Picasso: Lightweight Device Class Fingerprinting for Web Clients

Elie Bursztein Artem Malyshev Tadek Pietraszek Kurt Thomas
Google, Inc
{elieb, artemm, tadek, kurtthomas}@google.com

ABSTRACT

In this work we present Picasso: a lightweight device class fingerprinting protocol that allows a server to verify the software and hardware stack of a mobile or desktop client. As an example, Picasso can distinguish between traffic sent by an authentic iPhone running Safari on iOS from an emulator or desktop client spoofing the same configuration. Our fingerprinting scheme builds on unpredictable yet stable noise introduced by a client's browser, operating system, and graphical stack when rendering HTML5 canvases. Our algorithm is resistant to replay and includes a hardware-bound proof of work that forces a client to expend a configurable amount of CPU and memory to solve challenges. We demonstrate that Picasso can distinguish 52 million Android, iOS, Windows, and OSX clients running a diversity of browsers with 100% accuracy. We discuss applications of Picasso in abuse fighting, including protecting the Play Store or other mobile app marketplaces from inorganic interactions; or identifying login attempts to user accounts from previously unseen device classes.

1. INTRODUCTION

Mobile app stores and web services routinely face automated abuse from attackers who masquerade as legitimate clients in order to flood APIs and servers with fake installs, fake reviews, spam comments, scraping requests, and other synthetic interactions. More than a mere nuisance, these actions result in billions of inflated views on YouTube [13], millions of fake accounts on Facebook and Twitter [8, 9], and feed into other threats such as large-scale compromise that victimize a service's user base [14,26]. This problem is exacerbated in part by the indistinguishability of authentic clients from scripts and emulated environments that trivially spoof the personas (*e.g.*, User-Agent) of organic traffic.

We address the challenge of client spoofing by introducing Picasso: a lightweight *device class* fingerprinting protocol that allows a server to accurately determine the browser, operating system, and graphical stack of a web browsing client. As an example, Picasso allows a server to distinguish between traffic sent by an authentic iPhone running Safari on iOS from an emulator or desktop spoofing the same configuration. Our detection granularity is limited to device classes; unlike device fingerprinting, we cannot uniquely distinguish two clients operating the same browser and hardware. Nevertheless, Picasso has a wide range of applications in abuse fighting that includes verifying the authenticity of mobile clients that interact with the Play Store and other mobile app markets or identifying login attempts to user accounts from foreign, previously unseen device classes.

We design our fingerprinting scheme as a challenge-response protocol that builds on unpredictable yet stable noise introduced by a client's software and hardware. Our algorithm is resistant to replay and includes a hardware-bound proof of work that forces a client to expend a configurable amount of CPU and memory to produce a valid response that cannot be offloaded to more powerful devices of a different type or even emulators of the same device class. Finally, we design Picasso to support a real world deployment that encompasses a wide class of devices and bandwidth constrained environments.

In practice, Picasso relies on multiple rounds of drawing HTML5 canvas graphical primitives to surface divergent implementation behaviors across device classes. We expand on the work of Mowery *et al.* and Laperdrix *et al.* that found canvas rendering differences produce enough entropy to distinguish individual devices [16, 20]. However, unlike this previous work, we assume an adversarial model where attackers actively try to spoof responses and have perfect knowledge of our system's algorithm. Furthermore, the success of our scheme hinges on identifying HTML5 canvas operations that *minimize* entropy between devices of the same class, but maximize entropy across distinct device classes—antithetical requirements compared to prior device fingerprinting work. To accomplish this, we experimentally validate that a select collection of graphical primitives (*e.g.*, writing text, drawing Bézier curves) can produce both *unique* outputs for device classes that are *stable* across all devices within the class when tested on a diverse set of 250,000 web clients.

We illustrate Picasso's real-world effectiveness by serving Picasso challenges to a sample of 52 million clients connecting to Google. We distinguish between Android, iOS, Windows, and Mac clients running a diversity of browsers with 100% accuracy. In the process, we identify a brute force login attack that spoofed a variety of User-Agents that Picasso uniquely identified as PhantomJS running on cloud machinery. Furthermore, we show that Picasso accurately distinguishes between emulated hardware and real hardware otherwise operating identical software stacks.

2. DESIGN GOALS

The goal of our work is to design a fingerprinting protocol that enables web servers to accurately identify a client's *device class*. We define a device class as a unique collection of {browser, operat-

SPSM'16 October 24-24 2016, Vienna, Austria
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4564-4/16/10.
DOI: *http://dx.doi.org/10.1145/2994459.2994467*

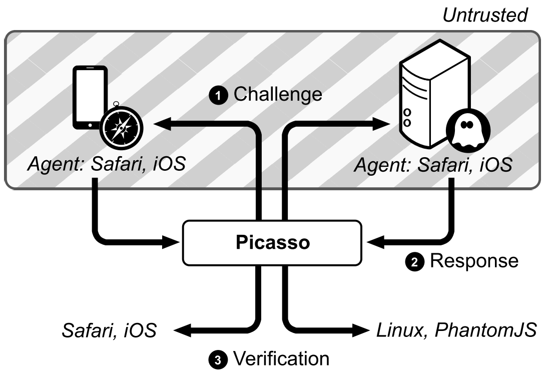


Figure 1: High-level overview of device class fingerprinting. We first send a challenge to an untrusted client asking it to prove its purported device class (*e.g.*, Safari on iOS) (❶). The client returns a response (❷) which we verify to distinguish between an authentic device and a system spoofing its User-Agent or any other aspects of its browser or operating system (❸).

ing system, graphics hardware}. We sketch a high-level overview of our scheme in Figure 1. Consider two systems: an iPhone running Safari on iOS and a cloud system running PhantomJS on Linux. Both present a User-Agent string for Safari on iOS in addition to any number of spoofable browser configurations. A fingerprinting server issues a challenge to both untrusted systems asking them to prove their purported device class (❶). The clients both return a response (❷) which we verify to distinguish between an authentic device and a spoofed system (❸). We decompose this challenge into its functional requirements as well as practical requirements necessary to support a large-scale live deployment scenario.

2.1 Functional Requirements

Device class fingerprinting: The primary goal of Picasso is to provide a challenge that accurately identifies the hardware and software stack of a web client. This requirement is weaker than per-device fingerprinting [16,20] that uniquely identifies individual devices. This proof should be unpredictable even if an adversary has perfect knowledge of our approach or access to previous challenge-response pairs.

Hardware-bound proof of work: Picasso's secondary objective is to bind challenges with a configurable proof of work. This guarantees that a device expends a certain amount of resources (*e.g.*, CPU, GPU, RAM) in order to solve a challenge. Picasso incorporates these resources as part of its challenge which effectively prohibits adversaries from offloading expensive computations to more powerful devices of a different class (*e.g.*, offload phone computation to a desktop).

2.2 Operational Requirements

Cross-platform: Our approach must be applicable to a wide class of devices and operating systems that include desktop clients, phones, and even TV and game consoles. As part of this, we must support both browsers and native applications.

No hardware modification: Picasso cannot require hardware modifications to legacy devices. As part of this requirement, our system cannot assume access to a TPM or custom hardware PUF [24] to guarantee tamper-free execution or access to a unique identifier.

Algorithm 1 Picasso One-Way Function

```
procedure CANVASHASH($s, N, A$)
    $initRand(s)$
    $canvas \leftarrow initCanvas()$
    $response \leftarrow \emptyset$
    for $i \in (0, N)$ do
        $primitive \leftarrow selectRandomPrimitive()$
        $color \leftarrow selectRandomColor()$
        $shadow \leftarrow selectRandomShadow()$
        $canvas.drawPrimitive(primitive, color, shadow)$
        $response \leftarrow hash(response, hash(canvas))$
    end for
    return $response$
end procedure
```

Low latency verification: While challenges produced by Picasso can be arbitrarily resource intensive, responses must be easily verified within a few milliseconds. This requirement does not preclude the possibility of offline pre-processing.

Bandwidth constrained: Any challenge and response must be sufficiently small to minimize network traffic produced during fingerprinting. We design for network speeds constrained to that of 2G devices and analog modems.

2.3 Threat Model

We assume a threat model where an attacker has complete control over the device under analysis and all network traffic that reaches the verifier. This includes complete access to the code that comprises a challenge and perfect knowledge of the subsystems queried by our computation. Finally, an attacker can have access to a computationally bound number of prior valid challenge-response pairs in order to attempt forging a device class fingerprint, the restrictions of which we explore in more detail in Section 3. Our threat model sets us apart from previous research into user and device fingerprinting that frequently rely on a non-adversarial environment where clients accurately report the results of tests or are unaware of what environment variables are under observation.

3. BUILDING PICASSO

We design Picasso to incorporate artifacts produced by a system's browser, operating system, and graphical rendering stack to produce a uniquely keyed one-way function that is equivalent in concept to a physically unclonable function. In particular, we rely on HTML canvas element rendering using JavaScript that was previously shown to produce divergent outputs based on the underlying software and system environment [16, 20]. We elect for JavaScript and canvas elements as both are widely supported by all browsers and modern devices. We can extend this support to native applications via the canvas APIs provided by Android and iOS. This satisfies our operational requirement of being both cross-platform and requiring no new hardware.

3.1 Generating Challenges

We sketch how we convert HTML canvas drawings into a one-way function with a configurably difficult proof of work in Algorithm 1. To start, a server sends a copy of our algorithm $canvasHash$ to the client along with a random seed s and number of rounds N. The client then initializes a pseudorandom number generator packaged with our canvasHash algorithm with the seed and creates a blank, hidden canvas of size A.[1] At each round the client elects a random graphical primitive such as drawing text, curves, Béziers, or gradients. We randomize the values to these elements each round to include color gradient, shadow, coordinates, shape, and other properties as determined by our seed s. Finally, we draw the element to the canvas before taking a snapshot of the canvas and iteratively hashing it with the output of the previous round. In our scheme, the number of rounds N and the screen size of the canvas A enable the server to force a client to expend a configurable amount of memory and computational resources. We use the seed s to prevent replay attacks.

An optimal $canvasHash$ implementation $F_{(s,N,A)}$ has two requirements. First, the underlying set of graphical primitives must produce a *unique* response on all devices of the same class when provided identical inputs that is never produced by any other device. Secondly, this output must be *stable* regardless of other operations occurring on the device or slight rendering deviations within a device class. This contrasts with CPU timing red pills which are susceptible to processing and network bottlenecks that inject noise into measurements [12, 15, 23]. We formally define both of these requirements using Figure 2 as a guide.

Uniqueness: Consider two distinct device classes D and D' that each represent a unique collection of clients with identical browsers, operating systems (*e.g.*, Chrome on Windows, Firefox on Windows), and hardware. We define the pairwise uniqueness between D and D' as the total number of devices $d \in D \cup D'$ that map via Picasso challenges into a response space $R_D \otimes R_{D'}$. In our hypothetical scenario presented in Figure 2 only four of six devices produce Picasso responses that are exclusive to their device class yielding a uniqueness of 66.6%. An optimal graphical primitive should produce 100% uniqueness for all challenges (s, N, A) for all possible pairs of device classes to prevent ever misclassifying a device's class.

Stability: Within a single device class D we measure stability as $1/|R_D|$. This captures the number of potential valid responses per device class. In our example, both D and D' are 50% stable. An optimal challenge should produce 100% stable signals for all possible (s, N, A). In practice, unlike uniqueness, we can afford a certain amount of instability. Subtle differences in GPUs or hardware manufacturing, if triggered by our graphical primitives, will make 100% stable signatures for every Windows desktop system using Firefox unlikely. To account for this Picasso can learn the *set* of responses that define a device class rather than an exact response. However, increasing instability runs the risk of producing collisions between device classes that degrades uniqueness. Equally problematic, it is untenable to store thousands of responses for verification as we outline shortly.

We experimentally validate which set of graphical primitives achieve both of these properties in Section 4 and confirm their effectiveness in practice in a live deployment scenario in Section 5.

3.2 Verifying Responses

The intended unpredictable output of our challenge poses a unique problem for verification: like attackers, we cannot produce the correct answer without access to the client's system. In the absence of a built-in trap door we rely instead on a computational constraint: we assume that the number of devices accessible to an online service (*e.g.*, its user base) is larger than the pool of devices

[1]In practice, canvases must be larger than 100 pixels to generate sufficient divergent properties. As we can hide canvas elements, this size has no discernible impact to clients other than increasing computation costs.

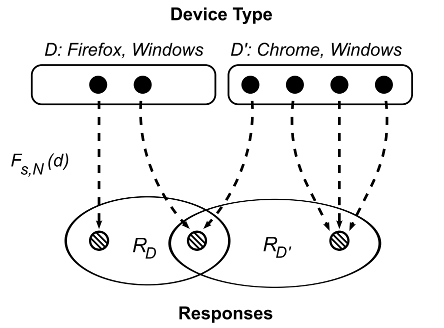


Figure 2: Surjection produced by $F_{(s,N,A)}$ between device classes D and the resulting space of canvas hash responses R.

controlled by an attacker. This assumption is unrealistic for personal websites but entirely feasible for large web services such as Google, Facebook, or Yahoo that are lucrative targets for automated abuse due to serving billions of users.

During a bootstrap phase our server constructs a dictionary of challenge-response pairs and their mapping to device classes. We propose three approaches to generate this initial dictionary: (1) identifying large clusters of clients (or trusted logged in users) that all generate the same Picasso response and report the same User-Agent string;[2] (2) querying a set of trusted users (*e.g.*, colleagues, employees) who own a diversity of devices; or in the worse case (3) purchasing the devices of interest ourselves and configuring their software stack to produce a challenge-response pair for every software combination. Regardless of the approach, this pain point occurs only once.

After bootstrapping, the server can generate an infinite number of challenge-response pairs by sending clients one challenge with a known response and an arbitrary number of challenges with unknown solutions. If clients provide an accurate response to the known challenge we consider all other responses valid. An incorrect response invalidates all other challenges. Our approach is similar to how reCAPTCHA serves unrecognized text from digitized books alongside known mutated text [11]. Like reCAPTCHA, we must resist pollution attacks where an adversary with one or more valid Picasso responses attempts to submit bogus responses to unknown challenges. We use a thresholding strategy where a fraction of at least τ devices must report the same challenge-response pairing. We refer readers to the ESP game structure proposed by Von Ahn *et al.* for combating cheating in user-generated labels that we re-use [28].

The end result is a challenge-response pair database that is constantly updated to prevent an attacker from pre computing a sufficiently large number of results to replay. Verification of live traffic is a simple dictionary lookup which satisfies our low latency requirement. If we find a match, we return the client's device class. A miss indicates we have never encountered the device before, or more likely, the client supplied a fake response as part of a brute force or pollution attack.

3.3 Assumptions & Limitations

We explicitly outline the assumptions built into our design as well as limitations of device class fingerprinting in the face of widespread compromise. First, we assume that a device's graphical stack in conjunction with OS and browser disparities produces an unpredictable output in kind to a PUF. As part of this assump-

[2]One limitation of this approach is that User-Agents do not report hardware configurations. As such, we would be limited to fingerprinting a client's browser and operating system.

tion, we postulate that the complexity of input and output combinations is intractable for an attacker to learn and use to produce an equivalent graphical emulator as researchers have previously demonstrated for some implementations of PUFs [25]. Second, we assume that an attacker's computational power is strictly less than the online service deploying Picasso. This is necessary to prevent an attacker from building a dictionary of known challenge-response pairs at a faster rate than the server Picasso protects. While smaller services cannot satisfy this requirement, we envision a system similar to reCAPTCHA that provides Picasso as a service to third parties.

We note that Picasso cannot prevent an attacker from fowarding device fingerprint challenges to compromised unemulated mobile or desktop systems for solving. To impede the creation of solution farms, Picasso can increase the computation complexity of each solution while balancing resource usage acceptable for legitimate clients.

Device fingerprinting as an abuse fighting tool is only useful in the absence of wide-spread compromise. With millions of infected desktop systems participating in botnets, a sufficiently motivated attacker can use a victim's native browser to satisfy a device fingerprint challenge that indicates a real operating system and browser. Picasso still helps to protect against headless browsers like PhantomJS and HtmlUnit as well as emulators. Conversely, mobile malware has yet to reach the pervasive infection levels of desktop environments [18]. We argue there is still a significant value in detecting differences between emulated, desktop, and mobile environments that motivates deploying a system like Picasso. We demonstrate this fact later in Section 5.

4. EXPERIMENTAL VALIDATION

We evaluate the overall feasibility of Picasso and derive which HTML5 canvas operations are best suited for accurate device class. In particular, we consider four HTML5 canvas drawing operations as candidate graphical primitives for our canvas hash algorithm: *arc()*, *strokeText()*, *bezierCurveTo()*, *quadraticCurveTo()*. Our intuition is that each method introduces aliasing artifacts and potentially divergent logic in calculating the angles between midpoints or font shapes as recognized by prior work [2, 7, 20]. Each method has a unique list of parameters (*e.g.*, coordinates, length, angle, width). We automatically generate a value for each that is within the range of valid values. We further style every object with *createRadialGradient()*, *shadowBlur()*, and *shadowColor()* to introduce as much graphical entropy per object as possible. We evaluate these primitives in a limited controlled setting before validating our findings on a dataset of 272,198 devices.

4.1 Initial Proof of Concept

As an initial proof of concept, we executed a prototype of Picasso on single Macbook Pro running Mac OSX. We then compared the canvases produced by three different browsers when provided an identical Picasso challenge. Figure 3 shows the per-pixel rendering differences between a canvas produced by Chrome, Firefox, and Safari as highlighted in red. We observe substantial divergent behavior between each browser's HTML5 canvas implementation.

We repeated this process for 1,000 different challenges on canvases of size $200px \times 200px$ and averaged our results. Figure 4a summarizes the average distance $|r_1 - r_2| + |g_1 - g_2| + |b_1 - b_2|$ between the RGB values for each pixel produced by Chrome vs. Firefox and Chrome vs. Safari, broken down by graphical primitive. The majority of rendering entropy is fairly minor in terms of absolute difference. However, in aggregate, the noise introduced by each browser's rendering stack causes a median of 40–70% of

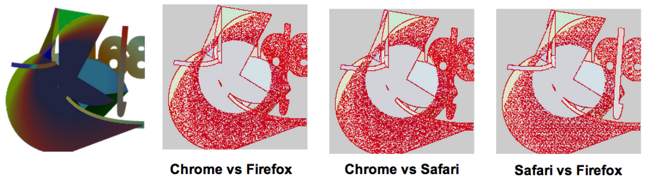
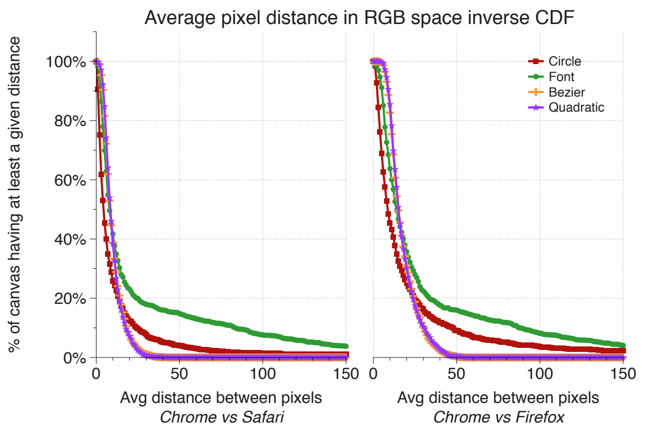
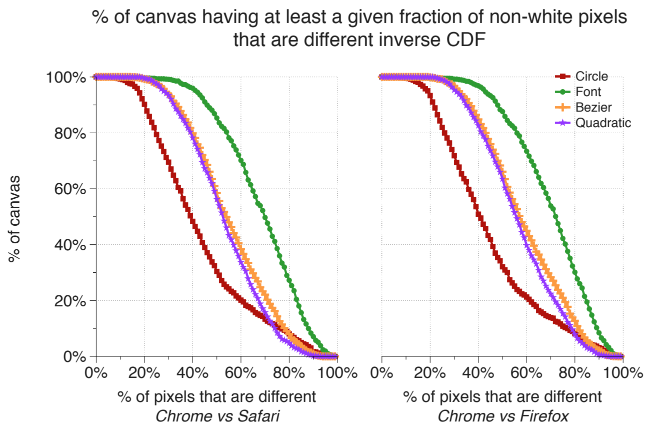


Figure 3: Visualization of the rendering differences between the same Picasso challenge for various browsers. We indicate in red the per-pixel difference between each browser pair.



(a) Average per-pixel RGB differences.



(b) Average per-canvas pixel differences.

Figure 4: Divergent canvas drawings produced by Chrome, Safari, and Firefox running on the same hardware and operating system.

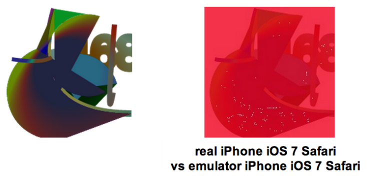


Figure 5: Visualization of the rendering differences between an emulated and real iOS device using the same software stack. We indicate in red the per-pixel Picasso solution differences.

Browser	**Sample Size**
Chrome 28	17,016
Chrome 29	184,634
Chrome 30	1,896
Chrome 31	1,107
iOS browser	35,644
Firefox 23	30,030
Safari 6	769
Other	1,082

Table 1: Breakdown of client browsers in our dataset.

Operating Systems	**Sample Size**
Windows Vista	7,231
Windows XP	60,535
Windows 7	138,416
Windows 8	22,542
OSX 10	6,483
iOS 6	31,655
iOS 7	3,989
Linux	719
Other	608

Table 2: Breakdown of client operating systems in our dataset.

all non-white pixels to differ as shown in Figure 4b. For an attacker to spoof a valid response without access to the device in question, she must learn all of the nuances of each rendering stack. Even a single pixel intensity or placement difference will cause an avalanche effect in our canvas hash algorithm that results in an invalid response.

Our device class detection extends to emulated devices versus real devices. As a proof of concept, we compare the visual output of an identical Picasso challenge solved by an emulated iOS device and the corresponding real iOS device, both running an identical software stack. Figure 5 shows the per-pixel rendering differences between the two devices which appear drastically different. These results highlight Picasso identifiers more than software variations; it extends to the graphical stack of devices. As such, an attacker cannot rely on off-the-shelf emulation to solve Picasso challenges in an attempt to masquerade as a real device.

4.2 Broader Experimental Validation

We prove that Picasso accurately generalizes to all device classes, not just our synthetic example, by evaluating our candidate primitives on a sample of inbound traffic to Google. As part of our experiment, we served each client a Picasso challenge in the form of a light-weight JavaScript payload consisting of 0.86KB. We randomized the challenge parameters as follows: we served each client a random seed s between $(0, 1000)$, number of rounds N between $(1, 3)$, and a fixed canvas size A of $200px \times 200px$. At each round we selected a pseudorandom primitive from the set $\{circle, font, bezier, quadratic\}$ as well as pseudorandom gradients, blurs, and colors. All parameter selection is uniquely tied to s so that all devices served a challenge (s, N, A) execute the same computation.

In total, we collected 272,178 Picasso challenge-response pairs in August, 2013.[3] We relied on each client's User-Agent string as a ground truth label of each device's browser and operating system. While we cannot guarantee that spoofing is absent from our measurement, we attempted to mitigate its potential by restricting our analysis to traffic from signed-in users at the web service under evaluation as well as using JavaScript to fetch the User-Agent rather than via user supplied headers. We note that Picasso provides more insight than a User-Agent: it also captures entropy induced by a system's underlying hardware. However, we have no way to query a client's physical system settings and thus cannot evaluate hardware-level uniqueness and stability.

We provide a breakdown of every client's browser and operating system in our dataset in Table 1 and Table 2 respectively. We note that for browsers we aggregate all iOS browsers including Safari, Chrome, and iCab as a single family as they all use the Apple WebView API. Our "other" category consists of clients with too few samples to be statistically significant for our study: Blackberry, Firefox on Android, Iron browser, and more.

Noticeably absent from our experiment is Internet Explorer. This is because older versions of the browser until version 9 do not support HTML5 canvases. Conversely, IE6–8 represented a substantial fraction of our traffic feed. In practice, we can likely extend our approach to these older browsers through Vector Markup Language (VML) or by supplying a third-party JavaScript implementation of the canvas element. For our initial validation we restrict ourselves to HTML5 compatible browsers.

4.3 Fingerprinting Performance

We use our dataset to measure two properties tied to each candidate graphical primitive to verify their effectiveness in device class fingerprint: uniqueness and stability as outlined previously in Section 3. We then measure the overall accuracy achieved by combinations of each primitive with respect to the number of canvas operations Picasso performs.

Uniqueness: We consider four types of pair-wise uniqueness: distinguishing OS families, OS versions, browser families, and browser versions. As performance will vary across challenges (s, N, A), we average the uniqueness results of all challenges. As Picasso randomizes the primitive used each round N, we restrict this part of our analysis to 92,490 challenges that consisted of only one round. Furthermore, we omit pair-wise evaluations where we lack more than 1,000 samples. As such, all reported uniqueness measures are within $\pm 3\%$ at 95% confidence.

Operating system vs. operating system: For our first calculation, we examine whether our four graphical primitives can identify a client's OS irrespective of the client's associated browser or hardware. Our results in Table 3a indicate that the graphical rendering stack provided by an OS produces unique Picasso results. This is true 100% of the time for font-specific text drawn by the *strokeText()* primitive. Surprisingly, we find that all other graphical primitives fail to distinguish between Mac OSX and Windows over 80% of the time.

OS version vs. OS version: When we apply the same pairwise measurement to clients with identical operating systems but different versions we observe a degradation in performance. We provide a detailed breakdown of results in Table 3b. We find that the graphical stack between iOS6 and iOS7 is completely different and yields 100% unique Picasso responses for three out of four of our primitives. In contrast, no primitive can produce a unique challenge that distinguishes Windows XP from 7 or 8. Our results indicate that 100% OS version granularity derived purely from graphical noise is likely intractable given our current set of primitives.

[3] As two years have passed since our initial experiment many of the browser versions we analyze are now obsolete. We confirm our experiments still hold for current systems in Section 5.

Operating System Pairing	**Circle**	**Font**	**Bézier**	**Quadratic**
Linux vs. iOS	100.0%	100.0%	100.0%	100.0%
Mac OSX vs. Linux	95.7%	100.0%	91.8%	81.0%
Mac OSX vs. iOS	94.3%	100.0%	99.2%	98.6%
Windows vs. Linux	95.2%	100.0%	91.1%	76.0%
Windows vs. Mac OSX	20.5%	100.0%	14.0%	16.4%
Windows vs. iOS	100.0%	100.0%	100.0%	100.0%

(a) Performance of candidate graphical primitives at uniquely identifying operating systems irrespective of browser or hardware.

Operating System Version Pairing	**Circle**	**Font**	**Bézier**	**Quadratic**
Windows 7 vs. Windows 8	6.6%	11.7%	5.7%	4.0%
Windows 7 vs. Windows Vista	12.0%	13.6%	7.3%	7.2%
Windows 7 vs. Windows XP	3.9%	7.4%	3.4%	4.5%
Windows 8 vs. Windows Vista	10.7%	11.7%	8.1%	8.0%
Windows 8 vs. Windows XP	10.0%	20.7%	8.0%	5.9%
Windows Vista vs. Windows XP	12.1%	19.7%	6.8%	3.7%
iOS 6 vs. iOS 7	99.7%	100.0%	100.0%	100.0%

(b) Performance of candidate graphical primitives at uniquely identifying operating systems versions within the same family irrespective of browser or hardware.

Browser Pairing	**Circle**	**Font**	**Bézier**	**Quadratic**
Chrome vs. Firefox	100.0%	100.0%	100.0%	100.0%
Chrome vs. Safari	100.0%	100.0%	100.0%	100.0%
Chrome vs. iOS Browser	100.0%	100.0%	100.0%	100.0%
Firefox vs. Safari	100.0%	100.0%	100.0%	100.0%
Firefox vs. iOS Browser	100.0%	100.0%	100.0%	100.0%
iOS Browser vs. Safari	88.7%	100.0%	98.3%	97.3%

(c) Performance of candidate graphical primitives at uniquely identifying browser families irrespective of operating system or hardware.

Browser Version Pairing	**Circle**	**Font**	**Bézier**	**Quadratic**
Chrome 28 vs. Chrome 29	72.3%	6.9%	0.1%	0.1%
Chrome 28 vs. Chrome 30	80.1%	17.7%	0.3%	0.6%
Chrome 28 vs. Chrome 31	92.7%	14.0%	0.3%	0.8%
Chrome 29 vs. Chrome 30	0.5%	17.7%	0.1%	0.2%
Chrome 29 vs. Chrome 31	0.6%	14.2%	0.1%	0.2%
Chrome 30 vs. Chrome 31	1.2%	11.9%	0.0%	0.5%

(d) Performance of candidate graphical primitives at uniquely identifying browser versions within the same family irrespective of operating system or hardware.

Table 3: Uniqueness metrics for graphical primitives.

Browser vs. browser: We present our pair-wise comparison of browser uniqueness irrespective of OS and hardware in Table 3c. We find that divergent canvas implementations per browser perform even better than OS-level differences at producing unique Picasso responses. Every primitive is nearly 100% successful at differentiating browsers with the exception of iOS implementation of Safari vs. non-iOS Safari.

Browser version vs. browser version: If we control for browser family and analyze only alternate browser versions, we find our responses are far less unique as shown in Table 3d. As with operating systems, it appears unlikely that Picasso can yield any insights on a client's browser version using our proposed primitives.

Stability: We measure the stability of responses produced by individual browsers and operating systems as the average number of responses generated per challenge (s, N, A). As with uniqueness, we omit all challenges that consisted of more than one round so we can examine primitives in isolation. We present our results in Table 4. We observe that Windows has the least stable of all device characteristics. We believe this is due to the variety of underlying hardware that composes the Windows ecosystem compared to iOS which is largely homogeneous. Our font primitive is the least stable. Intuitively, this is a byproduct of fonts producing the most unique device class signatures, but one that is highly system dependent. Our initial investigation reveals in the worst case Picasso must store 10 responses per challenge which we consider acceptable. In practice, we expect stability to decrease as a function of the number of devices in a class due to hardware differences. We explore this further in our live deployment scenario discussed in Section 5.

Chaining primitives: Our final evaluation examines the impact of chaining multiple primitives on Picasso's overall uniqueness and stability at distinguishing devices device classes of type {operating system, browser}, averaged across all pairs. We present our results in Table 5 broken down by the number of rounds. We find that absent using a primitive at least once that generates font text, Picasso can uniquely distinguish devices 28% of the time. This increases slightly with the number of rounds to a maximum of 32%. In contrast, when we include a font based primitive at least once, we observe 100% unique device class responses.[4] Independent of the primitive used, an increasing number of rounds slightly decreases the stability of responses. In effect, each element drawn to a canvas exacerbates the noise introduced by a system's hardware and software. We synthesize these observations into our final Picasso implementation later in Section 5.

4.4 System Performance

We conclude our feasibility experiments with an analysis of the CPU and memory required to generate an authentic Picasso response given a variable number of rounds and canvas sizes. We conduct our measurements on a local machine running Firefox, Chrome, and Safari to enable accurate reporting and to avoid subjecting live web clients to excessive resource consumption. We find that resource requirements for Picasso increase linearly with the number of rounds. This satisfies our design goal for a configurably difficult proof of work.

[4]We are unable to determine why uniqueness dropped for two round computations. We note that all metrics are within ± 3% of their true value.

Device	Circle	Font	Bézier	Quad.
Chrome	42.9%	14.1%	75.1%	68.6%
Firefox	19.4%	15.8%	19.8%	19.0%
Safari	84.0%	80.5%	78.0%	96.8%
iOS Browser	46.2%	29.4%	45.7%	47.6%
iOS	46.2%	29.4%	45.7%	47.6%
Linux	81.8%	79.3%	89.6%	74.4%
Mac OSX	38.1%	40.0%	42.6%	46.5%
Windows	16.1%	9.0%	19.3%	18.7%

Table 4: Average graphical primitive stability for various families of operating systems and browsers.

Metric	Round 1	Round 2	Round 3
Uniqueness$_{nofont}$	28%	30%	32%
Stability$_{nofont}$	53%	49%	47%
Uniqueness$_{font}$	100%	98%	100%
Stability$_{font}$	31%	30%	27%

Table 5: Browser and operating system uniqueness and stability as a function of the number of rounds N and whether we include font text as a graphical primitive.

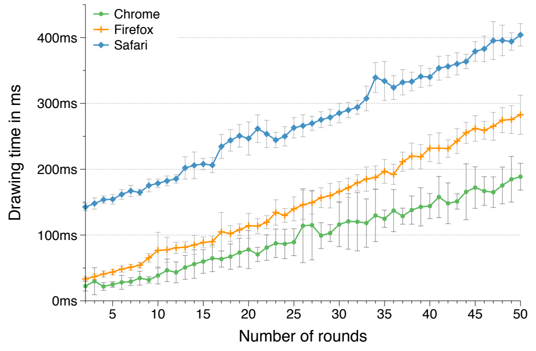


Figure 6: CPU consumption of Firefox, Chrome, and Safari in a controlled environment where Picasso required an increasing number of rounds.

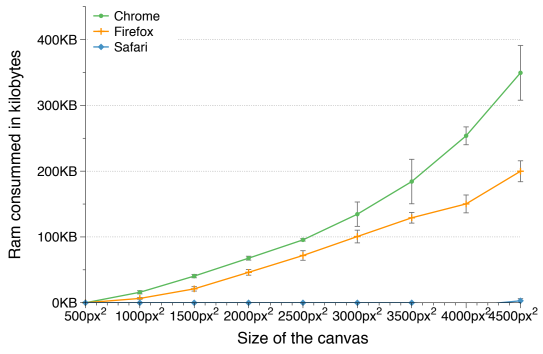


Figure 7: Memory consumption of Firefox, Chrome, and Safari in a controlled environment where Picasso required an increasing canvas sizes.

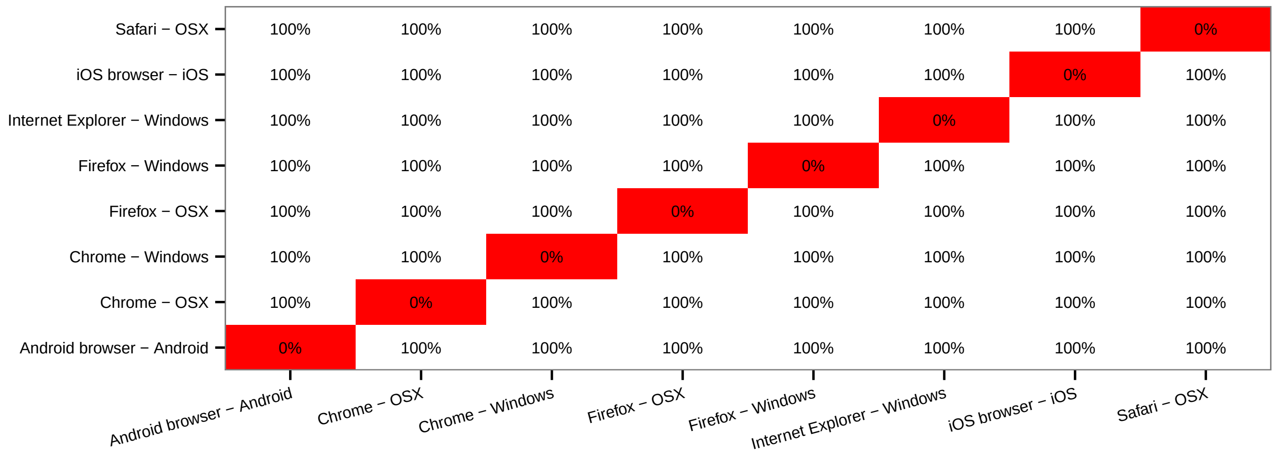


Figure 8: Pairwise uniqueness between all browsers and operating systems appearing in our dataset of 52 million clients.

CPU consumption: Using a controlled environment we computed the CPU usage of our JavaScript payload for challenges with $N = (1, 50)$ and averaged over 50 trials each with a randomized s. We present our results in Figure 6. We observe a linear increase in the time necessary to produce a Picasso response with the size of N. The most complex challenge requires upwards of 400ms to compute, well within a range suitable for multiple device classes.

Memory consumption: As a final experiment, we briefly evaluated the feasibility of controlling the amount of RAM required to solve a Picasso challenge as a function of the canvas size. For each test we varied the canvas size between $(500^2px, 4500^2px)$ at intervals of 500^2px with a randomized seed s per trial and fixed $N = 10$. To avoid any possible caching we terminated the browser process between each trial. Figure 7 reports the *additional* average real segment size used by each browser compared to the 500^2px. While memory needed to compute the challenge increased with the canvas size for Firefox and Chrome as expected, Safari showed (almost) no increase. We believe this is because Safari requests a large memory block (170MB) and then manages its own memory allocation afterwards. Our results indicate we can tune the memory usage of our system to create increasingly complex proofs of work.

5. REAL WORLD DEPLOYMENT

As a final validation we explore Picasso's effectiveness at detecting (spoofed) device classes in a live deployment scenario at Google. We issued Picasso challenges to 52 million clients over a two week period in February 2015. We present a detailed breakdown of Picasso's overall accuracy and demonstrate how Picasso detected a brute force password guessing attack.

5.1 Uniqueness & Stability

Our live deployment of Picasso ran the same algorithm presented in Section 3 with two modifications. First, we executed four rounds and selected graphical primitives in such a way that each primitive was used at least once. This maximized the potential uniqueness per client without sacrificing stability as we showed in Section 4. Next, we fixed the seed s for the entire deployment to minimize the number of unique responses we had to store.

Device class accuracy: We verify that Picasso uniquely identifies Windows, OSX, Android, and iOS systems running a multitude of browsers. As we lack ground truth of a client's device class, we again relied on a User-Agent derived from JavaScript. Due to the likelihood of spoofed devices in our dataset, we restricted our analysis to clusters of 10,000 or more clients (0.02% of traffic) that returned the same Picasso response and purported User-Agent. We present the pairwise uniqueness between all browser and operating systems in our dataset in Figure 8. We find our final Picasso implementation distinguishes device classes with 100% accuracy. This is true for all combinations of browsers and desktop and mobile operating systems.

Our accuracy is restricted to browser and operating system families, not versions. Figure 9 shows Picasso's performance at uniquely identifying Chrome versions all running on Windows 7 and unknown hardware. We find that, in the worst case, we conflate 40.8% Chrome 36 and 35 clients. A similar picture emerges in Figure 10 for the pairwise uniqueness of Windows systems all running Chrome 40 and unknown hardware. We conflate 56.4% of Windows 8 and Windows 8.1 systems. Our results indicate that Picasso achieves its goal of device class fingerprinting, but only at a family granularity.

Stability & storage requirements: Our live deployment yielded roughly 130,000 unique responses from the 52 million challenges we sent to clients. We find that responses form a long tail distribution: the top 100 responses cover 73% of all clients; the top 1,000 responses 95% of clients; and the top 10,000 responses 99% of clients. We reiterate that this distribution is at odds with device fingerprinting: Picasso's graphical primitives are incapable of unique client identification.

We present the number of responses a Picasso server must store to accurately distinguish individual operating system and browser pairs in Figure 11. We find that Mac OSX and iOS provide the most stable signatures: 100 responses capture 98% of all such clients. In contrast, the top 100 responses from purported Firefox clients on Windows accounts for 88% of clients. The same scenario covers 94% of Windows Internet Explorer users. As User-Agent spoofing is likely present in our dataset we argue these breakdowns present a worst case scenario. If we restrict Picasso's hash function to 32 bits of output, we must store a total of roughly 520KB per seed to guarantee 100% coverage. If we reduce coverage to 99% of clients, this storage requirement drops to 40KB per seed.

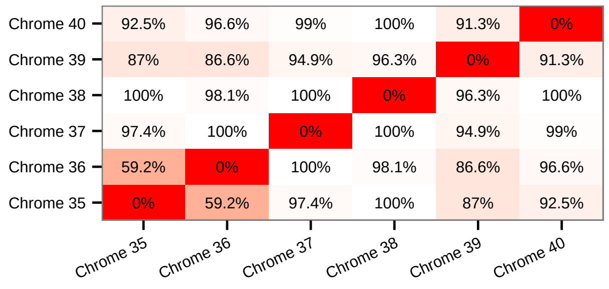


Figure 9: Pairwise uniqueness of Chrome versions for clients all running Windows 7 on unknown hardware.

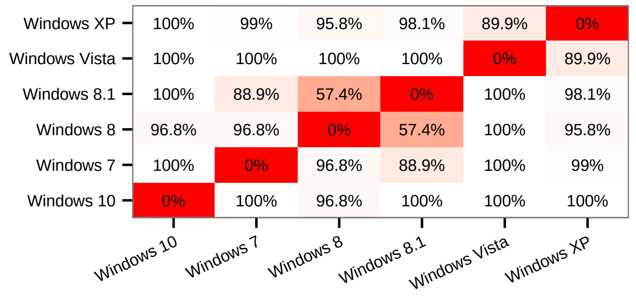


Figure 10: Pairwise uniqueness of Windows versions for clients all running Chrome 40 on unknown hardware.

5.2 Identifying Spoofed Clients

During our deployment we observed two attacks that Picasso surfaced. Both campaigns attempted to brute force the login page belonging to the web company we collaborated with. We detected each attack by scanning for large volumes of incoming requests with purported User-Agents that conflicted with the device class indicated by Picasso. We present a breakdown of the spoofed User-Agents used by each attack in Figure 12. Working back from the Picasso responses and IPs involved, we identified that one attack was launched directly from (potentially compromised) Amazon AWS instances, while the other attack was proxied through hosts in North America, Europe, and Russia—some of which also appear in Tor's directory listing. Both attacks share a device class signature we know to belong to PhantomJS running on Linux and EC2 hardware. This simple scenario highlights the effectiveness of device class signatures at helping to differentiate malicious clients.

6. RELATED WORK

User fingerprinting: Fingerprinting techniques attempt to uniquely detect devices and users in the absence of overt tracking mechanisms such as cookies. Previous schemes build on side effects induced by browsers, operating systems, and hardware. Researchers have considered a wealth of techniques ranging from User-Agent strings, header orders, font lists, enabled plugins, IP addresses, screen sizes, and time zones to uniquely identify clients [4, 6, 22]. Similarly, divergent JavaScript, HTML, and CSS implementations across browser stacks (and even versions) yield unique test outputs and timing information [19,21,27]. Researchers have proposed even lower level detection approaches to fingerprint CPU and device timings [3, 12, 15, 23], memory patterns [10], as

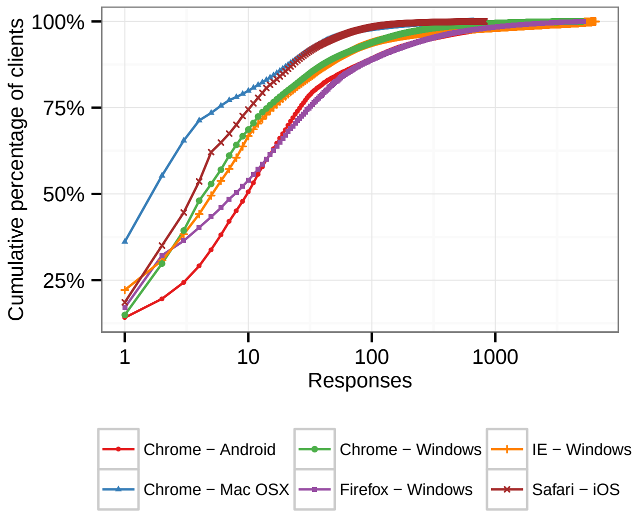


Figure 11: Cumulative coverage of device populations broken down by operating system and browser family. The top 100 responses for each category cover 88–98% of clients.

well as browser fonts and canvas elements that surface GPU and operating system divergences [7, 20], some of which are widely deployed by web services today [1]. Our system instead found a set of signals that are non-spoofable and accurately distinguish classes of devices, but have minimal variations between devices in the same class.

Puzzles & proofs of work: The central idea of a proof of work is to design a challenge that is difficult to solve but trivial to verify. In the process a client expends arbitrary computation or memory as specified by a server [5]. While simple scenarios consider finding at an input that hashes to a configurably difficult output (*e.g.*, a hash that starts with n ones), attackers can offload challenges to more powerful computational devices or spread work among compromised hosts [17]. Again, we built on this concept, but limited our hardware-bound proof of work to device classes in order to prevent attackers from offloading to other device classes (*e.g.*, solving challenges intended for mobile devices on cloud machinery.)

7. CONCLUSION

In this paper we presented Picasso, a system that leverages the complexity of a device's browser, operating system, and graphical stack to provide accurate device class fingerprinting with a hardware-bound proof of work. Our JavaScript implementation of Picasso, when properly configured using the right graphical primitives, is able to successfully distinguish the browser family (Chrome, Firefox, etc.) and the OS family (Windows, iOS, OSX, etc.) of over 52 million clients with 100% accuracy. Web services can use Picasso to filter inorganic traffic. We perceive a number of applications including blocking non-mobile clients from app marketplaces; detecting rogue login attempts to a client's account; and detecting emulated clients. As a consequence, attackers can no longer rely on simple automation techniques and instead must conform to the device class of organic users.

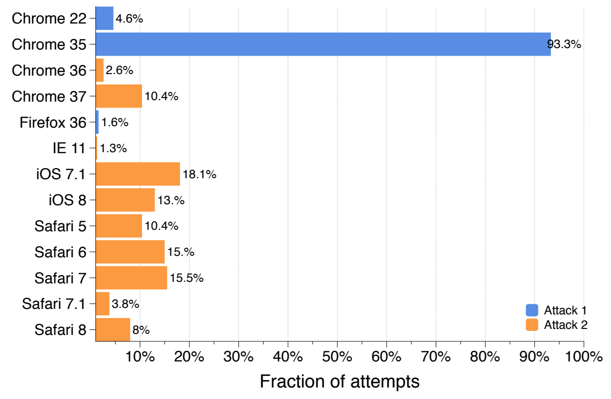


Figure 12: Purported User-Agents provided by malicious clients in two distinct attacks. Miscreants launched both attacks via Amazon EC2 instances running PhantomJS.

8. REFERENCES

[1] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the Conference on Computer and Communications Security*, 2014.

[2] T. Ahrens. Type rendering mix. *http://blog.typekit.com/2013/12/18/type-rendering-mix/*.

[3] A. Bates, R. Leonard, H. Pruse, D. Lowd, and K. Butler. Leveraging usb to establish host identity using commodity devices. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, 2014.

[4] K. Boda, Á. M. Földes, G. G. Gulyás, and S. Imre. User tracking on the web via cross-browser fingerprinting. In *Information Security Technology for Applications*. 2012.

[5] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In *In Proceedings of Advances in Cryptology*, 2003.

[6] P. Eckersley. How unique is your web browser? In *Proceedings of the Privacy Enhancing Technologies Symposium*, 2010.

[7] D. Fifield and S. Egelman. Fingerprinting web users through font metrics. In *Proceedings of the International Conference on Financial Cryptography and Data Security*, 2015.

[8] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Y. Zhao. Detecting and characterizing social spam campaigns. In *Proceedings of the ACM SIGCOM Internet Measurement Conference*, 2010.

[9] C. Grier, K. Thomas, V. Paxson, and M. Zhang. @ spam: the underground on 140 characters or less. In *Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[10] Y. Gu, Y. Fu, A. Prakash, Z. Lin, and H. Yin. Os-sommelier: Memory-only operating system fingerprinting in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[11] G. Gugliotta. Deciphering old texts, one woozy, curvy word at a time. *http://www.nytimes.com/2011/03/29/science/29recaptcha.html*, 2011.

[12] G. Ho, D. Boneh, L. Ballard, and N. Provos. Tick tock: building browser red pills from timing side channels. In *Proceedings of the USENIX Workshop on Offensive Technologies*, 2014.

[13] C. Hoffberger. Youtube strips universal and sony of 2 billion fake views. *http://bit.ly/10MpDse*, 2012.

[14] T.-K. Huang, M. S. Rahman, H. V. Madhyastha, and M. Faloutsos. An analysis of socware cascades in online social networks. In *Proceedings of the International Conference on the World Wide Web*, 2013.

[15] T. Kohno, A. Broido, and K. C. Claffy. Remote physical device fingerprinting. *Proceedings of the IEEE Transactions on Dependable and Secure Computing*, 2005.

[16] P. Laperdrix, W. Rudametkin, and B. Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2016.

[17] B. Laurie and R. Clayton. Proof-of-work proves not to work; version 0.2. In *In Proceedings of the Workshop on Economics and Information Security*, 2004.

[18] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, and W. Lee. The core of the matter: Analyzing malicious traffic in cellular carriers. In *Proceedings of the Network and Distributed System Security Conference*, 2013.

[19] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting information in javascript implementations. In *Proceedings of the Workshop on Web 2.0 Security and Privacy*, 2011.

[20] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in html5. In *Proceedings of the Workshop on Web 2.0 Security and Privacy*, 2012.

[21] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. Wien. Fast and reliable browser identification with javascript engine fingerprinting. In *Proceedings of the Workshop on Web 2.0 Security and Privacy*, 2013.

[22] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 541–555. IEEE, 2013.

[23] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the USENIX Workshop on Offensive Technologies*, 2009.

[24] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld. Physical one-way functions. *Science*, 2002.

[25] U. Rührmair, F. Sehnke, J. Sölter, G. Dror, S. Devadas, and J. Schmidhuber. Modeling attacks on physical unclonable functions. In *Proceedings of the Conference on Computer and Communications Security*, 2010.

[26] K. Thomas, F. Li, C. Grier, and V. Paxson. Consequences of connectivity: Characterizing account hijacking on twitter. In *Proceedings of the Conference on Computer and Communications Security*, 2014.

[27] T. Unger, M. Mulazzani, D. Fruhwirt, M. Huber, S. Schrittwieser, and E. Weippl. Shpf: enhancing http(s) session security with browser fingerprinting. In *Proceedings of the International Conference on Availability, Reliability and Security*, 2013.

[28] L. Von Ahn and L. Dabbish. Labeling images with a computer game. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2004.