Hails: Protecting Data Privacy in Untrusted Web Applications

Daniel B. Giffin, Amit Levy, Deian Stefan
David Terei, David Mazières, John C. Mitchell
Stanford

Alejandro Russo
Chalmers

Abstract

Modern extensible web platforms like Facebook and Yammer depend on third-party software to offer a rich experience to their users. Unfortunately, users running a third-party "app" have little control over what it does with their private data. Today's platforms offer only ad-hoc constraints on app behavior, leaving users an unfortunate trade-off between convenience and privacy. A principled approach to code confinement could allow the integration of untrusted code while enforcing flexible, end-to-end policies on data access. This paper presents a new web framework, Hails, that adds mandatory access control and a declarative policy language to the familiar MVC architecture. We demonstrate the flexibility of Hails through `GitStar.com`, a code-hosting website that enforces robust privacy policies on user data even while allowing untrusted apps to deliver extended features to users.

1 Introduction

Extensible web platforms that run third-party *apps* in a restricted manner represent a new way of developing and deploying software. Facebook, for example, has popularized this model for social networking and personal data, while Yammer provides a similar platform geared toward enterprises. The functionality available to users of such sites is no longer the product of a single entity, but the combination of a potentially trustworthy platform running code provided by less-trusted third parties.

Many apps are only useful when they are able to manipulate sensitive user data—personal information such as financial or medical details, or non-public social relationships—but once access to this data has been granted, there is no holistic mechanism to constrain what the app may do with it. For example, the Wall Street Journal reported that some of Facebook's most popular apps, including Zynga's FarmVille game, had been transmitting users' account identifiers (sufficient for obtaining personal information) to dozens of advertisers and online tracking companies [38].

In this conventional model, a user sets privacy settings regarding specific apps, or classes of apps. However, users who wish to benefit from the functionality of an app are forced to guess what risk is posed by granting an app access to sensitive information: the platform cannot provide any mechanistic guarantee that the app will not, for example, mine private messages for ad keywords or credit card numbers and export this information to a system run by the app's developer.

Even if they are aware of how an app behaves, users are generally poorly equipped to understand the consequences of data exfiltration. In fact, a wide range of sophisticated third-party tracking mechanisms are available for collecting and correlating user information, many based only on scant user data [27].

In order to protect the interests of its users, the operator of a conventional web platform is burdened with implementing a complicated security system. These systems are usually ad-hoc, relying on access control lists, human audits of app code, and optimistic trust in various software authors. Moreover, each platform provides a solution different from the other.

To address these problems, we have developed an alternate approach for confining untrusted apps. We demonstrate the system by describing `GitStar.com`, a social code hosting website inspired by GitHub. GitStar takes a new approach to the app model: we host third-party apps in an environment designed to protect data. Rather than ask users whether to disclose their data to certain apps, we support policies that restrict information flow into and out of apps, allowing them to give up communication privileges in exchange for access to user data.

GitStar is built on a new web framework called Hails. While other frameworks are geared towards monolithic web sites, Hails is explicitly designed for building web *platforms*, where it is expected that a site will comprise many mutually-distrustful components written by various entities.

Hails is distinguished by two design principles. First, access policies should be specified declaratively alongside data schemas, rather than strewn throughout the codebase as guards around each point of access. Second, access

policies should be mandatory even once code has obtained access to data.

The first principle leads to an architecture we call model–policy–view–controller (MPVC), an extension to the popular model–view–controller (MVC) pattern. In MVC, models represent a program's persistent data structures. A view is a presentation layer for the end user. Finally, controllers decide how to handle and respond to particular requests. The MVC paradigm does not give access policy a first-class role, making it easy for programmers to overlook checks and allow vulnerabilities [34]. By contrast, MPVC explicitly associates every model with a policy governing how the associated data may be used.

The second principle, that data access policies should be mandatory, means that policies must follow data throughout the system. Hails uses a form of mandatory access control (MAC) to enforce end-to-end policies on data as it passes through software components with different privileges. While MAC has traditionally been used for high-security and military operating systems, it can be applied effectively to the untrusted-app model when combined with a notion of decentralized privileges such as that introduced by the decentralized label model [32].

The MAC regime allows a complex system to be implemented by a reconfigurable assemblage of software components that do not necessarily trust each other. For example, when a user browses a software repository on GitStar, a code-viewing component formats files of source code for convenient viewing. Even if this component is flawed or malicious, the access policy attached to the data and enforced by MAC will prevent it from displaying a file to users without permission to see it, or transmitting a private file to the component's author. Thus, the central GitStar component can make repository contents available to any other component, and users can safely choose third-party viewers based solely on the features they deliver rather than on the trustworthiness of their authors.

A criticism of past MAC systems has been the perceived difficulty for application programmers to understand the security model. Hails offers a new design point in this space by introducing MAC to the popular MVC pattern and binding access control policy to the model component in MPVC. Because GitStar is a public site in production use by more than just its developers, we are able to report on the experiences of third-party app authors. While our sample is yet small, our experience suggests MAC security does not impede application development within an MPVC framework.

The remainder of this paper describes Hails, GitStar, and several add-on components built for GitStar. We discuss design patterns used in building Hails applications. We then evaluate our system, provide a discussion, survey related work, and conclude.

2 Design

The Hails MPVC architecture differs from traditional MVC frameworks such as Rails and Django by making security concerns explicit. An MVC framework has no inherent notion of security policy. The effective policy results from an ad-hoc collection of checks strewn throughout the application. By contrast, MPVC gives security policies a first-class role. Developers specify policies in a domain-specific language (DSL) alongside the data model. Relying primarily on language-level security, the framework then enforces these policies system-wide, regardless of the correctness or intentions of untrusted code.

MPVC applications are built from mutually distrustful components. These components fall into two categories: *MPs*, comprising model and policy logic, and *VCs*, comprising view and controller logic. An MP provides an API through which other components can access a particular database, subject to its associated policies.

MPs and VCs are explicitly segregated. An MP cannot interact directly with a user, while a VC cannot access a database without invoking the corresponding MP. Our language-level confinement mechanism enforces MAC, guaranteeing that a data-model's policy is respected throughout the system. For example, if an MP specifies that "only a user's friends may see his email address," then a VC (or other MP) reading a user's email address loses the ability to communicate over the network except to the user's friends (who are allowed to see that email address).

Figure 1 illustrates the interaction between different application components in the context of GitStar. Two MPs are depicted: GitStar, which manages projects and `git` data; and Follower, which manages a directional relationship between users. Three VCs are shown invoking these modules: a source-code viewer, a `git`-based wiki, and a bookmarking tool. Each VC provides a distinct interface to the same data. The Code Viewer presents syntax-highlighted source code and the results of static analysis tools such as splint [19]. Using the same MP, the wiki VC interprets text files using markdown to transform articles into HTML. Finally, the bookmarking VC leverages both MPs to give users quick access to projects owned by other users whom they follow.

Because an application's components are mutually distrustful, MPVC also leads to greater extensibility. Any of the VCs depicted in Figure 1 could be developed after the fact by someone other than the author of the MPs. Anyone who doesn't like GitStar's syntax highlighting is

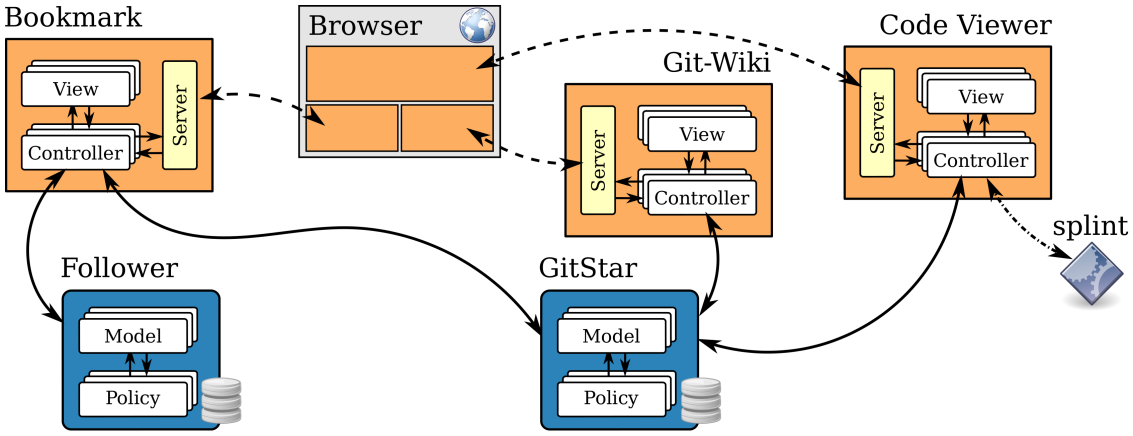

Figure 1: Hails platform with three VCs and two MPs. Dashed lines denote HTTP communication; solid lines denote local function calls; dashed-dotted lines denote communication with OS processes. MPs and VCs are confined at the programming language level; OS processes are jailed and only communicate with invoking VCs; the Browser is restricted to communicating with the target VCs.

free to run a different code viewer. No special privileges are required to access an MP's API, because Hails's MAC security continues to restrict what code can do with data even after gaining access to the data.

2.1 Principals and privileges

Hails specifies policy in terms of *principals* who are allowed to read or write data. There are four types of principal. Users are principals, identified by user-names (e.g., `alice`). Remote web sites that an app may communicate with are principals, identified by URL (e.g., `http://maps.google.com:80/`). Each VC has a unique principal, by convention starting with prefix "`@`", and each MP has a unique principal starting "`_`" (e.g., `@Bookmark` and `_GitStar` for the components in Figure 1).

An example policy an MP may want to enforce is "user `alice`'s mailing address can be read only by `alice` or by `http://maps.google.com:80/`." Such a policy would allow a VC to present `alice` her own address (when she views her profile) or to fetch a google map of her address and present it to her, but not to disclose the address or map to anyone else. For maximum flexibility, read and write permissions can each be expressed using arbitrary conjunctions and disjunctions of principals. Enforcing such policies requires knowing what principals an app represents locally and what principals it is communicating with remotely.

Remote principals are ascertained as one would expect. Hails uses a standard cookie-based authentication facility; a browser presenting a valid session cookie represents the logged-in user's principal. When VCs or MPs initiate outgoing requests to URLs, Hails considers the remote server to act on behalf of the URL principal of the web site.

Within the confines of Hails, code itself can act on behalf of principals. The trusted Hails runtime supports unforgeable objects called *privileges* with which code can assert the authority of principals. Hails passes appropriate privilege objects to MPs and VCs upon dynamically loading their code. For example, the GitStar MP is granted the `_GitStar` privilege. When a user wishes to use GitStar to manager her data, the policy on the data in question must specify `_GitStar` as a reader and writer so as to give GitStar permission to read the data and write it to its database should it chose to exercise its `_GitStar` privileges.

2.2 Labels and confinement

Hails associates a security policy with every piece of data in the system, specifying which principals can read and write the data. Such policies are known as *labels*. The particular labels used by Hails are called *DC labels*. We described and formalized DC labels in a separate paper [39], so limit our discussion to a brief overview of their format and use in MAC. We refer readers to the full DC labels paper for more details.

A DC label is a pair of positive boolean formulas over principals: a *secrecy* formula, specifying who can read the data, and an *integrity* formula, specifying who can write it. For example, a file labeled $\langle \texttt{alice} \vee \texttt{bob}, \texttt{alice} \rangle$ specifies that `alice` or `bob` can read from the file and only `alice` can write to the file. Such a label could be used by the Code Viewer of Figure 1 when fetching `alice`'s source code. The label allows the VC to present the source code to the project participants, `alice` and `bob`, but not disseminate it to others.

The trusted runtime checks that remote principals satisfy any relevant labels before permitting communication.

For instance, data labeled $\langle \texttt{alice} \vee \texttt{bob}, \texttt{alice} \rangle$ cannot be sent to a browser whose only principal is `charlie`. The actual checks performed involve verifying logical implications. Data labeled $\langle S, I \rangle$ can be sent to a principal (or combination of principals) p only when $p \Longrightarrow S$. Conversely, remote principal p can write data labeled $\langle S, I \rangle$ only when $p \Longrightarrow I$. Given these checks, $\langle \text{TRUE}, \text{TRUE} \rangle$ labels data readable and writable by any remote principal, i.e., the data is public, while $p = \text{TRUE}$ means a remote party is acting on behalf of no principals.

The same checks would be required for local data access if code had unrestricted network access. Hails could only allow code to access data it had explicit privileges to read. For example, code without the `alice` privilege should not be able to read data labeled $\langle \texttt{alice}, \text{TRUE} \rangle$ if it could subsequently send the data anywhere over the network. However, Hails offers a different possibility: code without privileges can read data labeled $\langle \texttt{alice}, \text{TRUE} \rangle$ so long as it first gives up the ability to communicate with remote principals other than `alice`. Such communication restrictions are the essence of MAC.

To keep track of communication restrictions, the runtime associates a *current label* with each thread. The utility of the current label stems from the transitivity of a partial order called "*can flow to*." We say a label $L_1 = \langle S_1, I_1 \rangle$ *can flow to* another label $L_2 = \langle S_2, I_2 \rangle$ when $S_2 \Longrightarrow S_1$ and $I_1 \Longrightarrow I_2$—in other words, any principals p allowed to read data labeled L_2 can also read data labeled L_1 (because $p \Longrightarrow S_2 \Longrightarrow S_1$) and any principals allowed to write data labeled L_1 can also write data labeled L_2 (because $p \Longrightarrow I_1 \Longrightarrow I_2$).

A thread can read a local data object only if the object's label can flow to the current label; it can write an object only when the current label can flow to the object's. Data sent over the network is always protected by the current label. (Data may originate in a labeled file or database record but always enters the network via a thread with a current label.) The transitivity of the *can flow to* relation ensures no amount of shuffling data through objects can result in sending the data to unauthorized principals.

A thread may adjust the current label to read otherwise prohibited data, only if the old value can flow to the new value. We refer to this as *raising* the current label. Allowing the current label to change without affecting security requires very carefully designed interfaces. Otherwise, labels themselves could leak information. In addition, threads could potentially leak information by not terminating (so called "termination channels") or by changing the order of observable events (so called "internal timing channels"). GitStar is the first production system to address these threats at the language level. We refer interested readers to [41] for the details and security proof of our solution.

A final point is that Hails prevents the current label from accumulating restrictions that would ultimately prevent the VC from communicating back to the user's browser. In MAC parlance, a VC's *clearance* is set according to the user making the request, and serves as an upper bound on the current label. Thus, an attempt to read data that could never be sent back to the browser will fail, confining observation to a "need-to-know" pattern.

2.3 Model-Policy (MP)

Hails applications rely on MPs to define the application's data model and security policies. An MP is a library with access to a dedicated database. The MP specifies what sort of data may be stored in the database and what access-control policies should be applied to it. Though MPs may contain arbitrary code, we provide and encourage the use of a DSL, described in Section 2.3.1, for specifying data policies in a concise manner.

The Hails database system is similar to and built atop MongoDB [7]. A Hails *database* consists of a set of *collections*, each storing a set of *documents*. In turn, each document contains a set of *fields*, or named values. Some fields are configured as *keys*, which are indexed and identify the document in its collection. All other fields are non-indexed *elements*.

An MP restricts access to the different database layers using labels. A static label is associated with every database, restricting who can access the collections in the database and, at a coarse level, who can read from and write to the database. Similarly, a static label is associated with a collection, restricting who can read and write documents in the collection. The collection label additionally serves the role of protecting the keys that identify documents—a computation that can read from a collection can also read all the key values.

2.3.1 Automatic, fine-grained labeling

In many web applications, dynamic fine-grained policies on documents and fields are desired. Consider the user model shown in Figure 2: each document contains fields corresponding to a user-name, email address, and list of friends. In this scenario, the Follower MP may configure user-names as keys in order to allow VCs to search for `alice`'s profile. Additionally, the MP may specify database and collection labels that restrict access to documents at a coarse grained level. However, these static labels are not sufficient to enforce fine grained dynamic policies such as "only `alice` may modify her profile information" and "only her friends (`bob`, `joe`, etc.) may see her email address."

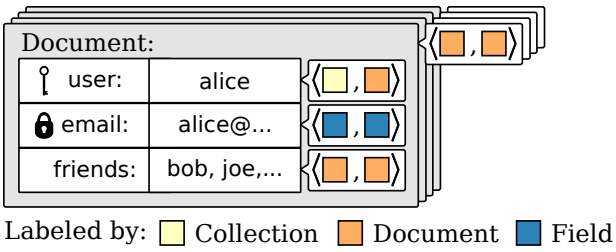

Figure 2: Hails user documents. Each document is indexed by a key (`user`-name) and contains the user's `email` address and list of `friends`. Documents and `email` fields are dynamically labeled using a data-dependent policy; the secrecy of the `user` key and is protected by the static collection label, the document label protects its integrity. The "unlabeled" `friends` fields are protected by their corresponding document labels.

Hails introduces a novel approach to specifying document and field policies by assigning labels to documents and fields as a function of the document contents itself.[1] This approach is based on the observation that, in many web applications, the authoritative source for who should access data resides in the data itself. For example, in Figure 2, the user-name and friends field values can be used to specify the document and field policies mentioned above: `alice`'s document is labeled $\langle \text{TRUE}, \texttt{alice} \vee \texttt{\_Follower} \rangle$, while the email field value is labeled $\langle \texttt{alice} \vee \texttt{bob} \vee \texttt{joe} \vee \cdots \vee \texttt{\_Follower}, \text{TRUE} \rangle$. The document label guarantees that only `alice` or the MP can modify any of the constituent fields. The label on the email-address field additionally guarantees that only `alice`, the MP, or her friends can read her address.

Hails's data-dependent "automatic labeling" simplifies reasoning about security policies and localizes label logic to a small amount of source code. Figure 3 shows the implementation of the Follower users policy, as described above, using our DSL. Specifying static labels on the database and collections is simply done by setting the respective `readers` and `writers` in the `database` and `collection` sections. Similarly, setting a document or field label is done using a function from the document itself to a pair of `readers` and `writers`.

2.3.2 Database access and policy application

MP policies are applied on every database insert. When a thread attempts to insert a document into an MP collection, the Hails runtime first checks that that the thread can read and write to the database and collection, by comparing the thread's current label with that of the database and collection. Subsequently, the field- and document-labeling policy functions are applied to the document and fields. If the policy application succeeds—it may fail if the thread cannot label data as requested—the Hails runtime removes all the labels on the document and performs the write.

[1] These labeling functions are pure: they cannot perform side effects and must always return the same value for the same input.

```
database $ do
  -- Set database label:
  access $ do
    readers ==> anybody
    writers ==> anybody

-- Set policy for new "users" collection:
collection "users" $ do
  -- Set collection label:
  access $ do
    readers ==> anybody
    writers ==> anybody
  -- Declare user field as a key:
  field "user" key
  -- Set document label, given document doc:
  document $ λdoc -> do
    readers ==> anybody
    writers ==> ("user" `from` doc) \/ _Follower
  -- Set email field label, given document doc:
  field "email" $ labeled $ λdoc -> do
    readers ==> ("user" `from` doc)
                \/ fromList ("friends" `from` doc)
                \/ _Follower
    writers ==> anybody
```

Figure 3: DSL-specification of the Follower users policy. Here, `anybody` corresponds to the boolean formula TRUE; `fromList` converts a list of principals to a disjunction of principals; and, `"x" `from` doc` retrieves the value of field `x` from document `doc`. The `database` and `collection` labels are static. Field `user` is configured as a key. Finally, each document and `email` field is labeled according to a function from the document itself to a set of `readers` and `writers`.

Hails also allows threads to insert already-labeled documents (e.g., documents retrieved from another MP or directly from the user). As before, when inserting a labeled document, the MP database and collection must be readable and writable at the current label. Different from above, the thread does not need to apply the policy functions; instead, the Hails runtime verifies that the labels on fields and the document agree with those specified by the MP. Finally, if the check succeeds, the Hails runtime strips the labels and performs the write.

Application components, including VCs, can fetch elements from an MP's database collection by specifying a query predicate. Predicates are restricted to solely involve indexed keys (or be TRUE). Similar to insert, when performing a fetch, the runtime first checks that that the thread can read from the database and collection. Next, the documents matching the predicate are retrieved from

the database. Finally, the field- and document-labeling policy functions are applied to each document and field; the resultant labeled documents are returned to the invoking thread.

Hails supports additional database operations, including update and delete. These operations are similar to those of MongoDB [7], though Hails enforces the MP's policies whenever its database is accessed. Since the restrictions on most operations are similar to those of insert and fetch, we do not describe them further.

2.4 View-Controller (VC)

VCs interact with users. Specifically, controllers handle user requests, and views present interfaces to the user. However, VCs do not define database-backed models. Instead, a controller invokes one or more MPs when it needs to store or retrieve user data. This data can also be passed on to views when rendering user interfaces.

Each VC is a standalone process, linked against the MP libraries it depends on to provide a data model. The VC author solely provides a definition for a *main* controller, which is a function from an HTTP request to an HTTP response. This function may perform side-effects: it may access a database-backed model by invoking an MP, read files from the labeled filesystem, etc. Hails uses language-level confinement to prevent the VC and MPs it invokes from modifying or leaking data in violation of access permissions. Additionally, since each VC is a process, OS-level isolation and resource management mechanisms can be leveraged to enforce additional platform-specific policies.

At the heart of every VC is the Hails HTTP server. The server, a privileged part of the trusted computing base (TCB), receives HTTP requests and invokes the main VC controller to process them. When a request is from an authenticated user, the server sets the `X-Hails-User` header to the user-name and attests to the request's contents for the benefit of MPs that care about request provenance and integrity. In turn, the main controller processes the supplied request, by potentially calling into MPs to interact with persistent state, and finally returns an HTTP response. The server returns the provided response to the browser on the condition that it depend only on data the user is permitted to observe.

In carrying out their duties, many VCs rely on communication with external web sites. Hence, Hails applications have access to an HTTP client. Before establishing a connection, and on each read or write, the HTTP client checks that the current label of the invoking thread is compatible with the remote server principal. In practice, this means VCs can only communicate with external hosts when they have not read any sensitive data or they have only read data *explicitly* labeled for the external server.

Additionally, VCs may need to run arbitrary programs. For example, as highlighted in Figure 1, GitStar's Code Viewer relies on splint, a standalone C program, to flag possible coding errors. Addressing this need, Hails provides a mechanism for spawning confined Linux processes with no network access, no visibility of other processes, and no writable file system shared by other processes. Each such processes is governed by a fixed label, namely the VC's current label at the time the program was spawned. In turn, labeled file handles can be used to communicate with the process, subject to the restrictions imposed by the current thread's label.

2.5 Life-cycle of an application

In this section, we use GitStar's deployment model to illustrate the life-cycle of a Hails application from development, through deployment, to a user-request.

2.5.1 Application development and deployment

A third-party application developer may introduce a new data model to the GitStar platform by writing an MP. For example, the Follower MP shown earlier specifies a data-model for storing a relation between users, as well as a policy specifying who is able to read, create and modify those relationships. Once written, the developer uploads the library code to the GitStar servers where it is compiled and installed. The platform administrator generates a unique privilege for the new MP and associates it with a specific database in a globally-accessible configuration file. Subsequently, any Hails code may import the MP, which when invoked, will be loaded with its privilege and database-access.

The third-party developer may build a user interface to the newly-created model by writing a VC controller. As with MPs, developers upload their VC code to the GitStar servers where it is compiled and linked against any MPs it depends on. Thereafter, a program called `hails`, which contains the Hails runtime and HTTP server, is used to dynamically load the main VC controller and service user requests on a dedicated TCP port.

While in this example both the VC and MP were implemented by a single developer, third-party developers can implement applications consisting solely of a VC that interacts with MPs created by others. In fact, in GitStar, most applications are simply VCs that use the GitStar MP to manage projects and retrieve `git` objects. For example, the `git`-based wiki application, as shown in Figure 1, is simply a VC that displays formatted text from a particular branch of a `git` repository.

2.5.2 An example user request

When an end-user request is sent to the GitStar platform, an HTTP proxy routes the request to the appropriate VC HTTP server based on the hostname in the request.

The Hails server receiving the forwarded request invokes the main controller of the corresponding VC in a newly spawned thread. The controller is executed with the VC's privileges and sanitized request. The HTTP server sanitizes the incoming request by removing headers such as `Cookie`; it also sets the `X-Hails-User` header to the user-name, if the request is from an authenticated user.

The main controller may be a simple request handler that returns a basic HTML page without accessing any sensitive data (e.g., an index or about page). A more interesting VC may access sensitive user data from an MP database before computing a response. In this case, the VC invokes the MP by performing a database operation such as insert or fetch. The invocation consists of several steps. First, the Hails runtime instantiates the MP with its privilege and establishes a connection to the associated database, as specified in the global configuration file. Then, the MP executes the database operations supplied by the VC, and, in coordination with the Hails runtime, labels the data according to its policies. While some database operations are not sensitive (e.g., accessing a public `git` repository in GitStar), many involve private information. In such cases, the database operation will also "raise" the current label of the VC, and thereby affect all its future communication.

When a VC produces an HTTP response, the runtime checks that the current label, which reflects all data accesses or other sensitive operations, is still compatible with the end-user's browser. For example, if `alice` has sent a request to the Code Viewer asking for code from a private repository, the response produced by Code Viewer will only be forwarded by the Hails server if the final label of Code Viewer can flow to $\langle \texttt{alice}, \text{TRUE} \rangle$

On the client side, the Hails browser extension, detailed in Section 3.3, restricts all incoming responses and outgoing requests according to the response label. For example, if the Code Viewer returns a response labeled $\langle \texttt{alice} \vee \texttt{http://code.google.com}, \text{TRUE} \rangle$, the rendered page may retrieve scripts for prettifying code from `http://code.google.com`, but not retrieve images from `http://haskell.org`. On the other hand, a publicly labeled response imposes no restrictions on the requests triggered by the page.

2.6 Trust assumptions

The Hails runtime, including the confinement mechanism, HTTP server, and libraries are part of the TCB. Parts of the system, namely our labels and confinement mechanism, have been formalized in [30, 39–41]. We remark that different from other work, our language-level concurrent confinement system is sound even in the presence of termination and timing covert channels [41]. However, similar to other MAC systems (e.g., [24]), we assume that the remaining Hails components are correct and that the underlying OS and network are not under the control of an attacker.

By visiting a web page, the MPs invoked by the VC presenting the page are trusted by users to preserve their privacy. This is a consequence of MPs being allowed to manage all aspects of their database. However, one MP cannot declassify data managed by another, and thus users can choose to use trustworthy MPs. Facilitating this choice, Hails makes the MP policies and dependency relationships between VCs and MPs available for inspection.

Since a user can choose to invoke a VC according to the MPs it depends on, VCs are *mostly* untrusted. On the server-side, VCs cannot exfiltrate user data from the database without collusion from an MP the user has trusted. Nevertheless, VCs cannot be considered completely untrusted since they directly interact with users through their browser. Unfortunately, in today's browsers, even with our client-side sandbox, a malicious VC can coerce a user to declassify sensitive data.

3 Implementation

Hails employs a combination of language-level, OS-level and browser-level confinement mechanisms spread across all layers of the application stack to achieve its security goals. Most notably, we use a language-level information flow control (IFC) framework to enforce fine-grained policies on VCs and MPs. This section describes this framework, and some of the implementation details of our OS and browser confinement mechanisms.

3.1 Language-level confinement

Hails applications are written in Haskell. Haskell is a statically- and strongly-typed, memory-safe language. Crucially, Haskell's type system distinguishes operations involving side-effects (such as potentially data-leaking I/O) from purely-functional computation. As a consequence, for example, compiling a VC's main controller with an appropriately specified type is sufficient to assert that the VC cannot perform arbitrary network communication.

Hails relies on the safety of the Haskell type system when incorporating untrusted code. However, like other languages, Haskell "suffers" from a set of features that allow programmers to perform unsafe, but useful, actions (e.g., type coercion). To address this, we ex-

tended the Glasgow Haskell Compiler (GHC) with `Safe Haskell` [44]. `Safe Haskell`, deployed with GHC as of version 7.2, guarantees type safety by removing the small set of language features that otherwise allow programs to violate the type system and break module boundaries.

With this change, Haskell permits the implementation of language-level dynamic IFC as a library. Accordingly, we implemeted LIO [40], which employs the label-tracking and confinement mechanisms of Section 2.2. Despite sharing many abstractions with OS-level IFC systems, such as HiStar [46] and Flume [17], LIO is more fine-grained (e.g., it allows labels to be associated with values, such as documents and email addresses) and thus better suited for web applications.

We believe the Hails architecture is equally realizable in other languages, though possibly with less backward compatibility. For example, JiF [33], Aeolus [5] and Breeze [15] provide similar confinement guarantees and are also good choices. However, to use existing libraries JiF and Aeolus typically require non-trivial modifications, while Breeze requires porting to a new language. Conversely, about 4,000 modules in Hackage (27%), a popular Haskell source distribution site, are currently safe for Hails applications to import. Of course, the functions that perform arbitrary I/O are not directly useful, and, like in JiF, must be modified to run in LIO. Nevertheless, many core libraries require no modifications. Moreover, we expect the number of safe modules to grow significantly with the next GHC release, which refactors core libraries to remove unsafe functions from general-purpose modules.

3.2 OS-level confinement

Hails uses Linux isolation mechanisms to confine processes spawned by VCs. These techniques are not novel, but it is important that they work properly. Using *clone* with the various `CLONE_NEW*` flags, we give each confined process its own mount table and process ID namespace, as well as a new network stack with a new loopback device and no external interfaces. Using a read-only bind-mount and the tmpfs file system, we create a system image in which the only writable directory is an empty `/tmp`. Using cgroups, we restrict the ability to create and use devices and consume resources. With `pivot_root` and `umount`, we hide filesystems outside of the read-only system image. The previous actions all occur in a setuid root wrapper utility, which finally calls `setuid` and drops capabilities before executing the confined process.

3.3 Browser-level confinement

VC responses are protected from inappropriate leaks on the client side using a sandbox. The sandbox, implemented as a browser extension for chrome, intercepts all network communication. In turn, all requests triggered by the page are allowed only if they are guaranteed to not leak information.

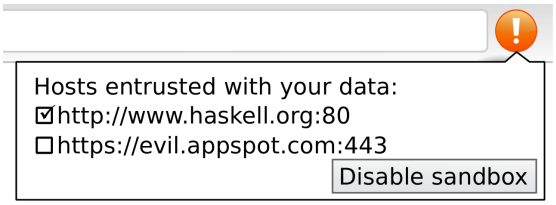

Figure 4: Hails client sandbox configuration. Users may (dis)allow communication to explicit hosts when the page label does not permit the flow directly.

The Hails client-side sandbox arbitrates traffic according to the label of the page, which is analogous to a server-side thread label. The Hails HTTP server sends the header `X-Hails-Label` with every VC response containing the initial page label, i.e., the label of response. As previously mentioned, if the page label is public, the sandbox does not impose any restrictions on the external requests triggered by the page. If the page label is not public, the sandbox only allows a request to a remote host if the page label is compatible with the principal implied by the remote host name. For instance, an image will be fetched from `maps.google.com` and a link will be followed to `hackage.haskell.org` if the page label is $\langle$`alice`$\vee$ `http://maps.google.com:80/` $\vee$ `http://hackage.haskell.org:80/`, TRUE$\rangle$. However, an XMLHttpRequest to `evil.appspot.com` will not be allowed. Similarly, if the page was instead labeled $\langle$`alice`, TRUE$\rangle$ the sandbox would reject all requests.

Users may approve otherwise disallowed network communication at the risk of potentially leaking their sensitive data to designated remote hosts. The first time a request to a disallowed host is intercepted, our extension requires the user to intervene. Specifically, the user is alerted and asked to approve network communication to the host in question. Clicking "No" blocks network access to the host for that `iframe` or tab. (The user can still view the contents of the page, except for resources, such as images or style-sheets, from the blocked host.) Conversely, clicking "Yes" allows the page to load normally; however, as illustrated in Figure 4, an icon is used to warn the user of a potential leak. In both cases, the user decision is saved for future requests and may easily be changed, as also highlighted in Figure 4.

The client-side sandbox is the least satisfying aspect of Hails's security, in part because it requires each user to install a new extension. In Section 7 we discuss the limitations of our current extension and future research

directions that could help address leaking sensitive data through the browser. Here, we finally remark that standards proposals such as Mozilla's CSP [42] show that browser vendors are open to incorporating mechanisms that coordinate with web servers to enforce security policies. Addressing data leaks on the server-side first, with systems like Hails, will help compel changes in tomorrow's browsers.

4 Applications

We built and deployed `GitStar.com`, a Hails platform centered around source code hosting and project management. We and others have authored a number Hails applications for the GitStar platform. Below we detail some of these applications including the core interface, a code viewer, follower application, wiki and messaging system.

GitStar At its core, GitStar includes a basic MP and VC. The MP manages users' SSH public-keys, project membership and project meta-data such as name and description; the VC provides a simple user interface for managing such projects and users.

Since Hails does not have built-in support for `git` or SSH, the GitStar platform includes an SSH server (and `git`'s transport utilities) as an external service. Our modified SSH server queries the GitStar VC when authenticating users and determining access control for repositories. Conversely, the GitStar MP communicates with an HTTP service atop this external `git`-repository server to access `git` objects.

GitStar allows users to create projects to which they can push files via `git`. Projects may be public (anyone can view or checkout repository contents) or private, in which case only specific users identified as *readers* or *collaborators* may access the project. In both cases, only collaborators may *push* contents to the project repository. GitStar provides an interface for managing these settings.

The rest of `GitStar.com` is provided by separately-administered, mutually-distrustful Hails applications, some of which were written by third-party developers. Each application is independently accessible through a unique subdomain of `GitStar.com`. When a user "installs" an application in a project, GitStar creates a link on the project page that embeds an `iframe` pointing to the application. This gives third-party applications a first-class role in extending the user experience.

Code Viewer One of the most useful features of source-code hosting sites is the ability to browse a project's code. We have implemented a code-viewing VC that allows users to navigate to different branches in a project's repository, view syntax-highlighted code, etc. Source code markup is done on the client-side using Google's Prettify JavaScript library [14]. Additionally, if the source file is written in C or Haskell, the VC provides the user with an option to see the output of static-analysis tools splint [19] and hlint [29], respectively.

Like all third-party applications, the Code Viewer is untrusted and accesses repository contents through the GitStar MP. When accessing objects in a private repository, the GitStar MP changes the VC's current label to restrict communication to authorized readers of the repository. Note that this may also restrict the VC from subsequently writing to the database.

`git`-based Wiki The `git`-based Wiki displays Markdown files from the "wiki" branch of a project repository as formatted HTML. It uses the pandoc library [25] to convert Markdown to HTML. Like the Code Viewer, the wiki VC accesses source files through the GitStar MP, meaning it cannot show private wiki pages to the wrong users. This application leverages functionality originally intended for the Code Viewer for different purposes, demonstrating the power of separating policies from application logic.

Standalone Wiki The standalone wiki is similar to the `git`-based Wiki, except that pages are stored directly in a database rather than in files checked into `git`. To accomplish this, the developer wrote both an MP and a VC. The MP stores a mapping between project names and wiki pages. Wiki pages are labeled dynamically to allow project readers and collaborators to read and write wiki pages. This is different from the `git`-based Wiki in that it allows a more relaxed policy: readers can create and modify wiki pages. Moreover, it is a concrete example of one MP that depends on another (namely the GitStar MP).

Follower GitHub introduced the notion of "social coding," which combines features from social networks with project collaboration. This requires that a user be able to "follow" other users and projects. GitStar does not provide this feature natively, but a Follower MP has been developed to manage such relationships. Users may now add the "Bookmark" application (implemented as a VC) to their project pages, which allows other users to add the project to their list of followed repositories.

Messenger The Messenger application provides a simple private-messaging system for users. Its MP, as implemented by the developer, defines a message model and policies on the messaging data. The policy allows any user to create a message, but restricts the reading of a message to the sender and intended recipient. Interfacing with the Messenger MP, the Messenger VC provides a page where users may compose messages, and a separate page

where they may read incoming messages.

5 Design Patterns

In this section, we detail the applicability of some existing security patterns within Hails, and various design patterns that we have identified in the process of building GitStar.

Privilege separation Since MPs are trusted by users to protect the confidentiality and integrity of their data, a well-designed MP should be coded defensively. Moreover, an MP should treat all invoking VCs as untrusted, including ones written by the same author.

The easiest way to program defensively is to minimize use of an MP's privileges, i.e., practice separation of privilege [35]. When doing so, invoking VCs will only be able to fetch data that the end user can observe, as opposed to all data when using the MP's privileges. Similarly, this restricts VCs to inserting already-labeled documents, as discussed in Section 2.3.2. This is important as it effectively limits a VC to inserting user-endorsed data, as opposed to almost-arbitrary data when using the MP privilege.

Trustworthy user input VC-constructed documents cannot necessarily be trusted to represent user intentions; thus, MPs should not allow VCs to arbitrarily insert data on behalf of the user. Consider, for example, the policy of the Follower MP imposed on user documents, as given in Figure 3. Here, a VC, even one running on behalf of `alice`, should not be allowed to construct and insert the document of Figure 2, without `alice` or the MP endorsing its contents.

Since VCs do not own user privilege and, as discussed above, MPs should not grant their privileges, Hails provides a mechanism for transforming user input data to a labeled document, that retains integrity. Recall that a VC's main controller is invoked, by the Hails server, with a pre-labeled HTTP request; the label on this request has the integrity of the user (e.g., $\langle \texttt{TRUE}, \texttt{alice} \rangle$). If the VC directly manipulates the request to construct an appropriate document, the integrity will be stripped. Hence, Hails provides a library for transforming a labeled, URL-encoded body (e.g., submitted from an HTML form in the user's browser) into a labeled document, that MPs may expose to VCs. This transformer takes a user-endorsed request and returns an MP-endorsed document that the VC may, in turn, insert into the database.

Users must still trust VCs to construct HTML forms that will reflect their intentions. However, an MP may inspect requests before transforming them to labeled documents. Moreover, policies, such as that of Figure 3, would prevent a VC trusted only by `bob` from modifying `alice`'s data.

Partial update The trustworthy user input pattern is suitable for inserting and updating documents in whole; it is not, however, directly applicable to partially updating documents. Returning to the Follower user model of Figure 2, a VC that wishes to present a form for updating the user's email address would have to include all the remaining fields as hidden input variables. Though this would allow the VC to update the email field by effectively inserting a new labeled document, this approach is error prone and not scalable.

Instead, we found that a partial document that contains the newly-updated fields, the document keys, and a token `$hailsDbOp` indicating the operation (`partialUpdate`, in this case) is sufficient for the MP to update an existing document. This partial-document must be endorsed by the user or MP, by, for example, applying the previous pattern. Directly, to carry out the partial update, the MP first verifies that the user is aware of the update by checking the presence of the operation token `$hailsDbOp`. Next, the MP uses the keys to fetch the existing document and merges the newly-updated fields into the document. Finally, the document update is performed, imposing restrictions similar to those of Section 2.3.2.

Delete We have found that most applications use a pattern similar to the partial update pattern when deleting documents: a VC invokes an MP with a document containing the target-document's keys and an operation token indicating a delete, i.e., `$hailsDbOp` set to `delete`. As in the partial update, this document must be endorsed by the user or MP by applying the trustworthy input pattern. Directly, the VC may invoke the MP with the labeled document, who, in turn, removes the target document after inspection.

Privilege delegation Hails provides a call-gate mechanism, inspired by [46], with which code can authenticate itself to a called function, i.e., prove possession of privileges, without actually granting any privileges to the called function. One use of call gates is to delegate privileges. For instance, an MP can provide a gate that simply returns its own privilege, on the condition that it was called by a particular VC.

While earlier version of GitStar utilized privilege delegation, we now largely avoid it; in many cases, we found modifying the policy to be a better alternative. For instance, the early version of the GitStar VC used the GitStar MP's privilege to look up project readers and collaborators for the SSH server. Now, we simply created a user account for the SSH server and added this principal as a reader in the `project` collection policy. Nevertheless, such refactoring may not always be possible and privilege delegation may prove necessary.

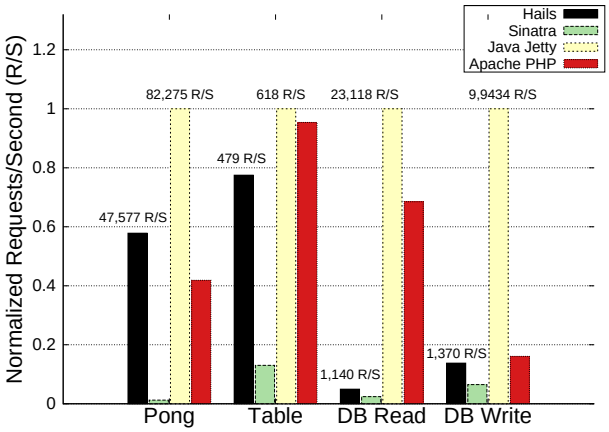

Figure 5: Micro-benchmarks of basic web application operations. The measurements are normalized to the Java Jetty throughput. All database operations are on MongoDB.

6 Evaluation

We compare the performance of the Hails framework against existing web frameworks, and report on the experience of application authors not involved in the design and implementation of the framework.

6.1 Performance Benchmarks

To demonstrate how Hails performs in comparison to other widely-used frameworks, we present the results of four micro-benchmarks that reflect basic operations common to web applications. Figure 5 shows the performance of Hails, compared with:

- Ruby Sinatra framework [36] on the Unicorn web server. Sinatra is a common application framework for small Ruby applications and APIs (e.g., the GitHub API is written using Sinatra).
- PHP on the Apache web server with `mod_php`. Apache+PHP is one of the most widely deployed technology for web applications, including WordPress blogs, Wikipedia, and earlier versions of Facebook.
- Java on the Jetty web server [10]. Jetty is a container for Oracle's Java Servlet specification, and is widely used in production Java web-applications including Twitter's streaming API, Zimbra and Google AppEngine.

We use `httperf` [31] to measure the throughput of each server setup when 100 client connections continuously make requests in a closed-loop—we report the average responses/second. The client and server were executed on separate machines, each with two Intel Xeon E5620 (2.4GHz) processors, and 48GB of RAM, connected over a Gigabit local network.

In the Pong benchmark the server simply responds with the text "PONG". This effectively measures the throughput of the web server itself and overhead of the framework. Hails responds to 1.7× fewer requests/second than Jetty. However, the measured throughput of 47,577 requests/second is roughly 28% and 47× higher than Apache+PHP and Sinatra, respectively.

In the Table benchmark, the server dynamically renders an HTML table containing 5,000 entries, effectively measuring the performance of the underlying language. Hails respectively responds to 30% and 23% fewer requests/second than Jetty and Apache+PHP, but 6× more than Sinatra. Hails is clearly less performant than Jetty and Apache+PHP for such workloads, even though Haskell should be faster than PHP at CPU workloads. We believe that this is primarily because Hails does not allow pipelined HTTP responses, so a large response body must be generated in memory and sent in its entirety at once (as opposed to sent in chunks as output is available). Nonetheless, Hails responds to 6× more requests/second than Sinatra.

The DB Read and DB Write benchmarks compare the performance of the read and write database throughput. Specifically, for the DB Read benchmark the server responds with a document stored in the MongoDB, while for the DB Write the server inserts (with MongoDB's `fsync` and `safe` settings on) a new document into a database collection and reports success. Like the Ruby library, the Haskell MongoDB library does not implement a connection pool, so we loose significant parallelism in the DB Read workload when compared to Jetty and Apache+PHP. In the DB Write workload, this effect is obviated since the `fsync` option serializes all writes.

6.2 Experience Report

We gathered experience reports from four developers that used Hails to build applications. Their reflections validate some of the design choices we made in Hails, as well as highlight some ways in which we could make Hails applications easier to write.

We conjectured that separating code into MPs and VCs leads to building applications for which it is easier to reason about security. This was validated by the application authors who remarked that although "experienced developers [need to] write the tough [MP] code and present a good interface," when compared to frameworks such as Rails, not having to "sprinkle [security] checks in the controller" made it easier to be sure that "a check was not missing." With Hails, they, instead, "spent time focusing on developing the [VC] functionality." We further found that developers (ourselves included) had a number of mass-assignment bugs in VC code [34]. Different from [20, 34], these bugs did not prove to be vulnerabilities in GitStar—the policies specified by the GitStar MP

trivially prevents attacks wherein one user tries to impersonate another. Though such vulnerabilities can be addressed differently, we found that similar bugs are easy to introduce and a well-specified policy can prevent them from becoming privacy concerns.

Implementing MPs using an earlier version of Hails proved challenging for most of the developers. In particular, while devising policies for a model was generally straightforward, developers felt that the API for actually implementing policy modules was difficult to learn. In fact, inspecting the code of one of the blog applications, we found that the developer had a bug that leaked blog posts regardless of whether the user decided to publish the blog post or not. This bug was a result of inadvertently making posts, as opposed to just post IDs, keys. We believe that this was due in part due to the terse policy-specification API.

Addressing the challenges with specifying policies, we designed the DSL presented in Section 2.3. We found this DSL to make policy specification much simpler. Equally important, developers have found it easier to understand what an MP enforces and thus make a more informed decision when deciding to use the library.

We are actively working on improving the Hails development experience. Compared to other frameworks, Hails needs more "good documentation with recipes." Developers found that the lack of "scaffolding tools for generating boiler-plate code [and] a template framework" impedes the development process. Part of our ongoing work includes building scaffolding tools for both VCs and MPs, adopting a templating language, and creating additional tutorials that illustrate typical application development.

7 Discussion and Limitations

In this section, we discuss the ramifications of the design and implementation of Hails and suggest solutions to some of its limitations.

Browser-level confinement As previously noted, we cannot expect all users to install the Hails browser extension which provides confinement in the browser. A different approach would be to re-write VC output at the server-side before sending it to the client, neutralizing data-exfiltration risks. Until recently, such content-rewriting was a dangerous proposition. In particular, Google [28], Yahoo [11], Facebook [13], and Microsoft [16] have all developed technology to constrain the effects of third-party web content such as advertisements; but the design of existing browser interfaces made those tools vulnerable to attack [26].

However, ECMAScript 5 Strict mode, now supported by most browsers, makes the prospect of safe re-writing far more tractable. For instance, SES [43], one promising approach with solid theoretical foundations, can now be implemented in about 200 lines of JavaScript. Though SES is not compatible with popular JavaScript libraries such as jQuery, this may well change. In our preliminary experimentation with Caja [28], a system which influenced SES, we successfully sandboxed VC responses in a similar fashion to our browser extension. Hence, if we cannot get traction from the browser vendors with our custom HTTP header, in the future we will experiment with a server-side filter that parses and regenerates HTML (so as to sanitize URLs in `src` and `href` attributes), and enforces JavaScript confinement with SES.

Query interface Hails queries are limited to expressions on keys. By separating keys from elements, the decision to permit a query is simple: if a Hails component can read from the database collection, it may perform a key-based query. This limited interface is sufficient for many VCs, which may perform further refinement of query results by inspecting labeled fields in their own execution contexts.

For larger datasets, better performance would result from filtering on all relevant fields in the underlying database system itself. Additionally, this would obviate the need to reason about the security semantics of keys. However, providing this more-general interface to a Hails application would require sensitivity to label policies inside the query engine. Since Hails builds atop MongoDB, which provides a JavaScript interface, we hope to compile policies to code that can implement the necessary label-checking logic.

8 Related Work

Information flow control and web applications A series of work based on Jif addresses security in web applications. SIF (Servlet Information Flow) is a framework that essentially allows programmers to write their web applications as Servlets in Jif [9]. Swift [8], based on Jif/split [45, 47], compiles Jif-like code for web applications into JavaScript code running on the client-side and Java code running on the server by applying a clever partitioning algorithm. SIF and Swift do not support information flow control with databases or untrusted executables; on the other hand, Hails provides weak security guarantees on the client side.

Ur/Web [6] is a domain specific language for web application development that includes a static information flow analysis called UrFlow. Policies are expressed in the form of SQL queries and while statically enforced, can depend

on dynamic data from the database. Security can also be enforced on the client side in a similar manner to Swift, with Ur/Web compiling to both the server and client. A crucial difference from Hails is that Ur/Web does not aim to support a platform architecture consisting of mutually distrustful applications as Hails does. Moreover, Hails is more amendable to extensions such as executing untrusted binaries or scaling to a distributed setting.

Logical attestation [37] allows specifying a security policy in first-order logic and the system ensures that the policy is obeyed by all server-side components. This system was implemented as a new OS, called Nexus. Hails's DC labels are similar to Nexus' logical attestation, but based on a simpler logic, namely propositional logic. A crucial difference between the Nexus OS [37] and Hails is that we provide very fine grained labeling and a framework for separating data-manipulating code from other application logic at the language level. For a web framework, fine grained policies are desirable; the language-level approach also addresses the limitations of cobufs used in Nexus [37]. Moreover, requiring users to install a new OS as opposed to a library is not always feasible. Nevertheless, their work is very much complimentary: GitStar can potentially use Nexus to execute untrusted executables in an environment that is less restricting that our Linux jail (e.g., it could have network access as directed by Nexus).

The closest related work to Hails is W5 [18]. Similar to Hails, they propose a separation of user data and policies (MPs), from from the application logic (VCs). Moreover, they propose an architecture that, like Hails, uses IFC to address issues with current website architectures. W5's design is structured around OS-level IFC systems. This approach is less flexible in being coarser grained, but, like Nexus, complimentary. A distinguishing factor from W5 is our ability to report on the implementation and evaluation of production system.

Trust management Trust Management is an approach to distributed access control and authorization, popularized in [2]. Related work includes [1, 3, 12, 21, 22]. One central idea in trust management, which we follow in the present paper, is to separate policy from other components of the system. However, trust management makes access control decisions based on policy supplied by multiple parties; in contrast, our approach draws on information flow concepts, avoiding the need for access requests and grant/deny decisions.

Persistent storage Li and Zdancewic [23] enforce information flow control in PHP programs that interact with a relational database. They statically indicate the types of the input fields and the results of a predetermined number of database queries. In contrast, Hails allows arbitrary queries on keys and automatically infers the security levels of the returned results.

Extending Jif, Fabric [24] is an IFC language that is used to build distributed programs with support for data stores and transactions. Fabric safely stores objects, with exactly one security label, into a persistent storage consisting of a collection of objects. Different from Fabric, Hails store units (documents) can have different security labels for individual elements. Like Fabric, Hails can only fetch documents based on key fields.

BStore [4] separates application and data storage code in a similar fashion to Hails's separation of code into VCs and MPs. Their abstraction is at the file system granularity, enforcing policies by associating labels with files. Our main contribution provides a mechanism for associating labels with finer grained objects—namely Haskell values. We believe that BStore is complimentary since they address similar issues, but on the client side.

9 Conclusion

Ad-hoc mechanisms based on access control lists are an awkward fit for modern web frameworks that incorporate third-party software components but must protect user data from inappropriate modification or sharing. By applying confinement mechanisms at the language, OS, and browser levels, Hails allows mutually-untrusted applications to interact safely. Because the framework promotes data-flow policies to first-class status, authors may specify policy concisely in one place and be assured that the desired constraints on confidentiality and integrity are enforced across all components in the system, in a mandatory fashion, whatever their quality or provenance.

As a demonstration of the expressiveness of Hails, we built a production system, GitStar, whose central function of hosting source-control repositories with user-configurable sharing is enriched by various third-party applications for viewing documents and collaborating within and between development projects. Through our active use of this system and the experience of other developers who built VCs and MPs for it, we were able to confirm the ability of the framework to support a modular system of heterogeneously-trusted software components that nevertheless can enforce flexible data-protection policies demanded by real-world users.

Acknowledgments

We thank Amy Shen, Eric Stratmann, Ashwin Siripurapu, and Enzo Haussecker for sharing their Hails development experience with us. We thank Diego Ongaro, Mike Piatek,

Justine Sherry, Joe Zimmerman, our shepard Jon Howell and the anonymous reviewers for their helpful comments on earlier drafts of this paper. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, by multiple gifts from Google, and by the Swedish research agency VR and STINT. Deian Stefan is supported by the DoD through the NDSEG Fellowship Program.

References

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, Oct. 1993.

[2] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 164–173, 1996.

[3] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust-Management System Version 2. RFC 2704 (Informational), Sept. 1999. URL `http://www.ietf.org/rfc/rfc2704.txt`.

[4] R. Chandra, P. Gupta, and N. Zeldovich. Separating web applications from user data storage with BSTORE. In *Proceedings of the 2010 USENIX conference on Web application development*, pages 1–1, 2010.

[5] 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 *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

[6] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'10, 2010.

[7] K. Chodorow and M. Dirolf. *MongoDB: the definitive guide*. O'Reilly Media, Inc., 2010.

[8] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. pages 31–44, Oct. 2007.

[9] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proc. USENIX Security Symposium*, pages 1–16, Aug. 2007.

[10] M. B. Consulting. Jetty webserver, March 2012. `http://jetty.codehaus.org/jetty/`.

[11] D. Crockford. Making JavaScript safe for advertising. `http://adsafe.org/`.

[12] J. DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 105–113. IEEE Computer Society Press, May 2002.

[13] Facebook. Fbjs (Facebook JavaScript). `http://developers.facebook.com/docs/fbjs/`.

[14] Google. Google code prettify, September 2012. `http://code.google.com/p/google-code-prettify/`.

[15] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. Exceptionally available dynamic IFC. Submitted to POPL, July 2012.

[16] S. Isaacs. Microsoft web sandbox. `http://www.websandbox.org/`.

[17] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st Symposium on Operating Systems Principles*, October 2007.

[18] M. Krohn, A. Yip, M. Brodsky, R. Morris, and M. Walfish. A World Wide Web Without Walls. In *6th ACM Workshop on Hot Topics in Networking (Hotnets)*, Atlanta, GA, November 2007.

[19] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, August 2001.

[20] L. Latif. Github suffers a Ruby on Rails public key vulnerability, March 2012. `http://www.theinquirer.net/inquirer/news/2157093/github-suffers-ruby-rails-public-key-vulnerability`.

[21] N. Li and J. C. Mitchell. RT: A role-based trust-management framework. In *The Third DARPA Information Survivability Conference and Exposition (DISCEX III)*. IEEE Computer Society Press, Apr. 2003.

[22] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, Feb. 2003.

[23] P. Li and S. Zdancewic. Practical information-flow control in web-based information systems. In *Proceedings of the 18th IEEE workshop on Computer Security Foundations*. IEEE Computer Society, 2005.

[24] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, , and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[25] J. MacFarlane. Pandoc:a universal document converter. `http://johnmacfarlane.net/pandoc/`.

[26] S. Maffeis and A. Taly. Language-based isolation of untrusted javascript. In *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE*, pages 77–91, 2009.

[27] J. Mayer and J. Mitchell. Third-party web tracking: Policy and technology. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 413–427, 2012.

[28] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`, June 2008.

[29] N. Mitchell. *HLint Manual*. `http://community.haskell.org/~ndm/darcs/hlint/hlint.htm`.

[30] B. Montagu, B. Pierce, R. Pollack, and A. Surée. A theory of information-flow labels. *Draft, July*, 2012.

[31] D. Mosberger and T. Jin. httperf-a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[32] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM symposium on Operating systems principles*, pages 129–142, 1997.

[33] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Computer Systems*, 9(4):410–442, October 2000.

[34] T. Preston-Werner. Public key security vulnerability and mitigation, March 2012. `https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation`.

[35] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[36] Sinatra. Sinatra, September 2012. `http://www.sinatrarb.com/`.

[37] E. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 249–264, 2011.

[38] E. Steel and G. Fowler. Facebook in privacy breach. *The Wall Street Journal*, 18, October 2010.

[39] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *Proceedings of the NordSec 2011 Conference*, October 2011.

[40] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th Symposium on Haskell*, pages 95–106, September 2011.

[41] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *The 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2012.

[42] B. Sterne, M. Corporation, A. Barg, and G. Inc. Content security policy, May 2012. `https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`.

[43] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical javascript APIs. In *IEEE Symposium on Security and Privacy*, 2011.

[44] D. Terei, S. Marlow, S. P. Jones, , and D. Mazières. Safe Haskell. In *Proceedings of the 5th Symposium on Haskell*, September 2012.

[45] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. Oct. 2001.

[46] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

[47] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, Washington, DC, USA, 2003. IEEE Computer Society.