-Party Anti-bot Services”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Clémentine Maurice et al. Cham: Springer International Publishing, 2020, pp. 135–159. ISBN: 978-3-030-52683-2.

[3] Sarah Bird et al. *Actions speak louder than words: Semi-supervised learning for browser fingerprinting detection*. 2020. arXiv: 2003.04463 [cs.CR].

[4] Elie Bursztein et al. “Picasso: Lightweight Device Class Fingerprinting for Web Clients”. In: *Workshop on Security and Privacy in Smartphones and Mobile Devices*. 2016.

[5] Anupam Das et al. “The Web’s Sixth Sense: A Study of Scripts Accessing Smartphone Sensors”. In: *Proceedings of the 25th ACM Conference on Computer and Communication Security (CCS)*. ACM, Oct. 2018. DOI: 10.1145/3243734.3243860.

[6] *Data Protection*. URL: https://www.edps.europa.eu/data-protection/data-protection_en (visited on 05/30/2024).

[7] Peter Eckersley. “How Unique Is Your Web Browser?” In: *International Symposium on Privacy Enhancing Technologies*. 2010. URL: https://api.semanticscholar.org/CorpusID:15233734.

[8] Steven Englehardt and Arvind Narayanan. “Online Tracking: A 1-Million-Site Measurement and Analysis”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 1388–1401. ISBN: 9781450341394. DOI: 10.1145/2976749.2978313.

[9] Tatiana Ermakova et al. “Web Tracking – A Literature Review on the State of Research”. In: Jan. 2018. DOI: 10.24251/HICSS.2018.596.

[10] David Fifield and Serge Egelman. “Fingerprinting Web Users Through Font Metrics”. In: *Financial Cryptography*. 2015. URL: https://api.semanticscholar.org/CorpusID:42777664.

[11] Glenn Fleishman. *Cartoon Captures Spirit of the Internet*. Dec. 2000. URL: https://www.nytimes.com/2000/12/14/technology/cartoon-captures-spirit-of-the-internet.html (visited on 06/03/2023).

[12] Colin M. Gray et al. "The Dark (Patterns) Side of UX Design". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. , Montreal QC, Canada, Association for Computing Machinery, 2018, pp. 1–14. ISBN: 9781450356206. DOI: 10.1145/3173574.3174108.

[13] imperva.com. *2022 Imperva Bad Bot Report*. 2022. URL: https://www.imperva.com/resources/resource-library/reports/bad-bot-report/ (visited on 04/14/2024).

[14] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. "Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1143–1161. DOI: 10.1109/SP40001.2021.00017.

[15] Nikhil Jha et al. "The Internet with Privacy Policies: Measuring The Web Upon Consent". In: *ACM Transactions on the Web* 16.3 (Aug. 2022), pp. 1–24. DOI: 10.1145/3555352.

[16] Keven Knight. *Why Data Breaches Are Increasing And What CISOs Can Do About It*. Apr. 2023. URL: https://www.forbes.com/sites/forbestechcouncil/2023/04/20/why-data-breaches-are-increasing-and-what-cisos-can-do-about-it/ (visited on 06/22/2023).

[17] Victor Le Pochat et al. "Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation". In: *Proceedings of the 26th Annual Network and Distributed System Security Symposium*. NDSS 2019. Feb. 2019. DOI: 10.14722/ndss.2019.23386.

[18] Vincent Lefrere et al. "The Impact of the GDPR on Content Providers". In: 2020. URL: https://weis2018.econinfosec.org/wp-content/uploads/sites/8/2020/06/weis20-final43.pdf.

[19] Arunesh Mathur et al. "Dark Patterns at Scale: Findings from a Crawl of 11K Shopping Websites". In: *Proc. ACM Hum.-Comput. Interact.* 3.CSCW (Nov. 2019). DOI: 10.1145/3359183.

[20] Jonathan R. Mayer and John C. Mitchell. "Third-Party Web Tracking: Policy and Technology". In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. SP '12. USA: IEEE Computer Society, 2012, pp. 413–427. ISBN: 9780769546810. DOI: 10.1109/SP.2012.47.

[21] Chris Mills. *Saying goodbye to third-party cookies in 2024*. Dec. 2023. URL: https://developer.mozilla.org/en-US/blog/goodbye-third-party-cookies/ (visited on 04/13/2024).

[22] Keaton Mowery and Hovav Shacham. "Pixel Perfect: Fingerprinting Canvas in HTML5". In: *Proceedings of W2SP 2012*. Ed. by Matt Fredrikson. IEEE Computer Society. May 2012. URL: https://hovav.net/ucsd/dist/canvas.pdf.

[23] Midas Nouwens et al. “Dark Patterns after the GDPR: Scraping Consent Pop-Ups and Demonstrating Their Influence”. In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI ’20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–13. ISBN: 9781450367080. DOI: 10.1145/3313831.3376321.

[24] Emmanouil Papadogiannakis et al. “User Tracking in the Post-Cookie Era: How Websites Bypass GDPR Consent to Track Users”. In: *Proceedings of the Web Conference 2021*. WWW ’21. 2021, pp. 2130–2141. DOI: 10.1145/3442381.3450056.

[25] Nayanamana Samarasinghe and Mohammad Mannan. “Towards a Global Perspective on Web Tracking”. In: *Comput. Secur.* 87.C (Nov. 2019). DOI: 10.1016/j.cose.2019.101569.

[26] surfshark.com. *Data breaches rise globally in Q3 of 2022*. Oct. 2022. URL: https://surfshark.com/blog/data-breach-statistics-2022-q3 (visited on 06/22/2023).

[27] “Universal Declaration of Human Rights”. In: (1948). URL: http://digitallibrary.un.org/record/666853.

[28] Antoine Vastel. *Demonstration of Picasso canvas fingerprinting*. Mar. 2019. URL: https://antoinevastel.com/browser%20fingerprinting/2019/03/21/picasso-canvas-fingerprinting.html (visited on 07/02/2023).

[29] Wikipedia contributors. *Jaccard index — Wikipedia, The Free Encyclopedia*. 2023. URL: https://en.wikipedia.org/w/index.php?title=Jaccard_index&oldid=1188130957 (visited on 12/10/2023).

Appendix A

Tranco list with ID 4K83X

List is available at `https://tranco-list.eu/list/4K83X`.
This list aggregates the ranks from the lists provided by **Alexa**, **Umbrella**, and **Majestic** from 02 March 2023 to 31 March 2023 (**30 days**).
The following filters were applied to the domains:

- Only pay-level domains were retained.
- Only domains included in the Chrome User Experience Report of February 2023, present in the global dataset, were retained.

Appendix B

Give-Consent Algorithm for Data Processing

B.1 Implementation

In OpenWPM we implemented a custom "`GiveConsentCommand`", which included the approach presented by *Jha et al.* [15]. Their Priv-Accept[1] solution is a Selenium-based crawler which is able to give consent for data processing while browsing. After a few tests, it appeared that the integration of this approach in OpenWPM degraded performance considerably. We additionally implemented a separate custom algorithm to give consent, which naively parsed the loaded HMTL page looking for `<button>` elements first and then any element with a `role` or `id` attribute set to "`button`". Each found element is then tested if its inner text contains any "accepting" word from the same list used in Priv-Accept.

We then performed a comparison test, crawling 200 URLs two times and checking the number of incomplete visits logged by OpenWPM. For reference we run additional crawls without giving consent, then one with Priv-Accept algorithm only and one last crawl with only our simple algorithm. We also tested a combination of the two, where first the Priv-Accept algorithm is executed and, if nothing was found, our algorithm attempted a search, and vice versa. The results are reported in Table B.1.

Give Consent Strategy	Clicked	Avg. Crawl Failure
None	-	8%
Priv-Accept	167	24%
Custom	132	8%
Priv-Accept + Custom	168	25%
Custom + Priv-Accept	184	21%

Table B.1: Results of tests with different "give consent" strategies. All tests were performed by navigating 400 websites each time.

As shown, although it appears that the Priv-Accept approach was able to perform a click

[1] `https://github.com/marty90/priv-accept`

around 25% more than our algorithm, it came at the cost of 3 times more degraded performance. Differently, our approach seemed to have no impact at all, when compared to crawling without giving consent. Looking at the combination of the two approaches seems to prove our point, with the crawl using our approach first and then Priv-Accept having the best result when it comes to "clicking", but with a failure degree very similar to the Priv-Accept only approach.

Based on these findings, and considering the performance degradation tested in Subsection 3.2.2, we accepted the custom algorithm to be good enough for the research purposes given the apparent non-degradation of performance it offered.

B.2 Evaluation

In Section 5.2 we reviewed how each crawl performed during the data collection campaign. Looking at this data we can see how crawling and giving consent for data processing performed similarly to crawling without. This demonstrates our consideration about the custom give-consent algorithm. This proved that our technique was in fact not particularly intrusive and did not have a notable negative impact when crawling. We reviewed the results of crawling and giving consent for data processing in Section 5.3.

Appendix C

Implementing Heuristic

We defined *Picasso*'s distinguishing features in Subsection 3.1.3 and their implementation was approached with two goals in mind. First, the defined distinguishing features might be too restrictive or too weak. This poses the risk of over-fitting or introducing too many false positives. It was important for the analysis to look at any combination of the defined features, in order to test and adapt the final analysis strategy. Each of the defined features was then implemented as an independent and configurable parameter. For a list of all parameters, see Appendix D. Second, to know if two scripts can be compared, we expect their URLs to be the same. However, this is not always the case and we had to anticipate this possibility by defining some aggregation strategies.

C.1 Implementing features

We approached the development of our tool with the need to flexibly change which feature to look at and in which combination. Most of *Picasso*'s device class fingerprinting indicators enumerated in Subsection 3.1.3 were implemented as independent checks, each controlled by its parameter.

It is also important to point out that most of the described features are used to discriminate whether the script is relevant for this research or is excluded completely. It is the case when checking for the presence of Canvas API's calls, which methods are used for image extraction, and the constraints on color and size of such images. Only a few of the described features are used to calculate the *probability* for the script to be an implementation of *Picasso*. These features are *Variability Of Primitives*, *Variability Of Arguments* and *Text Length Constraint*, as they are evaluated against different executions of the same script.

All features probability values are then averaged together to define the script's final probability. This value is then compared with a threshold parameter to verify if the script is a potential implementation of *Picasso* (as we will see in Subsection C.2). We decided to give each feature the same "weight" in the final average, although we know any of these features may have a greater impact than another.

Canvas API

As our analysis tool is designed to identify implementations of *Picasso*, it bases the analysis on the presence of calls to this API, making this feature the backbone of the analysis tool. This feature does not have its own parameter and is the sole exception to the above statements on configurability.

Variability Of Primitives

This check is implemented in two steps: first, for each execution of matching URLs script, its captured methods are filtered for this check's relevant methods (as defined in Table 3.1) and returned as a list. Then, all lists are compared with each other to determine the degree of differences (or variability of this script's execution). This is achieved by checking that each relevant method is called a different number of times across all executions and is expressed as:

$$variability_{primitives} = \frac{total\_differences\_found}{total\_num\_compared}$$

Values will range from 0 to 1, with a higher value implying a high variability of the executed methods, thus a sign of *Picasso* implementation.

Variability Of Arguments

This check is implemented with two different approaches and relative parameters. The first approach is implemented similarly to "variability of primitives", the second is implemented as a Jaccard index [1][29]. Both described approaches are kept due to some edge cases resulting in respective coefficients being opposite.

Custom comparison:
This check is implemented in two steps: first, for each execution of matching URLs script, captured methods are filtered for this check's relevant methods (as defined in Table 3.1), coupled with their calling arguments and then returned as a list. All lists are then compared with each other looking only at intersecting called methods. Then, for each method, it is checked if the arguments used are the same across all executions. The result is expressed as:

$$same\_args = \frac{total\_equal\_found}{total\_num\_compared}$$

As we aim to detect the variability between executions, we invert the result:

$$variability_{arguments} = 1 - same\_args$$

Values will range from 0 to 1, with a higher value implying a high difference in used arguments for the executed methods, thus a sign of *Picasso*'s implementation.

Inverted Jacacrd index:
This check is implemented in two steps: first, for each execution of matching URLs script,

[1]Also known as Jaccard similarity coefficient.

captured methods are filtered for this check's relevant methods (as defined in Table 3.1) and coupled with their calling arguments. As Jaccard works with sets, duplicates are removed and then returned. Then, all sets are compared with each other looking at the quotient between the set intersection of called methods and their set unions. Jaccard index is expressed as:

$$jaccard = \frac{interecting\_methods}{union\_of\_sets}$$

As the Jaccard index is an indicator of similarity and we aim at detecting differences between executions, we invert the result:

$$variability_{arguments} = 1 - jaccard$$

Values will range from 0 to 1, with a higher value implying a high difference in used arguments for the executed methods, thus a sign of *Picasso*'s implementation.

Image Extraction

This check is implemented with two different parameters, for `getImageData()` and `toDataURL()` respectively. This check is implemented in two steps: first, for each execution of matching URLs script, captured methods are filtered and coupled with their return value. Then, if any of these methods were found, image data information is kept and eventually extracted at a later step as visual proof to validate the tool's output.

Color Constraint

This check is dependent on the "image extraction" parameter. Once established that a canvas image is being extracted, the tool verifies that the image contains at least two colors and if not, the image is ignored. This exclusion can result in the whole script being discarded during the "image extraction" check.

Size Constraint

This check is dependent on the "image extraction" parameter. Once established that a canvas image is being extracted, the tool verifies that the image is at least 16×16 pixels and if not, the image is ignored. This exclusion can result in the whole script being discarded during the "image extraction" check.

No Lossy Compression

This check is dependent on the "image extraction" parameter. Once established that a canvas image is being extracted, the tool verifies that the image is not in a lossy compression format, eventually excluding it. This exclusion can result in the whole script being discarded during "image extraction" check.

Text Length Constraint

This check is implemented in two steps: first, for each execution of matching URLs script, captured methods are filtered and coupled with their calling arguments. Then, arguments are evaluated for their length based on two parameters.

The first parameter is called "`text_length_dcfp`" and allows to customise the text length's lower bound for which we consider an indication of *Picasso*'s fingerprinting. As mentioned in the definition of the feature, this value is set to 5 by default, but we allow to tweak it. The check is then implemented by giving a 100% chance of being *Picasso* to text which length is up to this parameter's value, and a 50% chance to text up to two times the parameter's value. The remaining cases are ignored (0% chance).

The last parameter is a relaxation of the evaluation algorithm described above. This parameter is called "`text_length_dcfp_only`" and forces to give a 100% chance to very short text[2] and ignores any other text, if present.

All percentage values are then averaged and the result will range from 0 to 1, with a higher value implying a high chance of the used text to be a sign of *Picasso*'s implementation.

C.2 Other Tool Parameters

As the tool grew larger and more complex, we felt the need to not constrain the analysis by specific design decisions. Therefore, we envisioned the possibility to let the user decide and configure these aspects as well. Here we present the most relevant configuration options. For a list of all parameters, see Appendix D.

Aggregation Capabilities

To know if two scripts can be compared, we expect their URLs to be the same. As this is not always the case, we anticipated this possibility by defining two separate *aggregation strategies*. The best way to be sure of the identity between two scripts would be to compare each script's content. We refer to this approach as "aggregation by hash" and was implemented by hashing the script's content (already offered by OpenWPM, see Subsection 4.1.2) to then compare the scripts' hashes.

We also envisioned the possibility of comparing script URLs without their parameters. We refer to this approach as "fuzzy aggregation" and is a less precise and more relaxed aggregation strategy. It was implemented by comparing each URL up to the "?" character, if present. To make this approach more consistent, the truncated URL is hashed and an "`S-`" is pre-pended to distinguish this key from the "aggregation by hash" one. Table C.1 shows a practical example of these aggregation strategies.

These *aggregation strategies* were developed as independent parameters, with the "aggregation by hash" having a higher execution priority (3.3) and were relevant for the selection of an appropriate *analysis strategy* (E.3).

[2]Text length up to the defined "`text_length_dcfp`" parameter value. See Appendix D.

a)	https://path/to/script.js?param1=abc&param2=123
b)	a1b2c3d4e5f6a7b8
c1)	https://path/to/script.js
c2)	S-a0b9c8d7e6f5a4b3

Table C.1: Aggregation strategies:
a) plain script URL - no aggregation strategy
b) script's content hash - "aggregation by hash" strategy
c1) script URL truncated at character "?" - "fuzzy aggregation" strategy
c2) hash of truncated script URL - "fuzzy aggregation" strategy

Detection Threshold

The analysis output produces a list of URLs and their relative percentage of being an indicator of *Picasso*. This parameter simply allows the configuration of the threshold for considering them as such and the final output will show only those scripts that hit at least that threshold.

Appendix D

Analysis Tool Parameters

Parameter name	Description
`check_variable_primitives`	Variability Of Primitives in C.1
`check_variable_args_custom` `check_variable_args_jaccard`	Variability Of Arguments in C.1
`check_getImageData` `check_toDataURL`	Image Extraction in C.1
`filter_color_constraint`	Color Constraint in C.1
`filter_size_constraint`	Size Constraint in C.1
`filter_no_lossy_compression`	No Lossy Compression in C.1
`check_text_length`	Text Length Constraint C.1
`text_length_min_compare`	See Appendix D
`text_length_dcfp`	Text Length Constraint in C.1
`text_length_dcfp_only`	Text Length Constraint in C.1

Table D.1: List of analysis tool parameters for *Picasso*'s distinguishing features

`text_length_min_compare`

It defines the minimum number of same URLs script needed for comparison. As a text written to canvas could be evaluated on its own, we allowed to tweak the minimum number of needed comparable scripts. By default, this value is set to 2.

`analyze_within_top_url`

As the main goal of this research is to detect implementations of *Picasso* around the web, the way script URLs will be compared is crucial. Intuitively, one would look at scripts called when navigating a specific web page, but what if the majority of repeated crawls for a given website failed? Although the final goal is to list top-level URLs that host a script that implements *Picasso*, that same script could be on other web pages with not enough reliable data. We imagined the possibility of improving detection results by allowing our tool to be able to perform an analysis either by comparing scripts' execution within a top-level domain or by considering all script

Parameter name	**Description**
`analyze_within_top_url`	See Appendix D
`agg_by_hash` `agg_fuzzy`	Aggregation Capabilities in C.2
`filter_exclude_incomplete`	See Appendix D
`max_url_file`	See Appendix D
`use_json`	See Appendix D
`dcfp_threshold`	Detection Threshold in C.2

`prepare`
`analyse`
`output`
`gen_all`
	Structure of the tool in 3.3

`exclusion_file`	Generates an exclusion file. It includes canvas images generated in step "output". It is used in step "gen_all" to skip already generated canvas images
`backup_prev_log`	Backups eventual previous analysis output. At each run, all previous temporary and output data is deleted.

Table D.2: List of other analysis tool parameters

URLs across the whole dataset. We will refer to the selection of this parameter as *navigation strategy* in Subsection E.2.

`filter_exclude_incomplete`

This parameter is set to "true" by default and excludes those single website crawls that were marked as "incomplete" by OpenWPM. An incomplete set of data would add a lot of noise, especially with our approach, which is based on identifying the differences in scripts between different crawls.

`max_url_file`

This parameter defines the maximum number of URLs allowed in a temporary file. Temporary files generated during analysis contain data organised by URLs. To avoid these files from growing too much, custom logic is in place to control how many URLs per file are allowed.

`use_json`

OpenWPM persists its crawled data to an SQLite database instance, but for performance reasons, data is extracted from the database and converted to JSON[1] format before analysis. Still, we allowed the analysis tool to work with both formats through this parameter.

[1]https://www.json.org/json-en.html

Appendix E

Definition and Selection of an *Analysis Strategy*

In this Appendix we present how we investigated data and compared different approaches to identify the best detection strategies. For this purpose, we decided to use the crawl data of the top 20K URLs crawled without giving consent for data processing or mobile emulation as a testing set. The threshold parameter introduced in Appendix C.2 was set to `0.5`. Based on this parameter, the analysis tool will detect any script that has more than 50% probability of being an implementation of *Picasso*. Although tweaking this value will definitely reduce the number of false positives, we purposely left it at `0.5` to avoid missing some true positives. Then, the selection process was carried out through various steps and with the introduction of some new concepts:

- We defined as *analysis approach* the combination of investigated distinguishing features;
- We defined as *navigation strategy* if the analysis will look at JavaScript calls within a top-level URL or across the entire dataset (See `analyze_within_top_url` in Appendix D);
- We defined as *aggregation strategy* the selection of a specific aggregation option (C.2).

The different combinations of these three concepts were defined as *analysis strategies*. Collected data was evaluated with different *analysis strategies* and their results in detecting implementations of *Picasso* are reported in Chapter 5.

E.1 Define Analysis Approaches

By reviewing the canvas images extracted during the *generate canvas* step (3.3), we noticed the presence of some interesting canvas images. First, we discovered the usage of *Picasso*-like canvas images on many scripts, but most of them generated the same image at each execution. Our analysis tool was then unable to detect these, as the relative scripts had a very constant usage of our defined primitives. Therefore, we could not define an additional strategy to detect them. Identified *Picasso*-like canvas images can be reviewed in Appendix F.4 and we discuss their distribution in Section 5.5.

We also identified two other canvas images which relied on some sort of randomness. The first one was a 16×16 pixel canvas image with a random, up to three-digit number in scripts that generated canvas known to be used in fingerprinting (`a)` in Fig. E.1). The second one was a seemingly randomly composed canvas image, which incorporated different shapes, like circles and stars. Also this "shapes" canvas image was found in scripts that generated other known canvas used in fingerprinting (`b)` Fig. E.1). These two canvas images were extracted through `toDataURL()` and `getImageData()` respectively.

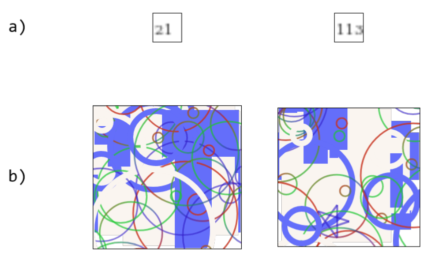

Figure E.1: Examples of "numbered" (a) and "shapes" (b) canvas images

The first analysis approach was labelled. In this approach we look at all the defined *Picasso* features altogether. We therefore defined three approaches with increased relaxation of included features.

"*Approach 1*" considers all the defined *Picasso* features altogether. "*Approach 2*" and "*Approach 3*" are designed to test the ability of the analysis tool to also detect scripts generating "numbered" and "shapes" canvas images. Although targeted at specific canvas images, relaxing the *analysis approaches* could help detect unexpected implementation of *Picasso*. Therefore, *Approach 2* and *Approach 3* should not be seen as an overfitting detection approach targeting a specific canvas image, as we are actually reducing the number of features to look at.

In *Approach 2*, the randomness of numbers and the position of the shapes could be detected through variations in arguments used by the defined canvas primitives. To give more relevance to the "*variability of arguments*", we decided to deactivate the "*variability of primitives*" parameter (3.1.3). Additionally, knowing that few text characters written on a canvas is a valuable feature on which canvas fingerprinting does not rely (it generally uses a pangram), we activated the relaxed text length search for "*text length constraint*" (C.1).

Approach 3 was defined as a further relaxation of *Approach 2*. Here the "*variability of arguments*" check was deactivated, leaving the remaining configurations unchanged. In this way, the only remaining feature to consider was the relaxed text length search for "*text length constraint*" (C.1).

Table E.1 displays the defined analysis approaches and their parameters. Please note that the constraints on image extractions are always active across the three approaches as they help reduce the number of false positives. For each analysis approach, only one *analysis strategy* was selected. For a detailed description of how *Analysis Strategy 1*, *2* and *3* were selected, please refer to Appendix E.3.

Parameter	**Approach 1**	**Approach 2**	**Approach 3**
variability of primitives	✓	-	-
variability of arguments C+J †	✓	✓	-
text length constraint	✓	✓	✓
relaxed text length	-	✓	✓
color constraint	✓	✓	✓
size constraint	✓	✓	✓
no lossy compression	✓	✓	✓

Table E.1: List of parameters active for each approach
†: C+J means custom + jaccard.
Note that *color constraint, size constraint* and *no lossy compression* remain always active as they help in reducing the number of false positives.

E.2 Aggregation and Navigation Strategies

As anticipated in Section 3.3, the analysis tool offers different aggregation possibilities Currently the analysis tool offers two aggregation possibilities to determine if two scripts are an identity: “aggregation by hash”, where the content hash of scripts is compared, and “fuzzy aggregation”, where only a portion of the script URLs is used for the comparison.

In addition, as presented in Appendix D, the analysis can be performed by looking for the same script either between scripts called during the navigation of a specific top-level URL, or by comparing all script URLs across the entire dataset. This possibility was identified in this Section as *navigation strategy*. To help us identify the best *analysis strategy*, we defined the matrix presented in Table E.2. Each combination of aggregation and navigation strategies has its own label, which will be tested with all defined *analysis approaches*.

	Navigation Strategy	
	Within URL	**Across Dataset**
No Agg.	Within-NoAgg	Across-NoAgg
Agg. by Hash	Within-HashAgg	Across-HashAgg
Fuzzy Agg.	Within-FuzzyAgg	Across-FuzzyAgg

Table E.2: Combination of possible Aggregation and Navigation strategies.

E.3 Selecting an Analysis Strategy

As introduced in the previous Subsection, to determine an *analysis strategy*, all the Tests presented in Table E.2 had to be performed for each *analysis approach* E.1, for a total of 36 possible combinations to test. Table E.3 reports these results.

It is interesting to note how the “aggregation by hash” performs almost the same as with “no aggregation” strategy. The more the approach gets relaxed, the more “aggregation by hash” performs better in identifying true positives with fewer false positives. The “fuzzy aggregation” on the other hand performs the opposite, giving best results with the approaches that perform a search for most or all distinguishing features. Additionally, the use of the *navigation strategy* “across dataset” did not seem to bring notable or valuable improvements in the results. However, its usage was able to detect also those scripts that happened to have data for one crawl only, with nothing to compare with.

Based on shown results, *Approach 1* seemed to be able to detect implementations of *Picasso* in its “toDataURL” *variant* at the cost of a few false positives, whereas the “getImageData” *variant* was unable to detect anything interesting. *Approach 2* in its “toDataURL” *variant* was able to detect “numbered” canvas images, as expected, as well as detecting implementations of *Picasso*. This shows that the relaxation of the approach allows to detect “more”, rather than pinpoint a specific canvas images. As anticipated in Section E.1, this was expected and is our countermeasure against over-fitting. The “getImageData” *variant* was able to detect only 1 “shapes” canvas image out of the more expected, and the rest of the output was still not interesting. Coming to *Approach 3*, the considerable amount of “detected” URLs in the “toDataURL” variant resulted in a spike of false positives, making this approach hard to analyse and use. On the other hand, this time, the “getImageData” *variant* returned the expected “shapes” canvas images with an acceptable number of false positives.

To pick the most fitting research approach, we computed their F1-score[1]. We selected *Approach 1* and *Approach 2* in their “toDataURL” *variant* with “Within-FuzzyAgg” as the preferred *Navigation* and *Aggregation* strategy combination as interesting *analysis strategies*. Regarding *Approach 3* in its “getImageData” *variant* and according to the F1-score, the “Across” *Navigation* strategy yields the best results. However, we decided to select the same *Navigation* and *Aggregation* strategy of *Approach 1* and *Approach 2* to allow for more direct and comparable performance results. Therefore, *Approach 3* in its “getImageData” *variant* with “Within-FuzzyAgg” *Navigation* and *Aggregation* strategy combination was selected as our third *analysis strategy*. These selections are highlighted in Table E.1 and will be referred to as *Analysis Strategy 1*, *2* and *3* respectively.

[1]https://en.wikipedia.org/wiki/F-score

Approach	Aggregation-Navigation	Detected	FP	F1-score
1-toDataURL	Within-NoAgg	4	2	25%
	Within-HashAgg	4	2	25%
	Within-FuzzyAgg	14	2	92%
	Across-NoAgg	5	3	50%
	Across-HashAgg	5	3	50%
	Across-FuzzyAgg	6	3	67%
1-getImageData	Within-NoAgg	1	1	0%
	Within-HashAgg	1	1	0%
	Within-FuzzyAgg	1	1	0%
	Across-NoAgg	1	1	0%
	Across-HashAgg	1	1	0%
	Across-FuzzyAgg	1	1	0%
2-toDataURL	Within-NoAgg	301	2	100%
	Within-HashAgg	305	2	99%
	Within-FuzzyAgg	311	2	100%
	Across-NoAgg	298	3	99%
	Across-HashAgg	10	3	82%
	Across-FuzzyAgg	299	3	99%
2-getImageData	Within-NoAgg	2	1	29%
	Within-HashAgg	2	1	29%
	Within-FuzzyAgg	2	1	29%
	Across-NoAgg	1	1	0%
	Across-HashAgg	2	1	29%
	Across-FuzzyAgg	2	1	29%
3-toDataURL	Within-NoAgg	1237	936	39%
	Within-HashAgg	800	491	56%
	Within-FuzzyAgg	1574	1265	33%
	Across-NoAgg	1214	919	39%
	Across-HashAgg	586	277	69%
	Across-FuzzyAgg	1535	1239	32%
3-getImageData	Within-NoAgg	14	9	53%
	Within-HashAgg	14	9	53%
	Within-FuzzyAgg	13	8	56%
	Across-NoAgg	11	6	63%
	Across-HashAgg	11	6	63%
	Across-FuzzyAgg	10	5	67%

Table E.3: Results for all defined Approaches with possible Aggregation and Navigation strategies and their relative F1-score.

Appendix F

Generated Canvas Images

F.1 Known Fingerprinting Canvas Images in False Positives

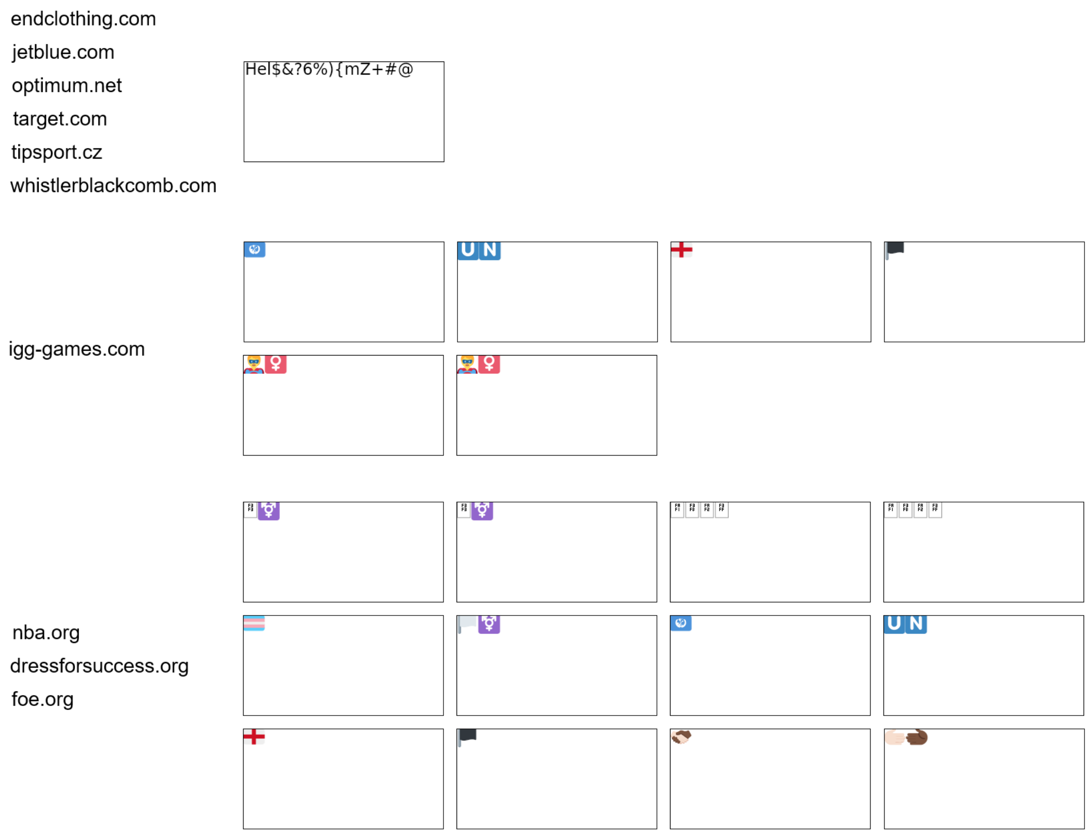

Figure F.1: Known Fingerprinting Canvas Images in False Positives.

F.2 Full Round of Canvas Images Generated by the Detected *Picasso*'s Script

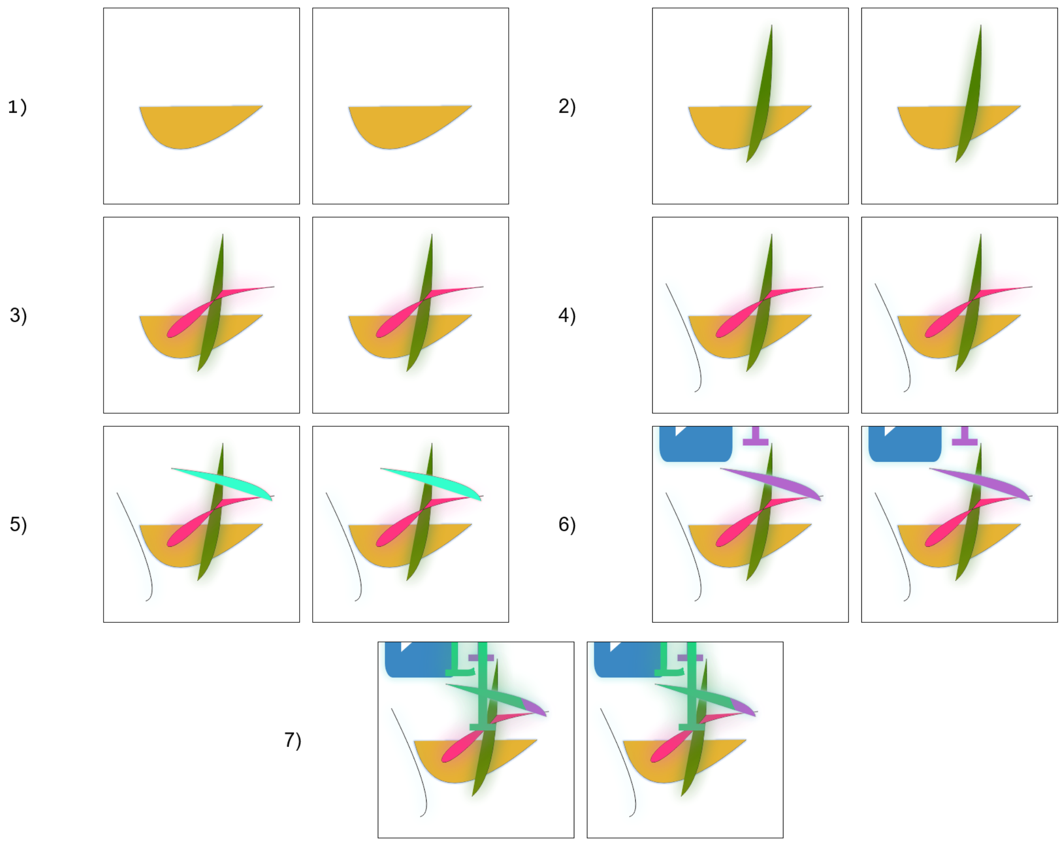

Figure F.2: A full round of generated canvas images for the detected *Picasso* implementation. It is clearly visible the fact that the underlying technique is performing rounds, adding each time a different primitive with variating position, color and gradient. Additionally, at each round, the canvas image is extracted twice, probably to strengthen the implementation against possible attacks.

F.3 All Canvas Images Rounds Identified and Performed by the Detected *Picasso*'s Script

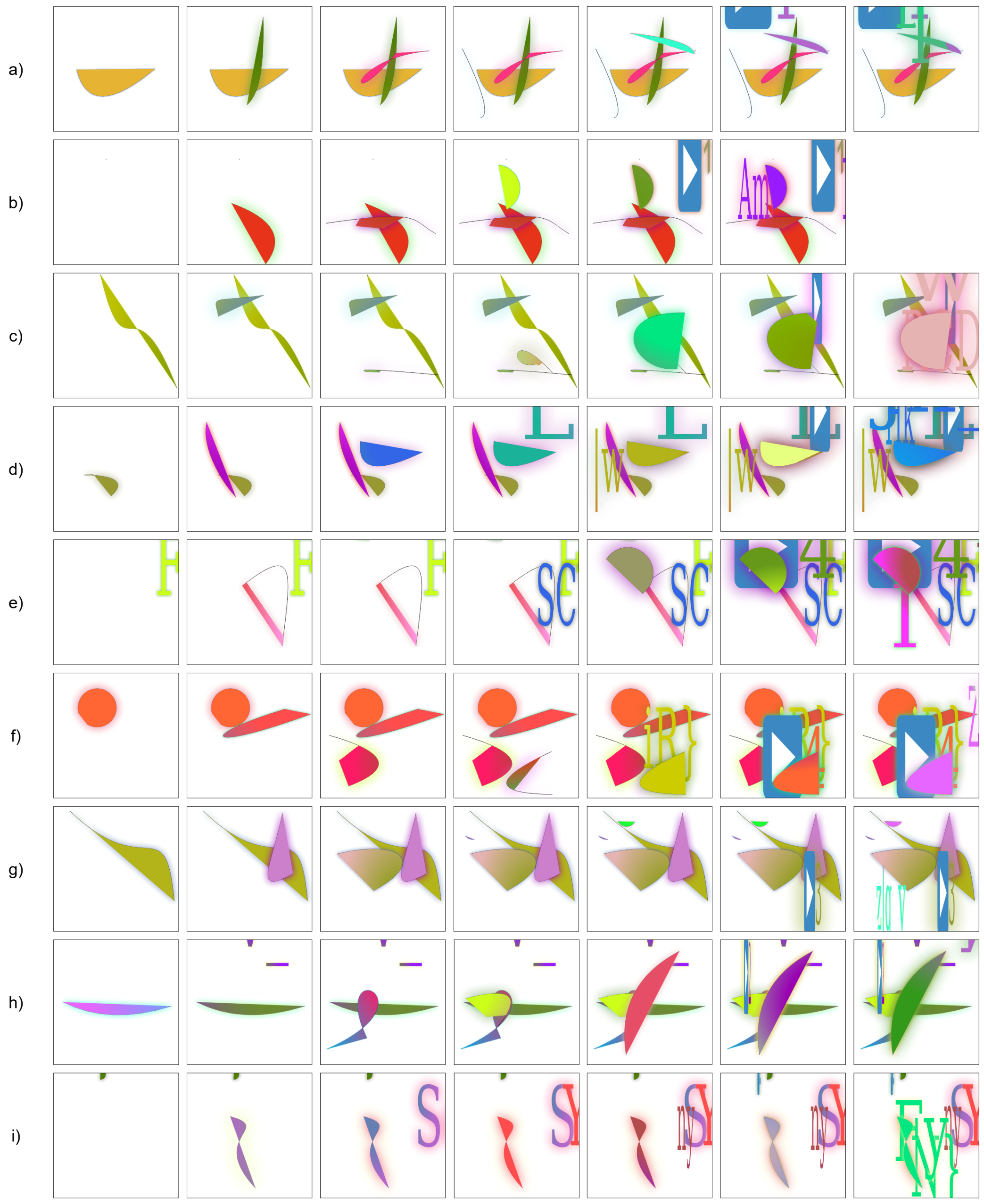

Figure F.3: Full set of generated canvas images for the detected *Picasso* implementation. These were collected from all true positives across all crawls.

F.4 *Picasso*-like Canvas Images

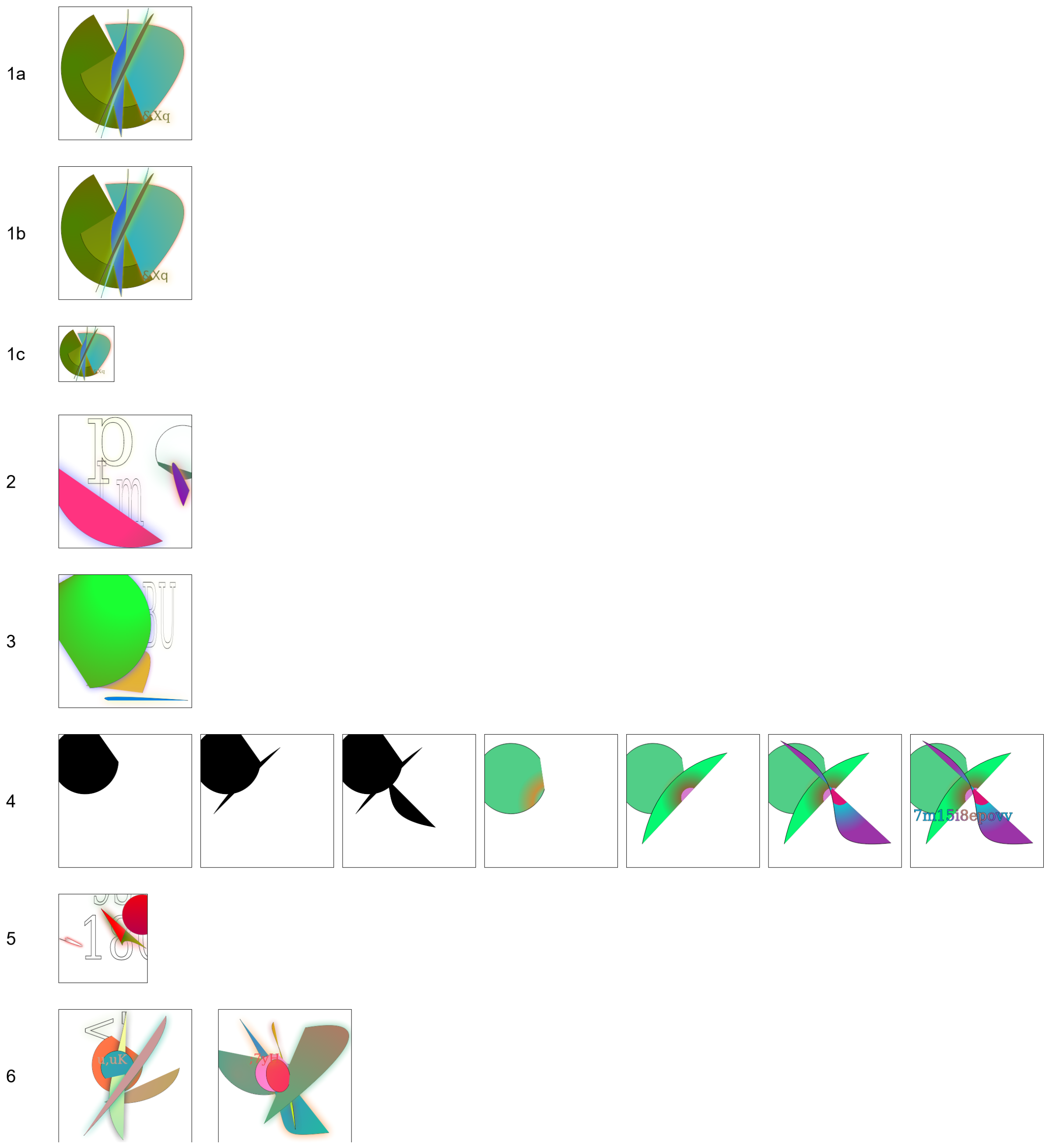

Figure F.4: Full set of identified *Picasso*-like canvas images.

F.5 "Shapes" Canvas Images

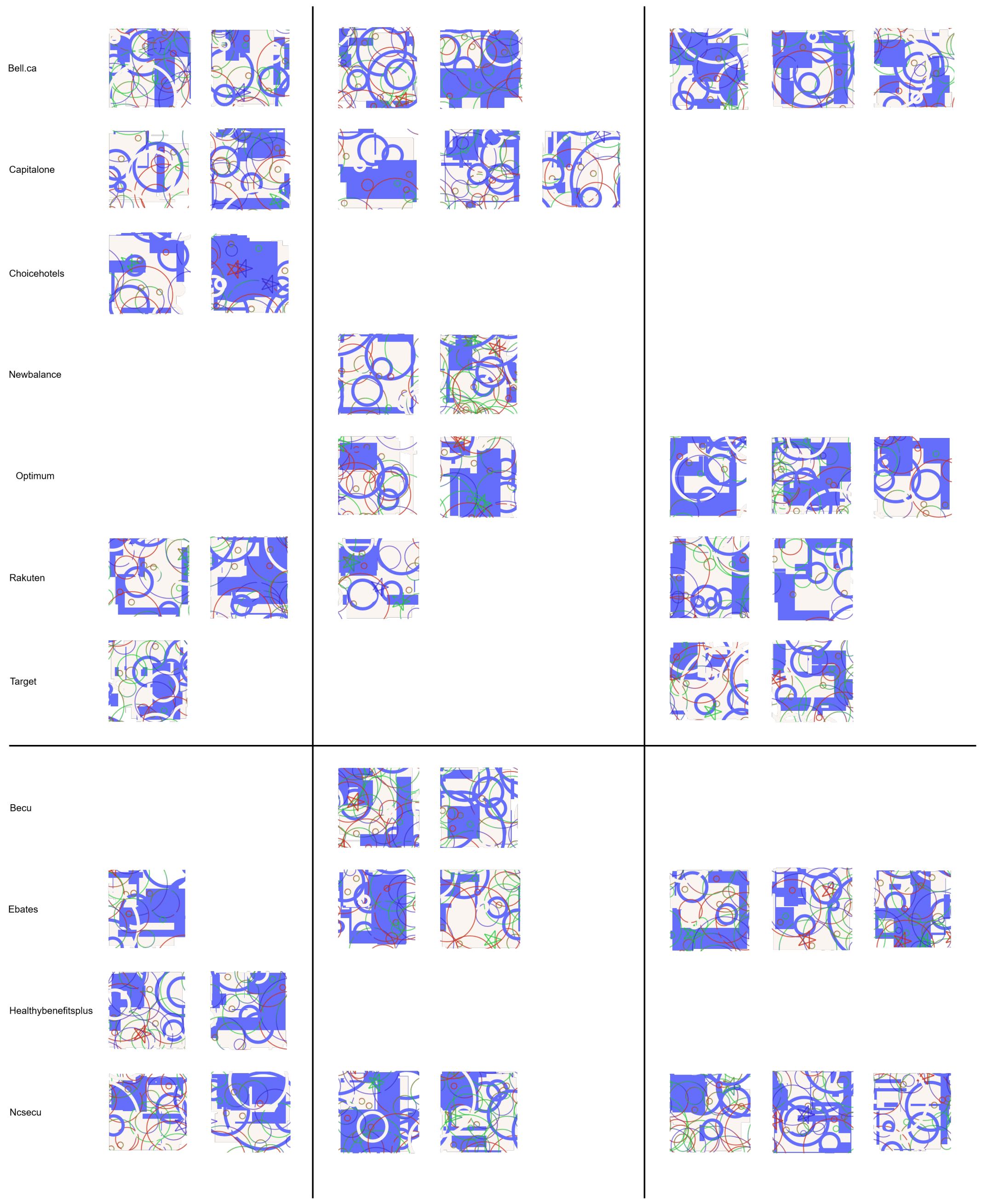

Figure F.5: Full set of identified "shapes" canvas images across all crawls.