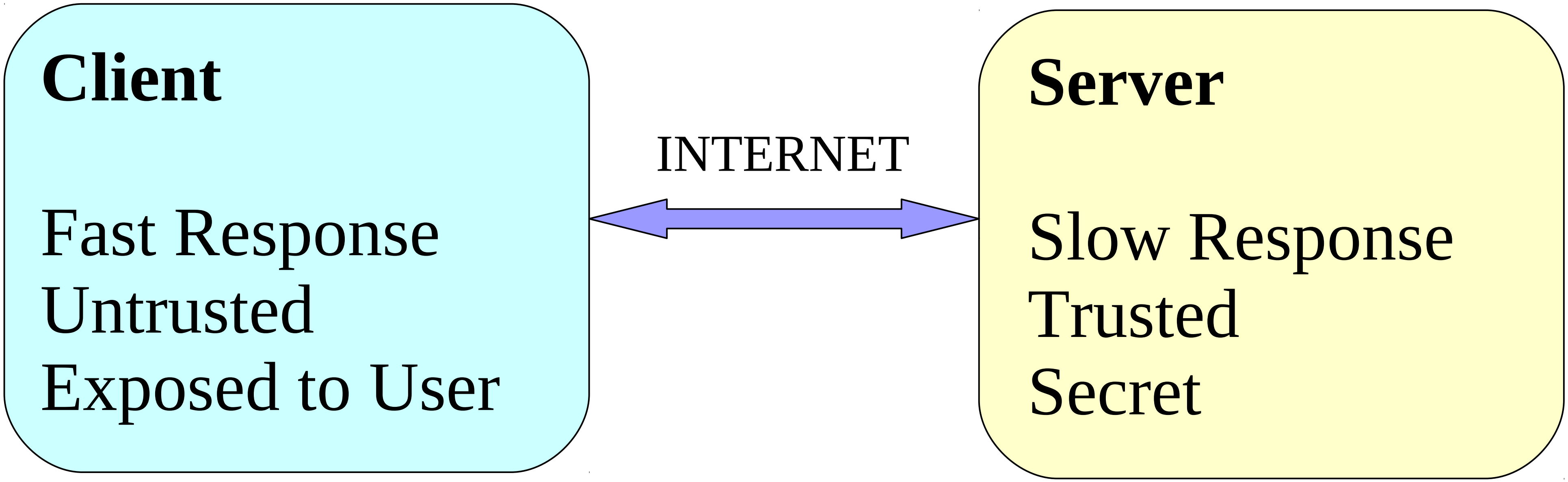
Integrity and Confidentiality in Web Applications

Jeff Johnson

Where We're At

- We have been mostly focusing on
  - Fixing vulnerabilities inherent to languages
  - Finding/protecting against programmer follies
- Today we will look at security at a slightly different angle
  - Assume we have good programmers
  - Concern ourself with the environment in which the code is run
  - Focus on **trust**

Security in Web Apps



Client

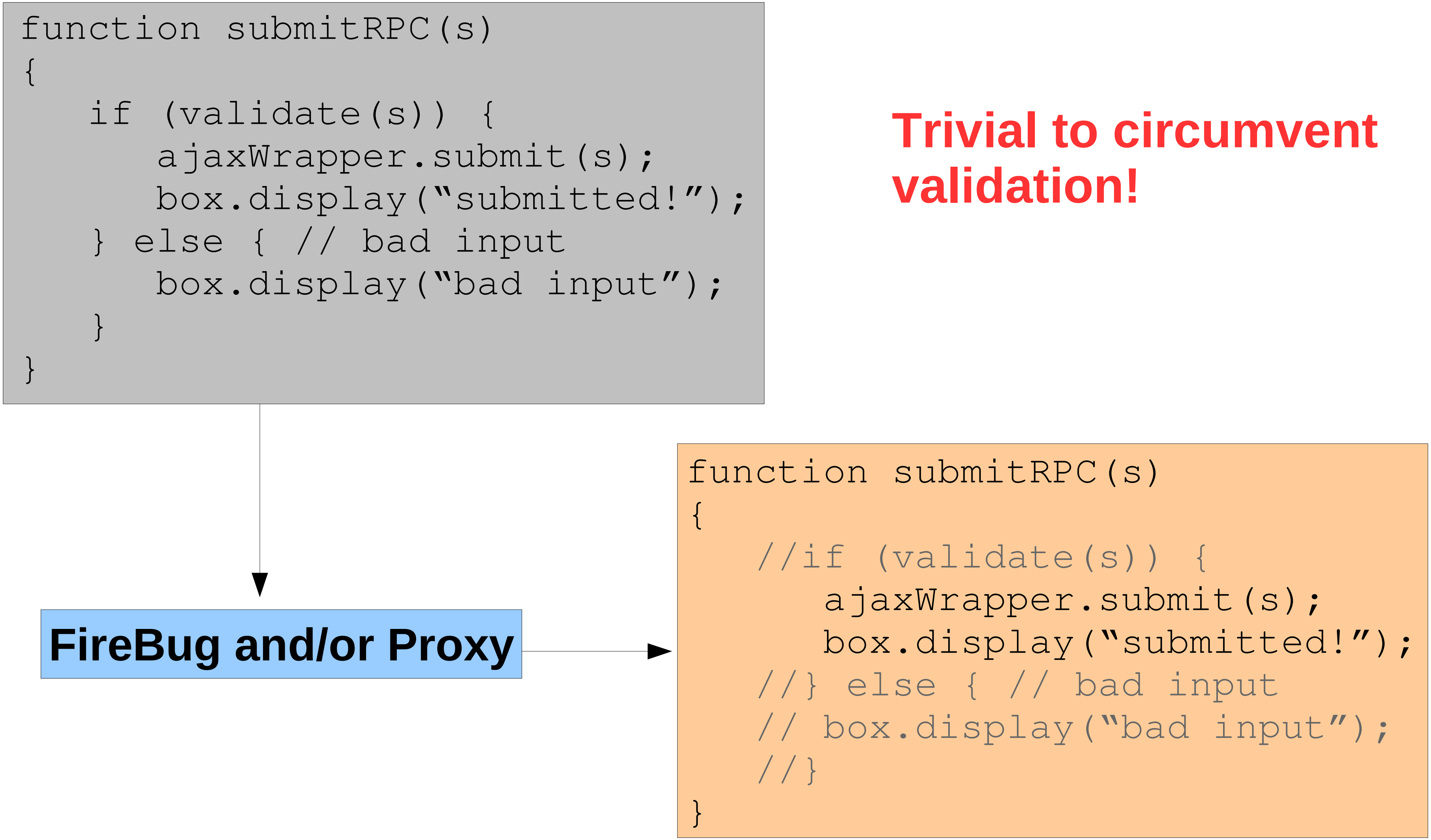
Fast Response
Untrusted
Exposed to User

INTERNET

Server

Slow Response
Trusted
Secret

Attack From the Client-Side



```
function submitRPC(s)
{
   if (validate(s)) {
      ajaxWrapper.submit(s);
      box.display(“submitted!”);
   } else { // bad input
      box.display(“bad input”);
   }
}
```

Trivial to circumvent validation!

FireBug and/or Proxy

```
function submitRPC(s)
{
   //if (validate(s)) {
      ajaxWrapper.submit(s);
      box.display(“submitted!”);
   //} else { // bad input
   // box.display(“bad input”);
   //}
}
```

Attack From the Client-Side

Bottom Line:

NEVER trust ANYTHING Client gives you
NEVER store secrets at the Client

Web Apps: Current State of the Art

- Write two closely-coupled programs glued together with AJAX
  - Client-side: Javascript
  - Server: Java, C#, etc.
- Programmer reasons about security
  - Who should do what computations?
    - Remember: the client-side is so very responsive but so very untrusted
  - What data goes where?
- This is difficult for many reasons

Alleviating Stress: Two Approaches

- Swift
  - Make the app's security requirements explicit
  - Use static checking to enforce
  - Secure by design
- Ripley
  - Replicate untrusted computation in a trusted environment
  - Compare results to detect misbehaving clients

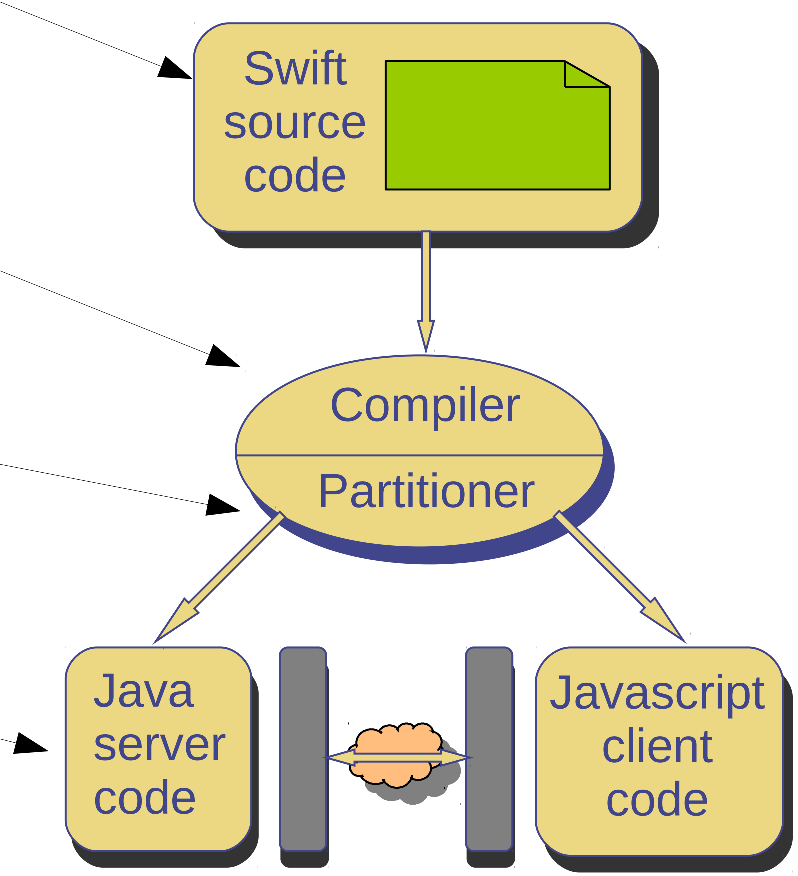
Secure Web Applications via Automatic Partitioning

S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, X. Zheng

Swift

- Key insight: reasoning about security is difficult because it is:
  - Across two entities
  - Not explicit
  - Not enforced or checked
- Swift says: Write **one** program using security labels
  - Enforce security with a static checker
  - Compiler: split code between client and server based on labels

Swift Overview



1) Write source with security labels
(in Jif programming language)

2) Check labels,
convert to WebIL
(what can go on
client, server?)

3) Determine placement,
convert to JS and Java

4) Runtime

Write Source Using Jif

- Java-like syntax
- Plus labels for specifying read/write capabilities between 'principals'

Confidentiality policy

Integrity policy

```
int {bob -> alice} x;

String {bob <- alice} s;
```

Jif (2)

Can strengthen/weaken policies as needed (dangerous but necessary):

declassify
change confidentiality of an expression or a statement

endorse
changes integrity of an expression or a statement

Jif in Swift

- Two principals
  - “*” denotes server
  - “client” denotes client
- * is trusted
  - In Jif terms: * `actsfor` client
- Bottom line
  - client sees data no greater than {* -> client}
  - client produces data no greater than {* <- client}

Example: Labels

```
int secret;
int count;
...
void guess(int i)
{
    if (i == secret) {
        messageBox.display(“winner”);
    } else {
        count++;
        messageBox.display(“looser”);
    }
}
```

Labeling Variables

```
int {*->*; *<-*} secret;
int {*->client; *<-*} count;
...
void guess(int i)
{
   if (i == secret) {
      messageBox.display(“winner”);
   } else {
      count++;
      messageBox.display(“looser”);
   }
}
```

Labeling Methods

```
int {*->*; *<-*} secret;
int {*->client; *<-*} count;
...
void guess{*->client}(int i)
where authority(*)
{
   if (i == secret) {
     messageBox.display(“winner”);
   } else {
     count++;
     messageBox.display(“looser”);
   }
}
```

Declassify

```
int {*->*; *<-*} secret;
int {*->client; *<-*} count;
...
void guess{*->client}(int i)
where authority(*)
{
   if (i == secret) {
     declassify({*->*} to {*->client})
       messageBox.display(“winner”);
   } else {
     count++;
     declassify({*->*} to {*->client})
       messageBox.display(“looser”);
   }
}
```

Endorse

```
int {*->*; *<-*} secret;
int {*->client; *<-*} count;
...
void guess{*->client}(int i)
where authority(*)
{
   if (i == secret) {
     declassify({*->*} to {*->client})
       messageBox.display(“winner”);
   } else {
     endorse(i, {*<-client} to {*<-*})
       count++;
     declassify({*->*} to {*->client})
       messageBox.display(“looser”);
   }
}
```

Next Step: WebIL

- Analyze source and verify the security specified by labels
- Determine what data/computations can go where by transforming to WebIL
- Not committing to placement yet
- **This step enforces and guarantees our security model expressed in source**

Example: Source to WebIL

```
int {*->*; *<-*} secret;
int {*->client; *<-*} count;
...
void guess{*->client}(int i)
where authority(*)
{
     if (i == secret) {
          declassify({*->*} to {*->client})
               messageBox.display("winner");

     } else {
          endorse(i, {*<-client} to {*<-*})
               count++;
          declassify({*->*} to {*->client})
               messageBox.display("looser");
     }
}
```

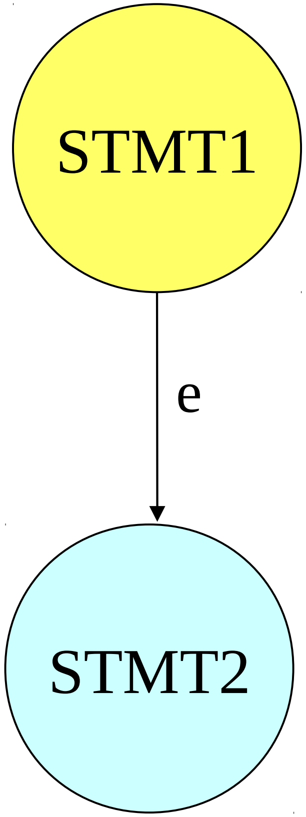
```
Sh       int secret;
C?Sh     int count;
         ...
         void guess(int i)
         {
Sh            if (i == secret) {
C                  messageBox.display("winner");
              } else {
C?Sh               count++;
C                  messageBox.display("looser");
              }
         }
```

C – Client
S – Server
? – Optional
h – high integrity

Next Step: Partitioning

- Goal: Place code to optimize performance without harming security
- Main cost is network traffic
- Approach:
  - Approximate weighted control flow graph over the whole program
  - Translate into an integer programming problem
  - Reduce to maximum flow problem
  - Solve

Partitioning



Message cost of edge e:

$W_e * (X_e + Y_e)$ where

$X_e = 1$ if STMT1 is on Client and STMT2 is on Server

$Y_e = 1$ if STMT1 is on Server and STMT2 is on Client

Minimize sum of message costs over the whole CFG while keeping the security policies in place

See the paper for more details...

Result of Partitioning

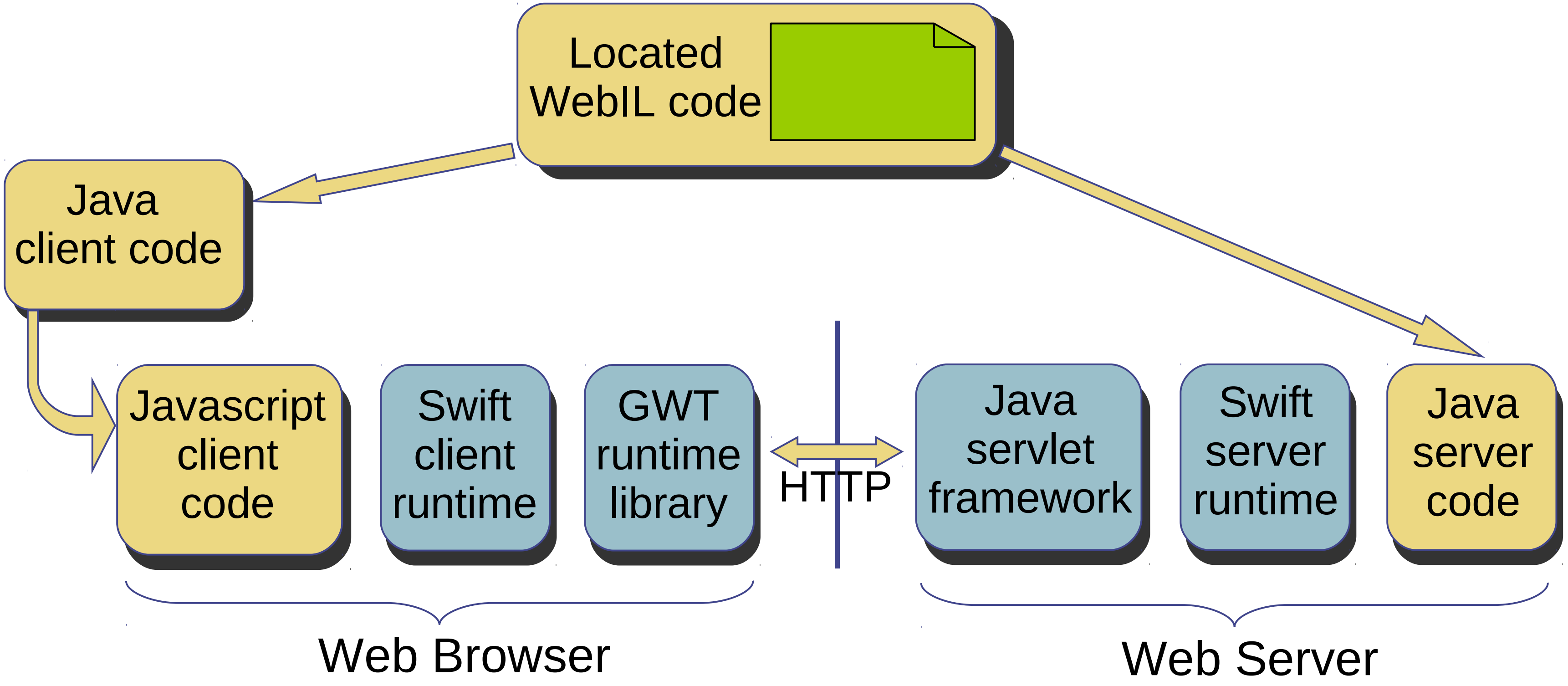
```
block1 (S):
    if (i == secret) goto block2;
    else goto block3;

block3 (SC):
    count++;
    goto block4;

block5 (SC):
    // end
```

SERVER

```
block2 (C):
    messageBox.display(“winner”);
    goto block5;

block3 (SC):
    count++;
    goto block4;

block4 (C):
    messageBox.display(“looser”);
    goto block5;

block5 (SC):
    // end
```

CLIENT

Last Step: Translation



Located WebIL code

Java client code

Javascript client code

Swift client runtime

GWT runtime library

HTTP

Java servlet framework

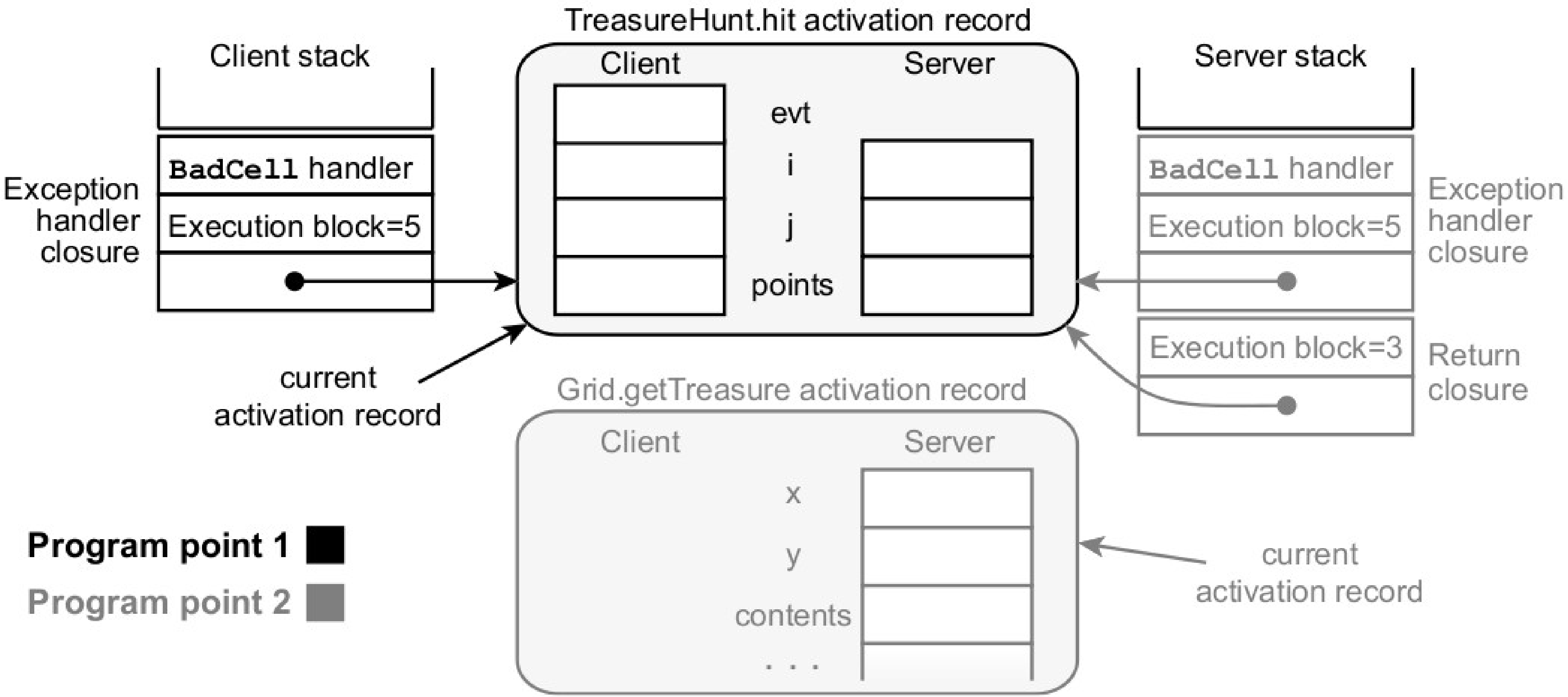
Swift server runtime

Java server code

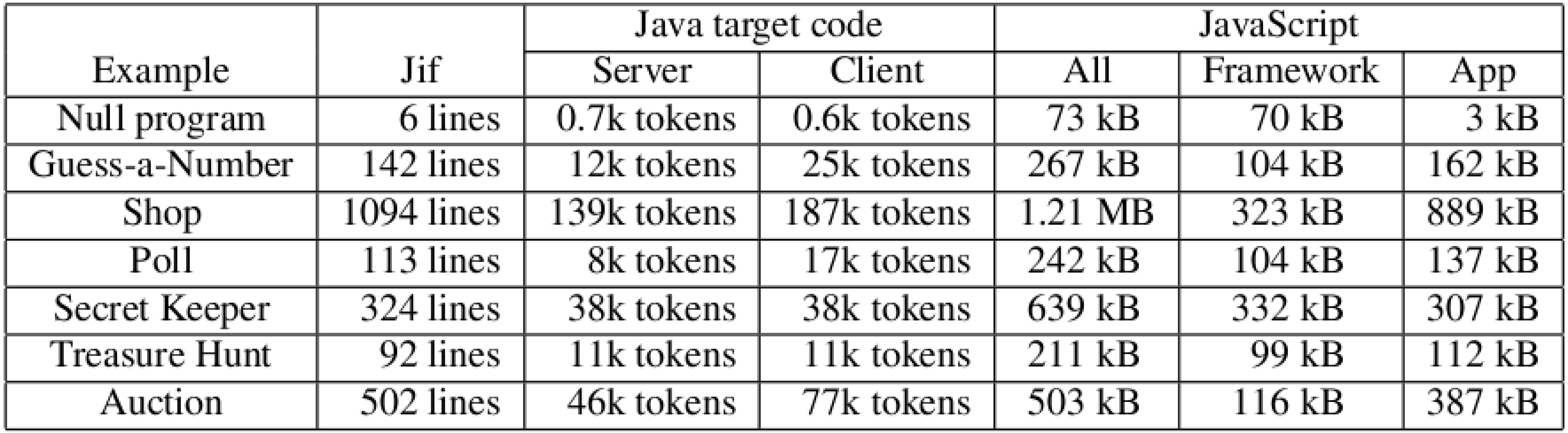
Web Browser

Web Server

Swift Runtime



Evaluation



Example	Jif	Java target code		JavaScript		
		Server	Client	All	Framework	App
Null program	6 lines	0.7k tokens	0.6k tokens	73 kB	70 kB	3 kB
Guess-a-Number	142 lines	12k tokens	25k tokens	267 kB	104 kB	162 kB
Shop	1094 lines	139k tokens	187k tokens	1.21 MB	323 kB	889 kB
Poll	113 lines	8k tokens	17k tokens	242 kB	104 kB	137 kB
Secret Keeper	324 lines	38k tokens	38k tokens	639 kB	332 kB	307 kB
Treasure Hunt	92 lines	11k tokens	11k tokens	211 kB	99 kB	112 kB
Auction	502 lines	46k tokens	77k tokens	503 kB	116 kB	387 kB

Discussion

- The good
  - Security policies are explicit and checked
  - One program makes it easier to reason about security
  - Minimizes network traffic
- The bad
  - Labeling is verbose and slightly confusing (~20-30% of lines are annotated)
  - Difficult to retrofit legacy code (that may already be split)
  - Still only as good as the programmer's ability to reason about security

Do Better?

- Can we ensure confidentiality and integrity without creating extra work for the programmer?
- Confidentiality – **no** (why not?)
  - programmer must mark confidential information as being such
- Integrity – **yes**

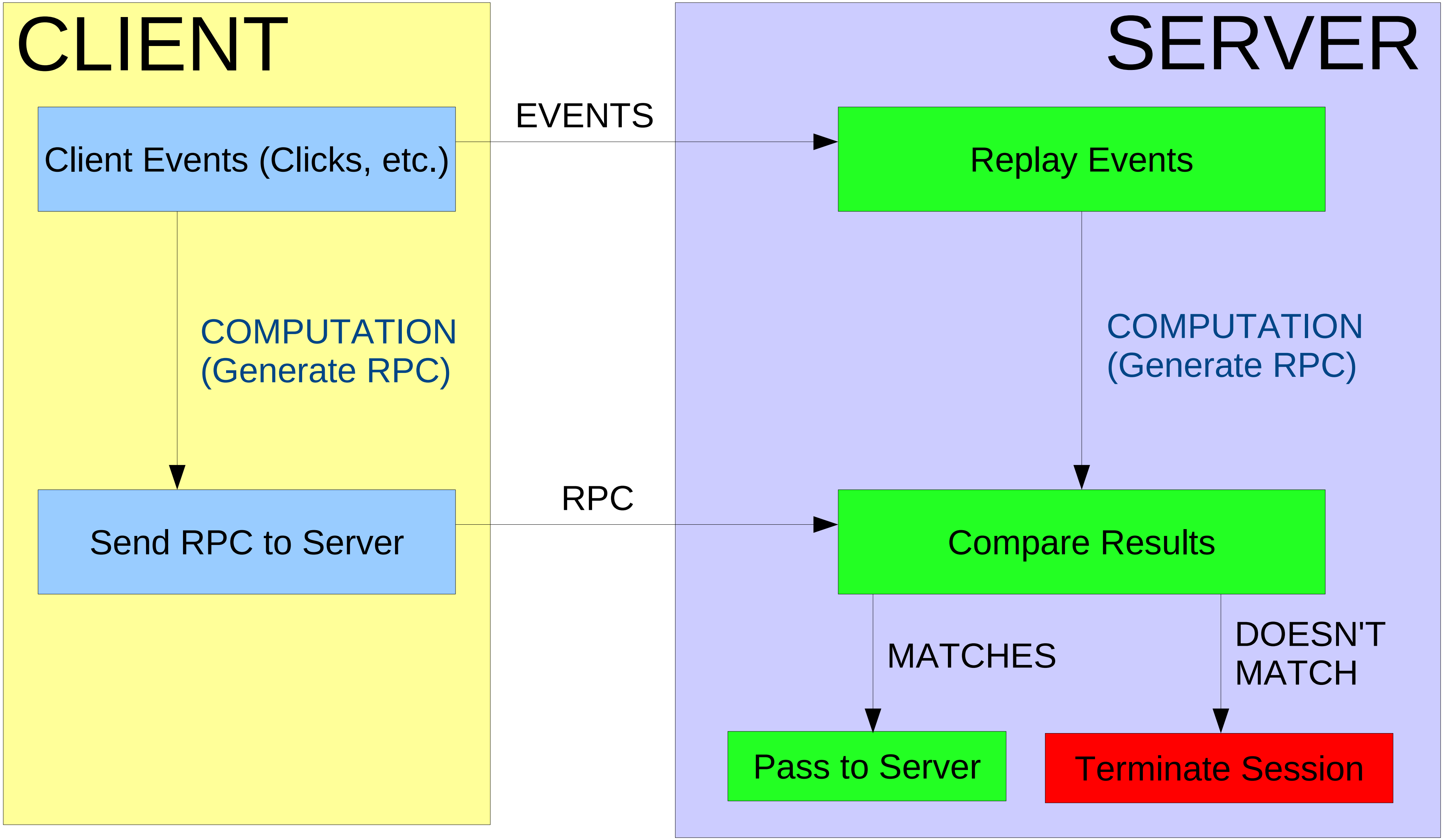
Integrity

- Integrity is directly tied to the location of computation
- Solution: move untrusted computation to trusted location
  - Move client-side computation to server
- But then we loose performance...
  - Better: **replicate** client-side computations on server, compare results

Ripley: Automatically Securing Web 2.0 Applications Through Replicated Execution

K. Vikram, A. Prateek, B. Livshits

Ripley Execution Model



CLIENT

Client Events (Clicks, etc.)

COMPUTATION
(Generate RPC)

Send RPC to Server

EVENTS

RPC

SERVER

Replay Events

COMPUTATION
(Generate RPC)

Compare Results

MATCHES

DOESN'T
MATCH

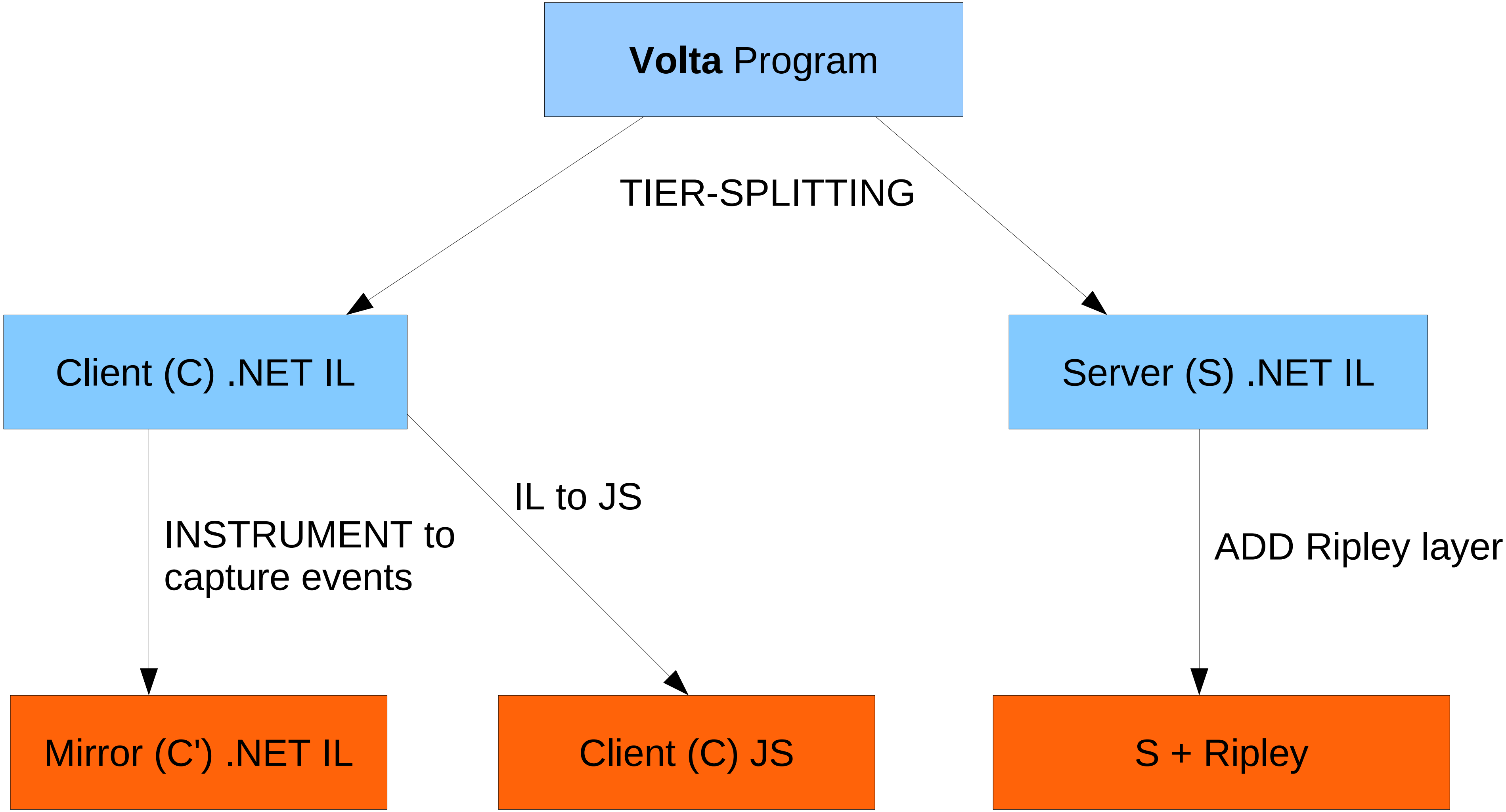
Pass to Server

Terminate Session

What Ripley Does and Doesn't Do

- Ripley ensures integrity of Client-run code
- Protects against any attacks involved in manipulating client-side code/state that results in malformed RPCs
  - Remember example from before
- Not for protecting against malformed input that is accepted by the client and server code
  - Ripley protects the developer-intended protection for the application

Implementation Workflow



Volta Program

TIER-SPLITTING

Client (C) .NET IL

Server (S) .NET IL

INSTRUMENT to
capture events

IL to JS

ADD Ripley layer

Mirror (C') .NET IL

Client (C) JS

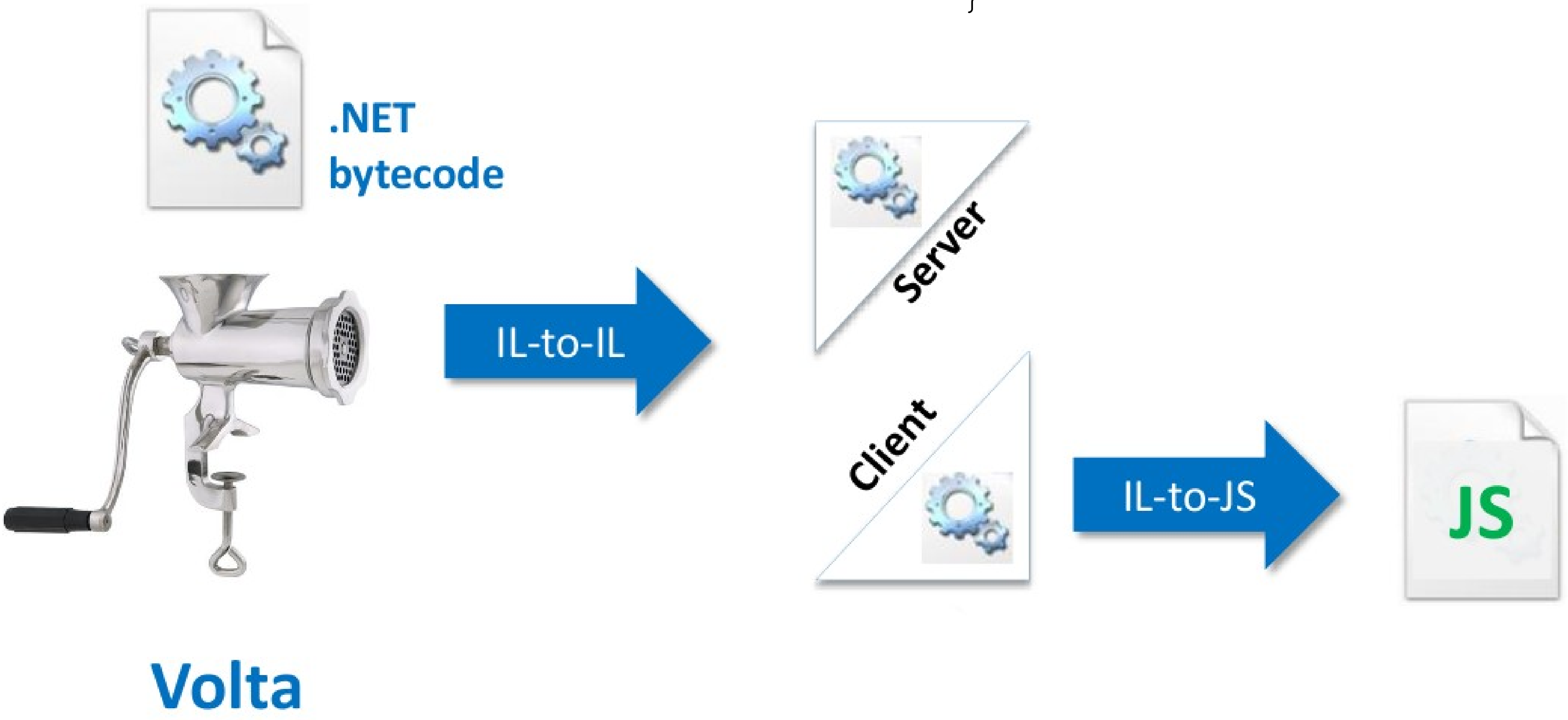
S + Ripley

Volta

Writing **one** program (similar to Jif)

```
class C1 {
    void foo(){...}
    void bar(){...}
}
```

```
[RunAt(“server”)]
class C2 {
    void baz(){...}
    void faz(){...}
}
```

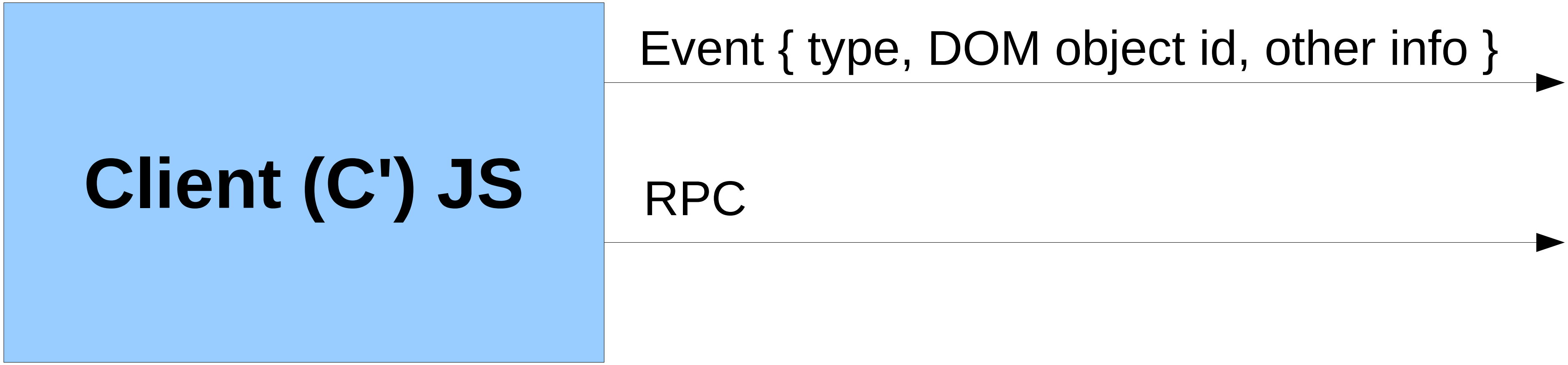


Volta Advantages

- Write one program
- Glue together with RPCs
- No funny JS (e.g. innerHTML editing)
- Produces fast Client mirror (C) in .NET IL

C' Instrumentation

- Intercept *primitive* and *custom* events via bytecode rewriting
- Transfer events to server
  - Batch: Queue events, send packet-sized set
  - Flush queue when client generates an RPC call

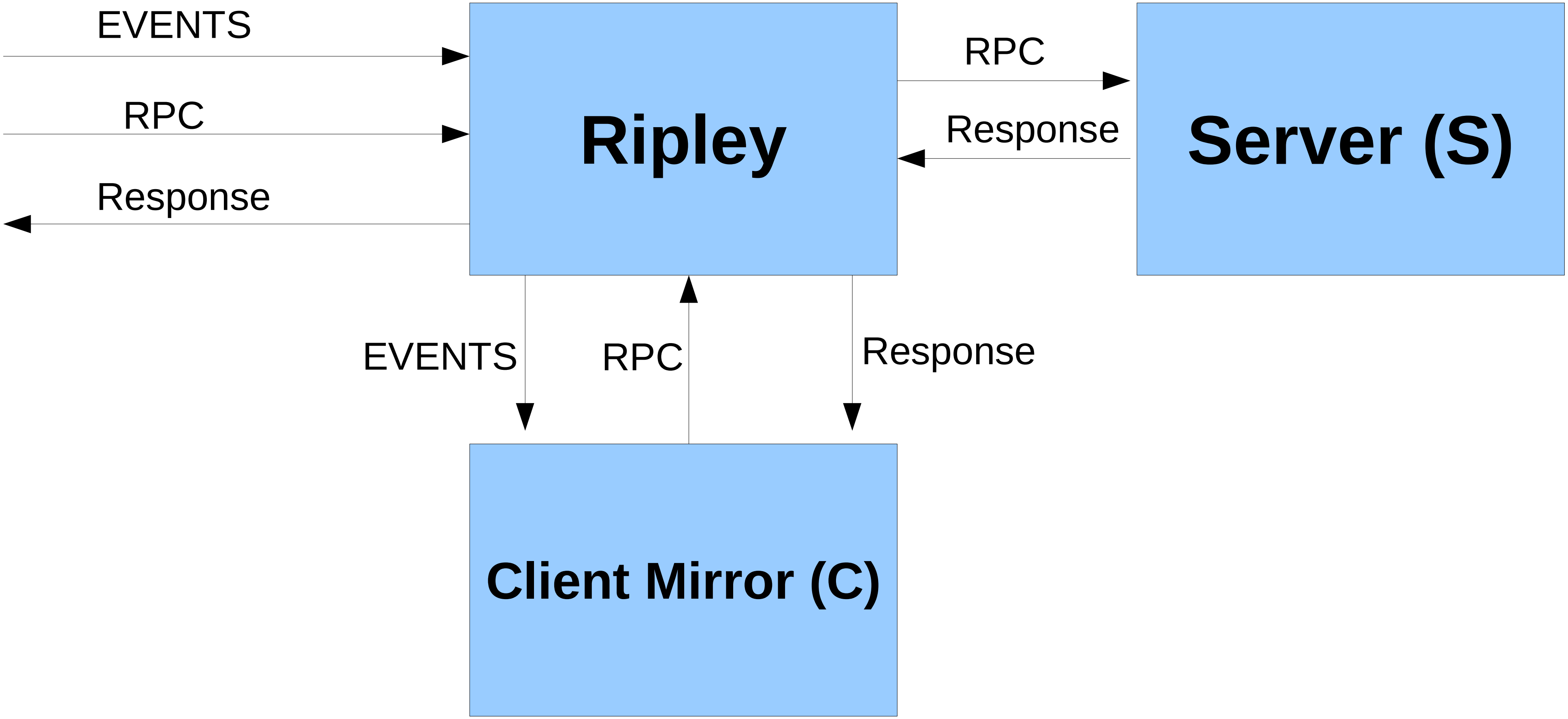


Client (C') JS

Event { type, DOM object id, other info }

RPC

Adding Ripley Checker to S



Replicating C' execution at C

- Run C as Jit-ed .NET IR
- Use lightweight browser emulator
    - headless
    - no rendering or layout computation
    - simple DOM manipulation interface
    - Each DOM node has unique id for event replay

Evaluation

Hotmail

Extras and Optimizations

- 0-latency RPCs
- MAC-ing RPCs
- Dependency analysis

Discussion and Conclusion