DATS — Programmability and Security by Design for Web Applications

Casen Hunger[1]
University of Texas at Austin
casen.h@utexas.edu

Lluís Vilanova[1]
Technion — Israel Institute of Tech.
vilanova@technion.ac.il

Charalampos Papamanthou
University of Maryland (UMD)
cpap@umd.edu

Yoav Etsion
Technion – Israel Institute of Technology
yetsion@tce.technion.ac.il

Mohit Tiwari
University of Texas at Austin
tiwari@austin.utexas.edu

Abstract

Using data-centric containers for isolation is a very effective way to give users control of their data and avoid information leaks from untrusted applications. However, they are hard to program for and lead to inefficiencies, since they require replicating an application inside each container.

We propose DATS — a system to run web applications that retains application usability and efficiency through a mix of hardware capability enhanced containers and the introduction of two new primitives modeled after the popular model-view-controller (MVC) pattern. (1) DATS introduces a *templating language* to create views that compose data across data containers. (2) DATS uses *authenticated storage and confinement* to enable an untrusted storage service such as memcached and deduplication to operate on *plain-text data* across containers. These two primitives act as *robust declassifiers* that allow DATS to enforce *non-interference* across containers, taking large applications out of the trusted computing base (TCB).

We showcase various web applications including gitlab and a Slack-like chat (eight in total), significantly improve the worst-case overheads due to application replication, and demonstrate usable performance for common-case usage.

CCS Concepts • **Security and privacy** → **Authentication**; **Access control**; **Authorization**; **Web application security**; *Operating systems security*; *Information flow control*

[1] Equal contributors.

ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4911-6/18/03...$15.00
DOI: https://doi.org/10.1145/3173162.3173213

1. Introduction

Web applications have to implement a wide array of security features, such as input sanitizing, authorization, and access control checks [13]. However, developers often implement these features incorrectly [8, 18] and do not update all libraries promptly [19]. Web applications are vulnerable to "zero-day" vulnerabilities [5]. A compromised web application can then exfiltrate data to unauthorized users and cause large data breaches. Worse, the threat of exploits forces considerable penetration-testing and compliance-certification work that slows down application development.

Ideally, users and enterprises would store their data on storage platforms (e.g., Google Drive or electronic medical record (EMR) systems), use *untrusted* web applications that integrate with these storage platforms, *and yet* protect their data from being breached — i.e., enforcing mandatory access control (MAC) over the untrusted applications. Figure 1 shows a simplified setting where a doctor (Dave) shares folders with patients Alice, Bob, and Eve, and uses untrusted applications for messaging and scheduling.

A natural strategy is to use *data containers*: run an entire application instance within the context of each data object, with each instance isolated into a separate container[35, 44, 69, 77]. The access control rules will hold by definition even if an application instance is compromised or malicious (no information can be transferred across containers for different data objects). One could use language-level information flow control (IFC) to achieve the same goals (e.g., Hails [39] provides a Haskell framework that attaches labels to data in database models), but this compromises programmability. Developers rely on a large body of existing frameworks and languages, and we cannot therefore limit them to only using vetted options. The advantage of most OS container technologies is that they can isolate unmodified code.

Data containers thus raise two new challenges. **Usability**: for example, a calendar application cannot aggregate into a

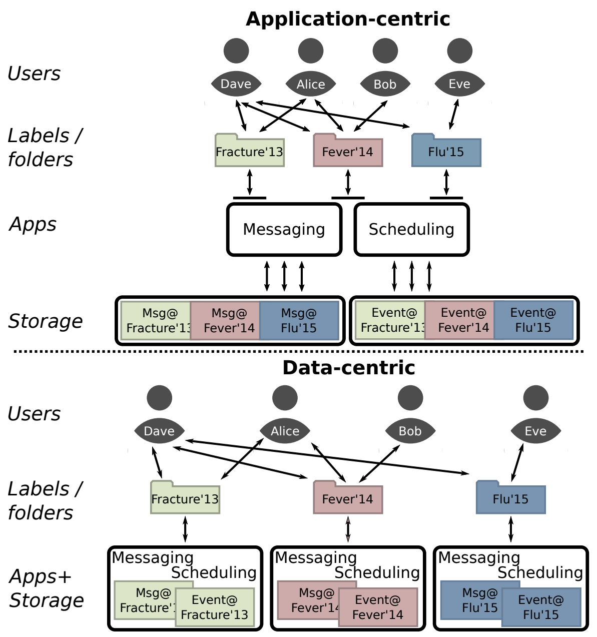


Figure 1: Current systems have *application-centric access controls* (top) and cannot prevent data leaks; e.g., a compromised or malicious "Scheduling" application can leak Alice's events to Eve (from "`Fracture'13`" to "`Flu'15`"). *Data-centric access controls* (bottom) enforce users's ACLs on all applications: data is confined to its respective label.

single page the information from the appointments in different data objects. **Efficiency**: applications cannot use a single storage services like deduplication across data objects.

In this paper we present the DATS system, which refactors authentication and access controls outside of *untrusted* web applications. DATS couples data containers in a novel way to trivially enforce access controls at the container level, together with two new mechanisms to *robustly declassify* [76] information from multiple data objects. It leverages the model-view-controller (MVC) pattern common in many web applications, while maintaining application *programmability* by presenting a familiar programming model to developers and supporting several existing web application frameworks and languages.

First, DATS recovers *usability* by securely composing views from multiple data objects. Applications can provide an **untrusted *view* template** to a trusted template processor to aggregate information from each per-data object container into a single page, like in many existing view templating languages [16]. The declassifier transparently applies language level IFC to prevent information leaks across data objects.

Second, DATS improves *efficiency* by securely sharing untrusted storage services across data objects. Applications can use a trusted storage declassifier that interposes between applications and **untrusted shared *model*** (e.g., for deduplication, key-values stores, compression, etc.). The declassifier performs integrity checks on each data read operation (i.e., "`get(key`$_x$`)`") to ensure that it only returns the value from the most recent "`put(key`$_x$`, val)`" operation on the same data object – interestingly, we use this to ensure that storage does not leak information across data objects.

Container technologies are available in production systems, but replicating an application instance on each of them has intrinsic inefficiencies known as multi-execution [26]. Performance is largely secondary for enterprises (they have far fewer users than internet-scale services), but multi-execution can become a problem when operating across a large number of data objects (e.g., search). We therefore also explored using harwdare-assisted thread containers to avoid multi-execution.

DATS and the client browser are the only components that reside in the trusted computing base (TCB), while ensuring applications remain programmable, usable and efficient. IFC efforts like Hails [39] and Jeeves [74] are instead a great fit to the trusted developers of DATS's TCB. Enterprises can then leverage the vast space of existing and untrusted web applications, frameworks, languages and developers, and at the same time avoid costly application code audits. We make the following contributions:

- We design and implement the DATS system (§§ 3 and 4).
- We evaluate programmability and security by developing 4 applications and porting 4 existing ones (§§ 5 and 6).
- We evaluate performance with existing OS-level containers [10, 61] and a HW-capability architecture [32, 66] (§ 7).

2. Motivation

Many enterprises use web-based applications for security-sensitive data (e.g., a hospital). Currently, the TCB includes *every* application since an application-level exploit can put all data at risk. Such applications are thus built, certified, and audited for security first, with performance being a secondary concern [9]. This accrues large costs from highly skilled security-aware programmers, and long and costly penetration-testing and compliance-certification work before the smallest changes can be pushed into production. This is clearly at odds with rapid application development and deployment cycles and cost-efficiency; one cannot employ the vast majority of existing web developers, who are not security-savvy, nor leverage existing applications and development frameworks. We therefore want a systematic approach to provide security and cost-efficiency for such security-sensitive applications.

Figure 1 shows a simplified setting where a doctor (Dave) shares **folders** (i.e., access control domains or security labels) with patients Alice, Bob, and Eve, and uses untrusted applications like "Messaging" and "Scheduling". Access control lists (ACLs) provide users a simple and intuitive way to express data confidentiality expectations for many applications; e.g., Dave explicitly decides to share folder

"`Fracture'13`" with Alice in Figure 1. Our "folders" are the "control domains" in the security literature, and folder ACLs serve as security labels for the data inside folders.

Existing systems provide *application-centric* policies (top of Figure 1); they can isolate applications from each other, but a compromised application can access data from multiple access control domains, even if users correctly set their folders's ACLs. For example, a buggy or malicious "Scheduling" application can store events intended for folder "`Fracture'13`" into "`Flu'15`" when processing Dave's appointments. Even if all applications restrict Eve to only access "`Flu'15`", this folder now contains events exfiltrated from "`Fracture'13`". In the worst case, data from *all* folders could be exfiltrated, violating all user confidentiality expectations. Instead, a *data-centric* MAC policy (bottom of Figure 1) provides an intuitive alternative, where data from different folders cannot interfere with each other.

In this section, we describe the challenges in enforcing data-centric MAC over untrusted applications, and the opportunity inherent in the structure of MVC applications to refactor security out of the application requirements.

2.1 Threat Model

Our threat model for web applications has the client browser and the server-side platform (i.e., the hardware, hypervisor and DATS's core components) as part of the TCB. The applications, their libraries and storage backends (such as key-value stores) are hosted on the platform, but are *outside* the TCB; this also includes application code running on the client's browser (e.g., using JavaScript). DATS aims to prevent the following attacks by refactoring authorization and access control out of the untrusted components:

Code-driven folder interference: Untrusted code and developers are considered an attacking subject — either directly through malicious intention, or indirectly through buggy code — and must not be able to leak data across *any* two folders (or arbitrary internet addresses). In information flow terms, the users's requirement is *non-interference [40] across folders*.

User-driven folder interference: Users (authorized or otherwise) are considered an attacking subject and must be prevented from violating non-interference across *unauthorized* folders. This includes exploiting the untrusted code through memory errors, malicious inputs, etc. or running arbitrary code to access unauthorized folders.

We otherwise consider authorized users trusted; they fully delegate trust in handling sensitive data to other users with whom they share folders. We do not protect against poor judgment when sharing a sensitive file with another user, who is free to manually communicate its contents to another *authorized* folder or anywhere else.

We limit the scope of our problem and list **complementary techniques** to handle other risks to user data:

- *Integrity:* We do not prevent untrusted applications from mangling user data or destroying it. Systems such as Frientegrity [37] handle it orthogonally.
- *Information leaks via timing or termination channels:* We assume techniques that either randomize (e.g., fuzzing [65]) or normalize (e.g., deterministic execution [25] or predictive mitigation [23]) the timing of outputs (e.g., on a `storage`$\rightarrow$`app` channel, possibly coupled with behavioral/anomaly detection for termination channels).
- *Privacy-preserving data mixing:* Cross-folder functionality such as analytics (e.g., clustering or training classifiers) fundamentally violates the access control policies set by users. Complementary approaches such as differential privacy [34, 50, 51, 55, 56] or quasi-identifier based privacy [47, 49, 64] show that this functionality can directly integrate with DATS: the functions can be executed in isolation and only their perturbed output released (e.g., GUPT [51]). Similar declassifiers can be built for advertisement impressions [45] and for sending debugging output [27] back to developers.

2.2 Challenges in enforcing Data-centric MAC

OS-level MAC: The simplest way to enforce data-centric MAC is to run an application process per folder inside an OS-level container using LinuX Containers [10], SELinux MAC [61], capabilities [69], or IFC at the OS-level (OS-IFC) [35, 44, 52, 77].

This approach curtails crucial functionality. It cannot support web pages that display data from multiple folders (i.e., cross-folder views), since the application has access to a single folder. For example, Dave will need to open each per-folder "Scheduling" application to manually find a free slot for a meeting, a tedious and error-prone operation.

A per-folder application instance can also be inefficient, since a single MVC model cannot optimize storage across folders. For example, *data deduplication* has to work across all folders in order to find *unencrypted* data to consolidate. Storage services such as in-memory key-value stores (e.g., Redis, memcached) or distributed coding also work across folders and will be inefficient if we run a separate instance on each folder. Furthermore, running multiple instances of an application can sometimes lead to poor resource utilization in the micro-architecture (i.e., *multi-execution* [26]).

PL-level MAC: Programming-language (PL) level IFC [39, 59, 75] can control the flow and aggregation of information *inside* an application (such as creating cross-folder views) and enforce data-centric MAC policies. But it requires careful annotation of program inputs, intermediate variables, and outputs to ensure information does not flow between inputs and outputs with incompatible ACLs. This makes PL-MAC notoriously complex to use in practice, requires developers to have extensive security expertise, and constrains them to specific programming frameworks. Further, PL-MAC only works as long as application developers are trusted [59], or when the trusted platform developers maintain the data model for all third-party applications [39].

Finally, optimizations such as deduplication cannot be represented in a PL-MAC solution, since it compares data across folders and executes code based on this comparison.

2.3 Opportunities in MVC Applications

MVC frameworks are popular in web applications and require developers to write separate components and strict interfaces between *views* and *controllers* as well as *controllers* and *models*. We can use these interfaces to transparently enforce data-centric MAC over the entire web application.

Views in MVC applications separate presentation from application logic using templating languages. These languages are meant to arrange data in different ways instead of creating new information from data, so they allow us to impose PL-level MAC without being undermined by pointers, reflection, and other features of full-featured languages.

Storage (i.e., model) optimizations implement diverse functionality in arbitrary languages, but they do not affect the ACLs of user data — i.e., storage components are invisible to users. Furthermore, several popular and complex web services expose variants of a simple `put-get` interface, like deduplication, in-memory key-value stores (memcached), and cloud-based storage services such as Amazon S3 and Google Drive. Interposing a transparent integrity-checking proxy on this common interface enables untrusted services to work with plain-text data. This is essential for services like deduplication (that will not work if data is encrypted) and allows services to compress, index, or otherwise optimize storage across all folders.

3. Design of the DATS System

DATS ensures that untrusted applications cannot mix information across folders by executing a unique web-application instance for each folder. In this section, we describe how DATS enables untrusted applications to implement cross-folder views and storage without breaking this folder non-interference policy. We also describe hardware-assisted OS containers that help reduce the overhead of multi-execution in DATS. We begin in Figure 2 with a typical user workflow, and walk through the key steps in the lifecycle of an application on DATS.

DATS exposes `Data` objects (called *folders*) to users, similar to platforms like Google Drive, Box, or Dropbox. Users express their confidentiality policies (i.e., who can access which folders) using simple per-folder ACLs. Every folder can hold arbitrary data, making them independent of application-specific constructs like messages, documents, medical records, etc., and DATS enforces folder-level ACLs on untrusted third-party applications that attach to users' Drive/Box accounts. Folders are color-coded in the figure (green, red or blue) to easily track information flow across the system. Like Google Drive, users, folders, and their ACLs are implemented by DATS – applications need not implement further ACL rules.

A typical user flow will start with an application landing page such as a calendar overview across all their folders (Ⓐ and Ⓑ in Figure 2). The user can then click on specific appointments to get or set additional information (Ⓒ), and move in and out of calendar (cross-folder) and appointment (per-folder) views. Users can also set preferences used across folders, such as profile pictures, names, etc. Since this data is made explicitly public to all folders without using any folder, we term such functionality as non-folder.

Based on this usage model, DATS starts an application in a *non-folder* container (❶), which cannot access user data from any folder (only sees the users' public settings and application's resources). DATS sets up a container to create *cross-folder* views (2a and 2b leading to Ⓑ), and then sets up *per-folder* containers in ❸ as the user traverses links out of *cross-folder* into *per-folder* views. We now describe the developer's view of application components on DATS.

`App` components contain entire (untrusted) applications and are instantiated multiple times. Each instance runs in an OS-level container [4, 10, 60] (`app` column in Figure 2) with a trivial data-centric MAC policy: it has exclusive access to a single folder, and no other external resources. Containers get a standard system view, including system calls, libraries, and runtimes for any development framework, and also cover the client browser. This is key for *programmability* given the diversity in toolkits and languages for developing applications.

This "data container" approach however forbids cross-folder functionality, breaking the usability and efficiency of most applications. DATS thus provides two *robust declassification* [76] mechanisms for cross-folder functionality, guaranteeing that untrusted code cannot affect declassified data.

`Template` files describe how to aggregate information from multiple folders into a single cross-folder view, making applications *usable*. The `template` in Figure 2 is provided by the non-folder `app` in 2a. A trusted *Template Processor* inflates `templates` with data from each per-folder `app` instance (2b) through a simple form of PL-level IFC: it generates HTML/JS by processing only one data element at a time, and each output (e.g., link) can only send information back to the `app` instance that produced it (e.g., ❸).

`Storage` components (untrusted `storage` column in Figure 2) implement the application's *cross-folder models* to make these *efficient*. DATS interposes a trusted *Storage Declassifier* on the `app`–`storage` communication. The *Storage Declassifier* uses integrity checking to ensure that each response `value` of a "`get(key)`" request from a folder is the same as the last "`put(key, value)`" for that folder. Interestingly, integrity checking the put-get interface and confining the `storage` services in a container enables these untrusted services to work with *plain-text* data.

We start with the security invariants in DATS (§ 3.1), followed by details about each component (§§ 3.2 to 3.6).

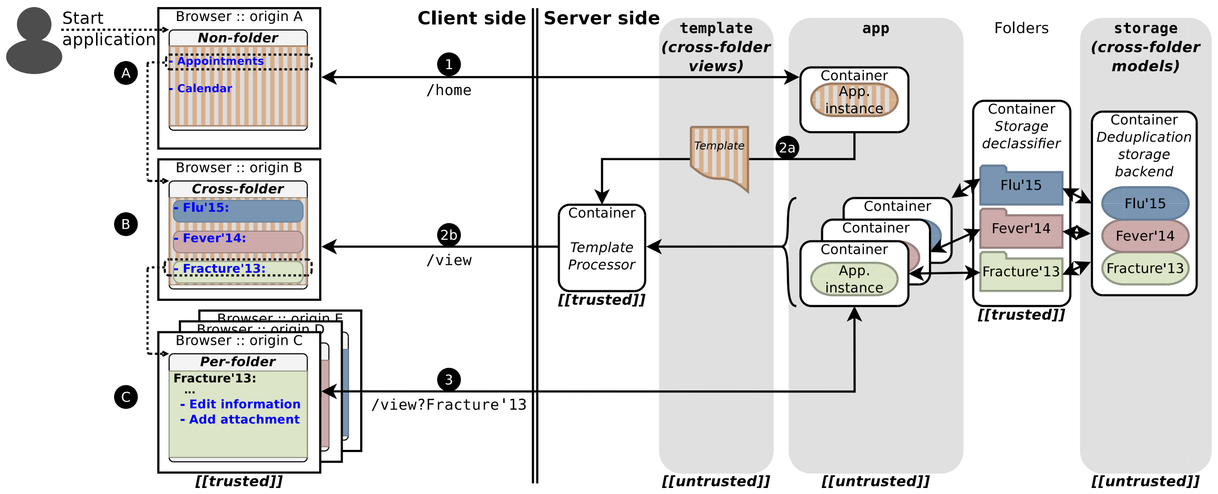


Figure 2: Example web page flow from a user ("client-side"), DATS's main components and an application's app-template-storage components (and their relation to MVC). Application code & data and storage backends are untrusted (grayed areas), while DATS enforces folder non-interference. Application components run inside OS-level containers, which can very easily enforce per-folder MAC policies. Note that the client's browser is allowed to run untrusted application code (e.g., JavaScript).

3.1 Security Invariants in DATS

DATS guarantees end-to-end folder non-interference, covering apps and information flow-secure views and storage.

(1) Each app instance runs in a confined container – conceptually, each container covers server and client devices – with access to only one folder. This ensures baseline folder non-interference for app instances.

(2) DATS's *Template Processor* uses untrusted templates from the application and untrusted results from many folders to construct a view, but ensures that communication from the view back to an app instance only uses data from that same instance. The *Template Processor* prevents explicit information flows by ensuring each HTML element that can send a request (e.g., a link) contains information from at most one folder, and points to the app instance that has access to it. It also prevents implicit information flows by using a template language that does not allow predicates and conditional loops over results from different folders. Fortunately, view templating languages are already moving towards such restrictions to minimize business logic inside the presentation layer [12].

(3) DATS's *Storage Declassifier* ensures that a get value returned to an app equals that last value that was put by the app – hence, the untrusted storage service can operate on plain-text data across folders and yet be prevented from breaking folder non-interference. Integrity checking in *Storage Declassifier* not only prevents storage from explicitly copying Alice's data into a get response for Bob, but also prevents implicit leaks where storage returns Bob's value X if Alice's secret bit is 0 and Bob's value Y if Alice's bit is 1. storage is confined to a container with no outputs other than to the *Storage Declassifier*, assuming that timing/termination channels from storage to app are addressed using complementary techniques (see § 2.1). DATS *Storage Declassifier* and confinement together demonstrate a novel method of using an authenticated storage interface to enforce non-interference.

DATS's guarantees rely on a few fundamental primitives: containers on server and client, IFC in the *Template Processor*, and integrity checking in the *Storage Declassifier*. The *Template Processor* and *Storage Declassifier* provide *robust declassification* [76] for cross-folder functionality – i.e., guarantee that untrusted code or data in a folder cannot affect messages/data that is sent from a cross-folder container back to a different per-folder container.

DATS extends containers into the client browser by using one sub-domain per app instance (browser origin in Figure 2). DATS then activates the Same-Origin Policy (SOP) and Content Security Policy (CSP) on the client browser to limit access to remote resources and confine untrusted client-side code (e.g., JavaScript) within each sub-domain (container). DATS also drops all cookies set for parent domains. Containers on the server can be implemented directly as reference monitors (e.g., SELinux or LXC), or using capabilities or information flow control – DATS deployers can pick one based on performance and compatibility constraints.

Components interact with DATS abstractions using an authenticated RESTful interface. Therefore, trusted components are no different from the untrusted ones, except that they are authorized to perform operations such as user authentication. Internally, each container is identified by the application it is running (e.g., *Health*), the user it is running on behalf of (e.g., *Alice*), and the folder it has access to (e.g., "*Fracture'13*"). As used throughout this section, the green container in Figure 2 would be identified as <Health, Alice, Fracture'13>.

3.2 `Data` object (*folder*) and User Management

DATS provides users two *trusted* applications – *Login* authenticates and manages users, and the *Desktop* allows users to create, delete, and share folders, and launch untrusted applications. Access control over folders lets users control the granularity at which their data is shared – a user may wish to assign a single medical encounter to a folder, allowing them to share each of their encounters individually, or a user may wish to place all their encounters within a single folder, allowing them to share their entire medical history at once.

DATS provides two services to retain application functionality after removing access-control and authentication: `app` instances contact the *User Service* to see which users have access to their current folder, and the *Template Processor* uses `templates` to create cross-folder views (§ 3.4).

DATS provides the *User Service* because applications work with the 'user' abstraction for a large portion of their functionality, in addition to access control, and removing the concept of users from applications only works for the simplest of applications. For instance, Mattermost (see § 5.1) has 92 different SQL queries involving its user table, some of which are complex and include joins across multiple different tables. To ease application development, DATS provides a trusted *User Service* that allows *per-folder* `app` instances to read information about the users who have access to the `apps`' folder and to trigger notifications to other `app` instances associated with the same folder (e.g., those running on behalf of another user).

3.3 `App` components

The `app` components contain most of the application logic.

DATS has a trusted *Proxy* that routes traffic between the sub-domains used by client browsers and the `app` instances (i.e., containers). Non-folder and per-folder functionality are the most common (❶ and ❸) and can have native performance (DATS acts as a reverse proxy between the client browser and `app` instances). The *Proxy* extends containers to the client side to enforce indirect folder non-interference: it activates the browser SOP and CSP and drops cookies for parent domains (see § 3.1). Every time an `app` is instantiated it is assigned a new random sub-domain. For example, DATS maps the non-folder view in Ⓐ of Figure 2 to the `app` instance `<Health, Alice>` (❶, without access to any folder), and the per-folder view in Ⓒ to the instance `<Health, Alice, Fracture'13>` (❸, with access to folder "*Fracture'13*"); the cross-folder view in Ⓑ also has its own sub-domain, but its contents are served directly by the *Proxy* (see § 3.4).

DATS has a trusted *Container Manager* that can be instantiated in multiple nodes to scale horizontally. It enforces direct folder non-interference by creating mutually isolated containers: each container is assigned an IP and port to listen for requests and only has access to its assigned folder (if any, since non-folder `app` instances cannot access any folder) and to DATS's public API (see § 3.6). To avoid the latency costs of spinning up containers on-demand, the *Container Manager* uses container reuse, pooling and prefetching.

3.3.1 Hardware-Accelerated Thread Containers

By default, DATS uses OS-level containers [4, 10, 60] to enforce isolation (such as LXC and SELinux as described in § 4.1) and make applications *programmable* (developers can pick from many application stacks in existence today). Nonetheless, using multiple `app` instances has inherent multi-execution overheads [26], which are specially acute when a cross-folder view such as search requires streaming through all folders that a user has access to. Clearly, such large sweeping operations will put container startup and network setup latencies in the critical path of a `view` query.

To address this slowdown, we have designed thread-level containers in DATS. We use SELinux [61] and fine-grained capability-based architectures [32, 46] together to build "thread containers" so that the DATS *Proxy* and `app` code can execute in the same process, and the *Proxy* can trigger a large number of thread containers – one `app` runs in each thread and performs the query (e.g. search) over data in one folder. §§ 4.1 and 7.2 describe our implementation of hardware-assisted thread containers, even though one could also use a DIFC architecture like Raksha [31] or Loki [78] instead (where each folder would be a mutually unordered label in the policy lattice). Interestingly, thread containers are in line with event-triggered plugins used on a web server frontend (e.g., WSGI [20]).

3.4 `Template` Components

Cross-folder views retain critical functionality to make applications *usable*, but must ensure folder non-interference. `Templates` compose views *declaratively* with information from multiple folders, without executing any untrusted code. Developers can use any static resource in cross-folder views, plus some trusted JavaScript provided by DATS.

`Templates` are written in a simple language, like views in many existing MVC web applications [16]. They are based on the stateless Mustache language [12] (without explicit control-flow operations like loops), making it easy to apply IFC to ensure folder non-interference[2]: each element in Ⓑ contains information from at most one folder and can only send a request to the `app` instance that provided that information (e.g., URLs for images, links and forms).

The *Template Processor* forbids arbitrary JavaScript in `templates`, since current browsers do not provide IFC on client-side code (e.g., unlike COWL [62]). Instead, the *Template Processor* recognises additional tags that expand to trusted snippets of JavaScript for searching, sorting and auto-completing elements (see Table 1). Such tags can also be

[2] The same approach could also be applied to any application generating user-facing views (e.g., using Android layout templates). A purposely simple templating language side-steps the precision vs. soundness problems of applying IFC to full-featured languages [59, 68].

```
<a href="/home">Home</a>
{{#DATS.results}}
{{#events}}
<a href=
  "{{#DATS.enter}}/view?{{title}}{{\DATS.enter}}">
    <span>{{title}}</span></a>
{{\events}}
{{\DATS.results}}
```

(a) Template

```
[{'events':
  [{'title':"Flu'15"}]}]
[{'events':
  [{'title':"Fever'14"}]}]
[{'events':
  [{'title':"Fracture'13"}]}]
```

(b) Per-folder JSON data

```
<a href="/home">Home</a>
<a href="/random1/view?Flu'15">
    <span>Flu'15</span></a>
<a href="/random2/view?Fever'14">
    <span>Fever'14</span></a>
<a href="/random3/view?Fracture'13">
    <span>Fracture'13</span></a>
```

(c) Expansion

Figure 3: A cross-folder `template` and an example expansion from the *Health* application. DATS' keywords are bold in black, and *Health*'s per-folder data is highlighted in red. Some of the HTML has been omitted for brevity.

used to generate scripts that integrate differential privacy databases like PINQ [50] into a DATS application.

In practice, we found that a most application functionality lies in non-folder and per-folder views (and can therefore include arbitrary JavaScript). Also, programmers can use their own templating language to produce a valid `template`.

3.4.1 Cross-Folder View Example

This section shows the steps involved in using a cross-folder view for the examples in Figures 2 and 3:

- When the user clicks the "Appointments" link Ⓐ in their browser an HTTPS request for `/view` is sent to the *Proxy*. The *Proxy* then forwards the request to the non-folder app instance running for that user based on the sub-domain the request was sent for.
- The non-folder application responds with a `template` 2a (Figure 3a) and triggers a view of data across all folders by setting the `x-dats-crossfolder` header field.
- The *Proxy* begins constructing the cross-folder view by redirecting the client to a new temporary sub-domain Ⓑ.
- In parallel, the *Proxy* creates (or reuses) an app instance for each of a user's folders and replays the `/view` request to each of them.
- Each per-folder app instance responds to the request with a JSON list of results (Figure 3b). The DATS API informs App instances if they are executing in non-folder or per-folder mode so that logic on the "shared" `/view` endpoint can change accordingly.
- The *Template Processor* collects the per-folder results and uses them to inflate the `template` (2b). The result is served by the *Proxy* as the cross-folder view in Ⓑ.

  The `template`'s top-level tagged block `DATS.results` (Figure 3a; there can be many, but not nested) is inflated with each per-folder result in turn (Figure 3b).

  URLs in inflated regions need to use the tagged block `DATS.enter` (returns an error otherwise), which adds a random prefix to them (tracked by the *Proxy*; see below). Programmers can use their own tags to reference per-folder information inside a `DATS.results` block (e.g., red tags for `events` and `title` in Figures 3a and 3b).
- The *Proxy* redirects a request for `/home` in Figure 3c (or any other resource outside the `DATS.results` region; outer orange region with vertical stripes in Ⓑ) to the sub-domain for the non-folder app instance (i.e., that serving Ⓐ). A request for `/random3/view?Fracture'13` (or any resource created with `DATS.enter`; inner folder-colored regions in Ⓑ) is instead redirected to the sub-domain for the corresponding per-folder app instance (i.e., that serving Ⓒ, by stripping the random prefix to get the endpoint `/view?Fracture'13` that ❸ will serve).

3.4.2 Non-Secure Alternatives

The intuitive solution to creating cross-folder `views` based on OS-level IFC would be to create an app instance with read-only access to all folders. This folder will acquire the highest secrecy label, and cannot let users click on links in views (e.g., an inbox) to go to a message in a folder. This is because the container could embed information from folder "*Fracture'13*" into a message to a container for folder "*Fever'14*" (e.g., to edit an appointment there) and violate folder non-interference. Alternatively, one could whitelist specific URLs *and* the data sent through each URL to prevent explicit leaks as above. In this case, a malicious app can create an implicit leak by picking white-listed message A vs. message B to indicate 0 vs. 1 – the `view` component in DATS fundamentally requires information flow control *inside* a container. We emphasize that DATS could have used any language with information flow control for `templates` – we picked a logicless templating language since software engineering practices simplify the *Template Processor*'s task.

3.5 `Storage` Components

`Storage` components increase application *efficiency* by running untrusted functionality on *plain-text* across folders (i.e., cross-folder storage services or models in an MVC application). Efficiency can be in terms of storage space (e.g., cross-folder data deduplication) or other resources (e.g., using a single service instance to avoid multi-execution [26]).

DATS isolates `storage` components inside a container, and `app` instances must use the *Proxy*'s `/port` endpoint to request access to `storage` instances. It returns a connection to a transparent trusted *Storage Declassifier* (also inside a container) interposed to the `storage` component to ensure that it cannot take data from one folder and output it to another folder's `app` component. *Storage Declassifiers* associate the requesting `app`'s IP to their assigned folder.

Endpoint	Usage
User Service	
`/users`	Get per-folder user info.
Proxy	
`/port`	Connect to a `storage` component
Application	
`/dats/start`	Send folder and user info to apps.
`/dats/update`	Send updated user info to apps.
`/dats/quit`	Apps return from per-folder view
Template	
`DATS.results`	Inflate region with per-folder info.
`DATS.enter`	Create per-folder URL
`DATS.search`	Trusted JS for searching tags.
`DATS.sort`	Trusted JS for sorting by tags.
`DATS.autocomplete`	Trusted JS for autocomplete by tags.

Table 1: Public API in DATS. User Serivce, Proxy, and Template endpoints are implemented by DATS. Application endpoints are implemented by untrusted apps.

Many popular applications have models using (variants of) `put–get` interfaces, where integrity checks are sufficient to ensure non-interference. The declassifier checks that a "`get(key)`" request returns the most recent value for "`put(key, value)`" (otherwise aborts the connection). The same approach can be transparently applied to other interfaces, like disk blocks or file systems [63].

Finally, many existing large code-bases use trusted databases that have already gone through extensive audits (e.g., trusted developers assign roles to implement access policies). In this case the declassifier creates per-folder databases on a single `storage` instance, and uses the existing access control mechanisms to limit each app instance to its corresponding folder's data. The database should maintain data integrity, but that is not a requirement (e.g., IntegriDB [79]).

3.6 Application Instance Lifecycle and Available APIs

Table 1 summarizes the API available to untrusted components, whose lifecycle is managed by the *Proxy*. Even if not shown in the figures for clarity, all containers have read-only access to a folder that contains their code. Non-folder containers also have read-write access to a folder that can be used to store user-specific settings, and per-folder containers get read-only access to it (ensuring folder non-interference). Per-folder instances cannot directly link back to a non-folder or cross-folder view since that could be used to exfiltrate information across folders. Instead, the *Proxy* intercepts requests to `/dats/quit` and redirects the client to the *cross-folder view* that initiated a transition to that *per-folder view*.

4. Implementation of the DATS System

Figure 4 shows all the components described in § 3 and how they interact, with an emphasis on DATS's TCB. It includes four *Storage Declassifiers* we wrote for MySQL,

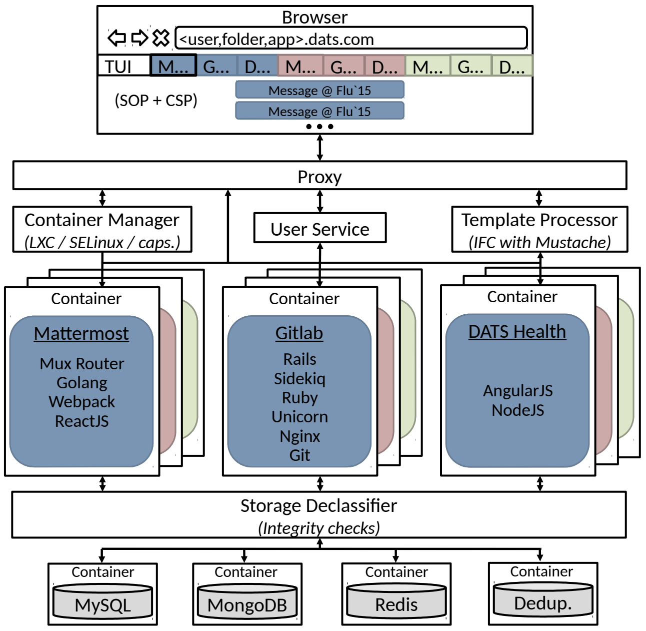


Figure 4: Implementation of DATS; colors denote untrusted code, and italics show the mechanisms used. All DATS components run in containers. The trusted Login and Backend applications are not show for brevity.

MongoDB, Redis and a custom deduplication backend. We wrote 13 K lines of TCB code in Python, Node.js and C. The TCB can be substantially reduced by running containers on security-oriented OSs like seL4 [43] or HiStar [77]. Instead of trusting the hypervisor, we could also use attestation and trusted boot [42, 54] to bootstrap DATS on a remote cloud.

MongoDB and Redis are not limited to `put/get`, but we find this sufficient for our diverse applications (see § 5). Also, app programmers can always use a per-folder instance.

The *Proxy* is a multi-process Flask [6] application running behind Apache, with MySQL as a persistent database, Redis [15] as a short-term database and to implement distributed locks, and Celery [2] to run cross-folder requests in parallel. API operations are authorized by inspecting the requestor's IP; this is the simplest mechanism that can monitor folder access grants (each container has a unique IP).

The *Proxy* authorizes access to container sub-domains using a session cookie set by the trusted *Login* application, and also performs a simple form of caching of the per-folder JSON results used to feed `templates`. Its API is not described due to space constraints, but follows concepts similar to existing web caching technologies.

4.1 Container Backends

Off-the-Shelf Container Techniques. LinuX Containers (LXC) [10] provide lightweight OS virtualization; we did not try conventional VMs since LXC is more efficient [38, 57]. Containers are created as a "clone" of a base file-system with additional folders overlaid on top using AUFS [1], and isolated at the network level using `iptables`.

SELinux [48, 61] implements a flexible and fine-grained MAC for Linux. Each container gets a virtual network interface and a binary policy module with the necessary labels to access files and network ports.

HW-Capabilities Based Thread Containers. When a *Proxy* thread "enters" a container (executes a per-folder request using the `app` code), we prevent it from sharing memory with other threads using the CODOMs capability architecture [66]. The `app` code is loaded into a separate CODOMs protection domain that isolated threads have read-execute access to, each thread gets a read-only capability pointing to its input request, and two read-write capabilities that point to their private stack and heap pool, respectively. To prevent sharing through the file system, we added a Linux system call (used by the *Proxy*) that privatizes the thread's file descriptor table and SELinux label. When a thread returns from its `app` call, the *Proxy* restores the per-process file descriptor table and SELinux tag, and frees the thread's private memory pages (the private heap and stack).

4.2 Enforcing Client-Side Non-Interference (Extending Containers to Client-Side)

DATS uses the *Template Processor* and standardized browser security controls (configured by the *Proxy* on the headers of the responses it serves to the client) to avoid:

1. Leaks through request URLs/contents:
   (a) The trusted *Template Processor* knows about HTML structure and the semantics of tags that trigger browser requests (e.g., links and forms) and include untrusted client-side code (which is forbidden in cross-folder views). Each tag includes information from a single `DATS.results` block, ensuring that request URLs and contents have information from a single folder (i.e., a `<form>` block cannot span accross multiple `DATS.results` blocks).

   Also, all URLs inside a `DATS.results` block are generated with a `DATS.enter` block — ensuring the URLs cannot leak per-folder data through a request to any other container.

   (b) DATS sets the browser's CSP to only allow trusted client-side code in cross-folder views (otherwise untrusted code could observe and modify a view to leak information).
2. Leaks through request headers:
   (a) DATS' *Proxy* drops any cookie set by untrusted app responses for any origin that is not assigned to that app instance/container (i.e., drops cookies for parent domains).

   The browser's CSP (and the newer "Referrer-Policy") ensures no data is leaked through the "Referrer" header (i.e., each view type is on a different origin).
3. Leaks to third-party domains:
   (a) The browser's SOP forbids client-side code to access other non-origin domains.
4. Leaks through URL guessing:
   (a) Randomizing container sub-domains ensures apps cannot exfiltrate information by generating a link or redirect to an arbitrary sub-domain/container.

5. Programmability Evaluation

Developers can write or port applications to DATS without any security expertise, since authentication and access control are offloaded to DATS. Instead, they need only take functionality-based decisions: establishing a minimum unit of sharing (§ 3.2), writing `templates` for cross-folder views (§ 3.4), and deciding what `storage` components to use (if any; § 3.5). We evaluate programmability in DATS by porting 4 open-source applications and writing 4 applications from scratch, all summarized in Table 2. They cover diverse use-cases, like messaging, voice/video chats, document editing, software development, and electronic medical records.

We describe Mattermost and DATS Health in detail since they stress user management and `template` creation tasks.

5.1 Mattermost

Mattermost (version 3.7) is a messaging application written using Go (server side), React [14] (client side), and MySQL for storage. It has the concepts of teams (user groups) and channels to organize conversations based on similar topics.

We selected channels as the minimum unit of sharing, since they are a natural choice. A user can thus choose to create multiple teams and channels on a single folder, or use only one channel per folder from the DATS trusted user interface (TUI). The developer only provides the functionality of channels and teams as a way to organize data without tying it to access control (see § 3.2).

5.1.1 Cross-folder views:

Mattermost allows users to create "teams" but not view a single unified "inbox" of messages across all teams (the user has to enter a team and channel to view messages). We wrote a new landing page with a `template` (140 lines of HTML, application tags, and DATS tags) that displays a timeline of recent activity across all teams. The `template` includes application-specific tags (e.g., `Teams`) embedded within the `DATS.results` block in order to show the data pulled from each folder. The per-folder request returns a JSON list with the teams stored on that folder and the three most recent messages for each team. The `template` also uses the `DATS.enter` tag on each message to link it to a per-folder `app` instance that will show the full conversation.

Application	**Framework**	**DATS Effort (LOC)**	**Total TCB (kLOC)**	**CVEs**	**CVEs w/ Leak**	**Description**
Gitlab	Rails	+386 -50 *A:* +36 / *T:* +100 / *P:* +250	145.2 / 1,352	13 / 83	9 / 45	Repository management [Ported] Minimum unit: project
Mattermost	Mux	+443 -800 *A:* +25 / *T:* +100 / *P:* +318	191.4 / 1,219	41 / 54	20 / 29	IRC chat [Ported] Minimum unit: channel
Hacker Slides*	Flask	+150 -100 *A:* +13 / *T:* +124 / *P:* +23	16.9 / 1,482	- / 40	- / 19	Slide presentation and editing [Ported] Minimum unit: presentation
Let's chat*	MEAN	+260 -500 *A:* +21 / *T:* +100 / *P:* +139	23.5 / 1,891	- / 25	- / 4	Real-time messaging using websockets [17] [Ported] Minimum unit: communication channel
DATS Health*	MEAN	+101 *A:* +36 / *T:* +30 / *P:* +35	5.6 / 1,284	- / 25	- / 4	Doctor-patient appt. management w/ calendar [New] Minimum unit: appointment
DATS Coding*	Django	+130 *A:* +15 / *T:* +100 / *P:* +15	1.2 / 1,545	- / 254	- / 169	IDE for code editing and terminal for compilation [New] Minimum unit: code files
DATS PDF*	Node.js	+90 *A:* +15 / *T:* +25 / *P:* +50	1.1 / 1,071	- / 25	- / 4	PDF document viewer [New] Minimum unit: PDF document
DATS Image*	MEAN	+90 *A:* +10 / *T:* +50 / *P:* +30	1.1 / 1,131	- / 25	- / 4	Upload and display images [New] Minimum unit: image file

Table 2: DATS Effort: lines of code (LOC) to make an application DATS-aware. Total TCB: LOC of just application code (left) and including all its dependencies (right). CVEs: public CVEs since 2013 [3] for the application (left) and entire application stack (right). CVE w/ Leak: CVEs which contain information disclosures for the application (left) and entire stack (right). With DATS, the entire application SW stack is outside the TCB and folder interference is systematically eliminated.
(*): CVE information not available. (*A*): Application code. (*T*): `template` contents. (*P*): Use of DATS's APIs.

5.1.2 User authentication and access control:

We removed user authentication from Mattermost, and instead use the *User Service* to have an up-to-date per-folder "Users" table (we added endpoints `/dats/start` and `/dats/update`; see § 3.6). This is crucial to Mattermost: e.g., it displays a list of recent activity for a particular team by issuing an SQL join query with its "Users", "Team-Members" and "Status" tables.

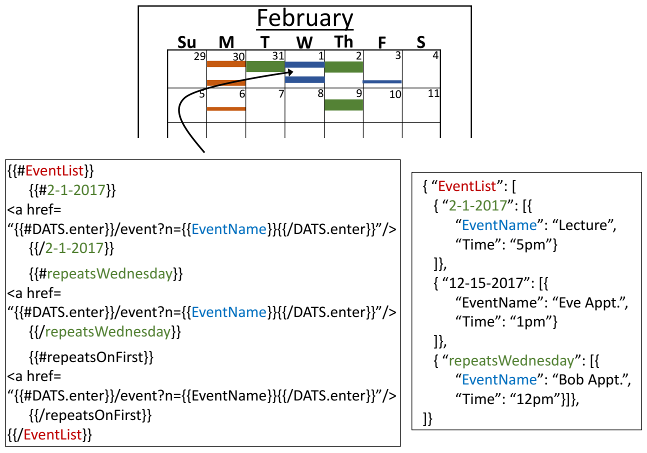
5.2 DATS Health

DATS Health is an MVC application written from scratch by undergrandate students. It allows doctors and patients to manage medical records (each medical visit is an "encounter") and allows users to schedule appointments through three different calendar views (month, week, and day). Encounters were selected as the minimum unit of sharing.

Unlike Mattermost, calendar views in DATS Health require a more extensive use of `templates` (endpoints `/month?<value>`, `/week?<value>` and `/day?<value>`).

Figure 5 shows an example of how we constructed the (non-folder) `template` and the (per-folder) JSON results for the `/month` endpoint (the other two work in a similar way). The `template` contains an empty grid with non-event information (e.g., month name and dates for each cell). Each cell is a `DATS.results` block that contains application-defined tags that match the corresponding JSON results.

The JSON results are pulled from a MongoDB database and returned when a per-folder app instance receives a request to the same `/month` endpoint. Each event is embedded in the `EventList` array in one of three types of tags. The first one is a list of all events for the actual date



```
{{#EventList}}
    {{#2-1-2017}}
<a href=
“{{#DATS.enter}}/event?n={{EventName}}{{/DATS.enter}}”/>
    {{/2-1-2017}}
    {{#repeatsWednesday}}
<a href=
“{{#DATS.enter}}/event?n={{EventName}}{{/DATS.enter}}”/>
    {{/repeatsWednesday}}
    {{#repeatsOnFirst}}
<a href=
“{{#DATS.enter}}/event?n={{EventName}}{{/DATS.enter}}”/>
    {{/repeatsOnFirst}}
{{/EventList}}
```

```
{ “EventList”: [
  { “2-1-2017”: [{
      “EventName”: “Lecture”,
      “Time”: “5pm”}
  ]},
  { “12-15-2017”: [{
      “EventName”: “Eve Appt.”,
      “Time”: “1pm”}
  ]},
  { “repeatsWednesday”: [{
      “EventName”: “Bob Appt.”,
      “Time”: “12pm”}]},
]}
```

Figure 5: Snippet from the DATS Health calendar `template` page. `Template` tags (left) are matched with the per-folder JSON (right) and inflated to create the calendar information.

(e.g., `2-1-2017`). The other two tags are for weekly (e.g., `repeatsMonday`, `repeatsTuesday`) and monthly events (e.g., `repeatsOnTheFirst`, `repeatsOnTheSecond`).

The *Template Processor* goes through each `DATS.results` block, and there iterates across each per-folder result (`EventList`). Any event that contains a matching template tag (`2-1-2017`, `repeatsWednesday` and `repeatsOnTheFirst` in Figure 5) is then inserted into the template.

The `repeats...` tags are crucial to avoid producing large JSON results where an event must contain a key for every date it should be displayed on (up to 365 entries per event per year). Additionally, they allow for JSON result caching (see § 4) across different `templates`.

6. Security Evaluation

DATS mitigates all data-disclosure vulnerabilities by refactoring authorization and access control out of untrusted web applications (i.e., moving applications outside the TCB). Table 2 summarizes the total number of vulnerabilites found in the applications (and their third-party dependencies) that we have run in DATS. Unprevented vulnerabilities include broken functionality that does not produce information disclosures, such as unauthenticated team creation in Mattermost[3]. We will now discuss those found in Mattermost and Gitlab to understand their root causes and how DATS prevents them.

These vulnerabilities are typically a consequence of the large number of user input and services modern web applications must handle in order to provide useful functionality.

Mattermost has 41 different vulnerabilities disclosed since 2015 [11], 20 of which DATS would prevent. They include cross-site scripting, remote code execution, denial-of-service, message spoofing, and authentication bypassing. The majority stem from missed access-control checks and incorrect input sanitation. For example, a missed authorization check in `/api/v1/users/find_teams` allowed attackers to view the invite link for any team on the system, provided that they knew the email of any user on the team.

We also examined the 13 CVEs (9 preventable by DATS) disclosed for Gitlab and discovered similar vulnerabilites. For example, Gitlab improperly sanitzes public keys uploaded by users before operating with them, allowing attackers to execute arbitrary code on a Gitlab server [7].

DATS prevents these improper data disclosures because each application instance is confined to its own folder, and user authorization is centralized in the TCB. Vulnerabilities can appear on application code, but also on the large stacks and third-party code used on web applications. In total, 402 CVEs have been reported for the stacks of the applications found here, of which 237 would be prevented by DATS.

Interestingly, DATS is also able to contain app-layer denial of service (DOS) attacks; we can apply per-container resource limits (e.g., using `cgroups` in Linux) to contain attacks to a per-user and folder instance. For example, incorrect sanitization in large file uploads allowed for a server-side DOS attack in Mattermost, consuming too much memory on the application. Similar attacks can also be seen in the CVEs for the applications and stacks in Table 2.

7. Performance Evaluation

We built DATS using two container technologies, and show that HW-assisted containers improve performance. We find that DATS provides reasonable performance for the majority of application operations (per-folder requests) and good scalability for cross-folder operations (albeit with a fixed latency cost due to multi-execution § 7.1.2). For worst case operations (searching across all folders), a naive implementation introduces overheads of 70× (13 secs) while hardware assisted solutions brings this down to 1.47× (47 msecs).

All experiments use an Intel Core i7-4770 (4 cores 2-way SMT @ 3.40GHz), 12 GB DDR3, an Intel 82574L NIC (1 Gbit), a Seagate ST3500413AS disk (500 GB, 7200 rpm, 16 MB cache) and Ubuntu server 14.04 LTS.

7.1 Existing Container Technologies

7.1.1 Per-Folder Application Throughput and Latency

Container latency is encountered when creating containers, starting applications inside containers, and destroying containers. Starting an LXC and SELinux container and an application within a container ranges from about 1second (e.g., DATS Health) to tens of seconds (e.g., Mattermost). The Container Manager maintains a pool of free containers and prefetches them, hence none of these operations are on the critical path.

We measured the average throughput and latency for a per-folder request to DATS Health (§ 5.2) with varying number of clients – using low-overhead query to access a single encounter to focus on DATS's overheads. The baseline application (one worker process per client) is compared against a version running in DATS using either the LXC or SELinux backends, both with and without a shared MongoDB `storage` component.

We find that all experiments have a pareto-optimal point at 8 clients, with latency overhead of 3× (10 msec) and throughput overhead 66% (1000 req/sec). DATS's scaling is limited due to its *Proxy* component since it routes all requests to `app` containers – a dynamic reverse proxy (for client authentication and routing to the target container) would greatly improve latency, contacting the *Proxy* only when a cross-folder operation is triggered. Throughput would improve by moving reference counting (now serialized in Redis) to a distributed algorithm with batched lazy releases. Latency and throughput would also improve with a simple page caching layer.

7.1.2 Cross-Folder Application Throughput and Latency

We also measured the latency of a worst-case cross-folder request with an increasing number of folders (same configurations, disabling the JSON result caching described in § 4).

DATS shows a base overhead of 30× (from 0.01 to 0.3 sec latency) due to container-agnostic factors: **(1)** the latency overheads in § 7.1.1; **(2)** the additional redirects for cross-folder views (§ 3.4); and **(3)** a sub-optimal *Template Processor* using Python libraries. The folder count has a small linear increase in latency overheads (up to 1 sec for 50 folders). This is inherent to the multi-execution of existing containers (each per-folder request needs a separate pro-

[3] The data we collected for this evaluation can be found in `https://bitbucket.org/datsplatform/security-evaluation`.

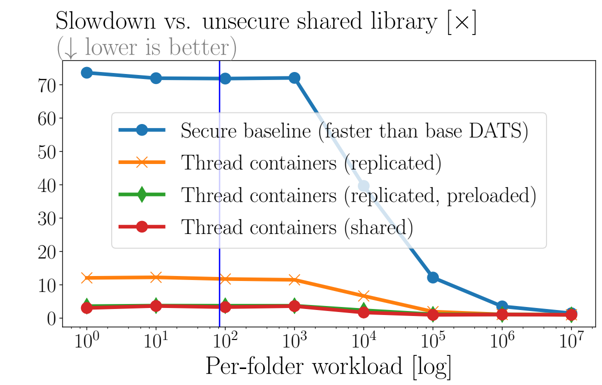


Figure 6: Performance of a cross-folder operation with hardware-accelerated thread containers, normalized to the performance of an unsecure shared library. The X-axis shows the processing work on each folder. The unsecure baseline goes from 47 to 13,721 msec, and generating a cross-folder result for § 7.1.2 corresponds to the thin vertical line.

cess), and `storage` consolidation shows a slight improvement on that.

DATS thus has reasonable folder scalability — container reuse and prefetching and running per-folder requests in parallel effectively hides costs — and the base overhead can be eliminated using known optimizations (§ 7.1.1) and a faster *Template Processor* implementation.

7.1.3 Cross-Folder Storage Declassifier

We also analyzed the latency overheads of using a *Storage Declassifier* with our deduplication `storage` (§ 4). The declassifier adds a 2× and 3× latency overhead for `get` and `put` operations, respectively. This is expected given our simple prototype (150 LOC of Python), but shows ease of implementation. More efficiency is possible through data blocking and using Merkle trees [63].

7.2 Hardware-Accelerated Containers

Cross-folder operations are critical in DATS and existing container technologies bring intrinsic overheads: the multi-execution of per-folder processes [26] and copying data across them. We find that hardware-accelerated thread containers (§§ 3.3 and 4.1) help eliminate these overheads while even existing containerization technologies have negligable overheads for large application workloads.

Figure 6 shows the performance of a cross-folder operation with 800 folders using 8 threads.The X-axis shows the workload necessary to make overheads negligible — each request calculates the factorial of the X-axis number (stored in a per-folder file, opened and closed on each request). The Y-axis is normalized to the regular non-isolated threads. We ran the experiments natively, using the same methodology of previous studies with CODOMs [67][4].

[4] Capability operations are emulated using regular memory accesses, and hardware registers are emulated using thread-local variables.

Secure baseline shows the most efficient existing container technology with perfect *Proxy* scalability and SELinux label prefetching, where `app` and *Proxy* communicate through a pipe. Slowdowns range from more than 70x to 1.47x. Importantly, real applications see the highest overhead; the workload in § 7.1.2 corresponds to the vertical line in Figure 6.

Thread container (replicated) shows the costs of running a thread container with a regular `app` instance; the *Proxy* uses `dlopen`/`dlclose` when entering/exiting the container, respectively. The slowdowns are drastically reduced to 12.25x–1.01x because of the less expensive thread containers, and by avoiding copies between the *Proxy* and the `app`.

Thread container (shared) shows the best results by sharing a single read-only `app` instance across thread containers. It removes the overheads of multi-execution and managing per-container library instances, reducing slowdowns to 3.65x–0.98x, but global writable data must be replaced with dynamic allocations. Finally, ***Thread container (replicated, preloaded)*** decouples the multi-execution overheads from library management by optimistically preloading all `app` instances and never unloading them. Slowdowns are 3.76x–1.01x, since multi-execution still takes a toll on performance; having multiple virtual copies of a library does not allow optimal sharing of read-only and micro-architectural state.

Thus, hardware-acceleration **reduces overheads from 70x to 3.65x** (worst) to within measurement noise (best).

8. Related work

Hardware-assisted reference monitors (Intel MPX and SGX) [22, 41], capability-based [43, 66, 69, 70], information flow tracking [31, 44, 77], and other systems [73] can be used to implement containers. On the client side, data-confined sandboxes and IFC [21, 62] can improve DATS's security guarantees and `template` flexibility. Verifiable SQL queries such as IntegriDB [79] can be integrated as *Storage Declassifiers*, moving SQL DBs outside the TCB.

BStore [28] and CryptoJails [60], propose a trusted storage layer enforces policies on the users' behalf. DATS goes further by enforcing MAC over arbitrary MVC applications. DATS is reminiscent of CLAMP [53], which refactors two security-critical pieces: a query restrictor (which guards a database) and a dispatcher (which authenticates the user). CLAMP does not enable controlled sharing of user data with untrusted code. DATS's cross-folder views are also related to trusted windowing systems [36, 58], but these cannot present cross-folder views as part of a single application.

Hails [39] lets developers associate access control policies to the *model*, while replicating *view* and *controller* components for each security label; applications are written in Haskell, developers have to label their models (or platform developers have to understand each application's models), and cross-label (i.e., cross-folder) views or storage backends like deduplication will require extending Hails with our robust declassifiers. Jeeves [74] provides similar dynamic IFC features, requiring developers to understand information flow policies and limiting choices to supported languages. Secure multi-execution [33], Maxoid [71] and Earp [72] provide non-interferent data containers, but target different use-cases and therefore lack support for cross-folder views and storage. The former is for client-side code (it spawns one process per container, although faceted execution can relax that [24]). Maxoid and Earp instead provide a confined view of resources to ensure processes in Android do not leak information. Self-protecting data [29] uses hardware IFC and a security policy component that does not protect against implicit information flows. Radiatus [30] runs applications inside *user containers* – collaborative containers will require DATS-like `templates` and *Storage Declassifier*. Radiatus's capability protocol requires applications to be ported to access per-user data, similar to our API. It also trusts third-party storage backends to maintain data separation between the per-user privacy domains. πBox [45] containerizes applications and uses a differential privacy declassifier for ad-impressions – we containerize data.

The IFC approaches above implement containers in different ways. DATS's declassification is applicable to all – it will safely remove declassifiable functionality from applications' TCB – while IFC can be applied to DATS's TCB itself.

9. Conclusions

DATS places applications out of the TCB while presenting a familiar security-oblivious programming model to developers. This is a major departure from the current application-centric security model, but is only one step forward towards placing users in control over their data. DATS motivates focusing language support for non-interference specifically towards template languages and architecture/OS support for multi-execution. In its current state, DATS is evidence that user privacy and developer productivity are not a zero-sum game.

Acknowledgements

This research was supported by the Israeli Ministry of Science, Technology and Space, and a collaboration grant through HiPEAC (European Union's H2020 research and innovation programme under grant agreement 687698). Lluís Vilanova was supported by an Andrew and Erna Finci Viterbi Post-Doctoral Fellowship. Charalampos Papamanthou was supported in part by NSF grants #1652259 and #1514261.

References

[1] Advanced multi layered unification filesystem (AUFS). http://aufs.sourceforge.net.

[2] Celery. http://www.celeryproject.org.

[3] CVE Details. http://www.cvedetails.com/vulnerability-list.

[4] Docker. http://docker.com.

[5] Recent zero-day exploits. https://www.fireeye.com/current-threats/recent-zero-day-attacks.html.

[6] Flask. http://flask.pocoo.org.

[7] Gitlab security vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-13074/Gitlab.html.

[8] 20 famous websites vulnerable to cross site scripting (XSS) attack. http://thehackernews.com/2011/09/20-famous-websites-vulnerable-to-cross.html.

[9] HITRUST alliance. https://hitrustalliance.net.

[10] Linux Containers. http://linuxcontainers.org.

[11] Mattermost security updates. https://about.mattermost.com/security-updates/.

[12] Mustache. http://mustache.github.io.

[13] OWASP top ten project. https://www.owasp.org/index.php/OWASP_Top_Ten_Project.

[14] React - a JavaScript library for building user interfaces. http://facebook.github.io/react.

[15] Redis. http://redis.io.

[16] Comparison of web template engines. https://en.wikipedia.org/wiki/Comparison_of_web_template_engines (accessed Aug 2017).

[17] RFC 6455 - the websocket protocol. https://tools.ietf.org/html/rfc6455.

[18] Wikipedia - SQL Injection. https://en.wikipedia.org/wiki/SQL_injection#Examples.

[19] The security flaws at the heart of the Panama Papers. http://www.wired.co.uk/article/panama-papers-mossack-fonseca-website-security-problems.

[20] *WSGI*. http://wsgi.org.

[21] D. Akhawe, F. Li, W. He, P. Saxena, and D. Song. Data-confined HTML5 applications. In *Computer Security – ESORICS*, 2013.

[22] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Symp. on Operating Systems Design and Implementation (OSDI)*, Nov 2016.

[23] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *ACM Conf. on Computer & Communications Security (CCS)*, Oct 2011.

[24] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. Jan 2012.

[25] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Symp. on Operating Systems Design and Implementation (OSDI)*, Oct 2012.

[26] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong. Multi-execution: Multicore caching for data-

similar executions. In *Intl. Symp. on Computer Architecture (ISCA)*, Jun 2009.

[27] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, Mar 2008.

[28] R. Chandra, P. Gupta, and N. Zeldovich. Separating web applications from user data storage with BStore. In *USENIX Conference on Web Application Development*, Jun 2010.

[29] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee. A software-hardware architecture for self-protecting data. In *ACM Conf. on Computer & Communications Security (CCS)*, Oct 2012.

[30] R. Cheng, W. Scott, P. Ellenbogen, J. Howell, F. Roesner, A. Krishnamurthy, and T. Anderson. Radiatus: Strong user isolation for scalable web applications. In *ACM Symp. on Cloud Computing (SoCC)*, Oct 2016.

[31] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Intl. Symp. on Computer Architecture (ISCA)*, May 2007.

[32] J. B. Dennis and E. C. V. Horn. Programming semantics for multiprogrammed computations. *Comm. ACM*, Mar 1966.

[33] D. Devriese and F. Piersens. Noninterference through secure multi-execution. In *IEEE Symp. on Security and Privacy*, May 2010.

[34] C. Dwork. Differential privacy. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2006.

[35] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and Event Processes in the Asbestos Operating System. ACM, 2005.

[36] J. Epstein, J. McHugh, R. Pascale, H. Ormau, G. Benson, C. Martin, A. Marmor-Squires, B. Danner, and M. Branstad. A prototype B3 trusted X Window system. In *Annual Computer Security Applications Conference (ACSAC)*, Dec 1991.

[37] A. J. Feldman, A. Blankstein, M. J. Freedman, and E. W. Felten. Social networking with frientegrity: privacy and integrity with an untrusted provider. In *USENIX Security Symposium*, Security'12, pages 31–31, 2012.

[38] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. Technical report, IBM Research Division, Jul 2014.

[39] D. B. Giffin, A. Levy, D. Stefan, D. Terei, J. Mitchell, D. Mazières, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *Symp. on Operating Systems Design and Implementation (OSDI)*, Oct 2012.

[40] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, Apr 1982.

[41] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Symp. on Operating Systems Design and Implementation (OSDI)*, Nov 2016.

[42] *Intel Software Guard Extensions Programming Reference*. Intel, Oct 2014.

[43] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *ACM Symp. on Operating Systems Principles (SOSP)*, Oct 2009.

[44] M. Krohn. *Information Flow Control for Secure Web Sites*. PhD thesis, MIT, 2008.

[45] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov. πBox: A platform for privacy-preserving apps. In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, Apr 2013.

[46] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[47] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *Intl. Conf. on Data Engineering (ICDE)*, Apr 2007.

[48] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *USENIX Annual Technical Conf. (ATC)*, 2001.

[49] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM Trans. Knowl. Discov. Data*, Mar 2007.

[50] F. McSherry. Privacy integrated queries. In *SIGMOD*, Jun 2009.

[51] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. E. Culler. GUPT: Privacy preserving data analysis made easy. In *SIGMOD*, May 2012.

[52] A. Nadkarni, B. Andow, W. Enck, and S. Jha. Practical DIFC enforcement on Android. In *USENIX Security Symposium*, Aug 2016.

[53] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. Clamp: Practical prevention of large-scale data leaks. In *IEEE Symp. on Security and Privacy*, May 2009.

[54] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *IEEE Symp. on Security and Privacy*, 2010.

[55] V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *SIGMOD*, Jun 2010.

[56] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, Apr 2010.

[57] B. Russell. KVM and Docker LXC Benchmarking with OpenStack. `http://bodenr.blogspot.com/2014/05/kvm-and-docker-lxc-benchmarking-with.html`.

[58] J. Rutkowska and R. Wojtczuk. Qubes OS architecture. Technical report, Invisible Things Lab, Jan 2010.

[59] A. Sabelfeld, A. C., and Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, Jan 2003.

[60] M. Sherr and M. Blaze. Application containers without virtual machines. In *ACM workshop on Virtual machine security*, Nov 2009.

[61] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux Security Module. Technical report, NAI Labs, Dec 2001.

[62] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières. Protecting users by confining javascript with COWL. In *Symp. on Operating Systems Design and Implementation (OSDI)*, Oct 2014.

[63] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea. Iris: a scalable cloud file system with efficient integrity checks. In *Annual Computer Security Applications Conference (ACSAC)*, 2010.

[64] L. Sweeney. k-anonimity: A model for protecting privacy. *International Journal on Uncertaint, Fuzziness and Knowledge-based Systems*, 2002.

[65] B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine grained timers in Xen (short paper). In T. Ristenpart and C. Cachin, editors, *Proceedings of CCSW 2011*. ACM Press, Oct 2011.

[66] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. CODOMs: Protecting software with code-centric memory domains. In *Intl. Symp. on Computer Architecture (ISCA)*, Jun 2014.

[67] L. Vilanova, M. Jordà, N. Navarro, Y. Etsion, and M. Valero. Direct inter-process communication (dipc): Repurposing the codoms architecture to accelerate ipc. In *European Conference on Computer Systems (EuroSys)*, Apr 2017.

[68] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, Sep 2009.

[69] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: practical capabilities for UNIX. In *USENIX Security Symposium*, 2010.

[70] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symp. on Security and Privacy*, May 2015.

[71] Y. Xu and E. Witchel. Maxoid: Transparently confining mobile applications with custom views of state. In *European Conference on Computer Systems (EuroSys)*, Apr 2015.

[72] Y. Xu and E. Witchel. Earp: Principled storage, sharing, and protection for mobile apps. In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, Mar 2016.

[73] Y. Xu, A. M. Dunn, O. S. Hofmann, M. Z. Lee, S. A. Mehdi, and E. Witchel. Application-defined decentralized access control. In *USENIX Annual Technical Conf. (ATC)*, Jun 2014.

[74] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. Jan 2012.

[75] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *ACM Symp. on Operating Systems Principles (SOSP)*, Oct 2009.

[76] S. Zdancewic and A. C. Myers. Robust declassification. In *IEEE Computer Security Foundations Workshop*, 2001.

[77] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *Symp. on Operating Systems Design and Implementation (OSDI)*, 2006.

[78] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *Symp. on Operating Systems Design and Implementation (OSDI)*, Dec 2008.

[79] Y. Zhang, J. Katz, and C. Papamanthou. Integridb: Verifiable sql for outsourced databases. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1480–1491, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813711. URL http://doi.acm.org/10.1145/2810103.2813711.