Vaportail: A Platform for Personal Data Applications

by

Kalan W. MacRow

B.Sc. Computer Science, University of British Columbia, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

August 2017

Abstract

Vaportrail is a privacy-preserving platform for personal data and applications. It allows users to archive their personal data and safely expose it to untrusted third-party applications. As a trusted hub for data sources like email, social updates, location data, and health metrics it enables new types of applications that combine several sensitive personal data streams. Through carefully designed isolation mechanisms, the platform prevents applications from exfiltrating data and unburdens developers of the fiduciary responsibility associated with handling personal data. Vaportrail provides an open package format and APIs for building service connectors that ingest data from external services, as well as a robust browser-based sandbox that mediates application access to sensitive data.

Lay Summary

Vaportrail is a software system that allows users to import their data from websites, social networks, smart devices, and other services. It provides a safe platform for exploring and leveraging personal data using third-party applications.

Preface

This dissertation is original, unpublished, independent work by the author, Kalan W. MacRow. It was conducted in the Networks, Systems and Security Laboratory at the University of British Columbia, Point Grey campus.

Table of Contents

List of Tables

List of Figures

List of Programs

Glossary

RDBMS Relational Database Management System

TCB Trusted Computing Base

ETL Extract Transform Load

DOM Document Object Model

CSS Cascading Style Sheets

AMI Amazon Machine Image

UUID Universally Unique Identifier

CSP Content Security Policy, A browser technology to mitigate cross site scripting and data injection attacks.

Acknowledgments

I would like to thank my supervisor, Andy Warfield, for his limitless patience and encouragement to complete this project. Also, Bill Aiello for the many conversations that have shaped this work, and my perspective on online privacy. I would like to thank my family, and Sylvanna, for their constant support, and my friends and fellow NSS students for their guidance over the years. Finally, I would like to thank UBC Graduate and Postdoctoral Studies for their financial support.

Chapter 1

Introduction

A man cannot be comfortable without his own approval.
— Mark Twain

Social networking and cloud-based services have ushered in an era of unprecedented data creation, sharing, and interaction[12]. Apps and websites encourage us to create, connect and share an increasingly intimate portrait of our daily lives with everyone around us. We generate a continuous stream of personal content in the form of pictures, videos, emails, documents, comments, likes, purchases and GPS location data. Behind each page impression, click, and scroll event is a trail of personal data. Detailed logs of every interaction are captured and stored, creating a valuable, high-resolution history of what we view, like, dislike and search for.

With the advent of Internet of Things (IoT) a growing array of "smart" devices, including everyday appliances and mundane household objects, are passively (and in many cases *actively*[23][2][21]) listening: contributing enormous volumes of personal data to our *vaportrail* in the cloud. In this thesis we present Vaportrail, a self-contained and self-hosted appliance that enables users to leverage the data they generate. Vaportrail has a plugin architecture that allows service connectors to continuously load and store personal data from external services. To leverage the data, it provides a browser-based sandbox in which untrusted third-party applications can operate without the ability to exfiltrate data over the network.

1.1 Deployment model

Vaportrail is packaged as a virtual machine image (or virtual appliance) that can be hosted on any public cloud or private infrastructure. All operational costs, including compute and storage, are the responsibility of the user. There is no central Vaportrail service: the appliance does not "phone home" or have dependencies on any external services. Vaportrail is an entirely self-contained device. Although this model imposes some cost and operational complexity beyond familiar cloud services, it ensures that costs are explicit and mitigates the need for an exploitative business model. The continued growth and maturity of public cloud infrastructure has brought the cost of non-trivial compute and storage resources well within the means of the hobbyist[22][5][14]. We believe that for privacy conscious users, the control and transparency afforded by operating the appliance is preferable to other deployment options.

1.2 Personal data warehousing

Inexpensive cloud storage and the expected (business) value of data has made collecting and warehousing it, regardless of its immediate utility, standard practice for Software as a Service (SaaS) providers[12]. We take inspiration from modern consumer scale data warehousing to propose a form of personal scale data warehousing. Instead of archiving raw data to flat files or coercing various schemas into a single traditional Relational Database Management System (RDBMS), we combine several specialized data stores to support analytical queries across a range of structured and unstructured personal data. The platform organizes the data stores under a unified namespace and API.

1.3 Service connectors

Service connectors leverage platform APIs and a flexible execution environment to authenticate with external services and load data into Vaportrail. The connectors write to schemas that the user has explicitly granted them access to at install time. Service connector developers can choose from various storage interfaces to suit the nature of the data and or the expected access pattern. A Gmail[17] connector might

store attachments as binary objects in `gmail.objects.inbox.attachments`, and messages as indexed text in `gmail.nosql.inbox.messages`.

1.4 Application sandbox

Applications are the primary way users interact with their Vaportrail. A browser-based sandbox provides a safe execution environment for untrusted applications, restricting their access to data and preventing them from exfiltrating it over the network. Applications can use a rich set of platform APIs to query data, render GUIs, trigger platform-mediated sharing and save their state. There is a substantial amount of related work in JavaScript sandboxing, including AdSafe[1], Treehouse[44], JS.JS[51] and others[52][39][38]. We build on this work to create a robust application monitor that works in any modern web browser. In contrast to related work, we have made trade offs in favour of security and reliability over performance and backwards compatibility for existing applications.

1.5 A new class of applications

We hope to enable a new class of applications characterised by combining personal data streams that were previously siloed and or too sensitive to share with untrusted third-parties. Often we suffer from data lock-in: our data is trapped in the platform in which it was created, limiting what we can do with it. When we can export it, our data is often too sensitive to expose to any except the most credible, trustworthy third-parties. Trust and credibility are rightfully difficult to attain, but this limits the ability of independent developers to build applications that consume personal data. With Vaportrail, we unburden the developer of traditional fiduciary responsibility, and the user of the fear of their privacy being compromised, shifting trust onto the platform and enabling a new application design space.

1.6 A simple trust model

With sensitive personal data at stake, a straightforward trust model is critical. To that end, we state Vaportrail's trust model here:
Trust the platform and infrastructure, but not applications or service connectors.

We expect the number of service connectors to be small compared to the number of applications, and rely on the community and trusted brands to endorse them. In practice, an acceptable level of privacy can be achieved by deploying Vaportrail on a reputable public cloud and using service connectors that have been vetted not to interfere with the services they integrate with.

1.7 Sharing with intent

Vaportrail supports a limited form of sharing via platform-mediated intents[35]. We provide an API that applications can use to signal that an item (e.g. an image or text) is shareable. The platform then provides the user with the option to share the content using any service connector that has registered as a handler for the relevant data type. Intents provide a mechanism to publish results out of the platform while completely decoupling the application from the channel used to do so. Sharing is currently limited to basic data types, but we imagine it could be extended to support richer peer-to-peer sharing over WebRTC[36] or other protocols, with the platform as a trusted broker. There is some risk of exfiltration inherent in sharing, which we make explicit by requiring the user to grant an application permission to share.

1.8 Call for an open ecosystem

The success of Vaportrail depends on community adoption and a low-friction experience for developers and users. To foster the growth of an ecosystem we take inspiration for the packaging and distribution model of service connectors and applications from the web browser extension model. Service connectors and applications are packaged as self-contained archives including their source code and a manifest describing the permissions they require. Vaportrail is a decentralized platform: there is no official “app store”, only simple packages that are easily created, inspected and shared using familiar tools.
The aim of this thesis is to describe the design, prototype implementation and work related to Vaportrail. We also explore the application design space enabled by such a platform. The contributions of this thesis are threefold:

- We describe the design space for a modern, privacy-preserving platform for

personal data and summarize related commercial and academic work;

- We present Vaportrail, a prototype implementation and discuss the specific challenges and tradeoffs encountered in its development;
- We discuss several examples of fun and useful apps enabled by the platform, as well as consider the viability of Vaportrail's deployment model as an alternative to conventional SaaS.

At present the system is very much a research prototype, however enough of the core functionality has been implemented to evaluate the design of key mechanisms and APIs. In Chapter 2 we discuss the overall design of the system. In Chapter 3 we present the implementation. In Chapter 4 we summarize related work and in Chapter 5 we conclude.

Chapter 2

Design

Vaportrail is a self-contained platform for personal data and applications. It must be inexpensive for users to operate, simple to maintain and extensible throughout: from the appliance architecture, to the services it can ingest data from, and the core isolation mechanisms and APIs exposed to applications. In addition to these goals, we aim to prove that such a system can provide a modern and familiar web-based user interface. We argue that user experience is a critical factor in the adoption and success of a system like Vaportrail. We believe Vaportrail achieves these goals through infrastructure-agnostic packaging that gives users a choice of hosting environments, a plugin architecture for service connectors, modular design throughout, and careful adherence to the principle that our implementation should not break the familiar browser-based web experience. This chapter provides an overview of the design space and specific considerations and trade-offs that were made in the pursuit of these design objectives. We provide a summary of the design requirements at the end of the chapter.

2.1 A personal appliance

One of the guiding design goals for Vaportrail was that it should be a completely self-contained appliance that a user can operate with minimal cost and effort. The rise of Software as a Service has shifted user expectations toward a software delivery model in which the complexity (and cost) of providing sophisticated appli-

cations is largely centralized and hidden. Using a new application usually requires little more effort than pointing a web browser at a URL. While the centralised nature of this model presents several challenges for privacy and data ownership, it has a few attributes from a user experience perspective that are worth maintaining: applications are platform-independent JavaScript and HTML, execution is confined to a trusted sandbox (the browser), and application lifecycle is managed through familiar browser system primitives (tabs, history, bookmarks, etc.).

With Vaportrail we took the decision to make the costs of operating the system explicit to the user by default, to ensure that the design did not depend on the recovery of these costs through some form of business model. At the same time, we embrace the merits of the SaaS model for application delivery and aim to provide the "best of both worlds": a standalone appliance that is simple and inexpensive to deploy, with a familiar browser-based web application for interacting with services and applications. The virtual appliance should support deployment without special networking or storage configuration.

We acknowledge that because the appliance is self-hosted and maintained by the user, it should be capable of long-term operation without intervention. To achieve this, we use a modular, component-oriented architecture throughout the system. Platform components are decoupled and isolated from one another. When possible we choose proven software, protocols and isolation mechanisms over newer or less stable options.

2.2 A modern user experience

The requirement that Vaportrail provide a familiar, browser-based UI presented several challenges in balancing robust application sandboxing with a seamless user experience. Specifically, it was important that it be possible to run several Vaportrail applications concurrently and that the sandbox not require multiple browser tabs, page reloads, popups, or frames. We also set the requirement that Vaportrail should not depend on a custom browser extension to provide a trusted/privileged execution environment because extension semantics and APIs vary across platforms and browsers. By restricting the design to standard browser APIs and features that do not vary across platforms, Vaportrail is usable on a larger num-

ber of devices, safer by not requiring elevated privileges, and provides a modern single-page application experience.

2.3 Simple trust model

A simple and plainly stated trust model should be central to any device that consolidates highly personal data, and Vaportrail is no exception. It is important from a practical perspective, in that the system should be simple to use and reason about. It also gets to the heart of one of our broader goals with Vaportrail: to demonstrate that there are architectural patterns that support familiar application primitives without nebulous privacy implications, and that these patterns can be deployed today. It is a choice, and not a technical necessity, to build software without privacy controls. We state the platform's trust model here and discuss the design space and implications for each item below:

1. The user must trust the infrastructure that they choose to run their Vaportrail instance on

2. The user must trust the core platform base that mediates access to their personal data

3. The user must grant service connectors a) access to the external service(s) they integrate with, and b) platform-mediated write access to personal data schemas

2.3.1 Trust the infrastructure

As a self-hosted virtual appliance, the user is at liberty to run the software on any compatible infrastructure platform. This allows the user to balance cost and privacy according to their own priorities. An extremely cautious user might deploy the appliance on physical hardware that they own, while others might find a public cloud server, or even an instance hosted by someone else acceptable. There is a gradient of options between the two extremes. Whatever the case, the user must trust that the underlying hardware and software will not compromise their data. Verifying the low level safety and integrity of the environment is beyond the scope

of this project; we rely on the user to make a choice that is consistent with their priorities.

2.3.2 Trust the platform

The user must trust the platform core, which together with the underlying infrastructure forms the Trusted Computing Base (TCB). The platform core consists of the isolation mechanisms and APIs that mediate access to personal data by untrusted third-party applications and connectors. The platform will ensure that the user is made aware of the specific data schemas and permissions that applications and connectors request at install time. The fundamental guarantee of the platform is that it will ensure data cannot be exfiltrated except through explicit, user-approved, platform-mediated sharing mechanisms on a case-by-base basis.

2.3.3 Service connectors

The platform prevents service connectors from negatively impacting the stability of the appliance or the integrity of the data stores, however it cannot police their interactions with external services. The user must trust the connectors they install to operate in good faith with respect to the API access granted to them and the data that flows through them. We expect the number of service connectors to be small compared to the number of applications. Our hope is that trusted service providers will build their own connector integrations, and that the community will vet third-party connectors based on developer reputation and code reviews.

2.4 Personal data warehousing

Vaportrail can be related to traditional information infrastructure. The platform is conceptually similar to a data warehouse (the combination of an archive and an analytical platform) but for a single individual's data, instead of a large enterprise. Service connectors could be framed as Extract Transform Load (ETL) jobs, and applications as their analytical processing counterparts.

Although there are high-level comparisons to be drawn with traditional data warehousing, Vaportrail is different in a couple of important ways: a) it is tailored to a single individual's data and not a large enterprise, and b) it leverages a com-

bination of specialized modern data stores instead of either a monolithic RDBMS or "lake" of unstructured data. Reducing the scope to a single individual alleviates most performance and scalability concerns: the data is decidedly *small* by modern standards. The relatively small scale allows us to consider more convenient row and document-oriented databases that make storing and querying diverse schemas easier.

Leveraging multiple specialized data stores simplifies the transformations required in service connectors and makes a richer application-facing query API possible. An important trade-off versus using a single SQL-driven column store or similar, is that the query interface is more complex and applications will tend to be more coupled to the design choices of upstream connectors. A potentially significant limitation of this design is that schema migrations due to connector changes are more likely to break downstream applications. Mitigating this through a platform-managed schema migration mechanism was beyond the scope of the design at this prototype stage. Our current solution is to recommend versioned schema names.

Determining a set of data store interfaces that could meet the needs of a representative group of service connectors was a key design goal. The database architecture for an IMAP connector will be substantially different from one that imports videos from a social media service. As in other parts of the system, we aim for an extensible data layer so that unforeseen use cases can be met by adding storage and query interfaces as needed. Motivated by a thorough examination of the requirements of a few example connectors (discussed in Chapter 3) we arrived at a minimum set of storage interfaces for the initial prototype:

- SQL store providing efficient create, update, delete operations with column indexes for highly structured, row-oriented data
- Object store with support for large binary objects for use as a bulk repository for video, images, text blobs or other large files
- Memory Key/Value store for values that change frequently, caches, persistent data structures and publish-subscribe functionality
- Document or "nosql" store for less structured data with variable or frequently

changing schemas

We could imagine adding other potentially useful interfaces as future work (e.g. a graph database for capturing networks and dependency graphs or a time series database for event or other time-indexed data) but most use cases are reasonably well served by a combination of the initial set of data stores.

By providing an abstraction layer on top of the native data store interfaces, the platform can restrict connector and application access by putting authorization logic on the data path and mapping platform permissions down to data store-specific access control mechanisms, e.g. a database user with a specific set of GRANTs, in the case of an SQL store. System resources can be managed similarly by mapping high-level resource limits onto configuration options on the data stores.

2.5 An open and extensible platform

A guiding principle of Vaportrail is that it should be an open and extensible platform driven and owned by the community. It should invite extension and personalisation by individuals. This is more than a "feel good" design goal: for a platform to succeed, developers need to build connectors, applications, and extend the core platform capabilities. The existence of several SaaS products in this space suggests there is broad interest in leveraging personal data. Our aim is to lay the foundation for an open, privacy-preserving alternative to commercial offerings.

2.5.1 Service connectors

Service connectors provide the logic to connect to external services and load data into Vaportrail. Each connector is a self-contained application that runs in an isolated environment with limited system resources. The platform does not restrict the outgoing network traffic of connectors or impose any requirements on how they communicate with external services. Service connectors are third-party components, and the wide array of protocols and authentication mechanisms used by external services makes over-specifying their implementation impractical. Instead we assume connectors are well intentioned and focus on insulating the platform

from well meaning, but perhaps badly behaved or broken connectors. This is discussed in detail in Section YYY.

Connectors are shared and distributed using an open format inspired by browser extensions: essentially an archive containing a manifest and the connector code. The platform requires that the user accept the terms of the manifest during installation. This simple, open format is easy to read and write with existing tools, and is easily distributed with no strings attached. We leave building a central package manager or "app store" to the community.

2.5.2 Applications

Applications are untrusted third-party code that can be installed to provide fun and useful new functionality built on personal data streams. Much like service connectors, applications are distributed in a simple package format containing a manifest and code, however there are important differences between connectors and applications. Applications are developed in any language that can be compiled to JavaScript and can only use a very narrow platform API to query data, persist application state, and render user interface (UI) elements. The runtime environment for applications is extremely restricted compared to the standard browser Document Object Model (DOM) and JavaScript APIs. While our aim for the Vaportrail prototype was to demonstrate a sandbox that enables purpose-built applications, and not to support running existing applications, we believe the sandbox could be extended to support a complete virtual DOM and thus many popular libraries and frameworks. We focus on providing a low-friction format for sharing applications, and a robust, extensible browser-based sandbox.

2.5.3 Platform core

The applications we use to create and interact with the content we generate is a diverse and ever evolving ecosystem. For Vaportrail to remain relevant in such a dynamic environment, we acknowledge that even the core facilities of the platform should be open and extensible. To this end, we imagine Vaportrail as a kind of personal data "hub" that should invite extension to the way service connectors and applications interact with the platform, including the isolation mechanisms and proto-

cols governing those interactions. We design for extensibility throughout the platform core by using a modular architecture that relies on simple, well-documented API contracts between components, allowing entirely different implementations of core facilities (e.g. the application sandbox) to be used interchangeably.

2.5.4 Intent-based sharing

Personal data does not exist in a vacuum and we argue that a viable platform for leveraging personal data would not either. Vaportrail must reconcile the need for robust privacy with the ability to *intentionally* share useful application results with external services. The platform accomplishes this with Intents[35], a platform-mediated sharing mechanism that is widely deployed in mobile operating systems. An intent allows the platform to match an application provided *share* object (text, image, or other data) with a *target* capable of publishing the object to an external service. The content of each share is inspected by the user via a UI workflow before being handed off to the selected target to be published. This mechanism fully decouples the untrusted application from the sharing target and puts the user in control of an approval process, making each instance of data leaving the platform explicit. While sharing unavoidably creates the possibility of abuse by a malicious application, we believe Vaportrail mitigates the risk of such an exploit in two ways: 1) the application cannot know to which external service the data will be published and thus would need access to several possible targets in order to recover the data for exploitation, and 2) the user must explicitly grant applications permission to share. We believe that in practice the value of a sensible sharing mechanism will outweigh the potential risk of its exploitation.

2.6 Enabling personal data applications

Applications that operate on even a single personal data stream (e.g. banking transactions, or email) demand a degree of trust that is extremely difficult to attain. The developer or service provider requires a level of credibility that is scarcely achieved except by enormous corporations, entities in highly regulated sectors, or public institutions. Consequently, our most personal data is siloed within these trusted systems, making it extremely difficult for us to make our own copy, or to derive

further value from it without unacceptable privacy risks. Services increasingly provide APIs or data export features that solve one aspect of the problem–allowing us to make our own copy–but loading that data into any other software means placing significant trust in a third-party.

Vaportrail enables a new breed of personal data-consuming applications by providing a platform that prevents untrusted applications from exfiltrating data. This unburdens developers of the enormous cost and responsibility of being a legitimate fiduciary, and the user from the fear of having their data exploited. An exciting corollary of this is that multiple sensitive data streams can safely be exposed to applications that we previously would not have trusted with any personal data. We believe this opens up an entirely new design space for personal data applications, and suggest that there are several new application archetypes in this space.

Watchdogs

Watchdogs passively monitor your data streams and alert you when something good, bad or out of the ordinary happens. They might employ outlier detection or other Machine Learning techniques to build a model of your activity and monitor deviations from it.

Aggregators

Aggregators extend the features of external services (e.g. search) by combining multiple data streams under a single UI. Imagine searching your emails and credit card transactions in one place, perhaps plotted on a timeline alongside your heart rate data.

Lifestyle

Lifestyle applications provide useful hints, insights and reminders by looking at correlations in your data streams. Perhaps your location data and your fuel purchases suggest that you could be achieving better fuel economy.

Productivity

Productivity apps will leverage your personal data to actively help you achieve goals. They may suggest you purchase healthier food, find more time or cost effi-

cient modes of transportation for common routes, or combine health metrics with activity streams to determine your most productive times to work.

2.7 Summary

In this chapter we have reviewed the design objectives for Vaportrail and discussed the key features, capabilities and trade-offs involved in meeting them. The major design goals for Vaportrail include:

- Simple deployment and management for an individual
- A familiar, modern user experience
- A simple trust model
- An open and extensible platform

We believe the Vaportrail prototype achieves these goals. In Chapter 3 we discuss the implementation in detail and in Chapter 4 we survey related work.

Chapter 3

Implementation

In this section we discuss the prototype implementation of Vaportrail. The prototype is a self-contained, standalone virtual appliance that provides a fully managed platform for importing and archiving personal data. The platform serves as a privacy-preserving runtime and mediator between personal data and untrusted third-party applications.

The primary goal of the implementation is to provide a working proof of concept that meets the design criteria set out in Chapter 2 and to demonstrate that even a naive "reference" implementation of Vaportrail can be practical, useful and fun. In particular, we aim to show that with a reasonable level of effort we can build a platform with a modern user experience that is easy and inexpensive to operate while providing strong, practical privacy guarantees. This implementation serves to validate the basic architecture and appliance form factor, isolation mechanisms, programming interfaces and cost expectations. We acknowledge that the true test of a system like Vaportrail would depend on observing it in the hands of users and developers over an extended period of time. Due to time constraints, we limit the scope of our evaluation to discussion of how well the prototype meets the basic design goals, and aspects of the system that can be measured at the scale of a single deployment.

Vaportrail was inspired by several privacy-preserving systems that came before it and our implementation builds on the lessons and tradeoffs articulated by them. Specifically we credit Priv.io[52] with the basic idea that users might "bring

their own infrastructure" to an application ecosystem, and Treehouse[44] with the notion of repurposing WebWorkers as execution environments for untrusted code. DataBox[42] provided a philosophical framework that guided the design specification of Vaportrail as a personal data hub, and inspired the virtual "form factor" of Vaportrail as a personal appliance.

Although there is a considerable amount of prior work in this area, we felt the development a new system was justified by a few broad themes (limitations) in the related work:

1. requiring modified environments (custom operating system extensions, browsers, etc.) that are impractical for regular users,

2. depending on adoption by trusted service providers and or changes to their business models and,

3. code isolation mechanisms that are difficult to verify or reason about in practice

We argue that in the development of privacy-preserving systems, practical usability is tantamount to robust privacy controls. In many ways Vaportrail is only a novel arrangement of existing good ideas with a focus on making them usable today. We also take advantage of external factors like the declining costs, and increasing reliability, of public cloud infrastructure to make a case for individuals operating their own virtual appliances.

We believe the prototype implementation of Vaportrail successfully meets the design criteria and demonstrates that, with further refinement, the system could be a practical fiduciary and platform for our personal data. While we cannot extrapolate from our findings to conclude that a broader ecosystem around Vaportrail would be successful, we are hopeful that this initial work makes a compelling case for the possibility of wider adoption.

3.1 The appliance

In this section we discuss the packaging and core components of the platform. Vaportrail is designed as a personal appliance that can be operated by an individual with minimal technical or operational intervention.

3.1.1 System overview

The platform is packaged as a virtual machine image based on a standard Ubuntu Server 16.04 LTS release. The LTS designation guarantees that the release is focused on stability for enterprise applications and will receive long term support and upgrades. We chose Ubuntu largely for its familiarity, though a number of other Linux distributions might have served equally well. Although the machine image could easily be built for almost any virtualization environment, we chose to target the Amazon Machine Image (AMI) used by Amazon's Elastic Compute Cloud, for convenience. By providing a Vaportrail AMI users can create a virtual server running Vaportrail with resources (storage, CPU, RAM) tailored to their needs and budget. With a publicly visible IP attached to the server, the user can point their browser at the platform as soon as the appliance is running.

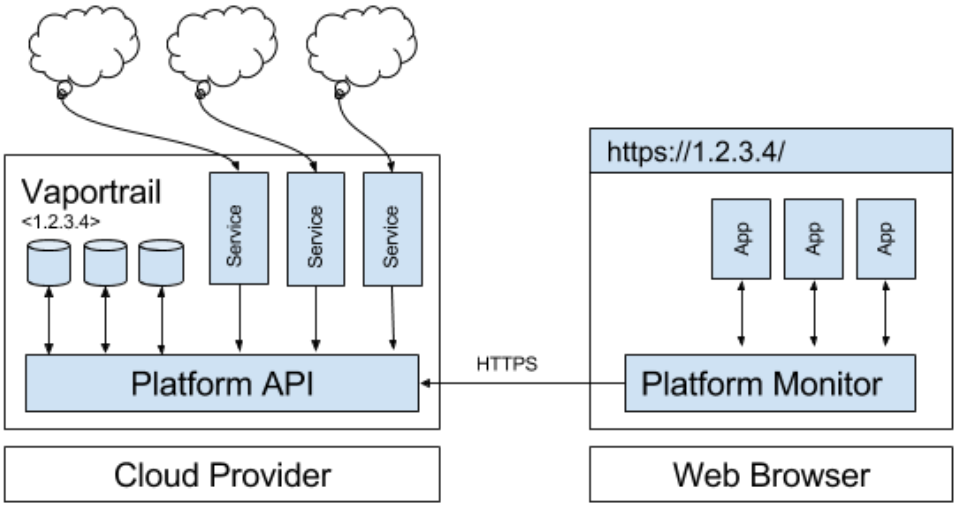

Figure 3.1: The Vaportrail appliance runs on a trusted infrastructure provider. Service connectors load and transform data which is then stored using the platform API. A web dashboard allows the user to safely expose their data to applications.

At a high level, the platform implementation is composed to two pieces: the *backend* (components that run directly on the virtual machine) and the *frontend*

(components that run in the browser). Backend components run in Linux containers managed by Docker[48] while the frontend components are served over secure HTTP to the users browser and executed client side. Running backend components (databases, service connectors, platform API servers, etc.) in containers affords us a) fine grained and dynamic control over their isolation from one another b) the ability to limit the system resources (CPU, memory) that any one component can consume. Docker images provide us with an open, versionable and familiar format for packaging and distributing Vaportrail components. By running the components in isolated containers and having them communicate via (generally narrow) API contracts, we achieve a highly modular design that facilitates testing, upgrades and even wholesale replacement of component implementations if the need arises. In order to orchestrate the various component containers, we run a special platform service in a privileged mode that allows it to create, configure, stop and start other containers. In keeping with the *principle of least privilege*, even the platform service does not have full control of the system, only a limited set of capabilities necessary to create containers at or below its own privilege level. The platform dashboard that the user interacts with is served by a single web application (`vaportrail-wsgi`) that both serves the static files (HTML, JavaScript and CSS) and acts as a RESTful API server providing endpoints for authentication and all other platform operations: installing, updating, removing applications and connectors, selecting data and persisting state. The frontend application is similarly modular: a privileged monitor establishes a session with the backend via the API and orchestrates the execution of sandboxed applications in isolated WebWorker containers. The application sandbox traps platform API calls and forwards them to the monitor, which authorizes the operation and makes the necessary calls into the backend before returning control (and any results) back to the sandboxed application. We achieve similar architectural benefits as in the backend by imposing a strict separation of concerns within the frontend implementation: the monitor, sandbox and API communicate only over asynchronous message-based interfaces allowing them to evolve independently and for different implementations to be swapped in and out. Separation of components via an asynchronous interface affords a particularly interesting opportunity for future work in which we could imagine migrating running applications to the server (i.e. by migrating the

sandbox) before the browser window is destroyed.

3.1.2 Threat model and security

Exposing any service on the internet is inherently *risky*, especially when that service is a box containing all of your most sensitive, personal data. We take a number of precautions in the Vaportrail architecture to minimize the network exposed surface of the platform by locking down all but the necessary ports, using random ports when a component needs to expose a service (e.g. to complete a third-party authentication workflow in a service connector), rate-limiting API endpoints, and applying the principle of least privilege throughout the system. We also employ strong password-based authentication over HTTPS and use techniques to prevent Cross Site Request Forgery (CSRF), Cross Site Scripting (XSS) and Phishing by malicious third-party applications (discussed in detail in Section 3.3). Beyond these precautions, the user has a considerable degree of control (and responsibility) to secure their appliance on the network. They might whitelist only a range of IPs or MAC addresses from which they intend to access their appliance. They might employ circuit-breaking middleware between the appliance and the internet to protect against denial of service attacks or to provide online traffic analysis and alerting. Automating the provisioning of a more sophisticated, production-grade security stack was beyond the scope of our initial prototype, but we could imagine providing a more comprehensive deployment template via Amazons CloudFormation[8] (or similar) as future work.

3.1.3 Appliance lifecycle and costs

It is important to consider the entire lifecycle of the appliance in terms of maintenance and costs. One of the advantages of centralized SaaS is that the costs and maintenance are entirely absorbed by the service provider. With Vaportrail, we exploit the fact that cloud computing resources have become increasingly affordable, and with the entrance of several new major competing providers (namely Google and Microsoft, in addition to Amazon) we expect the downward trend to continue. In 2017, a basic Vaportrail instance with 80GB of storage, a single CPU and 8GB of memory can be operated for roughly $10US per month. In many cases, these

resources fall under the limits of a free pricing tier. In general, Vaportrail is storage capacity-bound as it acts as an archive, offloading much of the application computing to the browser. Adding 100GB of storage would cost less than $5US on most cloud platforms[14][5][22] today.

Due to time limitations we were not able to complete a streamlined update process for the prototype appliance, however that process would essentially involve updating the operating system packages (e.g. `apt-get upgrade all`), pulling the latest platform service container image from the Vaportrail registry, and potentially restarting the appliance VM. Given that the supported lifetime of the operating system release is five years, we imagine that a full re-spin of the appliance would be necessary after that period of time. Data would be migrated either by mounting the outgoing appliance's root volume on the new appliance or by exporting/importing data from one appliance to the other.

Backups can be achieved using conventional approaches, i.e. by snapshotting the root volume of the appliance and storing those snapshots on secondary storage (for example Amazon S3[32]). Interestingly, a Vaportrail service connector with sufficient permissions could serve as a backup/export tool targeting external storage services such as Dropbox[13], Google Drive[20] or even another Vaportrail instance. Such a connector would require an uncommon degree of access and, as discussed in Section 3.2, the user would be required to explicitly grant it.

3.1.4 Backend platform components

The *backend* of the appliance hosts several important components of the platform. Each of these components runs in its own Docker container and is allocated a slice of the system resources. In this section we describe each of the major components as well as their communication and network topology within the appliance.

Container networking

The backend containers are network isolated from one another using virtual bridge networks. Specifically, the trusted platform components run on one network (`vplatform`) while third-party service components each run on their own private network. A trusted platform API service is attached to both the platform network and each of

the service connector networks as an “API gateway”, i.e. single point of entry, into the platform services.

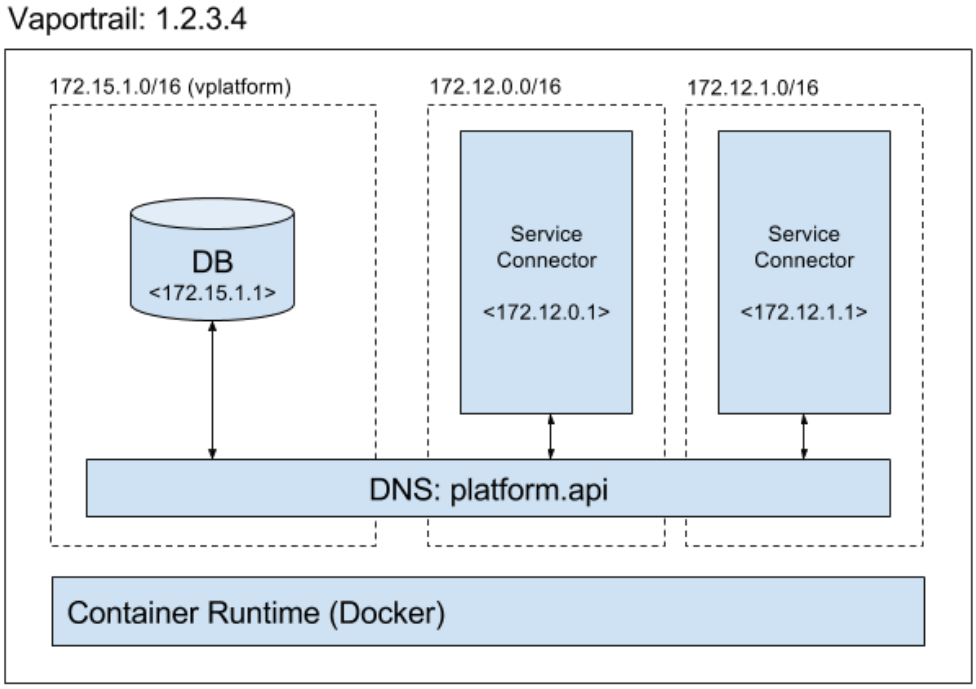

Figure 3.2: Platform components and service connector containers run on isolated bridge networks. The platform API container is routable from service connector networks using the DNS name `platform.api`.

All containers that share a network are IP addressable by one another and can resolve container names (e.g. `api.platform`) using DNS to IP addresses. The bridge network is a very lightweight abstraction that can easily accommodate thousands of networks and containers on a single host. This approach to container networking allows us very fine grained programmatic control over network isolation between components running on the appliance. It also alleviates a lot of challenges around service port collisions and accidental exposure of services to the internet (e.g. if a server binds to `0.0.0.0:80` inside of a container, it will not be bound to the appliances public IP) that are common when running multiple services on

a single host. Although we do not leverage them in the Vaportrail prototype, containers (which are essentially process groups) can also be run under AppArmor[3] profiles to further restrict their egress network traffic, access to the filesystem, and other capabilities.

Platform API

The platform API is the top-level web application that serves the static files (HTML, JavaScript, CSS) that make up the Vaportrail dashboard UI as well as the JSON-based REST endpoints that control the platform. The platform API is exposed as the default web server (port 80 and 443) on the appliance to enable access from the internet, as well as being discoverable as the DNS name `api.platform` from within service connector containers. The API is implemented as a Python web application and mounts several sub application modules at URL base paths corresponding to their functions. For example, the authentication module is available at `api.platform/api/v1.0/authn` while the Data API is accessed at `api.platform/api/v1.0/data`. All of the API endpoints expect JSON input and produce JSON output, and all except for the static HTML endpoint require a valid authentication token to be present in the request. A very basic rate-limiting scheme wraps all endpoints to curtail intentional or accidental abuse.

Authentication and authorization

The platform API has two authentication mechanisms used by the Vaportrail dashboard UI (frontend) and service connectors, respectively. The first mechanism is a fairly standard password-based challenge in which the user submits a password over a secure HTTPS form and receives a time-limited token with which to make subsequent API requests. A production implementation would likely employ a two-step authentication flow using a Time-based One-time Password in addition to the fixed password, but this was beyond the scope of our prototype. The second authentication mechanism, used by service connectors, exploits the container network topology to verify the identity of the client based on an immutable container name.

A service connector only needs to make a request to the authentication API in-

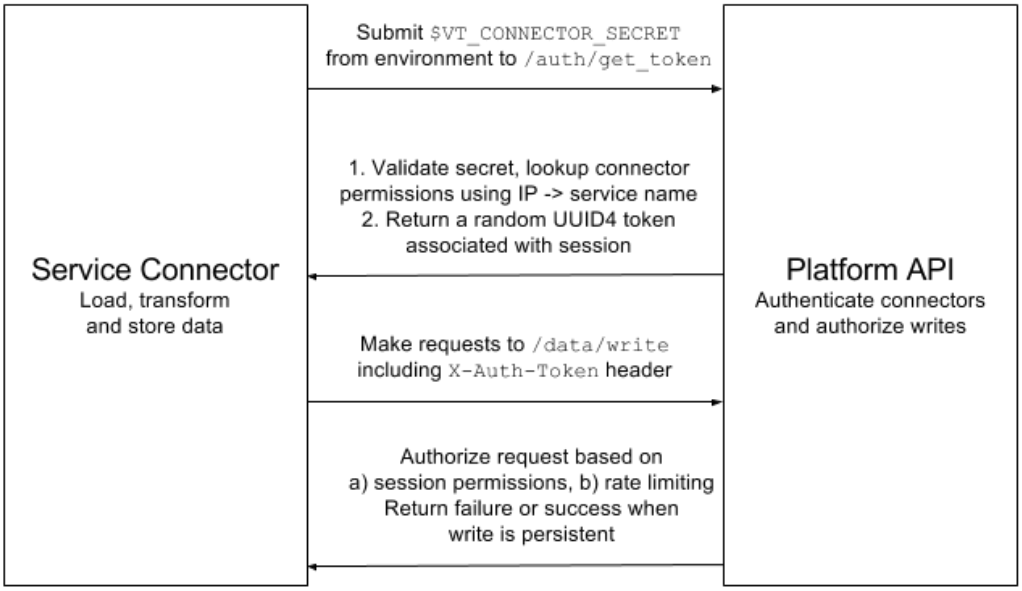

Figure 3.3: Service connectors authenticate with the platform API by presenting a secret the platform places in their environment at initialization. The connectors IP address and immutable container name are used to establish the set of permissions that will be enforced for subsequent requests.

cluding a random Universally Unique Identifier (UUID) Vaportrail instance secret in order to receive a valid API token. The container name is then automatically used to associate the session with the correct permission set for the connector. Once a client is in possession of a valid API token, each API call is authorized based on the permission set associated with the token. We implement both the authentication and authorization steps as middleware between the incoming HTTP request and the actual endpoint implementation, allowing endpoints to specify which authorizations they require in a declarative style. Authorizations can also be performed dynamically as needed: for example, if the permissions required vary between requests to a single endpoint.

Data stores

As discussed in Section 2.4, the platform is built around several specialized data stores, each tailored to different types of data and access patterns. Each data store is a standalone database server that provides at least one of the key storage interfaces identified by the design. The databases run on the trusted platform network and access to them is mediated by the platform API. None of the third-party components that read or write data to the stores have direct network access to the database servers; all operations go through the abstraction layer provided by the platform API. While this implementation choice comes with significant performance overhead (versus allowing clients to communicate directly over native protocols) the value of having a trusted platform component on the data path justifies it, providing both a single point of authorization for each operation and hiding the implementation backing each storage and query interface from clients. Because Vaportrail is a single tenant, privacy-oriented platform, we prefer correctness over performance. The relatively small, personal scale means that even a 200x throughput/latency overhead does not impact the user experience noticeably.

Table 3.1: The data store implementations used by Vaportrail

Interface	Implementation	Notes
SQL store	PostgreSQL[29] 8.4	Postgres is a flexible, performant, modern row-store that speaks an extended SQL variant.
Object store	Riak CS[31] 2.0	Riak CS is an S3 compatible object store built on the Riak KV store.
Memory key/value store	Redis[30] 3.0	Redis is an in-memory store sometimes described as a data structure server. It also provides publish-subscribe functionality.
Document store	MongoDB[27] 2.6	MongoDB is a schemaless (”nosql”) document store with a flexible JSON-based query language.

Each of the data stores has read-write access to a dedicated volume on the appliance's filesystem. A special volume mount is necessary in order to persist data across container restarts as the default container filesystem only lives as long as the container itself. Table 3.1 gives an overview of the specific data store implementations chosen to meet the design requirements. In general we chose the latest stable release of the most established system in each category. Many of the data stores are designed to run at scale with configurable replication and failover policies for high availability. For simplicity we run the data stores in their simplest single node configuration. We did not implement health checks or any kind of watchdog process for the data stores, which would be a wise addition in a production implementation.

Data API

The data API is an abstraction layer on top of the data stores and forms an integral part of the platform API. It provides a write interface with a driver tailored to each data store that enables create, update, and delete operations with semantics determined by the specific store. Similarly, the read (or query) interface provides a driver for each data store that can map a user provided query down to a data-store-specific request. All of the data stores are addressable under a single unified namespace with the structure `<component>.<store>.<collection>.<schema>` which allows the data API to map each request to a specific store driver and for that driver to apply whatever semantics it chooses to the (component, collection, schema) tuple. For example, a Facebook service connector might write to `facebook.sql.timeline.comments` to address a table named "comments" in the "timeline" schema of the "facebook" database in the SQL (PostgresSQL) data store. For writers, the component portion of the namespace tuple is fixed based on the identity of the writer. Unifying the data store namespace provides us with a convenient and familiar grammar for describing permissions across datasets. An application might request access to `facebook.*.*.*` (all of the users Facebook data) or only their images: `facebook.blobs.timeline.images`. The platform itself stores most of its state in `vaportrail.sql.metastore.{users, applications, connectors}` and `vaportrail.blobs.metastore.code`. Although constructing data API requests by hand can be tedious (and is currently

the only option), a well designed client SDK could abstract most of the complexity away for the user.

Service connectors

Service connectors are third-party components that connect to external services and load data into Vaportrail via the data API. Connectors can be built in any language using any framework or libraries the developer chooses, which is crucial given the degree of variation in the wider API ecosystem. Each connector runs in a container on a dedicated bridge network. The only other container that is routable from within the connector is the platform API service, which is discoverable through DNS. The random API secret required to connect to the platform API is injected into the container through an environment variable. Service connectors have unrestricted access to the internet to facilitate diverse authentication workflows and API access patterns. Service connectors are discussed in detail in Section 3.2.

3.1.5 Dashboard user interface

The dashboard (broady the *frontend*) of the appliance is the web application through which the user interacts with Vaportrail. The dashboard is a modern "single page" application composed of several JavaScript components. The entire application is loaded when the user navigates to the Vaportrail instance, and subsequently makes calls into the platform API from JavaScript. We provide an overview of the three major components of the UI in this section, and delve into greater implementation detail for each of them in subsequent sections. Throughout the frontend codebase we use only standard JavaScript, HTML5 APIs and the Bootstrap[6] UI component library. The platform components are largely decoupled from one another and communicate asynchronously across a publish-subscribe bus, event handlers, or the browsers *postMessage()* API.

Vaportrail dashboard. The dashboard is the first view the user sees when they login to Vaportrail. The dashboard provides an overview of which service connectors and applications are installed, as well as available storage capacity. The dashboard view integrates with the application monitor to launch and monitor Vaportrail applications, providing a "dock" along the top of the frame for running applications.

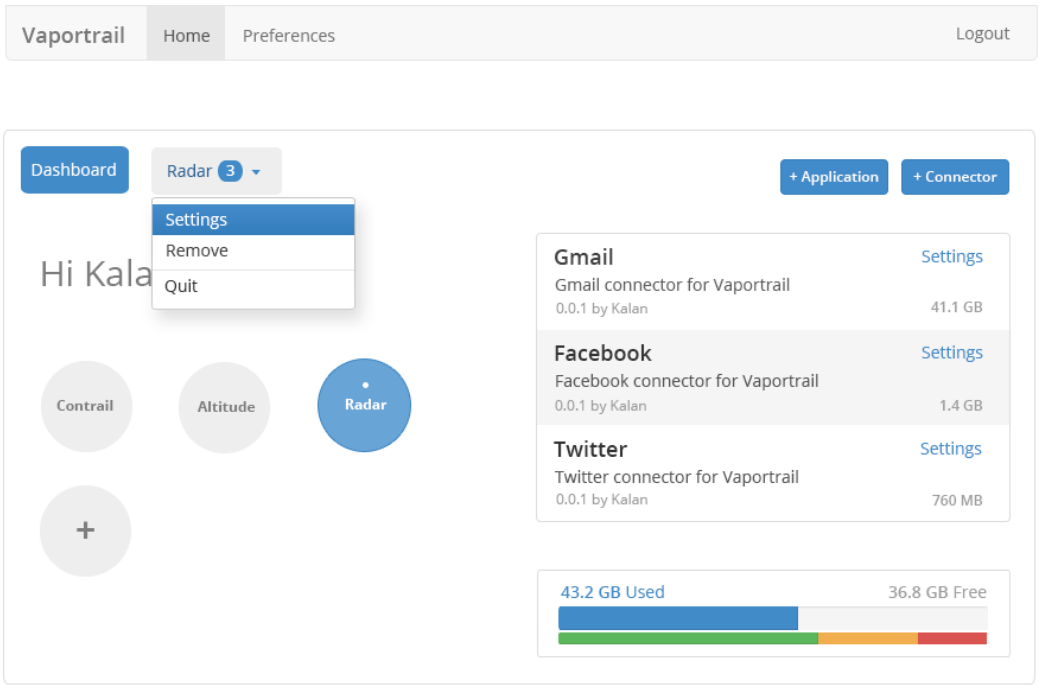

Figure 3.4: The Vaportrail dashboard provides an overview of installed components, applications and available storage capacity.

Application monitor. The application monitor runs in the background and serves as the trusted platform intermediary through which all platform API requests originating in sandboxed applications flow. It is also responsible for launching applications in the application sandbox, and hosts the engine that renders application UI elements. The monitor is responsible for enforcing application access to data stores via the data API.

Application sandbox. The sandbox provides a very restrictive runtime environment in which arbitrary JavaScript code can run completely isolated from the DOM and browser APIs. Each running application has a dedicated instance of the sandbox running in a browser thread separate from the application monitor and stream view. The sandbox hosts the untrusted application code and traps platform calls in

the guest code, routing them into the monitor.

3.2 Service connectors

Service connectors are an integral part of the Vaportrail platform, serving as the conduits through which data is exported from external services and loaded into the platform. Like applications, service connectors are developed by third-parties. They integrate with external services using the SDKs and APIs provided by those services. A typical service connector will complete an authentication workflow with a service as part of its installation process and subsequently synchronize with the service periodically or in real-time, depending on the nature of the service and the application use cases imaged by the developer. Once a service connector is installed, the user can install applications that know how to process data stored by that connector.

3.2.1 Packaging and distribution

As discussed in Section 3.1.4, each service connector runs in a Docker container on an isolated virtual network within the platform appliance. The Docker tooling provides us with a very convenient format for efficiently distributing versioned binary container images with built-in data integrity checks. We build on a packaging convention that has become common for browser extensions to create our own standalone service connector package format that consists of a gzipped tarfile containing a JSON manifest and, optionally, the container image in the Docker image format.

The manifest is a simple JSON file that specifies details about the connector (name, author, version, etc.), the platform permissions it requires to run, any settings that can be configured by the user, and a reference to the container image either as a package-relative path, or a Docker registry URL where the image can be downloaded from. This package format has a number of benefits:

- It is simple to inspect and construct using standard tools (vi, Docker, tar)
- It is built on an open, and increasingly standard, container image format
- The JSON manifest is extensible, human writable and machine readable

Program 3.1 A JSON service connector manifest specifying package metadata and platform permissions required by a Facebook connector.

```
{
    "name": "Facebook Connector",
    "author": "Kalan MacRow <kalanwm@cs.ubc.ca>",
    "description": "Facebook for Vaportrail",
    "version": "0.0.1",
    "image": "assets/image.tar",
    "permissions": [
        {"type": "write", "schema": "facebook.*.*.*"},
        {"type": "memory", "request": "1G"},
        {"type": "share", "type": "image/*"},
        {"type": "host_port", "port": 8080}
    ]
}
```

- It supports distribution either as a complete binary package, or as a lightweight installer

Although we do not implement or enforce any particular cryptographic features (e.g. package signing) the simple, self-contained, nature of the format invites the use of existing tools (e.g. GnuPG[18]) for this purpose. We could imagine incorporating package singing directly into the platform tooling in the future. Building on a simple, open package format is crucial to facilitating adoption in the open source community, and we believe the Vaportrail package format meets this design goal.

3.2.2 Isolation and developer flexibility

Running service connectors in resource-restricted, network-isolated containers helps us meet two important design goals: connectors are isolated from the core platform services, data stores, and each other while also ensuring developers have the flexibility to use whichever Linux distribution, frameworks, libraries, languages and general program structure they prefer in implementing the connector. The container environment provides a similar degree of self-determination as would a dedicated virtual machine, with the caveat that the operating system must be a flavour

of Linux. Flexibility is critical because of the large degree of variation in how external services choose to expose APIs and data. There are a multitude of authentication protocols and workflows, some open standards (OAuth2[43], OpenID[49]), and many proprietary. While SDKs are often made available in several languages, the process of installing and configuring them usually involves installing dependencies, updating or setting environment variables, and creating files at sensitive filesystem locations. In short, restricting the service connector sandbox to a more managed environment, as we do with applications, would significantly limit the likelihood that many connectors are built. We believe implementing service connectors this way balances the need for isolation, in accordance with the platform trust model, with the need for developer flexibility in integrating with a highly fragmented wider API ecosystem.

3.2.3 Permissions and capabilities

Service connectors are able to request *required* and *optional* permissions from the user via the package manifest. The permissions are presented to the user upon installation for explicit approval before the connector image is loaded into the appliance environment and scheduled to run. The permissions fall into two broad categories.

System resources. Connectors can request specific memory, CPU slices and appliance port forwarding. These advanced permissions exist to accommodate connectors that a) may perform non-trivial data transformations requiring more than the default memory and CPU allocations, and b) authentication workflows or APIs that involve "callbacks" (e.g. WebHooks) from external services. The platform UI clearly identifies the gravity of these advanced permissions at install time so that the user can make an informed decision.

Data store access. Connectors request access to the specific data store schemas they need to read or write to. The schemas need not exist at install time: they will be created on the first write operation, however, the user must explicitly approve the connector access. The platform UI clearly identifies when a connector is request-

ing a) read access to any schema, given the risk of exfiltration and b) any access to a schema that already exists, given the risk of data corruption or exfiltration.

3.2.4 Connector lifecycle

The user can install connectors by uploading a package through the dashboard UI. The platform API places the package in a staging location and returns the manifest to the UI for the approval workflow. If the user approves the permissions and settings for the connector, the manifest is stored in the platform metabase and the connector image is imported into the appliance. The connector container is created and started using either the default system resource limits or the (approved) requested limits. If port forwarding is requested, the platform chooses a random free port on the appliance and maps it to the requested port in the container. The host IP address and forwarded port are then passed into the container in an environment variable so that they can advertised as needed.

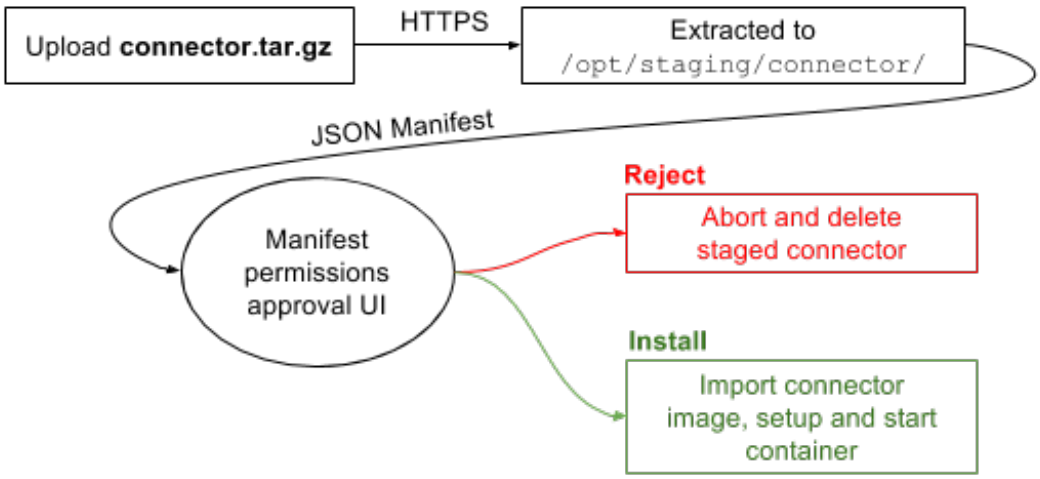

Figure 3.5: The installation flow for a service connector. A connector package is uploaded through the dashboard and unpacked to a staging location on the appliance filesystem. The user then approves or rejects the platform permissions required by the connector manifest via a UI workflow.

Once the connector has started, the user is free to access its configuration page

which can be used for changing settings or completing authentication workflows. The configuration page is served by the service connector itself, and is available through a random, temporarily mapped port accessible only from the users current IP address. The connector then runs indefinitely, scheduling data ingestion as and when it deems appropriate. If the user no longer wants the service connector, it can be stopped and completely removed through the dashboard. Although the connector software can be removed, we do not currently provide a mechanism for wholesale deletion of any associated data store schemas.

3.2.5 Reading and writing data

The primary purpose of service connectors is to load personal data streams into the platform. They accomplish this by writing into the data API endpoints of the platform API. The API is accessed from within the container using the DNS name `api.platform`. In the simplest case, the connector will load data from its upstream service(s) and write the objects or records directly to a schema that it has access to, with minimal transformation. In some cases, the connector may need to load some state (e.g. a checkpoint marker of some kind) from the data API or its local filesystem, and or may perform some non-trivial operations or transformations (e.g. aggregation, rollup, deduplication, downsampling) before writing it into the platform. The platform will automatically provide back-pressure in the form of API rate limiting (communicated back to the connector as an `HTTP 429 -- ''Too Many Requests''` response) and enforce the schema access permissions configured at install time. In general, connectors can "fire and forget" data into the platform.

3.2.6 Example connectors

We implement three prototype service connectors for popular services to demonstrate the capabilities of the platform, and to serve as test mules in evaluating the design and implementation of the platform API and service connector runtime. We planned the connector implementations by referring to the best practices prescribed by the respective services, and without regard for any limitations that the platform design might impose on us.

Facebook

Facebook remains the dominant social network globally with over 1 billion active users[40] in 2017. It is a significant personal data sink for many users, who generate a continuous stream of status updates, photo uploads, likes and comments everyday. The company provides a comprehensive API that gives users the ability to search, export and post content programmatically with nearly as much flexibility as the main Facebook website itself. Given its prominence and excellent API, Facebook would be a canonical data source for Vaportrail and thus a good candidate for prototyping. The service connector is based on a standard Ubuntu 14.04 base image and uses the Python bindings for the Facebook Graph API[16] to query the user's posts every 10 minutes. The connector requests write access to three data stores:

- `facebook.sql.posts.status` (status updates)
- `facebook.sql.posts.comment` (comments posted by the user)
- `facebook.blob.posts.photo` (photos posted by the user)

It creates the relational tables for statuses and comments using the data API `create_table_if_not_exists`. The connector code simply waits until a file is created at `/var/auth_token` by completion of the OAuth authentication workflow with Facebook and then enters an infinite loop, syncing new posts before sleeping for 10 minutes. The connector only requests a very limited set of read-only Facebook permissions: `public_profile`, `user_posts` and `user_photos`. Although our prototype only imports a small subset of the data available, it is sufficient to validate our design goal that it should be simple and frictionless to develop connectors that load diverse types of objects.

Twitter

Twitter is the world's leading microblogging website in 2017 with over 320 million active users globally[50]. For many users, their stream of Twitter updates (tweets) represents an important personal log of daily activities and interactions. Like the Facebook connector, we base the Twitter service connector on an Ubuntu 14.04

base image and use the service connector configuration page to trigger an OAuth flow with Twitter. We then use a Python script to access the Twitter User Stream REST API[33]. The User stream is an HTTP long-polling endpoint that returns a stream of JSON objects representing the activity of the authenticated user. We filter the stream to only include the users updates. The connector only populates a single data schema:

- `twitter.nosql.stream.events`

In this case we push all stream updates (tweets, retweets, likes) into a single unstructured `stream.events` collection, which seemed more natural than routing them into separate tables. At the time of writing, the User stream API is being phased out in favour of a WebHook-based "push" model. We could readily support this change by having the service connector request an appliance host port mapping into a small stateless HTTP handler that would write the JSON payload into the `stream.events` collection.

Gmail

Email is a crucial archive of personal data in the form of personal correspondences,purchase receipts, travel itineraries, photos, and documents. Googles Gmail is one the leading free email services and offers a number of options for programmatic access to the inbox, including traditional IMAP and RESTful APIs[19].

We built the Gmail connector in much the same way as the Facebook and Twitter prototypes, and opted for the more convenient, modern RESTful API that provides high-level JSON interfaces to messages threads and message content. We designed the data schema to align with the top level JSON objects:

- `gmail.nosql.inbox.threads`
- `gmail.nosql.inbox.messages`
- `gmail.blobs.inbox.attachments`

The unstructured data store allowed us to beginning storing data with a minimum of table or schema design effort. To demonstrate the flexibility of the platform service connector environment we elected to use the Gmail Java SDK to develop

the connector. To accomplish this we install the open Java 8 runtime[28] in the container and bundle the Gmail SDK JAR files. The connector runs a Java application with a single threaded web server for configuration and authentication, and the syncing logic on a separate thread. Currently the prototype only fetches messages that arrive after the connector is installed. We imagine a more sophisticated implementation would allow the user to backfill messages for a period of time in additional to capturing new ones as they are sent or received.

3.2.7 Summary

In this section we have discussed the implementation of service connectors including how they are deployed and isolated within the platform, how they are packaged for sharing and distribution, what permissions and capabilities they have at runtime and how they read and write data into the platform. We have also discussed the implementation of three non-trivial prototype connectors that demonstrate the flexibility of the runtime environment and ability of the platform to accommodate real-world data models.

3.3 Application Sandbox

In this section we discuss the implementation of the application sandbox, which provides the isolation necessary to expose personal data streams to untrusted applications running in the browser. The sandbox hosts JavaScript code written by third-party developers and works in an unmodified, modern HTML5 browser. Several platform APIs are baked into the sandbox environment that allow applications to query personal data stores, render UI elements, share results through *intents* and persist state. Because our aim is foster an ecosystem of purpose-built Vaportrail applications, and not to support existing web applications general, we trade emulation of the standard browser environment (e.g. DOM and related APIs) for more robust isolation through a completely virtualized JavaScript interpreter. We argue that the straightline performance overhead incurred, while significant, is quite acceptable in practice; vaportrail applications tend to be I/O bound. We also hope to demonstrate that even the limited set of APIs we implemented in the prototype are sufficient for building interesting applications. Furthermore, with some

effort it would be feasible to fully emulate the DOM and native APIs in the hosted JavaScript environment.

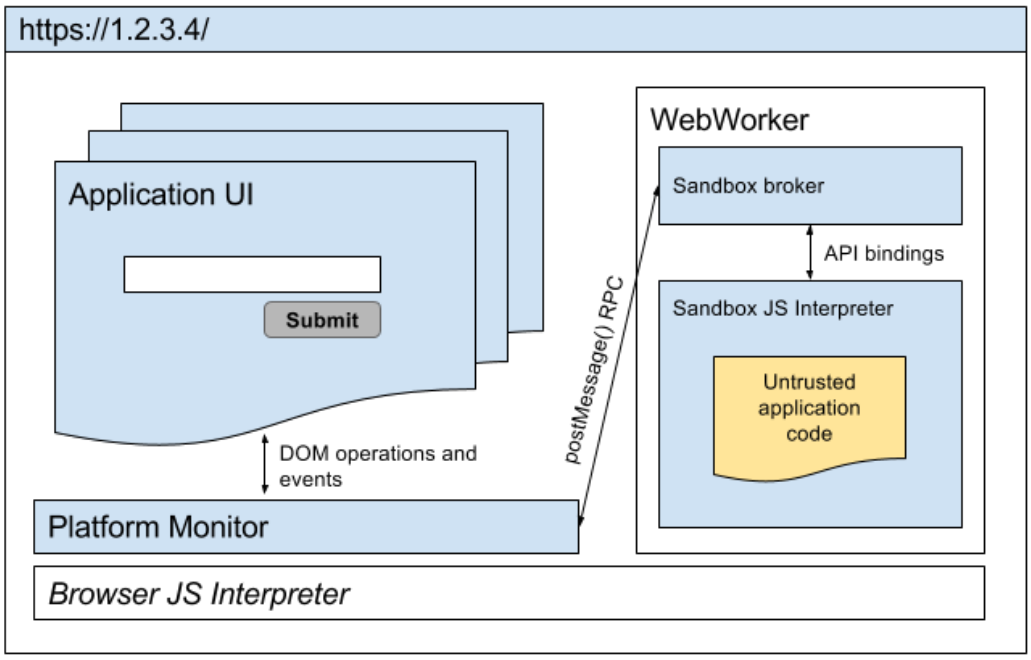

Figure 3.6: Vaportrail applications run in a dedicated JavaScript interpreter within a WebWorker thread. A trusted broker marshals application API calls over an asynchronous RPC interface and into the platform monitor where authorization checks are made before updating the DOM, or making requests into the platform API over the network.

3.3.1 WebWorkers as execution containers

Much like Treehouse[44], we exploit WebWorkers as a standard, natively supported execution container. The WebWorker provides a thread of execution (usually an OS thread, though it depends on browser implementation) and a JavaScript context separate from that of the main web page. WebWorkers do not share memory or object references directly with the main page and can only communicate with it via a very narrow message-based interface (*postMessage()*). Out of the box, the worker provides a useful degree of isolation from the main page in that a) the

DOM and window objects are not accessible from the worker context and, b) the worker process can be monitored and terminated from the main page context. For simply preventing guest code from modifying the visible page, the worker abstraction alone may be enough. However, although WebWorkers cannot manipulate the DOM, they do have the ability to spawn child workers, make network requests and import arbitrary JavaScript code. Other sandbox implementations (discussed in Chapter 4) have attempted to limit access to these capabilities by overwriting or interposing on native APIs before loading guest code, applying Content Security Policy (CSP) or statically enforcing a safe subset of JavaScript. The multiplicity of non-standard browser implementations and corner cases makes it difficult to argue convincingly that solutions based on these approaches provide comprehensive isolation. As in Treehouse, we initialize the WebWorker with a monitor (or broker) module that prepares the environment for executing guest code. Instead of locking or freezing native APIs, we embed a complete JavaScript engine and wire our platform APIs into it so that they appear "native" to guest application code. When the environment is ready, we hand control over to the application.

3.3.2 Hosted JavaScript runtime

The application sandbox is built on the js.js[51] runtime, which is a stripped down version of Mozilla's JavaScript engine compiled first to LLVM[25] and then to asm.js[4] (a highly optimizable subset of JavaScript) using emscripten[15]. Js.js is a complete JavaScript interpreter, equal in capability (if not performance) to the interpreter hosting it in the browser. We use broadly the same interfaces to "wire" our platform APIs into the interpreter as would a web browser to implement the DOM and other standard native APIs. The platform broker initializes a new instance of the JavaScript virtual machine, installs the platform APIs, and then loads the guest code into the runtime. A platform-defined entrypoint triggers a lifecycle event that the application code registers a handler on. The application-defined handler serves as the application's `main()` function.

Although compute-intensive code running in js.js is roughly two orders of magnitude(200x) slower than the same code running in a native interpreter, we find the overhead almost unnoticeable in practice: applications tend to be IO bound,

and much of the heavy-lifting (animation, DOM updates) is offloaded to native JavaScript through high-level platform APIs. Applications that apply machine-learning algorithms, image processing or other tasks that require straightline performance will be affected the most, however, the relatively small scale of personal data and the promise of WebAssembly as a target replacement for asm.js helps mitigate concerns looking forward. We could have taken the virtualization a step further and provided a limited Linux environment within the browser, however, this would have come with increased complexity and would completely divorce the sandbox environment from the ergonomics of traditional web development.

The JavaScript virtual machine provides robust isolation at an acceptable performance cost. We believe it achieves the design goal of building a practical system based on mechanisms strong enough for users to trust with their personal data today.

3.3.3 Application sandbox lifecycle

Each Vaportrail application instance runs in a dedicated WebWorker and JavaScript virtual runtime created and managed by the platform monitor from the main page context. When the user launches an application from their dashboard, a new tab is created in the application dock and in the background the monitor creates a new WebWorker based on the platform sandbox broker. When the new worker starts, the broker code initializes its hosted JavaScript interpreter by installing references to all of the global platform API objects. When the worker is ready to execute guest (application) code, it notifies the monitor via the postMessage() interface. The platform then responds with the application JavaScript code, which the broker loads into the interpreter.

Any compilation errors are passed back to the monitor, which then terminates the worker. If the application code compiles, the sandbox is placed in the *ready* state for the monitor to start at any time. When the broker receives the *start* command, the control is handed over to the hosted interpreter, which executes some bootstrapping code and triggers an application lifecycle event, ultimately entering the application handler. The platform monitor pings the worker at regular intervals to check for liveness, and will terminate the worker if it does not receive a re-

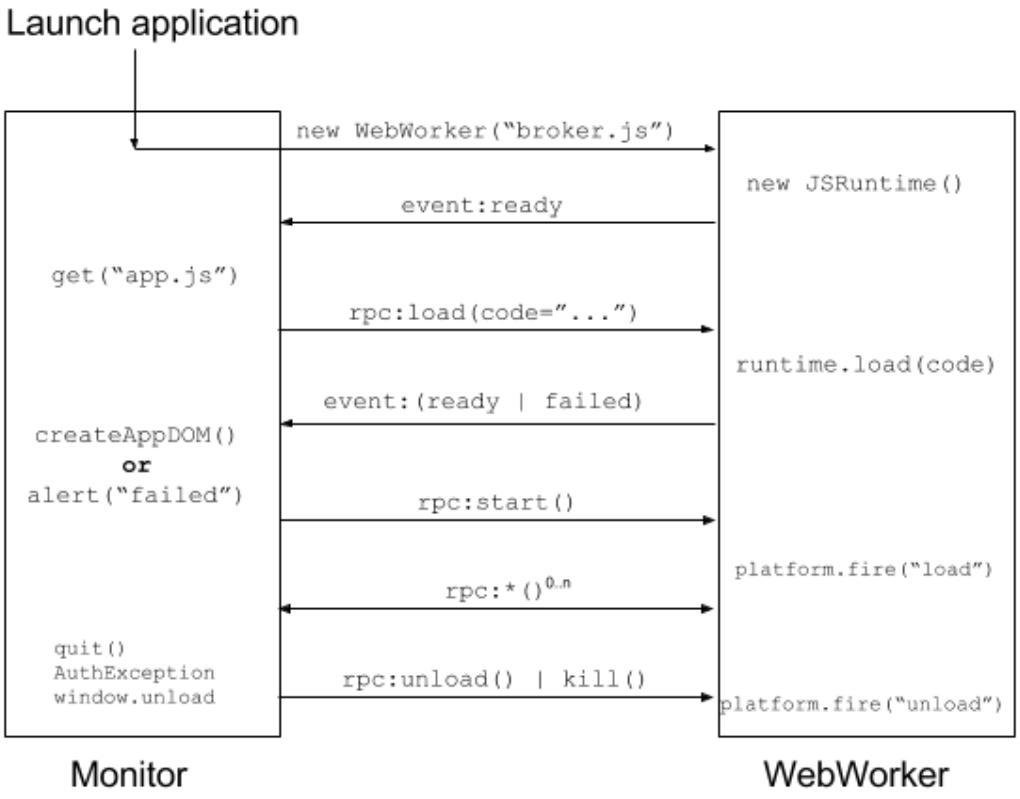

Figure 3.7: When an application is launched, the monitor and the sandbox broker coordinate to setup the new environment for the application instance, creating a new DOM root in the dashboard and loading the application code into the sandbox interpreter.

sponse. When the user quits the application from the dashboard, the monitor sends an *unload* command to the broker, which then fires the unload application lifecycle event inside the interpreter. The application can handle this event to persist any state before exiting. Finally, the broker notifies the monitor that the application has unloaded, and the monitor terminates the worker. As discussed in Section 3.3.4, unauthorized API calls will also trigger termination of the application without notice.

3.3.4 Application monitoring

The Vaportrail monitor runs as a component within the dashboard and is responsible for launching applications (creating instances of the sandbox) and supervising them at runtime via a) continuous health checks and b) authorizing platform API calls made by application code and enforcing platform policies.

While an application is running, the monitor *pings* the application worker every 10 seconds and expects a response within 1 second that includes any error codes from the sandbox interpreter. The purpose of the health check is to ensure that the JavaScript context within the worker has not become wedged or halted unexpectedly, e.g. due to entering a tight infinite loop or an unhandled exception, respectively. The WebWorker isolation ensures that a broken or misbehaving worker cannot adversely affect the responsiveness of the main page or other Vaportrail applications, however, without a health check the application could crash or become unresponsive without the platform (or the user) knowing.

The monitor is also responsible for authorizing platform API calls that originate within sandboxed applications. If an application attempts to make an unauthorized API call, we terminate it immediately, bypassing the normal *unload* lifecycle flow. The reason for this is two-fold. First, we provide a platform permissions API that allows an application to check whether it has permission to execute a particular API call. This is necessary to support optional permission grants, and to allow applications with pattern-based permission grants (discussed in Section 3.4) to test specific access at runtime. Thus, there is no need for an application to issue an unauthorized API call to probe for permission. Second, issuing unauthorized calls strongly implies the app may be malicious, or at least broken. In either case, we err on the side of caution by implementing a zero tolerance policy.

With relatively heavy-handed isolation of application code in place, it could be argued that there is little need for limiting application access to data through fine grained permissions: with no access to native APIs and only the small platform API and mature JavaScript interpreter as an attack surface, data exfiltration is unlikely. We argue that there are at least three compelling reasons to gate access to data and sharing:

1. It is conceivable that in the future we would add permissions allowing ap-

plications more access to the network, increasing exfiltration risks. Having a framework in place that limits exposure to specific data sets will be important.

2. As a design axiom of secure systems, we apply the principle of least privilege throughout the platform. Restricting applications to only the data they need is in keeping with this principle.

3. Clearly defined permissions allow the platform to maintain a dependency graph of applications and data schemas, facilitating a better user experience in the event that a service connector is being upgraded or removed, potentially affecting the downstream application(s).

Although the application sandbox provides robust isolation, we implement these additional layers of monitoring and authorization to further protect against broken or malicious applications, and or the event of a sandbox escape.

3.3.5 Platform APIs

The application sandbox embeds a number of platform objects and APIs that applications can use to query data stores, trigger sharing intents, render UI elements and persist application state. The platform APIs are broken into namespaces (ui, net, sharing, query, etc.) and wired directly into the sandbox interpreter so that they appear to guest code as "native" APIs along with other JavaScript standard library classes and objects.

RPC mechanism

We implement a custom RPC protocol that transparently proxies operations on platform API objects within the sandbox interpreter through the broker and across the *postMessage()* interface to corresponding components in the monitor. In the other direction, we route events from the monitor side into the application by replaying them on objects in the interpreter. The objects that the application code interacts with inside the sandbox are little more than lightweight proxies instrumented to forward method calls and property accesses through the RPC mechanism. Component implementations sanitize all arguments and data originating

within applications to protect against script injection.

Program 3.2 Method calls on remote objects are implemented by mapping a (taskID, objectID) pair to a specific object instance and applying the method on a set of wrapped arguments. Argument wrapping allows us to transparently support callbacks into the sandbox interpreter by hiding function pointer semantics in simple callable functions.

```
rpc['__method'] = (function(objId, method, args, rpc){
    var task = this._tasks[rpc.taskId],
        obj = task.objects[objId];
args = this.wrapMethodArgs(args, rpc);
obj[method].apply(obj, args);
return true;
});
```

platform.ui

The platform UI toolkit provides a number of user interface components that applications can use to render static or interactive UIs. The platform establishes a DOM root node for the application and platform API classes based on Bootstrap Components[6]. The components created by platform.ui toolkit are consistent in form with the rest of the Vaportrail dashboard, but are styled to be visually distinct from trusted platform UI elements. Applications are given very little control over the visual style of the elements which ensures a) applications cannot "disguise" themselves as platform features and, b) a consistent visual style is maintained across applications. A complete listing of the available UI components is available in Appendix A.

platform.net

The platform network API allows applications to interact with the network subject to very restrictive permissions. Currently the only network access available is to fetch a pre-approved URL once every 24hrs. Construction of the actual HTTP request is handled in the monitor; the application only passes an identifier corresponding to a URL that was included in the application manifest and approved at

install time. Preventing the application from fetching the URL more than once per day mitigates the risk of the mechanism being abused (to any practical extent) as an exfiltration channel. Providing a unidirectional mechanism to fetch a URL enables a class of applications that rely on evolving external data sources (e.g. gas prices, interest rates or machine-learning models).

platform.share

The platform sharing API allows applications to integrate with a platform mediated sharing mechanism. The application signals that an item is *shareable* and the platform matches the item with a list of installed service connectors capable of handling it, based on the type of data. Currently `platform.share` supports only two rudimentary forms of sharing, `text/*` and `image/*`, however we imagine the API evolving to support more sophisticated, platform-managed channels. We did not have time to implement a service connector with sharing support, although the mechanism is supported. Service connectors can specify which share types they support (based on MIME-type pattern) in their manifest. They can then poll a platform API to receive shares that have been queued for them to handle.

platform.query

The platform query API allows applications to execute queries against the personal data stores that they have permission to access. The API is read-only and presents a unified interface to the various storage interfaces. In our prototype, applications need to be aware of the query language used by the underlying data store. A more sophisticated implementation might provide a unified, higher-level query language. All results are returned as JSON, including binary objects. The exact format of the JSON depends on the type of data store. For example, the SQL store will return a list of lists representing a result set, while the document store will return a list of objects. We do not support pagination or streaming for large result sets: the entire response is returned for each query. Applications can implement streaming/pagination by making repeated queries with the appropriate filter clauses. Even once a specific query API call has been authorized by the monitor for the application, the monitor itself only has read-only access to non-platform data schemas through the

data API.

platform.localStorage

We emulate the standard HTML5 LocalStorage[37] API (window.localStorage) in the sandbox environment to provide applications with a general-purpose facility for persisting state across runs. The LocalStorage API is implemented as a global object with a `getItem(key)`, `setItem(key, value)` interface. Any properties set on the global object are automatically relayed into the monitor and saved to a platform-managed application state store. When the platform instantiates a new application sandbox, the saved state is passed into the worker along with the application code, and the LocalStorage is initialized before the application code executes. All values are automatically converted to strings, as in the standard LocalStorage implementation. We do not support the `StorageEvent` interface for notifying other instances of the same application of changes to the storage object.

3.3.6 Summary

Vaportrail applications run in a robust and flexible sandbox environment based on dedicated JavaScript interpreters using WebWorkers as execution containers. Our implementation is cross-platform and runs in unmodified modern browsers. We find the performance overhead imposed by the hosted JavaScript interpreter to be acceptable in practice today, and we expect the performance of this approach to improve with time. We leverage js.js's interface for binding functions and objects into the interpreter context to expose a range of platform APIs that developers can use to build rich data-driven applications. The platform APIs available to applications are built on a simple RPC mechanism between the sandbox and the platform, and are easily extend. We believe our prototype application sandbox meets the design goals of providing robust isolation that is practical and works today, while also providing a familiar developer and user experience that will foster adoption of the platform.

3.4 Applications

Vaportrail applications complement service connectors by providing the user with a way to leverage their personal data in new tools, visualizations, and games developed by untrusted third-parties. In this section we discuss how applications are packaged and distributed, general development using the platform-provided APIs, permissions and capabilities, and finally we present three prototype applications that demonstrate the flexibility of the platform.

3.4.1 Packaging and distribution

Much like their service connector counterparts, applications are packaged as a simple, self-contained format that facilitates sharing through conventional channels (e.g. email or HTTP download). Like service connectors, application packages can be built using standard, familiar tools like `tar` and a text editor. The package is a gzipped tarfile containing an application manifest, the application code and any assets (images or other dependencies). As with service connectors, the application manifest is specified in JSON and describes the permissions the application requires to run.

Vaportrail applications can be written in JavaScript, or any programming language that can be compiled to JavaScript, e.g. TypeScript[34], CoffeeScript[9], Dart[11], or Caja[7]. In our prototype implementation, all of the application code must reside in a single file. This would be an unreasonable limitation for a release implementation, however, a standard form of module loading could easily be added.

3.4.2 The application manifest

The application manifest is a machine (and human) readable JSON file that describes various properties of the application (name, version, author, release date, etc.) as well as the platform permissions it requires. The manifest fully specifies which data schemas the application requests access to, as well as which URLs it can fetch data from, and which types of data it is capable of sharing. Any of these permissions can be marked as required or optional: an application cannot be installed if one or more of its required permissions are not granted by the user. Whether

Program 3.3 The JSON application manifest specifying package metadata and the platform permissions required by the Radar application.

```
{
    "name": "Radar App",
    "author": "Kalan MacRow <kalanwm@cs.ubc.ca>",
    "description": "Radar for Vaportrail",
    "version": "0.0.1",
    "code": "src/radar.js",
    "permissions": [
        {"type": "read", "schema": "facebook.*.*.*"},
        {"type": "read", "schema": "twitter.*.*.*"},
        {"type": "read", "schema": "gmail.*.*.*"},
        {"type": "network_get", "url": "http://", "id": "SPAM_DB"},
        {"type": "window_open", "*.facebook.com"},
        {"type": "window_open", "*.twitter.com"},
        {"type": "window_open", "*.gmail.com"}
    ]
}
```

or not optional permissions were granted can be discovered by the application at runtime using the `platform.permissions.can(`*`permission`*`)` API. We outline the application permission classes in Table 3.2. All applications have access to all `platform.ui` components as well as `platform.localStorage`.

3.4.3 Application lifecycle

Vaportrail applications have an event-driven lifecycle that is closely related to the platform sandbox lifecycle, and should be familiar to developers accustomed to building browser-based software. When the user launches an application from the platform dashboard, the monitor creates and initializes a new instance of the application sandbox.

When the sandbox reaches the ready state, and the application code has successfully been loaded into the interpreter, the application lifecycle begins. The broker managing the interpreter hands control over to the interpreter, which evaluates all code in the global scope, including a synthetic (i.e. generated) bootstrapping

Table 3.2: Application permission classes

Permission class	Example	Description
Query data	`{"type": "read", "schema": "gmail.sql.inbox.messages"}`	Request access to a data schema. Wildcards can be used in any component of the schema name.
Share content	`{"type": "share", "content": "text/plain"}`	Request access to share content by MIME-type. Wildcards can be used.
Network access	`{"type": "network_get", "url": "http://.../prices.json", "id": "PRICE_DB"}`	Network access capabilities. We only support fetching approved URLs by name. Wildcard patterns are *not* supported in URLs.
Navigation	`{"type": "open_window", "domain": "(.*).facebook.com"}`	If enabled, the application can use `platform.open` to open browser windows to domains that match the specified domain pattern.

function that in turn fires the `platform.onload` event. The application code should use `platform.addEventListener('load')` to register a handler on the event. The onload handler is effectively the application's "main" function, and should be used to load state, issue queries and create UI elements.

Program 3.4 A "Hello, World!" Vaportrail application that stores the date of its last run in localStorage, logs a message to the browser console, and creates a modal dialog with a familiar salutation.

```
use strict;
!function(){
platform.addEventListener('load', function(e) {
localStorage['lastRun'] = new Date() + '';
console.log('Hello, log!');
platform.ui.alert('Hello, world!');
});
}();
```

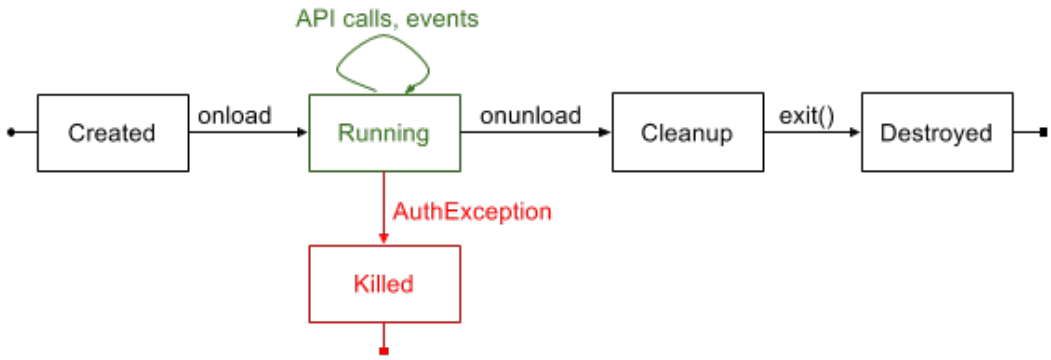

Figure 3.8: The lifecycle of a Vaportrail application.

Nearly all interaction with the platform API is asynchronous. The application may also register a handler on the `platform.onunload` event, which is fired by the monitor when a) the user closes the application or, b) the browser window is closed. The `unload` handler can be used to persist state using `platform.localStorage`. As discussed in Section 3.3.4, the monitor will forcibly terminate the application and sandbox (skipping the `unload` event) if an unauthorized API call is made. The most important phase of life for a Vaportrail application happens between the `load` and `unload` events, during which time it responds to events triggered by the user (via UI components), processes data, and updates its UI.

3.4.4 Example applications

We implement three prototype applications to help motivate the design and implementation of the platform APIs, and to demonstrate the platforms ability to support interesting, non-trivial applications. Each application consumes data from multiple services, depends on persistent state and provides a simple UI.

Contrail

Contrail is a simple application that renders events from multiple services as icons on a horizontal timeline. Each event (e.g. email, post, upload) is plotted as an icon sized proportional to the "impact" of the event. The definition of impact depends on the type of event: it corresponds to likes, retweets (replies for private messages)

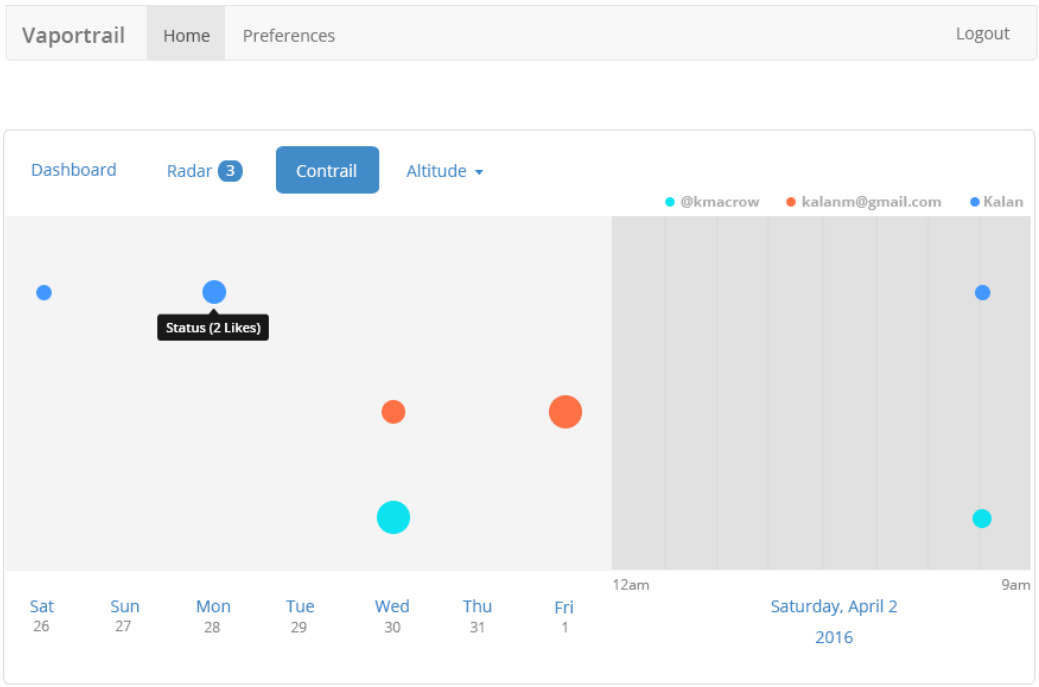

Figure 3.9: The *Contrail* timeline application.

and thread size for Facebook, Twitter and Gmail, respectively. The application shows the current day's events at hourly resolution, and the trailing week at daily resolution. Contrail demonstrates a simple, but useful visualization that without Vaportrail would have required granting a third-party application access to three personal data sources.

Altitude

Altitude is a basic search engine that provides an aggregate free-form search across several data sources. It leverages the native text search capabilities of the backend data stores (i.e. SQLs `LIKE %` clause, and MongoDB's `$text` operator) to combine events, messages, email attachments and images into one result set. The application uses `platform.open()` in the onclick handler of `platform.ui.Link` components to link the user out of Vaportrail into the originating service's view for the item. *Altitude* demonstrates the flexibility of the platform APIs and the utility

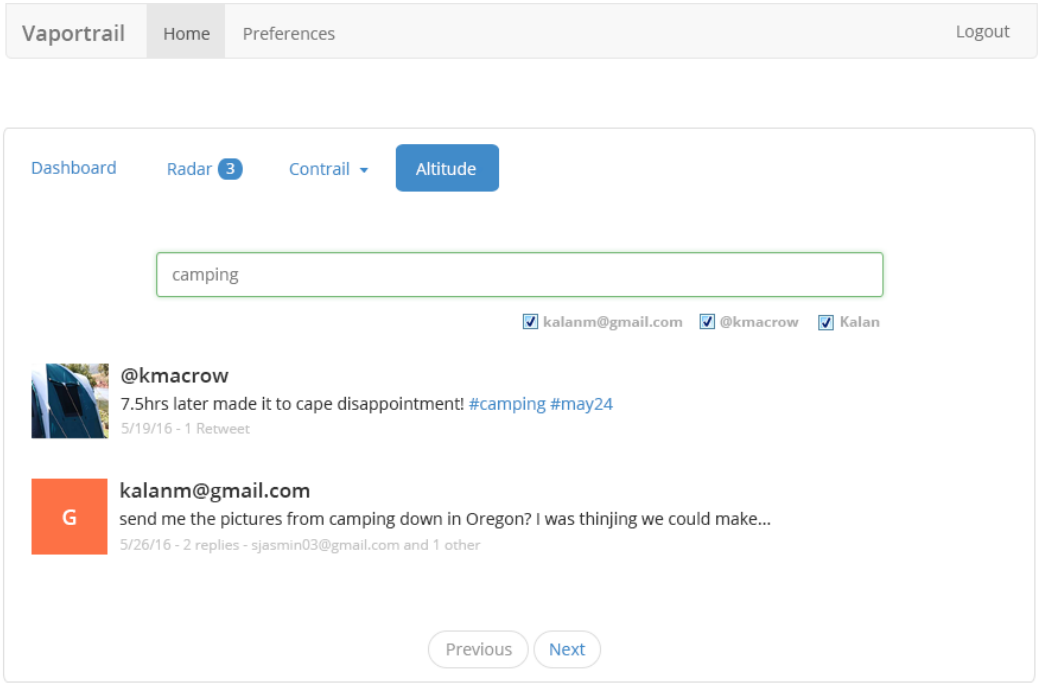

Figure 3.10: The *Altitude* search application.

of applications that provide a familiar primitive (in this case search) over a unified view of personal data.

Radar

Radar is a rudimentary fraud detection application that leverages the network access permission to periodically download a database of spam detection patterns which it applies to all outgoing messages across Facebook, Twitter and Gmail. The rationale is that if a spam message has been sent from a personal account, the account has likely been compromised. Each monitored service appears in the UI along with its current status (Green - all clear, Yellow - possible spam detected, or Red - account compromised). When a message that is likely spam is detected, it is shown in the UI with a direct link to view in the context of the originating service. Radar uses a simple detection model based on a set of regular expressions, each mapping to a score and human-readable reason that a match implies spam. Each

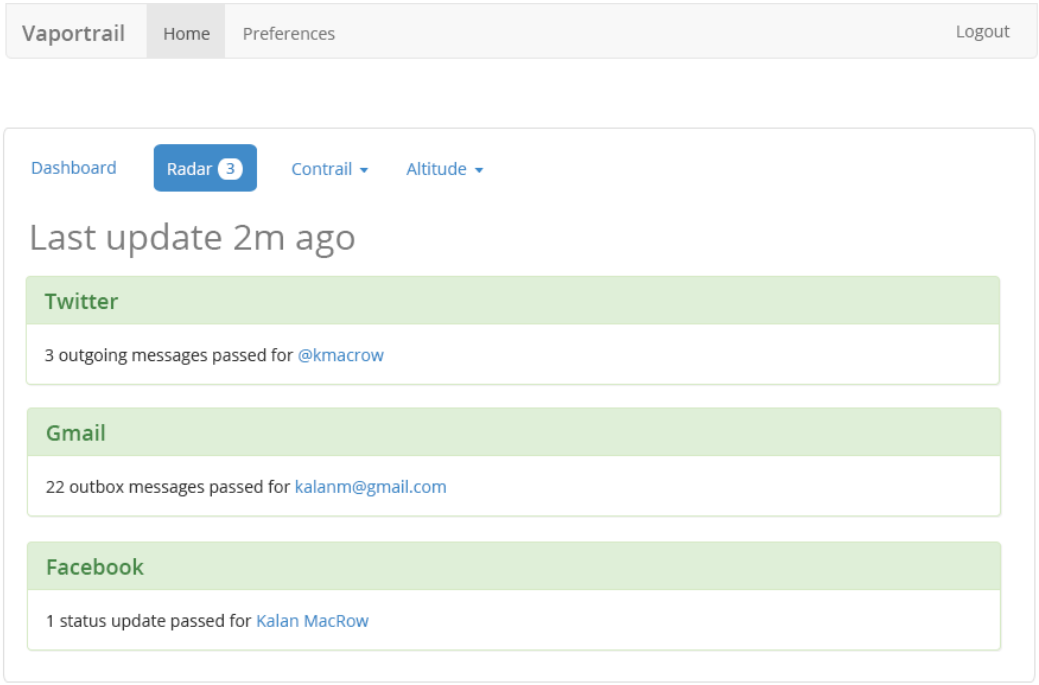

Figure 3.11: The *Radar* account fraud detection application.

score is multiplied by the number of matches and then a subtotal is calculated for the message. Alerts are raised based on two thresholds: *warning* and *critical*. At less than 100 lines of code, *Radar* demonstrates that useful monitoring applications can be developed for the platform with a minimum of effort.

3.4.5 Summary

The ability to run untrusted third-party applications on personal data is the core contribution of the platform. In this section we have reviewed how applications are packaged and shared, the high-level capabilities available to them, and their runtime lifecycle within the sandbox. We have also presented three prototype applications that illustrate the flexibility of the platform APIs and the ability of the platform to host interesting, non-trivial applications. We believe the application packaging and runtime implementation balances the design requirements of openness, flexibility and familiar development style with the need for robust privacy

controls.

3.5 Summary

In this section we have discussed the implementation of the Vaportrail platform prototype, including several example service connectors and applications. We have implemented a self-contained personal appliance that leverages Linux containers to isolate and network platform and third-party components within the appliance. We discuss how this architecture provides an open, robust, extensible and maintainable platform that is simple and inexpensive to operate. We argue that our choice of industry-proven abstractions (containers, RESTful APIs, databases) and careful decoupling of components will allow the appliance to operate without intervention over long periods of time, though a long-term user study would be required to verify this.

We discuss service connectors in Section 3.2, and the implementation challenges involved in ensuring developers have sufficient flexibility to integrate with a diverse array of external services, while retaining enough platform control to safely manage misbehaving connectors. We aim to balance safety and security with the networking capabilities required to build non-trivial integrations.

In Section 3.3 we discuss our application sandbox in detail. We build on ideas developed by Treehouse[44] and js.js[51] to implement a robust browser-based application sandbox capable of hosting untrusted JavaScript applications and enforcing policies and permissions through a platform monitor. We implement APIs in the sandbox environment that enable guest code to query data sources, render UI components and persist application state.

Finally, we discuss the platform implementation from the perspective of applications: how they are packaged and distributed, the programming environment, platform permissions and APIs. We present three example applications to demonstrate the ability of the platform to support interesting applications that combine personal data sources.

Although only a prototype, we believe the implementation succeeds in addressing the primary design goals of the system and would serve as a good starting point for a more production-ready implementation.

Chapter 4

Related Work

Vaportrail is a self-contained platform for personal data. The guiding philosophy behind Vaportrail is that it should be simple and inexpensive to operate while providing a safe and practical environment for running untrusted applications. This is largely in contrast to existing commercial products which may trade privacy for opportunities to monetize users, and also existing academic works that provide only a call for solutions, or present implementations that limit the likelihood of wider adoption by regular users.

4.1 Personal data platforms

There have been several commercially successful products that help a user aggregate their personal data into a single location and provide some value in the form of applications or integrations with other systems. If-This-Then-That[24] (IFTTT) provides a fully-managed SaaS platform that allows a user to select from over 4,000 "applets" that connect to external services and trigger various platform-mediated actions based on events in the data stream. Applets can have triggers on multiple service streams and generally provide simple, stateless utilities such as automatically re-posting content from one social network to another, or emailing a daily digest of activity. Unlike Vaportrail, IFTTT makes no claims about privacy and provides the service for free.

Cue[10] (formerly Greplin) allowed users to link several personal accounts into

a single dashboard and provides search capabilities over multiple data streams at once. Before being acquired by Apple, Cue did have a tiered pricing model in which users could pay for more storage capacity. Like IFTTT, Cue made no particular claims about privacy or data ownership once it was loaded into the platform.

Loggacy[26] represents a special class of personal data platforms designed specifically for establishing a digital legacy that can be shared across generations. Loggacy does not provide an application runtime, however, long term personal data archiving is a central capability of Vaportrail. We imagine establishing a digital legacy in a format that invites exploration and discovery as an important use case for Vaportrail. We also argue that as a self-hosted, privacy-preserving platform, Vaportrail is much better positioned to fulfil this long-term responsibility than a SaaS product that is subject to changeable business conditions.

The existence of several related products in this space, and indeed their ability to attract venture capital and buyers like Apple demonstrate the potential business value of consolidating personal data, as well as user interest in the capabilities it enables. Although generally lacking in privacy controls, we have taken inspiration from the user experience design, seamless integrations with the services people use, and the types of applications (or applets) they have developed.

Databox[42] proposed a trusted personal data platform for collating personal data, managing it, and providing controlled access to it. Although Databox did not present a specific implementation, we take considerable inspiration from their exploration of the design space and suggestions as to what a Databox should be. Although our implementation diverges in a few important ways from the Databox proposal (Vaportrail does not provide a mechanism for the user to monetize or broker their data out of the platform, nor does it allow applications to take copies of data, or for external devices to access the data over the network) Vaportrail does implement a subset of the Databox features. Furthermore, while the Databox authors envision the Databox as a core piece of infrastructure for the user, acting as a clearinghouse for sensitive data, we do not position Vaportrail on the user's critical path. Instead, Vaportrail is a trusted passive consumer of data providing a largely offline means of interacting with it. Databox suggests that an implementation would provide a means for users to explicitly trade their data in exchange for services, enabling targeted advertising as an incentive for developers. Coordinating

with a multitude of of external service providers to broker data (or "take money" as the Databox authors put it) is a much less tractable problem than building the core platform in relative isolation. For this reason, we leave the ability to act as a broker to future work.

4.2 Privacy-preserving web applications

PiBox[46] presents a privacy-preserving platform with design goals very similar to Vaportrail but with a significantly different implementation. Like Vaportrail, PiBox shifts the burden of establishing trust from applications to the platform itself, and provides a sandbox with communication, storage and control primitives that restrict application access to sensitive personal data. While Vaportrail employs a single user-controlled virtual appliance for deployment, the PiBox sandbox spans a user device and a virtual server hosted by one of a few trusted cloud providers. The platform uses differential privacy techniques to expose anonymized usage patterns to the application publisher to enable ad-driven revenue streams. We recognize that systems do not exist in an economic vacuum, but have elected to make the costs of hosting and application development as explicit as possible in Vaportrail: we argue that cloud infrastructure for a personal appliance is already sufficiently inexpensive for an individual, and that privacy-conscious users will pay for useful applications. The differences in architecture have important implications for the possibility of wider adoption as well. The PiBox sandbox is based on a modified Android kernel that manipulates existing IPC, filesystem and network isolation mechanisms to limit the capabilities of applications. It also assumes the willingness of large trusted service providers (e.g. Google, Apple) to host the cloud portion of the platform, acting as fiduciaries on the basis of their reputations. This is in stark contrast to Vaportrails sandbox which is platform independent (runs in the browser) and does not rely on adoption by large third-party vendors whose interests may not be well aligned with those of the individual. Although the motivating philosophy, trust model and types of applications that the platforms enable are very comparable, we believe Vaportrail is better positioned for real-world adoption.

Priv.io[52] is a platform for building and running privacy-preserving web services. In Priv.io, users pay for their share of cloud resources (storage, messaging

and bandwidth) and all computing is performed by applications in a browser sandbox. The authors argue that an always-on cloud server is too expensive for most people, based on an analysis of how much users would pay for services like Facebook and Twitter if they were built on Priv.io. Using strong encryption to ensure that plain text personal data is never exposed to the infrastructure provider, Priv.io can support a wide range of existing applications through a browser-based sandbox and API. With Vaportrail we do not aim for a general alternative deployment model for web services, nor to support existing services, but focus on a much narrower class of applications specifically built around creating value from personal data streams. Given the more specialized nature of Vaportrail as a tool for safely exploring data generated on other platforms, and not to host those platforms with stronger privacy controls as Priv.io does, we believe the Priv.io cost analysis does not translate well to Vaportrail. In terms of sandbox implementations, Priv.io takes an approach that balances the need for robust control over the flow of sensitive data into and out of the hosted applications with the flexibility to support existing browser-based code. They use an HTML iframe to host the untrusted application code and expose the platform API over the *postMessage()* interface while restricting the network access using a browser-enforced CSP. The advantage of this approach is that an iframe hosts a completely isolated and fully functional DOM, enabling existing applications to be easily ported to function in the sandbox. There are a couple of significant limitations: 1) the DOM is a very large and complex API implemented differently across browsers, making the CSP susceptible to circumvention by clever use of JavaScript or Cascading Style Sheets (CSS) DOM manipulations that trigger network requests. 2) as a complete web page, the untrusted content of the iframe is very capable of mounting phishing attacks to subvert the platform, i.e. by presenting UI that tricks the user into entering sensitive information that can then be exfiltrated through (1). For these reasons, and others discussed in Section 3.3, we prefer a more robust (albeit restrictive) approach that involves complete virtualization of the JavaScript runtime in which untrusted code runs. Consequently a successfull attack would require escaping a proven JavaScript engine, or a very narrow platform API.

4.3 JavaScript sandboxes

Treehouse[44] presents a JavaScript sandbox for running untrusted code that shares a number of similar goals to ours, namely that it should work without browser modifications and should be able to prevent guest code from directly manipulating the DOM and using network APIs. We take considerable inspiration from Treehouse, in particular the repurposing of WebWorkers as containers for executing untrusted code, but take a stronger (albeit less performant) approach to isolating guest code from the native browser APIs. Treehouse loads a "broker" into the WebWorker context that creates a virtual DOM and interposes itself on several sensitive browser APIs before loading and executing guest code. The virtual DOM allows sandboxed code to run largely unmodified as the broker asynchronously propagates changes into the real DOM via the WebWorker *postMessage()* interface, and routes events back into the virtual DOM. Fine-grained control over which browser objects, methods and arguments the sandbox code can use is specified by the user through policies, and enforced by the broker at runtime when the API is called. This ensures better performance than running a virtualized JavaScript interpreter as we do, but has an important limitation: the lack of standard API implementations across browsers means that it is difficult to verify, even for a single browser, that the broker has "locked down" and interposed on the entire API surface. Even if the JavaScript API has been secured, syncing virtual DOM manipulations into the real DOM enables an attacker to carefully construct DOM operations that, when applied outside the sandbox, could conceivably trigger network requests, inject code or present misleading UI elements. Instead of engaging in an API isolation "arms race", we take advantage of highly performant modern JavaScript engines and opt for a fully virtualized interpreter. Consequently Vaportrail cannot easily run unmodified applications, though our approach does not preclude exposing a virtual DOM compatibility layer as future work. Doing so would require very careful consideration of the DOM syncing implementation for the same reasons that apply to Treehouse.

Several other projects have investigated sandboxing or otherwise restricting the capabilities of client-side JavaScript. AdSafe[1], Caja[7] and FBJS[16] identify a "safe" subset of JavaScript that can be statically checked ahead of execution. There

are a number of iframe-based sandboxing implementations including AdJail[47], SMash[41] and Subspace[45]. These projects have provided inspiration for Vaportrail but often make trade-offs in favour of supporting existing applications, or supporting use cases (e.g. rendering ads) that are different from ours. JS.js[51], which we build on in our sandbox implementation, compiles an existing JavaScript interpreter to a subset of JavaScript, enabling it to run in modern browsers. JS.js incurs non-trivial (though acceptable) overhead and completely isolates sandboxed code from the browser JavaScript environment and APIs.

4.4 Summary

There is a considerable body of work related to aggregating and leveraging personal data streams, building privacy-preserving web applications and services, and sandboxing client-side browser JavaScript. We note that the commercial success of several personal data platforms (IFTTT, Cue, Loggacy) highlights the interest and, we argue, the need for an alternative that makes the costs and privacy controls explicit to the user. We review DataBox and its influence on our design of the Vaportrail prototype, as well as PiBox and Priv.io which attempt to solve similar problems by leveraging centralized fiduciaries, and peer-to-peer infrastructure sharing approaches, respectively. Finally we summarize client-side JavaScript sandboxing work, and Treehouse in particular for its influence on our implementation.

Chapter 5

Conclusions

We have described the design and implementation of Vaportrail, a platform for personal data and applications. Vaportrail provides a self-contained network appliance that individuals can use to archive their own personal data and safely run applications from untrusted third-party developers on their data. We have developed a simple trust model that puts the user in control of their data, as well as an architecture that implements this model with modern web-based ergonomics. We were inspired by several other privacy-preserving systems and built on them in an effort to design a platform that is open, extensible and practical.

Our aim with Vaportrail was to build a platform for personal data that, unlike commercial offerings, puts the user in control of their data and makes the operating costs explicit. Our hope is that by building Vaportrail using standard, open technologies that are readily deployable today, we can demonstrate that an alternative model to ad-driven Software-as-a-Service is practically feasible, and can deliver a modern user experience. We acknowledge that the economics of Vaportrail (and other privacy-conscious systems) are challenging: it remains to be seen whether the public will pay for privacy, however, without viable privacy-preserving alternatives we may never know. We argue that the decreasing costs, and increasing reliability, of cloud infrastructure are promising indications that at least storage and computing resources may not be a significant barrier now and in the future.

Although the current implementation is only an early-stage prototype, it demonstrates several of the core components of a personal data hub, as identified by

projects such as DataBox before us. The concept of a networked hub for personal data is not new, and we believe it could be an important primitive in a more privacy-aware future. Vaportrail is an attempt to refine these ideas in a modern design context and, hopefully, provide inspiration for others to continue the development of tools and systems to support privacy and personal data provenance. We observe similarities with public key infrastructures, which have only recently begun to enjoy wider consumer adoption as a result of ongoing refinement of the user experience around them, and less so of the core technology and ideas.

The near-term next steps for Vaportrail would be to produce a sufficiently stable version of the platform software to conduct a user study. In the longer-term, we would like to see Vaportrail extended to support private peer-to-peer data sharing primitives (perhaps via WebRTC[36]), enabling direct social sharing and fully decentralized services. We would also extend the application sandbox and runtime with graphics APIs to support data-driven 3D games and visualizations. Porting the js.js interpreter to run on WebAssembly would significantly improve the runtime performance of applications.

Vaportrail is far from having all of the features and properties that we would like from a personal data platform, however, we submit that building privacy-preserving software is difficult on several broad fronts: technically, economically, and socially. That being said, we feel that privacy is a worthwhile pursuit and that Vaportrail is a step towards a platform that puts users in control of their data, and gives developers the freedom to innovate using personal data sources.

Bibliography

[1] ADsafe: A safe JavaScript widget framework for advertising and other mashups. https://github.com/douglascrockford/ADsafe. Accessed: 2017-02-04. → pages 3, 58

[2] Amazon Echo and Alexa Devices. https://www.amazon.com/Amazon-Echo-And-Alexa-Devices/. Accessed: 2017-02-04. → pages 1

[3] AppArmor - Ubuntu wiki. https://wiki.ubuntu.com/AppArmor. Accessed: 2017-02-04. → pages 23

[4] asm.js. http://asmjs.org. Accessed: 2017-07-09. → pages 38

[5] Microsoft Azure Pricing. https://azure.microsoft.com/en-ca/pricing. Accessed: 2017-02-04. → pages 2, 21

[6] Bootstrap CSS. http://getbootstrap.com. Accessed: 2017-07-09. → pages 27, 43

[7] The LLVM Compiler Infrastructure. http://llvm.org. Accessed: 2017-07-09. → pages 46, 58

[8] Amazon CloudFormation. https://aws.amazon.com/cloudformation/. Accessed: 2017-02-04. → pages 20

[9] CoffeeScript. http://coffeescript.org. Accessed: 2017-07-09. → pages 46

[10] Cue Search Engine (Greplin). https://en.wikipedia.org/wiki/Cue_(search_engine). Accessed: 2017-02-04. → pages 54

[11] The Dart Programming Language. https://www.dartlang.org. Accessed: 2017-07-09. → pages 46

[12] Big Data, for better or worse: 90years. https://www.sciencedaily.com/releases/2013/05/130522085217.htm. Accessed: 2017-02-04. → pages 1, 2

[13] Dropbox. https://www.dropbox.com. Accessed: 2017-02-04. → pages 21

[14] Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing. Accessed: 2017-02-04. → pages 2, 21

[15] Emscripten: An LLVM-to-JavaScript Compiler. https://github.com/kripken/emscripten. Accessed: 2017-07-09. → pages 38

[16] Facebook for Developers. https://developers.facebook.com. Accessed: 2017-02-04. → pages 34, 58

[17] Gmail: Free Storage and Email from Google. https://www.google.com/gmail/about/. Accessed: 2017-02-04. → pages 2

[18] The GNU Privacy Guard. https://www.gnupg.org. Accessed: 2017-07-09. → pages 30

[19] Google Data APIs. https://developers.google.com/gdata/docs/directory, . Accessed: 2017-02-04. → pages 35

[20] Google Drive - Cloud Storage. https://www.google.com/drive/, . Accessed: 2017-02-04. → pages 21

[21] Google Home. https://madeby.google.com/home/, . Accessed: 2017-02-04. → pages 1

[22] Google Compute Platform Pricing. https://cloud.google.com/compute/pricing, . Accessed: 2017-02-04. → pages 2, 21

[23] Apple HomePod. https://www.apple.com/homepod/. Accessed: 2017-02-04. → pages 1

[24] If-This-Then-That. https://ifttt.com. Accessed: 2017-02-05. → pages 54

[25] The LLVM Compiler Infrastructure. http://llvm.org. Accessed: 2017-07-09. → pages 38

[26] Loggacy. https://www.loggacy.com. Accessed: 2017-02-04. → pages 55

[27] MongoDB. https://www.mongodb.com. Accessed: 2017-07-09. → pages 25

[28] OpenJDK. http://openjdk.java.net. Accessed: 2017-02-04. → pages 36

[29] PostgreSQL. https://www.postgresql.org. Accessed: 2017-07-09. → pages 25

[30] Redis in-memory data structure store. https://redis.io. Accessed: 2017-07-09. → pages 25

[31] Riak CS. https://github.com/basho/riak_cs. Accessed: 2017-07-09. → pages 25

[32] Amazon Simple Storage Service. https://aws.amazon.com/s3/. Accessed: 2017-02-04. → pages 21

[33] Twitter Developer Documentation. https://dev.twitter.com/rest/public. Accessed: 2017-02-04. → pages 35

[34] TypeScript: JavaScript that scales. https://www.typescriptlang.org. Accessed: 2017-07-09. → pages 46

[35] Web Intents. http://webintents.org, . Accessed: 2017-02-04. → pages 4, 13

[36] WebRTC: Browser Real-Time Communications. https://webrtc.org, . Accessed: 2017-02-04. → pages 4, 61

[37] Window.localStorage - Web APIs — MDN. https://developer.mozilla.org/en/docs/Web/API/Window/localStorage. Accessed: 2017-07-09. → pages 45

[38] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: An online social network with user-defined privacy. *SIGCOMM Comput. Commun. Rev.*, 39(4):135–146, Aug. 2009. ISSN 0146-4833. doi:10.1145/1594977.1592585. URL http://doi.acm.org/10.1145/1594977.1592585. Accessed: 2017-02-04. → pages 3

[39] T. Chajed, J. Gjengset, J. van den Hooff, M. F. Kaashoek, J. Mickens, R. Morris, and N. Zeldovich. Amber: Decoupling user data from web applications. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association. URL http://blogs.usenix.org/conference/hotos15/workshop-program/presentation/chajed. Accessed: 2017-02-04. → pages 3

[40] J. Constine. Facebook now has 2 billion monthly users... and responsibility. https://techcrunch.com/2017/06/27/facebook-2-billion-users/. Accessed: 2017-02-04. → pages 34

[41] F. De Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. Smash: Secure component model for cross-domain mashups on unmodified browsers. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 535–544, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-085-2. doi:10.1145/1367497.1367570. URL http://doi.acm.org/10.1145/1367497.1367570. Accessed: 2017-02-04. → pages 59

[42] H. Haddadi, H. Howard, A. Chaudhry, J. Crowcroft, A. Madhavapeddy, and R. Mortier. Personal data: Thinking inside the box. *CoRR*, abs/1501.04737, 2015. URL http://arxiv.org/abs/1501.04737. Accessed: 2017-02-04. → pages 17, 55

[43] D. Hardt. The OAuth 2.0 authorization framework. RFC 6749, RFC Editor, Fremont, CA, USA, Oct. 2012. URL http://www.rfc-editor.org/rfc/rfc6749.txt. Accessed: 2017-02-04. → pages 31

[44] L. ingram and M. Walfish. Treehouse: Javascript sandboxes to help web developers help themselves. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 153–164, Boston, MA, 2012. USENIX. ISBN 978-931971-93-5. URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/ingram. Accessed: 2017-02-04. → pages 3, 17, 37, 53, 58

[45] C. Jackson and H. J. Wang. Subspace: Secure cross-domain communication for web mashups. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 611–620, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-654-7. doi:10.1145/1242572.1242655. URL http://doi.acm.org/10.1145/1242572.1242655. Accessed: 2017-02-04. → pages 59

[46] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov. Box: A Platform for Privacy-Preserving Apps. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 501–514, Lombard, IL, 2013. USENIX. ISBN 978-1-931971-00-3. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lee_sangmin. Accessed: 2017-02-04. → pages 56

[47] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 24–24, Berkeley, CA, USA, 2010. USENIX Association. ISBN 888-7-6666-5555-4. URL http://dl.acm.org/citation.cfm?id=1929820.1929852. Accessed: 2017-02-04. → pages 59

[48] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=2600239.2600241. Accessed: 2017-02-04. → pages 19

[49] D. Recordon and D. Reed. Openid 2.0: A platform for user-centric identity management. In *Proceedings of the Second ACM Workshop on Digital Identity Management*, DIM '06, pages 11–16, New York, NY, USA, 2006. ACM. ISBN 1-59593-547-9. doi:10.1145/1179529.1179532. URL http://doi.acm.org/10.1145/1179529.1179532. Accessed: 2017-02-04. → pages 31

[50] T. Team. Twitter's Surprising User Growth Bodes Well For 2017. https://www.forbes.com/sites/greatspeculations/2017/04/27/twitters-surprising-user-growth-bodes-well-for-2017/. Accessed: 2017-07-09. → pages 34

[51] J. Terrace, S. R. Beard, and N. P. K. Katta. Javascript in javascript (js.js): Sandboxing third-party scripts. In *Presented as part of the 3rd USENIX Conference on Web Application Development (WebApps 12)*, pages 95–100, Boston, MA, 2012. USENIX. ISBN 978-931971-94-2. URL https://www.usenix.org/conference/webapps12/technical-sessions/presentation/terrace. Accessed: 2017-02-04. → pages 3, 38, 53, 59

[52] L. Zhang and A. Mislove. Building confederated web-based services with priv.io. In *Proceedings of the First ACM Conference on Online Social Networks*, COSN '13, pages 189–200, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2084-9. doi:10.1145/2512938.2512943. URL http://doi.acm.org/10.1145/2512938.2512943. Accessed: 2017-02-04. → pages 3, 16, 56

Appendix A

Platform APIs

In this section we provide a listing of platform APIs and components. The platform API surface consists of methods facing service connectors, applications, and internally (i.e. used only by the platform itself in the service of higher level APIs or functions.

platform.authn
The authentication endpoints handle authenticating service connectors and the web dashboard to establish a session that can be reused for subsequent requests.

- POST /authn/auth
  - Requires password or `VT_CONNECTOR_SECRET`
  - Returns access token that must be included in all subsequent requests using a custom `X-Auth-Token` HTTP header.

platform.authz
The authorization module determines whether a given API request is authorized for a previously authenticated client. Authorization logic wraps platform API server HTTP handlers and permission-restricted calls in the application monitor.

- `platform.permission.can(`*`permission`*`)`

  - Application facing API that allows applications to check whether the platform monitor would allow calls related to a specific permission, e.g. {`"type": "share", "content": "image/png"`}.

- POST /authz/can
  - An equivalent connector-facing API takes a permission in the request body.

platform.data

These APIs are used to write data to the platform data stores as well as query data from applications. `platform.query` provides a slightly higher level application-facing API.

- POST /data/write
  - Takes a data store schema and store-type specific write request and returns when the write is persistent or has failed.
- GET /data/read
  - Takes a data store schema and store-specific query request and returns data in a store-specific format (e.g. JSON list of rows, or a binary blob).

platform.meta

Platform metabase APIs are used largely internally for platform bookkeeping: storing and retrieving connector and application manifests, unpacking and parsing packages, setting up internal networks and creating and managing containers.

platform.ui

The platform application UI components. Component colours are restricted to a platform-defined palette. Similarly, sizing and positioning is restricted to prevent components from being abused.

- `Button` - simple button with text

- `Checkbox` - stateful checkbox button with label
- `Dropdown` - simple dropdown list of options
- `Input` - a text input field
- `Image` - static image label
- `Layer` - container that can be sized and styled within certain visual parameters
- `Label` - static text label
- `Link` - text label that appears as a clickable hyperlink
- `Panel` - a container with a heading and an outline
- `TableLayout` - grid-based layout container for other components
- `Tooltip` - simple tooltip with static text

platform.net

The platform application networking APIs.

- `get(`*`uri_id`*`)`
  - Fetches and returns content from a URI by ID specified in the application manifest. Note that the sandbox interpreter does not have an `eval()` function, making it very difficult to use this mechanism to load code.

platform.localStorage

The platform sandbox implementation of window.localStorage used by applications to persist state.

- `get(`*`key`*`)` : `String`
- `set(`*`key, Object`*`)` : `String`

platform.share

The application-facing APIs for triggering sharing intents. These largely wrap calls into `platform.data`.

- `share(`*`type`*`, `*`content`*`)`
  - Trigger a share request of *content* with content-type *type*.