Riverbed

“Enforcing User-defined Privacy Constraints in Distributed Web Services”
Hypothetical Scenario
Hypothetical Scenario

- Slack announces that they send the contents of your chat messages directly to advertising companies (say, doubleclick)
- For people that want to opt out they provide a setting to do so
- You check the box to opt out
- But how do you know that Slack does what you want them to do?
  - Does opting out also cover Slack sending analyses of your data (ie. statistics about your messages) to third party companies?
- In general, how do we trust that Slack isn’t doing anything shady?
Enter Riverbed
Riverbed

- Distributed IFC system that allows the user to specify the policies that a server will abide by when handling/processing that user’s data

- Consists of:
  - (1) A client-side web proxy
  - (2) A modified server-side runtime (PyRB, which is a modified PyPy)
  - (3) Trusted hardware (TPM)
  - (4) A server-side daemon for RB connections
Overview of Riverbed
How it works

The user can specify in their policy:

- The environment that their data will be processed in
- Whether or not their data can be written to disk
- To which web hosts the data could be passed to

1. The client’s RB web proxy initiates an attestation challenge to the server
2. The proxy then sends the user’s data along with their policy to the server
3. The RB runtime on the server then uses taint tracking to ensure that the application code respects the user’s policies (no unauthorized leaks to disk or network)
Details - User Policies
Example User Policy

USER-ID: ALICE
AGGREGATION: False
PERSISTENT-STORE: True
ALLOW-TO-NETWORK: x.com
ALLOW-TO-NETWORK: y.com
TRUSTED-SERVER-STACK:

```
83145c082bbf608989f05e85c3c211f83,
c8cd7ac93cab2b94f65a5b2de5709767f,
...
590f01d8d18b1141988ee1975b3ce3b30
```
Details - Attestation
Attestation - Some Jargon

- Jargon
  - Verifier = Client
  - Attestor = Server
Attestation - Setup

- Requires trusted hardware
  - **TPM - Trusted Platform Module**: A cryptoprocessor with tamper-resistant, non-volatile RAM and a pub/priv key pair burned into hardware
    - The private key lives only in the chip (i.e. never gets released to the rest of the hardware)
    - Upon boot, initializes and then one-by-one extends (“adds to”) a cumulative hash using the hashes of the first the BIOS, then the Bootloader, then Kernel Image, Kernel Modules, Software and so on, all within the TPM’s memory
      - Exact formula given in paper is given as
        - \( \text{reg}_{TPM} = \text{SHA1}(\text{reg}_{TPM} \| \text{value}) \)
        - Where \text{value} is the SHA1 hash of whatever layer we’re verifying
  - Attestor machine also has a **certificate** signed by the TPM manufacturer that binds the attester to its public key
    - TPM manufacturer \(\sim=\) HTTPS Certificate Authority
Attestation - Process

- The client [verifier] downloads the certificate of the TPM manufacturer
- Client generates nonce and sends to server [attestor]
- Server has its TPM chip generate a signature over the nonce and the cumulative hash of the software stack via the TPM’s private key (the signature is called a “quote”)
- Server sends back:
  - (1) Server’s attestor cert
  - (2) The quote (cumulative signature)
  - (3) Cumulative hash
  - (4) List of SHA1 hashes for each of the software components on the stack
- Then client then verifies the quote, cumulative hash, then the list of the software components
- If all checks out, then it’s good! (all the software is what’s expected)
- If not, then reject
Issues? Merits?
Details: Universes
RB Universes

- *Universe* = Copy of a server-side service
- Users with the same/compatible policies get to run on the same copy of a service
- Users with incompatible policies get run in different universes
- Users with $AGGREGATION = False$ get run in their own universes
Example Policy Interaction

Data policies

USER-ID: Alice
AGGREGATION: True
PERSISTENT-STORAGE: True
ALLOW-TO-NETWORK: x.com
TRUSTED-SERVER-STACK: S0

USER-ID: Bob
AGGREGATION: True
PERSISTENT-STORAGE: True
ALLOW-TO-NETWORK: x.com
TRUSTED-SERVER-STACK: S0

USER-ID: Charlie
AGGREGATION: False
PERSISTENT-STORAGE: True
ALLOW-TO-NETWORK: x.com
TRUSTED-SERVER-STACK: S0

Physical server

Universe 0
Application code
Riverbed managed runtime (IFC)

Universe 1
Application code
Riverbed managed runtime (IFC)

IO devices
Issues? Merits?
Details - Threat Model
Who’s trusted?
Threat Model

- Weakly Adversarial
  - Honest mistakes, but not dishonest developers
- Datacenter operators are trusted
- Entire client side is trusted
  - Except for web content
- HTTPS system is trusted
- (TPM hardware/manufacturer is trusted)
Riverbed + Slack
How would we incorporate Riverbed into the Slack system?
Riverbed + {FB, Twitter, News sites, et. al}
How suitable is Riverbed for other sorts of systems?
Performance
Test Apps

- **MiniTwit**
  - Mini Twitter App
  - Core features: posting messages, following users
  - Flask + SQLite
  - Policy: Agg, Store = Yes; Data only to researchers’ MiniTwit deployment

- **Ionic Backup**
  - Mini Dropbox clone
  - Upload+Download+List+Delete files
  - Policy: Store = Yes; User data can’t be sent to other servers

- **Thrifty P2P**
  - Peer-to-Peer distributed hash table
  - PUT(key, val), GET(key, val)
  - Policy: Agg, Store = Yes; Data could only be written to researchers’ Thrifty Servers
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Django</td>
<td>1.14x</td>
</tr>
<tr>
<td>Render HTML table</td>
<td>1.16x</td>
</tr>
<tr>
<td>Code run in PyPy interpreter</td>
<td>1.08x</td>
</tr>
<tr>
<td>JSON parsing</td>
<td>1.13x</td>
</tr>
<tr>
<td>Python git operations</td>
<td>1.01x</td>
</tr>
<tr>
<td>SQL Alchemy</td>
<td>1.05x</td>
</tr>
<tr>
<td>Spitfire</td>
<td>1.19x</td>
</tr>
<tr>
<td>Twisted</td>
<td>1.17x</td>
</tr>
<tr>
<td>Fractal Generation</td>
<td>1.18x</td>
</tr>
<tr>
<td>Spectral Norm</td>
<td>1.10x</td>
</tr>
<tr>
<td>Raytracing</td>
<td>1.19x</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Without Riverbed</th>
<th>With Riverbed</th>
</tr>
</thead>
<tbody>
<tr>
<td>MiniTwit view timeline</td>
<td>229 ms</td>
<td>252 ms</td>
</tr>
<tr>
<td>Ionic download</td>
<td>82.5 ms</td>
<td>83.1 ms</td>
</tr>
<tr>
<td>Ionic ls</td>
<td>14.1 ms</td>
<td>14.2 ms</td>
</tr>
<tr>
<td>Thrifty GET request</td>
<td>27.5 ms</td>
<td>28.0 ms</td>
</tr>
</tbody>
</table>

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.

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

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.

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.