Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services

Frank Wang, *MIT CSAIL;* Ronny Ko and James Mickens, *Harvard University*

https://www.usenix.org/conference/nsdi19/presentation/wang-frank

This paper is included in the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).

February 26–28, 2019 • Boston, MA, USA

ISBN 978-1-931971-49-2

Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services

Frank Wang
MIT CSAIL

Ronny Ko, James Mickens
Harvard University

Abstract

Riverbed is a new framework for building privacy-respecting web services. Using a simple policy language, users define restrictions on how a remote service can process and store sensitive data. A transparent Riverbed proxy sits between a user's front-end client (e.g., a web browser) and the back-end server code. The back-end code remotely attests to the proxy, demonstrating that the code respects user policies; in particular, the server code attests that it executes within a Riverbed-compatible managed runtime that uses IFC to enforce user policies. If attestation succeeds, the proxy releases the user's data, tagging it with the user-defined policies. On the server-side, the Riverbed runtime places all data with compatible policies into the same universe (i.e., the same isolated instance of the full web service). The universe mechanism allows Riverbed to work with unmodified, legacy software; unlike prior IFC systems, Riverbed does not require developers to reason about security lattices, or manually annotate code with labels. Riverbed imposes only modest performance overheads, with worst-case slowdowns of 10% for several real applications.

1 Introduction

In a web service, a client like a desktop browser or smartphone app interacts with datacenter machines. Although smartphones and web browsers provide rich platforms for computation, the core application state typically resides in cloud storage. This state accrues much of its value from server-side computations that involve no participation (or explicit consent) from end-user devices.

By running the bulk of an application atop VMs in a commodity cloud, developers receive two benefits. First, developers shift the burden of server administration to professional datacenter operators. Second, developers gain access to scale-out resources that vastly exceed those that are available to a single user device. Scale-out storage allows developers to co-locate large amounts of data from multiple users; scale-out computation allows developers to process the co-located data for the benefit of users (e.g., by providing tailored search results) and the benefit of the application (e.g., by providing targeted advertising).

1.1 A Loss of User Control

Unfortunately, there is a disadvantage to migrating application code and user data from a user's local machine to a remote datacenter server: the user loses control over where her data is stored, how it is computed upon, and how the data (and its derivatives) are shared with other services. Users are increasingly aware of the risks associated with unauthorized data leakage [11, 62, 82], and some governments have begun to mandate that online services provide users with more control over how their data is processed. For example, in 2016, the EU passed the General Data Protection Regulation [28]. Articles 6, 7, and 8 of the GDPR state that users must give consent for their data to be accessed. Article 17 defines a user's right to request her data to be deleted; Article 32 requires a company to implement "appropriate" security measures for data-handling pipelines. Unfortunately, requirements like these *lack strong definitions and enforcement mechanisms at the systems level.* Laws like GDPR provide little technical guidance to a developer who wants to comply with the laws while still providing the sophisticated applications that users enjoy.

The research community has proposed information flow control (IFC) as a way to constrain how sensitive data spreads throughout a complex system [35, 42]. IFC assigns labels to program variables or OS-level resources like processes and pipes; given a partial ordering which defines the desired security relationships between labels, an IFC system can enforce rich properties involving data secrecy and integrity. Unfortunately, traditional IFC is too burdensome to use in modern, large-scale web services. The reason is that creating and maintaining a partial ordering of labels is too difficult—the average programmer or end-user struggles to reason about data safety via the abstraction of fine-grained label hierarchies. As a result, no popular, large-scale web service uses IFC to restrict how sensitive data is processed and shared.

1.2 Our Solution: Riverbed

In this paper, we introduce Riverbed, a distributed web platform for safeguarding the privacy of user data. Riverbed provides benefits to both web developers and end users. To web developers, Riverbed provides a *practical* IFC system which

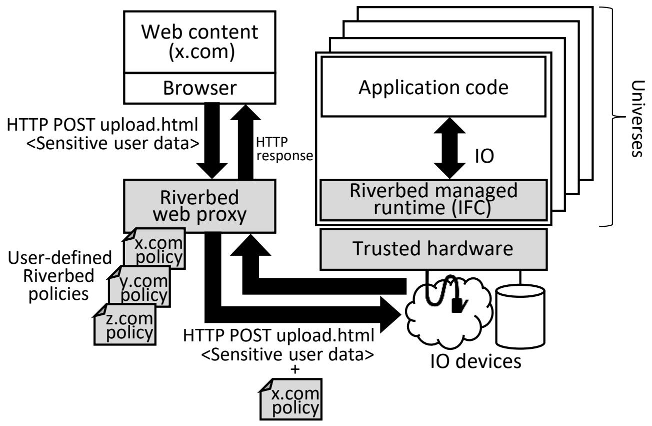
Web content (x.com)
Browser
HTTP POST upload.html <Sensitive user data>
HTTP response
Riverbed web proxy
User-defined Riverbed policies
x.com policy
y.com policy
z.com policy
HTTP POST upload.html <Sensitive user data> + x.com policy
Application code
IO
Riverbed managed runtime (IFC)
Trusted hardware
IO devices
Universes

Figure 1: Riverbed's architecture. The user's client device is on the left, and the web service is on the right. Unmodified components are white; modified or new components are grey.

allows developers to easily "bolt on" stronger security policies for complex applications written in standard managed languages. To end users, Riverbed provides a straightforward mechanism to verify that server-side code is running within a privacy-preserving environment.

Figure 1 describes Riverbed's architecture. For each Riverbed web service, a user defines an information flow policy using simple, human-understandable constraints like "do not save my data to persistent storage" or "my data may only be sent over the network to `x.com`." In the common case, users employ predefined, templated policy files that are designed by user advocacy groups like the EFF. When a user generates an HTTP request, a web proxy on the user's device transparently adds the appropriate data flow policy as a special HTTP header.

Within a datacenter, Riverbed leverages the fact that many services run atop managed runtimes like Python, .NET, or the JVM. Riverbed modifies such a runtime to automatically taint incoming HTTP data with the associated user policies. As the application derives new data from tainted bytes, the runtime ensures that the new data is also marked as tainted. If the application tries to externalize data via the disk or the network, the externalization is only allowed if it is permitted by user policies. The Riverbed runtime terminates an application process which attempts a disallowed externalization.

In Riverbed, application code (i.e., the code which the managed runtime executes) is totally unaware that IFC is occurring. Application developers have no way to read, write, create, or destroy taints and data flow policies. The advantage of this scheme is that it makes Riverbed compatible with code that has not been explicitly annotated with traditional IFC labels. However, different end users will likely define incompatible data flow policies. As a result, policy-agnostic code would quickly generate a policy violation for some subset of users; Riverbed would then terminate the application. To avoid this problem, Riverbed spawns multiple, lightweight copies of the back-end service, one for each set of users who share the same data flow policies. We call each copy a *universe*. Since users in the same universe allow the same types of data manipulations, any policy violations indicate true problems with the application (e.g., the application tried to transmit sensitive data to a server that was not whitelisted by the inhabitants of the universe).

Before a user's Riverbed proxy sends data to a server, the proxy employs remote attestation [9, 15] to verify that the server is running an IFC-enforcing Riverbed runtime. Importantly, a trusted server will perform *next-hop attestation*—the server will not transmit sensitive data to another network endpoint unless that endpoint is an attested Riverbed runtime whose TLS certificate name is explicitly whitelisted by the user's data flow policy. In this manner, Riverbed enables controlled data sharing between machines that span different domains.

1.3 Our Contributions

To the best of our knowledge, Riverbed is the first distributed IFC framework which is practical enough to support large-scale, feature-rich web services that are written in general-purpose managed languages. Riverbed preserves the traditional advantages of cloud-based applications, allowing developers to offload administrative tasks and leverage scale-out resources. However, Riverbed's universe mechanism, coupled with a simple policy language, provides users with understandable, enforceable abstractions for controlling how datacenters manipulate sensitive data. Riverbed makes it easier for developers to comply with laws like GDPR—users give explicit consent for data access via Riverbed policies, with server-side universes constraining how user data may be processed, and where its derivatives can be stored.

We have ported several non-trivial applications to Riverbed, and written data flow policies for those applications. Our experiments show that Riverbed enforces policies with worst-case end-to-end overheads of 10%. Riverbed also supports legacy code with little or no developer intervention, making it easy for well-intentioned (but average-skill) developers to write services that respect user privacy.

2 Related Work

In this section, we compare Riverbed to representative instances of prior IFC systems. At a high level, Riverbed's innovation is the leveraging of universes and human-understandable, user-defined policies to enforce data flow constraints in IFC-unaware programs. Riverbed enforces these constraints without requiring developers to add security annotations to source code.

2.1 Explicit Labeling

In a classic IFC system, developers explicitly label program state, and construct a lattice which defines the ways in which differently-labeled state can interact. Roughly speaking, a program is composed of assignment statements; the IFC system only allows a particular assignment if all of the policies involving righthand objects are compatible with the policies of the lefthand side.

IFC-visible assignments can be defined at various levels of granularity. For example, Jif [48], Fabric [45], and similar frameworks [14, 29, 79, 80] modify the compiler and runtime for a managed language, tracking information flow at the granularity of individual program variables. In contrast, frameworks like Thoth [25], Flume [39], Camflow [50], and DStar [81] modify the OS, associating labels with processes, IO channels, and OS-visible objects like files. Taint can be tracked at even high levels of abstraction, e.g., at the granularity of inputs and outputs to MapReduce tasks [66].

All of these approaches require developers to reason about a complex security lattice which captures relationships between a large number of privileges and privilege-using entities like users and groups. Porting a complex legacy application to such a framework would be prohibitively expensive, and to the best of our knowledge, there is no large-scale, deployed system that was written from scratch using IFC with explicit labeling. Developer-specified labels are also a poor fit for our problem domain of *user-specified* access policies.

Tracking data flows at too-high levels of abstraction can introduce problems of overtainting—to avoid false negatives, systems must often use pessimistic assumptions about how outputs should be tainted. For different reasons, overtainting is also a challenge for ISA-level taint tracking [27, 65], For example, if taint is accidentally assigned to `%ebp` or `%esp`, then taint will rapidly propagate throughout the system, yielding many false positives [67]. To avoid these problems, Riverbed taints at the managed runtime level, a level which does not expose raw pointers, and defines data types with less ambiguous tainting semantics.

In Jeeves [7, 78], a developer explicitly associates each sensitive data object with a high-confidentiality value, a low-confidentiality value, and a policy which describes the contexts in which a particular value should be exposed. An object's value is symbolic until the object is passed to an output sink, at which point Jeeves uses the context of the sink to assign a concrete value to the object. Riverbed avoids the need for developers to label objects with policies or concrete values with different fidelities; via the universe mechanism, Riverbed applications always compute on high-fidelity data while satisfying user-defined constraints on data propagation.

2.2 Implicit Labeling

Some IFC systems use predefined taint sources and IFC policies. For example, TaintDroid [26] uses a modified JVM to track information flows in Android applications. TaintDroid predefines a group of sensors and databases that generate sensitive data; examples of these sources include a smartphone's GPS unit and SMS database. The only sink of interest is the network, because TaintDroid's only goal is to prevent sensitive information from leaking via the network. Because TaintDroid uses a fixed, application-agnostic set of IFC rules, TaintDroid works on unmodified applications. Riverbed also works on unmodified applications. However, TaintDroid operates on a single-user device, whereas Riverbed targets a web service that has many users, each of whom may have unique preferences for how their data should be used. Thus, Riverbed requires users (but not developers or applications) to explicitly define information flow policies. Riverbed also requires the universe mechanism (§4.4) to prevent the mingling of data from users with incompatible flow policies.

2.3 Formal Verification

IronClad [33] servers, like Riverbed servers, use remote attestation to inform clients about the server-side software stack. In Ironclad, server-side code is written in Dafny [41], a language that is amenable to static verification of functional correctness. Nothing prevents Riverbed from executing formally-verified programs; however, Riverbed's emphasis on running complex code in arbitrary managed languages means that Riverbed is generally unable to provide formal assurances about server-side code.

3 Threat Model

Riverbed assumes that developers want to enforce user-defined privacy policies, but are loathe to refactor code to do so. Thus, Riverbed assumes that server-side code is weakly adversarial: poorly-designed applications may unintentionally try to leak data via explicit flows, but developers will not intentionally write code that attempts to surreptitiously leak data, e.g., via implicit flows, or via targeted attacks on the taint-tracking managed runtime. Riverbed is compatible with mechanisms for tracking implicit flows [5, 6, 10, 61], but our Riverbed prototype does not track them for several reasons. One reason is that the punitive aspects of laws like the GDPR disincentivize companies from writing code that intentionally subverts compliance mechanisms like Riverbed. Furthermore, in many common programming languages, mechanisms for detecting implicit flows have undesirable properties like flagging some well-behaved programs as malicious [5], or requiring annotations from developers [6]. Riverbed strives for compatibility with legacy, non-annotated code written in popular languages.

A datacenter operator has physical access to servers, which enables direct manipulation of server RAM. So, our current Riverbed prototype assumes that datacenter operators are trusted. To ease this assumption, Riverbed could leverage a hardware-enforced isolation mechanism like SGX [20, 37]. However, SGX places limits on the memory size of secure applications. SGX also requires the applications to run in ring 3, forcing the code to rely on an untrusted OS in ring 0 to perform IO; the result is a large number of context switches for applications that perform many IOs [8]. Riverbed strives to be compatible with complex applications that issue frequent IOs. Thus, our Riverbed prototype eschews mechanisms like SGX, and must be content with not protecting against activelymalicious datacenter operators. To implement remote attestation, Riverbed does rely on tamper-resistant, server-side TPM hardware (§4.3), but TPMs do not protect against physical attacks on the rest of the server hardware.

Riverbed assumes that the entire client-side is trusted, with the exception of the web content in a particular page. Buggy or malicious content may try to disclose too much information to a server. However, Riverbed ensures that whatever data is sent will be properly tagged. Since Riverbed uses TLS to authenticate network endpoints, the HTTPS certificate infrastructure must be trusted.

On a server, Riverbed's TCB consists of a taint-tracking managed runtime, a reverse proxy that forwards requests to the appropriate universes (§4.4), the TPM hardware that provides the root of trust for attestation, and a daemon which servers use to attest to clients. We make standard cryptographic assumptions about the strength of the ciphers and hash functions used by the attestation protocol. Between the TPM hardware and the managed runtime are a boot loader, a hypervisor, and other systems software. Each end-user can choose a different collection of intermediate software to trust. A user's preferences are expressed in her policies (§4.2), so that Riverbed's client-side proxy can refuse to disclose data to untrusted server-side systems code.

4 DESIGN

Figure 1 provides a high-level overview of the Riverbed architecture. In this section, we provide more details on how users specify their policies, and how Riverbed enforces those polices on the server-side.

4.1 Riverbed-amenable Services

Riverbed is best-suited for certain types of web services.

- **Services with per-user silos for application state, and no cross-user sharing:** Examples include back-up services like Ionic [51], and private note-taking apps like Turtl [47]. Riverbed prevents information leakage between per-user silos (although an individual silo may span multiple server-side hostnames and cloud providers).
- **Services that silo user data according to explicitly-defined group affinities:** For example, a social networking site can create a universe for the state belonging to a corporation's private group. The corporation's users map to the same Riverbed user (§4.2), with no data flows between different corporations. Financial analysis sites and email services can use this decomposition to isolate data belonging to a particular business or social group.
- **Services which aggregate unaffiliated users by shared polices:** For example, in a news site, users can define policies that impact whether the site may aggregate user data for targeted advertising. Riverbed places users with equivalent policies into the same universe, ensuring that the site respects each user's preferences.

```
USER-ID: ALICE
AGGREGATION: False
PERSISTENT-STORAGE: True
ALLOW-TO-NETWORK: x.com
ALLOW-TO-NETWORK: y.com
TRUSTED-SERVER-STACK: {
   83145c082bbf608989f05e85c3c211f83,
   c8cd7ac93cab2b94f65a5b2de5709767f,
       ...
   590f01d8d18b1141988ee1975b3ce3b30
}
```

Figure 2: An example of a Riverbed policy. For simplicity, we elide graph-based contextual attestation predicates (§4.3).

Child policies (§4.2) can whitelist communication between server-side endpoints with otherwise incompatible policies. However, such whitelisting is easier when the server-side application consists of small, well-defined components, so that whitelisting individual components has well-understood security implications.

4.2 Expressing Policies

Figure 2 provides an example of a Riverbed policy. A policy consists of several parts, as described below.

The `USER-ID` field describes the owner of the policy. User ids only need to be unique within the context of a particular web service. Riverbed is agnostic about the mechanism that a service uses to authenticate users and log them into the service. However, Riverbed's server-side reverse proxy must know who owns the policy that is associated with each user request, so that the proxy can forward the request to the appropriate universe (§4.4).

Since Riverbed is agnostic about a service's login mechanism, a `USER-ID` field could actually be bound to a group of users. In this scenario, the users in the group would have different service-specific usernames, but share the same `USER-ID` field in their Riverbed policies. From Riverbed's perspective, the sensitive data of each individual user would all belong to a single logical Riverbed user.

The `AGGREGATION` flag specifies whether a user's data can be involved in server computations that include the data of other users. For example, suppose that a server wishes to add two numbers, each of which was derived from the data of a different user. If both users allow aggregation, Riverbed can execute the addition in the same universe. If one or both users disallow aggregation, then Riverbed must create separate universes for the two users. The `AGGREGATION` field specifies a yes/no policy—either arbitrary aggregation is allowed, or all aggregation is disallowed.

The binary `PERSISTENT-STORAGE` flag indicates whether server-side code can write a user's data to persistent storage. If so, the user expects that when the data is read again by the server-side application, the application will treat the data as tainted. A Riverbed managed runtime terminates applications that try to write tainted data to persistent storage, but lack the appropriate permissions.

A policy can optionally include an email address that belongs to the policy owner. If a Riverbed managed runtime must terminate policy-violating code, Riverbed can email the policy owner, informing the user about the thwarted policy breach. The user can then complain to the service operator, or take another corrective action.

The `ALLOW-TO-NETWORK` field is optional, and allows a user to whitelist network endpoints to which user data may flow. Endpoints are represented by hostnames; each whitelisted hostname is expected to have a valid X.509 certificate, e.g., as used by HTTPS. Before a Riverbed managed runtime allows tainted data to externalize via a socket, the runtime will check whether the remote endpoint is whitelisted by the tainted data's policy. If so, the runtime forces the remote endpoint to attest its software stack. If that stack is whitelisted by the policy, the runtime allows the transfer to complete. Otherwise, the runtime terminates the application. Note that Riverbed allows untainted data to be sent to arbitrary remote servers.

The final item in a policy is typically one or more `TRUSTED-SERVER-STACK` entries. Each trusted stack is represented by a list of hash values; see Section 4.3 for more details about how these hash values are generated by servers, and later consumed by the attestation protocol.

As discussed in Sections 4.3 and 4.6, a client-side proxy leverages attestation to validate the server-side software stack up to, but not including, the application-defined managed code. Once the proxy determines that Riverbed's taint-tracking managed runtime is executing on the server, the proxy will trust the runtime to enforce the policies described earlier in this section. However, the policies from earlier in this section only enable aggregation at a binary granularity (i.e., "allowed" or "disallowed"); a universe which disallows aggregation can never permit data to flow to a universe which *does* allow aggregation. This restriction prevents several useful types of selective aggregation. For example, two email servers in separate no-aggregate universes could ideally send emails to a trusted spam filter application which trains across all inboxes, and then returns a filter to each universe. To allow such aggregation by explicitly trusted components, Riverbed policies can decorate an `ALLOW-TO-NETWORK` field with a child policy. The child policy can override settings in the parent policy, allowing aggregation to occur at the endpoint. The child policy must specify a full-stack attestation record, to allow Riverbed to verify the identity of a *particular type* of trusted application-level code (e.g., SpamAssassin [2]). Data received from a trusted aggregator is marked with the taint descriptor of the receiving universe.

Riverbed allows a user to define her own policy for each web service that she uses. However, some policies may be fundamentally incompatible with certain services. As a trivial example, a Dropbox-like service that provides online storage is intrinsically incompatible with a `PERSISTENT-STORAGE: False` policy.[1] In the common case, we expect users to rely on trusted outside authorities, called *policy generators*, to define reasonable policies for sites. For example, consider a web site that wants to deliver targeted advertising via a third-party ad network `evil-ads.com`. A consumer advocacy group can advise users to avoid policies that whitelist `evil-ads.com`. Consumer advocacy groups can also publish suggested policy files for particular sites, based on research about what reasonable permissions for those sites should be.

Note that modern web browsing is already influenced by a variety of curated policies. For example, Google maintains a set of known-malicious URLs; multiple browser types consult this list to prevent accidental user navigation to attacker-controlled pages [31]. As another example, ad blockers [18] interpose on a page load, blocking content from sites deemed objectionable by the creators of the ad blocker. Riverbed introduces a new kind of web policy, but does not shatter prior expectations that web browsing must be an unmediated experience.

4.3 Server Attestation

The client-side proxy shepherds the interactions between the client and server portions of a Riverbed application. In this section, we describe the proxy in the context of a traditional web service whose client/server protocol is HTTP. Proxies are easily written for other protocols like SMTP (§4.5). We assume that the reader understands the basics of remote attestation, but readers who lack this knowledge can refer to the appendix for the necessary background material.

A user configures her browser to use the Riverbed proxy to connect to the Internet. At start-up time, the proxy searches a well-known directory for the user's policy files; the proxy assumes that each filename corresponds to the hostname in a server-side X.509 certificate (e.g., `x.com`). When the proxy receives an HTTP request that is destined for `x.com`, the proxy opens a TLS connection to `x.com`'s server, and forces that server to remotely attest its software stack. If the attestation succeeds, the proxy issues the HTTP request that triggered the attestation. Later, upon receiving a response from the server, the proxy forwards the response to the browser. By default, the proxy assumes that an attestation is valid for one day before a new attestation is necessary.

Riverbed strives to be practical, but traditional remote attestation [9, 15] has some unfortunate practical limitations. Consider the following challenges.

Server-side ambiguity: In traditional attestation, servers establish trust with clients by providing an explicit list of server-side software components. However, servers may not wish to share a perfectly-accurate view of their local software environment. For example, servers might be concerned that a

[1]...unless the service is intentionally exporting a RAM-only storage abstraction.

malicious client will launch zero-day attacks against vulnerable (and precisely-identified) server components.

Potentially safe code: A server-side component may be intrinsically secure, but currently unvetted by the creator of a user's Riverbed policies. Alternatively, a server-side executable might be intrinsically insecure, but perfectly safe to run if launched within a sandboxed environment like a virtual machine. Traditional attestation protocols are ill-suited to handle cases like these, since trust decisions are binary—a hash value in an attestation message corresponds to a categorically trusted component, or a categorically untrusted component.

Policy updating: A virtuous server administrator will be diligent about applying the latest patches to server-side code. If the user's policy generator is not as diligent, then users will reject legitimately trustworthy stacks as suspicious. Similarly, if users are more aggressive about updating policies than a server administrator, then out-of-date server-side stacks will be legitimately rejected as untrustworthy, but the server administrator will lack an immediate explanation for why. Traditional attestation protocols focus on the cryptographic aspects of client-server communication, but cannot resolve these kinds of policy disputes.

Riverbed uses the Cobweb attestation system [74] to handle these practical concerns. In traditional attestation, the attestor sends a TPM-signed `PCR[10]` value, and a list of <*filename, filehash*> tuples representing the objects that are covered by the cumulative hash in `PCR[10]`. Cobweb allows the attestor to augment the traditional attestation report with a *contextual graph* that provides additional information about the attestor's software stack. For example, a contextual graph might represent a process tree, where each vertex is a process and each edge represents a parent/child `fork()` relationship. An edge could also represent a dynamic information flow, e.g., indicating that two processes have communicated via IPC. Attestation verifiers specify policies as graph predicates that look for desired structural properties in the contextual graph or the regular attestation list of <*filename, filehash*> tuples.

Riverbed uses contextual graphs, and policy specification via graph predicates, to eliminate some of the practical difficulties with traditional attestation. For example:

- If attestation fails (i.e., if a client-side Riverbed proxy discovers that a graph predicate cannot be satisfied), the proxy sends the failed predicate to the server. The server can then initiate concrete remediating steps, e.g., by updating software packages, or removing a blacklisted application.
- A Riverbed server can also dispute the failure of a graph predicate. For example, if a user's proxy believes that a particular server-side component is out-of-date, the server can respond with a list of signed, vendor-supplied updates for which the user's proxy may be unaware. The proxy can then ask the user's policy generator for a new policy.

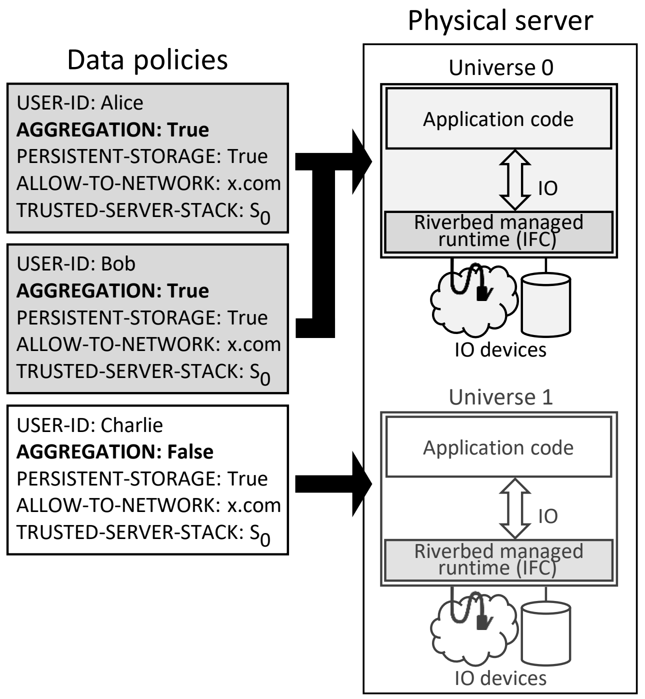

Figure 3: Alice and Bob have compatible policies, so Riverbed maps them to the same universe. Charlie has an incompatible policy because he disallows aggregation. Thus, Charlie must receive his own universe.

- A user's Riverbed policy can tolerate an unknown or normally untrusted server binary if that binary is launched within a sandbox that isolates the component from other components which the user does require to be trusted. To provide confidence in the sandbox, the server's contextual graph should contain the `fork()/exec()` history for the server, as well as the configuration files for the sandbox environment. As a concrete example, suppose that a server needs to run a `telnet` daemon to communicate with a legacy internal service. The `telnet` protocol is known to be insecure, but a Riverbed proxy can trust the server's Apache instance if the server uses a virtual machine or a Docker container to isolate `telnetd`.

Riverbed also leverages Cobweb's support for server-side software ambiguity, but we refer the reader to the Cobweb paper [74] for a discussion of how Cobweb implements this feature.

4.4 Universes

Consider Alice, Bob, and Charlie, three Riverbed users whose policies are shown in Figure 3. The policies of these users are *almost* the same—they differ only with respect to the `AGGREGATION` token. Alice and Bob allow aggregation, but Charlie does not. How should Riverbed handle the data of these users on the server-side?

Riverbed could optimistically assume that the server-side application code will never try to aggregate Charlie's data with that of Alice or Bob. Riverbed executes the code atop a taint-tracking runtime (§4.5), so Riverbed could synchronously detect attempted violations of Charlie's policy. Unfortunately, attempted violations are likely, since Riverbed executes unmodified applications that are unaware of Riverbed policies. If a violation occurs, Riverbed would lack good options for moving forward. Riverbed could permanently terminate the application, which would prevent the disallowed aggregation of Charlie's data. However, all three users would be locked out of the now-dead service. To avoid this outcome, Riverbed could try to synchronously clone the application at policy violation time, creating two different versions: one for Alice and Bob, and another for Charlie. However, determining which pieces of in-memory and on-disk state should belong in which clone is difficult without application-specific knowledge; a primary goal of Riverbed is to enforce security in a service-agnostic manner.

Riverbed's solution emerges from the insight that Riverbed does not need to run any code to determine whether a set of policies might conflict. Instead, Riverbed can simply examine the policies themselves. For example, if a policy does not allow aggregation, then Riverbed can *preemptively* spawn a separate copy of the service for the policy's owner. Riverbed can spawn this copy on-demand, upon receiving the first request from the owner. Now consider a policy *P* that allows aggregation and a particular set of storage and network permissions (e.g., `PERSISTENT-STORAGE: True` and `ALLOW-TO-NETWORK: x.com`). All users whose policies match *P* can be placed in the same copy of the service. Riverbed can spawn the copy upon receiving the first request that is tagged with *P*.

We call each service copy a *universe*. To implement the universe mechanism, Riverbed places a reverse proxy in front of the actual servers which run application code. Clients send their requests to the reverse proxy; the reverse proxy examines the policy in each request, spawns a new universe if necessary, and then forwards the request to the appropriate universe. Our Riverbed prototype instantiates each universe component inside of a Docker [21] instance that contains a taint-tracking runtime (§4.5) and the component-specific code and data. Docker containers are much smaller than traditional virtual machines since Docker virtualizes at the POSIX layer instead of the hardware layer. As a result, creating, destroying, and suspending universes is fast (§6.4). Docker runs each container atop a copy-on-write file system that belongs to the host [24]. Thus, universes share the storage that is associated with application code and other user-agnostic files.

Universes provide a final advantage: since all of the sensitive data in a universe has the same policy, a universe's taint-tracking runtime only needs to associate a single logical bit of taint with each object ("tainted" or "untainted"). If data from all users resided in the same universe, the runtime would have to associate each object with a value that represented a specific taint pattern.

The relationship between the number of users and the number of universes is application-specific. Some web services will specifically target a 1-1 mapping. For example, in a "private Dropbox" service that implements confidential online storage, users will naturally specify data policies that prevent aggregation (and thus require a universe per user). In contrast, social networking applications intrinsically derive their value from the sharing of raw user data, and the extraction of interesting cross-user patterns. For these applications, users must allow aggregation (although the scope of aggregation can be restricted using groups (§4.2)).

4.5 Taint Tracking

A managed language like Python, Go, or Java does not expose raw pointers to applications, or allow those applications to directly issue system calls. Instead, the language runtime acts as a mediation layer, controlling how a program interacts with the outside world. Like much of the prior work on dynamic tainting [26, 34, 46, 13], Riverbed enforces information flow control inside the managed runtime. Our Riverbed prototype modifies PyPy [53], a state-of-the-art Python interpreter, to extract Riverbed policies from incoming network data, and assign taint to derived information.

PyPy translates Python source files to bytecodes. Those bytecode are then interpreted. Riverbed adds taint-tracking instrumentation to the interpreter, injecting propagation rules that are similar to those of TaintDroid [26]. For example, in a binary operation like `ADD`, the lefthand side of the assignment receives the union of the taints of the righthand sides. Assigning a constant value to a variable clears the taint of the variable. If an array element is used as a righthand side, the lefthand side receives the taint of both the array and the index.

We call Riverbed's modified Python runtime PyRB. If a Python application tries to send tainted data to remote host `x.com`, PyRB first checks whether externalization to `x.com` is permitted by the tainted data's policy. If so, PyRB forces `x.com` to remotely attest its software stack; in this scenario, PyRB acts as the client in the protocol from Section 4.3. If `x.com`'s stack is trusted by the tainted data's policy, PyRB allows the data to flow to `x.com`. Otherwise, PyRB terminates the application. Riverbed provides a standalone attestation daemon that a server can use to respond to attestation requests.

PyRB must also taint incoming network data that was sent by end-user clients like web browsers. To do so without requiring modifications to legacy application code, PyRB assumes two things. First, clients are assumed to use standard network protocols like HTTP or SMTP. Second, PyRB assumes that when clients send requests using those protocols, clients embed Riverbed policies in a known way. Ensuring the second property is easy if unmodified clients run atop Riverbed proxies; for example, when an unmodified web browser sends requests through a client-side Riverbed proxy, the proxy will automatically embed Riverbed policies using the `Riverbed-policy` HTTP header. Similarly, a client-side SMTP proxy can attach Riverbed policies using a custom SMTP command.

On the server-side, PyRB assumes that traffic intended for well-known ports uses the associated well-known protocol. Upon receiving a connection to such a port, PyRB reads the initial bytes from the socket before passing those bytes to the application. If the initial bytes cannot be parsed as the expected protocol, PyRB forcibly terminates the connection. Otherwise, if PyRB finds a Riverbed policy, PyRB taints the socket bytes and then hands the tainted bytes to the higher-level application code. If there is no policy attached to the bytes, PyRB hands untainted bytes to the higher-level code. Importantly, the application code is unaware of the tainting process, and cannot read or write the taint labels.

If policies allow server-side code to write to persistent storage, PyRB taints the files that the application writes. PyRB does whole-file tainting, storing taint information in per-file extended attributes [40]. PyRB prevents application code from reading or writing those attributes. Whole-file tainting minimizes the storage overhead for taints, but Riverbed is compatible with taint-aware storage layers (§6.2) that perform fine-grained tainting, e.g., at the level of individual database rows; the use of such storage layers will minimize the likelihood of overtainting.

When an application reads data from a tainted file, PyRB taints the incoming bytes, preventing the application from laundering taint through the file system. Note that, even though a policy contains multiple constraints (§4.2), all of the users within a universe share the same policy; thus, PyRB only needs to associate a single logical bit with each Python object (§4.4). PyRB does need to store one copy of the full policy, so that the policy can be consulted when tainted data reaches an output sink.

Managed languages sometimes offer "escape hatches" that allow an application to directly interact with the unmanaged world. For example, in Java, the JNI mechanism [49] enables applications to invoke code written in native languages like C. In Python, interfaces like `os.system()` and `subprocess.call()` allow managed code to spawn native binaries. A Riverbed runtime can use one of three strategies to handle a particular escape hatch.

- The runtime can disallow the escape hatch by fiat.
- Alternatively, the runtime can whitelist the binaries that can be launched by the escape hatch. Each whitelisted binary must have a pre-generated taint model attached to it [26], such that the runtime can determine whether the binary is safe to launch given a particular set of tainted inputs, and if so, how taint should be assigned in the managed world when the binary terminates.
- The runtime can track instruction-level information flows in binaries launched by an escape hatch. To do so, the runtime must execute the native instructions via emulation [54, 76]. Strictly speaking, the runtime only needs to emulate instructions that touch sensitive data; the runtime can use page table permissions to detect when native code tries to access tainted data [36, 55, 56]. This optimization allows most native code to execute unemulated, i.e., directly atop the hardware.

PyRB could use any or all of these strategies. Our current PyRB prototype uses the first two. PyRB disallows C bindings by fiat, and only allows applications to spawn a child process if that process will be an instance of the PyRB interpreter (with the Python code to run in the child process specified as an argument to the child process). The parent and child PyRB interpreters will introspect on cross-process file descriptor communication, encapsulating the raw bytes within a custom protocol which ensures that taint is correctly propagated between the two runtimes.

4.6 Discussion

The necessity of IFC: A Riverbed server attests its systems software and its Riverbed managed runtime. However, the server does not attest the contents of higher-level code belonging to the web service. At first glance, this approach might seem odd: why not have the server attest all application code as well? If clients trust the attested application code, then server-side IFC might be unnecessary. However, in many cases, application code is not open source, e.g., because the code contains proprietary intellectual property that confers a competitive advantage to the web service owner. Code like this cannot be audited by a trusted third party, so end-users would gain little confidence from remote attestations of that code. Even if the server-side code were open source and publicly auditable, there are many more server applications than OSes and low-level systems software. Given a finite amount of resources that can be devoted to auditing, those resources are best spent inspecting the lowest levels of the stack. Indeed, if those levels are not secure, then even audited higher-level code will be untrustworthy. Also note that, even if the web service code has been audited, Riverbed provides security in depth, by catching any disallowed information flows that the audit may have missed.

Universe migration: Due to server-side load balancing or fail-over, a container belonging to a universe can migrate across different physical servers. From a user's perspective, migration is transparent if a user-facing container is placed on a server with a trusted stack—attestation involving the new server and the user's Riverbed proxy will succeed as expected. However, before migration occurs, the old server must force the new server to attest; in this fashion, the old server ensures that the new server runs a trusted Riverbed stack (and will therefore respect the data policies associated with the universe being migrated).

Preventing denial-of-service via spurious universe creation: Attackers might generate a large number of fake users, each of which has a policy that requires a separate universe; the attacker's goal would be to force the application to exhaust resources trying to manage all of the universes. Fortunately, in a given Riverbed application, each universe employs copy-on-write storage layered atop a base image. As a result, a new universe consumes essentially zero storage resources until the universe starts receiving actual client requests that write to storage. Riverbed also suspends cold universes to disk. Thus, a maliciously-created universe that is cold will consume no CPU cycles and no RAM space; storage overhead will be proportional to the write volume generated by client requests, but this overhead is no different than in a non-Riverbed application. Regardless, a Riverbed application should perform the same user verification [32, 73, 75] that a traditional web service performs.

Hostname management: Applications which use a microservice architecture will contain many small pieces of code that are executed by a potentially large number of hostnames. An application that uses elastic scaling may also dynamically bind service state to a large set of hostnames. User policies can employ wildcarded TLS hostnames [30] to avoid the need for a priori knowledge of all possible hostnames.

Taint relabeling: Consider a user named Alice. A Riverbed service assigns Alice to a universe upon receiving the first request from Alice (§4.4). What happens if Alice later wants to re-taint her data, i.e., assign a different policy to that data?

Suppose that Alice lives in a singleton universe that only contains herself. Further suppose that her policy modification keeps her in a singleton universe. In this scenario, re-tainting data is straightforward. If storage permissions were enabled but now are not, Riverbed deletes Alice's data on persistent storage. If network permissions changed, then Riverbed will only allow tainted data to flow to the new set of whitelisted endpoints. Nothing special must be done to handle tainted memory in the managed runtime—since Alice still lives in a singleton universe, there is no way for the service to combine her in-memory data with the data of others. If Alice later wants to invoke her "right to be forgotten," Riverbed just destroys Alice's universe.

The preceding discussion assumed that Alice only has universe state in a single TLS domain (e.g., `x.com`). However, Alice's singleton universe will span multiple domains if Alice's original policy enabled cross-domain data transfers. In these scenarios, Riverbed must disseminate a policy modification request to all relevant domains. Doing so is mostly straightforward, since the relevant domains are explicitly enumerated in Alice's original policy. Riverbed does need to pay special attention to wildcarded network sinks like `*.x.com`; such domains must expose a directory service that allows Riverbed to enumerate the concrete hostnames that are covered by the wildcard.

Now consider a different user Bob who wants to change his policy. If Bob lives in a universe that is shared with others, then re-tainting is harder, regardless of whether Bob wishes to transfer to a shared universe or a singleton one. The challenges are the same ones faced by a synchronous universe clone at policy-violation time (§4.4): since Riverbed is application-agnostic, Riverbed has no easy way to cleanly splice a user's data out of one universe and into another. Thus, if Bob lives in a shared universe and wishes to move to a different one, Riverbed must first use application-specific mechanisms to extract his data from his current universe. Then, Riverbed deletes Bob's current universe. Finally, Riverbed must re-inject Bob's data into the appropriate universe via application-specific requests. This migration process may be tedious, but importantly, *Riverbed narrows the scope of data finding and extraction*. When re-tainting must occur, the application only needs to look for Bob's data within Bob's original universe, not the full set of application resources belonging to all users. Before and after re-tainting, Riverbed ensures that Bob's IFC policies are respected.

CDNs: Large-scale web services use CDNs to host static objects that many users will need to fetch. CDN servers do not run application logic, but they do see user cookies which may contain sensitive information. So, by default, client-side Riverbed proxies force CDN nodes to attest. However, a proxy can explicitly whitelist CDN domains that should not be forced to attest.

Policy creep: Traditional end-user license agreements represent a crude form of data consent. In a EULA, a service provider employs natural language to describe how a service will handle user data; a user can then decide whether to opt into the service. Riverbed tries to empower users by giving *users* the ability to define policies for data manipulation. However, Riverbed cannot force a service to regard a user-defined policy as acceptable. Furthermore, the history of traditional EULAs suggests that, in a Riverbed world, services will prefer less restrictive Riverbed policies. For example, a service may refuse to accept a user if the user's Riverbed policy will not allow data flows to a particular advertising network. In this situation, the service can mandate that a less restrictive policy is the cost of admission to the service. Riverbed cannot prevent such behavior. However, Riverbed does force services to be more transparent about data promiscuity, because any service-suggested policy must be explicit about how data will be used. Riverbed also uses IFC to force services to adhere to policies.

Deployment considerations: Riverbed assumes that datacenter machines have TPM hardware. This assumption is reasonable, since TPMs are already present in many commodity servers.

In a complex, multi-tier application, components may span multiple administrative domains. The failure of some domains to run up-to-date stacks may lead to cascading problems with the overall application, as trusted stacks refuse to share data with unpatched ones. This behavior is actually desirable from the security perspective, and it incentivizes domains to keep their software up-to-date.

5 Implementation

The core of our Riverbed prototype consists of a client-side proxy (§4.3), a server-side reverse proxy (§4.4), and a taint-tracking Python runtime (§4.5). The two proxies, which are written in Python, share parts of their code bases, and comprise 773 lines in total, not counting external libraries to handle HTTP traffic [57] and manipulate Docker instances [23]. PyRB is a derivative of the PyPy interpreter [53], and contains roughly 500 lines of new or modified source code.

To implement remote attestation, servers used LG's UEFI firmware, which implemented the TPM 2.0 specification [70]. At boot time, the firmware extended a PCR with a TPM-aware version of the GRUB2 bootloader [17]. GRUB2 then extended the PCR with a TPM-aware version of the Linux 4.8 kernel. The kernel used Linux's Integrity Management Architecture [44] to automatically extend the PCR when loading kernel modules or user-mode binaries. Contextual attestation graphs were generated by Cobweb [74], with servers and client-side Riverbed proxies using the Cobweb library to implement the attestation protocol.

6 Evaluation

In this section, we demonstrate that Riverbed induces only modest performance penalties, allowing Riverbed to be a practical security framework for realistic applications. In all experiments, server code ran on an Amazon `c4` instance which had a 4-core Intel Xeon E5-2666 processor and 16 GB of RAM. The client was a 3.1 GHz Intel Core i7 laptop with 16 GB of RAM. The network latency between the client and the server was 14 ms.

6.1 Attestation Overhead

Before a client-side Riverbed proxy will send data to a server, the proxy will force the server to attest. We evaluated attestation performance under a variety of emulated network latencies and bandwidths. The client's policy required the attesting server to run a trusted version of `/sbin/init`, as well as trusted versions of 31 low-level system binaries like `/bin/sh`. The policy also used a Cobweb graph predicate (§4.3) to validate the process tree belonging to the Docker subsystem, ensuring that the tree contained no extraneous or missing processes.

Due to space restrictions, we only provide a summary of the results. Attestations were small (112 KB), so attestation time was largely governed by network latency, the cost of the slow TPM `quote()` operation (which took 215 ms on our server hardware), and Cobweb overheads for graph serialization, deserialization, and predicate matching (which required 562 ms of aggregate compute time on the server and the client-side proxy). On a client/server network link with a 14 ms RTT, the client-perceived time needed to fetch and validate an attestation was 846 ms. Proxies cache attestation results (§4.3), so this attestation penalty is amortized.

6.2 Case Studies

To study Riverbed's post-attestation overheads, we ported three Python applications to Riverbed.

- MiniTwit [59] is a Twitter clone that implements core Twitter features like posting messages and following users. Application code runs in Flask [58], a popular server-side web framework. MiniTwit uses a SQLite database to store persistent information. We defined a Riverbed policy which allowed user data aggregation, and allowed tainted data to be written to storage and to other network servers in our MiniTwit deployment.
- Ionic Backup [51] is a Dropbox clone that provides a user with online storage. Ionic allows a user to upload, download, list, and delete files on the storage server. The Ionic client uses HTTP to communicate with the server. For this application, we defined a Riverbed policy which allowed user data to be written to disk, but disallowed aggregation, and prevented user data from being sent to other network servers.
- Thrifty P2P [43] implements a peer-to-peer distributed hash table [60, 68]. The primary client-facing operations are `PUT(key,value)` and `GET(key)`. Internally, Thrifty peers issue their own traffic to detect failed hosts, route puts and gets to the appropriate peers, and so on. For this application, we defined a Riverbed policy which allowed aggregation and storage, but only allowed tainted data to be written to endpoints that resided in our test deployment of Thrifty servers.

Ionic required no modifications to run atop Riverbed. Thrifty peers used a custom network protocol to communicate; so, we had to build a proxy for the Thrift RPC layer [3] that injected Riverbed policies into outgoing messages, and tainted incoming data appropriately. MiniTwit's core application logic required no changes, but, to reduce the likelihood of over-tainting, we did modify MiniTwit's Python-based database engine to be natively taint-aware, e.g., so that each database row had an associated on-disk taint bit, and so that query results were tagged with the appropriate union taints, based on the items that were read and written to satisfy the query. Our modifications are hidden beneath a narrow abstraction layer, making it easy to integrate the Python-level MiniTwit logic with off-the-shelf taint-tracking databases [63, 64, 77].

Figure 4 depicts end-to-end performance results for MiniTwit, Ionic, and Thrifty. The results demonstrate that Riverbed imposes small client-perceived overheads (1.01x–1.10x). Figure 5 isolates Riverbed's server-side computational penalties. For each request type, we compare server-side performance when using unmodified PyPy, PyRB in which no data is tainted, or PyRB in which data is tainted according to the policies that we described earlier in this section. For MiniTwit, Riverbed had overheads of 1.02x–1.15x. For Ionic, Riverbed imposed overheads of 1.04x–1.16x. For Thrifty, puts and gets had slowdowns of 1.18x and 1.26x respectively. Riverbed imposed the least overhead for Ionic's "remove" and

Operation	Without Riverbed	With Riverbed
MiniTwit view timeline	229 ms	252 ms
Ionic download	82.5 ms	83.1 ms
Ionic ls	14.1 ms	14.2 ms
Thrifty GET request	27.5 ms	28.0 ms

Figure 4: End-to-end response times for processing various user requests. For MiniTwit, the user viewed her timeline. For Ionic, the user downloaded a 300 KB file, or asked for a list of the contents of a server-side directory. For Thrifty, the client fetched a 20 byte value from a DHT that contained 2 nodes; the DHT was intentionally kept small to emphasize the computational overheads of Riverbed. The client/server network latency was 14 ms. Each result is the average of 50 trials.

Operation	Regular PyPy	PyRB (no taint)	PyRB (taint)
MiniTwit post message	14 ms	15 ms	15 ms
MiniTwit view timeline	4.1 ms	4.2 ms	4.2 ms
MiniTwit follow user	13 ms	15 ms	15 ms
Ionic upload	2.3 ms	2.5 ms	2.5 ms
Ionic download	4.8 ms	5.0 ms	5.0 ms
Ionic ls	0.43 ms	0.50 ms	0.50 ms
Thrifty PUT request	0.16 ms	0.17 ms	0.19 ms
Thrifty GET request	0.19 ms	0.24 ms	0.24 ms

Figure 5: Server-side overheads for processing various user requests. The workloads are a superset of the ones in Figure 4. Each result is the average of 50 trials.

"delete" operations, since PyRB could handle these operations merely by issuing file system calls, without handling much in-memory data that had to be checked for taint. In contrast, operations that involved reading or writing network data required PyRB to interpose on data processing code, even if no data was tainted, and perform extra work at data sources and sinks.

6.3 PyPy Benchmarks

For a wider perspective on PyRB's performance, we used PyRB to run the benchmarking suite from the Performance project [52]. The suite focuses on real Python applications, downloading the necessary packages for those applications and then running the real application code. Figure 6 shows PyRB's performance on a representative set of benchmarks. The benchmarks that are above the thin black line resemble applications that might run inside of a Riverbed universe; these benchmarks perform actions that are common to web services, like parsing HTML, responding to HTTP requests, and performing database queries. These benchmarks tend to be IO-heavy, with occasional CPU idling as code waits for IOs to complete. In contrast, the benchmarks beneath the thin black line are CPU-intensive. PyRB does not affect the speed of IOs, but does affect the speed of computation, so PyRB has slightly higher overhead for the bottom set of benchmarks. Overall, PyRB is at most 1.19x slower. These results overestimate PyRB's overheads because clients and servers resided on the same machine (and thus incurred zero network latency).

Benchmark	Overhead
Django	1.14x
Render HTML table	1.16x
Code run in PyPy interpreter	1.08x
JSON parsing	1.13x
Python git operations	1.01x
SQL Alchemy	1.05x
Spitfire	1.19x
Twisted	1.17x
Fractal Generation	1.18x
Spectral Norm	1.10x
Raytracing	1.19x

Figure 6: PyRB's performance on representative benchmarks from the Performance benchmark suite [52]. PyRB's performance is normalized with respect to that of regular PyPy. No data was tainted in these experiments.

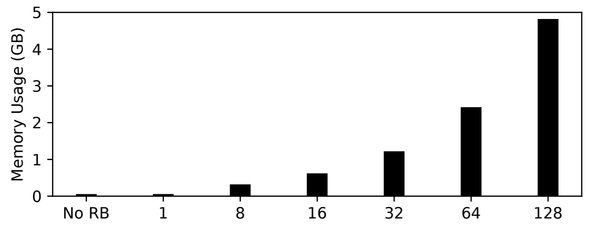

Figure 7: Physical memory pressure in MiniTwit when run without Riverbed, or with Riverbed using various numbers of universes. Note that in MiniTwit, each universe requires only one container. In each test configuration, we measured memory pressure after submitting 1000 requests to each MiniTwit instance that existed in the configuration.

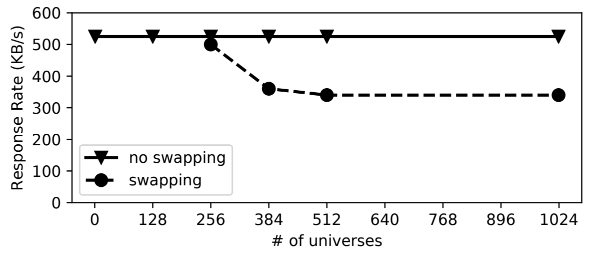

Figure 8: MiniTwit server response rate as a function of (1) the number of universes, and (2) whether the server had 60 GB of RAM or 16 GB of RAM. We used the Apache Benchmark tool [1] to simulate clients that requested MiniTwit timelines which had 100 messages. In each trial, we submitted 1000 requests, with 100 outstanding requests at any given time. For the server with 16 GB of RAM, swapping began with 256 universes.

6.4 Universe Overhead

The size for a base Riverbed Docker image is 212 MB. The image contains the state that belongs to the PyRB runtime, and is similar in size to the official PyPy Docker image [22]. Each Riverbed service adds application-specific code and data to the base Riverbed image. However, a live Docker instance uses copy-on-write storage, so multiple Riverbed universes share disk space (and in-memory page cache space) for common data.

We believe that for most Riverbed applications, the universe abstraction will not increase overall storage requirements; in other words, the space needed for per-universe data plus shared-universe data will be similar to the space needed for the non-Riverbed version of the application. For example, in MiniTwit, for a given number of timelines with a given amount of posts, the storage requirements are the same if the timelines are partitioned across multiple Riverbed universes, or kept inside a single, regular MiniTwit deployment. However, Docker's copy-on-write file system does result in slower disk IOs. As a concrete example, we measured MiniTwit's database throughput when MiniTwit ran directly atop ext4, and when MiniTwit ran inside a universe that used Docker's overlayfs file system [24]. We examined database workloads with read/write ratios of 95/5 and 50/50, akin to the YCSB workloads A and B [16]. The targeted database rows were drawn from a Zipf distribution with $\beta = 0.53$, similar to the distribution observed in real-life web services [4, 72]. We found that, inside a Riverbed universe, transaction throughput slowed by 7.7% for the 95/5 workload, and by 17.3% for the 50/50 workload.

For our three sample applications, spawning a new Docker container required 260–280 ms on our test server. In Riverbed, the container creation penalty is rarely paid; the reverse proxy only has to create a new universe upon seeing a request with a policy that is incompatible with all pre-existing universes. Subsequent requests which are tagged with that policy will be routed to the pre-existing universe.

Creating new universes is rare, but pausing and unpausing old ones may not be. If an application has many universes, and memory pressure on a particular physical server is high, then temporarily-quiescent universes can be suspended to disk. On our test server with 512 live containers, pausing or unpausing a single Docker instance took roughly 30 ms. However, recent empirical research has shown that in datacenters, a tenant's resource requirements are often predictable [19]. Thus, universes can be assigned to physical servers in ways that reduce suspension/resumption costs.

Docker virtualizes at the POSIX level, so the processes inside of a Riverbed universe are just processes inside of the host OS. As a result, the RAM footprint for a Riverbed universe is just the memory that is associated with the host processes for the universe. Our Riverbed prototype was able to spawn up to 1023 live containers on a single server. This 1023 bound is a well-known limitation of the current Docker implementation. Docker associates a virtual network card with each instance, and attaches the virtual card to a Linux network bridge [69]; a Linux bridge can only accept 1023 interfaces. Regardless, the current bound of 1023 containers per machine does not imply that a single application can have at most 1023 universes. The bound just means that, if an application has more than 1023 universes, then those universes must be spread across multiple servers. Riverbed's reverse proxy (§4.4) considers server load when determining where to create or resurrect a universe; thus, the per-server container limit is not a concern in practice.

Figure 7 demonstrates that Riverbed's memory pressure is linear in the number of active containers. As shown in Figure 8, a large number of universes has no impact on server throughput if all of the hot universes fit in memory. Unsurprisingly, throughput drops if active universes must be swapped between RAM and disk. However, a Docker container is just a set of Linux processes that are constrained using namespaces [38] and cgroups [12]; thus, the memory overhead for launching a Riverbed universe with *N* processes is similar to the memory overhead of scaling out a regular application by creating *N* regular processes. That being said, a Riverbed application does create processes more aggressively than a normal application. In Riverbed, incompatible policies require separate universes (and therefore separate processes), even if aggregate load across all universes is low.

7 Conclusion

Riverbed is a platform that simplifies the creation of web services that respect user-defined privacy policies. A Riverbed universe allows a web service to isolate the data that belongs to users with the same privacy policy; Riverbed's taint tracking ensures that the data cannot flow to disallowed sinks. Riverbed's client-side proxy will not divulge sensitive user data until servers have attested their trustworthiness. Riverbed is compatible with commodity managed languages, and does not force developers to annotate their source code or reason about security lattices. Experiments with real applications demonstrate that Riverbed imposes no more than a 10% performance degradation, while giving both users and developers more confidence that sensitive data is being handled correctly.

References

[1] Apache Software Foundation. Apache Benchmark. https://httpd.apache.org/docs/2.4/programs/ab.html.

[2] Apache Software Foundation. Apache SpamAssassin: Open-source Spam Filter. http://spamassassin.apache.org/.

[3] Apache Software Foundation. Apache Thrift. https://thrift.apache.org/.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of SIGMETRICS*, pages 53–64, 2012.

[5] T. Austin and C. Flanagan. Efficient Purely-dynamic Information Flow Analysis. *ACM SIGPLAN Notices*, 44(8):20–31, 2009.

[6] T. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of PLAS*, 2010.

[7] T. Austin, J. Yang, and C. F. A. Solar-Lezama. Faceted execution of policy-agnostic programs. In *Proceedings of the SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 15–26, 2013.

[8] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of OSDI*, pages 267–283, 2014.

[9] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of USENIX Security*, pages 305–320, 2006.

[10] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information Flow Control in WebKit's JavaScript Bytecode. In *International Conference on Principles of Security and Trust*, pages 159–178, 2014.

[11] A. Booth. Charities Hit with Fines for Sharing Donors' Data Without Consent, December 7, 2016. Sophos Naked Security Blog. https://nakedsecurity.sophos.com/2016/12/07/charities-hit-with-fines-for-sharing-donors-data-without-consent/.

[12] N. Brown. Control groups, July 7, 2014. LWN. https://lwn.net/Articles/604609/.

[13] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Proceedings of the Computer Security Applications Conference*, pages 463–475, 2007.

[14] S. Chong, K. Vikram, and A. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of USENIX Security*, pages 1–16, 2007.

[15] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, and B. Sniffen. Principles of Remote Attestation. *International Journal of Information Security*, 10(2):63–81, June 2011.

[16] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of SOCC*, pages 143–154, 2010.

[17] CoreOS. GRand Unified Bootloader 2.0. https://github.com/coreos/grub.

[18] J. Corpuz. Best Ad Blockers and Privacy Extensions: Chrome, Safari, Firefox, and IE. Tom's Guide. https://www.tomsguide.com/us/pictures-story/565-best-adblockers-privacy-extensions.html.

[19] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of SOSP*, pages 153–167, 2017.

[20] V. Costan and S. Devadas. Intel SGX Explained, February 20, 2017. Cryptology ePrint Archive: Version 20170221:054353. https://eprint.iacr.org/2016/086.pdf.

[21] Docker. Docker Home Page. https://docker.com.

[22] Docker. Docker PyPy Images. https://hub.docker.com/_/pypy/.

[23] Docker. Docker SDK for Python. https://docker-py.readthedocs.io/en/stable/.

[24] Docker Docs. Using the OverlayFS storage driver. https://docs.docker.com/storage/storagedriver/overlayfs-driver/.

[25] E. Elnikety, A. Mehta, A. Vahldiek-Oberwagner, D. Garg, and P. Druschel. Thoth: Comprehensive Policy Compliance in Data Retrieval Systems. In *Proceedings of USENIX Security*, pages 637–654, 2016.

[26] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2), 2014.

[27] A. Ermolinskiy, S. Katti, S. Shenker, L. Fowler, and M. McCauley. Towards Practical Taint Tracking. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-92*, 2010.

[28] EU Parliament. GDPR Portal, 2017. http://www.eugdpr.org/eugdpr.org.html.

[29] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazieres, J. C. Mitchell, and A. Russo. Hails: Protecting Data Privacy in Untrusted Web Applications. In *Proceedings of OSDI*, pages 47–60, 2012.

[30] GoDaddy. What is a Wildcard SSL certificate? https://www.godaddy.com/help/what-is-a-wildcard-ssl-certificate-567.

[31] Google. What is Safe Browsing? https://developers.google.com/safe-browsing/.

[32] S. Gurajala, J. White, B. Hudson, and J. Matthews. Fake Twitter Accounts: Profile Characteristics Obtained Using an Activity-based Pattern Detection Approach. In *Proceedings of the International Conference on Social Media and Society*, 2015.

[33] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Proceedings of OSDI*, pages 165–181, 2014.

[34] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1663–1671, 2014.

[35] D. Hedin and A. Sabelfeld. A Perspective on Information-Flow Control. In *Proceedings of the Marktoberdorf Summer School*, August 2011.

[36] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection Using Demand Emulation. In *Proceedings of EuroSys*, pages 29–41, April 2006.

[37] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of OSDI*, pages 533–549,

2016.

[38] M. Kerrisk. Namespaces in Operation, Part 1: Namespaces Overview, January 4, 2013. LWN. https://lwn.net/Articles/531114/.

[39] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *Proceedings of SOSP*, 2007.

[40] J. Layton. Extended File Attribute Rock!, June 29, 2011. http://www.linux-mag.com/id/8741/.

[41] K. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370, 2010.

[42] P. Li, Y. Mao, and S. Zdancewic. Information Integrity Policies. In *Proceedings of the Workshop on Formal Aspects in Security and Trust*, September 2003.

[43] A. Lindsay. Thrifty P2P. https://github.com/atl/thrifty-p2p.

[44] Linux. Integrity Measurement Architecture. https://sourceforge.net/p/linux-ima/wiki/Home/.

[45] J. Liu, M. George, K. Vikram, X. Qi, L. Waye, and A. Myers. Fabric: A Platform for Secure Distributed Computation and Storage. In *Proceedings of SOSP*, pages 321–334, October 2009.

[46] B. Livshits. Dynamic taint tracking in managed runtimes. *Technical Report MSR-TR-2012-114, Microsoft*, 2012.

[47] Lyon Brothers Enterprises. Turtl: Find Your Private Space. https://turtlapp.com/.

[48] A. Myers and B. Liskov. Protecting Privacy Using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[49] Oracle Corporation. Java Native Interface. http://docs.oracle.com/javase/8/docs/technotes/guides/jni/.

[50] T. Pasquier, J. Singh, J. Bacon, and D. Eyers. Information Flow Audit for PaaS Clouds. In *Proceedings of the IEEE International Conference on Cloud Engineering*, pages 42–51, 2016.

[51] F. Primerano. Ionic Backup. https://github.com/Max00355/IonicBackup.

[52] PyPy. PyPy Benchmarks. https://bitbucket.org/pypy/benchmarks.

[53] PyPy. PyPy Home Page. https://pypy.org/.

[54] C. Qian, X. Luo, Y. Shao, and A. Chan. On Tracking Information Flows Through JNI in Android Applications. In *Proceedings of DSN*, pages 180–191, June 2014.

[55] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of MICRO*, pages 135–148, December 2006.

[56] A. Razeen, A. Lebeck, D. Liu, A. Meijer, V. Pistol, and L. Cox. SandTrap: Tracking Information Flows On Demand with Parallel Permissions. In *Proceedings of MobiSys*, June 2018.

[57] K. Reitz. Requests: HTTP for Humans. http://docs.python-requests.org/en/master/.

[58] A. Ronacher. Flask. http://flask.pocoo.org/.

[59] A. Ronacher. Minitwit. https://github.com/pallets/flask/blob/master/examples/minitwit/.

[60] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-scale Peer-to-peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350, 2001.

[61] A. Russo and A. Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of CSF*, pages 186–199, 2010.

[62] P. Sayer. German Consumer Groups Sue WhatsApp Over Privacy Policy Changes, January 30, 2017. PCWorld. http://www.pcworld.com/article/3163027/private-cloud/german-consumer-groups-sue-whatsapp-over-privacy-policy-changes.html.

[63] D. Schoepe, D. Hedin, and A. Sabelfeld. SeLINQ: Tracking Information Across Application-Database Boundaries. In *ACM SIGPLAN Notices*, volume 49, pages 25–38, 2014.

[64] D. Schultz and B. Liskov. IFDB: Decentralized Information Flow Control for Databases. In *Proceedings of EuroSys*, pages 43–56, 2013.

[65] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 317–331, 2010.

[66] S. Sen, S. Guha, A. Datta, S. Rajamani, J. Tsai, and J. Wing. Bootstrapping Privacy Compliance in Big Data Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 327–342, 2014.

[67] A. Slowinska and H. Bos. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *Proceedings of EuroSys*, pages 61–74, 2009.

[68] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[69] The Linux Foundation. Linux Network Bridge. https://wiki.linuxfoundation.org/networking/bridge.

[70] Trusted Computing Group. TPM 2.0 Library Specification. https://trustedcomputinggroup.org/tpm-library-specification/.

[71] Trusted Computing Group. Trusted Platform Module (TPM) Summary. https://trustedcomputinggroup.org/trusted-platform-module-tpm-summary/.

[72] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia Workload Analysis for Decentralized Hosting. *International Journal of Computer and Telecommunications*

Networking, 53(11):1830–1845, 2009.

[73] E. van der Walt and J. Eloff. Using Machine Learning to Detect Fake Identities: Bots vs Humans. *IEEE Access*, 6:6540–6549, January 2018.

[74] F. Wang, Y. Joung, and J. Mickens. Cobweb: Practical Remote Attestation Using Contextual Graphs. In *Proceedings of SysTEX*, 2017.

[75] C. Xiao, D. Freeman, and T. Hwa. Detecting Clusters of Fake Accounts in Online Social Networks. In *Proceedings of the ACM Workshop on Artificial Intelligence and Security*, pages 91–101, 2015.

[76] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu. Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART. In *Proceedings of USENIX Security*, pages 289–306, August 2017.

[77] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong. Precise, Dynamic Information Flow for Database-backed Applications. In *Proceedings of PLDI*, pages 631–647, 2016.

[78] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *ACM SIGPLAN Notices*, volume 47, pages 85–96, 2012.

[79] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving Application Security with Data Flow Assertions. In *Proceedings of SOSP*, pages 291–304, 2009.

[80] A. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Untrusted Hosts and Confidentiality: Secure Program Partitioning. In *Proceedings of SOSP*, pages 1–14, 2001.

[81] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing Distributed Systems with Information Flow Control. In *Proceedings of NSDI*, pages 293–308, 2008.

[82] K. Zetter. Hackers Finally Post Stolen Ashley Madison Data, August 18, 2015. Wired. https://www.wired.com/2015/08/happened-hackers-posted-stolen-ashley-madison-data/.

Appendix: Overview of Attestation

In this section, we give a slightly simplified description of the classic attestation protocol. We explain how an *attestor* (i.e., a potentially untrustworthy machine) securely describes its software stack to a remote *verifier* machine. For more details, we refer the interested reader to other work [15, 20, 9].

Setup: The attestor's trusted hardware (called a TPM chip [71]) possesses a unique public/private key pair that is burned into the hardware. The private key is never exposed to the rest of the machine. The attestor also has a certificate, signed by the manufacturer of the TPM, that binds the attestor to its public key. Thus, the hardware manufacturer acts as a certificate authority (CA). Before the remote attestation protocol begins, the verifier must download the public key of the CA.

A TPM contains a small number of platform configuration registers (PCRs). Each PCR is made of tamper-resistant, non-volatile RAM that only the TPM can access. At boot time, the TPM resets each PCR to a well-known value. The TPM's `extend(index, value)` is the only way that entities external to the TPM can update a PCR. An extension sets `PCR[index] = SHA1(PCR[index] || value)`. During the boot process, the BIOS automatically extends `PCR[10]` with a `value` equal to the SHA1 hash of the BIOS code. The BIOS then reads the bootloader from the disk, extends `PCR[10]` with the hash of the bootloader, and jumps to the first instruction of the bootloader. The bootloader reads the kernel binary into RAM, extends `PCR[10]` with the hash of the kernel image, and then jumps to the first instruction of the kernel. These PCR extensions continue as the OS loads additional kernel modules and user-level system binaries. Thus, the attestor's `PCR[10]` register will contain a cumulative hash of the local software stack.

Remote attestation: The verifier generates a random nonce and sends it to the attestor. The attestor asks its local TPM to generate a signature over the nonce and the value of `PCR[10]`; this signature, which is called a "quote" in TPM parlance, uses the attestor's unique private key (whose corresponding public key is validated by a certificate from the CA). The attestor returns the following information to the verifier:

- the attestor's certificate,
- the quote,
- the value of `PCR[10]` that is attested by the quote,
- a list of the SHA1 hashes that were used to extend `PCR[10]`, and
- optionally, a mapping from each hash to the server-side file name representing the content that was hashed.

The verifier checks the validity of the attestor's public key using the certificate. The verifier then checks the validity of the quote signature, and confirms that cumulatively extending `PCR[10]` with the attestor-reported hash list results in the attestor-reported `PCR[10]` value. If these checks succeed, the verifier sees whether the hash list corresponds to a trusted ordering of trusted system components. If so, the remote attestation succeeds.

Attesting VMs: The traditional attestation protocol can be extended to cover the software stack inside of a VM [9]. During the initial boot sequence of a physical server, the physical `PCR[10]` will be extended with the bootloader code, the hypervisor binary, and other low-level software. The hypervisor will then extend `PCR[10]` using content associated with the virtual TPM manager; this content includes the binary of the manager itself, as well as certificates needed to vouch for the signatures produced by a VM's virtual TPM. When a VM launches, the manager initializes the VM's virtual PCRs using the values in the physical PCRs. The VM then boots, extending (virtual) PCRs as usual. In this manner, the attestation produced by a virtual TPM will be linked to a non-virtualized root of trust (i.e., the physical TPM of the server).