Enforcing User Privacy in Web Applications using Erlang

Ioannis Papagiannis*, Matteo Migliavacca*, David M. Eyers†, Brian Shand‡, Jean Bacon† and Peter Pietzuch*
*Department of Computing, Imperial College London, United Kingdom, {ip108, migliava, prp}@doc.ic.ac.uk
†Computer Laboratory, University of Cambridge, United Kingdom, {firstname.lastname}@cl.cam.ac.uk
‡CBCU, Eastern Cancer Registry, National Health Service, United Kingdom, brian.shand@cbcu.nhs.uk

***Abstract*—Social networking applications on the web handle the personal data of a large number of concurrently active users. These applications must comply with complex privacy requirements, while achieving scalability and high performance. Applying constraints to the flow of data through such applications to enforce privacy policy is challenging because individual components process data belonging to many different users.**

We introduce a practical approach for uniformly enforcing privacy requirements in such applications using the actor-based Erlang programming language. To isolate the personal data of users, we exploit Erlang's inexpensive process model and use Erlang's message passing mechanism to add policy checks. We illustrate this approach by describing the architecture of a privacy-preserving message dispatcher in a micro-blogging service. Our performance evaluation of a prototype implementation shows that this approach can enforce fine-grained privacy guarantees with a low performance overhead.

I. Introduction

Today's web applications have to comply with complex privacy requirements. Social networking websites, such as Facebook, Twitter and LinkedIn, handle a large amount of user-contributed data that is potentially private. They must manage this data in compliance with the privacy settings specified by users. For example, the visibility of status updates by users may have to be restricted to pre-approved friends only. In a micro-blogging website, users subscribing to the same anonymous discussion channel may have to be prevented from discovering each other's identities. The enforcement of such privacy policies in web applications is challenging in practice. A single programming error or design flaw in the application can result in violations of user privacy.

Ideally, web applications should uniformly enforce privacy policies by isolating data belonging to different users. Previous research on *information flow control* (IFC) [1], [2], [3], [4] has shown a possible solution by controlling the spread of data through a system. In IFC, the system tracks the taint caused by the data across components and prevents undesirable data flows that may potentially violate privacy policy. However, when applying IFC to web applications, two open challenges remain:

First, web applications must include IFC checks at component boundaries to enforce flow constraints. Current IFC approaches rely on static analysis or manual annotation to identify component boundaries [5], [6]. Employing unnecessary IFC checks will reduce performance but on the other hand, a single missing check may result in the leakage of private data.

Second, data from different users may be processed by the same component because web applications must scale to a large number of concurrent users. This prevents IFC from guaranteeing data isolation because data belonging to different users is interspersed within the component. For example, a dispatching component in a micro-blogging service may process messages on behalf of many users.

In this paper, we show how these problems can be solved by applying IFC to web applications implemented in Erlang [7], an actor-based programming language that relies on message passing. By avoiding shared state, IFC constraints in Erlang only need to be checked when passing messages between Erlang processes, resulting in a small amount of trusted code. To avoid that data flows belonging to different users mix in application components, Erlang's lightweight process primitives can be used to create fresh instances of components cheaply and thus to maintain isolation between components.

This approach enables the building of massively concurrent web applications that enforce IFC constraints with a low performance impact. We demonstrate this through the design and evaluation of a prototype implementation of a Twitter-like micro-blogging service that guarantees users' privacy. Our system can provide different privacy guarantees by maintaining isolation through the dispatching of messages. The results from an experimental evaluation show that this has an acceptable impact on performance and scalability of the dispatching service.

In summary, the main contributions of the paper are:

- an approach for practical enforcement of IFC constraints in Erlang with massive concurrency;
- a corresponding architecture for privacy-preserving message dispatching in a micro-blogging service;
- an experimental evaluation of a prototype implementation in comparison with a traditional dynamically-typed programming language.

The next section explores the privacy requirements in a micro-blogging web application. In §III, we describe how IFC can achieve these privacy goals and how it can be applied to Erlang processes. We describe the architecture

of a Twitter-like dispatching service based on this idea in §IV, followed by evaluation results in §V. The paper finishes with related work (§VI), future work (§VII) and conclusions (§VIII).

II. Privacy in Micro-Blogging

We chose a micro-blogging web application as a case study because it (1) must support complex privacy policies; (2) must handle a large number of concurrent users; and (3) includes non-trivial processing for disseminating messages to users. Today, micro-blogging allows only limited control over privacy. For example, while publishers can limit the set of subscribers who can view their messages, subscribers cannot control who can observe their membership; nor can they ensure that their subscription requests are delivered to the correct party.

For example, Twitter [8] is a micro-blogging service that allows users to exchange short text messages known as "tweets". Users can *publish* messages and *subscribe* to other users to receive their messages. Twitter currently supports a simple privacy model: users can choose to protect their messages by authorising subscribers before they start receiving messages. The list of subscribers is public and accessible from user profiles. Thus, compared to other social networks, Twitter offers limited privacy by default, with few options that users can customise.

Our approach can enhance such a micro-blogging service with more fine-grained, enforceable privacy controls, without sacrificing performance and scalability. Our goal is to provide the following *privacy guarantees* in micro-blogging:

- *G1* Messages from a publisher shall be received only by authorised subscribers. This ensures that a publisher can maintain control over its set of subscribers.
- *G2* Authorised subscribers shall not be disclosed to any other publisher or subscriber. This means that each subscription is kept private and no other user of the system, apart from the authorising publisher, may link a subscriber to a publisher.
- *G3* Subscription authorisation requests from subscribers shall be delivered only to the relevant publisher. This guarantees that no other user can perceive them and either authorise or leak them to third parties.

These privacy guarantees enable use cases that diverge from the default "share everything" paradigm of Twitter. *G1* is already an option in Twitter because many users want to control the distribution of their tweets. On the other hand, *G2* is not provided and subscription lists are public. As a result, having Bill Gates subscribe to his favourite singer's messages can become a worldwide topic of interest [9]. Finally, *G3* can be seen as a necessary step in order to achieve *G1* and *G2*.

Privacy guarantees, such as the ones stated above, are essentially end-to-end constraints on the system behaviour: they depend on how an application controls the flow of information, its processing and its external visibility. In practice, users' privacy is threatened by bugs, error conditions and unforeseen interactions between components. Despite the need for end-to-end enforcement, privacy often depends on the correct composition of local mechanisms scattered throughout an application. To increase confidence that the system has a correct, privacy-preserving implementation, IFC provides a systematic, end-to-end approach to the capturing of privacy policies and enforces them uniformly.

Designing global security mechanisms for web applications is challenging: the number of users of popular services is still growing exponentially, which makes scalability and performance requirements paramount. Selecting a mechanism that enforces privacy policy at the right granularity is crucial: mechanisms that require many policy checks cause high overhead, conversely mechanisms that are too coarse-grained risk missing violations of privacy policy. Next we describe how IFC applied to an actor-based programming language can achieve fine-grained privacy control for web applications.

III. Fine-Grained Privacy Control

Information flow control (IFC), and particularly its Decentralised variant (DIFC) [10], is a flexible technique for enforcing end-to-end security guarantees. It has been applied to various domains, ranging from programming languages [5] to OS design [1], [2] and web services [4], [11]. IFC is a form of mandatory access control that restricts the way information flows in a system. This is achieved by associating data with *tags*. Tagged data "taints" components that receive it, potentially restricting future flows of data to and from those components. Components can possess *privileges* to override IFC restrictions.

When using IFC for privacy enforcement, a system can be divided into three stages: (1) data enters the system, the privacy concern is identified and a tag is used to label the data; (2) data is processed by the system respecting the tag, preventing unauthorised components from accessing the data; and (3) a privileged component can decide when the privacy concern is satisfied or no longer relevant, and can thus remove the tag and possibly output the data externally.

Figure 1 shows these three stages in the context of a micro-blogging service that uses IFC to enforce the privacy guarantees from §II. The confidentiality of a *subscriber* should be protected by disclosing the subscription only to the relevant *publisher*. In step 1, after the subscriber has received a new subscription, it creates a subscription request and attaches to it the new tag t. The request has to be mediated by a *request processor* before it is delivered to the publisher. For this to happen, the subscriber grants a privilege t^+, i.e. *clearance* to access data in the request message, only to the request processor. In order to exercise clearance over t, a component must taint itself with t and afterwards, it can only modify resources that are also tainted by t. In step 2,

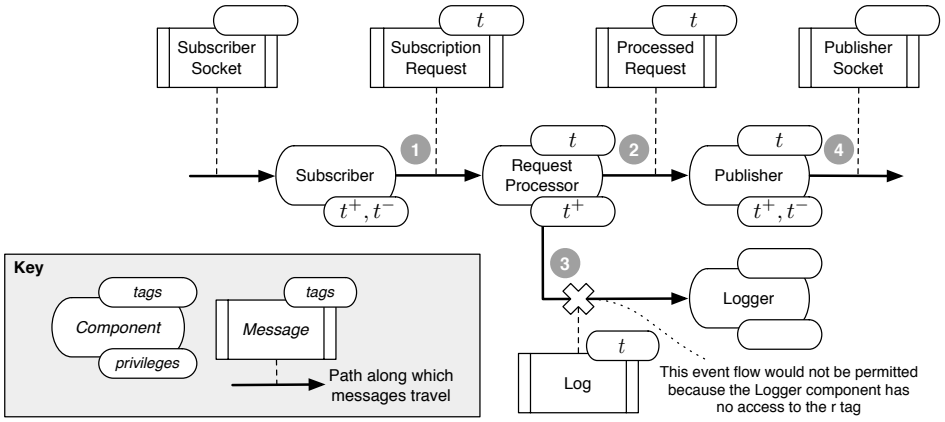

Figure 1. **Enforcing privacy guarantees with IFC in micro-blogging.** A subscriber produces a subscription request protected with a unique tag t. The request can only be delivered to the intended publisher, even if processed by a component that contains a bug that would otherwise violate this policy.

the request processor has used the t^+ clearance and is able to output the processed request, but again protected by t.

Now consider the case in which the request processor erroneously tries to send the request data to an unauthorised component, e.g. by posting the request to an unconstrained logger. As shown in step 3, the mandatory part of IFC (i.e. the "no-write-down" property) prevents this from occurring: the logger has no t^+ and thus cannot perceive any message published by the request processor. This mechanism guarantees that once data is labeled by a tag, the tag is preserved while the data is processed or copied. The protection on the request must be lifted when the request reaches its intended destination, the publisher. There, the subscriber must delegate to the publisher, along with t^+, the privilege to remove t, i.e. the t^- *declassification* privilege. Using t^-, the publisher component can ask the user for approval of the request in step 4.

While the above example shows the use of a single tag, multiple tags can be used to put additional restrictions on selected data. The set of all taints associated with components (and resources) is collected in the *confidentiality label*. Labels are partially ordered under the "can-flow-to" relation, which in this case is the subset relation. An IFC system ensures that any flow of data between execution components A and B only occurs if $\text{label}(A) \subseteq \text{label}(B)$ [3]. It is thus easy to translate a privacy guarantee into initial taints and privileges and use IFC to provide its end-to-end enforcement.

To enforce IFC, the interaction between any components in Figure 1 must be controlled. Any leakage of information between components would defeat the purpose of IFC. In addition, we only focused on a single publisher-subscriber interaction but in a micro-blogging application, thousands of users are connected to the service concurrently. This precludes the use of IFC-secure operating systems [1], [3], [2], in which labels are assigned to OS processes. OS processes impose a high cost for each independent entity that needs to be protected by a label. Splitting the application into separate OS processes would force all communication to happen through IPC during the dispatching of published messages, severely limiting the maximum throughput [12].

Instead we propose an approach where the granularity of enforcement remains flexible, similar to OS-level IFC, but that has lower overhead than using separate OS processes. The gap between the two approaches can be closed by using languages such as Erlang, which adopt a model of computation that imposes isolation of functional components except for when explicit communication is performed through message passing.

A. Erlang

Erlang is an actor-based programming language [7] whose shared-nothing concurrency model makes it a natural fit for IFC. Shared-nothing concurrency is seeing increasing interest from research and industry because it supports scaling up applications over multi-core infrastructure well. Erlang is a widespread and mature language that uses this paradigm. Applications in Erlang are partitioned into communicating *processes* that encapsulate concurrent computation. We exploit the following features to provide efficient IFC and thus privacy guarantees:

Message passing. Erlang processes communicate through explicit message passing, with no shared state between them. The actor model helps effect an enforcement mechanism for IFC because it avoids sharing data between threads; shared memory concurrency complicates reasoning about data ownership of processes by effectively blending the distinction between different protection domains [13]. Secondly, the actor model provides natural boundaries at which to carry out IFC checks: when messages are sent and received, label checks can be applied.

Isolation. To enforce IFC constraints, it is important that compartments annotated with tags are kept isolated. The

sequential part of Erlang is a functional language with single assignment and no direct access to global state. These features provide process isolation trivially; there is no other way of communication apart from message passing. In a system without such a property, IFC requires interception of every possible interaction between compartments. This is a challenging task that potentially reduces performance [12].

Lightweight processes. Erlang processes are lightweight compared to threads in most conventional programming languages: due to a small memory footprint, they are inexpensive to create and context switch to. This is important for IFC: in many applications, a component must handle the confidential data of many different users. IFC then imposes the partitioning of data into isolated compartments. Using conventional threads or processes to handle each compartment introduces significant load to a system. As we show in §IV, maintaining the privacy of publishers requires the use of individual dispatching threads per publisher.

B. Supporting IFC in Erlang

We support IFC in Erlang by attaching labels to Erlang process identifiers (`pid`), which are checked when messages are sent. Erlang processes can create new tags by invoking the `new_tag` function:

```
new_tag() -> tag()
```

Creating a fresh tag automatically grants the calling process clearance and declassification privileges over that tag.

Processes are created in Erlang by the `spawn` function. We add two parameters to this function, `TagsAdd` and `TagsRemove`, to specify the label of the spawned process with respect to the label of the spawning process:

```
spawn(TagsAdd, TagsRemove, Fun,
      Params) -> pid
```

The invoker must have clearance privileges for all tags specified in `TagsAdd` and declassification privileges for all tags specified in `TagsRemove`.

To send messages, processes use a `send` primitive, instead of the standard Erlang send operator (`!`):

```
send(PidReceiver, TagsAdd,
  TagsRemove, Message) -> ok|fail
```

Before sending a message, `send` checks if the label of the sending process, after adding tags in `TagsAdd` and removing those in `TagsRemove`, can flow to the label of the destination. The sending process needs to have declassification privileges for all tags in `TagsRemove`. An IFC-compliant Erlang implementation has to enforce the use of `send` using one of several approaches, from simple source-level transformation, through transparent bytecode rewriting, to a direct modification of the Erlang runtime. We leave this for future work.

Processes can grant their privileges to other processes by calling the `delegate` function:

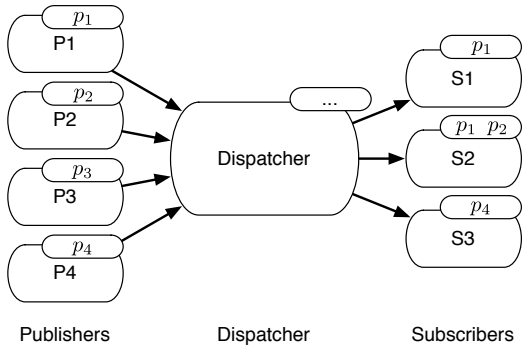

Figure 2. **Typical architecture for dispatching messages.** The use of tags to protect the privacy of the publishers is not possible because of a single dispatcher process.

```
delegate(PidReciever, Tag,
         Type) -> ok|fail
```

The function checks that the delegating process has the privilege to be delegated. `Type` specifies whether clearance or declassification is delegated.

To reduce the cost of IFC label checks, our Erlang IFC implementation *caches* the results of previous checks. This is possible because in our label model, the label of an Erlang process does not change over its lifetime. For a process to use a dynamic privilege, it must spawn a child and continue processing in the new context. After an IFC ("can flow to") check is performed between two processes, its result will always hold. We believe that this approach would be infeasible in other IFC systems because the overhead of instantiating isolated compartments, for example, when implemented as traditional OS processes, outweighs the benefit of faster label checking.

IV. Architecture

The core functionality of a micro-blogging service is the dispatch of messages between users. Figure 2 shows a typical architecture for dispatching messages. *Publishers* on the left-hand side send messages to a *dispatcher* component, which forwards a copy to each interested *subscriber*. A bug in the dispatcher implementation could send messages to the wrong subscribers or leak sensitive subscription choices to other users, thus violating our privacy guarantees (§II).

Using IFC, these threats can be mitigated by a system that tags, tracks and confines personal data. In the above example, publisher P1 could be tainted to only output messages with a tag p_1. Subscriber S3, which is not subscribed to P1, is not granted p_1^+ (clearance over p_1) and thus is unable to receive messages from P1, even if the dispatcher sends them to S3 by mistake. This means that privacy guarantee *G1* from §II is enforced.

Publisher-side. However, the approach above cannot be applied directly: the dispatcher would need to be tainted with all publisher tags to be able to receive published messages.

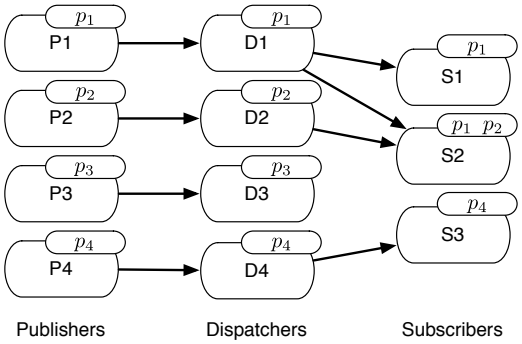

Figure 3. **Protecting publisher privacy by partitioning the dispatcher into per-publisher instances.**

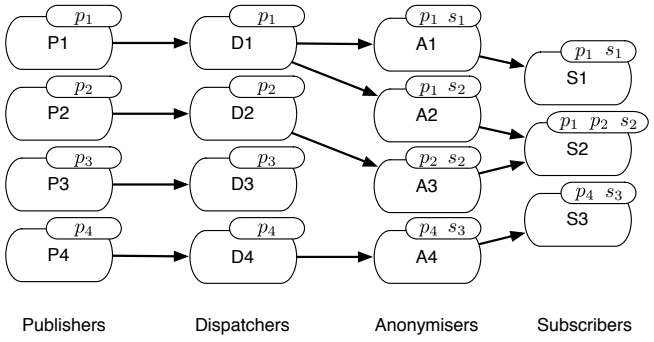

Figure 4. **Protecting subscriber privacy by further partitioning the dispatcher state with anonymisers.**

As a consequence, its output would be contaminated by all publisher tags, resulting in taint levels that would prevent it from communicating with any of the subscribers.

A simple solution is a forked server model, that uses a separate dispatcher per publisher, as shown in Figure 3. This is impractical if the cost of creating and switching between component instances is high. However, with Erlang's process model, switching processes is efficient, allowing many processes to be active at the same time with little overhead. These processes, when scheduled on a multi-core system, can increase performance with the number of cores.

Subscriber-side. While the above approach is sufficient to provide privacy guarantee *G1* to the publisher, the confidentiality of subscriptions is still at risk: a crash report from D1 may contain all of P1's subscribers. The report is protected only by the p_1 tag. It can thus be sent to any of P1's subscribers and reveal all of P1's subscribers, violating *G2*.

Subscriber S1 could, in principle, tag the subscription data with a tag s_1, similarly to the publisher case, to protect its privacy. However, as explained above, the dispatcher would hold subscription information from many subscribers, resulting in an overly-tainted dispatching process. To solve this issue and guarantee *G2*, we again partition the dispatcher into multiple instances so that each identifier of a subscriber is contained in its own protected instance (see Figure 4): D1 from Figure 3 is split into D1, A1 and A2 in Figure 4. To avoid disclosing its identity to D1, S1 first spawns a tainted A1 and then sends a message to D1, passing A1's `pid`, thus anonymising its subscription. A1 learns S1's `pid` but is tainted and thus cannot communicate with any other component except S1.

Finally, a subscription request must also be guaranteed only to reach the relevant publisher (*G3*). For this, the approach from §III can be applied. Alternatively, each publisher can allocate another tag and never share clearance over it. In the example above, P1 allocates a new tag q_1. Each subscriber only has to protect its authorisation request to P1 with q_1. Since only P1 has clearance q_1^+, only P1 can read the authorisation request, effectively enforcing *G3*.

A. Discussion

In the above architecture, components have to trade off two conflicting requirements: (1) they have to observe potentially sensitive data belonging to multiple users and (2) they have to respect user privacy using IFC. This makes a forked server model necessary, in which separate component instances per user are created. Although this is infeasible for conventional programming language threads or OS processes, it is reasonable in the context of Erlang processes. An Erlang process has an overhead of 1 KB of memory [14], thus effectively enabling millions of processes. Process creation time is also low, allowing 200,000 process creations per second even on modest hardware [15]. As a result, we can support per-user dispatcher and anonymiser components in our architecture and still scale to a large number of concurrently active users (as shown in §V).

A limitation of our approach is that it complicates the architecture of the application. A software designer must ensure that separate components are created per user and carefully manage the interaction between them to avoid unnecessary contamination. Messages sent by components must be tagged correctly according to the privacy policy goals. As typical of DIFC systems, components must also allocate tags and posses the correct privileges. Ongoing research on high-level policy specification for IFC and techniques for transparently inferring IFC policy may help manage this complexity [16].

Of course an implementation of a dispatcher is only one part of a fully-featured micro-blogging website. In this work, we ignore any presentation and storage functionality that would be commonly found in such an application. In practice, a database backend would be used to make all messages persistent and often this becomes a scalability bottleneck. Considering current architectures of scalable web applications, such as Twitter and Facebook, they rely on caching layers (e.g. memcached [17] and ehcache [18]) to avoid database requests on the critical path. Through caching, such applications try to keep data relating to active users in memory. In that sense, they resemble our

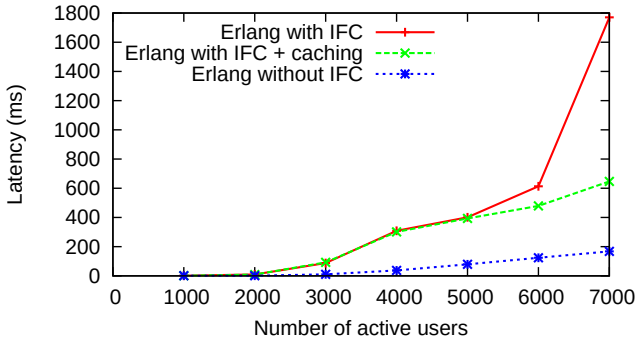

Figure 5. **Latency of message delivery with increasing number of users (i.e. subscribers and publishers).** Publishers produce messages at a rate of 10 msgs/sec.

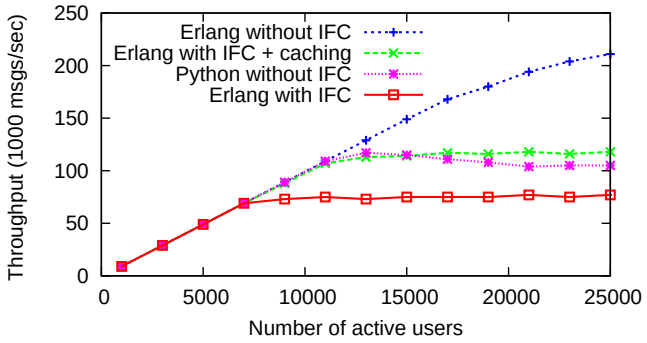

Figure 6. **Message throughput with increasing number of users (i.e. publishers and subscribers).** Publishers produce messages at a rate of 10 msgs/sec.

dispatching architecture after relevant user data was retrieved from the database to memory.

V. Evaluation

We evaluate the potential of our approach for providing an efficient mechanism for privacy enforcement. The goals of our evaluation are twofold: (1) to show that Erlang compares favourably to imperative languages commonly used for web development, and (2) to confirm that the extra IFC label checks do not introduce undue overhead.

We compare three implementations of our simple micro-blogging service: in Python, in Erlang and in Erlang with IFC. In each case, the implementation attempts to follow that language's best practice. The Python prototype was modelled after Figure 2. Because Python threads have significant overhead compared to those in Erlang, the Python implementation is synchronous to maximise throughput.[1] The Erlang implementation follows the architecture in Figure 3 and the Erlang with IFC design is shown in Figure 4. Unlike the Erlang with IFC prototype, the other two do not include labels and thus do not enforce any of the privacy requirements described in §II.

We use two performance metrics in our experiments: end-to-end message *throughput* and *latency*. We assume that all users, at all times, are about to send or receive messages. In other words, even if the application has millions of registered users, in these experiments we only focus on the ones that we consider to be constantly active.

All experiments were run on a dual processor Intel Xeon E5540 2.53 GHz machine with 24 GiB of memory running 64 bit Ubuntu Linux 9.04. We used the Erlang (BEAM) emulator version 5.6.5 and CPython version 2.6.2.

A. Experimental Results

In the first experiment, we measure the 90th percentile of latencies observed when dispatching a message from a publisher to its subscribers, as we increase the number of users in the system. In each run, for each user we instantiate a publisher and a subscriber process. In order to avoid the need to offload inactive users' data to a database, as discussed in §IV, the publication rate per publisher is 10 msgs/sec. Finally, we assume 10 subscriptions per subscriber.

The results are shown in Figure 5. Latencies in an under-loaded system are low: for 1000 users without IFC, it takes 0.5 ms to deliver a message and, with IFC, it takes 0.7 ms. The overhead of 0.2 ms is caused by the three label checks in the corresponding message exchanges en route from the publisher to its subscribers. As the number of users increases the load in the system, the switching and scheduling overhead also increases the latency significantly. This is mainly because each subscription in the IFC case requires an additional anonymiser, thus the total number of Erlang processes increases tenfold. The Python prototype (not shown in the graph) remained consistently below the 1 ms range because of the structured, synchronous manner in which the code delivers messages. This shows that intimate knowledge about the dispatching behaviour of a system can be used to optimise for low latency message delivery.

The second experiment measures the achieved throughput in terms of published messages/second. In Figure 6, we again have a fixed message rate of 10 msgs/sec per user as we increase the number of users. With 10 subscriptions per user, the total delivery rate in Figure 6 reaches 1 million msgs/sec. Here the caching of the label checks leads to a 30% increase in the throughput at saturation compared to regular IFC. Without IFC, Erlang's throughput at saturation is 30% higher than the cached version. Python performs similarly to Erlang with cached IFC checks even though it is confined to a single OS process. This can be attributed to the label checks being of similar complexity to the application's procedural code: an application with more features (e.g. complex filtering of message content) would have lowered the load on the communication channels.

Note that these results constitute a worst case scenario:

[1]Note that CPython implementations currently cannot make efficient use of multi-core architectures.

all IFC-related code operates on top of the runtime and is written in Erlang itself. An implementation within the runtime not only could enforce the use of IFC mechanisms, but would have the potential to improve performance further. Critical operations, such as label checks, could be also optimised in more efficient procedural languages.

VI. Related Work

Privacy in Social Web Applications. A great deal of research has been done into the user-facing aspects of privacy in social networking applications, both in terms of the risks to users [19], and techniques for policy specification [20]. However, the insubstantial impact, and then demise of the W3C P3P project [21], suggests that implementation and enforcement of privacy controls tends to be done in an ad hoc manner. To address this gap, a number of systematic approaches that rely on IFC have recently been proposed to offer the necessary privacy guarantees.

Information Flow Control. IFC originated in the military domain when implementing Multi Level Security (MLS). Myers and Liskov's seminal work [10] extended the label model to provide Decentralised Information Flow Control (DIFC). In this model, new labels can be created at runtime by unprivileged application components, which can delegate declassification privileges to other components. The availability of fine-grained labels with decentralised declassification makes the "principle of least privilege" enforceable [22].

Approaches for applying IFC to applications differ in (a) how they define labels to protect flows, (b) how they track labels as data is processed and copied, and (c) when labels are checked at certain propagation boundaries.

OS-level IFC, such as Asbestos [1], Flume [3] and HiStar [2], define dataflow boundaries around OS processes. Each process is assigned a label at runtime and labels form a lattice under a "can-flow-to" ordering relation. For example, in Flume labels are attached by spawning processes and by creating communication channels. Taint tracking is provided by the system that performs IFC checks between processes and channels. It guarantees that tags are preserved as data is exchanged among processes. As discussed previously, OS-level IFC is too coarse-grained for our goals.

Language-level IFC provides protection at the granularity of individual variables. Jif [10] extends Java to control data flow. It uses a static approach, in which labels are attached to variable declarations and checked by a compiler. This makes it unsuitable for our scenario because new users (and associated data) are created dynamically at runtime. On the other hand, proposals for checking variable labels at runtime [4], [6], [23] impact performance because they introduce a constant overhead, as described below.

xBook [11] is a framework for developing extensions for social networking platforms. It addresses the problem that such extensions have access to personal information, yet they are not confined. xBook uses the Jif label model to monitor and limit communication at runtime. Extensions are written in ADSafe, a subset of JavaScript that prevents two components from communicating unless they use the provided asynchronous message passing interface that verifies label compatibility. xBook highlights the applicability of Erlang for enforcing IFC; the programming style imposed by xBook closely follows the mainstream Erlang paradigm. In addition, Erlang offers superior scalability on multi-core platforms and a fast message passing interface. Thus, it is more suitable to support the high message rates and a large number of isolated components, as required in our micro-blogging scenario.

SIF [4] is similar to xBook in that it aims to support general purpose web applications with privacy guarantees. It provides an alternative Java Servlet Framework where servlets are written in Jif. In order to support new users, SIF augments Jif with principals and labels that are checked at runtime. The authors do not report to what extent variable-level runtime checks affect the performance of CPU-bound applications.

Trishul [23] and *Resin* [6] provide dynamic, fine-grained IFC by modifying the language runtime to attach labels to values. In Resin, labels do not form a lattice but are sets of policy objects. For example, a password could be annotated by a policy object preventing it from being sent over an unencrypted connection. When data is copied, the runtime maintains the link with the policy. Policies are enforced when data crosses specific boundaries, such as marshalling a string before a socket write. This approach requires programmers to define these boundaries explicitly. If a dangerous boundary is not identified, private user data could be revealed. By instrumenting message passing in Erlang, we avoid this problem in our work.

VII. Future Work

In this paper, we address only one component required in a micro-blogging service, namely the message dispatching system. Our focus on the dispatching engine means that we ignore other parts of a practical web application, such as low-level communication, message marshalling, user management and the web front-end. Integrating our message queue with an Erlang-based web server would allow us to apply previous research on securing web servers using IFC [6]. Exploiting IFC in conjunction with Erlang's seamless distribution features can potentially secure distributed applications.

We also do not consider data persistence using a database back-end. Integration of IFC security with persistent storage is still in its infancy and constitutes an interesting area for future research. It is orthogonal to the runtime aspects described in this work. In particular, we plan to augment Erlang's distributed Mnesia database with IFC support.

From a language perspective, Erlang is not a pure functional language: there are, in fact, several library functions that allow Erlang processes to communicate without exchanging messages, e.g. by using non-private term stores or by accessing external resources such as files. We plan to secure these features in the language runtime to achieve complete isolation.

Moreover, aspects that facilitate lightweight processes in Erlang are beginning to appear in other languages: we are keen to test the applicability of the IFC model in environments such as Stackless Python and the Kilim [15] actor environment for Java.

VIII. Conclusions

We have shown the viability of applying (decentralised) Information Flow Control (IFC) to preserve the privacy of users within a social micro-blogging application, both from the publishers' and the subscribers' perspective. Our implementation uses the Erlang programming language and exploits its key features, such as shared-nothing computation, message passing and lightweight processes. These features make Erlang an ideal choice for implementing IFC with low overhead and minimal effort. We have described the architecture of a Twitter-like dispatching service in Erlang that creates fresh instances of components to isolate data belonging to different users. Our evaluation results show that applying IFC in this way achieves uniform privacy preservation, even in an application with significant numbers of concurrent users, at a reasonable cost.

Acknowledgements

We would like to thank Cristian Cadar and Steven Hand for comments on an earlier paper draft. This work was supported by grants EP/F042469 and EP/F044216 ("SmartFlow: Extendable Event-Based Middleware") from the UK Engineering and Physical Sciences Research Council (EPSRC).

References

[1] P. Efstathopoulos, M. Krohn, S. VanDeBogart *et al.*, in *ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, pp. 17–30.

[2] N. Zeldovich, E. Kohler *et al.*, "Making information flow explicit in HiStar," in *OSDI '06*, Seattle, WA, USA.

[3] M. Krohn, A. Yip, M. Brodsky *et al.*, "Information flow control for standard OS abstractions," in *ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, USA, 2007, pp. 321–334.

[4] S. Chong, K. Vikram, and A. C. Myers, "SIF: enforcing confidentiality and integrity in web applications," in *USENIX Security Symposium*, Boston, MA, USA, 2007, pp. 1–16.

[5] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *POPL '99*, San Antonio, TX, USA.

[6] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving application security with data flow assertions," in *ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MA, USA, 2009, pp. 291–304.

[7] J. Armstrong, *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[8] Twitter Website, www.twitter.com.

[9] G. Clarke, "Bill Gates hits Twitter, re-opens Facebook," The Register, 2010.

[10] A. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 4, 2000.

[11] S. Kapil, B. Sumeer, and L. Wenke, "xBook: Redesigning privacy control in social networking platforms," in *USENIX Security Symposium*, Montreal, Canada, 2009.

[12] M. Miglivacca, I. Papagiannis, D. Eyers, B. Shand, J. Bacon, and P. Pietzuch, "High-performance event processing with information security," in *USENIX Annual Technical Conference*, Boston, MA, USA, 2010.

[13] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken, "Implementing multiple protection domains in Java," in *USENIX Annual Technical Conference*, Berkeley, CA, USA, 1998.

[14] J. Haln, R. Karlsson, and M. Nilsson, "Performance measurements of threads in Java and processes in Erlang," Ericsson, Tech. Rep., 1998.

[15] S. Srinivasan and A. Mycroft, "Kilim: Isolation-typed actors for Java," in *European Conference on Object-Oriented Programming (ECOOP)*, Paphos, Cyprus, 2008.

[16] P. Efstathopoulos and E. Kohler, "Manageable fine-grained information flow," in *European Conference on Computer Systems (EuroSys)*. Glasgow, UK: ACM, 2008.

[17] Memcached Website, www.memcached.org.

[18] Ehcache Website, www.ehcache.org.

[19] D. Rosenblum, "What anyone can know: The privacy risks of social networking sites," *IEEE Security and Privacy*, vol. 5, no. 3, pp. 40–49, 2007.

[20] G. Danezis, "Inferring privacy policies for social networking services," in *AISec*, 2009, pp. 5–10.

[21] L. Cranor, M. Langheinrich, M. Marchiori, and J. Reagle, "The platform for privacy preferences 1.0 (P3P 1.0) specification," W3C Recommendation, 2002. [Online]. Available: www.w3.org/TR/P3P

[22] M. Krohn, P. Efstathopoulos, C. Frey, and all, "Make least privilege a right (not a privilege)," in *Hot Topics in Operating Systems (HOTOS)*, Santa Fe, NM, USA, 2005, pp. 21–21.

[23] S. Nair, P. Simpson *et al.*, "A virtual machine based information flow control system for policy enforcement," *Electronic Notes in Theoretical Computer Science*, vol. 197, no. 1, 2008.