

Hails: Protecting Data Privacy in Untrusted Web Applications

Submission to special issue on Verified Information Flow Security

Daniel Giffin^a, Amit Levy^a, Deian Stefan^{a,*,**}, David Terei^a, David Mazières^a,
John Mitchell^a, and Alejandro Russo^b

^aStanford University, 353 Serra Mall, Stanford, CA 94305, USA

^bChalmers University of Technology, Rännvägen 6B, 41296 Gothenburg, Sweden

Abstract. Many modern web platforms are no longer written by a single entity, such as a company or individual, but consist of a trusted core that can be extended by untrusted third-party authors. Examples of this approach include Facebook, Yammer, and Salesforce. 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 framework, Hails, for building web platforms, that adds mandatory access control and a declarative policy language to the familiar MVC architecture. We demonstrate the flexibility of Hails by building several platforms, including GitStar, a code-hosting web site that enforces robust privacy policies on user data even while allowing untrusted apps to deliver extended features to users.

Keywords: Web security, confinement, information flow control, MAC, MPVC, functional programming, Haskell, LIO, COWL

1. Introduction

Extensible web platforms represent an increasingly common way of developing and deploying software. Such platforms provide extensibility by allowing third-party *apps* to integrate, in a restricted manner, with the rest of the platform to offer additional features to users. Facebook popularized this extension model for social networking. Others have emulated it: Yammer provides a similar social platform for enterprises (running behind the firewall), while Dropbox and BitBucket provide a similar extension model for their project management platforms. The functionality users experience on these sites is no longer the product of a single entity. Instead, it is a combination of a core trusted platform, and apps written 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, unfortunately, once access to this data has been granted, there is no holistic mechanism to constrain what the app may do with it on

*Corresponding author. E-mail: deian@cs.stanford.edu.

**Part of this work was done while the author was at GitStar, Inc.

today's platforms. This puts users' privacy at risk. For example, the Wall Street Journal reported that some of Facebook's most popular apps, including Zynga's FarmVille game, were transmitting users' account identifiers (sufficient for obtaining personal information) to dozens of advertisers and online tracking companies [74]. While Business Insider reported that nearly 7 million Dropbox credentials were leaked by third-party apps [38].

In a conventional platform model, a user sets a security policy on specific apps, or classes of apps, but these policies either grant or deny access to information, they do not constrain how it can be used. Apps are written to only function with all their access requests granted, giving them unfettered access to sensitive information. This forces users to choose between privacy or functionality. The platform cannot guarantee that the app will not mine private messages for credit card numbers and send this information to the app's developer. Furthermore, third-party apps run on servers outside of the control of the trusted platform, meaning all data the app accesses is exfiltrated. Unfortunately, even if users understand an app's behavior, they are poorly equipped to understand the consequences of 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 [53].

In order to protect 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, while some of these techniques are common, each platform ends up providing a different solution from others.

To address these problems, we have developed an alternate approach for building platforms that need to confine untrusted apps. We demonstrate the system by describing GitStar, a social code hosting web platform inspired by GitHub and BitBucket. GitStar takes a new approach to the app model: we host third-party apps in an environment designed to protect data, rather than allow developers to host them on an arbitrary server. In doing so, we can enforce security policies that restrict information flow into and out of apps. This, in turn, enables a model wherein users can safely use apps without being asked whether to disclose data.

We built GitStar using a new web framework we developed 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. Of course, this also fits the degenerate case where a single entity is building the web site in full—in this case, Hails allows the site developers to protect user data from third-party libraries and bugs in their own application code.

Hails is distinguished by two design principles. First, security policies should be specified declaratively alongside data schemas, rather than spread throughout the codebase as guards around each point of access. Second, security 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, views provide a presentation layer for the user, and controllers decide how to handle and respond to particular requests. The MVC paradigm does not give security and privacy a first-class role, making it easy for programmers to introduce vulnerabilities [64,37,33,22]. By contrast, MPVC explicitly associates every model with a security 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 and be enforced even once code has access to data. 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 app-platform model when combined with a notion of decentralized privileges such as that introduced by the decentralized label model [59].

Unlike the access control lists used by today's web platforms, 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 GitStar core 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. By extending the popular MVC pattern by binding security policy to the model, giving us MPVC, Hails offers a new design point that we believe addresses these concerns. To investigate this, we report on our experience and the experiences of other developers in using Hails to both build web platforms and third-party apps for GitStar. While our sample is yet small, our experience suggests MAC security does not impede application development within an MPVC framework.

This paper extends an earlier conference version [29] with a more expressive policy-specification language, additional model persistence layers, integration with the COWL browser confinement system [81], and several new applications. The remainder of this paper describes the design of Hails and several applications built on top of Hails, including the GitStar platform. We discuss design patterns used in building Hails web platforms, evaluate our system, provide a discussion, survey related work, and conclude.

2. Design

The Hails MPVC architecture differs from traditional MVC web 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. Unsurprisingly, these checks usually do not extend to third-party code, which constitute large parts of modern web apps. By contrast, MPVC gives security policies a first-class role. Developers specify policies in a domain-specific language (DSL) alongside the data model. The framework then enforces these policies system-wide, regardless of the correctness or intentions of untrusted code, primarily using language-level security.

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. A VC, on the other hand, interacts with the user, invoking different MPs to fetch and store data. Our language-level confinement ensures that a data-model's policy is respected throughout the system, across the different components. For example, if an MP specifies that "only the user's friends may see his/her email address," then a VC (or another MP) reading the 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).

Fig. 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. The code viewer presents syntax-highlighted source code and the results of static analysis tools such as splint [42]. Using the same MP, the wiki VC interprets

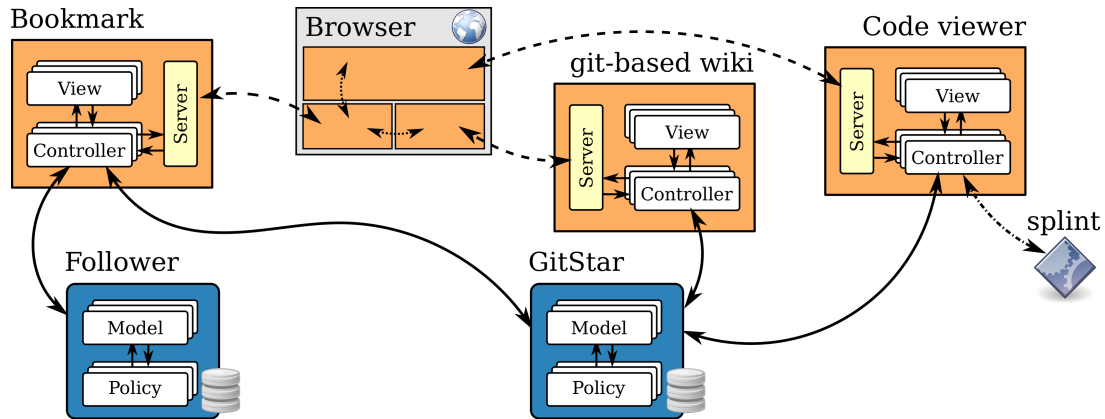


Fig. 1. Hails platform with three VCs and two MPs. Dashed lines denote HTTP communication; solid lines denote local function calls; dotted lines denote message passing between browsing contexts; dashed-dotted lines denote communication with OS processes. MPs VCs are confined at the programming language level; browser execution contexts (e.g., iframes) are confined by COWL; OS processes are jailed and only communicate with invoking VCs.

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, the Hails design directly leads to greater extensibility. For example, anyone who doesn’t like GitStar’s syntax highlighting is free to run a different code viewer. Indeed, any of the VCs depicted in Fig. 1 could be developed after the fact by someone other than the author of the MPs; because Hails’s MAC security continues to restrict what code can do with data even after gaining access to it, no special privileges are required to access an MP’s API.

2.1. Principals and privileges

Hails specifies policy in terms of *principals* who are allowed to read or write data. There are three types of principal. Users are principals, identified by usernames (e.g., *alice*). VCs and remote web sites that a component may communicate with are principals, identified by their origin URL [3] (e.g., <https://wiki.gitstar.org/> and <https://maps.google.com/>). MPs also have unique principals which, by convention, start with the prefix “_” (e.g., the principal of the GitStar MP of Fig. 1 is `_GitStar`).

An example policy an MP may want to enforce is “user *alice*’s mailing address can be read only by *alice* or by <https://maps.google.com/>.” 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. To be flexible, Hails allows read and write permissions to 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 standard authentication facilities (e.g., Mozilla’s Persona [58], and OpenID [65]); a browser presenting a valid session token represents the logged-in user’s principal. When code, server-side or in the browser, initiates outgoing HTTP requests to remote web sites, we consider the remote server to act on behalf of the principal of the web site, i.e., the origin URL.

Within the confines of Hails, code itself can act on behalf of principals. In particular, Hails provides unforgeable objects called *privileges* with which code can assert the authority of principals. The trusted Hails runtime passes appropriate privilege objects to MPs and VCs upon loading their code. For example, the GitStar MP is granted the `_GitStar` privilege. Thus, when a user wishes to use GitStar to manage 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. In the browser, our trusted COWL runtime grants code running in a page the privilege of the page’s origin; the VC serving the page can, however, specify a sub- or weakened-privilege.

2.2. Labels and MAC-based 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 [76], 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 $\langle \text{alice} \vee \text{bob}, \text{alice} \rangle$ specifies that `alice` or `bob` can read from the file and only `alice` can write to the file. Such a label is subjected on the code viewer VC of Fig. 1, for example, 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.

To ensure data protection, the Hails trusted runtime checks that remote principals satisfy any relevant labels before permitting communication. For instance, data labeled $\langle \text{alice} \vee \text{bob}, \text{alice} \rangle$ 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 $p \implies S$. Conversely, remote principal p can write data labeled $\langle S, I \rangle$ only when $p \implies I$. 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 (e.g., when a user session is not authenticated).

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 $\langle \text{alice}, \text{TRUE} \rangle$ if it could subsequently send the data anywhere over the network. However, Hails offers a different possibility: code without privileges can read data labeled $\langle \text{alice}, \text{TRUE} \rangle$ 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 $S_2 \implies S_1$ and $I_1 \implies I_2$ —in other words, any principals p allowed to read data labeled L_2 can also read data labeled L_1 (because $p \implies S_2 \implies S_1$) and any principals allowed to write data labeled L_1 can also write data labeled L_2 (because $p \implies I_1 \implies I_2$).

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 label. 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 or components 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”). The MAC enforcement system we developed for Hails, called LIO, is the first production system to address these threats at the language level [78,31].

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. This is an important design choice for both usability, since it ensures that a VC’s current label is never raised to a point where it cannot reply to the end user, and security, since it ensures that a malicious VC can only “leak” data to the remote user (who may be both the VC author and attacker) via a covert channels (e.g., “external timing channels” which encode information by carefully delaying HTTP responses [8,26]) insofar as it can read the data. Clearance ensures that a component cannot read data more sensitive than data it could otherwise send to the user directly.

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. Though MPs may contain arbitrary code and can expose an arbitrary API, we encourage using the dedicated database. In doing so, MP code only needs to specify what sort of data may be stored in the database and what access-control policies should be applied to it. This also ensures that other components can access the MP via a common interface—the Hails database API. Better still, it allows MP developers to leverage our DSL, described in Section 2.3.1, for specifying data policies in a concise manner.

The Hails database system provides a common document-oriented interface, similar to MongoDB [13], atop different persistence layers (e.g., MongoDB, file system, or even a REST API). Logically, a Hails *database* consists of a set of *collections*, each storing a set of *documents*. Each document, in turn, 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 can restrict access to the different database layers using labels. A 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 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

While static policies on databases and collections are often sufficient, in many web applications, dynamic fine-grained policies on documents and fields are desired. Consider the simplified models shown in Fig. 2. Each user profile model contains fields corresponding to a user’s username, email address, and full name, while each follower model is a mapping from one username to another—the username of the user they are following. In this scenario, the MP may configure usernames as keys in order to, for example, 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 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.”

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. This approach is based on the

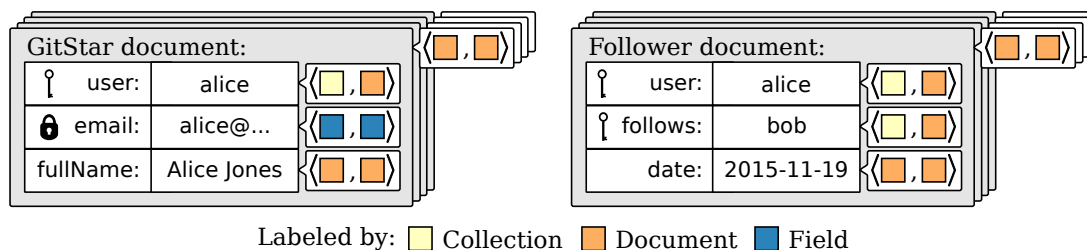


Fig. 2. GitStar MP user documents and Follower MP friend relationship documents. Each user document is indexed by a key (`user-name`) and contains the user’s `email` address and `fullName`. Documents and `email` fields are dynamically labeled using a data-dependent policy; the secrecy of the `user` key and is protected by the static collection label, the document label protects its integrity. The “unlabeled” `city` fields are protected by their corresponding document labels. Each follower relationship document contains two indexable keys—the `user` and `follows`, the name of the user they follow—and a field containing the `date` the user started following their friend.

observation that, in many web applications, the authoritative source for who should access data is a function of the data itself. For example, in Fig. 2, the username field values can be used to specify the user profile document and field policies mentioned above: `alice`’s document is labeled $\langle \text{TRUE}, \text{alice} \vee _ \text{GitStar} \rangle$, while the `email` field value is labeled $\langle \text{alice} \vee \text{bob} \vee \text{joe} \vee \dots \vee _ \text{GitStar}, \text{TRUE} \rangle$. 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—information available in the followers model—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. Fig. 3 shows the implementation of the GitStar users policy, as described above, using our DSL. Specifying 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`.

In contrast to the original DSL [29], which disallowed side effects when computing labels, our current DSL allows MPs to perform arbitrarily complex actions, such as database lookups, using the `run` keyword. Such actions can be used when labeling collections, documents, and fields. For example, in Fig. 3, the GitStar MP performs a database lookup of the user’s list of friends (via the Follower MP) when computing the label of their email address. Importantly, when an MP executes such actions, the Hails runtime ensures that they are confined according to MAC. We remark that, while this added flexibility may appear to make it harder to reason about policy, our experience suggests otherwise; in sections 6 and 7, we discuss this and the trade-off of this design choice.

2.3.2. Database access and policy application

MP policies are applied on every database operation. For example, when a thread attempts to insert a document into an MP collection, the Hails runtime first checks that that the thread can read from 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. By removing labels and ensuring that every database operation is mediated according to the policy, we ensure that policy code is localized to MPs.

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

```

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 public indexable key:
field "user" $ publicIndex
  -- Set document label, given document doc:
document $ λdoc -> do
  readers ==> anybody
  writers ==> ("user" 'from' doc) \/ _GitStar
  -- Set email field label, given document doc:
field "email" $ labeled $ λdoc -> do
  let user = "user" 'from' doc
      -- Fetch all the user's friends via the Follower MP:
      friends ← run $ withFollowerMP $
          map (get "follows") ◦ findAll $ select ["user" -: user ] "followers"
  readers ==> user \/ (fromList friends) \/ _GitStar
  writers ==> anybody

```

Fig. 3. DSL-specification for the GitStar 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 public indexable key. Finally, each document and `email` field is labeled according to a function from the document itself to a set of `readers` and `writers`. The latter invokes the Follower MP to fetch the user's list of friends (i.e., users they follow).

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 can also fetch stored data. When performing a fetch, application components specify a query predicate on indexed keys (or `TRUE`). As with insert, when fetching data, the runtime first checks that the thread can read from the database and collection. Next, the documents matching the predicate are retrieved from the database. Lastly, the field- and document-labeling policy functions are applied to each document and field; the resultant labeled documents are returned to the invoking component.

We remark that while the Hails database system supports other standard operations, including (partial) update and delete, the restrictions on most of these operations are similar to those of insert and fetch. We thus elide their details and refer the interested reader to the Hails documentation for details [84].

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 file system, etc. Hails relies on LIO's language-level confinement to prevent the VC and MPs it invokes from modifying or leaking data in violation of access permissions; we use OS-level resource management and isolation mechanisms to enforce platform-specific policies not otherwise enforced at the language level.

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 handle them. When a request is from an authenticated user, the server sets the `X-Hails-User` header to the username and attests to the request's contents for the benefit of VCs and MPs that care about request provenance and integrity. The request is also labeled according to the `Sec-COWL` header supplied by the browser (see Section 2.5.2). 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.

2.4.1. Enhancing VC functionality with MAC-aware libraries

To carry out their duties, components typically need access to various capabilities. While LIO provides many libraries (e.g., a file system and threads library) "out of the box," we extended LIO with several libraries. Below, we describe two of them: the Hails HTTP client and a library for safely executing external programs.

Since many VCs (and MPs) rely on communication with external services, usually over HTTP, we implemented an HTTP client on top of LIO. Before establishing a connection, and on each read and 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.

However, when communicating with other Hails servers (via HTTPS), the HTTP client can be more flexible by not imposing any read restrictions. Instead, it can return a fine-grained labeled response, which the VC can then inspect at its own will (e.g., when ready to raise its label). The label of the response is supplied by the remote server via a `Sec-COWL` header. Following the COWL spec [75], the HTTP client only considers responses with valid labels, i.e., labels that the principal of the remote server can satisfy. A more flexible, but also more complex, approach to communicating between different MAC-confined domains is to use the DStar protocol [92]; we leave this to future work.

In addition to communicating with external services, real-world applications also rely on external programs to implement different functionality. For example, as highlighted in Fig. 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 process is governed by a fixed label, namely the VC's current label at the time the external 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.

2.4.2. Safely executing code in the browser

When a user examines a private repository through an app such as the GitStar code viewer, Hails prevents the app VC from leaking private contents directly (e.g., using the HTTP client) and indirectly (e.g., via the database) within the confines of the server-side environment. However, VCs typically ship content to the browser where JavaScript or HTML may also attempt to leak data.

Hails prevents code from inappropriately leaking sensitive data on the client-side with COWL [81,75]. COWL is a language-level confinement system that adopts the MAC-based mechanisms of Section 2.2 to the browser. Most notably, COWL extends the browser with labeled browsing contexts, i.e., labeled pages and iframes. These labeled browsing contexts are analogous to server-side labeled threads: code running within a context is confined according to MAC, our trusted runtime restricts contexts from communicating with other contexts or web servers according labels. For instance, a GitStar user profile page labeled `<alice ∨ https://gitstar.org ∨ https://gravatar.com, TRUE>` is allowed to fetch `alice`'s image avatar from the Gravatar servers or use the XMLHttpRequest (XHR) constructor to fetch data from the main GitStar VC, but it cannot, for example, communicate with `evil.appspot.com`.

To ensure that code running in the browser is appropriately confined, the server-side Hails HTTP server supplies a `Sec-COWL` HTTP response header with every VC response. The response header value specifies the initial label (and privilege) of the browsing context, which is in turn set by the COWL runtime. By default, a page's label is set to the label of the VC thread that produced the response. But, code running in the browser can also raise the context label to subsequently read more sensitive information. For instance, in the example above, the initial profile page may have been public (e.g., because it did not yet fetch any sensitive data). But in fetching `alice`'s profile information via XHR, the page's label was raised to protect her sensitive data.

In addition to setting context labels, the `Sec-COWL` header is used to communicate labels on data between the server and client. For example, when responding to an XHR request for profile data, the GitStar VC may specify (explicitly or implicitly, via its current label) that the label of the response is `<alice ∨ https://gitstar.org ∨ https://gravatar.com, TRUE>`; of course, in most cases such labels are specified by the MP using our DSL.

Within the confines of the browser, VCs can execute arbitrary code and use most of the Web platform APIs [81,75], subject to confinement. COWL, however, provides additional APIs for labeling data client-side and thus imposing restrictions on how such data is used by other contexts (e.g., an iframe of a different VC, or a third-party service such as Google maps). This allows much of the VC implementation to be client-side and even consider scenarios not previously possible (see [81]).

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 handle.

The third-party developer may build a user interface to the newly-created model by writing a VC and registering a subdomain with the platform. 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, the platform administrator generates a privilege for the new VC, which corresponds to the hostname of the VC, and uses a program called `hails`, which contains the Hails runtime and HTTP server, to dynamically load the main VC controller and service user requests on the dedicated subdomain.

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 Fig. 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 sensitive headers such as `Cookie`; it also sets the `X-Hails-User` header to the user-name, if the request is from an authenticated user. To ensure that sensitive data from the browser is not leaked server-side, the request is labeled according to the `Sec-COWL` HTTP request header, which encodes the label of the browsing context that performed the request.

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. 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 thread, 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 VC 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 VC thread can flow to `<alice, TRUE>`; otherwise, the Hails server responds with an error message. Recall that clearance ensures that the VC label ends up being compatible with the user's browser label in most cases.¹ Indeed, clearance ensures that such failures are not delayed and occur early into the VC computation (e.g., when the VC attempts to read overly sensitive data).

¹

It is possible for an MP (or VC) to raise the clearance using its privilege as to allow the thread to read data more sensitive than what the end-user is allowed to see. However, this data should not be sent client-side and thus the reason for performing this check. MPs must explicitly declassify such data.

To prevent leaks client-side, the Hails HTTP server associates a `Sec-COWL` HTTP response header with every response. This header conveys the label of the response to browsers that support COWL [75]. When the response is data (e.g., JSON) the COWL-enabled browser ensures that the data cannot be leaked by code running in the browser. On the other hand, when the response is active content (e.g., an HTML page), the browser ensures that the content code is confined according to the label.

As detailed in Section 3.3, our client-side confinement system, COWL [81,75], restricts all incoming responses and outgoing requests according to the response label. For example, if the code viewer returns a response labeled `<alice∨ https://code.google.com, TRUE>`, the rendered page may retrieve scripts for prettifying code from `https://code.google.com`, but not retrieve images from `https://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 mechanisms, HTTP server, and libraries are part of the TCB. Parts of the system, namely our labels and confinement mechanisms (LIO and COWL), have been formalized in [77,76,78,56,31]. We remark that different from other work, our server-side language-level concurrent confinement system is sound even in the presence of termination and internal timing covert channels [78,31]. However, similar to other MAC systems (e.g., [48]), we assume that the remaining Hails components are correct and that the underlying OS, browser, 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 the confidentiality and integrity of their data. 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. To facilitate this choice, platforms should make 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 COWL, a malicious VC can coerce a user to declassify sensitive data (e.g., by tricking users to execute code in their browser console).

3. Implementation

Hails employs a combination of application-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 language-level information flow control (IFC) server-side and in the browser to enforce fine-grained policies. This section describes some of the implementation details of these language-level mechanisms and our OS sandbox.

3.1. Server-side 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 IO) from purely-functional computations. 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 extended the Glasgow Haskell Compiler (GHC) with `Safe Haskell` [85]. `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 implemented LIO [77,78], which employs the label-tracking and confinement mechanisms of Section 2.2. Despite sharing many abstractions with OS-level IFC systems, such as HiStar [91] and Flume [40], 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 [60], Aeolus [11], and Breeze [32] 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 libraries to a new language. Conversely, about 12,500 modules in Hackage (34%), a popular Haskell source distribution site, are currently safe for Hails applications to import. Of course, the functions that perform arbitrary IO are not directly useful, and, like in JiF, must be modified to run in LIO. Nevertheless, many core libraries require no modifications.

3.2. OS-level confinement

Hails uses Linux isolation mechanisms to confine processes spawned by application components. 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 file systems 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-side language-level confinement

As mentioned in Section 2.4.2, Hails prevents code from inappropriately leaking sensitive data on the client-side with COWL [81,75]. COWL is a language-level confinement system that adopts the confinement mechanisms of Section 2.2 to the browser, while retaining backward compatibility with the existing Web. We implemented COWL as modifications to the Chromium and Firefox browsers, largely reusing existing mechanisms which are already in place for the Same-origin Policy [3] and Content Security Policy (CSP) [82]. We refer the interested reader to [81] for a full description of COWL and its implementation. Here, we instead remark on the challenge of browser-side confinement deployment.

Different from server-side, where platform developers can ensure that apps are implemented using Hails, we cannot impose that users download and use our custom browsers to reap the benefits of COWL. However, COWL is undergoing standardization and on the roadmap to be incorporated in browsers, by default [75]. Nevertheless, it is important to support legacy browsers, when possible.

When communicating with older browsers—browsers that do not support COWL but do implement CSP—Hails can provide some confinement guarantees at the cost of flexibility. Specifically, Hails can confine the content of a page according to the response label by associating a CSP header that whitelists the origins that content can communicate with [82]. Unfortunately, this weakens our trust model since CSP whitelists do not encompass navigation (or in-browser message passing), so even with CSP, a malicious VC could leak sensitive user data by navigating to a URL that encodes the sensitive information. Moreover,

it also constrains application functionality: CSP is a discretionary access control mechanism [86] and thus not suitable for certain scenarios, such as mashups, where one needs to share sensitive data with a page whose label (and thus CSP header) is not compatible with the label of the data (see [81]).

4. Applications

In this section we describe several applications built with Hails, focusing primarily on the GitStar platform and its apps.

4.1. GitStar platform

We built and deployed GitStar, a Hails platform centered around source code hosting and project management. We and others have authored a number Hails apps for the GitStar platform. Below we detail some of these applications including the core management application, a code viewer, a follower/bookmarking application, a wiki, and a 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 the project 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 via HTTP when authenticating users and determining access control permissions 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 the platform functionality 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. 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 [30]. Additionally, if the source file is written in C or Haskell, the VC provides the user with an option to run static-analysis tools—respectively, `splint` [42] and `hint` [55]—on the checked-in code.

Like all third-party applications, the code viewer is considered untrusted and accesses repository contents through the GitStar MP. When accessing objects in a private repository, the GitStar MP raises 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 [51] 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—fetched via the GitStar MP—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.

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 that interacts with the follower MP) to their project pages, which allows other users to add the project to their list of followed repositories. This same application can be used to allow one user to follow other users.

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 MP, the messenger VC provides a page where users may compose messages, and a separate page where they may read incoming messages.

4.2. LearnByHacking platform

We built and deployed LearnByHacking, a Hails blogging platform in the style of School of Haskell [27]. LearnByHacking allows users to write articles (e.g., blog posts, tutorials, or lectures) that contain *active code* snippets, i.e., code that end users can modify and execute in the browser, without installing any software on their machine. Below we describe the LearnByHacking core and a commenting application built by another developer.

LearnByHacking At its core, LearnByHacking contains an MP and VC for managing user profiles, articles, and tags. The application allows users to collaborate on articles; until published, the MP ensures that articles can only be read and edited by its authors. Tags are public keywords that authors can associate with their articles, as to allow users to more easily subscribe to articles of interests.

The LearnByHacking VC provides authors with an interface for managing articles (e.g., collaborators, tags, published/draft state) and an interface for editing the article Markdown source, atop the CodeMirror in-browser editor [16]. The VC also renders articles, using the pandoc library [51], which users can read and interact with. Users can also subscribe to RSS/ATOM feeds (according to tags, users, etc.) to read articles offline.

To support active code, we extended Markdown code blocks with *directives*. Authors use these directive to specify the language of the code block, whether it is executable, a name for the block, and any other code blocks it depends on. The latter two directives allow authors to chain different code blocks to, for example, illustrate different execution sub-paths of a program to users (e.g., failure and success). The VC relies on our OS-level confinement mechanism to safely execute active code. We currently support code written in C, C++, JavaScript, Bash, and Haskell. However, extending LearnByHacking with additional languages is straight forward and, importantly, does not rely on any modifying any trusted code, i.e., the MP.

Commenter The commenter application provides a simple way for users of LearnByHacking to comment on published articles. To provide this feature, the developer defined an MP for storing comments. The MP policy ensures that the author of a comment is the only user allowed to modify their comment; comments are publicly readable. The commenter VC is embedded in published LearnByHacking articles as an iframe,

providing users with ways for writing comments (both in response to an article and as replies to other comments), editing their comments, and viewing all the comments associated with the article. We remark that while the developer implemented the application for LearnByHacking, the commenting system can easily be used by other applications as an alternative to Disqus commenting system [20]. Indeed a more tightly-integrated commenting system would have reused LearnByHacking’s MP policy to allow for comments to unpublished articles.

4.3. λ Chair

We built a simple conference management system called λ Chair, inspired by EasyChair. λ Chair is a Hails application that can be used by conference chairs to manage paper submissions and reviews. The application defines data models and policies for handling users, papers, and reviews. After a user creates an account, the chair can add them to the committee, assign them to review papers, and manage their conflicts of interests. Logged-in users can only upload, view, manage conflicts, and edit their uploaded papers. Modifications are only allowed during the submission period. During the review period, committee members can participate in the review process by reading and writing comments and reviews for papers they are not in conflict with. Finally, after the review period, authors can additionally see their papers’ reviews.

While λ Chair’s functionality is relatively straight forward, the application is particularly interesting because of its security policy. Hails must ensure that only the *right* users are allowed to read and modify a particular paper or review, and this changes according to assignments, conflicts of interests, and conference state (in submission, review, or done). Indeed, we introduced side-effects into our policy DSL because of this use case; as further detailed in Section 7, the MP of our first λ Chair implementation was overly complex because the policy language was not flexible enough.

4.4. Taskr

Taskr is a task and project management application. Taskr was built by a group of three undergraduate students over a summer research internship. The students learned Haskell and Hails at the beginning of the summer, and built Taskr in the last month of their internship.

The application defines data models and policies for users to collaborate on projects through task assignment and a commenting system. Each project has one or more project leaders, who can change project settings, as well as project members, who may contribute to the project. Any member may add a task or comment on a project, but only leaders can make administrative changes to a project, such as changing the membership list.

Projects may be public, in which case anyone can view them, including non-members. Alternatively, private projects are only visible to their members.

Hails enforces all of these policies in a single place: the MP. We found that, although the students building the application were novices in both Haskell and Hails, once the policy was well-defined, they were able to build the rest of the application rapidly. In Section 6.3 we discuss our improvements to Hails based on feedback from these students.

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 the applications described in Section 4.

Minimizing privilege usage Since MPs are trusted by users to protect the confidentiality and integrity of their data, a well-designed MP should be programmed defensively. For example, an MP should treat all

invoking VCs as untrusted, including ones written by the same author. This minimizes the damage that any VC—whether malicious or vulnerable—could have on user data.

The easiest way to program defensively is to minimize the use of MP privileges and avoid granting privilege, for example, to other components unnecessarily, i.e., follow the principles of least privilege and privilege separation [68]. When doing so, VCs that access the MP’s database will only be able to fetch data that the end user can observe. This is in contrast to having MP privileges, which provide unfettered access to its data. Without privileges, VCs are also restricted to inserting already-labeled documents (see Section 2.3.2) on behalf of users, i.e., without privileges VCs can typically only insert data endorsed by the Hails HTTP server on behalf of the user, as opposed to arbitrary data (when using the MP privilege).

Trustworthy user input Since VCs can craft arbitrary HTTP requests, VC-constructed documents should not always be trusted to represent the user’s intentions. Hence, MPs should ensure that VCs cannot arbitrarily insert or modify data on behalf of users. To this end, MPs should, as discussed above, minimize privilege usage and, moreover, they should set policies that disallow code from acting on behalf of users. An example of such a policy is the policy on user documents given in Fig. 3. The policy specifies that only `alice` (or the GitStar MP) is allowed insert and modify documents with the username set to `alice`. Since Hails does not grant user privileges to any code, a VC, even one handling a request from `alice`, is disallowed from constructing and inserting a document with the username set to `alice` (e.g., the document of Fig. 2), without `alice` or the MP first endorsing it.

We, however, need to allow VCs to insert, modify, and delete user data when requested by the user to do so. To this end, the Hails HTTP server endorses requests on behalf of users before invoking a VC’s main controller. This reflects the fact that requests may convey the user’s intent. Since VCs cannot directly manipulate requests (e.g., to transform them into arbitrary database actions) without stripping off their integrity labels, Hails also provides a library for transforming labeled requests into labeled documents. MPs may use this library to expose transformers to VCs. These transformers take, as input, user-endorsed requests and return MP-endorsed documents that VCs may, in turn, insert into the database (potentially in place of an existing document).

In practice, MPs typically inspect requests before transforming them into labeled documents. This avoids the need to completely trust VCs to construct HTTP requests that reflect the user’s intentions. Indeed an MP may choose to only transform requests from VCs it trusts, or from VCs the user has approved. Nevertheless, we recognize this as a limitation of our approach—since VCs ultimately interface with users they cannot be considered completely untrusted. We, however, remark that policies, such as that of Fig. 3 prevent a VC trusted only by `bob` from modifying `alice`’s data. Moreover, using this design pattern further proved to be useful in reducing and reasoning about the attack surface of Hails applications: a VC cannot perform arbitrary actions that may result in data corruption without going through an MP filter.

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. And, in many cases, it is desirable to update only certain elements of a document. For example, in GitStar users sometimes need to (only) update their SSH keys, while in λ Chair authors need to be able to update a paper’s title and abstract without uploading a new PDF. Updating a whole document is both error prone, since it requires developers to include all document fields with every HTML form, and inefficient, since it requires sending all the data from the server to the browser with every form.

Fortunately, the HTTP request PATCH method precisely addresses this issue [23]; in contrast to PUT, which is used by browsers to indicate that a document should be replaced with the supplied resource, PATCH is used to convey partial modifications to a stored document. Naturally, Hails VCs use HTTP

request methods to convey the user’s intent to MPs (GET for fetching a document, POST for creating a new document, PUT for updating a document, PATCH for partially updating a document, and DELETE for deleting a document). However, since many browsers still only support a subset of these methods, we still rely on request bodies to convey this information. In particular, when partially updating a document, we found that a partial document that contains the newly-updated fields, the document keys, and a token `$hailsDbOp` indicating the operation (PATCH, in this case) is sufficient for the MP to update an existing document. This partial document must be endorsed by the user or MP, by, for example, applying the previous pattern, before the VC attempts to update the corresponding persistent document. Whenever a VC invokes an MP to carry out a partial update, the MP first verifies that the user is aware of the update by checking the presence of the operation token `$hailsDbOp`. Next, the MP uses the keys to fetch the stored document. Finally, it merges the newly-updated fields into the original document and writes the document to the database, imposing restrictions similar to those of Section 2.3.2.

Delete We have found that most applications require a pattern similar to the partial update pattern when deleting documents. To delete a document, a VC invokes the appropriate 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. When invoking an MP with such a labeled document, the MP simply inspects it and removes the target document.

Privilege delegation Hails provides a call-gate mechanism, inspired by [91], 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.

An early version of GitStar relied on privileged delegation to allow the GitStar SSH server to read the SSH keys stored in the GitStar MP database. Specifically, the GitStar (project management) VC used a call gate to retrieve the GitStar MP privilege when looking up project readers and collaborators on behalf of the GitStar SSH server. We’ve since refactored this code to not use privilege delegation. Instead of relying on the MP privilege, we created a dedicated user account for the SSH server and added this principal as a reader to the GitStar project collection policy.

While we have managed to avoid privilege delegation in most applications we developed, changing policies as in GitStar is not always possible and privilege delegation may prove necessary. We especially foresee this being useful for non-platform applications, i.e., applications written by a single team, at the cost of placing more trust in VCs. Indeed, we’ve built several simple applications, similar to those of Section 4, but using the call gate mechanism to retrieve a privilege corresponding to the current user;² in most cases, the code turned out to be simpler than the corresponding code without privilege—but, of course, at the cost of trusting VC code to properly utilize the user’s privilege.

6. Evaluation

We evaluate Hails on three dimensions:

1. **Performance:** we compare the performance of the Hails framework against existing web frameworks.
2. **Security:** we give measurements of the TCB sizes of Hails applications we and other have built as rough estimates of the applications’ attack surfaces.

²

Since MPs do not have access to user’s privilege, the privilege corresponding to the current user is actually an MP subdivided privilege (e.g., `_GitStar ∨ alice`).

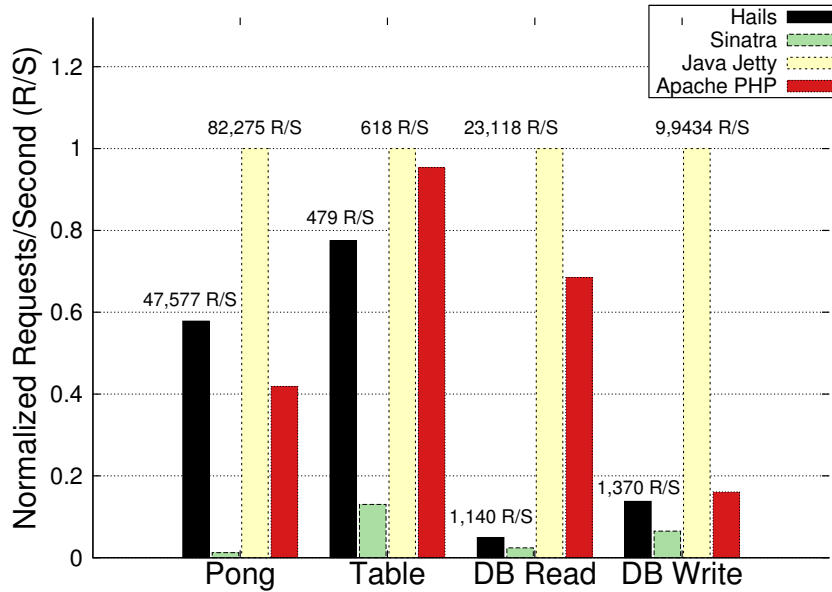


Fig. 4. Micro-benchmarks of basic web application operations. The measurements are normalized to the Java Jetty throughput. All database operations are on MongoDB.

3. **Usability:** we report on our experience and the experience of application authors not involved in the design and implementation of the framework in building Hails applications.

Below we describe our methodology and evaluation results. We refer the interested reader to [81] for a detailed performance evaluation of our client-side confinement system, COWL.

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. Fig. 4 shows the performance of Hails, compared with:

- Ruby Sinatra framework [72] 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 [17]. 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` [57] 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

Application	Trusted MP code	Untrusted app code	Third-party library code
GitStar manager	251	1,590	54,109
GitStar code viewer	0	1,454	66,786
GitStar git-wiki	0	859	66,258
LearnByHacking	224	1,352	96,727
Commenter	34	212	46,685
Taskr	90	894	37,879
λ Chair	132	613	33,769

Table 1

Application line count, broken down into the amount of code that is essentially trusted (the MP code), the application code that is not trusted (i.e., essentially the VC code), and third-party libraries. We only give the line count for the subset of applications that have been deployed. We count every line in a Haskell source file, whether it is ultimately execute in the application or not.

requests/second than Jetty. However, the measured throughput of 47,577 requests/second is roughly 28% and 47 \times 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 \times 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 \times 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 lose 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. Evaluating the attack surface of Hails applications

Our confinement mechanisms have underlying theoretical foundations with accompanying proofs of strong security properties. For example, we proved that programs written in LIO and COWL satisfy non-interference [77,79,31], i.e., they cannot leak or corrupt user-sensitive data. In contrast to most dynamic language-level confinement systems, our formal models consider real-world language features (e.g., exceptions and threads) and our theorems hold even in the presence of typical covert channels, such as the termination, internal timing [78,31], and even hardware cache-timing channels [80].

Unfortunately, these results do not trivially extend to Hails applications. This is primarily because MPs (and VCs) rely on privileges to accomplish their tasks and our formal language models, like most results in this area, do not account for privileges. We consider such extensions to our formal models as part our future work. Here, we evaluate the attack surface of typical Hails applications more qualitatively, under the trust assumptions of Section 2.6.

In particular, we report the TCB line count of some of the Hails applications described in Section 4. Unlike traditional web frameworks, where the TCB of an application is essentially the whole application

codebase, including the third-party libraries it depends on, the TCB of a Hails application is limited to the MP policy code and any code that uses MP privileges. Since a bug in any of this code could potentially be a vulnerability, we consider this to be the attack surface the application. This is precisely the reason we encourage developers to use the privilege separation patterns of Section 5.

Table 1 gives the line counts (of Haskell code) for our applications; these numbers include the application TCB size, i.e., the trusted MP code, and the untrusted application code, which entails the VC code and third-party libraries. We remark that in general the amount of code that developers must get right is relatively low—on the order of a few hundred lines of Haskell—especially when considering the rest of the application code, which orders on tens of thousands of lines of Haskell. More interestingly, we remark on the evolution of the GitStar and LearnByHacking platforms. In both cases the applications grew with new features and functionalities, respectively adding over 100,000 and 40,000 lines of code, but the attack surface grew sub-linearly and remained under 300 lines of code.

6.3. Experience report on building Hails applications

When building systems such as Hails it is important to also consider its usability. Indeed, it is important to consider both, usability and security, jointly when designing systems, since addressing traditional usability concerns alone can negatively impact security and vice versa [36]. We iterated on the Hails design and the underlying confinement mechanisms to address usability issues that arose when we and other developer were building applications. In the process, we gathered experience reports from seven developers, four of which we interviewed as part of a small, mostly informal, usability analysis. We remark that none of the developers had experience building web applications in Haskell. Their reflections validate some of the design choices we made in Hails, as well as highlight some ways in which we could make Hails more usable. Below we summarize these reports and our own experience building platforms and applications. We refer the interested reader to [79] for details on the LIO design evolution, which was largely influence by our work on Hails.

Understanding the security model We conjectured that MAC and the separation of code into MPs and VCs leads to building applications for which it is easier to *understand* and reason about security. To measure understanding we compared the mental model of the developers with our expert model, in the style of [36]. Beyond validating their understanding through discussion, their understanding of policies was also reflected in the code they wrote. For example, their VC code often gracefully handled failures due to policy.

More broadly, we found that developers also understood the implications of the MPVC paradigm on security, i.e., that MP code is security critical and that VCs need not be trusted to enforce policy. 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.” Indeed, we found this to also be reflected in the code they produced—most developers, ourselves included, did not include checks in VCs that are typical to web apps (e.g., *can these actions be performed by the current user?*). In fact, most applications written in Hails would traditionally be susceptible to the mass-assignment vulnerability that affected GitHub [64,43], or the access control check vulnerabilities that affected Facebook [37] and United [22]. But, because our developers specified MP policies that would disallow one user from impersonating another, and because policies are enforced in a mandatory fashion, such “bugs” have no security implications in Hails—exploiting such bugs typically revealed UI bugs, which arguably, should not even be addressed.

Policy specification usability Implementing MPs using an early version of Hails proved to be challenging for most of the developers. In this version, we did not have the declarative DSL of Section 2.3.1; instead, the “policy specification” entailed imperatively labeling the different database components (collections, documents, etc.). While developers were typically effective in devising policies for a model, the API for implementing policy modules was difficult to learn, inefficient, and error prone. Unsurprisingly, the policy code was hard to understand, also negatively impacting the previous usability factor—understanding the security model.

To address this, we designed the DSL described in the conference version of this paper [29], similar to a subset of the DSL given in Section 2.3.1. We found that this DSL, while less flexible than imperative code, makes policy specification simpler and more efficient—developers were able to specify correct policies in less time. Equally important, developers found it easier to understand what policy an MP was enforcing and thus make a more informed decision when deciding to use the library. Moreover, this also improved Hails’ satisfiability—developers were no longer dreading writing MP code.

Unfortunately, since [29], we found the DSL to still have some usability-security concerns. First, we found that new developers misunderstood the security implications of declaring fields to be keys. For example, one blog developer marked the body fields of articles as keys, despite specifying a relatively strict document (article) policy. This was motivated by their want to implement a feature that would allow full-text searching over articles. Unfortunately, this was not an isolated instance; we found that developers that do not (yet) understand the intricacies of our database model and are mostly motivated to implement features, are unlikely to be cautious when declaring a field to be indexable, especially if they’ve already specified a policy for a document. To address this, we modified our DSL to use the modifier `publicIndex` instead of `key` when declaring a field to be a key; this small change imposes fewer assumptions on the developer, i.e., that they know that declaring fields as keys has security implications, and makes it easier for them to be vigilant about not declaring fields “public.”³

Second, we found that restricting DSL policies to pure functions negatively affects efficiency and understandability. In many cases purely-functional policies are sufficient and easy to reason about. However, when a policy relies on data not present in the document supplied as an argument, developers would have to resort to using a level indirection to accomplish their goal. Consider an example from our λ Chair implementation. In λ Chair, we needed to specify that a paper can be read by any committee member not in conflict with the paper. But, since committee members and conflict of interest relationships are stored in different collections, we didn’t have access to the data and ended up encoding this information in the labels themselves. For example, `alice`’s submission, whose ID is 1, would initially be labeled `<alice ∨ #paper : 1 ∨ _Chair, alice ∨ _Chair>`. The λ Chair MP would then use a transformer to relabel papers by expanding group principals (e.g., `#paper : 1`, above) into label formulae (e.g., `john ∨ claire`, the PC members not in conflict with `alice`) before returning control to the invoking VC. This proved to be both inefficient and overly complicated—most developers had a hard time understanding the additional level of indirection.

To address this, we extended the DSL with the `run` keyword which allows an MP to execute arbitrary Hails code in the policy. When used to compute document or field labels, `run` restricts the computation clearance to to the MP collection label. (Similarly, `run` sets the clearance of its computation to the database label when computing collection labels as such.) This ensures that the labels are not computed from overly

³

Key are protected by the collection label. But, collection labels in many cases correspond to the public label.

sensitive data. We imposed these semantics to demotivate MP developers from using sensitive data in labels.⁴

We remark that our primary reason for originally making DSL policies pure functions was to make it easy for developers to specify and understand policies; the extension with `run` is very much still in line with this reasoning. Indeed `run` makes it easy to specify complex policies in a straight forward manner. Moreover, it extends the usage of the DSL to use cases where imperative code would otherwise be required: specifying dynamic policies on collections. This, for example, makes it possible for λ Chair to support multiple conference while ensuring isolation between the different conferences' content.

Usability of framework libraries Given the task of building VCs and MPs, we learned that developers found Hails to be an effective framework, i.e., they were able to build the applications at hand. But, in addition to addressing the above usability concerns, we refactored several APIs, which are primarily used by VCs, to address various usability concerns (e.g., efficiency and satisfaction) that arose during the development process. For example, to improve the efficiency of VC development, we extended our database APIs to provide developers with functionality that was necessary in common tasks—e.g., filtering results based on the VC's clearance and unlabeled documents before returning them. Similarly, we introduced support for template frameworks to address concerns raised by developers who found that our lack of “scaffolding tools for generating boiler-plate code [and] a template framework” impedes the development process, when compared to frameworks such as Rails. We are still working on improving the Hails development experience as a whole, currently focusing on developers' want for better documentation, recipes, debugging information, and scaffolding tools that will make it easier to build both VCs and MPs.

We remark that, while we believe that Hails is a usable framework, i.e., *it can be used to achieve specified goals with effectiveness, efficiency, and satisfaction* [35,71], our conclusion is drawn from external adoption of our systems and the short usability analysis we conducted.⁵ We believe that a more formal and extensive usability analysis (e.g., one that defines small tasks for users to accomplish) is an important future research goal that can lead to a more useably-secure framework. In retrospect, we should have consulted HCI experts throughout the design process.

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.

OS-level confinement Since [29], both Docker and CoreOS have released products that use Linux isolation mechanisms, namespacing, and cgroups to isolate Linux applications from each other. Their underlying approach is very similar to ours (see Section 3.2). Until recently, however, neither was an appropriate replacement for our OS-level confinement (e.g., Docker did not originally provide different user namespaces [93,61]). Indeed, because of their focus on providing app-deployment solutions, both Docker and CoreOS/rkt are more complicated than our solution (and the reason for some of their vulnerabilities [47]) and not well-suited to to be used as throw-away containers.⁶ Nevertheless, we believe that using one of these solutions in place of our current OS-level confinement mechanisms can have

⁴

They can still read sensitive data, but have to go through the additional step of raising the clearance.

⁵

Our MAC-based confinement mechanisms are seeing some adoption in both academia (e.g., LIO and COWL have been used in several university courses) and industry (e.g., COWL is a spec at the W3C while Hails and LIO are used in a commercial product at GitStar, Inc.). We are actively trying to incorporate feedback from the different use cases to improve the systems.

⁶

numerous benefits. Usability is the most obvious. But, we can also leverage their recent support for AppArmor and SELinux (see [28,21]) to make the confinement more flexible (e.g., to allow an OS-level confined process to communicate with some remote servers); our OS-level confinement is unnecessarily restricting in disallowing any network communication or shared file system access.

Browser-level confinement In Section 2.4.2 we discussed the current status of our browser confinement system, COWL, and an alternative approach to providing a limited form of confinement with CSP for older browsers. A different, but complimentary, approach to both would be to re-write VC output at the server-side before sending it to the client, neutralizing data-exfiltration risks. While such content-rewriting used to be considered a dangerous proposition (e.g., because tools implemented by Google [54], Yahoo [18], Facebook [24], and Microsoft [34] proved to all have vulnerabilities [52]), most browsers now have JavaScript engines that support ECMAScript 5 (ES5) strict mode, which makes the prospect of safe re-writing far more tractable. In addition, client-side tools with solid theoretical foundations, such as SES [83], which builds on ES5 strict mode, and DJS [4], have shown promise in confining and isolating increasingly complex applications which contain untrusted code. We leave the exploration of using such tools and output re-writing to future work.

Query interface Hails queries are predicates on keys. By separating keys from the other fields, 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 more complex queries on other, labeled fields by inspecting them in their own execution contexts.

For large 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 most of the persistence layers we consider (e.g., MongoDB and PostgreSQL) allow developers to plug in custom logic in their engines, we believe that compiling policy to code to run in the database layer is viable and a useful improvement on our implementation.

We, remark, however, that since our policy DSL allows developers to execute arbitrary code in `run` blocks, this may introduce some challenges. For example, if the MP code was fetching data from a remote server in the policy code, it may not be sensible for such code to execute in the query engine. Luckily, Haskell's monads makes it easy to define sub-languages within Haskell. This allows us to, for example, restrict (certain) `run` blocks to only be composed of pure code and database queries as to facilitate compilation to query engines.

Side-effecting policy code The ability to execute arbitrary code in policy code has several trade-offs, as as discussed in Section 6.3. For example, with the introduction of `run`, it is now possible for one developer to define a policy that depends on data from another MP, the policy of which in turn depends on the first developer's MP. Unlike non-terminating pure code, debugging such a policy may prove to be more difficult. Similarly, it is possible for developers to carelessly increase an application's latency by performing network requests (without short timeouts) and other IO within the policy code. While we have not run into such cases in practice, we foresee the need for static analysis to help eliminate such bugs in policy code. Indeed, even for purely-functional policy code, static analysis (e.g., in the form of refinement types [66]) can be used to eliminate certain kinds of bugs (e.g., VCs trying to read from a collection whose static label restricts read access to the MP).

Their typical use case is long-lived applications, i.e., servers. This is different from ours, wherein containers are typically short lived, usually for the duration of a request.

Policy as code When code wishes to access data stored in the database, the Hails database layer will always invoke the corresponding MP's policy code to compute labels. An alternative approach could have serialized labels alongside data (e.g., as in HiStar [91] and LIO's file system). This has the benefit of allowing code—and, in particular, code written in another language—to access labeled data without invoking MP code. Unfortunately, this also has many drawbacks that Hails' approach does not. For example, in taking this approach Hails policies can be specified in a single location in a high-level, generic fashion that applies to all the documents in the collection. This not only makes it easy and less error prone to specify policy, but also easier to inspect and understand an application's policy—reasoning about security policy in terms of the many serialized labels is very difficult. More importantly, changing a policy simply amounts to changing the MP policy specification code, updating the MP's version in the platform's global configuration file, and re-linking affected apps—it importantly does not require relabeling data stored in the database, a process that is both less efficient and safe. Nevertheless, allowing applications written in different languages to access labeled is important, especially in enterprise scenarios. Hence, the compilation of queries to an intermediate form that can be interpreted in different languages (e.g., by JavaScript in MongoDB) seems like an interesting balance.

Deployment We have described the deployment of mutually-untrusted software components on a particular web “platform,” GitStar, which simply provides the minimal policy modules and apps necessary to coordinate shared data and to unify the user interface. But there is nothing to prevent a constellation of such platforms being deployed together, providing a rich ecology of functionality in which third-party apps could thrive; for example, social-networking apps could operate on data managed by disparate platforms. Services such as App Engine and Heroku have already shown that many developers are willing to trust a centralized hosting provider to run their applications. Indeed, an interesting direction for future work may be to implement a Hails platform wherein MPs simply re-expose users' Facebook, Twitter, etc. data, which is available via their platform APIs, by attaching labels to it. Such a hosting platform would empower users by allowing them to use third-party apps that Hails ensures to protect their privacy.

DoS attacks The Hails web framework does not address address denial-of-service (DoS) attacks in the form of resource exhaustion or heavy network load, except when apps execute external programs. For example, we do not restrict an app from participating in a distributed DoS attack by sending many HTTP requests to a target host. Such concerns are platform-specific and outside the scope of our web framework. We, however, remark that Hails composes well with existing OS-level isolation, resource management and workload distribution mechanisms since each Hails app is run as a separate process.

8. Related Work

Information flow control and web applications A series of work based on Jif addresses security in web applications. Servlet Information Flow (SIF) [15] is a framework that allows programmers to write their web applications as servlets in Jif. Swift [14], based on Jif/split [90,94], 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 involving databases or untrusted executables as Hails does. However, we believe that their partitioning approach could potentially be used by Hails to confine JavaScript in legacy browsers.

Ur/Web [12] 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. Moreover, Hails is more amendable to extensions such as executing untrusted binaries or scaling to a distributed setting.

Logical attestation [73] 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 [73]. Hails’s DC labels are similar to Nexus’ logical attestation, but based on a simpler logic: propositional logic. A crucial difference between Nexus and Hails is that Hails provides 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 used by Hails also addresses the limitations of cobuFs used in Nexus, as discussed in [73]. Moreover, requiring users to install a new OS as opposed to a library is not always feasible. However, Nexus is very much complimentary: a Hails platform could potentially use Nexus to execute untrusted executables in an environment that is less restricting than our OS sandbox (e.g., it could have network access as directed by Nexus).

Laminar [67] combines operating systems and programming languages IFC techniques to jointly provide application and OS end-to-end guarantees. At the application level, Laminar enforces IFC within certain code regions named *security regions*—where labeled data can be accessed. This is similar to our underlying confinement system, LIO, which enforces IFC at the thread level. Unlike Hails, Laminar does not extend enforcement to the browser nor does it address a challenge that underlies most IFC systems: how to specify policy. However, their OS confinement approach is considerably more flexible than ours; as with Nexus, using Laminar’s OS-level confinement in Hails would be an interesting direction.

Another closely related work is Jeeves [87]. Jeeves is a language-level IFC system that separates policy specification from the rest of the program that implements functionality. This is similar to our approach of separating applications into MPs, where policy and data models are specified, and VCs, where the app functionality is implemented. They, however, impose this design principle in the language design, whereas Hails imposes this in the framework, atop LIO. This has allowed us to change the policy specification language over time to address usability factors—and, importantly, still allows others to use wholly different policy languages. Moreover, this design choice has allowed us to iterate on the design of LIO itself; in contrast to Jeeves, and most dynamic language-level IFC systems, LIO proves a stronger security theorem (*termination-sensitive noninterference*), supports advanced language features (e.g., threads, exceptions, recovery from IFC failures, etc.), provides good performance (as demonstrated by Hails) and does not impose new language semantics.

In Jeeves, programmers that implement functionality do so by writing policy-agnostic code. This is a desirable property since it would mean, in the context of Hails, that a VC developer would not need to be aware of the policies specified by an MP. And indeed, a subset of the Hails database API does handle labels transparently (e.g., `findOne` unlabels documents) to hide some details from the developer. However, our experience implementing web applications so far suggests that programmers *need* to be aware of and able to inspect labels. This is needed, for instance, to cleanly handle policy violations when they occur. It would, however, be an interesting exploration to extend Hails to support a programming model closer to Jeeves and refining what the balance is between exposing and abstracting policy mechanisms. This could be done by building on the work of Austin *et al.* [2].

Jeeves was recently used in the Jacqueline web framework [88], which allows Jeeves policies to be specified on data stored in a database. As discussed in Section 7, Hails does not yet support this—all application code must interface with an MP. Jacqueline’s guarantees, however, do not extend to the browser or OS.

Another closely related work is W5 [41]. 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 web site 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 a system that has been used in production.

Secure web frameworks OKWS [39], Diesel [25], and Radiatus [10] are web frameworks that use privilege separation and least privilege to reduce damages due vulnerable app code. OKWS applies privilege separation to application services, Radiatus to users, and Diesel focuses on separating database access rights. Unlike IFC, the mechanisms underlying these frameworks provide weaker guarantees. For example, they assume that applications are written by developers that have the user's best interests at heart and thus cannot protect against untrusted code or code injection attacks.

Resin [89] allows developers to specify data flow policies that the runtime then enforces on application code. This is sufficient for many applications, but like the aforementioned (non-IFC) secure frameworks, Resin provides weaker guarantees and is not appropriate for platform deployments.

Passe [5] isolates applications (into controllers) and enforces data and control-flow dependencies between the app, browser and database. The system's guarantees are not as strong as Hails' IFC guarantees and its somewhat complex training phase make its difficult for Passe to be applied to platforms. However, Passe can be used to secure existing applications, whereas Hails and most other frameworks require developers to change their applications. Moreover, Passe infers an application's policy from tests and does not require developers to a-priory specify policies. We believe that it is possible for us to infer policies from applications traces (with most-permissive policies) in much the same way, but we leave this to future work.

Mylar [63] is a web framework that provides data confidentiality and authenticity even when a server is under the control of an attacker. Mylar relies on CryptDB [62] to protect data stored in the database and assumes that code running in the browser is trusted. Mylar is mostly complimentary to our approach. In particular, we believe that COWL can be used to address the deficiencies of Mylar—that of leaking data client-side—while Mylar and CryptDB can be used to reduce (or eliminate) the need to trust MPs in Hails, which for certain applications is very desirable.

Trust management Trust Management is an approach to distributed access control and authorization, popularized in [6]. Related work includes [1,7,19,45,44]. 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 [46] 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 [48] 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 [9] 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.

SeLINQ [69], IFDB [70], and [49,50] enforce information flow control in database systems. Most of these works are complimentary. One important aspect of making Hails usable is the policy DSL (which is tied to the data model); the dependent types approach of [49,50] have a similar data model and policy application approach to ours, but in a static setting. Using database systems such as these would potentially allow Hails applications to fetch and store data without invoking MP code and thus facilitate multi-language platforms.

9. Conclusion

Ad hoc security and privacy mechanisms based on access control lists are an awkward fit for modern web frameworks that must protect sensitive user data while incorporating third-party apps. To address this, we developed Hails, a framework for building web platforms that applies confinement mechanisms at the language, OS, and browser levels, allowing mutually-untrusted apps to interact safely. Because the framework promotes information flow policies to first-class status, platform and app authors may specify policy concisely in one place and be assured that the desired constraints on confidentiality and integrity are enforced in a mandatory fashion across all components in the system, 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 apps. Beyond GitStar, we built several other platforms using Hails and enlisted several third-party developers to build VCs and MPs for these platforms. These experiences demonstrate the ability of Hails to support a platform consisting of mutually-distrustful apps written by numerous authors, where flexible security policies, as required by real-world users, can nevertheless be enforced.

Acknowledgments

We thank Amy Shen, Samuel Alazar, Nicole Crawford, Ryan Diaz, Enzo Haussecker, Michael Lublin, Ashwin Siripurapu, Eric Stratmann, for sharing their Hails development experience with us. We thank Jon Howell, Diego Ongaro, Mike Piatek, Justine Sherry, Joe Zimmerman, and the OSDI’12 reviewers for their helpful comments on earlier drafts of this paper. We thank the GitStar and UPenn CRASH teams for their helpful comments on the confinement mechanism designs and implementations. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, by multiple gifts from Google, by a gift from Mozilla, and by the Swedish research agency VR and STINT. Deian Stefan was 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] T. H. Austin, K. Knowles, and C. Flanagan. Typed faceted values for secure information flow in Haskell. *MIT Technical Report UCSC-SOE-14-07*, 2014.
- [3] A. Barth. The web origin concept. Technical report, IETF, December 2011. <https://tools.ietf.org/html/rfc6454>.
- [4] K. Bhargavan, A. Delignat-Lavaud, and S. Maffei. Language-based defenses against untrusted browser origins. In *Proceedings of the 22nd USENIX Conference on Security*, pages 653–670. USENIX, 2013.
- [5] A. Blankstein and M. J. Freedman. Automating isolation and least privilege in web services. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, pages 133–148. IEEE, 2014.

- [6] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 164–173, 1996.
- [7] 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>.
- [8] A. Bortz and D. Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th World Wide Web*, pages 621–628. ACM, 2007.
- [9] 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.
- [10] R. Cheng, W. Scott, P. Ellenbogen, J. Howell, and T. Anderson. Radiatus: Strong user isolation for scalable web applications. *University of Washington Technical Report*, 2014.
- [11] W. Cheng, D. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shriram, and B. Liskov. Abstractions for usable information flow control in Aeolus. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 12–12. USENIX, 2012.
- [12] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, pages 105–118. USENIX, 2010.
- [13] K. Chodorow and M. Dirolf. *MongoDB: the definitive guide*. O’Reilly Media, Inc., 2010.
- [14] 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.
- [15] 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.
- [16] CodeMirror. CodeMirror, November 2015. <http://codemirror.net/>.
- [17] M. B. Consulting. Jetty webserver, March 2012. <http://jetty.codehaus.org/jetty/>.
- [18] D. Crockford. Making JavaScript safe for advertising. <http://adsafe.org/>.
- [19] 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.
- [20] Disqus. Disqus, November 2015. <https://disqus.com/>.
- [21] Docker. Introduction to container security, March 2015. https://d3oypxn00j2a10.cloudfront.net/assets/img/Docker%20Security/WP_Intro_to_container_security_03.20.2015.pdf.
- [22] C. Doctorow. United website breach let fliers see each others’ private data, January 2015. <https://boingboing.net/2015/01/28/united-website-breach-let-fliers.html>.
- [23] L. Dusseault and J. M. Snell. PATCH method for HTTP. Technical report, IETF, March 2010. <https://tools.ietf.org/html/rfc5789>.
- [24] Facebook. Fbjs (Facebook JavaScript). <http://developers.facebook.com/docs/fbjs/>.
- [25] A. P. Felt, M. Finifter, J. Weinberger, and D. Wagner. Diesel: applying privilege separation to database access. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 416–422. ACM, 2011.
- [26] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 25–32. ACM, 2000.
- [27] FP Complete. School of Haskell, November 2015. <https://www.fpcomplete.com/school>.
- [28] M. Garrett. Container security with SELinux and CoreOS, September 2015. <https://coreos.com/blog/container-security-selinux-coreos/>.
- [29] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*. USENIX, October 2012.
- [30] Google. Google code prettify, September 2012. <http://code.google.com/p/google-code-prettify/>.
- [31] S. Heule, D. Stefan, E. Yang, J. C. Mitchell, and A. Russo. IFC inside: Retrofitting languages with dynamic information flow control. In *Proceedings of the 4th Conference on Principles of Security and Trust*. Springer, April 2015.
- [32] C. Hrițcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. Exceptionally available dynamic IFC. Submitted to POPL, July 2012.
- [33] M. Huilgol. Facebook’s latest security vulnerability allows third party applications to delete Facebook pages permanently, August 2015. <http://techpp.com/2015/08/27/facebook-pages-security-vulnerability/>.
- [34] S. Isaacs. Microsoft web sandbox. <http://www.websandbox.org/>.

- [35] W. ISO. 9241-11. ergonomic requirements for office work with visual display terminals (vdts). *The international organization for standardization*, 1998.
- [36] R. Kainda, I. Flechais, and A. Roscoe. Security and usability: Analysis and evaluation. In *Proceedings of the 5th International Conference on Availability, Reliability, and Security*, pages 275–282. CPS, 2010.
- [37] S. Khandelwal. Facebook vulnerability allows hacker to delete any photo album, February 2015. <https://thehackernews.com/2015/02/hacking-facebook-photo-album.html>.
- [38] S. Kovach. Nearly 7 million Dropbox passwords have been hacked, October 2014. <http://www.businessinsider.com/dropbox-hacked-2014-10>.
- [39] M. Krohn. Building secure high-performance web services with okws. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 185–198. USENIX, 2004.
- [40] 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.
- [41] M. Krohn, A. Yip, M. Brodsky, R. Morris, and M. Walfish. A world wide web without walls. In *Proceedings of the 6th ACM Workshop on Hot Topics in Networking*. ACM, 2007.
- [42] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, August 2001.
- [43] 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>.
- [44] 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.
- [45] 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.
- [46] 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.
- [47] F. Lifton. Advancing Docker security: Docker 1.4.0 and 1.3.3 releases, December 2014. <https://blog.docker.com/2014/12/advancing-docker-security-docker-1-4-0-and-1-3-3-releases/>.
- [48] 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.
- [49] L. Lourenço and L. Caires. Information flow analysis for valued-indexed data security compartments. In *Trustworthy Global Computing*, pages 180–198. Springer, 2014.
- [50] L. Lourenço and L. Caires. Dependent information flow types. In *Proceedings of the 42nd Symposium on Principles of Programming Languages*, pages 317–328. ACM, 2015.
- [51] J. MacFarlane. Pandoc: a universal document converter. <http://johnmacfarlane.net/pandoc/>.
- [52] S. Maffei and A. Taly. Language-based isolation of untrusted javascript. In *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE*, pages 77–91, 2009.
- [53] J. Mayer and J. Mitchell. Third-party web tracking: Policy and technology. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pages 413–427. IEEE, 2012.
- [54] 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.
- [55] N. Mitchell. *HLint Manual*. <http://community.haskell.org/~ndm/darcs/hlint/hlint.htm>.
- [56] B. Montagu, B. Pierce, R. Pollack, and A. Surée. A theory of information-flow labels. *Draft*, July, 2012.
- [57] D. Mosberger and T. Jin. httpperf-a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, pages 31–37, 1998.
- [58] Mozilla. Persona, October 2015. <https://developer.mozilla.org/en-US/Persona>.
- [59] 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.
- [60] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Computer Systems*, 9(4):410–442, October 2000.
- [61] J. Petazzoni. Containers & docker: How secure are they?, August 2013. <https://blog.docker.com/2013/08/containers-docker-how-secure-are-they/>.
- [62] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

- [63] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan. Building web applications on top of encrypted data using mylar. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation*, pages 157–172. USENIX, 2014.
- [64] T. Preston-Werner. Public key security vulnerability and mitigation, March 2012. <https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation>.
- [65] D. Recordon and D. Reed. Openid 2.0: a platform for user-centric identity management. In *Proceedings of the 2nd ACM workshop on Digital Identity Management*, pages 11–16. ACM, 2006.
- [66] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *Proceedings of the 29th Conference on Programming Language Design and Implementation*, pages 159–169. ACM, 2008.
- [67] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical Fine-grained Decentralized Information Flow Control. In *Proceedings of the 30th Conference on Programming Language Design and Implementation*, pages 63–74. ACM, 2009.
- [68] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9): 1278–1308, September 1975.
- [69] D. Schoepe, D. Hedin, and A. Sabelfeld. Seling: tracking information across application-database boundaries. In *Proceedings of the 19th International Conference on Functional Programming*, pages 25–38. ACM, 2014.
- [70] D. Schultz and B. Liskov. IFDB: decentralized information flow control for databases. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 43–56. ACM, 2013.
- [71] B. Shackel. Usability-context, framework, definition, design and evaluation. *Human factors for informatics usability*, pages 21–37, 1991.
- [72] Sinatra. Sinatra, September 2012. <http://www.sinatrarb.com/>.
- [73] 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.
- [74] E. Steel and G. Fowler. Facebook in privacy breach. *The Wall Street Journal*, 18, October 2010. <http://online.wsj.com/article/SB10001424052702304772804575558484075236968.html>.
- [75] D. Stefan. Confinement with origin web labels, October 2015. <http://www.w3.org/TR/2015/WD-COWL-20151015/>.
- [76] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *Proceedings of the NordSec 2011 Conference*, October 2011.
- [77] 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.
- [78] 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.
- [79] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming*, 2012. Under revision.
- [80] D. Stefan, P. Buiras, E. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *Proceedings of the 18th European Symposium on Research in Computer Security*, pages 781–735. Springer, 2013.
- [81] D. Stefan, E. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières. Protecting users by confining JavaScript with COWL. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*. USENIX, October 2014.
- [82] 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>.
- [83] 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.
- [84] H. team. The hail [Haskell] package. <http://hackage.haskell.org/package/hails>.
- [85] D. Terei, S. Marlow, S. P. Jones, , and D. Mazières. Safe Haskell. In *Proceedings of the 5th Symposium on Haskell*, September 2012.
- [86] E. Yang, D. Stefan, J. Mitchell, D. Mazières, P. Marchenko, and B. Karp. Toward principled browser security. In *Workshop on Hot Topics in Operating Systems*. USENIX, May 2013.
- [87] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. pages 85–96, 2012.

- [88] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong. End-to-end policy-agnostic security for database-backed applications. *CoRR*, abs/1507.03513, 2015. URL <http://arxiv.org/abs/1507.03513>.
- [89] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22nd Symposium on Operating Systems Principles*, pages 291–304. ACM, 2009.
- [90] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. Oct. 2001.
- [91] 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.
- [92] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation*, pages 293–308, San Francisco, CA, April 2008.
- [93] L. Zeltser. Security risks and benefits of docker application containers, June 2015. <https://zeltser.com/security-risks-and-benefits-of-docker-application/>.
- [94] 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.