Who is Fiddling with Prices?

Building and Deploying a Watchdog Service for E-commerce

Costas Iordanou
Universidad Carlos III de Madrid, Telefonica Research
kostas.iordanou@telefonica.com

Claudio Soriente
Telefonica Research
claudio.soriente@telefonica.com

Michael Sirivianos
Cyprus University of Technology
michael.sirivianos@cut.ac.cy

Nikolaos Laoutaris
Data Transparency Lab
nikos@datatransparencylab.org

ABSTRACT

We present the design, implementation, validation, and deployment of the Price $heriff, a highly distributed system for detecting various types of online price discrimination in e-commerce. The Price $heriff uses a peer-to-peer architecture, sandboxing, and secure multiparty computation to allow users to tunnel price check requests through the browsers of other peers without tainting their local or server-side browsing history and state. Having operated the Price $heriff for several months with approximately one thousand real users, we identify several instances of cross-border price discrimination based on the country of origin. Even within national borders, we identify several retailers that return different prices for the same product to different users. We examine whether the observed differences are due to personal-data-induced discrimination or A/B testing, and conclude that it is the latter.

CCS CONCEPTS

• General and reference → Design; **• Information systems → Crowdsourcing**; **Online shopping**; **• Computer systems organization → Peer-to-peer architectures**;

KEYWORDS

Online Price Discrimination

ACM Reference format:
Costas Iordanou, Claudio Soriente, Michael Sirivianos, and Nikolaos Laoutaris. 2017. Who is Fiddling with Prices?. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21–25, 2017,* 21 pages.
https://doi.org/10.1145/3098822.3098850

1 INTRODUCTION

Over the last few years, a handful of measurement studies have indicated that online price discrimination (PD), *i.e.*, the practice of selling the same product to distinct customers at different prices that depend on the customer's online behavior, is becoming increasingly commonplace among e-commerce sites. These studies have mostly established that the location of a customer, and in particular the country of origin, inferred via his IP address and language settings, often affects the observed price in ways that cannot be explained in terms of currency, taxation, duty, or shipping costs.

In a few cases, researchers have even managed to reverse engineer, or at least hypothesize, about the suspected causal relationship between location and price and have shown, for example, that prices appear to be adjusted using simple multiplicative factors depending on the country of the customer [24]. Despite this initial progress in unveiling *cross-border online PD*, little is known about other aspects of dynamic pricing. For example, despite anecdotal evidence, there's little work in measuring dynamic pricing within national borders. Do customers within the same country see different prices for the same product by the same vendor? If they do, can this be attributed to Personal-data-induced price discrimination (PDI-PD) based on non-location specific customer data (*e.g.*, browsing history as opposed to IP address) collected by e-commerce sites, or third party trackers? In which cases are the observed price variations a result of plain A/B testing performed by pricing software (e.g., [7]) that tries to learn the underlying elasticity curve without using the personal information of individuals in a discriminatory manner?

Contributions: We have designed, implemented, and operated the *Price $heriff* (also referred to as plainly the $heriff), a hybrid infrastructure / peer-to-peer (P2P) system that uses a network of dedicated measurement servers around the world and a P2P network formed by the Firefox and Chrome users of the $heriff add-on. The dedicated servers of the system measure the price of products using cleanly installed web-browsers and operating systems that do not maintain any browsing history or cookies. These reference product prices are compared to the prices observed by the peer clients. A peer client is either the initiator of a price check request or fetches product pages on behalf of the initiator. Both, the peer clients and the dedicated servers fetch the product price at the same time in order to factor out temporal price variations.

The peer clients' browsing history and cookies are known to the online tracking ecosystem, which can use them to drive PDI-PD. The P2P component equips the system with multiple measurement points within the same location. These measurement points exhibit diverse and real-world browsing behaviors, which are used by the system to detect PDI-PD.

Tunneling requests through other peers broadens our observational capability but also imposes difficult security and privacy challenges. We use white-listing to ensure that the P2P system

SIGCOMM '17, August 21–25, 2017, Los Angeles, CA, USA

ACM ISBN 978-1-4503-4653-5/17/08…$15.00
https://doi.org/10.1145/3098822.3098850

cannot be exploited for fetching illegal content. We also use sandboxing to protect and cleanse the local state of a browser after fetching a product page on behalf of another peer user. Protecting the server-side state of a user, *i.e.*, the information maintained by trackers at their own servers is more involved. To this end, we enforce an upper threshold on the number of page fetches that a user conducts every week on behalf of other peers. Once this thresholds is exceeded, instead of sending back his actual cookies to e-retailers and third party trackers, a peer client sends the tracking cookies of his assigned *Doppelganger*. A small set of such doppelgangers are trained and maintained by the back-end of our system to be used as shield against server-side state pollution of real $heriff peers. A doppelganger is a browser instance built to closely represent the browsing profiles of a cluster of real users for whom it is used as a "double". The set of maintained doppelgangers is decided by running a novel and secure version of k-means that, for privacy reasons, does not require our system to know the actual browsing profiles of its real $heriff users.

The $heriff has more than 1000 users in 55 countries. Protecting the server-side state of these users can be achieved with as little as 40 doppelgangers. In the last 12 months, these users have generated more than 5700 requests, checking the price of more than 4800 products across 1994 e-commerce sites. Using the collected data from real users as a compass, we have identified a number of e-stores generating dynamic prices for distinct customers. We focus on these sites and conduct a large-scale measurement study by artificially generating requests for multiple products and tunneling them through both our infrastructure and peer proxies. The generated dataset includes more than 12000 requests across 1000 products, enabling an in-depth analysis that would be unattainable only with the data generated by our real users.

Findings: The analysis of the aforementioned data yields the following results:

• 76 out of the 1994 checked e-commerce sites return prices that may vary depending on the country or other characteristics of the user after having excluded to the best of our ability the effects of taxation, duties, or currency. The observed price variations are substantial (*e.g.*, $\times 7$) and can result in actual price differences of more than $10000 (professional digital camera).

• 7 out of the 76 e-commerce sites where price difference was observed returned different prices even for users within the same country. The dispersion of prices within countries (up to $\sim 8\%$) appears to be smaller than the ones across countries (up to $\sim 700\%$).

• Looking at certain e-retailers within specific countries we have detected signs of A/B price testing as well as biases of some peers towards consistently high or low prices. However, by analyzing prices using various statistical models, we conclude that the specific e-retailers do not perform PDI-PD.

• To extend the scope of our search for price variation within the same country we also examined the 400 most popular Alexa e-commerce sites. Yet, we did not find any of them returning different prices to distinct users within the same country. This also implies that we did not detect PDI-PD among those 400. Although there probably exist several other retailers that return dynamic prices within the same country, and thus might also be engaging in PDI-PD, we do not believe that this practice is popular.

• In the course of our temporal analysis we came across complex strategies under which the majority of the products of a retailer become cheaper through successive small price drops over 20 days. At the same time, we observed a series of large price jumps for a few products. If these products are popular, these price jumps result in an overall revenue and (presumably) profit increase for the retailer.

Online price discrimination is one of many instances of *algorithmic discrimination* [16] discussed in the context of an intense ongoing debate around big data mining, online tracking, privacy, and the business models of the web. The $heriff exemplifies that making sense of such technologies, and the controversies around them, will require the development of a new breed of *transparency software*. Although our study did not result in the detection of PDI-PD by the domains we examined, our software has "watchdog" value. To the best of our knowledge, this is the first system of its kind. Its implementation and deployment showcases the challenges and design principles for such a system. Our design lessons are not limited to PDI-PD detection; our system's paradigm can find applications to domains beyond price discrimination, such as geoblocking, automatic personalisation, and filter-bubble detection.

2 BACKGROUND AND REQUIREMENTS

Although price discrimination (PD) is probably as old as commerce itself, its application in e-commerce is fairly recent. Odlyzko [25] postulated in 2003 that the ease of e-commerce could eventually backfire for customers due to online PD driven by the personal information that users leave behind in their "digital trace". For example, users visiting websites that carry expensive products or users who are geo-located to affluent ZIP codes could be steered to more expensive products or be displayed higher prices. In terms of the legality of the practice, there are several barriers, such as the US Robinson-Patman Act of 1936, or Article 20.2 of the "Services Directive" of the EU; this Directive prohibits PD based on country of origin or country of residence in the member states. Beyond legislation, there is mounting public concern around "online personalization" of services and the point at which it becomes discriminatory, especially when driven by personal or sensitive data. In all these cases, examining the legality or the ethics of a situation is difficult if not impossible without evidence. Collecting such evidence, especially evidence of *personal-data-induced* PD is exceedingly difficult due to several technical challenges.

Within the context of this work, we are concerned with product price variations at the same URL. We define the following types of price variations:

Location-based PD is any product price difference observed at approximately the same time between two or more geographical locations (e.g., city or country) excluding any taxation and shipping costs.

A/B Testing is the practice of serving two or more different prices for the same product and observing how users respond to them in order to determine a new price for the product.

PDI-PD is the practice of serving different prices for the same product to different users within the same geographical location (e.g., city or country) based on some knowledge about the user interests and behavior.

We clarify that our definition of PDI-PD does not include price differentiation due to the browser or OS used. In our measurements, we control for the desktop browser or OS as discussed in Sect 7.5. Also, Hannak et al. [20] found that mobile devices were shown different prices than desktop ones. The $heriff is not yet ported to smartphone browsers, so we do not have measurement points from mobile devices.

In addition, our definition of PDI-PD also applies in cases of price steering [20]. Online price steering is the practice of showing different products (or the same products in a different order) to distinct users for the same search query. Regardless of the search result, if two users in the same location end up checking the same product and the price varies, the $heriff will detect the discrepancy as PDI-PD. However, the $heriff cannot discern whether price steering took place.

Lastly, in our analysis (see Sect. 7.5), we treat all price variations that are not attributable to location-based PD or PDI-PD as if they stem from A/B testing. Such unclassified price variations may, among others, be due to divergent currency converters or price transitions that occur during a price check request (e.g., due to algorithmic pricing, which enables hundreds of changes per day [18]).

Next we elaborate on why collecting evidence of price discrimination is a challenging distributed systems problem. We do so by listing the requirements that our Price $heriff has to fulfill.

2.1 General requirements for PD detection

1. User-friendly and adoptable. To detect location-based PD we need multiple vantage points at distinct cities, counties or states. Providing sufficient coverage with infrastructure hosts alone would entail a prohibitive cost for the administrators of our system. The Price $heriff follows a crowdsourcing-based approach to collect product prices from varying e-commerce sites from diverse locations all around the world. Therefore, we need to attract numerous users. To this end, the system needs to be highly usable by untrained users with minimal technical background.

2. Scalable and elastic. The underlying measurement system should be able to keep up with an increasing number of users and demand for price checks. Since results need to be presented in real time, the system needs to be able to dynamically allocate more resources during load peaks.

3. Universal price extraction algorithm. The system needs to be able to extract product prices from a large variety of e-commerce sites. Retailers use complex site layouts, different scripting languages, and pack multiple recommendations in the same page of a given product, thus making price extraction a non-trivial task.

4. Automated currency conversion. Retailers state prices based on their local currency or the currency of the customer, which they try to infer through IP geo-location, browser settings, etc. Furthermore, they often deviate from standardized currency codes, thus hindering automated currency conversion.

2.2 PDI-PD detection requirements

1. Distinguishing between location-based and PDI-PD. The system must be able to discern if a price difference is due to the user location as opposed to other personal data, such as their past browsing history. Placing dedicated proxy servers at distinct locations suffices for detecting location-based PD. Detecting, PDI-PD, however, is substantially more involved since it requires having multiple measurement points within the same location. E-commerce sites attempt to bundle and hide PD among complex tax, duty, or shipping costs. Obtaining multiple prices in the same location factors out these complications.

2. Detecting and collecting information about trackers. To go beyond just detecting PDI-PD and, for example, to attempt to attribute it to specific reasons, one needs to be able to detect possible sources of information that may have caused the discrimination. This requires, among others, to be able to detect the presence of third party trackers and investigate whether it correlates with observed price variations.

3. Collecting samples of browser history. The need to collect browsing history samples stems from our hypothesis that recently visited domains can be used to influence product prices, similar to how web search results can be influenced by recently searched keywords [20]. The presence of trackers indicates a possible channel for obtaining information to drive PDI-PD. However, to reverse engineer how this information is being used, one needs to obtain some description of the "profile of a user" as seen by advertisers and marketers. User profiling relies largely on the observed browsing behavior of a user, hence a PDI-PD detection system needs to have access to a sample of a user's recent browsing history at a domain level. Note that accessing the entire browsing history of the user at the granularity of a full URL is not recommended since the full URLs are prone to leak personally identifiable information about the user (e.g., the user' Facebook profile page). The donated samples of browsing history can then be used to check whether visiting specific domains impacts the prices observed in other domains, which would constitute PDI-PD.

4. Preventing the pollution of real user profiles. The system issues price check requests towards other peers in order to utilize the diverse profiles of the real users and their geographical location in search for evidence of PDI-PD. Yet, the system should be able to isolate the real browsing behavior of its users from any measurement introduced by the system. That is, the tool needs to avoid polluting the browsing profile of users with an excessive number of tunneled product page visits.

5. Protecting user privacy. The tool should prevent private user information leakage to other users. In addition, unless the users have explicitly volunteered to donate information about the presence of trackers or samples of their unencrypted past browsing history, the system should not leak personally identifiable information to our infrastructure. Furthermore, the previous requirement motivates us to introduce the concept of doppelgangers. Our infrastructure creates them by using the browsing profile of real users. Therefore, the process of creating doppelganger profiles needs to provide strong privacy guarantees. In addition, our infrastructure should not be able to link doppelganger profiles to real user profiles.

We analyzed various existing tools to determine if their functionality matches our requirements. We opted to build upon the $heriff [24] PD detection tool because it already satisfies a small portion of our requirements, thus saving us development effort. In Appendix, Section 10.1, we provide details on which requirements $heriff satisfies and which it does not.

2.3 Ethics, user privacy and security

We have ensured compliance with the EU General Data Protection Regulation pertaining to collecting, handling and storing data generated by real users. To that end, we have acquired all the proper approvals by our institutions and the Spanish Data Protection Authority. Note that we do not collect any personally identifiable information about our users. By addressing requirements (3) and (5) in Sect. 2.2, we do a best effort to minimize the PII exposed to our system. We also blacklist the URLs of user profile or account management pages of e-retailers because they are likely to include PII, such as the name of the user. Thus, even if the user accidentally or knowingly activates the add-on on that page, our system will not fetch the content. We also periodically analyze our collected data to discern if PII has accidentally been stored by our system, e.g., due to omitting to blacklist a URL. In case this happens, we will immediately delete the pertinent information and update our blacklist. For more information about the information being collected please visit http://sheriff-v2.dynu.net/views/home.

Furthermore, next to the installation button of the $heriff add-on on each web browser store, we provide a *"Before you install"* section explaining that the add-on is not intended for children and that our tool performs page requests from e-commerce websites on behalf of other users. In addition, the first page shown to the user after installation is the informed consent one, along with a button to easily uninstall the add-on. Unless the user consents, the add-on is not activated and does not collect any user information.

Importantly, we also ensure that only e-commerce domains are allowed during any product price request by filtering against a whitelist that is manually constructed and updated over time. Therefore, the peer clients cannot be requested to visit malicious or controversial websites.

Operating our tool for more than a year we did not observe any instances of malicious behavior on behalf of our users (i.e., aggressive number of price check requests towards specific retailers) nor any attempts to send price check requests towards suspicious domains. It is worth mentioning that we did not receive any complaints from our user pool regarding any misbehavior of the add-on or any other anomalies related to user experience.

3 THE PRICE $HERIFF

In this section, we describe how our design and implementation satisfies the aforementioned requirements. As it will soon become apparent, the $heriff is a complex distributed system and therefore we have chosen to discuss only some of its most interesting and challenging architectural and implementation aspects.

3.1 Architectural overview

A Price $heriff user can issue a request to check for discrepancies in the price of a product by employing multiple and diverse clients. We do so by fetching and comparing the price from a set of fixed vantage points (Infrastructure Proxy Client - IPC) and a set of other users (Peer Proxy Client - PPCs) located close to the user who initiated the price check request. Figure 1 depicts the seven main components of our architecture: the Browser add-on, the Measurement servers, the Coordinator, the Database server, the Network of IPCs and PPCs, the Aggregator, and the Doppelgangers.

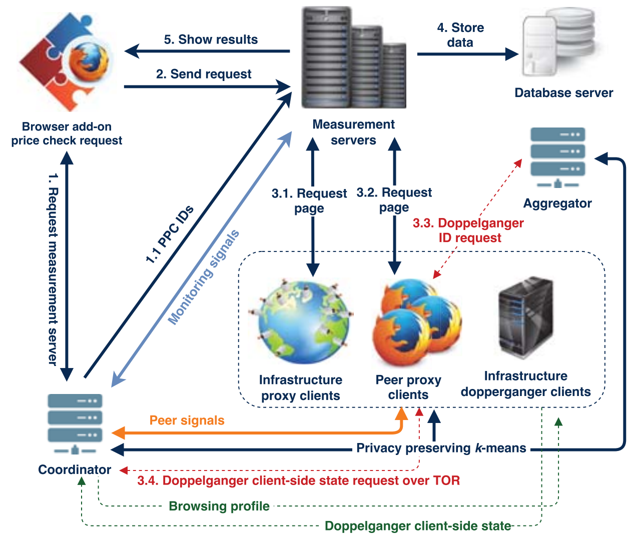


Figure 1: The Price $heriff architecture overview. The seven main components, and the flow of messages during a single price check request.

All Prices Results | Results Details

Variant	Converted Value	Original Text
You	€ 654	EUR654
Windows 7, Chrome, Spain	€ 654	EUR654
Mac OS, Safari, Spain	€ 654	EUR654
Linux, Firefox, Spain	€ 654	EUR654
United States, Tennessee	€ 617.65*	$699
United States, Massachusetts	€ 617.65*	$699
United States, Washington	€ 617.65*	$699
Canada, British Columbia	€ 646.26	CAD912
Canada, Ontario	€ 646.26	CAD912
Canada, Ontario	€ 646.26	CAD912
Israel, Beer-Sheva	€ 665.07	ILS2,963
Sweden, Scandinavia	€ 667.37	SEK6,283
Japan, Tokyo	€ 655.60	JPY88,204
Japan, Hiroshima	€ 655.60	JPY88,204
Czech Republic, Praha	€ 662.00	CZK18,215
Korea, Seoul	€ 668.29	KRW829,075
New Zealand, Dunedin	€ 668.28	NZD997

* Currency detection confidence is low. Please double check the result.

Currency converter:
Price in Euro € to United States Dollars $ Convert It

Figure 2: A sample result page with all the currencies automatically detected and converted to Euro.

3.1.1 Coordinator and Measurement servers. The Coordinator along with the Measurement servers, the Aggregator and the Database server constitute the back-end of our system. The *Coordinator* acts as a load balancer that distributes price check requests from the browser add-ons to the multiple Measurement servers. Importantly, the Coordinator also tracks all the PPCs in the system. This is because it is tasked with forwarding to the selected measurement server the list of PPCs that reside in the same geographical location as the initiator of the price check request (see Sec. 3.2). Our system can dynamically attach and detach Measurement servers according to the number of concurrent user requests.

The structure of the system back-end is presented in Fig.3.

The *Measurement servers* carry out price checks by distributing requests to (Infrastructure and Peer) Proxy Clients. The *Database*

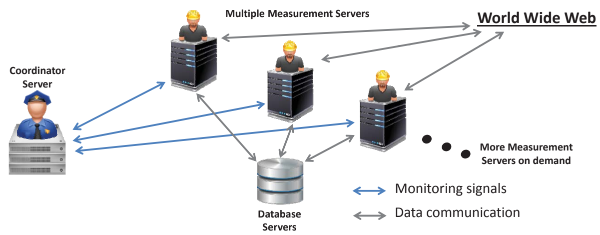


Figure 3: The structure of the Price Detective back-end.

server stores all the information collected from the Measurement servers. Initially, the system was implemented with an RDBMS running on each Measurement server. This design led to consistency issues between multiple RDBMS running at distinct Measurement servers. Therefore, we switched to a scheme with a single shared database running on its own separate server. The Database server is only responsible to host the back-end database (MySQL [3]) of the system. We performed several RDBMS performance tunings such as increasing the number of connection threads kept in memory, using fast stored procedure calls, *etc.* Centralizing the database on a separate node has the additional benefit of reducing the number of required high-end servers since the Measurement servers can now run in more lightweight hardware (See Appendix, Sect. 10.2.1 for more details).

3.1.2 Browser add-on. The add-on enables users to initiate a price check request by navigating to a product page and highlighting the price. Using the API of the Firefox and Chrome browsers, we are able to directly access all major services that the browsers offer, such as the history service, cookie service, options, cache memory, HTTP headers, etc. Thus, it is also able to access third party domain and browser history information. No information leaves the browser unless the user explicitly opts-in to help us search for PDI-PD by donating such info. The third party domains are also available at the result page to inform the user about what domains she has visited in the background.

3.1.3 Network of proxy clients and doppelgangers. Proxy clients accept instructions from a Measurement server to fetch a specific product page and return the HTML code back to the Measurement server. The two different types of Proxy Client processes reside on dedicated infrastructure nodes that are dispersed in diverse geographic locations (*IPC*) and on the browser add-ons (*PPC*). The Measurement server issues requests to proxy clients on behalf of a user. When the proxy clients send to the Measurement server the page's code, it parses it and shows the results back to the user who initiated the request.

When a Measurement server asks a PPC to fetch a product page from an e-retailer, we are inevitably altering the state the e-retailer keeps for that peer. That is, the e-retailer server may infer that the user behind that PPC is interested in the visited product, while in reality the product page was downloaded only to serve the purpose of our system.

In order to avoid overly altering (i.e., polluting) the server-side state of PPCs, we introduce the notion of *doppelgangers* (see Sect. 3.6.2). A doppelganger is a browser instance built to closely represent the browsing profiles of a cluster of real users. A PPC can fetch product pages by using the client-side state (*i.e.*, cookies or other tokens) of its assigned doppelganger, thus protecting his own cookies from being connected to page downloads that do not match his true interests. Doppelgangers are created and maintained on a set of infrastructure clients managed by the Coordinator. Assigning doppelgangers to users is carried out by the Coordinator in cooperation with the Aggregator, by using a privacy-preserving clustering protocol that protects the browsing profile of our users. After a doppelganger has served a certain number of price check requests, its profile is considered polluted. Thus, it is discarded and is regenerated with a new client- and server-side state.

3.2 Price check request protocol

In Fig. 1 we can see the steps involved in a product price check request. The user performs step 1, which includes the navigation to an e-store and the selection of the product price. The add-on contacts the Coordinator to get a globally unique job ID and the address of the available Measurement server that will serve the request. Upon receiving the request, the Coordinator compares the requested domain against a whitelist of acceptable domains. This step is required to make sure that we only allow requests towards sanctioned e-commerce websites. Rejected requests are collected in the background for manual inspection and update of the whitelist.

Besides coordinating the Measurement servers, managing their load and whitelisting price check requests, the Coordinator is also responsible for tracking the PPCs (browsers with the add-on installed) within the system. Each time a web browser with the Price $heriff add-on starts, it sends a message to the Coordinator with its peer ID and location. The Coordinator maintains lists with peer IDs grouped together based on each browser's location at a zip-code, city or country level, depending on the granularity of the available geo-location service. During step 1.1 the Coordinator sends the list of other PPC IDs in the location of the initiating user to the selected Measurement server.

During step 2, the browser add-on constructs a path of HTML Tags, which we refer to as *Tags Path* (see Fig. 4), towards the product price that has been highlighted by the user using his cursor, and forwards the request to the Measurement server. The request includes the URL of the product's web page alongside the Tags Path.

During step 3.1, the Measurement server requests from *all* the IPCs to download the product page. (We currently have 30 deployed IPCs.) At the same time, in step 3.2, the Measurement server asks from the PPCs received during step 1.1 to download the product page (*remote page request*). At this point, if any of the PPCs reaches a predefined page request threshold (see Sect. 3.6.2), it does not fetch the page using its own client-side state. Instead it sends, in step 3.3, a request for its corresponding doppelganger ID to the Aggregator ("Doppelganger ID request" - red dotted arrow). It subsequently uses this ID in step 3.4 to request the cookies and other tokens (client-side state) from the Coordinator ("Doppelganger client-side state request" - red dotted arrow). The add-on installs that client-side state and requests the page from its sandboxed browser environment. At the end of steps 3.1 and 3.2, the IPCs and PPCs send the downloaded product page to the Measurement server.

Note that to preserve the privacy of the initiator peer against other PPCs, the PPCs are not directly contacted by the initiator. Thus they never learn an association between a unique peer identifier (e.g., IP) and the pages the peer visits. The only information released to the PPC about the initiator is that it is a peer residing in the same geographic location as the PPC.

Upon receiving the product pages, the Measurement server uses the Tags Path to locate the product price within the pages. The price is automatically extracted and converted to the currency requested by the user who initiated the request. We obtain exchange rates in real time and use several heuristics to accurately detect various currencies. Next, the Measurement server saves all the information in the Database server (step 4). At step 5, it forwards the results to the user's browser add-on. In Fig. 2 we present an example of the add-on's results page for a price check.

At this point the browser executes AJAX requests to the Measurement server to receive any result updates until the measurement server replies with a 'request finish' response.

The presented architecture is able to handle multiple concurrent price check requests. Each measurement server can also handle multiple price check requests in parallel. New measurement servers can be assigned to the system dynamically according to load and availability policies.

Discussion. We now discuss how the retailers can detect and actively subvert our tool. As mentioned above, the tool uses two types of measurement points, the IPCs and the PPCs. The IPCs are more prone to detection since their IP addresses are usually the same over time. A retailer can detect any abnormal activity of the IPC by counting the frequency of the visits from the same IP. If the number of page requests is above some internal frequency threshold then the retailer may block the IPC request or introduce a CAPTCHA before serving the final product web page. On the other hand, PPCs are more diverse in IP addresses since they reside at real user devices and they are greater in number. Furthermore, the IP addresses of the PPCs typically change over time by their internet service providers. From the e-retailers' perspective, detecting and blocking the PPCs requests is very difficult. Note that during the experiments reported in this paper, we did not observe any IPC or PPC product page requests being blocked by retailers.

In the remainder of this section we provide more details on several aspects of our design.

3.3 Tags Path construction

Fig.4 illustrates how the browser add-on builds the Tags Path, which enables the Measurement server to locate the price on the product pages fetched from the IPCs and PPCs. During price selection (step 1), the add-on extracts the HTML tag element that includes the product price. In our simplified example (Fig.4), the add-on extracts the tag $10.00 marked with the number 4 at the beginning of the line. This HTML tag will be the final tag on the path that leads to the product price.

The next step is to create a path with HTML tags that can be used to programmatically locate the price. The add-on starts from the bottom of the page because the HTML document format requires some additional information at the beginning of the document within the HTML Tag <head></head>. The final HTML Tags Path

```
     | <!DOCTYPE html>
     | <html>
     |   <head>
     |     <title>Hi there</title>
     |   </head>
     |   <body>
     |     This is a simple web page
     |     <div class="product">
     |       Here is the product image
     |       <img src="product.jpg" alt="Product View" style="width:304px;height:228px;">
4.->|       <span class="price">$10.00</span>
3.->|     </div>
2.->|   </body>
1.->| </html>

Tags Path = Bottom, </html>, </body>, </div>, <span class="price">
```

Figure 4: The browser add-on creates the Tags Path when the user selects the price. In the above simplified example, the algorithm starts from the bottom of the page moving upwards. It creates a new entry for every HTML tag towards the price.

is located at the bottom of Fig. 4. The first element of the graph is the starting point (in our example, the word Bottom). Subsequently, the algorithm moves upwards. Each step is marked at the beginning of the line with a number. At each step, the algorithm appends the corresponding HTML element to the graph.

At line 1, the algorithm appends the ending tag </html>, at line 2 the ending HTML tag </body> and so on, until we reach the HTML tag that the user initially selected. Now the add-on "knows" where the product price is located within the HTML code of the web page. Subsequently, the add-on forwards the Tags Path to the Measurement server, which uses the Tags Path to extract the product price from the HTML code returned by different proxy clients.

Note that the example presented here is simplified. It does not capture the complexity involved in extracting a product price when the HTML code includes multiple product prices and when the result varies between remote page requests. The latter happens because web pages can be created dynamically or they include different ads or content tailored to the corresponding user or the location of the proxy client that requests the page.

3.4 Price check request distribution

After the initial release of the browser add-on and a number of a few articles and blog posts explaining the purpose of our research the system encountered significant traffic spikes. Fig. 5 presents the number of downloads and active users over time for the Firefox version of the add-on. As can be seen, three major spikes appeared after such event. The result of these spikes was to have a high number of remote page requests in a short time window (usually for a few hours after the release of a new post) pushing our system to its limits. To overcome this issue and minimize the loss of data and poor user experience, we decided to implement a load balancing policy that enables us to minimize the cost of the system, in terms of resources required to support dynamic number of users, and deal with this type of traffic spikes.

To achieve load balancing among the Measurement servers, we devised a simple *price check request distribution protocol* taking into

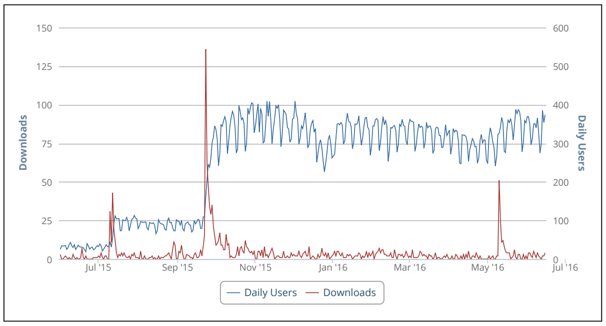


Figure 5: User statistics from the Firefox add-ons service, presenting the number of downloads and the number of active users over time. Similar statistics were observed for the respective Google Chrome add-on.

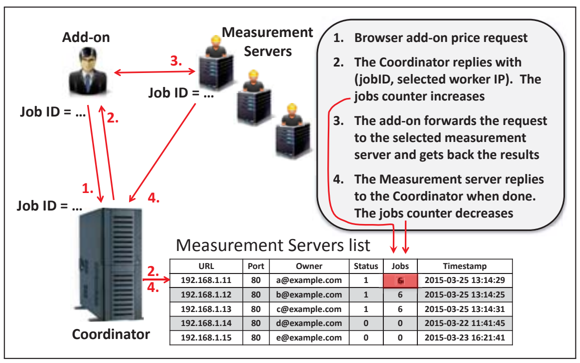


Figure 6: The steps of the request distribution protocol.

consideration the following observations. Distinct websites yield varying response times depending on the price-related content they serve and their capacity. At the same time, we have distinct Measurement servers with varying processing and networking capabilities. As a result, we cannot use a naive algorithm to distribute the jobs among available Measurements servers. For example, the round robin algorithm [8] would introduce long pending queues to Measurement servers with lower specifications and slow connections. To this end, our request distribution protocol addresses a variation of the *job shop problem* [22] using an online heuristic.

Fig. 6 shows the steps of the price check request distribution protocol. The Coordinator uses the *Measurement server list* (shown at the bottom of Fig. 6) to monitor the status of all available Measurement servers and number of pending jobs assigned to them. The list holds the following information for each measurement server: a) the IP or URL; b) the port number; c) the server status (online or offline); d) the number of pending jobs; and e) a time-stamp. (See Appendix, Sect. 10.3 for more details.)

The Coordinator monitors each Measurement server's workload in real time and assigns new incoming job requests accordingly. The protocol starts when a browser add-on contacts the Coordinator with a new request (Fig. 6 - Step 1). Upon receiving the request, the Coordinator creates a new unique job ID to be able to track

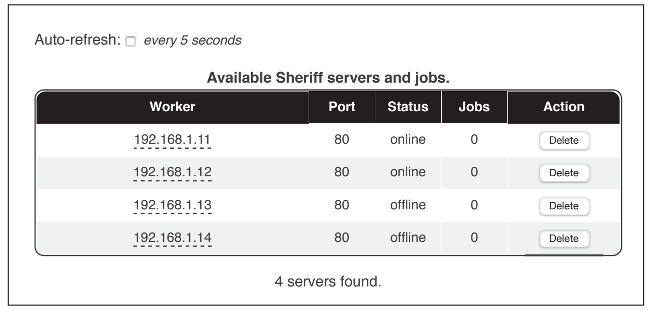


Figure 7: Measurement servers monitoring panel. For each server we monitor the current status (online or offline) and the number of pending jobs assigned to each server, in real time.

the request within the system. Subsequently, it selects the server with the lowest number of pending jobs among all servers that are currently online. The Coordinator responds to the add-on with the unique job ID, URL, and port of the selected Measurement server. At the same time, it increases the job counter of the selected server (Fig. 6 - Step 2).

After the step 2, the browser add-on has the required information to contact a Measurement server and request the price check. The message between the add-on and the Measurement server includes the product URL, the job ID and the Tags Path (Fig. 6 - Step 3). After the Measurement server has processed a price check request, it informs the Coordinator that the corresponding job is completed (Fig. 6 - Step 4). The Coordinator uses the job ID to determine the Measurement server that was serving this job, and decreases its counter.

By using the aforementioned approach, the response time of the system improves as "slower" servers are assigned fewer requests. Furthermore, new servers can be dynamically attached to the system when traffic spikes are observed to reduce the cost of running the system.

3.5 The currency detection problem

The system has to be able to measure price differences from a large number of e-commerce web pages. This task is hindered by the fact that e-retailers use varying currency notations. Furthermore, for each IPC, the resulting prices are usually in different currencies depending on each IPC's location. To help users to identify price differences on the results page, we implemented a currency detection and conversion algorithm to convert all the prices in a common currency. In Fig. 2 we present an example of the results page for a price check.

The currency detection and conversion algorithm runs on each Measurement server and is divided into three main parts. The first part removes any newline characters and multiple spaces. The second part of the algorithm aims at detecting the price currency. The currencies can be presented in three different ways: a) The most common and generally accepted way is the 3 letter notation (*e.g.*, USD), b) a custom notation that e-retailer may decide to use (*e.g.*, US$); and c) a currency symbol (*e.g.*, $). The algorithm follows the above order to identify the currency. More specifically, in the case (a) the detection is straight forward. In the case (b) the algorithm tries to match the word with a record in the custom currency list that we empirically built based on notations that some popular e-retailers use. In the case (c) the converted amounts are annotated with a red asterisk to notify the user that the currency detection algorithm's confidence is low. This can happen, for example, when e-retailers use the dollar sign ($) to represent ambiguously the Australian, Canadian or US dollar. To handle such cases, we also include a currency conversion tool at the bottom of the result page (Fig. 2) so that the user can manually perform the conversion. Lastly, if no match is found, then either the selected text does not include any currency symbol or the e-retailer uses some unknown notation to represent currencies. In this case, the displayed prices are not converted to a common currency until we update the custom currency notation list.

The third part of the algorithm tries to extract the price. Note that the system accepts requests when the selected string has less than 25 characters and at least one digit. These two constraints, combined with input sanitization, are in place as a sanity check and to prevent code injection attacks. If we are unable to detect the price value, this means that the processed word is a concatenation of letter and number words. In such case, we split the single word into words of letters and words of digits and repeat the part 2 of the algorithm.

3.6 Avoiding PPC state pollution

Price discrimination may take place by leveraging client-side and server-side state. By *client-side state* we refer to 3rd-party cookies (which among others indicate pages the user has visited), cookies set by e-retailers themselves (which may authenticate the user or store shopping carts, etc.), JSON Web Tokens (JWT), and in general any state that is stored on the browser as a result of the user browsing activity. With *serve-side state*, we denote any information that an e-retailer may retain about a user. This includes product pages viewed, purchase history, etc. User identification to build the server-site state may be achieved by means of account credentials or cookies (hence, client-side state). It can also be achieved by utilizing their IP or by performing device or browser fingerprinting [14, 15].

E-retailers may discriminate using the client-side state that browsers sent to them (*e.g.*, domains visited). If they have uniquely identified a user, they may also discriminate using the server-side state they have stored for that given user. We need to be able to detect both types of discrimination. Therefore, when a PPC serves a price check request initiated by another peer, it must submit its own client-side state to the requested domain. However, this will alter any server-side state kept for that PPC by that domain. Furthermore, cookies set when fetching the remotely requested page may also pollute the client-side state.

Therefore the PPC state, either at the client-side or at the server-side, becomes polluted with state that does not represent the PPC's local user's browsing behavior. The side-effects of such pollution are multifaceted. First, profile pollution hinders the detection of PDI-PD by progressively making all peers' browsing behavior appear uniform. In particular, if we do not constrain the number of price check requests in which we expose the real profile of a user towards the various e-retailers, the users will end up showing interest towards the same set of products reducing the diversity of our user pool. Consequently, our capacity to observe PDI-PD would be diminished. Second, it can cause the profile of a PPC to change substantially from its original one, thereby creating undesirable consequences. That is, irrelevant advertisements and recommendations being shown to the local user based on visits to product pages fetched for price check requests initiated by other users.

We prevent pollution of the client-side state by sandboxing product price checks and deleting the cookies set when the product page is fetched (see Section 3.6.1). Preventing pollution of the server-side state is more involved and employs *doppelgangers* (see Section 3.6.2).

3.6.1 Sandboxing PPCs. To sandbox the request for product pages by PPCs caused by price check requests, we use several browser extension APIs [4, 5] that alter the default behavior of the corresponding browser during the request execution. We combine multiple API calls including the cookie service, the HTTP(S) connection service, the browser history service and the browser cache memory service. At the end of such request, the sandboxed environment is deleted keeping the browser history and cookies clean of any trace of the price check request.

We are able to detect, add and delete cookies and other tokens that are inserted due to product page requests, irrespective of the techniques used to install them. For example, to delete the cookie from the HTTP(S) header, we monitor the HTTP(S) connection and delete the cookie before it reaches the browser's cookie service. The add-on is also able to detect cookies that are created dynamically by JavaScript code. Such cookies can be detected by monitoring the cookie service for any change during the page requests. To delete any traces of the requested product page URL, we use the browser history service and browser cache memory service to clean up the corresponding records.

We evaluated both versions of the $heriff add-on (Google Chrome and Mozilla Firefox) in three popular operating systems (Windows, Macintosh and Linux) with more than 30 beta testers. We also assessed the add-on on a number of virtual machines (VMs) with freshly installed operating system and browsers. We ran the beta testing phase for one week sending price check requests between all testers, while the VMs only serving remote product page requests. We did not observe any cookies installed nor any traces of remote product page requests in any VM. We received similar feedback from all the beta testers.

3.6.2 Mitigating server-side state pollution. Our solution to the server-side pollution problem involves PPCs serving price check requests up to a predefined threshold of tolerable profile pollution. If the real user of the PPC has never visited the targetted domain, it executes the request and deletes the client-side state assosiated with that domain as described in the previous section. Normally in this case, no server-side state pollution takes place and no threshold needs to be enforced. On the other hand, if the real PPC user has already visited the domain, we should not delete its associated client-side state because she needs it. For each e-commerce domain, we consider that 25% of additional products visits due to price check requests constitutes a tolerable level of pollution[1]. Hence, we allow

[1]Note that further study is required to determine the precise tolerable pollution per domain.

one new product page request for every 4 product pages that the real user of the PPC has visited on the given domain.

Above this per-domain threshold, a PPC no longer serves requests for the given domain using its client-side state, rather it uses the client-side state of its *doppelganger* (see Sect 3.7). A doppelganger is a *fake* user with a browsing profile similar to the ones of real users, which however does not correspond to the browsing profile of any real user. Note that doppelgangers cannot prevent pollution due to server-side state built via IP tracking or fingerprinting.[2]

To protect the confidentiality of PPC browsing profiles, we should not have one doppelganger per PPC. Therefore, the Coordinator clusters users according to their browsing profiles and assigns to each user one doppelganger (k-mean) browsing profile vector computed as described in Sect. 3.7. The Coordinator asks from dedicated infrastructure clients to execute the doppelganger browsing profile vectors by fetching websites and accumulating client-state, which they send back to the Coordinator. Besides enhancing confidentiality, the clustering reduces the load on our infrastructure because it needs to train and manage a small number of doppelgangers, while ensuring that their browsing profiles are still representative of our real users' browsing behaviors.

Therefore, each doppelganger profile corresponds to a single cluster comprising real user profiles that resemble each other. By definition, the real profile of a user has a higher visited domain diversity compared to its doppelganger. Thus, we partially allow the use of real user profiles to increase the probability to observe any instances of PDI-PD with the downside of introducing a small amount of pollution to the user profile. When a PPC reaches the predefined level of acceptable pollution, instead of rejecting any consequent price check request it swaps in its doppelganger profile client-state to execute the product page request. By doing so the PPC exposes a less accurate representation of the real user, yet it remains an active vantage point in its current geographical location (IP) rather than being altogether discarded. Without the doppelganger profiles the geographical diversity of the P2P network would decrease dramatically during price check request peeks, especially for countries with only few peers available.

If a PPC has already reached the threshold of tolerable pollution and is called upon to serve a product page request, it asks the Coordinator for the client-side state of its doppelganger. It subsequently installs the doppelganger's client-side state in a *sandboxed* environment and fetches the page of the requested product while emulating a doppelganger from its own IP address. Our system guarantees that only the doppelganger's client-side state leaves the browser even if the visited page includes cookie matching code or other re-directions. Note that we apply a similar pollution-prevention rationale with the one we employ for real PPCs. If the doppelganger has never visited a domain, we simply delete the associated client-side state. If it has, we allow one product page request for every 4 requests performed during the creation of the doppelganger. If 50% of the domains visited by the doppelganger are saturated, we request from infrastructure clients to regenerate the doppelganger.

[2] In 2013 only 0.04% of the Alexa top 1M websites where observed to use fingerprinting code [15]. A similar study one year later [14] showed that 5.5% of the Alexa top 100K domains was serving fingerprinting code and the 95% of the overall 5.5% was served by the addthis.com domain. These studies show that the likelihood of exposing our users to domains that serve fingerprinting code is low.

3.7 Doppelganger creation

The Price $heriff infrastructure must build doppelgangers with profiles that "look" similar to the profiles of the PPCs. To this end, we cluster users based on their *browsing profile vectors*. The browsing profile vector of a user is a (normalized) one dimensional vector that defines the frequency of visits to each of m domains. The frequency values are in $[0, 1]$, where 0 indicates that the user has no visits to that domain and 1 indicates that is the most visited domain of the user. (We defer a discussion on how to select such domains to Section 4.) We use k-means clustering and create k centroids that define the browsing profile vectors of the doppelganger. Hence, the doppelganger derived from a given cluster centroid is assigned to all users included in that cluster.

In a straw-man solution, the PPCs would send their browsing profile vectors to our infrastructure as cleartext so that we cluster them and define doppelgangers. However, this would have undesirable privacy implications. First, peers who share their browsing history, even if they use an anonymity network (*e.g.*, Tor [11]), are subject to unique identification by an infrastructure that colludes with other domains. As shown in [26], a browsing profile vector with only a handful of entries suffices to uniquely identify a user. If our infrastructure colludes with a few external domains, it can unveil sensitive information (username, credit card, etc) of a uniquely identified user, while offering full behavioral profiles to the colluding parties. For example, a colluding e-store can now learn not only the browsing history of a user with respect to its own site, but also his browsing history at other domains too. This information is valuable enough to motivate the owners of the infrastructure to sell the browsing profiles to e-retailers. Second, even if the exact peer browsing profiles are somehow concealed from the infrastructure, the mapping between a peer's IP and a doppelganger unveils to the infrastructure sufficient knowledge about the peer's browsing behavior, which it can then exploit by colluding with external domains.

To address the above privacy-related shortcomings we devised a cryptographic protocol for privacy-preserving k-means computation. Under this protocol, PPCs share their browsing profile in an encrypted form. The k-means computation is split between the Coordinator and a second trusted entity called Aggregator. At the end of the computation, the Coordinator only learns the browsing profile vectors of the k cluster centroids (*i.e.*, the profiles of the doppelgangers). The Aggregator only learns which PPCs are mapped to each cluster, but it does not learn the profile of any PPC nor the one of any centroid. As long as the Coordinator and the Aggregator do not collude, our protocols allows to cluster users based on their browsing profiles vectors, while keeping the actual profiles private.

We envision that a trustworthy non-governmental organization or a data protection authority will run the Aggregator in the future, while the Coordinator runs at our facilities. One may argue that since we place trust on the Aggregator, we could entrust it with the cleartext profiles of the PPCs. However we chose to compartmentalize the shared information and the computation, in order to minimize trust on a single system component. We trust the Aggregator with the mapping between PPCs and clusters, but not with cleartext profiles or the cluster centroids. At the same time, user information is less vulnerable to an Aggregator security breach that does not involve cooperation with the Price $heriff infrastructure. We believe that such reduced liability renders our system more adoptable by external entities.

To download the client-side state of the assigned doppelganger, a PPC contacts the Aggregator to learn the ID of its assigned doppelganger. To prevent the Coordinator from learning to which centroid a PPC maps, the PPC contacts the Coordinator through an anonymity network to obtain the client-side state of the doppelganger. Because users remain anonymous to the Coordinator, anybody could abuse the service and query for all the doppelganger profiles. This would facilitate the blacklisting of doppelganger remote page requests. To address this issue, doppelganger IDs are random and sufficiently long (256 bits). In this way, the doppelganger IDs act as a bearer token and the Coordinator grants the doppelganger client-side state only to those who submit the correct token.

3.8 Privacy-preserving k-means computation

We define the browsing profile of a user as the number of visits to each of m domains over a given period of time. Therefore, each PPC is represented by a point in an m-dimensional space where each dimension is an Internet domain and the (normalized) coordinate represents the amount of visits to that domain within the browsing history of that PPC.

We face the issue of clustering users based on their browsing profiles, while keeping such information private to its owner. This could be achieved with any of the secure Multi-Party Computation (MPC) frameworks available [17, 21], but it would require each user to be online while clustering takes place. In our web setting, it is not practical to require all clients to be online at a given time. Rather, we want a client to provide its (encrypted) point and then be able to go offline. Towards this goal, the *Aggregator*, helps computing clusters and releases PPCs from the burden of being online. In particular, the Aggregator maintains the mapping between a client ID and a cluster ID, but does not learn the actual client point, nor the cluster centroids. At the same time, the Coordinator learns the cluster centroids, without learning the private point of any client, or which clients are mapped to a given cluster. (The Coordinator learns, however, the cardinality of each cluster at each iteration.)

Our private k-means protocol builds on top of the functional encryption scheme of [13] to compute the dot-product of two private vectors. In particular, it builds on top of the additively homomorphic version of ElGamal [19] where messages are encrypted "at the exponent" (See Sect.10.4).

Our intuition is that given two m-dimensional points $\mathbf{a} = (a_i)_{i\in[m]}$, $\mathbf{b} = (b_i)_{i\in[m]}$ we compute the squared distance $d^2(\mathbf{a}, \mathbf{b}) = \sum_{i=[m]} a_i^2 + \sum_{i=[m]} b_i^2 - 2\sum_{i=[m]} a_i b_i$ by evaluating the dot-product of the vectors $\mathbf{c} = \sum_{i=[m]} a_i^2, 1, a_1, \ldots, a_m, \mathbf{s} = 1, \sum_{i=[m]} a_i^2, s_1, \ldots, s_m$ Therefore, we set point $\mathbf{a}$ to be the browsing profile of a user and point $\mathbf{b}$ to be the browsing profile of a centroid. Given $\mathbf{a}$ (resp. $\mathbf{b}$), the user (resp. the Coordinator) can privately compute vector $\mathbf{c}$ (resp. $\mathbf{s}$).

Before clustering actually starts, a client is only asked to encrypt $\mathbf{c}$ under the Coordinator public key and send it to the Aggregator. Once the Aggregator has received the encrypted client vectors, the protocol iterates over two phases: a) client-cluster mapping and (b) cluster centroid update. During phase (a), the Aggregator maps clients to clusters by learning the distance between a client point and all the cluster centroids. Distance computation between the two points is carried out by leveraging the dot-product protocol of [13] where the Coordinator act as the server with private input $\mathbf{s}$ and the Aggregator runs as the client with input the encrypted vector $\mathbf{c}$ as received by the client. Note that the Aggregator learns $d^2(\mathbf{a}, \mathbf{b})$ but it does not learn the client point, nor it learns the cluster centroids. Phase (b) starts when all clients have been assigned to clusters. It allows the Coordinator to compute the new centroid of a cluster as the average of all client points assigned to that cluster.

To do so, we use the additive homomorphism of the encryption scheme used in [13]. This allows the Aggregator to compute the ciphertext of the sum of all the points, without actually learning anything. The Aggregator forwards the aggregated ciphertext to the Coordinator, which decrypts and divides the result by the cardinality of the cluster, in order to compute the new cluster centroid. The two phases iterate until a halting condition is reached. In our scenario, we halt when the difference in client-cluster mapping across two iterations, as observed by the Aggregator, is below a given threshold.

The security of our protocol relies on the Aggregator and the Coordinator being honest-but-curious. In particular, we do not consider the case when the Aggregator or the Coordinator create fake clients in order to infer the browsing history of other victim clients. Also, we require the Aggregator and the Coordinator to be in different administrative domains and to not collude.

In the Appendix , Sect. 10.4 of this technical report, we provide a more detailed description of the cryptographic protocol and we formally argue about its security.

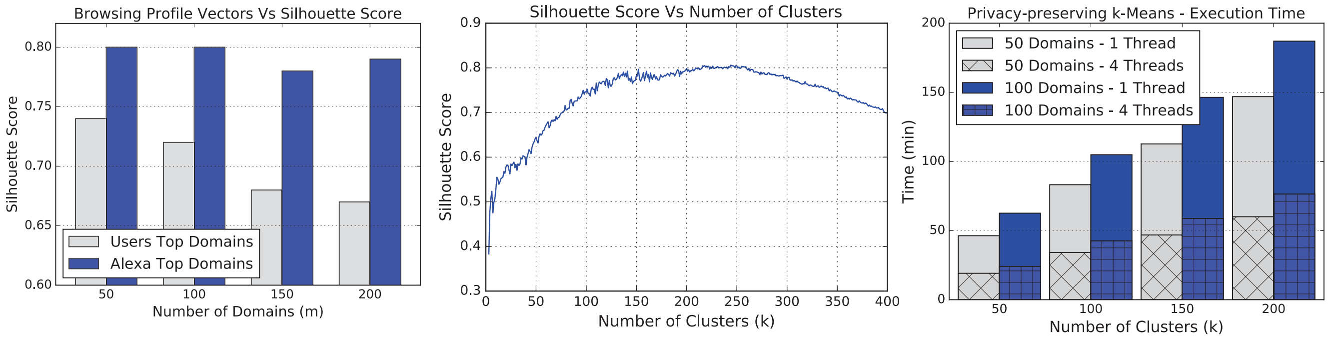
4 DOPPELGANGER EVALUATION

We present several experiments that we conducted in order to fine-tune the doppelganger part of our system. We start by empirically determining which and how many domains to use in defining the profile of a user (*browsing profile vector*). We devise two options for clustering ≈ 500 Price $heriff users that donated their cleartext history during a time window of 3 months. We compare the two options by computing the clustering quality via silhouette scores.

The silhouette score [27] is a measure of similarity between a single point and its own cluster compared to the other clusters.The silhouette score gets values in the range of $[-1, 1]$, where high values indicate that the data points within a cluster are more similar among each other than they are to other nearby clusters. It is therefore a measure of the clustering quality.

For the option "Users top Domains", we take into account the top m domains of our user-base. For the option "Alexa Top Domains", we consider the top m domains from the Alexa top domains list. For each of the two options, we vary m between 50 and 200 and present the results in Fig.8(a).

We observe that "Alexa top Domains" yields a higher silhouette score than "User top Domain". Also, the clustering quality drops as the number of domains increases. We decided to use "Alexa top Domains" and set $m = 100$. This is because it exhibits a good clustering quality while it keeps a fairly large number of domains that are sufficiently representative of the various user type profiles.



(a) (b) (c)

Figure 8: (a) Varying browsing profile vectors and the maximum silhouette score of their clusters for 500 users. (b) A representative example of how the silhouette score varies as a function of the number of clusters (k) for a data set of 500 users. (c) Privacy-preserving k-means execution time for a single thread and for four parallel threads.

"User top Domains", in some cases, captures domains that are popular only among a few users but not visited by the majority of the other users, thus yielding a sparser browsing profile vector. This affects clustering since it can lead to clusters with a very small number of users. On the other hand, "Alexa top Domains" captures domains that are more popular across all users, thus yielding a denser browsing profile vector and consequently a more balanced and accurate clustering.

To determine the optimal number of clusters (k), and in turn the number of doppelgangers, we look at how silhouette scores change with respect to k. Our goal is to strike a balance between clustering quality and the overhead of creating and maintaining doppelgangers. Fig.8(b) shows that the silhouette score curve reaches up to around 0.6 with as little as 40 clusters. Higher values of k provide better clustering quality but also translate to higher overhead. They can also compromise user privacy as discussed in Sect. 3.6.2. We have repeated this experiment on a weekly basis for two months and have always experienced a behavior similar to the one in Fig.8(b), with clustering scores around 0.6 for $k \in [40, 60]$. Based on the above observations we set an upper threshold for k to be the 10% of the number of user independently of the silhouette score. Setting an upper threshold for k ensures that doppleganger creation and maintenance does not saturate our system's resources.

Finally, Fig.8(c) depicts the execution time of the privacy-preserving k-means algorithm for a single iteration. We use a synthetic dataset of $\approx$ 500 users and set $m \in \{50, 100\}$. On the x-axis we have the target number of clusters (k) starting from 50 up to 200 clusters in steps of 50. The y-axis depicts the execution time in minutes for a single iteration of the clustering process. The grey bars represent the execution time for 50 dimensions and the blue bars for 100 dimensions of the browsing history vector for a single thread execution. The hashed part of each bar represents the execution time with multiple threads running in parallel. Observing the execution time of the hashed highlighted part of each bar, we conclude that the protocol is highly parallelizable. On average, the privacy-preserving k-means algorithm requires between 6 to 10 iterations to converge. We can achieve further improved performance by utilizing a Graphical Processing Unit (GPU) such as nVidia Tesla [10].

The last step of the process involves the creation of the doppelganger profiles based on the final centroids of each cluster. During this step the Coordinator sends the centroid vectors to a set of PlanetLab server nodes that can build the profiles in parallel in a few hours. The final identifiers for each domain in each group are then pushed back to the Aggregator server.

5 PERFORMANCE ANALYSIS

In this section we present a comparative performance analysis of Price Detective's implementation with the $heriff v.1 system.

To be able to quantify the overall performance improvement of the Price Detective system over the original $heriff tool we deploy both systems in a cloud service and automate few clients to start sending price check requests to each version of the system. The Table 1 presents the performance evaluation results for both systems under different workloads.

The stress-test setup is relatively simple. We set up few web browsers with the Selenium plugin [9] installed (Table 1 - # Clients column) and automate a number of price check requests from different e-stores on each one of them. We utilize the *Monitoring Web Interface* (Fig. 7) to monitor the number of pending jobs (Table 1 - # Tasks) on each server. To measure the response time (Table 1 - Response Time Per Task) for each request, we manually monitor the log files of the measurement servers (Table 1 - # Servers). Each experiment starts using two client browsers to raise the workload to the appropriate number of parallel tasks. After reaching the desired workload only one client was sending requests. At the beginning, we try few dry runs to be able to estimate the price check request rate on the clients. This is needed to estimate the average response time per task for a window of at least 15 minutes. A high price check request rate causes the $heriff v.1 Measurement server to crash before the 15 minute window, and a low rate was unable to increase the workload over time.

In Table 1, the first two rows present the results from the $heriff v.1. In the first row we have the results for one client and one server. The overall response time is approximately two minutes per task. Assuming a stable task rate without any spikes, we can serve up to 3600 requests per day. The second line present the results for the same setup but this time we increase the workload near the critical point of 10 parallel tasks. Note that the response time now

Table 1: System Performance Analysis

	# Clients	# Servers	# Tasks	Response Time Per Task (min)	Max Daily Requests
Old Version	1	1	≈ 5	≈ 2	3600
	2	1	≈ 10	≈ 5	2880
New Version	1	1	≈ 5	≈ 1	7200
	2	1	≈ 10	≈ 1.5	9600
	3	4	≈ 10	≈ 1.5	38400

increases up to 5 minutes per task. The two main reasons behind this observation are the CPU context switching and the attached database on the measurement server (See Sect. 10.2.1).

The last three lines of Table 1 present the results from the new version of $heriff. The first line presents the results for a single client and a single server. Interestingly the response time per task value, is approximately 1 minute. This result is bounded by the proxy servers response time and not by the measurement server. Specifically, we notice that some PlanetLab servers are sometimes overloaded, imposing delay on our proxy servers response time. Consequently, this delay is reflected on the overall time required for a task to finish[3]. In the second line of the *New Version* tests, we present the results with 10 parallel task using 2 client browsers. The approximate response time is around 1.5 minutes. The additional 0.5 minute increase, compared to the previous test, is due to CPU context switching of the proxy request threads. Compare the results with the corresponding old version stress-test, we have approximately 3.5 minutes response time reduction. This reduction is due to the separated shared database server and the overall code optimization (See Sect. 10.2.1).

The final line of Table 1 present the results of a large scale performance test with 3 client browsers and 4 measurement servers. By monitoring the number of pending tasks on the measurement server and setting a safe threshold (in this test, 10 parallel tasks) we can fire up additional servers to handle requests spikes without compromising the response time per task. With 4 measurement servers and a safe threshold of 10 parallel tasks the new version of $heriff can serve up to 38400 price check requests per day.

Note that the above stress-test results are an approximation of the overall system performance. Each component, such us, the Coordinator, the Measurement server, the various proxy clients, and the network state, can impose varying time delays during the test. To avoid any server crashing on the production system, we reduce the number of parallel tasks threshold to the two thirds of the critical workload for each deployment. In our final production deployment, using a different cloud service, the average response time is approximately 30 seconds per task for the 3 client browsers and 4 measurement servers configuration.

6 LIVE VALIDATION

Before we dive into our results, we provide some information on our user recruitment processto bootstrap our deployment. The initial recruitment step involved uploading the browser add-on to the corresponding website for each web browser (Mozilla Firefox and Google Chrome). Then, we leveraged the users' curiosity about online product pricing by spreading the word in online social networks. At this point we managed to get the attention of a few journalists around the world. After the publication of a few articles in the popular press (businessinsider.com, businessoffashion.com, mathbabe.org, idlewords.com, incibe.es) and a TV documentary in the Swiss national TV (RTS Un), we managed to recruit more than 1000 new users[4] from all over the world.

Next, we present results from the live deployment of the Price $heriff system. All data presented and analyzed in this section are generated as a result of price check requests by real-world users.

6.1 Methodology

We analyze results obtained from August 2015 through September 2016. In total, we observed 1265 unique users from 55 countries. Table 2 depicts the top-10 countries ranked by the number of users' price check requests. Each price check request is tunneled through 30 IPCs and approximately 3 PPCs. The requests involve 1994 checked domains and 4856 checked products. These requests yielded 160248 responses. The number of users that donated browsing history during this experiment is 459.

6.2 General findings

Out of the 1994 checked e-commerce sites, 76 (3.8%) were involved in at least one price check that resulted in some difference of price between either infrastructure proxies or peer proxies. Fig. 9 depicts the number of requests and the observed price difference (standard box-plots) for 29 domains where we observed price difference in at least 10 price checks performed by users.

Fig. 9 illustrates that there are several e-commerce sites with median measured price difference in the range of 20%-30% (*e.g.*, digitalrev.com, luisaviaroma.com, overstock.com, steampowered.com, suitsupply.com), as well as few where the median is near 40% (abercrombie.com, jcpenney.com). The list includes e-commerce sites across diverse fields, including clothing, digital/electronics, travel, bookstores, art/gallery, bicycles, *etc.* Table 3 depicts the extreme observed differences in terms of relative price between cheapest and most expensive observation point and the resulting absolute difference in price. As can be seen, there were cases where the measured price could differ by a factor of 2.55 between measurement points (*i.e.*, 155% more expensive). In terms of absolute price difference, the maximum difference was € 1201. A special case that we observed in multiple occasions is an expensive digital camera (Phase One IQ280) from www.digitalrev.com of which the retail

[3] To overcame this problem in the production system, we set a 2 minutes upper bound for each thread request towards a proxy server, before we kill it.

[4] The overall number of installations is much higher but we only count users that initiated at least one price check request.

Table 2: Top 10 countries with the higher number of user requests

Country	Spain	France	USA	Switzerland	Germany	Belgium	UK	Netherlands	Cyprus	Canada
# Users	2554	917	581	387	217	161	126	96	95	92

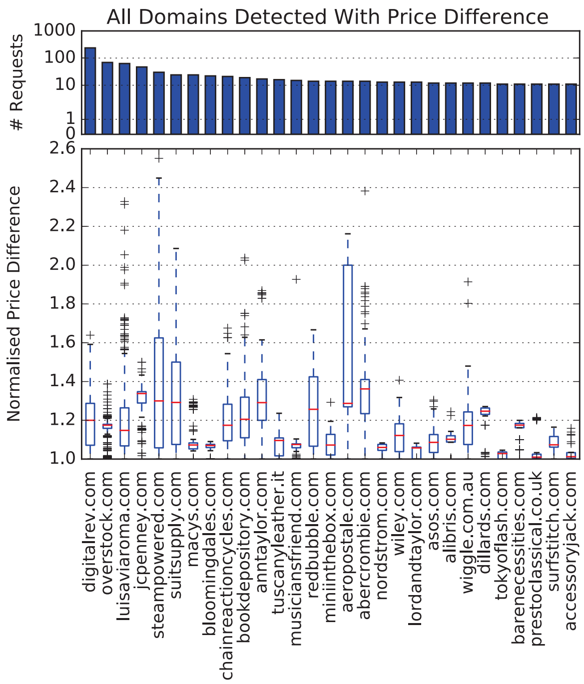


Figure 9: Initial analysis of the live dataset created by real users. (Top) Domains with the highest number of requests where price difference occurred. (Bottom) Magnitude of normalized price difference per domain.

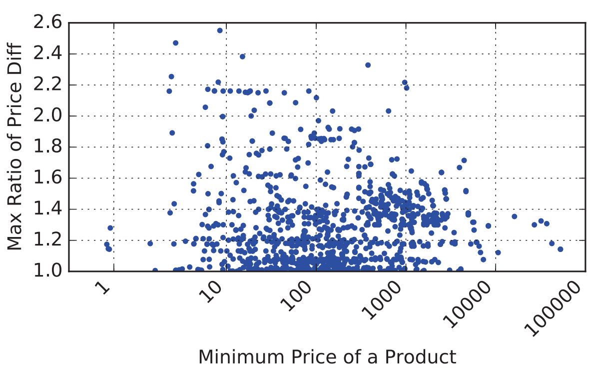


Figure 10: Ratio of the maximum price over the minimum of a product (y-axis). The x-axis depicts the products sorted in increasing order of the minimum price. We use the live dataset depicted in Fig 9.

price in Europe was around €34.5k,in Canada around €45k, in the US almost €41k and in Brazil above €46k. Thus, between the two extremes we have more than €10k price difference. We manually checked that shipping and duty costs were not included in the product prices. Similarly, excluding a single case, we checked that VAT was always either not included or was the VAT of the location of the seller and, thus, independent of measurement point.

For each price check that yielded some price difference, we identified the country where the maximum and the minimum price was observed. Table 4 sorts the countries in terms of the number of products for which they were found to be the most expensive or the cheapest. (Note that the two lists need not be non-overlapping as one country can be among the most expensive ones for some products but among the cheapest ones for others.)

Table 3: Extreme Observed Differences

Domain	Product Description	Difference	
		Relative (Times)	Absolute (EUR)
steampowered.com	Computer Game	2.55	13.12
abercrombie.com	Clothing	2.38	21.00
luisaviarome.com	Clothing	2.32	502.17
luisaviarome.com	Clothing	2.18	1201.00
aeropostale.com	Clothing	2.16	96.12
suitsupply.com	Clothing	2.08	64.00
raffaello-network.com	Men’s Accesories	2.03	660.00
bookdepository.com	Book rental	2.03	21.18

We also examine the price ratio between maximum and minimum price observed for all measurement points in the live dataset (Fig. 10). The highest price differences are between products costing €5 to €1000 and can be up to ×2.5, thus, 150% price difference. For products between €1K - €10K the price difference is as high as ×1.7. For the expensive products, in the range of €10K to €100K, the maximum price difference is 30%.

6.3 Personal-data-related findings

Next we looked at e-stores that were involved in at least one price check that returned a price difference *within the same country* between the requesting add-on and other PPCs or an IPC in the same country. We found 7 such cases, the top 3 being: amazon.com (12 cases), jcpenney.com (7 cases), and chegg.com (6 cases). In the next section, we examine these domains in more detail.

7 SYSTEMATIC MEASUREMENT STUDY

We now describe the results of our systematic large scale measurement study, which focused on e-commerce sites that showed signs of discriminatory behavior based on location or personal data.

7.1 Methodology

We used the live system’s results as a compass to direct us towards domains from which we observed at least one request with price variation within the same country. We selected those domains for systematic crawling. We also included domains with price difference between different countries that fall below the ninety fifth percentile of the total number of requests to have a better insight. The crawling entailed generating artificial price check requests towards the selected domains.

Table 4: Most expensive (Top row) and cheapest countries (Bottom row) found in our dataset sorted from left to right.

Expensive Countries	Spain	USA	New Zealand	Portugal	Ireland	Japan	Czech Republic	Korea	Hong Kong	Canada
Cheapest Countries	USA	Spain	Canada	Brazil	Japan	Czech Rebuplic	New Zealand	Australia	Singapore	Thailand

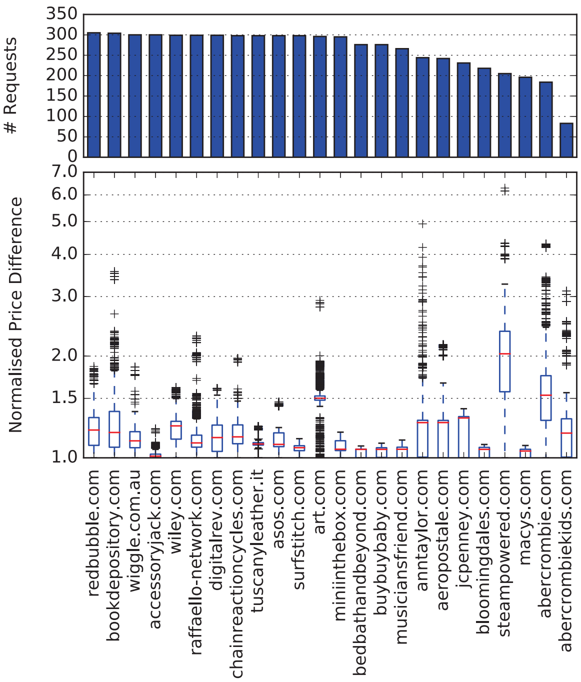


Figure 11: Analysis of the crawled dataset created using peers within Spain. (Top) Domains with the highest number of requests where price difference occurred. (Bottom) Magnitude of normalized price difference per domain.

To prevent the artificial price check requests from influencing our live deployment measurements, we set up a parallel back-end infrastructure to isolate the crawling requests from the live system back-end. The network of PPCs was shared between the two parallel back-ends. That is, the crawling back-end was able to distribute requests to the same PPCs used by the live system and store the results separately.

For the initial systematic crawling we used 24 domains and 30 products per domain. We repeated each experiment 15 times for each of the 30 products per domain yielding 10800 requests. We used 30 IPCs and on average 3 PPCs (these peers reside in the same country)[5] for 10800 requests, yielding 356400 responses.

After assessing the results collected by the systematic crawling alongside the ones from the live system, we look at domains with suspicious price variations between users within the same county, *i.e.*, amazon.com, jcpenney.com, and chegg.com. We consider a price variation to be suspicious if it appears at least 10 times. For each of these 3 domains we create a set of 25 representative products covering multiple distinct categories and all product price ranges (cheap and expensive ones).

As is the case with the above setup, we repeated each experiment 15 times for 25 products from each of the 3 domains. This time we repeated the experiment with the PPCs residing in a new country. We selected 4 European countries (Spain, France, Germany and United Kingdom)[6] resulting to a total of 4500 requests. The 15 experiment repetitions took place in varying times of the day in an attempt to maximize the number of different PPCs used. In the end, from the 30 IPCs plus 3 PPCs we obtained 33×4500 measurement points.

Crawling setup details: To perform the systematic measurement, we use the Mozilla Firefox browser with the iMacros automation add-on installed alongside our custom version of the $heriff add-on. To make our crawling look and behave like a real user so that we evade bot detection and blocking, we created a custom Python driver around Firefox that was able to dynamically create iMacros scripts and load them to the browser. The Python driver injected random delays between requests to mimic a normal human behavior during crawling. Every 4 price check requests, the Python driver reset the Firefox browser to its default state (clean profile) and restarted the process for the rest of the products.

7.2 Analysis on price variation

Figure 11 confirms the results of the live study depicted in Fig. 9. As can be seen, there are e-commerce sites where the maximum price was more than ×4 higher than the minimum price (*e.g.*, anntaylor.com, steampowered.com, and abercrombie.com). Turning to the three e-commerce sites where we observed price difference within the same country, we observe that such differences appear also in our systematic crawling study as shown in Table 5. In all four countries, jcpenney.com has the maximum percentage of requests with price variations between 35 to 70% of the minimum price, followed by chegg.com with the maximum percentage observed in Spain being almost 40%. Finally amazon.com has the lower percentage in all four countries which is below 14%. We investigate each domain in more detail in the next section.

We now compare our results with those in [24]. From the list of reported domains that exhibit price variations, 22.2% were no longer valid, 11.1% of them stopped offering different prices to different locations, 22.2% redirect users to a different URL according to the customer location, and for 44.4% of them we observe that they are still serving different prices across countries. Surprisingly, for those domains we observe that the median price variation across countries is approximately the same (*e.g.*luisaviaroma.com - 1.15%,

[5]The number of PPCs depends on the availability of the real users during the request. The maximum number of PPCs per request was 5.

[6]We intentionally select only European countries to avoid taxation variations between regions within the same country. For more information see "Council Directive 2006/112/EC", which is available at http://eur-lex.europa.eu/eli/dir/2006/112/oj

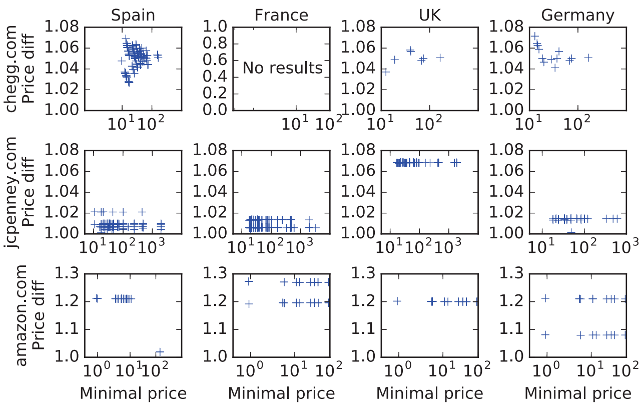


Figure 12: Results per country for the three domains where we detect price variations for users within the same country.

Table 5: Percentage of requests with price difference

	Spain	France	United Kingdom	Germany
chegg.com	38.98%	0.0%	15.44%	2.45%
jcpenney.com	58.62%	67.26%	57.87%	34.72%
amazon.com	6.84%	13.27%	8.79%	7.50%

tuscanyleather.it - 1.12%, abercrombie.com - 1.53%). Some exceptions are overstock.com with a 30% decrease (1.48% reported by [24] *vs.*1.18% in this work) and digitalrev.com with a 6% increase (1.16% reported by [24] *vs.*1.22% in this work).

7.3 Case studies in four countries

For each one of the three e-retailers where we observed price differences within the same country during the live validation phase, we generated ∼ 300 artificial requests for its products and re-routed them through PPC and IPC clients (if one exists in the country). We repeated the experiment for four European countries and depict the results in Fig. 12. Each plot reports on a single retailer within a given country. Each point of a plot refers to a single product: the x-axis indicates the minimum price observed by either a PPC or an IPC. The y-axis indicates the maximum relative price difference between any pair of measurement points, either PPC or IPC, for the given product in the same country.

The total number of results varies based on the number of available IPCs and PPCs within the country. It can be seen that in Spain we have the higher number of results because we have three IPCs located in Spain and the higher number of PPCs. The results clearly indicate the existence of price differences between measurement points even within the same country. The magnitude of difference however is noticeably smaller than that across measurement points in different countries.

In the case of chegg.com, for Spain, the UK, and Germany, we observe a 3% to 7% price difference between distinct PPCs in the same country. Differences exist for products across a range of prices between € 10 and € 100, which are typical prices for textbooks carried by the site. The resulting maximum relative price difference between minimum and maximum measured price is almost uniformly spread between 3% and 7% of the minimum price. In the case of jcpenney.com, price differences within the same country are below 2% in Spain, France, and Germany, and exactly 7% in UK. In the case of amazon.com we observe higher differences in all four countries but they are concentrated on a small set of discrete values, namely 21%, 27%, 19% and 7%. These values match almost perfectly the VAT scales within each one of the four countries.

Discussion. In the previous four case studies we wanted to investigate in detail why we were seeing price differences, consequently we did not use doppelgangers. We did verify, however, that the number of peer requests for any individual PPC were small. In the case of amazon.com, the results seem to indicate that amazon.com is applying the corresponding country VAT based on the category of each product in the country where the user resides. Due to its high penetration, it is likely that several of our PPC users were already logged in with their amazon accounts and thus the prices they were shown included their national tax for the corresponding category. This naturally creates a price difference compared to showing only the base price without tax when one is not sure about the exact delivery address (*e.g.*, in the case of a guest user that is not logged in to the store).

In the case of jcpenney.com, price differences are smaller. Depending on the country, they may be scattered across multiple (*e.g.*, Spain) or few values (France:2, UK:1, Germany:1). We did not detect a connection between these difference and the VAT rates in the said countries. The case of the UK is of particular interest since the price difference seen by different users is not trivial (7%), especially if one takes into consideration that profit margins for each one of these products is typically a small fraction of the actual price of a product. Similar observations apply to chegg.com, where the observed differences are more scattered than in jcpenney.com.

7.4 Testing for bias towards higher or lower prices

In Fig. 13 we focus on jcpenney.com. We plot for each of the PPC users in France (left) and UK (right) the relative price difference with respect to the cheapest PPC user in the same country for the same product, across all the checked products. Each point in the box-plot represents a single PPC user. The number of measurements points that we have for each user is depicted at the labels of the x-axis.The y-axis depicts the range of price differences observed by the same user during the experiment alongside the median difference highlighted in red. This figure is a more detailed view of the corresponding subplots of Fig. 12 where we depicted only maximum price differences.

We can see that the relative differences in France are small (<2%), which is consistent with the earlier presented results. In addition, we can observe that French users obtain both low and high prices in an almost uniform fashion, which does not indicate any clear trend towards high or low values for any of the users. Having excluded all other possible explanations we conclude that the observed price variations are likely to be due to A/B testing that randomly increases or decreases the price by small amounts to gauge how users react.In the right part of Fig. 13 we see that price differences in the UK are higher (∼ 7%). Interestingly, certain peers tend to receive consistently low (first 8 peers) or high (last 2 peers) prices. We attempted to further study the causes of this behavior with respect to those 10 peers. Unfortunately, out of the peers implicated in these results, only few had donated third party tracking cookies and browsing history. We also attempted a regression analysis based on the 400+

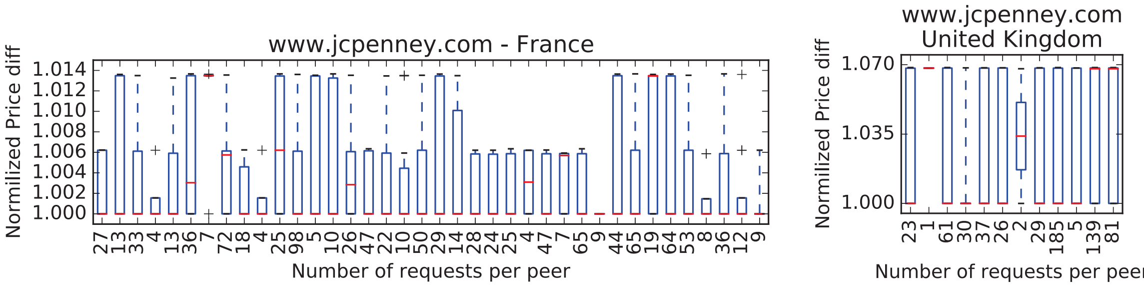


Figure 13: Per peer proxy price difference distribution within the same country. Left: France. Right: UK.

other peers that had donated such data, but the great majority of them were not involved in instances of price variation within the same country.

7.5 Testing for A/B testing and pricing tricks

In this section, we attempt to confirm that the observed price variations are not personal-data-driven, but are instead A/B-testing- or time-driven. To this end, we set up a number of PPCs with an empty profile (no browsing history), operated by us in Spain, and a set of user agents mimicking all possible combinations of popular operating systems and browsers using the phantomJS headless browser [2]. The combinations include Windows 7, Mac OSX and Linux, as well as Google Chrome, Mozilla Firefox and Safari. We use combinations of varying user agents to examine if there is any bias of specific browsers and operating systems towards high prices. For each domain, we randomly selected 30 products. We repeated the experiment for 20 days[7] requesting prices twice a day.

Figure 14 and Fig. 15 depict the price of 5 products from jcpenney.com and chegg.com, respectively over time. For improved readability, we selected 5 representative products out of 30 for each temporal trend we observed in each dataset. The jcpenney.com e-retailer has a diverse inventory of products including clothing, cosmetics, jewelry and household products. We select random products from each of the aforementioned categories during all our experiments. Each box in the figure corresponds to a single product. Observing Fig. 14 from left to right, the first plot corresponds to a refrigerator, the second and third plot to a cosmetic product (Whipped Mud Mask) and a man shaving cream, respectively. The forth plot corresponds to a furniture product (3-seeds living room sofa). Finally, the last plot corresponds to a leather bag. For each day of the experiment (x-axis) we plot the box-plot for all prices observed from each measurement point during that day (y-axis). We annotate every plot with the regression line based on the highest price we observe each day to illustrate the overall price trend (either increasing or decreasing) over time.

Focusing on jcpenney.com and taking This is significantly less than what we observed with jcpenney.com's products. On the other hand, the daily price fluctuation is on average 8.3%, which is 4.6% higher than jcpenney.com. The overall price trend for all 30 products is very similar to the five plots depicted in Fig. 15. Over consecutive days the price is slowly drifting upwards or downwards. Abrupt price changes are rare and at a much smaller scale compared to jcpenney.com's trend.

We now turn our focus on the actual price variation over time. Based on the regression line of each product we estimate a measure of the overall price difference between the first and the last day for all products of each e-store. For jcpenney.com and chegg.com, if we assume that all products we crawled are sold once, we compute an overall €452 and €225 revenue increase, respectively. We tend to believe that this trend of increasing prices is not entirely random. If popular products drift towards higher prices, this may lead to an overall profit increase for the retailer.

We contemplate whether our experiments can influence the product prices. During our measurements we artificially increase the number of visits towards the selected products. An increase in the number of visits may be interpreted by the e-retailers as an increase in demand. Thus it can result in the selected products becoming more expensive. Nevertheless, by analyzing our results we observe that prices become both more and less expensive after successive observations. In addition, consumers typically surf the retailers' inventory jumping between products before doing an actual purchase. Thus, we expect that our measurements introduce a negligible amount of additional visits per product. Furthermore, the number of sales is a substantially more influential signal for pricing compared to the number of visits. The above lead us to the conclusion that the additional visits during our experiments have not influenced the products price over time.

Since in this series of measurements all our PPCs had a clean browsing history, we expected to observe similar prices among them. Indeed, by analyzing the prices for each PPC for each day we do not observe any correlation towards higher or lower prices, similar to Fig. 13. By plotting the cumulative distribution function (CDF) for each PPC and IPC for both experiments in Sect.7.4 and Sect.7.5, we observe an almost equal probability (around 50%) for higher or lower price among all the measurement points. We run a pairwise comparison between all CDFs using the *Kolmogorov-Smirnov test* (K-S test) to examine if the results seen by all of our measurement points (IPCs and PPCs) are drawn from the same distribution. Indeed, the lower D value we observe is 0.3 with all comparisons *p-values* above 0.55. Hence, we conclude that different prices are randomly presented to our PPCs and IPCs with an approximately 50% probability to observe a higher price, which indicates A/B testing.

[7] We present results for 20 days although the experiment was running for 30 days because some products started running out of stock.

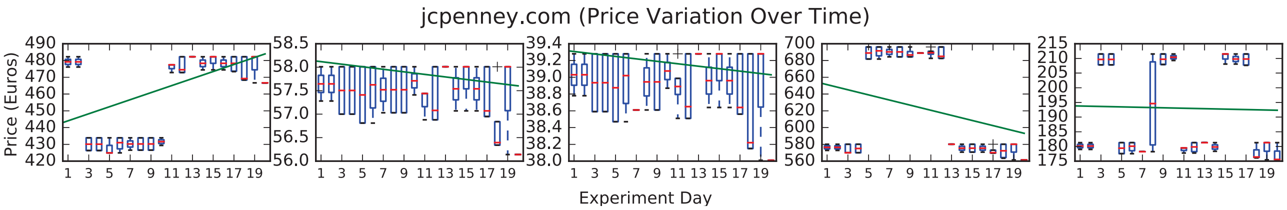


Figure 14: Temporal trends observed for jcpenney.com products.

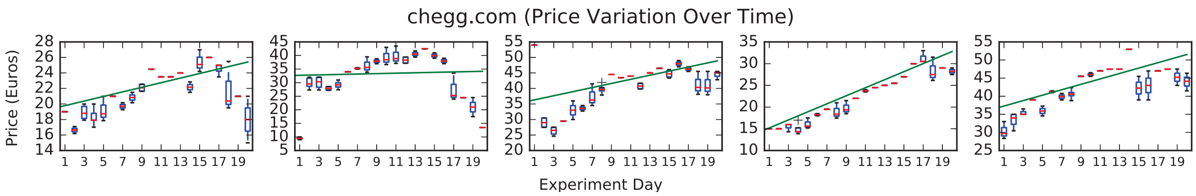


Figure 15: Temporal trends observed for chegg.com products.

Next we attempt to correlate price differences with the type of operating system and browser. Additional features are also introduced, i.e., the time of the day split into quarters and the day of the week. Using linear and multi-linear regression models, we combine the various features but we find no correlation. Our best fit multi-linear regression has an R-Square value equal to 0.431 with all features having p-values greater than 0.05. Next, we perform Random Forests to confirm our conclusions. It turns out that the value of the feature importance factor and the *ROC* is low with no statistical significance for all the features we tried.

Considering the aforementioned results, we reach the conclusion that the two e-retailers under examination do not use personal information to alter their product prices. It is likely that they utilize a combination of A/B testing and a temporal tuning of their products’ prices based on some internal process unknown to us.

7.6 Alexa top-400 e-retailers

Our systematic study has so far been limited to 24 domains in which the live study with real users showed signs of price variation (out of the 1994 domains checked by real users in total). Of those 24 domains, only 3 had price differences within the same country, which in the end we attribute to A/B testing. To answer whether there might exist other domains exhibiting price differences within the same country we examined the top-400 most popular e-commerce sites according to Alexa. For each of these web-sites, we randomly selected 5 products and checked them for 3 consecutive days using the PPCs of Spain. With the exception of the 3 domains already reported, we did not find any additional domains having price differences within the same country.

8 RELATED WORK

The work that is most closely related to ours is the one by Mikians et al. [23, 24]. In [23], they presented the first evidence of online price discrimination using a distributed system that created synthetic requests to examine a handful of popular e-commerce sites from Alexa. In their subsequent (short) paper, they presented initial measurements from a crowdsourced study using the first version of the $heriff add-on for Firefox. Our work with the Price $heriff goes beyond [24] in several important dimensions: a) we develop the first of its kind peer-to-peer measurement system that is capable of privacy-preserving PDI-PD detection, thereby extending [24], which could detect only location-based PD using dedicated servers; b) we present a full system design and implementation addressing important challenges (see Sect. 2) that are not discussed, *e.g.*, scalability, or even faced by [24], *e.g.*, how to tunnel requests through peers in a way that uses their context (history, cookies) without “tainting” it locally or at the server-side, while maintaining user privacy; and c) we present an order of magnitude larger measurement study that, among others, aims to uncover evidence of PDI-PD.

Comparing our results with the work in [24], we observe similar price variation based on location in the domains examined by both this and their work, with only a few exceptions. In addition, our tool was able to examine more than 1900 e-commerce domains as opposed to only 600 domains examined in [24], revealing 76 domains that exhibit evidence of *location based PD*. On the other hand, the measurements in [24] revealed only 20 domains.

Hannak et. al. [20] have followed upon the work of Mikians et al., presenting additional evidence that online PD exists on the web. They developed a research prototype that targets specific preselected web pages with a barrage of price comparison tests that were conducted with the help of users recruited through Amazon’s Mechanical Turk [1]. The authors manage to extract some user features suspected of triggering discrimination by using artificially created personas with different characteristics. The Price $heriff, like [20], uses crowdsourcing, but enables the users to check arbitrary rather than predefined web-sites and products.

We also attempted to compare our results with the results reported by Hannak et al. [20]. The only domains examined by both us and Hannak et al. were jcpenney.com and macys.com. The authors did not further investigate those two domains due to their observed price differences being below their 0.5% threshold. Therefore we cannot compare with those results.

Vissers et al. [29] have crawled the prices of 25 airlines for a period of 3 weeks looking for signs of on-line price discrimination.

Their approach is domain-specific and has relied upon simulating (playing back) real users profiles. Despite the large number of anecdotal reports about price discrimination in airline pricing, the authors could not confirm it for this category of product.

In the economics literature, there is a large body of work on online price discrimination but it is mostly theoretical with little empirical validation. In a recent study, Sinkinson and Seim [28] used empirical data to look into mixed pricing strategies by large office supply chains in the US. Their measurements confirm the existence of location-based discriminatory practices. They also reveal that such strategies are often "mixed", *i.e.*, with a level of randomisation, which is also confirmed by our measurements in Sect. 7.4.

9 CONCLUSION

The Price $heriff is a first-of-its-kind application designed from the ground up to help users detect instances of price discrimination on the Internet. In this paper we have attempted to communicate the difficult challenges involved in the development of such a system. To the best of our knowledge, the *Price $heriff* is the first distributed system for observing the content of web pages from multiple vantage points to detect differentiation based on location and personal data. To address this challenge, we designed and implemented novel concepts, such as a hybrid infrastructure/P2P architecture for comparing e-store prices, profile pollution prevention, and privacy-preserving profile sharing for the creation of doppelganger profiles via a k-means computation cryptographic protocol. We envision that our architectural and implementation choices will inform the design of future crowdsourced services for tackling various types of discrimination. Having completed the development phase of the project, our future work will primarily focus on expanding our user base, collecting more data from real users, and analyzing them in greater detail in order to understand whether personal-data induced price discrimination is taking place anywhere on the Web. Our findings of price variation within the same country and the constant bias of some peers towards higher or lower prices leaves such a possibility open.

Acknowledgments We thank the anonymous reviewers and our shepherd David Choffnes for their thoughtful reviews and suggestions. The work of Costas Iordanou was fully supported by the EU ITN-METRICS FP7 project (Grant agreement No.:607728)

REFERENCES

[1] 2015. AMT - Amazon Mechanical Turk . https://www.mturk.com. (2015).
[2] 2015. PhantomJS - Headless Web Browser. http://phantomjs.org/. (2015).
[3] 2016. MySQL, The world's Most Popular Open Dource Database. Software available at:. http://dev.mysql.com/downloads/mysql/. (2016).
[4] 2017. Google Chrome Extension JavaScript APIs. https://developer.chrome.com/extensions/api_index. (2017).
[5] 2017. Monzilla Web Extension JavaScript APIs. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API. (2017).
[6] 2017. The peerJS library. http://peerjs.com/. (2017). Version: 0.3.14, Last accessed: 15-06-2016.
[7] 2017. Price Optimization Strategies. http://www.pros.com/solutions/price-optimization-software. (2017).
[8] 2017. Round-Robin Scheduling. https://en.wikipedia.org/wiki/Round-robin_scheduling. (2017).
[9] 2017. SeleniumHQ - Browser Automation. http://www.seleniumhq.org/. (2017).
[10] 2017. Telsa GPU Accelerator for Servers. http://www.nvidia.com/object/tesla-servers.html. (2017).
[11] 2017. The Onion Router. https://www.torproject.org/. (2017).
[12] 2017. WebRTC, Real-Time Communication for Browsers and Mobile Applications. http://www.webrtc.org/. (2017).
[13] Michel Abdalla, Florian Bourse, Angelo De Caro, and David Pointcheval. 2015. Simple Functional Encryption Schemes for Inner Products. In *Proc. of Public-Key Cryptography (PKC)*. 733–751.
[14] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. 2014. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 674–689.
[15] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. 2013. FPDetective: Dusting the Web for Fingerprinters. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*. 1129–1140.
[16] Solon Barocas. 2014. Data Mining and the Discourse on Discrimination. In *Proc. of Data Ethics Workshop of KDD*.
[17] Assaf Ben-David, Noam Nisan, and Benny Pinkas. 2008. FairplayMP: a System for Secure Multi-party Computation. In *Proc. of ACM Computer and Communications Security (CCS)*. 257–266.
[18] Le Chen, Alan Mislove, and Christo Wilson. 2016. An Empirical Analysis of Algorithmic Pricing on Amazon Marketplace. In *Proceedings of the 25th International World Wide Web Conference (WWW 2016)*. MontrÃĺal, Canada.
[19] Taher El Gamal. 1984. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *CRYPTO*. 10–18.
[20] Aniko Hannak, Gary Soeller, David Lazer, Alan Mislove, and Christo Wilson. 2014. Measuring Price Discrimination and Steering on E-commerce Web Sites. In *Proc. of USENIX/ACM Internet Measurement Conference (IMC)*.
[21] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. 2010. TASTY: Tool for Automating Secure Two-party Computations. In *Proc. of ACM Computer and Communications Security (CCS)*. 451–462.
[22] Alan S. Manne. 1960. On the Job-Shop Scheduling Problem. *Operations Research* 8, 2 (1960), 219–223.
[23] Jakub Mikians, László Gyarmati, Vijay Erramilli, and Nikolaos Laoutaris. 2012. Detecting Price and Search Discrimination on the Internet. In *Proc. of Workshop on Hot Topics in Networks (HotNets)*.
[24] Jakub Mikians, László Gyarmati, Vijay Erramilli, and Nikolaos Laoutaris. 2013. Crowd-assisted Search for Price Discrimination in e-Commerce: First Results. In *Proc. of Conference on Emerging Networking Experiments and Technologies (CoNEXT)*.
[25] Andrew Odlyzko. 2003. Privacy, economics, and price discrimination on the Internet. In *Proc. International Conference on Electronic Commerce (ICEC)*.
[26] Fotios Papaodyssefs, Costas Iordanou, Jeremy Blackburn, Nikolaos Laoutaris, and Konstantina Papagiannaki. 2015. Web Identity Translator: Behavioral Advertising and Identity Privacy with WIT. In *Proc. of Workshop on Hot Topics in Networks (HotNets)*. ACM.
[27] Peter J. Rousseeuw. 1987. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.* 20 (1987), 53 – 65. http://www.sciencedirect.com/science/article/pii/0377042787901257
[28] Michael Sinkinson and Katja Seim. 2015. Mixed Pricing in Online Marketplaces. *Work in Progress*, http://assets.wharton.upenn.edu/~msink/mixed_pricing.pdf. (2015).
[29] Thomas Vissers, Nick Nikiforakis, Nataliia Bielova, and Wouter Joosen. 2014. Crying Wolf? On the Price Discrimination of Online Airline Tickets. In *Proc. of Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs)*.

10 APPENDIX

10.1 Supplement to Sect. 2

$heriff satisfies the following requirements:

- It is user-friendly. The user can get price results after a few simple to understand mouse clicks.
- Generic product price extraction algorithm that works on a large number of e-stores.
- Location-based price monitoring. The tool is able to collect prices from distinct locations worldwide.

$heriff does not fulfill the following important requirements satisfied by Price Detective:

- Its single Measurement server architecture causes scalability concerns.
- It is not able to fetch product prices from other real users within the same city or country (while preventing peer state pollution and preserving privacy), making it unsuitable for detection of PDI-PD based on client-state or fingerprinting.

- It is not able to collect the browsing history from the users, rendering it unable to detect PDI-PD based on the sites the user has visited.
- The product prices are shown in different currencies in the result page. Hence the user faces difficulty in observing price differences.

10.2 Supplement to Sect. 3.1

10.2.1 Measurement server. After the initial analysis of the base system, we realized that the measurement server was the bottleneck, introducing long response times during demand spikes for price checks. A single measurement server was responsible to process all incoming requests, to distribute a number of jobs to available proxy servers, to record the proxy servers' results to the database and replies back the results to the user. To overcome the scalability issue, and make the system more responsive, we managed to slim down the Measurement server by removing functionality and assigning it to the Coordinator and Database server. We removed the integrated database from the Measurement server and we host it on a separate Database server, shared between multiple measurement servers. Furthermore, we optimized existing parts of the code to reduce the overall response time while allowing the price check request distribution protocol to operate smoothly.

By offloading some of the system's functionalities (request handling and data storage) to the Coordination and the Database server, we ensure that the Measurement server's CPU and memory footprint is kept small. Measurement servers can be added and removed from the system dynamically depending on workload and maintenance needs. The process for attaching and detaching machines to and from the system is simple for the system's administrator who uses an intuitive web interface. First, one needs to setup a new Measurement server [8]. Then, she needs to register it with the system by using the Coordinator's web interface. The Coordinator executes some internal tests to confirm that the new machine is actually running the Measurement server code. If the new machine passes the tests, the Coordinator includes it in the request distribution protocol and starts using it to serve price check requests. To remove a Measurement server, one can use the same web interface. As soon as the selected Measurement server has no pending jobs, it can be removed from the system.

For the dedicated centralized database, we tweak the configuration file to increase the number of connection threads and keep them in memory. This optimization minimizes the time required to create a new connection to the database. Furthermore, by keeping the connections in memory, we managed to minimize the database response time during multiple sequential connection requests from multiple measurement servers. During a price check request, the delay for a proxy client's response depends on the overall system workload, Internet connection speed or requested web page size. Thus, multiple connections need to be utilized simultaneously during workload spikes for a relatively large time window of approximately one minute.

Another optimization is the use of stored procedure calls. Stored procedures can be used to optimize frequently used queries and minimize the data transfer between the measurement servers and the database. At the operating system (OS) level, since the server was only hosting the database, we also tweaked the OS to assign more hardware resources to MySQL process, such as, memory space and CPU time.

[8] Source code and installation scripts are currently available for Ubuntu Linux 12.04 LTS

Peer ID	IP	Country	Region	City	Select
SQN9cSHiZA7o_1	195.235.92.38	Spain	Catalonia	Barcelona	
N6esch7nbxdk	81.38.218.228	Spain	Catalonia	Barcelona	
om0V6hjHBj7x_3	195.235.92.38	Spain	Catalonia	Barcelona	
SQN9cSHiZA7o_12	195.235.92.38	Spain	Catalonia	Barcelona	
SQN9cSHiZA7o_10	195.235.92.38	Spain	Catalonia	Barcelona	
costasWorker	81.38.218.228	Spain	Catalonia	Barcelona	SELF

Figure 16: The Price Detective peer proxy monitoring panel. For each online peer we can monitor the IP, Country, Region and City, in real time.

10.2.2 Network of Peer Proxy Clients. Within the Price Detective add-on, we use the *peerjs* [6] webRTC library to directly connect with the add-on peers. We also developed a communication protocol on top of webRTC that enables the exchange of different instruction messages between the peers and the Measurement Servers. Each peer client has a unique ID, which the systems uses to track it.

The Coordinator monitors the number of available peers in real time with a low overhead. We have implemented a web interface that retrieves pertinent information from the Coordination server such as the number of available peer proxies along with their IP, peer ID, country and city. A screen shot of the web interface is presented in Fig. 16.

10.3 Supplement to Sect. 3.4

Below we provide a short explanation of the key fields located at the bottom of Fig. 6:

Time-stamp: Each Measurement server sends heartbeat messages to the Coordinator. When the coordinator server receives a heartbeat message, it updates the time-stamp field of the corresponding measurement server. Absence of heartbeat messages for a specified time threshold results in the Measurement server being marked as offline.

Jobs: The jobs field is a simple counter that holds the number of jobs that each measurement server processes at any given time. The Coordinator uses this field to choose a measurement server to assign a new price check request.

URL, Port: The Coordinator forwards these two fields to the browser add-on after the coordinator server selects a measurement server to process the incoming request.

For the price check request distribution protocol, during step 4, the Measurement server responds back to the Coordinator using the unique job ID to inform it that the specific request has been completed, so that the Coordinator decreases the corresponding job counter. For step 4, we also consider the case when the Coordinator is not informed of the job completion due to network issues. Appropriate corrective measures are put in place to handle this case. Furthermore, the information kept at the Coordinator facilitated

Aggregator		Server
$\alpha, (\beta_i)_{i\in[t]}$		$\mathbf{a} = a_1 \dots, a_m$
		$\mathbf{s} = 1, \sum_{i\in[m]} b_i^2, -2b_1, \dots, -2b_m$ $\mathbf{S} = \sum_{i\in[t]} x_i s_i$
	$\xleftarrow{\mathbf{S}}$	
$r \leftarrow_\$ Z_p^*$ For $i \in [t]$ $\epsilon_i = (\beta_i)^r$		
	$\xrightarrow{(\epsilon_i)_{i\in[m]}}$	
		For $i \in [t]$ $\epsilon_i^* = (\epsilon_i)^{s_i}$
For $i \in [t]$ $\epsilon_i^{**} = (\epsilon_i^*)^{1/r}$ $\gamma = \mathtt{Dec}_{\mathbf{S}}(\alpha, (\epsilon_i^{**})_{i\in[t]})$ $d = \mathtt{DL}(\gamma, p)$		

Figure 17: Interaction between a client, the aggregator, and the server to compute the distance between two points.

Aggregator		Server
$(\alpha_j, (\beta_i)_{i\in[m]})_{j\in[n]}$		$\mathbf{x}$
$\rho = \prod_{j=[n]} \mathbf{c}_{j,0}$ For $i \in [3, t]$ $\varrho_i = \prod_{j\in[n]} \mathbf{c}_{j,i}$		
	$\xrightarrow{\rho, (\varrho_i)_{i\in[3,t]}}$	
		$\mathbf{w} = \mathtt{Dec}_{\mathbf{x}}(n, \rho, (\varrho_i)_{i\in[3,t]})$ For $i \in [1, m-2]$ $s_i = \mathtt{DL}(w_i/n, p)$

Figure 18: Centroid update. Note that decryption at the server only uses the private keys at positions in $[3, m]$.

the implementation of a real-time web interface for monitoring and managing the system.

10.4 Supplement to Sect. 3.8

System setup generates the description of a multiplicative group G of order q where Decisional Diffie-Hellman is hard, and a generator g of G. Key generation outputs an m-dimensional vector of secret keys $\mathbf{x} = (x_i)_{i\in[m]}$ and a vector of corresponding public keys $\mathbf{h} = (h_i)_{i\in[m]}$ where $h_i = g^{x_i}$. Encryption of vector $\mathbf{c} = (c_i)_{i\in[m]}$ under public key $\mathbf{h}$ is denoted by $\mathtt{Enc}_{\mathbf{h}}(\mathbf{c})$ and outputs $\alpha = g^r, (\beta_i = h_i^r g^{c_i})_{i\in[m]}$ for random r in Z_p^*. (Note that operations are $\mathtt{Dec}_{\mathbf{x}}(\alpha, (\beta_i)_{i\in[m]})$ and outputs $(\gamma_i)_{i\in[m]}$ where $\gamma_i = \beta_i/\alpha^{x_i}$. Note that because encryption is at the exponent, recovering the original plaintext requires computing the discrete logarithm of γ_i in base p — this operation is feasible if the range of admissible cleartexts is small.

The holder of the private keys can compute and outsource the function key $\mathbf{f} = \sum_{i=1,m} x_i s_i$ for a (private) vector $\mathbf{s} = (s_1)_{i\in[m]}$. Given an encryption of $\mathbf{c}$ as $\alpha = g^r, (\beta_i = h_i^r g^{c_i})_{i\in[m]}$, the holder of the function key can evaluate the dot-product between $\mathbf{c}$ and $\mathbf{s}$ by computing $\gamma = \prod_{i=1,m} \beta_i^{s_i}/\alpha^{\mathbf{f}}$ and then finding the discrete logarithm of γ in base p. Note that computation of the predicate does not require knowing $\mathbf{c}$, but only an encryption of $\mathbf{c}$ under public key vector $\mathbf{h}$.

10.4.1 Distance computation and client-cluster mapping. At system setup, the Coordinator generates parameters G, q, g as described above, picks $t = m + 2$ private keys $\mathbf{x} = (x_i)_{i\in[t]}$ and publishes the corresponding public keys $\mathbf{h} = (h_i)_{i\in[t]}$ where $h_i = g^{x_i}$.

The client uses its private point $\mathbf{a} = (a_i)_{i\in[m]}$ to compute $\mathbf{c} = \sum_{i=[m]} a_i^2, 1, a_1, \dots, a_m$, encrypts $\mathbf{c}$ as $\alpha, (\beta_i)_{i\in[t]} = \mathtt{Enc}_{\mathbf{h}}(\mathbf{c})$, sends the ciphertext to the Aggregator, and goes offline.

The online part of the protocol to privately compute the distance between an encrypted client point and a cluster centroid is depicted in Fig. 17. In particular, the Aggregator acts as the client of [13] and uses the encrypted client point as its private input; the Coordinator acts as the server of [13] and uses a cluster centroid $\mathbf{b}$ as its private input.

The protocol of Fig. 17 must be repeated for each of the k cluster centroid held by the Coordinator. (All k instances could run in parallel.) At the end of the k runs, the Aggregator learns the distance between all centroids and $\mathbf{c}$, so it can assign the client to the correct cluster. The Aggregator, however, does not learn the actual point $\mathbf{c}$, nor it learns the cluster centroids.

10.4.2 Cluster center update. The protocol that allows the Coordinator to update a cluster centroid is depicted in Fig. 18. Let n be the number of clients assigned to a given cluster. We now slightly update the notation and use subscript j to denote a specific client. For example, $\alpha_j, (\beta_{i,j})_{i\in[t]}$ is the encrypted point of client j. The Aggregator aggregates dimension-wise all the encrypted client points that belong to the cluster. In particular, aggregation is only needed for elements at positions in $[3, t]$ since the first two dimensions are artificially added and do not reflect the browsing history of a client. The aggregated ciphertext is forwarded to the Coordiantor along with the cluster cardinality. The Coordiantor simply decrypts to learn the (dimension-wise) sum of all points, divides each sum by the cluster cardinality, and computes the discrete logarithm to get the new cluster centroid.

10.4.3 Security Analysis. The security of the above protocols stems from the security of [13] and the one of the ElGamal encryption scheme. For the protocol of Fig. 17, the Aggregator learns the distances between a client's point and k centroids belonging to the Coordinator. The Aggregator, however, does not learn any of those points. Furthermore an encrypted client point cannot be decrypted as long as the Coordinator and the Aggregator do not collude. The security of the protocol in Fig. 18 relies on the security of the ElGamal encryption scheme. The Aggregator simply aggregates ciphertexts while the Coordinator only learns the dimension-wise sum and the cardinality of the cluster.

10.5 Implementation

Browser add-on. The add-on implementation consists of 5 modules: a) the View module, which is responsible for all visual elements and user preferences; b) the Collector module, which is responsible to detect third party domains, create the HTML tag graph and handle the price request between the user, the Coordinator and the Measurement server; c) the Peer handler module, which implements the P2P communication protocol using webRTC [12]; d) 4. the Sandbox module that handles the incoming remote URL requests, installs doppelgangers, and cleans up the user's profile; and e) the Controller module, which manages the interaction between all modules. The browser add-on is implemented using standard web development languages and established third party libraries, such as peerjs [6]. The Chrome and the Firefox add-ons are written in 1800+ and 2300+ LoC, respectively.

Coordinator. This component consists of: a) three monitoring subsystems for the Measurement servers, the PPCs, and the dedicated doppelganger clients; b) the request distribution protocol to balance the load among the Measurement servers; c) the network of PPCs coordination and contact information distribution protocol; and d) the doppelganger creation and management subsystem. It was implemented using PHP, HTML/CSS, JavaScript, Python and nodejs. It is written in 5100 LoC.

Measurement server. The Measurement server comprises five modules: a) the Currency Detection and Conversion module; b) the DiffStorage module to minimize the size of HTML code we store in the RDBMS by saving the full HTML page code reported by the user's add-on and just saving the difference for the HTML code responses from the IPCs and PPCs; c) the PhantomJS wrapper module, which handles the requests to the IPCs; d) The Tag Graph module, which extracts the price from the HTML code; and e) the PDSystem module, which coordinates all the interaction between the modules and serves the results to the user. It is implemented in 7400+ LoC of HTML/CSS, PHP and Python.