# BrowserFlow: Imprecise Data Flow Tracking to Prevent Accidental Data Disclosure

Ioannis Papagiannis[*]
Facebook
yiannis@fb.com

Pijika Watcharapichat
Imperial College London
pw610@imperial.ac.uk

Divya Muthukumaran
Imperial College London
dmuthuku@imperial.ac.uk

Peter Pietzuch
Imperial College London
prp@doc.ic.ac.uk

## ABSTRACT

With the use of external cloud services such as Google Docs or Evernote in an enterprise setting, the loss of control over sensitive data becomes a major concern for organisations. It is typical for regular users to violate data disclosure policies accidentally, e.g. when sharing text between documents in browser tabs. Our goal is to help such users comply with data disclosure policies: we want to alert them about potentially unauthorised data disclosure from trusted to untrusted cloud services. This is particularly challenging when users can modify data in arbitrary ways, they employ multiple cloud services, and cloud services cannot be changed.

To track the propagation of text data robustly across cloud services, we introduce *imprecise data flow tracking*, which identifies data flows implicitly by detecting and quantifying the similarity between text fragments. To reason about violations of data disclosure policies, we describe a new *text disclosure model* that, based on similarity, associates text fragments in web browsers with security tags and identifies unauthorised data flows to untrusted services. We demonstrate the applicability of imprecise data tracking through BrowserFlow, a browser-based middleware that alerts users when they expose potentially sensitive text to an untrusted cloud service. Our experiments show that BrowserFlow can robustly track data flows and manage security tags for many documents with no noticeable performance impact.

## Keywords

Data disclosure; browser-based middleware, data tracking; cloud security

[*]Work performed while at Imperial College London

*Middleware'16, December 12 - 16, 2016, Trento, Italy*

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4300-8/16/12...$15.00

DOI: http://dx.doi.org/10.1145/2988336.2988345

## 1. INTRODUCTION

Many organisations use their own web-based applications for internal communication, content management, reporting and business processes [12], but cloud services, especially ones according to a software-as-a-service (SaaS) model, have gained rapid adoption in enterprise settings. As previous studies have shown [24], enterprise users favour consumer cloud services, such as Google Docs [25], Microsoft Office 365 [42] and Zoho [70], over in-house solutions due to their familiarity and ease-of-use. As a consequence, users in their web browsers employ external cloud services alongside internal services that may contain sensitive enterprise data.

This creates a major headache for IT departments that, for legal and competitive reasons, must enforce *data disclosure policies* that mandate how sensitive data is permitted to propagate between services. With browser-based services, however, users can easily move data between internal and external services, e.g. by copying and pasting text from an internal document to one hosted on Google Docs.

IT departments often respond to this loss of control by limiting users to a small set of "approved" cloud services with built-in safeguards. Users, however, bypass such restrictions—Frost and Sullivan [24] report that over 80% of employees admit to using unapproved SaaS applications concluding: "Rather than attempt to restrict usage, the goal should be to enable the freedom employees need to do their jobs better, without compromising company security and liability."

In this paper, our goal is to provide an automated approach that helps users comply with data disclosure policies, e.g. we want to advise users when it is not safe to disclose text on Google Docs. Typically enterprise users want to safeguard their organisation's data, and most data disclosure happens by accident when users do not realise the results of their actions [8]. Therefore we want to inform employees of potential policy violations but give them the freedom to make final disclosure decisions. Such an advisory model does not interfere with existing business workflows and is thus more likely to gain adoption, in particular, compared to mandatory restrictions.

Client-side data flow tracking across cloud services raises multiple challenges: (i) it is difficult to *track data flow robustly* when users can copy and modify text in arbitrary ways, including transferring it to an external application outside the browser. Prior approaches for data flow tracking [23, 35] cannot reason about such data flows without a system-wide, closed-world deployment; (ii) the tracking must also account for *decreased information disclosure*, e.g. if text is modified to the point at which it bears no resemblance to the source text, it becomes safe to disclose. Existing data flow tracking approaches suffer from false positives in this case; and (iii) it is unclear at what *granularity* data flow tracking should occur. In different contexts, the disclosure of a whole document, multiple paragraphs or individual sentences from different documents may constitute a policy violation.

We address the above challenges by combining two ideas: (i) data disclosure in the above setting can be modeled as a *decentralised information flow control* (DIFC) problem [48]—administrators set default data flow policies but employees may declassify data as they see fit; and (ii) data flow between cloud services can be tracked robustly using text similarity matching [56]. Text similarity analysis can discover data disclosure even when the data flow that led to the disclosure is hidden from the tracking system.

We describe BrowserFlow, a practical client-side middleware for tracking the propagation of unstructured text data across cloud services and advising users about violations of data disclosure policies. Administrators specify an enterprise-wide data disclosure policy according to a *text disclosure model*, which BrowserFlow enforces through an *imprecise data flow tracking technique*:

**Text Disclosure Model.** To define a data disclosure policy, each cloud service is assigned a *confidentiality* and a *privilege* label. Text segments first observed in a given cloud service are assigned that service's confidentiality label. When text with a confidentiality label is observed in a service with an incompatible privilege label, BrowserFlow generates a warning. A user can then explicitly permit the text propagation by changing the label, or BrowserFlow intercepts the data transfer to the cloud service.

**Imprecise Data Flow Tracking.** A key difference to prior data flow tracking systems is that BrowserFlow does not attach labels explicitly to the data, which would require modifications to binaries and incur a high cost at runtime [35, 49]. Instead, to maintain these labels, BrowserFlow tracks data flow *implicitly*: when a new text segment appears in a document, BrowserFlow generates a fingerprint [56] that represents its similarity to other text, and checks if a similar fingerprint was seen previously. If so, the confidentiality label of the new text segment is modified to reflect its origin.

Imprecise data flow tracking has two benefits: (1) positive data flows are reported only while the text maintains significant resemblance, which avoids false positives after the text was modified sufficiently; and (2) there is no need to track data flow in external applications such as text editors, which would come at a runtime performance cost [35, 68] and would be impractical in production environments.

We implemented a prototype version of BrowserFlow as a plug-in for the Google Chrome web browser, which supports Google Docs and other form-based cloud services. Our evaluation shows that, across a range of different datasets, BrowserFlow correctly identifies information disclosure using imprecise data flow tracking. It also makes disclosure decisions with negligible impact on user-perceived performance because they occur asynchronously to the main request processing in the browser.

The rest of the paper begins with the problem definition and threat model in §2. We describe our model for text disclosure in §3, and introduce our approach for imprecise data flow tracking in §4. §5 reports on implementation details of BrowserFlow, in particular our experience of its integration with current cloud services. We present our evaluation results, both in terms of effectiveness and performance overhead, in §6 and conclude in §7.

## 2. BACKGROUND

Our problem statement considers employees in an enterprise setting accessing and modifying data via web-based services with a web browser acting as the client. The services handle unstructured text data, e.g. following a document-centric model as found in Google Docs, Evernote or Zoho. For example, Figure 1 shows a user's browser instance with three application tabs, used as part of a workflow when interviewing job applicants. Two of the applications, Interview Tool and Wiki, are internally-hosted and may process sensitive text. The user also relies on Google Docs, which is untrusted.

The organisation's IT department may define a data disclosure policy that imposes various restrictions on data propagation. For example, it may dictate that transferring text from the internal Wiki to the Interview Tool is permitted, but not the reverse. In addition, data from either of the two internal applications must not propagate to Google Docs. If the employee transfers a text fragment from the Wiki tab to Google Docs, and it is uploaded to a remote Google server, it would constitute a policy violation.

### 2.1 Threat model

Based on conversations with companies, we adopt a practical threat model in which users are considered to be *trusted-but-careless* when handling sensitive documents [60]. For example, company employees are typically not malicious because they are bound contractually by confidentiality agreements, and violations may incur legal repercussions. Nevertheless a user may unintentionally disclose data during a business workflow. We believe that such a threat model realistically captures the risk of data disclosure in enterprise environments with external cloud services [8].

In the above scenario, an interviewer may accidentally copy a candidate evaluation from the Interview Tool to the internal Wiki, which is accessible by all employees. Another user who uses Google Docs for collaborative document editing may paste confidential interviewing guidelines from the internal Wiki to a Google Docs document and share this document with an external client. Both data flows would constitute significant, yet unintended violations of the organisation's data disclosure policy.

The text may also be modified during propagation. For example, the interviewer may copy an edited version of the candidate evaluation to the internal Wiki. They may remove some sentences, rephrase others or alter the order in which sentences appear in the final text. As long as the text fragment has a given level of similarity to the original text in the source document, it violates the data disclosure policy.

We do not assume that users or cloud service providers are malicious and actively try to disclose data. This is an orthogonal issue, which should be handled by access control restrictions and governed by agreements based on trust in third-party cloud services.

We assume that the IT department of an organisation is able to install software in all devices used by employees and that employees do not tamper with the software installed.

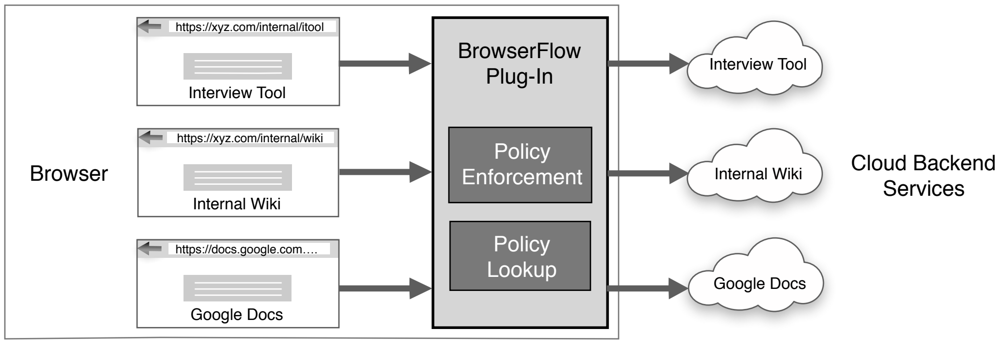


**Figure 1: Overview of the BrowserFlow web browser plug-in**

This capability is typically obtained as part of a device provisioning process enforced in most organisations even for employee-owned devices.

## 2.2 Existing approaches

Preventing unauthorised data disclosure has led to solutions in a number of research areas, each with specific limitations when applied to the above enterprise scenario.

**Data leakage prevention (DLP) systems** [10,13,41,47,61] protect sensitive data on client endpoints by inspecting outgoing network traffic to prevent confidential data from leaving an organisation's network. Implementations range from application-level firewalls [6,16], which monitor outgoing network traffic for confidential files, to specialised solutions [47], which employ text similarity techniques to detect information disclosure in network streams.

Different from the network focus of existing DLP approaches, BrowserFlow prevents disclosure of text data in web browsers. Its Text Disclosure Model combined with imprecise data flow tracking permits it to reason about the transitive propagation of data across multiple cloud services. Its realisation as part of a web browser means that it does not require reverse-engineering of network protocols (see §5).

**Data flow tracking systems** [9,17,22,35,53,57,67–69] attach labels to data, which is tracked while used by programs. Data derived from other sensitive data inherits the corresponding security label. Such *precise* data flow tracking systems are typically used to analyse applications handling short confidential data such as passwords [23]. Precise data flow tracking systems detect data flow accurately but suffer from substantial performance overhead [35,49]. More fundamentally, data can only be tracked if all accesses are intercepted [35], and tracking can be evaded through implicit flows [4,5,54].

Given these limitations, it is not surprising that, despite the large body of research on precise data flow tracking, such approaches are not widely used in practice. Imprecise data flow tracking can be regarded as a practical alternative that uses text similarity to establish a robust link between data and security policies specified as labels.

**Static data flow analysis** [1,3,48,55,64] tracks the data flow of the source code using program analysis. Since this leads to conservative results, the accuracy can be improved by augmenting the programming language types with security labels [55], which is infeasible for legacy programs [14,29]. Static techniques also cannot express disclosure policies that make runtime decisions [15] unless combined with dynamic data flow analysis [44]. While BrowserFlow is a runtime approach, its label model for specifying policy shares the simplicity of ones first suggested for static data flow analysis [48].

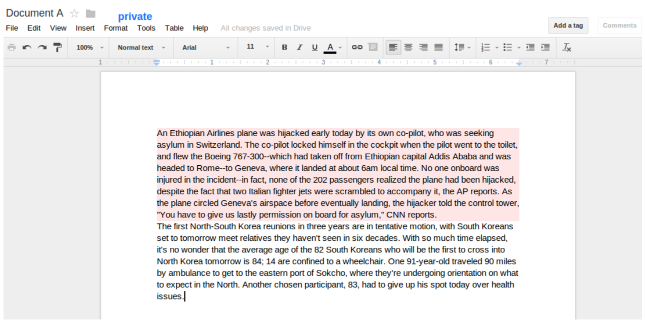
**Browser-side enforcement.** There is prior work on preventing confidential data disclosure in cloud services [18,31]. A common assumption is that external cloud services are untrusted, and therefore all data must be encrypted prior to upload to an external service [28,51]. This is often infeasible, however, because services may need to index, search, and inspect the original data. Data encryption also becomes impractical when cloud services support collaborative editing with users outside of an organisation or when users require access to documents on personal devices. BrowserFlow provides more flexibility: users can use external applications freely as long as they do not disclose sensitive data.

**Client-side middleware** [32,45,50] protects the confidentiality of user's data from the untrusted cloud service providers by transparently managing data encryption (or obfuscation) between user applications and cloud. Since data protection is completely decoupled from application logic, client-side middleware can be developed independently, is compatible with legacy applications, and can be customised according to organisational requirements.

COWL [59] and BFlow [66] allow untrusted JavaScript to process confidential data without risking disclosure. FlowFox [26] tracks data flows by executing JavaScript in an isolated environment using multi-execution [19]. These (and other similar [39, 43, 54]) approaches target either violations of the same-origin or intra-origin policy [20] within browsers. Mowbray et al. [46] describe how policies can be integrated with a privacy manager for controlling data protection. BrowserFlow does not address vulnerabilities in cloud services but instead protects against inadvertent data disclosure by users themselves.

## 3. BrowserFlow DESIGN

We give an overview of BrowserFlow in Figure 1. BrowserFlow intercepts data from browser tabs before it is sent to the remote servers. It is composed of two modules: (i) a *policy lookup* module extracts the security label associated with the text segment being uploaded; and (ii) a *policy enforcement* module uses the security label to reason about the compliance of the data propagation in relation to the


Document A
private
File Edit View Insert Format Tools Table Help
All changes saved in Drive
Add a tag
Comments
Normal text
Arial
11

An Ethiopian Airlines plane was hijacked early today by its own co-pilot, who was seeking asylum in Switzerland. The co-pilot locked himself in the cockpit when the pilot went to the toilet, and flew the Boeing 767-300--which had taken off from Ethiopian capital Addis Ababa and was headed to Rome--to Geneva, where it landed at about 6am local time. No one onboard was injured in the incident--in fact, none of the 202 passengers realized the plane had been hijacked, despite the fact that two Italian fighter jets were scrambled to accompany it, the AP reports. As the plane circled Geneva's airspace before eventually landing, the hijacker told the control tower, "You have to give us lastly permission on board for asylum," CNN reports.

The first North-South Korea reunions in three years are in tentative motion, with South Koreans set to tomorrow meet relatives they haven't seen in six decades. With so much time elapsed, it's no wonder that the average age of the 82 South Koreans who will be the first to cross into North Korea tomorrow is 84; 14 are confined to a wheelchair. One 91-year-old traveled 90 miles by ambulance to get to the eastern port of Sokcho, where they're undergoing orientation on what to expect in the North. Another chosen participant, 83, had to give up his spot today over health issues.

**Figure 2: Disclosure decision by BrowserFlow for Google Docs**

organisation's data disclosure policy. BrowserFlow then takes appropriate action, either permitting the data upload or preventing it, e.g. by encrypting the data before transmission.

As shown in Figure 2, BrowserFlow informs the user of a cloud service about the result of the disclosure decision by changing the background colour of an affected text segment, such as a paragraph. While a user edits a paragraph, the paragraph is marked with a red background when it discloses sensitive data from another source.

## 3.1 Text Disclosure Model

We express the actions of copying text from one browser tab to another as a data flow in a *Text Disclosure Model* (*TDM*). Data disclosure policies are specified using a decentralised label model [14, 37, 48]. Policies are set by enterprise-wide administrators once, but they may be refined by users.

In the TDM, *security labels* are associated with text in documents and with cloud services. A label consists of a set of *tags*. Each tag is a unique, human-readable string that expresses a separate concern about data disclosure to cloud services. Tags may be used for broad categories of sensitive data (e.g. a tag interview-data in the Interview Tool) or be created for specific data (e.g. a tag product-announcement-x).

An administrator assigns each cloud service a pair of labels: a *service privilege label* $L^p$ and a *service confidentiality label* $L^c$. The privilege label $L^p$ marks the highest level of confidential data that a service is trusted to receive; the confidentiality label $L^c$ determines the default confidentiality of data created within that service.

The TDM associates *text segment labels* to *text segments*. When a text segment that has not been observed before is created in a service, it is assigned the label $L^c$ of that service. The text segment label restricts how the text segment can propagate to services. Together with service labels, text segment labels are thus used to reason about data disclosure to services. We discuss the different options for text segment lengths in §4.

A text segment with label $L_i$ should be released to a service with privilege label $L^p$ only if $L_i \subseteq L^p$. The policy enforcement module ensures that this condition is satisfied for every text segment that is uploaded to a service in plain text.

**Default tag assignment.** Next we describe how text segment labels can be used to control data propagation. Figure 3 shows a label assignment for the example in Figure 1, in which Interview Tool and Wiki documents must remain separate. The administrator has created two tags, $t_i$ for the Interview Tool and $t_w$ for the Wiki. Initially, $L^p$ and $L^c$ for both ser-

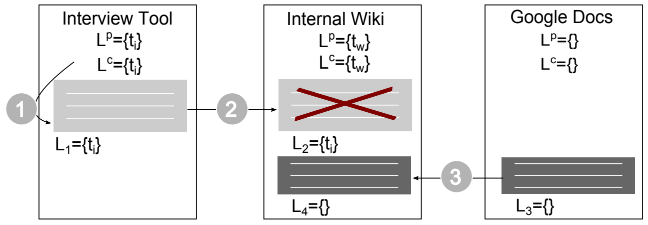


**Figure 3: Using tags and labels to enforce data disclosure policy between cloud services**

vices are set to $\{t_i\}$ and $\{t_w\}$, respectively. By using unique tags for the two services, data generated by one service may not be disclosed to the other.

In step 1 of Figure 3, text created in the Interview Tool is assigned automatically the default confidentiality label $L^c = \{t_i\}$ of that service. In step 2, the user copies the text from the Interview Tool to the Wiki. When the text is about to be uploaded, the policy lookup module determines the origin of each text segment and its associated label. In this case, the module retrieves the label $L_1$. Next, the enforcement mechanism compares $L_1$ with the privilege label $L^p = \{t_w\}$ of the Wiki. Since the text segment label is not a subset of $L^p$, i.e. $\{t_i\} \not\subseteq \{t_w\}$, BrowserFlow prevents the Wiki from sending Interview Tool data to its servers.

In step 3, the user copies text from Google Docs to the internal Wiki. Since Google Docs is an external untrusted service, it is assigned no tags. Setting $L^c = \{\}$ indicates that data generated in Google Docs is public and may flow to other services. Therefore the text segment generated in Google Docs can be sent successfully to the Wiki.

**User tag suppression.** A distinctive design choice in TDM is that it permits flexible declassification of data without sacrificing accountability. In contrast, previous DIFC models require users to have explicit privileges to perform declassification [22, 37].

In TDM, users may override policy restrictions by *suppressing existing tags* from text segment labels when they copy that segment. A suppressed tag is ignored when doing a subset comparison between labels, thereby allowing the data to propagate. Figure 4 shows the scenario from Figure 3 with tag suppression. In step 1, the user suppresses $t_i$ in the paragraph copied to the Wiki, permitting the upload to succeed.

Tag suppression incurs an audit trail because it may result in sensitive data disclosure. The suppressed tag remains attached to the label of the text segment in the target service of the data propagation. Along with a suppressed tag, we also store an identifier of the user who initiated the suppression and a justification to facilitate future audits. Note that tag suppression is done on a case-by-case basis, i.e. each time a user wishes to declassify the same text segment, they need to explicitly perform a tag suppression; otherwise the original source label will continue to be used for subset comparisons.

**Custom tag allocation.** In addition to the default tag assignment, users may allocate custom tags and use them freely in labels. Adding a custom tag to a text segment label further restricts the set of services that may be used to process the data. A user who allocates a new tag $t_n$ can add that tag to and remove that tag from the privilege labels of cloud services—they control which services may process data

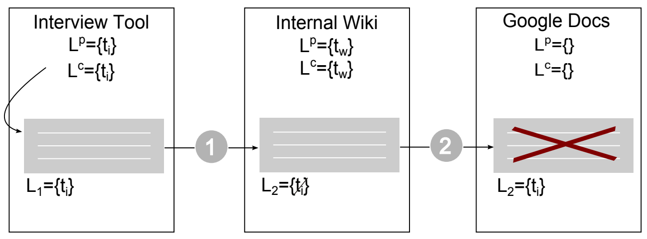


**Figure 4: Suppressing tags allows users to declassify text** (Suppressed tags are crossed out.)

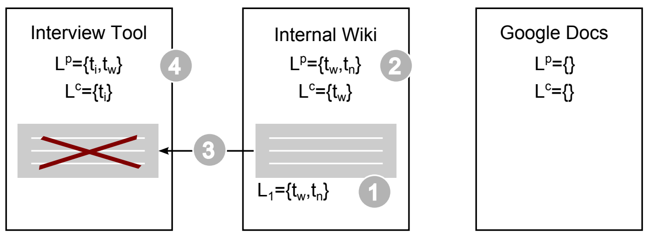


**Figure 5: Custom tags enable users to make data propagation more restrictive**

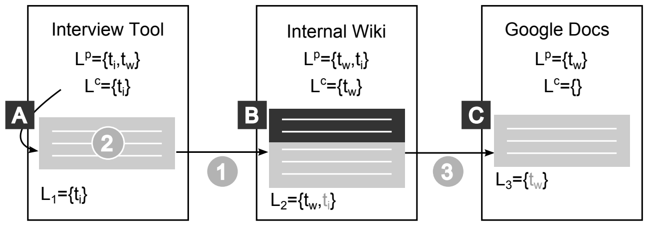


**Figure 6: Propagation of outdated tags occurs if tags remain attached to segments after significant edits** (We avoid this problem with implicit tags, which appear in grey.)

with $t_n$ in its label.

Figure 5 shows the same services as in Figure 3 but now the administrator has added $t_w$ to $L^p$ in the Interview Tool, permitting it to access data from the Wiki. Users, however, may use custom tags to prevent this. In step 1, a user who creates a new text segment in the Wiki allocates a new tag $t_n$ and adds it to the label $L_1$ of that text segment. The label $L^p$ of the Wiki is updated automatically to reflect its ability to process data protected with $t_n$ (step 2). Since the user did not add $t_n$ to the privilege label of the Interview Tool, the text from the Wiki may not propagate there (step 3).

Using a new custom tag to protect a text segment only prevents access for those services that do not already store a copy of the text segment. The TDM enforces that any service that already stores the text segment labelled with the new tag $t_n$ also receive $t_n$ as part of their privilege label. In the example from Figure 5, if the text segment had been observed first in the Interview Tool, the Interview Tool would have required $t_n$ in $L^P$ (step 4). This ensures that if such a text segment is seen again by a service, the TDM does not restrict its propagation.

## 3.2 Tag propagation

Tags must propagate from a source text segment to a destination while segments remain similar. When text is *copied* between documents tag propagation is straightforward—the source tags should become the tags of the destination text segment. It is more challenging to propagate tags correctly when text segments change. Once the text segment at the source or destination changes, the two may bear little resemblance. Tags from the source text segment should therefore no longer propagate to the destination text segment because text lineage is typically less important than the current content.

Consider the example in Figure 6: the Wiki contains $t_i$ in $L^P$ and Google Docs contains $t_w$ in $L^P$. This allows data propagation between the Interview Tool and the Wiki, and text segments created by the Wiki can flow to Google Docs (but Interview Tool text segments cannot). Consider two text segments in this scenario: segment $A$ labelled with $\{t_i\}$ and segment $B$ labelled with $\{t_w\}$. We assume that, as $B$ is edited and the user appends enough text from $A$, $B$ begins to disclose significant information from $A$ (step 1).

If all tags from $A$ were added to $B$'s label permanently, it would lead to overly conservative decisions after there is no similarity between $A$ and $B$: assume that, in step 1, $B$'s label becomes $\{t_w, t_i\}$. In step 2, $A$ is edited sufficiently to lose resemblance to its original version. Consider now what happens in step 3 when the second part of $B$, which came from $A$, is copied to Google Docs. Labelling $C$ with $\{t_i, t_w\}$ as if it originated from both the Interview Tool and the Wiki would be a false positive because $C$ discloses no sensitive information from the current version of $A$. Instead, given that the current authoritative source of $C$ is the Wiki, $C$ should only be marked $\{t_w\}$.

We prevent propagation of outdated tags in TDM using *implicit tags*. Implicit tags indicate that a given text segment is not the authoritative source of sensitive information.

**Explicit and implicit tags.** A segment label splits into explicit and implicit tags: *explicit tags* are those assigned by default due to the confidentiality label $L^c$ of a service and those assigned by users; *implicit tags* appear because the segment was found to disclose sensitive information in the past, i.e. these are tags copied from a source text segment to a destination text segment. After information disclosure to a destination text segment is detected, the explicit tags of the source are added to the destination as implicit tags.

Implicit tags enable the TDM to reason efficiently about the sensitivity of text segments. When editing a text segment, BrowserFlow only updates the label of the text segment being edited—it is not necessary to update proactively the labels of other text segments that were found to be similar in the past. This improves performance because it avoids having to update labels for text segments other than the current one—a user may never again inspect the documents found to be similar in the past.

## 4. IMPRECISE DATA FLOW TRACKING

The novelty of the BrowserFlow approach is that it casts data flow tracking to text similarity detection. Given a database of documents $db$ and some text $t$, we want to answer the question: *"what is the set of the original sources $s$ in $db$ that $t$ discloses significant information from currently?"* We call this the *information disclosure problem.*

This problem is closely related to the well-studied problem of plagiarism detection [40, 56, 58]. Existing solutions [30] can be divided into *information retrieval* techniques [40, 58], which directly compare the content of different documents, e.g. by counting word frequencies, and *fingerprinting* techniques [7, 56], which rely on hashes.

We propose an efficient fingerprinting algorithm for solving the information disclosure problem, as an extension of the *winnowing* algorithm for plagiarism detection [56].

## 4.1 Text fingerprinting

To measure information disclosure, BrowserFlow calculates a *fingerprint* for each text segment. When two fingerprints match perfectly, the segments contain largely the same text. A fingerprint is a set of hashes carefully chosen from particular passages in the paragraph and is calculated using an efficient hash function [34].

BrowserFlow uses the same strategy as the winnowing algorithm [56] to decide which hashes to include in a text segment's fingerprint, which has two useful properties: first, hashes are selected from the text segment at regular intervals, making the fingerprint linear to the segment size. This guarantees that if two text segments share a passage longer than a minimum threshold, their fingerprints share at least one hash. Provided that the location of the corresponding source text for each hash in the fingerprint is also stored, it becomes possible to attribute accurately which text segment passages caused information disclosure. Second, the selected hashes are not affected strongly by the addition or removal of characters in different parts of the text segment, or by shuffling the content of a document. As small modifications of a text segment result in small changes to its fingerprint, our proposed disclosure metric becomes robust (see §4.2).

To calculate a text segment's fingerprint, BrowserFlow performs four steps:

S1 It normalises the text segment by removing punctuation, whitespace and character case. For example, `"Hello World!"` is transformed to `"helloworld"`.

S2 It calculates the hash for each *n-gram* of a given length. In our example, assuming 6-grams, yields `"hellow"`, `"ellowo"`, `"llowor"`, `"lowor l"` and `"oworld"`. These correspond to 5 hash values, e.g. $\{52, 40, 53, 13, 22\}$.

S3 It defines overlapping windows over the set of hashes and chooses one hash from each window. With a window size of 3, we obtain the following windows: $\{52, 40, 53\}$, $\{40, 53, 13\}$ and $\{53, 13, 22\}$.

S4 It chooses the hash with the minimum value in each window to be added to the fingerprint. In this example, the fingerprint becomes $\{40, 13\}$.

By choosing the minimum hash value in each window, the same hash is likely to be found in consecutive windows and after the input text segment is modified only slightly. This reduces the fingerprint size and avoids large changes to the fingerprint after small changes to the text segment.

An important decision is the granularity at which text propagation is tracked, which determines what a text segment is. We assume that text documents are structured as traditional paper documents, i.e. each document has multiple paragraphs. For some documents, a significant number of individual paragraphs can be revealed without disclosing the document's content, but revealing one sentence from *each* paragraph would disclose the document. BrowserFlow therefore tracks text segments at *two granularities* independently, namely individual paragraphs and entire documents. This enables BrowserFlow to report information disclosure both when an employee discloses information from a single paragraph and from across multiple paragraphs.

## 4.2 Computing information disclosure

After calculating document and paragraph fingerprints we need to compute information disclosure. Our approach for this extends the definition of *containment* [7].

We define the *document disclosure* of document $A$ towards document $B$ as

$$D_{doc}(A, B) = \frac{|F(A) \cap F(B)|}{|F(A)|}$$

where $F$ returns the fingerprint of the entire document. Document disclosure quantifies how much text from document $A$ is found in document $B$. It has a value in $[0, 1]$: 0 means no disclosure; and 1 is full disclosure, i.e. all fingerprint hashes of document $A$ are also found in the fingerprint of document $B$. Similarly, we define *paragraph disclosure* as

$$D_{par}(A_p, B) = \frac{|F(A_p) \cap F(B)|}{|F(A_p)|}$$

where $A_p$ is a paragraph of document $A$, and $F$ returns its fingerprint.

Document and paragraph disclosure offer the flexibility to adjust the sensitivity when detecting disclosure for each document and paragraph individually. We define a *document disclosure threshold* $T_{doc}$ and a *paragraph disclosure threshold* $T_{par}$, set e.g. by the author of a document and paragraph, respectively. There is significant information disclosure for document $A$ to document $B$ when

$$D_{doc}(A, B) \geq T_{doc}(A) \ \text{ or } \ \exists A_p \in A : D_{par}(A_p, B) \geq T_{par}(A_p)$$

We refer to these as the *document* and *paragraph disclosure requirements*, respectively.

For example, with $T_{par}(A_{p1}) = 0$ and $T_{par}(A_{p2}) = 0.8$, information disclosure is detected when any hash in the first paragraph's fingerprint is leaked, but only when 80% of the hashes of the second paragraph's fingerprint are found in another document.

Users should adjust the paragraph and document disclosure thresholds of the text that they generate according to their requirements and the confidentiality of the text. In §6.1, we explore the effect of different threshold values.

Document and paragraph disclosure thresholds express orthogonal constraints. Given a database of a single document $A$ and a new document $B$, the document and paragraph disclosures are calculated independently. If any of the two constraints for detecting disclosure is satisfied, BrowserFlow detects information disclosure.

## 4.3 Overlapping documents

When similar text exists in multiple text segments, this can inadvertently boost the value of the disclosure metrics. Our metrics only measure pairwise information disclosure between a source and target segment. We may misreport disclosure requirement violations if we attempt to measure disclosure from multiple similar source text segments.

Consider Figure 7, which shows two documents $A$ and $B$, each with one paragraph. The paragraph in $B$ is a superset of the paragraph in $A$, with some additional text. A user pastes another copy of the overlapping text to document $C$. Assuming that $T_{par} = 0.5$ for the paragraphs in $A$ and

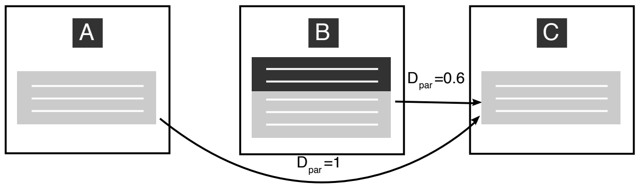


**Figure 7: With overlap between documents, disclosure metrics may misreport the true source of sensitive information.**

$B$, more than 50% for the hashes in each paragraph must appear in $C$ for violating the pairwise paragraph disclosure requirement. This is indeed the case, which would lead to a report that $C$ discloses significant information from *both* paragraphs in $A$ and $B$. In reality all sensitive information in $C$ originates only from $A$.

We compensate for this by (i) recording a *timestamp* when observing each hash in a paragraph's fingerprint for the first time; and (ii) adjusting the document and paragraph disclosure equations to ignore hashes that appear first in fingerprints of different paragraphs or documents. Our document and paragraph disclosure equations thus become

$$D_{doc}(A,B) = \frac{|F_{authoritative}(A) \cap F(B)|}{|F(A)|}$$

$$D_{par}(A_p,B) = \frac{|F_{authoritative}(A_p) \cap F(B)|}{|F(A_p)|}$$

where $F_{authoritative}$ returns part of a fingerprint: the *authoritative* fingerprint only contains hashes for which no earlier associations with other text segments exist.

Calculating $D_{par}(B,C)$ in the example from Figure 7 returns a value less than 0.5 because $F_{authoritative}(B)$ contains only hashes from the new text in $B$.

**Text disclosure algorithm.** Algorithm 1 calculates the set of paragraphs from which a given paragraph $P$ discloses significant information according to paragraph disclosure.

The algorithm leverages two data structures to quickly retrieve existing paragraphs and their fingerprints given a fingerprint hash. The first data structure ($DB_{hash}$) stores associations of fingerprint hashes to paragraphs that have been found to contain those hashes along with timestamps. The second data structure ($DB_{par}$) stores associations of paragraphs to the last fingerprint that has been calculated for each paragraph. To maximise lookup performance we recommend using an in-memory, hashtable-based implementation for both $DB_{hash}$ and $DB_{par}$.

Given paragraph $P$ the algorithm iterates over the hashes in the paragraph's fingerprint. It identifies every candidate paragraph $p$ that shares at least one hash with $P$. It then proceeds to calculate the pairwise paragraph disclosure $D_{par}(p,P)$. If the result is greater than the disclosure threshold $t$ the algorithm inserts $p$ into the result set $R$. Note that the algorithm quickly discards candidate paragraphs based on fingerprint length, i.e. short candidate paragraphs are discarded early.

The algorithm has a time complexity that is linear to the number of paragraphs that $P$ shares at least one hash with. It can operate in an incremental fashion: if a user edits paragraph $P$ by adding one hash $h$, the algorithm's main loop only needs to inspect $h$. This is an important property because it means that only text that is close to the text being edited may trigger disclosure calculations from other documents.

**Algorithm 1: Computes source paragraphs that are disclosed by a given paragraph** (A similar algorithm can be used to identify source documents based on document disclosure.)

```
Data: $P$: Paragraph
Data: $t$: Paragraph disclosure threshold
Data: $DB_{hash}$: Hashes database
Data: $DB_{par}$: Paragraph fingerprint database
Result: $R$: Set of origin paragraphs $O$ that satisfy the
        paragraph disclosure requirement to paragraph $P$
$f_{par} \leftarrow F(P)$;
$R \leftarrow \{\}$;
for hash $h$ in $f_{par}$ do
    $p \leftarrow oldestParagraphWith(h, DB_{hash})$;
    if $p = P$ then
        continue;
    $t \leftarrow p.threshold$;
    $f_{origin} \leftarrow hashesOf(p, DB_{par})$;
    if $count(f_{origin}) * t > count(f_{par})$ then
        continue;
    $f_{authoritative} \leftarrow \{\}$;
    for hash $l$ in $f_{origin}$ do
        if $p = oldestParagraphWith(l, DB_{hash})$ then
            $f_{authoritative}$.add($l$);
    $overlap \leftarrow f_{authoritive} \cap f_{par}$;
    if $count(overlap) > count(f_{origin}) * t$ then
        $R$.add($p$);
```

## 4.4 Limitations

Imprecise data flow tracking is not effective at a finer granularity than paragraphs due to the inability of the fingerprinting technique to quantify similarity of short text segments without a lot of false positives. Short but sensitive text, however, is typically only relevant from a confidentiality perspective in specific scenarios, e.g. when the text is used as a password. For such specific use cases, for example password reuse prevention, specialised systems which rely on data equality only [38] are more effective.

While many services used for document editing (e.g. Google Docs or Zoho) or user communication (e.g. Facebook's newsfeed or web forums) have the concept of documents and paragraphs, some services do not. They may be supported by BrowserFlow if there is a service-specific transformation of the service's data to text segments.

Imprecise data flow tracking cannot track data propagation once users rephrase entire paragraphs. This can happen both as part of a legitimate user workflow and as an explicit attempt to avoid BrowserFlow. We believe that the former is not an important limiting factor and that the latter is an exception rather than the rule. In addition, our approach is ineffective if cloud services explicitly try to evade the tracking. We argue that this is an acceptable limitation—most cloud services in the enterprise space have an incentive to cooperate with organisations regarding the control of sensitive data.

Data processed outside the browser in native applications introduces data propagation false positives and false negatives. Imprecise data flow tracking should be extended to be aware of data sources outside the browser. This can be achieved by integrating with DLP systems that monitor data flow in native applications (§2.2).

Imprecise data flow tracking may introduce privacy considerations. Monitoring of all text that users edit can be deemed unacceptable if a device is also used for personal workflows. Such monitoring practices are, however, already considered fair [62], e.g. in the context of phishing detection many IT departments aggressively scan all employee email.

Imprecise data flow tracking may also introduce security considerations. Storing fingerprints long-term to facilitate disclosure calculations (e.g. $DB_{par}$) can introduce an additional attack target if a device gets compromised. To mitigate this we recommend encrypting all fingerprint data at rest and performing periodic removal of old fingerprints.

## 5. IMPLEMENTATION

We implemented BrowserFlow as a plug-in for the Google Chrome web browser. An implementation of BrowserFlow needs the ability to (i) observe data that appears in a cloud service; and (ii) intercept outgoing data transfers when enforcing compliance decisions according to a TDM policy. Our plug-in reports policy violations to the user by changing the background colour of paragraphs and can also encrypt confidential data before upload [18, 31]. We describe two approaches for interception: one for primarily static web pages and one for modern, AJAX-based services [63].

### 5.1 Static web pages

**Text extraction.** We adopt an approach similar to Readability [2] to extract text passages from web pages. The BrowserFlow plug-in inspects the DOM tree of each page after loading, searching for HTML elements with significant text. We apply a set of heuristics to rank elements according to how much "interesting" text they contain and select the element with the highest score. These heuristics reward the existence of `<p>` tags, text that contains commas, and `id` attributes, which have known representative values such as `article`. Similarly, they penalise bad `class` attribute names such as `footer` or `meta` and high number of links over text length. After identifying the most interesting elements, BrowserFlow extracts the text from them by removing all HTML tags. This approach can identify text in static HTML pages, e.g. as generated by Drupal [21] and WordPress [65].

**Form-based interception.** BrowserFlow intercepts outgoing data transfers via *HTML forms* [52]. It adds an event listener for the `submit` event of the `<form>` elements of web pages. When a user submits a form, the listener suppresses the outgoing web request, inspects all non-hidden `<input>` elements in the form and extracts their `value` attributes. If the action is not found to leak sensitive data according to the TDM, the listener allows the `submit` event to trigger the form submission. This approach is sufficient to intercept data transfers in a wide range of cloud services, such as the Facebook composer, forums based on vBulletin [33] and the comments system in WordPress.

### 5.2 Dynamic web pages

Generic data interception in modern cloud services that use asynchronous requests is more challenging. Such services (i) may embed user-provided text in the DOM tree outside of common HTML input elements such as `<input>` and `<textarea>`; (ii) they may rely heavily on CSS for formatting, avoiding standard HTML elements such as `<p>`; and (iii) they may obfuscate user input in outgoing web requests. For example, Google Docs embeds directly into the DOM tree, uses custom formatting to make elements form paragraphs and pages, and communicates document mutations via AJAX requests each time a character is added or deleted.

| Dataset | Documents | Versions | Paragraphs | Size (in KB) |
|---|---|---|---|---|
| Wikipedia | Articles | 1000 | 60 | 30 |
| Manuals | | | | |
| IPhone | Camera | 4 | 40 | 6.1 |
| IPhone | Message | 4 | 20 | 3.3 |
| MySQL | New Features | 4 | 28 | 5.6 |
| MySQL | What's MySQL | 4 | 8 | 5.0 |
| News | Articles | 2 | 27 | 5.5 |
| Ebooks | Books | 1 | 1500 | 470 |

**Table 1: Datasets used for information disclosure evaluation** (The paragraph and size columns show average values across document versions.)

We describe two generic browser mechanisms to support such cloud services, including Google Docs and Evernote. These mechanisms can be used to support other services with minimal effort compared to DLP systems implemented as application firewalls, which need to understand the wire format used by each service (see §2.2).

**Mutation observers.** BrowserFlow receives notifications about the existence of new data in a cloud service via *mutation observers* [36]. A mutation observer is an object that can be attached to an element in the DOM tree and receives notifications when any change occurs in the subtree rooted at that element. When a mutation observer triggers, it can access information about the changes that occurred. Mutation observers therefore simplify data interception because they do not require the service to perform externally visible actions, such as external communication to access data. Since interception occurs in the browser, every modification to the DOM tree is visible.

BrowserFlow uses mutation observers to intercept Google Docs. A *document observer* monitors the creation or deletion of paragraphs, and a *paragraph observer* monitors changes to paragraphs. When a user edits the document, the paragraph observer collects the changed text and passes it to the label lookup module for inspection.

**JavaScript prototypes.** BrowserFlow intercepts communication to the remote back- end servers by redefining the `send` method in JavaScript's `XMLHttpRequest` object. JavaScript dispatches method invocations dynamically. If an object does not contain a method, the method call is dispatched to its prototype object. BrowserFlow sets a custom `XMLHttpRequest.prototype.send` method, exposing an interception point to observe all HTTP requests. This permits BrowserFlow to inspect all data that gets transmitted, allowing or preventing the request. Compared to traffic interception outside the browser, this approach is applicable when traffic is encrypted and allows for interaction with the user if so required, e.g. to communicate policy violations.

## 6. EVALUATION

The goals of our experimental evaluation are to assess the effectiveness of BrowserFlow in detecting disclosure (§6.1)

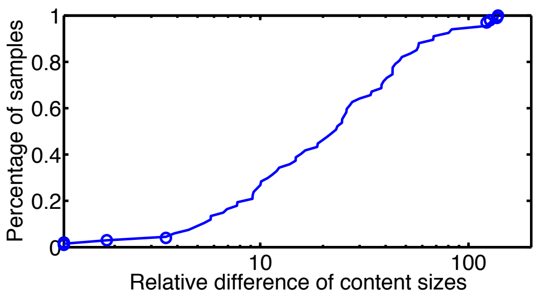


**Figure 8: Changes in article length**

and to measure its impact on performance (§6.2).

## 6.1 Effectiveness

BrowserFlow's ability to detect data disclosure depends on the effectiveness of imprecise data flow tracking. We perform two studies: (i) we investigate the accuracy of information disclosure reported by BrowserFlow for two datasets that contain documents across multiple revisions; and (ii) we explore the behaviour of BrowserFlow under different values for the paragraph disclosure threshold $T_{par}$ (see §4.2).

We configure BrowserFlow to calculate 32-bit hashes over n-grams of 15 characters with a window size of 30 characters. We focus on the paragraph tracking granularity; the results for the document granularity are similar. We set the paragraph disclosure threshold to $T_{par} = 0.5$, according to the results of the analysis below.

**Information disclosure detection.** To evaluate the accuracy of BrowserFlow in detecting information disclosure, we need a corpus of documents that evolves over time while maintaining overlap between revisions. We also need to have a ground truth to decide if the disclosure decisions made by BrowserFlow are appropriate.

As summarised in Table 1, we conduct experiments with two datasets of documents: (i) a corpus of 100 Wikipedia articles that contains the last 1000 revisions for each article (Wikipedia); and (ii) two chapters from two technical manuals that include the past 4 versions of each chapter (Manuals). Since the Wikipedia dataset is large, we use a heuristic to obtain the ground truth about disclosure between paragraphs across revisions (see below); for the smaller Manuals dataset, the ground truth is given by a human expert.

*Wikipedia dataset.* We collect articles for popular topics such as “Chicago” and “Chemotherapy”. The length of the articles varies across different revisions. Figure 8 shows the cumulative distribution of the changes of article lengths between the oldest and most recent revision derived by additive comparison. We make the assumption that articles that retain similar lengths between two revisions remain largely unchanged and are only subject to minor modifications. In contrast, we consider articles with large article-length differences to change more substantially. We use this measure as a heuristic to estimate the amount of data disclosure between article versions.

Figures 9a and 9b show the percentage of paragraphs from the oldest article revision that BrowserFlow detects to be disclosed by newer revisions. They focus on articles with the minimum and the maximum length changes from Figure 8, respectively.

Figure 9a shows that BrowserFlow reports disclosure for almost all paragraphs across revisions in articles with stable lengths. Indeed, those articles are on subjects that remain largely the same over time and have mature content (e.g. “Chicago” or “C++”). In contrast, the articles in Figure 9b with large length variations show that the text from the original version cannot always be found in later revisions. This includes articles on controversial or less mature topics (e.g. “Dow Jones” or “Dementia”). For both types of articles, BrowserFlow reports paragraph disclosure that follows our intuition.

*Manuals dataset.* We conduct a small-scale experiment where a human expert provides ground truth for two chapters from the iPhone and MySQL manuals. We use a set of different manual versions, in which some chapters have changed significantly (e.g. iPhone “Message” chapter), whereas others have largely stayed unaltered across versions (e.g. MySQL “What's MySQL”). We use the oldest version of each chapter as the base, and calculate how many of its paragraphs are disclosed by updated versions. The human expert reports disclosure when similar content or concepts are mentioned, regardless of the actual words used in the disclosing paragraphs.

Figures 10a–10d show BrowserFlow's results and those from the human inspection for our four chapters. Overall BrowserFlow's disclosure decisions match the human expert. Both chapters of the iPhone manual change significantly over time. The latest document version (iOS7) discloses almost no information from the base version (iOS3). For the MySQL manual, the “New Features” chapter shows reduced disclosure after version 4.1, while the “What's MySQL” chapter remains unchanged across versions.

We observe a systematic, small number of false negatives for short paragraphs without enough characters to fill a fingerprinting window. In addition, paragraphs rephrased extensively only get reported by the human expert.

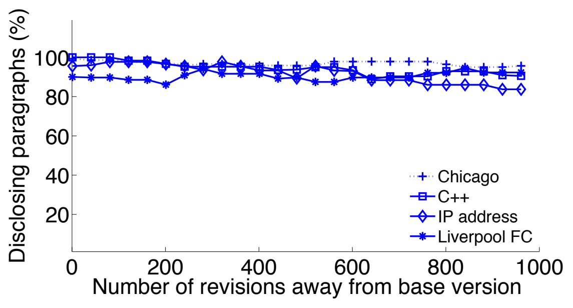
**Paragraph disclosure threshold.** Next we explore the choice of the paragraph disclosure threshold $T_{par}$. For the Manuals dataset, we vary $T_{par}$ and observe how it affects false positives and negatives. To reduce the impact of systematic errors, we ignore paragraphs for which the fingerprint set is empty.

Figure 11 shows the ratio of the total number of paragraphs that BrowserFlow reports as being disclosed in newer chapter versions over the number reported by the human expert: a value 1 indicates that there is agreement; values above or below 1 indicate false positives and negatives, respectively.

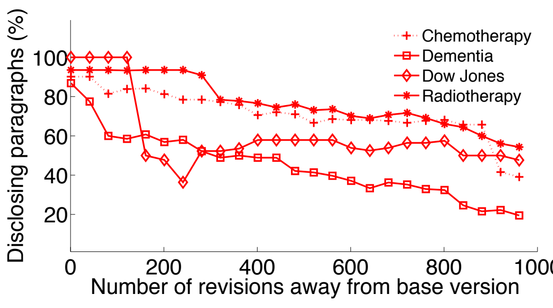
We observe that, for values of $T_{par}$ between 0.2 and 0.8, BrowserFlow agrees with the ground truth for more than 90% of the paragraphs. The value of $T_{par}$ has less impact when paragraphs share almost all content, or none at all. In such cases, $D_{par}$ has either a small or large value compared to $T_{par}$. As a result, BrowserFlow can detect information disclosure robustly, independently of $T_{par}$, if the text overlap is low (e.g. paragraphs with no shared text) or high (e.g. identical paragraphs due to copy and paste). Based on these results, we adopt a default value of $T_{par} = 0.5$, i.e. disclosure is reported when at least 50% of an original paragraph is found in another paragraph.

## 6.2 Performance overhead

We evaluate the performance of BrowserFlow in terms

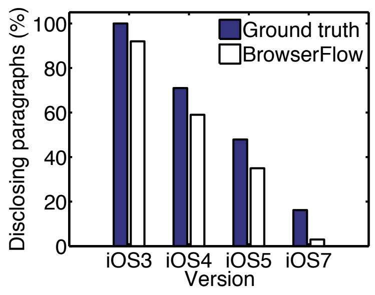


**(a)** Articles with low length variations

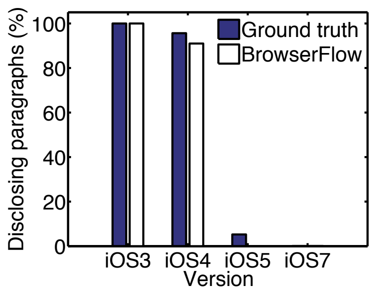


**(b)** Articles with high length variations

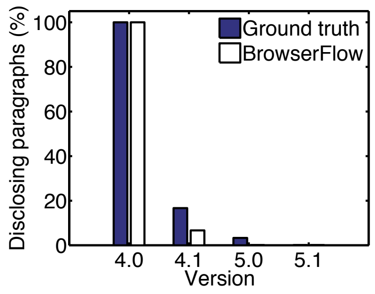
**Figure 9: Paragraph disclosure (Wikipedia dataset)**



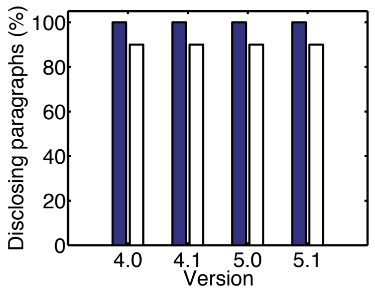
**(a)** IPhone Camera



**(b)** IPhone Message



**(c)** MySQL New Features



**(d)** MySQL What's MySQL

**Figure 10: Paragraph disclosure (Manuals dataset)**

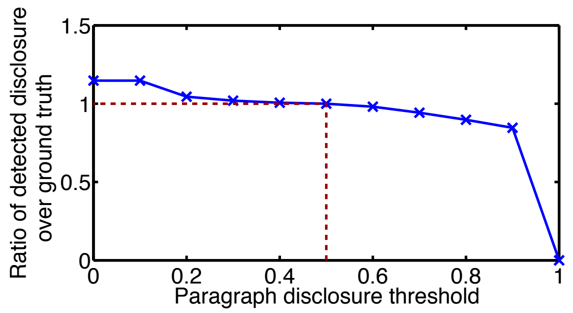


**Figure 11: Impact of paragraph disclosure threshold**

of its response time, i.e. how quickly information disclosure decisions are made when editing documents in Google Docs. When a user modifies a document in Google Docs, BrowserFlow is triggered asynchronously on each key press. This means that users do not perceive any additional delay when typing—independently of BrowserFlow's response time—because the disclosure calculation occurs in a different process [11]. The response time should be low, however, otherwise the delay in disclosure decisions would be noticeable to the cloud services, e.g. triggering "limited connectivity" error messages by Google Docs.

We use a large dataset of e-books from Project Gutenberg [27], which includes 180 e-books with sizes ranging between 300 KB and 5.5 MB. The total size is 90 MB. We load the dataset into BrowserFlow, creating 10 million distinct hashes in the fingerprint database. The experiments are performed on a 3.4 GHz Intel Core i7 machine with 16 GB of RAM, running Ubuntu 13.04 and Google Chrome version 33.0.1712.2.

**Response time.** We edit the documents in three different workflows:

- W1 A user creates a new document and enters a single page from an existing e-book;
- W2 A user enters an article that does not share text with

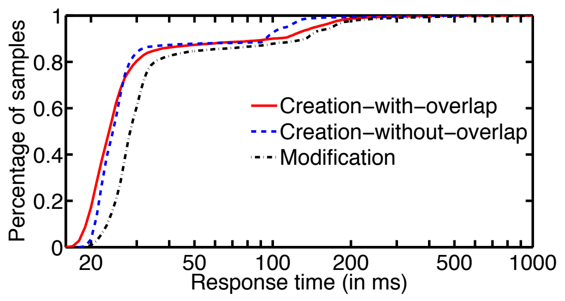


**Figure 12: Distribution of response times for disclosure decisions**

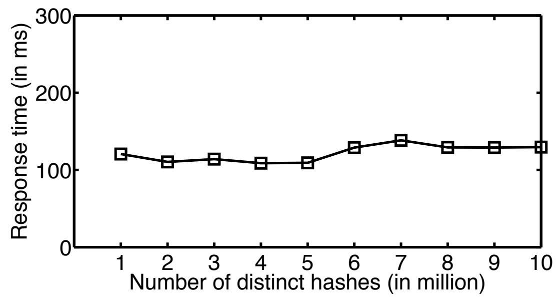


**Figure 13: Response time when varying the size of the hashes database**

any of the existing e-books;

W3 A user edits a previously-modified version of an e-book page to make it match the original version.

Figure 12 shows the distribution of response times for each of the above workflows measured as the time between the request and the disclosure decision. Response times are low: BrowserFlow is able to respond within 200 ms for 99% of the requests, and 85% of these responses arrive in less than 30 ms.

The response time depends on (i) caching and (ii) whether the edited text has significant overlap with other text in the system. Caching affects the majority of requests, which exhibit the lowest latencies (less than 30 ms). Requests are served quickly because one keystroke typically does not alter the winnowing fingerprint of a paragraph, permitting BrowserFlow to reuse its previous response. Requests that trigger disclosure calculation have higher latencies (90–200 ms). The impact of overlapping text is seen when comparing response times of W1 and W3 with W2—response times in the former are typically higher. As described in §4.2, when a paragraph overlaps with others, the hashes of every overlapping paragraph have to be inspected to calculate $D_{par}$.

In conclusion, the HTTP traffic interception that BrowserFlow performs is not noticeable by users. BrowserFlow only blocks data exchange between the main application window and the remote back-end servers for short periods of time, thus not affecting functions such as spell-checking or document synchronisation across multiple devices. Overall the use of BrowserFlow is similar to experiencing slightly increased network latency—a scenario which cloud services are designed to tolerate.

**Scalability.** We investigate how the number of hashes maintained by BrowserFlow affects performance. We vary the number of documents loaded into BrowserFlow from 5 MB (1 million hashes) to 90 MB (10 million hashes). For each size of the hashes database, we create a new empty document and paste a 500-character long paragraph from an existing book to trigger the disclosure calculation.

Figure 13 shows the 95th percentile of response times with an increasing number of hashes. The response time scales sub-linearly with the size of the hashes database, remaining typically below 200 ms. This is due to the use of index data structures in the BrowserFlow implementation to improve the lookup performance.

We conclude that, in practice, the size of the hashes database is not the limiting factor for scaling to larger document numbers. Performance is determined primarily by how many popular text passages appear in multiple different paragraphs.

## 7. CONCLUSIONS

Cloud services increasingly replace desktop applications in organisations and this makes accidental disclosure of sensitive data common. We suggested imprecise data flow tracking, a robust and practical technique for tracking sensitive data in the browser. We also introduced the Text Disclosure Model (TDM) that enables organisations and users to collaboratively specify data disclosure policies. We demonstrated the feasibility of our ideas through BrowserFlow, a prototype plug-in for Chrome that intercepts document editing in Google Docs and prevents policy violations. Based on an experimental evaluation, we showed that BrowserFlow can prevent sensitive data disclosure before it occurs, while having a negligible performance impact on user experience.

## 8. ACKNOWLEDGEMENT

This work was supported by CloudFilter (EP/J020370/1) and CloudSafetyNet (EP/K008129/1) grants from the UK Engineering and Physical Sciences Research Council (EPSRC).

## 9. REFERENCES

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[2] Arc90labs. Readability. `code.google.com/p/arc90labs-readability/`, 2016.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, 2014. ACM.

[4] T. Austin and C. Flanagan. Efficient Purely-Dynamic Information Flow Analysis. In *Programming Languages and Analysis for Security (PLAS)*, Dublin, Ireland, 2009. ACM.

[5] T. Austin and C. Flanagan. Permissive Dynamic Information Flow Analysis. In *Programming Languages and Analysis for Security (PLAS)*, Toronto, Canada, 2010. ACM.

[6] Barracuda. Web application firewall. `www.barracuda.com/products/webapplicationfirewall`, 2011.

[7] A. Broder. On the Resemblance and Containment of Documents. *Compression and Complexity of SEQUENCES*, 1997.

[8] B. Burke and C. Christiansen. Insider Risk Management: A Framework Approach to Internal Security. Technical report, RSA, 2009.

[9] J. Burket, P. Mutchler, M. Weaver, M. Zaveri, and D. Evans. GuardRails: A Data-centric Web Application Security Framework. In *Web Application Development (WebApps)*, Portland, OR, 2011. USENIX.

[10] C. Coles. What are the Top Data Loss Prevention Tools? `https://www.skyhighnetworks.com/cloud-security-blog/what-are-the-top-data-loss-prevention-tools/`, May 2016.

[11] N. Carlini, A. Felt, and D. Wagner. An Evaluation of the Google Chrome Extension Security Architecture. In *Security Symposium*, Bellevue, WA, 2012. USENIX.

[12] CDW. State of the cloud report. `www.cdwnewsroom.com/wp-content/uploads/2013/02/CDW_2013_State_of_The_Cloud_Report_021113_FINAL.pdf`, 2013.

[13] Check Point. DLP Software Blade. `www.checkpoint.com/products/dlp-software-blade/`, 2015.

[14] W. Cheng, D. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov. Abstractions for Usable Information Flow Control in Aeolus. In *Annual Technical Conference (ATC)*, Boston, MA, 2012. USENIX.

[15] S. Chong, K. Vikram, and A. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Security Symposium*, Boston, MA, 2007. USENIX.

[16] Citrix. NetScaler. `www.citrix.com/products/netscaler-appfirewall/`, 2011.

[17] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing Authentication and Access Control Vulnerabilities in Web Applications. In *Security Symposium*, Montreal, Canada, 2009. USENIX.

[18] G. D'Angelo, F. Vitali, and S. Zacchiroli. Content Cloaking: Preserving Privacy with Google Docs and Other Web Applications. In *Symposium on Applied Computing (SAC)*, Sierre, Switzerland, 2010. ACM.

[19] D. Devriese and F. Piessens. Noninterference through Secure Multi-execution. In *Symposium on Security and Privacy*, Oakland, CA, 2010. IEEE.

[20] X. Dong, Z. Chen, H. Siadati, S. Tople, P. Saxena, and Z. Liang. Protecting Sensitive Web Content from Client-side Vulnerabilities with CRYPTONS. In *Computer and Communications Security (CCS)*, Berlin, Germany, 2013. ACM.

[21] Drupal. `www.drupal.org`, 2016.

[22] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and Event Processes in the Asbestos Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Brighton, United Kingdom, 2005. ACM.

[23] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information Flow Tracking System for Real-time Privacy Monitoring on Smartphones. *Communications of the ACM*, 57(3), 2014.

[24] Frost and Sullivan. The hidden truth behind shadow IT. `www.mcafee.com/us/resources/reports/rp-six-trends-security.pdf`, November 2013.

[25] Google. Google Docs. `docs.google.com`, 2016.

[26] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: A Web Browser With Flexible and Precise Information Flow Control. In *Computer and Communications Security (CCS)*, Raleigh, NC, 2012. ACM.

[27] Project Gutenberg. `www.gutenberg.org/`, 2016.

[28] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song. ShadowCrypt: Encrypted Web Applications for Everyone. In *Computer and Communications Security (CCS)*, Scottsdale, Arizona, 2014. ACM.

[29] B. Hicks, K. Ahmadizadeh, and P. McDaniel. From Languages to Systems: Understanding Practical Application Development in Security-typed Languages. In *Annual Computer Security Applications Conference (ACSAC)*, Miami Beach, FL, 2006. IEEE.

[30] T. Hoad and J. Zobel. Methods for Identifying Versioned and Plagiarised Documents. *Journal of the American Society for Information Science and Technology*, 54, 2003.

[31] Y. Huang and D. Evans. Private Editing Using Untrusted Cloud Services. In *International Conference on Distributed Computing Systems Workshops (ICDCSW)*, Minneapolis, MN, 2011. IEEE.

[32] R. C. Jammalamadaka, R. Gamboni, S. Mehrotra, K. Seamons, and N. Venkatasubramanian. A Middleware Approach for Outsourcing Data Securely. *Elsevier Computers & Security*, 32, 2013.

[33] vBulletin Connect. `www.vbulletin.com/`, 2016.

[34] R. Karp and M. Rabin. Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development*, 31(2), 1987.

[35] V. Kemerlis, P. Georgios, K. Jee, and A. Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Virtual Excecution Environments (VEE)*, London, UK, 2012. ACM.

[36] A. Kesteren, A. Gregor, Ms2ger, A. Russell, and R. Berjon. W3C DOM4. Technical report, W3C, 2014.

[37] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, 2007. ACM.

[38] LastPass. `www.lastpass.com`, 2016.

[39] S. Maffeis, J. C. Mitchell, and A. Taly. Object Capabilities and Isolation of Untrusted Web Applications. In *Symposium on Security and Privacy*, Oakland, CA, 2010. IEEE.

[40] G. S. Manku, A. Jain, and A. Das Sarma. Detecting Near-Duplicates for Web Crawling. In *World Wide Web (WWW)*, Banku, Canada, 2007. ACM.

[41] McAfee. Total Protection for Data Loss Prevention. `www.mcafee.com/us/products/total-protection-for-data-loss-prevention.aspx`, 2016.

[42] Microsoft. Office 365. `portal.microsoftonline.com`, 2016.

[43] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe Active Content in Sanitized JavaScript. Technical report, Google, 2007.

[44] M. Mongiovì, G. Giannone, A. Fornaia, G. Pappalardo, and E. Tramontana. Combining Static and Dynamic Data Flow Analysis: A Hybrid Approach for Detecting Data Leaks in Java Applications. In *Symposium on Applied Computing (SAC)*, Salamanca, Spain, 2015. ACM.

[45] M. Mowbray and S. Pearson. A Client-based Privacy Manager for Cloud Computing. In *Communication System Software and Middleware (COMSWARE)*, Dublin, Ireland, 2009. ACM.

[46] M. Mowbray, S. Pearson, and Y. Shen. Enhancing Privacy in Cloud Computing via Policy-based Obfuscation. *Springer Journal of Supercomputing*, 61(2), 2012.

[47] MyDLP. `www.mydlp.com/`, 2016.

[48] A. Myers and B. Liskov. Protecting Privacy Using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology*, 9(4), Oct. 2000.

[49] I. Papagiannis, M. Migliavacca, and P. Pietzuch. PHP Aspis: Using Partial Taint Tracking to Protect Against Injection Attacks. In *Web Application Development (WebApps)*, Portland, OR, 2011. USENIX.

[50] S. Pearson, Y. Shen, and A. Mowbray. A Privacy Manager for Cloud Computing. In *International Conference on Cloud Computing*. IEEE, 2009.

[51] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, and H. Balakrishnan. Building Web Applications on Top of Encrypted Data Using Mylar. In *Networked Systems Design and Implementation (NSDI)*, Seattle, WA, 2014. USENIX.

[52] D. Raggett, A. Le Hors, and I. Jacobs. HTML 4.01 Specification. Technical report, W3C, 1999.

[53] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer. Information Flow Control for Event Handling and the DOM in Web Browsers. In *Computer Security Foundations Symposium (CSF)*, Verona, Italy, 2015. IEEE.

[54] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking Information Fow in Dynamic Tree Structures. In *European Symposium on Research in Computer Security (ESORICS)*, Saint Malo, France, 2009.

[55] A. Sabelfeld and A. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1), Jan. 2003.

[56] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *International Conference Management of Data (SIGMOD)*, San Diego, CA, 2003. ACM.

[57] E. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (But Might Have Been Afraid to Ask). In *Symposium on Security and Privacy*, Berkeley, CA, 2010. IEEE.

[58] N. Shivakumar and H. Garcia-molina. SCAM: A Copy Detection Mechanism for Digital Documents. In *International Conference in Theory and Practice of Digital Libraries*, 1995.

[59] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières. Protecting Users by Confining JavaScript with COWL. In *Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, 2014. USENIX.

[60] Symantec. What risks are employees taking with information? `www.symantec.com/connect/blogs/what-risks-are-employees-taking-information`, 2011.

[61] Symantec. Data Loss Prevention. `www.symantec.com/data-loss-prevention/`, 2016.

[62] Your Right to Privacy At Work. `https://www.tuc.org.uk/sites/default/files/tuc/privacyatwork.pdf/`, 2016.

[63] C. Ullman and L. Dykes. *Begining Ajax*. Wrox, 2007.

[64] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Computer and Communications Security (CCS)*, Scottsdale, Arizona, USA, 2014. ACM.

[65] Wordpress. `www.wordpress.org`, 2016.

[66] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-Preserving Browser-Side Scripting with BFlow. In *European Conference on Computer Systems (EuroSys)*, Nuremberg, Germany, 2009. ACM.

[67] A. Yip, X. Wang, N. Zeldovich, and F. Kaashoek. Improving Application Security with Data Flow Assertions. In *Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, 2009. ACM.

[68] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieres. Making Information Flow Explicit in HiStar. In *Operating Systems Design and Implementation (OSDI)*, Seattle, WA, 2006. USENIX.

[69] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking. *ACM Operating Systems Review*, 45(1), 2011.

[70] Zoho. Zoho docs. `www.zoho.com/docs`, 2016.