Ripley: Automatically Securing Distributed Web Applications Through Replicated Execution

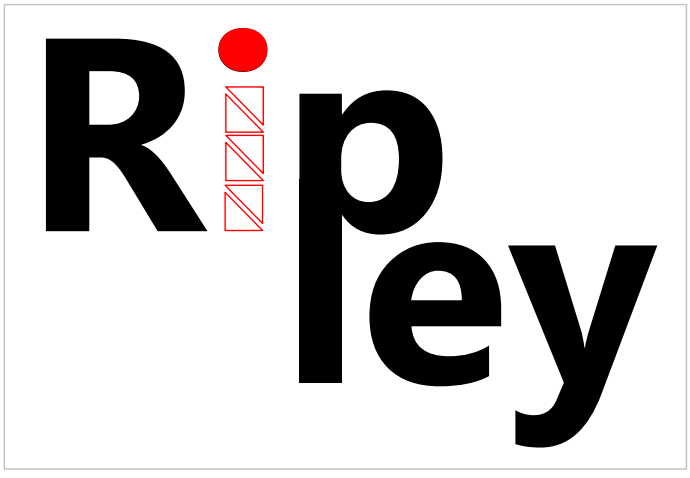
Benjamin Livshits
Microsoft Research

Abhishek Prateek
IIT Delhi

K. Vikram
Cornell University

November 18, 2008

Microsoft Research Technical Report MSR-TR-2008-174



Abstract

Rich Internet applications are becoming increasingly distributed, as demonstrated by the popularity of AJAX/Web 2.0 applications such as Hotmail, Google Maps, Facebook, and many others. A typical multi-tier AJAX application consists of a server component implemented in Java J2EE, PHP or ASP.NET and a client-side component executing in JavaScript. The resulting application is more performant and responsive because computation is moved closer to the client, and thus avoids unnecessary network round trips for frequent user actions.

However, once a portion of the code is moved to the client, a malicious user can easily subvert the client side of the computation and potentially jeopardize sensitive server state. In this paper we propose RIPLEY, a system that uses replicated execution to *automatically* preserve the integrity of a distributed computation. RIPLEY replicates a copy of the client-side computation on the trusted server tier. Every client-side event is transferred to the replica of the client for execution. RIPLEY observes results of the computation, both as computed on the client-side and on the server side using the replica of the client-side code. Any discrepancy is flagged as a potential violation of computational integrity. Our evaluation of RIPLEY on five complex and representative AJAX applications suggests that RIPLEY is a promising method for building secure distributed web applications.

1 Introduction

Web applications are becoming increasingly distributed, marked by the emergence of popular AJAX (asynchronous JavaScript and XML) applications such as Hotmail, Google Maps, Facebook, and many others. A typical multi-tier AJAX application consists of a server component implemented in Java J2EE or Microsoft .NET and a client-side component executing JavaScript in the browser. The resulting application is more performant and responsive, because computation is moved closer to the client, thus avoiding unnecessary network round trips. Unlike a computation performed entirely on the server, once a portion of the code is moved to the client, the overall computation can no longer be trusted.

Indeed, a malicious client can easily manipulate both data that resides *and* code that runs within the browser using one of many readily available data tampering or debugging tools [20, 32]. Imagine a JavaScript-based shopping cart within a typical e-commerce retail site such as Amazon.com that allows the user to add items, adjust their quantities, add coupons, compute the shopping cart totals, etc. When run on the client, this application can be compromised in a variety of ways. For instance, coupon validation checks can be dodged, allowing the user to reduce the total. Even simpler, the total computation can be compromised to set the total to an arbitrary, potentially even negative amount.

Due to the possibility of these attacks, almost every action in a typical shopping cart application today requires a round trip to the server, the latency of which can be quite noticeable, especially on mobile or long-distance connections. For non-malicious users, who constitute the majority, this unnecessary precaution leads to a much less responsive user experience.

Moreover the developer currently is responsible for splitting the application in a way that places all security-sensitive operations on the server. While some language-based approaches have recently been proposed to address this problem [5, 6], these techniques still require a great deal of developer involvement, making them difficult to use for existing large-scale projects.

In this paper we propose RIPLEY, a system that uses replicated execution to automatically preserve the *integrity* of a distributed computation, such as a typical AJAX application. RIPLEY replicates a copy of the client-side computation on the trusted server tier. Every user-initiated event is transferred to the replica of the client for execution. RIPLEY observes results of the computation, both as computed on the client side and on the server side using the replica of the client-side code. Any discrepancy is flagged as a potential violation of computational integrity. Note that RIPLEY is primarily designed to protect the integrity of distributed applications. RIPLEY does not remove the need for input validation nor does it eliminate confidentiality concerns. Nonetheless, RIPLEY automatically protects the application without requiring the developer to reason about code placement and trust implications. Note that confidentiality and input validation are important orthogonal issues addressed by prior work [11, 19, 25, 31, 33, 40].

As shown in Figure 1, with RIPLEY, a distributed Web application can combine the best of both worlds: the application is still responsive because of client-side execution, but the results of this execution do not have to be trusted because they are replayed on

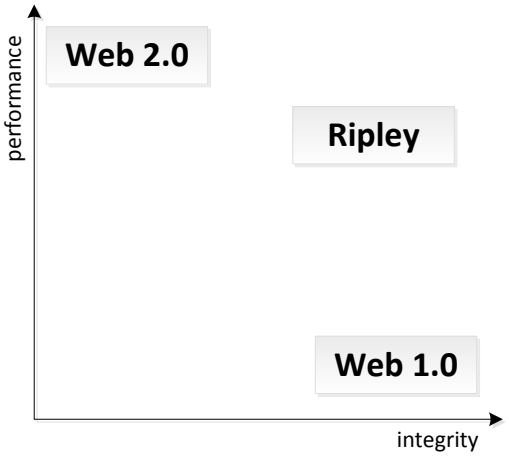


Figure 1: Performance vs. integrity trade-off for various Web application execution models. RIPLEY combines the best of both.

the server. In other words, the integrity of the overall distributed computation is the same as *if the application had been run entirely on the server*. In certain cases RIPLEY can even lead to better performance: since the application is replicated on the server, and the server is typically faster than the client, the client replica running on the server enables it to anticipate RPCs from the client in advance. This helps it to prepare and send the reply to the client ahead of time. In the best case, the client has the illusion of the server taking zero time for executing the RPC.

RIPLEY capitalizes on a recent trend towards distributing compilers such as Links [8], Hilda [42], Swift [5], and GWT [10]. Distributing compilers allow both the client- and the server portion of the distributed application to be developed together. We have closely integrated RIPLEY with Volta [30], a distributing compiler that splits .NET applications, translating them into JavaScript as needed. Integration with Volta significantly simplifies the process of code replication because the entire application is given to the Volta compiler at compile time. RIPLEY also integrates into the RPC infrastructure of Volta, making the process of communication between RIPLEY components on different tiers convenient. However, the ideas of RIPLEY are fully applicable to Silverlight or regular AJAX applications.

1.1 Contributions

This paper makes the following contributions:

- RIPLEY provides a practical and effective solution for the to the pressing problem of preserving the computational integrity of distributed Web applications. Unlike previously proposed approaches, RIPLEY requires no developer involvement, and automatically enforces application integrity at runtime.

- RIPLEY not only protects against malicious users, it is also able to protect benign users affected by a hostile execution environment. For instance, RIPLEY is able to prevent a a JavaScript worm from spreading through an application such as a social networking or a blogging site.
- We propose a number of performance optimizations that alleviate the CPU and network overhead imposed by the use of the replication technique
- To demonstrate the practicality of our approach, we evaluate the effectiveness and overhead of RIPLEY on five realistic and representative security-sensitive Volta applications. RIPLEY is able to foil security attacks with a low runtime overhead.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 summarizes the threat model and provides an overview of RIPLEY architecture. Section 3 gives a detailed description of RIPLEY implementation choices. Section 4 describes the results of applying RIPLEY to five security-sensitive AJAX applications. Section 5 presents a discussion of RIPLEY design. Section 6 presents related work and Section 7 concludes.

2 Overview

We first present an overview of Volta in Section 2.1 to make our description of RIPLEY more concrete. Although RIPLEY does not have to rely on Volta to work, integrating with Volta makes our approach considerably cleaner. The techniques in RIPLEY are in fact suitable for any distributed Web application written in JavaScript, Silverlight or Flash once the client logic is mimicked on the server. Section 2.2 presents a summary of RIPLEY architecture. Finally, Section 2.3 presents the threat model and shows the security assurance offered by RIPLEY against various threats that exist in today's distributed Web applications.

2.1 Volta Background

While the RIPLEY approach can be used for general AJAX-based Web applications, integrating with Volta provides a number of clear advantages. As illustrated in Figure 2, the Volta compiler is a distributing compiler that takes a .NET application as input and *tier-splits* it into a client and a server component by replacing appropriate cross-tier method calls with AJAX RPCs. Data is serialized before being sent to the server and deserialized on the server once received. A similar serialization-deserialization happens when the server returns control to the client. The client-side component is translated into JavaScript for execution within a standard browser [27].

Volta requires the developer to declaratively define which portion of the application runs on the server and which part on the client with the help of class-level annotations. Tier-splitting is performed subsequently as a .NET bytecode rewriting pass that reads the placement annotations, introducing RPCs as needed. To implement RIPLEY, we

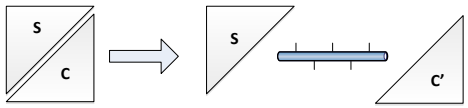


Figure 2: Tier-splitting in Volta: an application is split into a server-side component S and a client-side component C. The client-side component C is translated into JavaScript C' to be run within the browser.

have augmented the Volta tier-splitter to perform additional rewriting steps described in Section 3. We have also augmented the base Volta libraries to provide support for browser emulation, as described in Section 3.3. Note that, while relying on the Volta compiler and runtime makes our implementation easier, the RIPLEY approach does not require Volta: we could have implemented RIPLEY on top of Silvelight or regular AJAX applications.

2.2 Architecture of RIPLEY

The architecture of RIPLEY is shown in Figure 3. RIPLEY adds to Volta in the following key ways:

1. **Capture user events**: RIPLEY augments the client to capture user events within the browser.

2. **Transmit events to the server for replay:** The client run-time is modified to transmit user events to the client's replica C for replay.

3. **Compare server and client results:** The server component S is augmented with a RIPLEY *checker* that compares arriving RPCs m' and m received from the client C' and server-based client replica C, respectively, looking for discrepancies.

These steps are described in detail in Section 3. In summary, RIPLEY relies on re-execution to produce the correct result within C based on user events that it receives, effectively ignoring malicious data and code changes that occur on the client. If the malicious changes result in different RPCs issued to the server, RIPLEY will flag a potential exploit and terminate that client's connection. The re-execution takes place within client replica that runs in .NET on the server. For efficiency and scalability, we run the replica within a *browser emulator* instead of a full-fledged browser, as described in Section 3.3.

2.3 Security Threats and Assurances

While Volta and similar distributing compilers [8, 36, 42] propose a powerful programming model for distributed application development, moving execution to the untrusted client tier clearly *diminishes* the security of the resulting distributed application compared to the single-tier original [12]. It is the primary goal of RIPLEY to *restore* the level of security that has been lost. Note that RIPLEY does not try to enhance the security beyond that: a SQL injection [1] or a cross-site scripting vulnerability [3, 9] in the

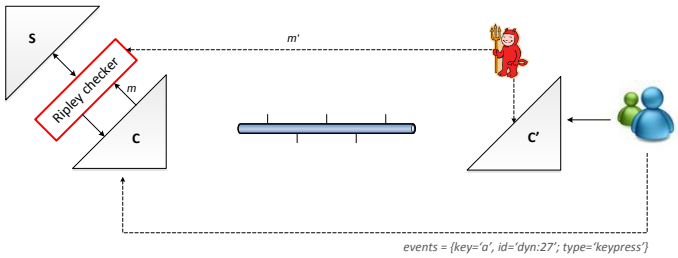


Figure 3: Architecture of RIPLEY: user events are delivered to both the JavaScript client-side component C' and its server-side replica C. RPCs arriving from both to the server component S m and m' are compared by the RIPLEY checker.

original application will persist in the distributed version; reliance on RIPLEY does not negate the need for input sanitization. However, with RIPLEY, we ensure that distributing the application will not worsen the application security posture. For instance, input sanitization checks are automatically replicated on the server, ensuring that a malicious client cannot bypass them. Replicating such checks also assures that the client- and the server-side sanitization checks are consistent with each other.

2.3.1 Threat Model

In this section we consider some of the typical threats against distributed Web applications that exist today and how RIPLEY addresses them; the reader is referred to [13] for more details about specific vulnerabilities and exploits.

Data manipulation. The most obvious kind of attack against a distributed Web application involves manipulation of data that is sent to the server. As in the shopping cart example in Section 1, where the cart total could be forged easily, any piece of data that is transferred to the server can be easily manipulated within the browser using one of many readily available data tampering and debugging tools [22]. Moreover, the integrity of data may also be compromised on the wire by a man-in-the-middle attack.

Not only can a malicious client change existing data before it is sent over to the server, it can also choose to manufacture new messages. If you consider the interface the server exposes as a set of commands, the client may choose to "drive" the server by invoking any of them out of order, potentially violating internal application logic.

Protection: As mentioned above, RIPLEY uses re-execution to produce the correct result within C based on user events that it receives, effectively ignoring malicious data changes that occur on the client. If the malicious changes result in discrepancies in the RPCs, this will cause RIPLEY to flag a potential exploit.

Code manipulation. The code sent over to the client can be easily edited within the browser to produce a variety of undesired effects. For instance, consistency or input validation check can be easily removed, which is why these checks have been traditionally relegated to the server, thus making even the benign users incur a round trip overhead. In a game application, the user may manipulate the code to make it possible

to circumvent the rules of the game. Often these changes are as simple as replacing the conditional of an `if` statement with `true` [14].

In a language as dynamic as JavaScript, code changes may affect not only the current application, but others running within the same interpreter. A prime example of this is the prototype hijacking vulnerability [4], where a malicious widget in a mashup overrides the `Array` constructor, thus allowing it to snoop on any of the other widgets.

Protection: Note that RIPLEY does not try to prevent code tampering in general; indeed, adding a semicolon that does not change the program semantics will never be detected by RIPLEY. However, RIPLEY *will* in fact prevent code modifications that result in different RPCs being issued by the client.

Script injection and JavaScript worms. While the threats above deal with the case of a malicious user, RIPLEY can actually help detect situations when benign users are affected by a malicious environment. Two prime examples of such a situation are injection attacks such as cross-site scripting [9] and JavaScript worms [23], both of which allow for potentially malicious actions to be executed on part of an innocent user, as long as malicious activity results in RPCs to the server.

As an example, consider an auction site such as eBay.com where users are either buyers or sellers. A malicious seller may embed JavaScript in the item description page so that when the item description page is viewed, a bid would be placed automatically on behalf of the viewer. Another common case is a worm on a social networking site such as the Samy worm on MySpace.com [38]. When a particular page was viewed, a hidden embedded malicious script would add the viewer as Samy's MySpace friend, which will result in a extra RPC easily spotted by RIPLEY.

Protection: RIPLEY protects against script injection in a particularly elegant way: referring to Figure 3 the replica C executes in the .NET CLR, not JavaScript, thus rendering injected JavaScript code non-executable when run within C. So, if the example above, the client-side component C' will produce an RPC which will not even be issued by C, thus causing RIPLEY to immediately observe a discrepancy.

2.3.2 Underlying Assumptions

The basic assumption throughout this paper is that anything executing on the server tier is believed to be uncompromised and trusted, whereas the client tier may be compromised. For simplicity, we assume that program execution is deterministic. Clearly, allowing non-determinism will lead to differences in the execution of C and C' that are not captured by RIPLEY, thus resulting in false positives. Fortunately, there is a way to "virtualize" sources of randomness, as discussed in Section 5. For instance, if a random number generator is used, the client can block its execution until it gets the random number from the server. Similarly, for a computation that accesses local time, the server component can block until the time measurement arrives from the client.

2.3.3 Security Assurance and Integrity Guarantees

The key focus of RIPLEY is to provide assurance to application developers or deployers. RIPLEY does not eliminate the need for input sanitization, however, the ability

to not worry about the placement of sanitizers illustrates the convenience of the RIPLEY model: a sanitizer check will be first performed on the client and then re-executed within the replica. So, for a benign user unintentionally supplying malformed input, the check will fail quickly on the client.

Unlike with some prior work, it is not our goal to convince the user that a particular security policy is satisfied within the application; this is the focus on much recent work in language-based security [5–7]. Neither it is our focus to ensure that the user is communicating with the right application or that the browser or user machine are uncompromised. This can be accomplished through remote attestation methods [21]. Also, man-in-the middle attacks can be addressed with SSL.

The key observation about RIPLEY model is that the execution that is trusted takes place *entirely* on the server. The RIPLEY server and replica pair execute based on the event stream received from the client. The client-side component is only there to enhance the responsiveness of the application. It is possible for the client-side state to deviate from the replica state; this may not be noticed until the next RPC or ever, if that difference does not affect RPCs at all. However, we are not concerned with preserving the client-state of a malicious user. We are, however, concerned with preserving the *externally observable behavior* of the application, which might include database queries, file system operations, etc.

As a result, RIPLEY ensures that a distributed Web application has the same observable behavior as the application that is run entirely on the trusted server tier, as it would have been in a Web 1.0 application. We assume that the Volta translation (RPC introduction, etc.) preserves the original application semantics. We also assume that that the emulator further described in Section 3.3 is not going to faithfully represent key portions of the client state such as the DOM and cookies. Given enough assumptions about the original-to-Volta program and Volta program-to-Ripley program mappings, we can for example argue that server is connected to an external store, such as a SQL database, running a RIPLEY-protected version of the application and a standalone version of the application will result in the same queries sent to the database. Formalization of this argument is part of future work.

3 Ripley Implementation

This section provides a deep dive into the RIPLEY implementation. Throughout this section we will find it helpful to refer to the following components shown in Figure 3:

- The server-side component S running in a .NET CLR within the Web application server;
- The client-side component C' running in JavaScript within the browser;
- The replica of the client-side component C running in a .NET CLR within the Web application server.

Overall, RIPLEY is implemented as an optional extension to the Volta tier-splitting process. This process takes the original application and produces S and C, optionally translating C into C' that runs in JavaScript. Integrating with the Volta tier-splitter allows RIPLEY to be implemented as several simple IL-to-IL bytecode rewriting passes.

```
// a custom button handler
this.button.Click += delegate {
    var name = this.userName.Value;
    var pass = this.passWord.Value;
    Login l = new Login();
    l.attempt(name, pass);
}
```

```
// our rewriter adds the following handler
this.button.Click += delegate {
    // capture the event
    HtmlEventArgs evt = this.Window.Event;
    // read target object ID
    var id = evt.__ObjectId;
    // event type: keyboard, click, etc.
    var type = evt.Type;
    // extra event-specific data
    var data = serializeData(evt);

    // enqueue event for transfer
    __ClientManager.
          enqueueEvent(type, data, id);
}
```

Figure 4: (a) A typical button on-click handler and **(b)** RIPLEY-generated handler for event interception.

Of course, from the standpoint of the developer, enabling RIPLEY on an existing Volta application is as easy as ticking a checkbox in a Volta project configuration. In the rest of this section, we shall describe each of the components above in detail.

3.1 C': Instrumenting the Client

Prior to being translated to JavaScript, the client binary C generated by the tier-splitter is rewritten to capture client-side user events.

3.1.1 Event Handling

In RIPLEY, events are classified into two kinds — primitive events and custom events. *Primitive events* include each key press and mouse click event, regardless of whether the application actually has registered any handlers for them. *Custom events* are those that the application has registered explicit handlers for. A typical handler for a button click event is shown in Figure 4a. Clearly, it is crucial to intercept these events on the client and relay them to C for replay.

Tracking primitive events helps maintain the state of crucial elements such as text areas and radio buttons. For instance, each keystroke a user types into an HTML form will produce a separate keyboard event that is intercepted by RIPLEY and transferred to the replica.

Note that we do not handle *all* JavaScript events that occur on the client; doing so would involve listening to *all* `MouseMove` events, for example, which occur every time the user repositions the mouse pointer. Clearly, this would be prohibitively expensive. The second reason is that our DOM emulation discussed in Section 3.3 is only an approximation of the real DOM and does not maintain information about the mouse position, etc. It is therefore conceivable that an application that relies on `MouseMove` events may break under RIPLEY. However, such an application is very likely to register a custom event handler for `MouseMove` events, which will lead RIPLEY to instrument these events properly.

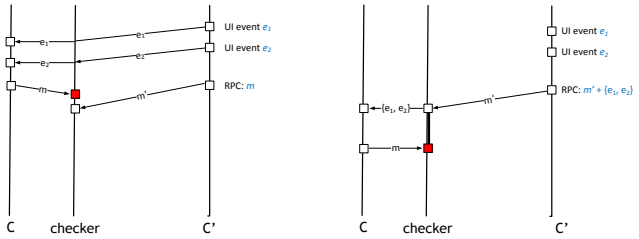


Figure 5: Eager (a) and lazy (b) event transfer. Events e_1 and e_2 arrive one after another. In **a**, they are sent over to the server right away. In **b**, they are queued-up and sent with the next RPC. While the overall completion time is later in the lazy case (as shown by a red filled square), one network message vs. three messages is used.

3.1.2 Event Interception

Primitive events are intercepted by registering a handler for each on the HTML `BODY` element. Since in the HTML event model, all events bubble up (or propagate) to the top-level document `BODY` element, it is a convenient point to intercept them. To intercept custom events, RIPLEY registers an extra handler shown in pseudocode in Figure 4b for each event of interest, via bytecode rewriting.

RIPLEY-generated event handlers enqueue details about the event into an application-specific queue. In addition to the event type (key press, key release, etc.), the serialized event details include the key code for keyboard-related events, mouse button information for mouse events, etc. Finally, the unique identifier corresponding to the DOM object which raised this event is also sent over, so that the event can be delivered to the corresponding DOM object within the replica.

3.1.3 Event Transfer

To reduce the number of round trips to the server, which is likely to become a bottleneck on high-latency connections, events are asynchronously relayed to the server in batches. Figure 5a and 5b show two scenarios of how events may be batched on the client and transmitted to the server. There is a natural trade-off between eager and lazy event transfer. As Figure 5a demonstrates, sending events eagerly will result in excess of network usage, which might be costly on a mobile connection, for instance, but will ensure speedy replication on the server. On the other hand, batching events longer as in Figure 5b would result in minimal network usage, but will delay the integrity checking and resulting server updates and responses.

To resolve this trade-off between responsiveness and network usage, we adopt a simple middle-path strategy. Events are batched until the queue reaches the maximum size of a network packet, in which case they are sent over immediately. Otherwise, whenever there is a RPC call, all events in the queue are flushed to the server.

3.2 S: Adding a Ripley Checker

RIPLEY modifies the server binary S to receive and properly handle events arriving

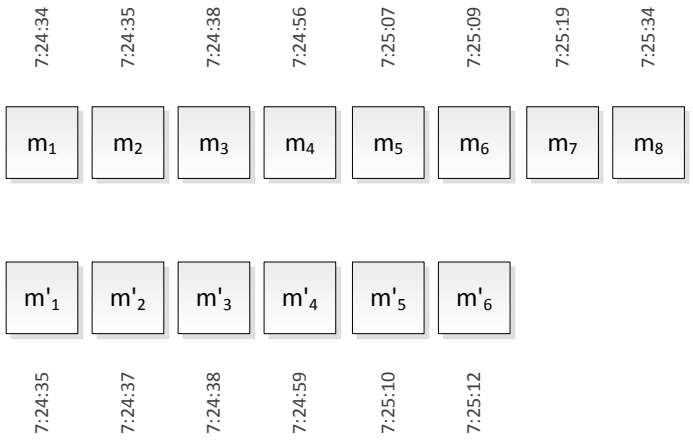


Figure 6: Audit logs from C and C'.

from the client and relay them to the client replica C for replay. Events are deserialized from the wire before being delivered to C. RIPLEY intercepts the RPCs that are received from both the JavaScript client and the replica and records them into *audit logs*, as shown in Figure 6.

By default, RIPLEY waits until it receives and compares RPCs m and m'. Only when they are equivalent does the runtime relay the RPC call to the application server code. The return response from the server is again intercepted as a string at the HTTP level. Copies of the response are relayed to both the client replica C and the actual client C' over the network.

Note that lock-step execution fashion is not the only option. Alternatively, RIPLEY could allow the server-side client replica C to move ahead, by relaying m to the server and sending back the response. When m' arrives, the server can confirm its equivalence with m. This is a likely scenario with well-provisioned servers and relatively slow clients. An alternative approach consists of keeping audit logs for messages arriving from both C and C' and to do periodic randomized cross-checking offering a lower overhead at the cost of a probabilistic integrity guarantee. Moreover, if RPCs are large, sending the entire RPCs is unnecessary — to save bandwidth, we can simply compute Message Authentication Codes (MAC) and send them over.

Since there could be multiple clients connected to the same server, the client replica C is executed in its own APPDOMAIN, a lightweight process-like abstraction in the .NET runtime [34]. At runtime, RIPLEY maintains a separate APPDOMAIN associated with each user session, and looks it up each time a batch of events is received from the client.

The main advantage of using separate APPDOMAINs is memory isolation: each uses its own heap and loads its own copy of dynamically linked libraries and maintains its copy of global data structures. Moreover, cross-APPDOMAIN communications are cheaper than inter-process communication in general as they do not require a process

context switch and AppDomains can share certain DLLs.

We should point out that on a multi-core machine the Ripley replicas can be put on the extra unused cores. In this architecture, it would also be desirable to co-locate the client-side replica on the same core as the server thread it is communicating with. We further address the question of server scalability in the next section.

3.3 C: Emulator and the Client Replica

By now, one questions begs to be asked: how are we going to scale a Ripley server? Not only are we running the existing server code, for reasons of security, we have also migrated client replicas for all clients connected to the server. Our goal of faithfully replicating the client execution on the server can be accomplished by running an instance of the actual full-fledged browser loaded with the application code on the server, one per user, as proposed by Deepfish [29]. However, for a popular and complex application, this approach is difficult to scale because the browser is a highly memory- and CPU-intensive piece of software.

There are two primary reasons that we believe that our solution will scale. First, we run the replicas in .NET instead of JavaScript, making it about two orders of magnitude faster. Second, we use a lightweight emulator instead of a full-fledged browser to reduce the memory and CPU utilization, as demonstrated in Section 4.2.

Much of the execution and state of the client does not affect the server state. For instance, any of the DOM rendering code or the state associated with the layout of the UI widgets do not feature in the application logic that updates application state on the server or the database. Clearly, such details can be abstracted away when we execute the client replica. We accomplish this by building a *browser emulator* that hosts the client replica C instead of an actual browser. The emulator is a lightweight browser that keeps track of the relevant UI state including the structure of the DOM and contents of editable elements. Since it performs no rendering or layout related computations, it avoids a lot of computation. As shown in Section 4.2, the memory footprint is an order of magnitude less for the emulator compared to a full browser.

The emulator is built as a dynamically linked library that exposes a DOM manipulation interface, with which the client replica C links at runtime. For reasons of efficiency, in addition to using the emulator, the replica is linked against a slightly modified Volta client runtime, that relays the HTTP requests to the server component S directly using a .NET method call instead of sending it over the network.

To ensure that the replica exhibits the same *observable behavior* as an actual JavaScript client, some further machinery is required. Relaying events to the right object within the replica is done by associating each DOM node with a unique ID. Each time a new DOM node is created, either on the actual client or on the replica, a new ID is created and stored within the node. Since the runtime behavior of the actual client and its replica is identical, new DOM objects are created in the same order, providing a deterministic mapping between DOM elements of the client and its replica.

When an event is raised on a client DOM object, the ID of the target object is sent over the wire to the replica on the server, as shown in Figure 3. The AppDomain hosting the replica maintains a lookup table of IDs-to-object references, which allows Ripley to identify the appropriate object instance to deliver the event to. The method

Benchmark	Lines of code C#	Lines of code JavaScript	Remote procedure calls (RPCs) with RIPLEY	Remote procedure calls (RPCs) w/o RIPLEY
Shopping Cart	594	698,832	one at checkout	on every cart update
Game of Sudoku	658	699,873	one at the end	on every game cell entry
Blog Application	341	699,071	submit, load each blog	submit, load each blog
Speed Typing Test	363	697,782	initialization, finish	on every word entry
Online Quiz	416	699,056	load questions, finish	on every question

Figure 7: Summary of statistics pertaining to the RIPLEY benchmark applications.

to be invoked on that instance and the parameters that need to be sent are provided as part of the event.

4 Experimental Results

This section is organized as follows. Section 4.1 discusses the benchmark applications we have built to test the ideas of RIPLEY. Section 4.2 focuses on the runtime overhead that RIPLEY imposes on application execution.

4.1 Benchmark Applications

In this section we describe the benchmark applications we used to test RIPLEY. All of these applications have been developed on top of Volta. A summary of information about these applications is given in Figure 7. All of these applications were originally developed in C# and (partially) translated into JavaScript by the Volta compiler. Columns 2 and 3 provide the line-of-code metric for the original application and the resulting JavaScript code. Note that the JavaScript code includes the translated versions of the required system classes that may be needed at runtime, which causes it to be quite substantial; if GWT experience is any indication, we expect code size to decrease drastically in subsequent Volta releases [24]. Column 4 shows the frequency of RPCs in the version of the application protected with RIPLEY. In most cases, there is only one RPC required at the end of the execution.

To put this into perspective, for each of our benchmarks we also consider an application that would have the *same strong integrity properties* written by hand or with the help of a compiler such as Jif [5, 6]. In the majority of cases, engineering such an application requires manually moving significant portions of the computation to the server to preserve the integrity. Column 5 shows the number of RPCs for such an application. Clearly, RIPLEY results in fewer RPCs for the same integrity guarantee.

4.1.1 Shopping Cart

As in a typical shopping cart within an e-commerce application, one can add and remove items to the cart, update their amounts, and eventually check-out. There is a provision for using coupons values where designed coupons `C5`, `C10`, and `C15` denote 5%, 10%, and 15% discounts, applied to the cart total. As described in Section 1, it

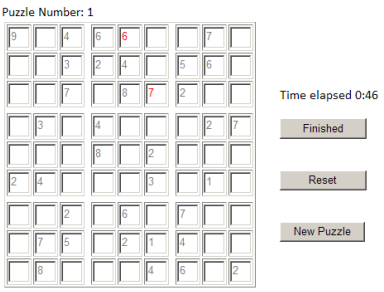


Figure 8: Sudoku UI: invalid entries are highlighted in red.

is typical for such an application to carry out the total calculation on the sever side, which means that every cart update results in a RPC to the server. Our shopping cart is implemented entirely on the client, with only one message containing the cart total sent to the server upon check-out.

Security threats: In many ways, the shopping cart application is a demonstration of typical client-side security threats described in Section 2.3. For example, a malicious user may attempt to manipulate the discount computation by using invalid coupons or using the same coupon multiple times. Better yet, the malicious client can just manually set the resulting total before it is sent over to the server without even touching the code.

Benefits of RIPLEY**:** With RIPLEY enabled, we can afford to do these computations on the client side, thereby preserving the application responsiveness. Since the user events are replicated on the server side, the server also maintains an abstract state of the cart, which includes values of various form fields, and can easily verify the total amount as soon as it is received from the client.

4.1.2 Game of Sudoku

This online game presents one of five hard-coded Sudoku puzzles for the user to solve. The solution is checked on the client and sent over to the server to be recorded for computing user ratings, etc. A sample Sudoku session is shown in Figure 8. As the game progresses, there are two kinds of validation checks being performed. After a number is typed into a game cell, a the row and the column is checked to look for repetitions; repeated numbers are flagged in red. When the user is ready to submit a solution, the entire grid is checked for validity.

Security threats: Both the local and the global validation checks of the game state can be easily bypassed by a malicious user, leading them to declare the puzzle as finished without even making an effort.

Benefits of RIPLEY**:** When the result of the game is submitted to the server, RIPLEY will check the validity of the final solution based on the event stream that it receives as input. A single RPC may be used to submit all the events at once without creating extra network traffic.

4.1.3 Blog Application

This online blog application allows the user to view a blog, and to post and edit blog entries.

Security threats: Unlike the previous two applications that address the issue of a malicious client, the focus here is on protecting the *benign* client from the effects of script injection and worm attacks. By default, the blog application does not perform extensive data sanitization, leaving itself open to cross-site scripting attacks. Worms can be used to amplify the effects of cross-site scripting. In the case of a blog, a worm may post a blog entry on behalf of an unsuspecting user.

Benefits of RIPLEY**:** In case of a JavaScript worm, when the worm tries to propagate by uploading executable contents to the server, it will do so by sending extra RPCs. Because the client replica runs on the server side in .NET, it is impervious to JavaScript code injection. As a result, the mismatch in the stream of RPCs will be detected by RIPLEY. Also, client-side checks can now be reliably performed on the client.

4.1.4 Speed Typing Test

In this application, a set of words are randomly chosen from a dictionary and displayed to the user as a paragraph. The objective for the user is to type as many words as she can within the time limit of one minute. The user's word-per-minute count and accuracy is calculated once the time limit has passed. As the words are typed in, their correct spelling is checked and highlighted on the fly. An interesting twist in this application is that events arrive at a very rapid rate, thereby stressing the performance side of RIPLEY.

Security threats: A malicious user may tamper with per-word spelling checks and also manipulate the time measurements to further rig the test.

4.1.5 Online Quiz

In this quiz application, trivia questions appear one by one, and depending on the correctness of the current answer, the next question is selected, of a higher or same point value, respectively. After answering a total of ten questions, the user's score is calculated and sent to the server for recording. The answer to each question consists of a single word.

In an online quiz application such as this, the answer would be sent to the server for checking after each question and the next question would be returned. This requires a round trip after every question, making the application less responsive. Moreover, if the quiz is timed, the round trip overhead needs to be properly taken into account. In contrast, our design moves the entire database of questions (62 questions total) to the client. The next question's selection is performed on the client, only a single RPC is required at the end.

Security threats: An interesting twist in the Quiz application compared to the ones above is that the confidentiality of the data on the client is important. Indeed, if the client can easily learn and enter the proper answers, cheating on the quiz would

	Application RPCs		Network overhead for event transfer					
					Uncompressed		Compressed	
Benchmark	**RPCs**	**Bytes**	**Events**	**RPCs**	**Total**	**Norm.**	**Total**	**Norm.**
Shopping Cart	1	157	13	1	1,548	119	300	23
Game of Sudoku	1	160	146	8	16,953	116	812	5.6
Blog Application	9	1,595	252	11	31,090	123	863	3.4
Speed Typing Test	4	1,598	556	28	63,945	115	1422	2.6
Online Quiz	2	275	66	4	7,801	118	445	6.7

Figure 9: Network overhead measurements after applying RIPLEY.

be trivial. In general, RIPLEY does not do anything to address confidentiality concerns, relegating these concerns to the developer. For this application, we use a simple confidentiality-preserving approach.

We only send hash values of the proper answers instead of the answers themselves. This allows us to compare hash values of the provided answers with the correct ones. We chose to allow for one-word answers to each question instead of multiple-choice; this way we avoid dictionary attacks, which would be trivial if the space of answers were small. Additionally, just as for the applications above, a malicious client can manipulate the solution checking code and related data.

Benefits of RIPLEY**:** As the entire application is run on the client side, integrity issues like bypassing solution checking etc can be handled by Ripley as the checks are replicated on the server side. RIPLEY cannot address confidentiality concerns in general, though.

4.2 Overhead Measurements

We focus on three dimensions of overhead: extra network utilization, extra CPU time, and extra memory utilization. Subsequent sections discuss these issues in turn.

4.2.1 Network Overhead

The first group of columns, columns 2–3, in Figure 9 shows the network usage of the application itself. Most applications in our benchmark suite send only a few RPC messages to the server. The Blog application has been written to produce one RPC per blog entry read, and so it uses more messages than other applications.

Columns 4–7 show the network overhead introduced by using RIPLEY. The "Total" column shows the total number of bytes and the "Norm." column shows number of bytes per event. Extra network activity is only due to transmission of event data to the server. Unsurprisingly, applications such as the Speed Typing and the Blog, that generate a lot of key strokes consume more network resources. However, network messages containing event data are sent asynchronously and thus do not significantly slow down the client-side execution. The bandwidth requirement is directly proportional to the number of events, as can be seen in the last column. All applications use up about 120 bytes per event, uncompressed.

Benchmark	Server checks			Event capture			
	Max	Min	Avg	Max	Min	Avg	Med
Shopping Cart	0.083	0.083	0.083	8	0	1.21	1
Game of Sudoku	0.462	0.462	0.462	87	0	1.25	1
Blog Application	0.079	0.002	0.012	8	0	0.676	1
Speed Typing Test	0.078	0.004	0.023	84	0	0.8	1
Online Quiz	0.078	0.004	0.041	162	0	3.044	1

Figure 10: CPU overhead measurements in ms after applying RIPLEY.

Fortunately, the event stream is highly compressible; applying GZIP compression reduces the size of a single event on the wire to just 3–4 bytes on average, as shown in columns 8–9. The effectiveness of compression is be most noticeable in highly interactive benchmarks such as Speed typing, reducing the number of needed network packets to just a single one in most cases. Unfortunately, the current generation of browsers do not support automatic compression of HTTP requests, only HTTP responses, requiring it to make compression part of Volta tier splitting, which is part of future work.

4.2.2 CPU Overhead

RIPLEY introduces CPU overhead on both the server and the client. Clearly, running the replica on the server also consumes CPU resources, despite the fact that running within the emulator described in Section 3.3 makes things considerably faster. RIPLEY assumes that we have enough server resources, such as extra processor cores, to be able to run the RIPLEY replicas without slowing down the server-side execution of the application. Further scalability issues under high client workloads are subject of future work.

RIPLEY checking on the server introduces some latency for regular requests, as shown in columns 2–4 in Figure 10. The server runs an ASP.NET application server on a dual-core 3 GHz machine with 4 GB of RAM running Microsoft Vista. We measure the time that each client-side request spends waiting for the replica to generate the corresponding request and to compare the two to verify its integrity. In most cases, the former component forms the bulk of the overhead, since the replica receives the event information in batches and needs time to catch up with the actual client.

The maximum overhead of the Shopping Cart and Sudoku is due to this effect, since the events are sent to the replica right before the `checkout` and `finishgame` RPCs are sent to the server. The maximum overhead for the other applications is observed during the application initialization phase and typically involves application-specific IO on the server. For instance, in the Blog application, it involves fetching the blog data and in the Quiz application, it involves initializing the database of questions. Despite this, this overhead is negligible for pretty much all benchmarks. The minimum times were fractions of milliseconds, since for these requests, the replica is already in sync when the requests arrive. The overhead is only due to the string comparison of messages.

Client-side instrumentation for capturing and serializing event information to the server adds execution overhead in the browser. The overhead is low as shown in columns 5–8 in Figure 10, typically about a couple of milliseconds on average. The

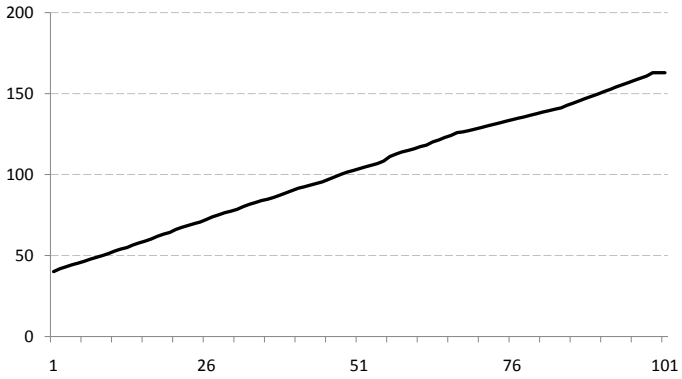


Figure 12: Memory footprint in MB as a function of the number of replicas.

client runs in an Internet Explorer browser, with a plugin that can run .NET CLR code. The extremes of minimum and maximum are shown to indicate the spread. The high extremes are believed to be statistical anomalies since the median overhead is 1 ms for all application, which is not noticeable for interactive GUI applications. This is the typical overhead we might expect, since the events are sent asynchronously to the server. Note that moving event capture to within the browser as discussed in Section 5 is likely to reduce the client-side CPU overhead even further.

4.2.3 Memory Overhead

In the presence of multiple replicas running alongside the server it is possible for the replicas to use up quite a bit of extra memory. Of course, the emulator is significantly less memory-consuming than running a full-fledged version of the browser.

To experimentally demonstrate this point we first considered a version of the Shopping cart application running without RIPLEY and then with RIPLEY enabled, with both Internet Explorer and Firefox running on the client. A summary of information about this experiment is shown in Figure 11. The table shows the range of memory utilization, in megabytes, for each version with and without RIPLEY; in most cases, more memory was allocated as the application progressed. We used Internet Explorer version 7.0.6001 and Firefox version 2.0.0.16 on Windows Vista to perform these measurements. The server memory utilization goes up by about 5 MB by adding the RIPLEY emulator. This is *an order of magnitude* cheaper than adding a full-fledged browser with a memory footprint of over 60 MB.

	Volta	RIPLEY
Server	23 – 26	27 – 32
Client (IE)	59 – 64	59 – 65
Client (FF)	69 – 77	69 – 78

Figure 11: Comparison of memory utilization.

Furthermore, we modified the server to create more client replicas to simulate the process of a multitude of clients that are simultaneously connected to the server. Figure 12 shows the server memory size as we increase the number of replicas to 100.

Because of DLL sharing across the different APPDOMAINs, the marginal cost of an additional replica is only about 1.3 MB compared to 5 MB. In the future we are planning to perform additional scalability studies by observing how the number of replicas affect the server throughput with and without RIPLEY.

5 Discussion

Section 5.1 further discusses our assumptions specified in Section 2.3.2. Section 5.2 addresses scalability issues.

5.1 Dealing with Non-Determinism

We can introduce fully deterministic execution with the help of additional instrumentation. The following sources of non-determinism are most common in Web applications.

Using the `Random` family of functions. JavaScript exposes a random number generator through function `Math.Random`. Clearly, unless additional measures are taken, the value returned by calls to this function on the client and the replica will disagree. A uniform approach to treating randomness is to perform the computation on one, "canonical" tier. In this case, we can instrument the client-side code C' to send the result of the call to `Math.Random` in the event stream. We can further instrument the replica C to block until the outcome of the random call is received. Once received, the result of the call is substituted in place.

Reading and measuring time. Access to time is provided through the `Date` object in JavaScript. Similarly to the approach described above, access to time routines can be instrumented and the replica can be blocked until the time measured on the client is delivered to continue the computation.

Accessing third-party servers. A systematic way to deal with accessing third-party servers is to require that these accesses be tunneled through the server. For servers in a different domain, this is necessary anyway, because of the same origin policy in JavaScript. This allows for easy centralized access to outside data for both the replica and the client-side code. Because calls to external services are performed only once, this also deals with the issue of non-idempotent calls with side-effects.

Browser Enhancements In fact, a set of small changes to the JavaScript interpreter would help us secure event capture and delivery and would also address the sources of non-determinism discussed above.

In particular, instrumenting `Math.Random` and `Date` routines as well as event handlers as described in Section 3.1 *in the interpreter* is the easiest and most systematic way to treat these issues that ensures that malicious JavaScript code co-existing within the same page is unable to gain access to this data. This effectively makes a portion of the browser or the JavaScript interpreter part of the trusted computing base. Since event capture is done outside of JavaScript, it will also ensure that the overhead of this instrumentation is low. To ensure that event streams are not tampered with, standard techniques such as Message Authentication Codes (MACs) can be used.

5.2 Performance and Scalability

RIPLEY enables the following optimizations.

0-latency RPCs. An advantage of the RIPLEY architecture is that, once computed, RPC results can be actively pushed to the client. This way, when the RPC is finally issued on the client, its result will already be available, leading to 0-latency RPCs. This demonstrates that not only does RIPLEY make the application more secure, in many cases it can also make it more responsive.

MAC-ing RPCs. To further reduce the network overhead we may send MACs (message authentication codes) of RPCs m' instead of their actual values.

Deployment strategy. RIPLEY meshes nicely with the traditional load-balancing approach to deployment of large-scale Web 2.0 applications. In particular, a load balancer could be used to repeatedly direct the same user to the server where both its replica and the corresponding server threads run. Currently, this functionality is implemented in the RIPLEY checker, which looks up the appropriate APPDOMAIN for a user session. Moreover, to save memory, both the server thread *and* the replica can be serialized on high server load for long-running sessions and then brought back from disk.

Dependency analysis. An important observation is that not the entire client-side code base has to be included in the replica. In particular, display code does not need to be executed because the replica is essentially "headless" — there is no user to see the GUI. To reduce the amount of code the replica must run, we can use a slicing analysis [39] to only include portion of the client-side code that contribute to values included into RPCs.

6 Related Work

The security of the web infrastructure has been a subject of much previous work. The various approaches to solving the problem can be categorized roughly along four lines of inquiry. A sizable body of literature has focused on the static analysis of web applications using techniques such as taint-checking. Runtime monitoring of web applications has also proved to be effective. Others have addressed the problem at a higher level by developing a cleaner and more secure programming model, often erasing the boundaries between various tiers. Recent work has also developed techniques to protect against untrusted clients in a networked environment. Finally, the idea of security through replication has also been well studied in earlier work. We elaborate further on each of these.

6.1 Analysis and Monitoring

There has been a great deal of interest in static and runtime protection techniques to improve the security posture of traditional "Web 1.0" applications. Static analysis allows the developer to avoid issues such as cross-site scripting before the application goes into production. Runtime analysis allows exploit prevention and recovery.

The WebSSARI project pioneered this line of research. WebSSARI uses combined unsound static and dynamic analysis in the context of analyzing PHP programs [16].

WebSSARI has successfully been applied to find many SQL injection and cross-site scripting vulnerabilities in PHP code. Several projects that came after WebSSARI improve on the quality of static analysis for PHP [19, 40]. The Griffin project proposes a scalable and precise sound static and runtime analysis techniques for finding security vulnerabilities in large Java applications [25, 28]. Based on a vulnerability description, both a static checker and a runtime instrumentation is generated. Static analysis is also used to drastically the runtime overhead in most cases. The runtime system allows vulnerability recovery by applying user-provided sanitizers on execution paths that lack them. Several other runtime systems for taint tracking have been proposed as well, including Haldar et al. for Java [11] and Pietraszek et al. [33] and Nguyen-Tuong et al. for PHP [31].

While server-side enforcement mechanisms are applicable for traditional Web applications that are composed entirely on the server side [19, 25, 40], Web 2.0 applications that make use of AJAX often fetch both data and JavaScript code from many sources, with the entire final HTML only available within the browser, making runtime client-side enforcement a natural choice. Recently, there has been a number of proposals for runtime enforcement mechanisms to ensure that security properties of interest hold for rich-client applications executing within the browser [7, 15, 18, 44]. Erlingsson et al. make an end-to-end argument for the client-side enforcement of security policies that apply to client behavior [7]. Their proposed mechanisms use server-specified, programmatic security policies that allow for flexible client-side enforcement, even to the point of runtime data tainting. Unlike RIPLEY, their technique can enforce some necessary, but not sufficient conditions for establishing distributed application integrity.

6.2 Web Programming Models

Tier-splitting has been proposed in setting other than Volta as a way to program distributed Web applications. Popular systems in this space include the GWT [10], Links [8], Hop [36], Hilda [42], etc. To the best of our knowledge, RIPLEY is the first realistic security solution for these kinds of frameworks.

BASS is a recent attempt to build security into a declarative high-level web programming model, working on the observation that security issues are often orthogonal to the main Web application logic [43]. It enables the programmer to specify the business logic of the application without needing to write the security related logic. Abstractions for common operations, such as form input, are baked into the model. Secure coding practices that prevent common attacks such as CSRF, XSS and session fixation are applied by the language compiler. A prototype implementation of the translation exists, but no applications seem to have been written in BASS. Ripley, on the other hand, is a realistic programming model integrated with a full-fledged Volta compiler being used for numerous real-world applications. Instead of protecting only against common exploits, Ripley defends against any client attack that attempts to compromise application integrity. BASS does not deal with client-side scripting at all, whereas Ripley works in a model where a significant portion of the application is run on the client for enhanced responsiveness.

6.3 Untrusted Clients

Protection against untrusted clients and eavesdropping over the network has received much attention, especially in the context of online gaming [14, 41]. In a distributed online game, part of the application workload is typically delegated to the clients and the server keeps track of only an abstract state of the game environment. As a result, the game is rendered vulnerable to malicious clients compromising the physical and logical rules governing the simulation in the game. Hacking popular online games is a financially viable undertaking as game "items" can be converted to real-world currency or sold on eBay.

Jha et al. propose a solution to the distributed online game integrity problem by performing random audits of the client state verifying that the client has not manipulated its state in violation of the semantic rules of the game [17]. Our approach, in contrast, provides a non-probabilistic guarantee of integrity at a potentially higher cost. In particular, if the client-side computation is highly CPU-intenstive, as ray-tracing in games tends to be, despite replying on an emulator and running in a faster .NET environment, with sufficiently many connected clients, the RIPLEY server might eventually become overwhelmed.

6.4 Replication for Security

Replication is a well-known way to increase security assurance, studies in file systems and replicated state machines [2, 26, 35, 37]. The work closest to ours is that of Zheng et al. [45, 46]. In many ways a precursor to SWIFT [5, 6], this work focuses on splitting programs while conforming to a set of integrity and privacy policies. The latter are addressed by computing in the hash space, not unlike our Quiz application in Section 4.

A high-level difference in philosophy with our work is that we avoid using annotations, believing that having to write annotations places an undue burden on the developer. For example, Zheng et al. report having about 3 annotations per line of code. Instead, we "blindly" replicate the entire client-side portion of the program on the trusted server tier, using runtime optimizations to make this approach scalable. Beyond our focus on computational integrity violations caused by malicious users, we also address the situation of a "malicious environment", such as a propagating JavaScript worm.

7 Conclusions

This paper presents RIPLEY, the first fully automated approach to ensuring integrity of distributed Web applications. To demonstrate the efficacy of RIPLEY in practice, we have applied the RIPLEY to five realistic AJAX applications. The performance overhead introduced by RIPLEY was minimal, in terms of CPU, memory, and network overhead. We have also formalized the approach taken by RIPLEY and proved that a RIPLEY-protected application has the same integrity properties as a non-distributed one. While we have demonstrated our ideas in the context of the Volta compiler, the

ideas of code replication can be easily extended to other runtime environments such as Silverlight or server-side JavaScript.

Our work closely follows the secure-by-construction philosophy of building application software. In particular, we envision RIPLEY becoming an integral part of the next generation of application servers. All the application developer will have to do in order to obtain the integrity-preservation benefits of RIPLEY, is to "drop" their Web application into the application server, with automatic replication becoming part of the deployment process.

References

[1] Chris Anley. Advanced SQL injection in SQL server applications, 2002.

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[3] CGI Security. The cross-site scripting FAQ. `http://www.cgisecurity.net/articles/xss-faq.shtml`.

[4] Brian Chess, Yekaterina Tsipenyuk O'Neil, and Jacob West. JavaScript hijacking. `www.fortifysoftware.com/servlet/downloads/public/JavaScript_Hijacking.pdf`, March 2007.

[5] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web application via automatic partitioning. *SIGOPS Operating Systems Review*, 41(6):31–44, 2007.

[6] Stephen Chong, K. Vikram, and Andrew C. Myers. Sif: enforcing confidentiality and integrity in Web applications. In *Proceedings of USENIX Security Symposium*, pages 1–16, 2007.

[7] Úlfar Erlingsson, Benjamin Livshits, and Yinglian Xie. End-to-end Web application security. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.

[8] Ezra Cooper and Sam Lindley and Philip Wadler and Jeremy Yallop. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects*, pages 266–296. Springer, 2007.

[9] Seth Fogie, Jeremiah Grossman, Robert Hansen, Anton Rager, and Petko D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, 2007.

[10] Google Web toolkit. `http://code.google.com/webtoolkit`.

[11] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, December 2005.

[12] Billy Hoffman. Ajax security dangers. `http://www.spidynamics.com/assets/documents/AJAXdangers.pdf`, 2006.

[13] Billy Hoffman and Bryan Sullivan. *AJAX security*. Addison-Wesley Professional, 2007.

[14] Greg Hoglund and Gary McGraw. *Exploiting Online Games: Cheating Massively Distributed Systems*. Addison-Wesley Professional, 2007.

[15] Jon Howell, Collin Jackson, Helen J. Wang, and Xiaofeng Fan. MashupOS: Operating system abstractions for client mashups. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.

[16] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing Web application code by static analysis and runtime protection. In *WWW*, 2004.

[17] Somesh Jha, Stefan Katzenbeisser, and Helmut Veith. Enforcing semantic integrity on untrusted clients in networked virtual environments. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 179–186, Washington, DC, USA, 2007. IEEE Computer Society.

[18] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the International World Wide Web Conference*, 2007.

[19] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting Web application vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[20] Adam Judson. Tamper data Firefox add-on. `https://addons.mozilla.org/en-US/firefox/addon/966`.

[21] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the USENIX Security Symposium*, pages 21–21, 2003.

[22] T. J. Klevinsky, Scott Laliberte, and Ajay Gupta. *Hack I.T.: Security Through Penetration Testing*. Addison-Wesley Professional, 2002.

[23] Benjamin Livshits and Weidong Cui. Spectator: Detection and containment of JavaScript worms. In *Proceedings of the Usenix Annual Technical Conference*, July 2008.

[24] Benjamin Livshits and Emre Kiciman. Doloto: Code splitting for network-bound Web 2.0 applications. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, September 2008.

[25] Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the Usenix Security Symposium*, 2005.

[26] Dahlia Malkhi and Michael K. Reiter. Secure and scalable replication in phalanx.

In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, page 51, 1998.

[27] Dragos Manolescu, Brian Beckman, and Benjamin Livshits. Redeeming distributed application development with recompilers. *IEEE Software*, October 2008.

[28] Michael Martin, Benjamin Livshits, and Monica S. Lam. SecuriFly: Runtime vulnerability protection for Web applications. Technical report, Stanford University, 2006.

[29] Microsoft Corporation. Microsoft Live Labs Deepfish. `http://labs.live.com/deepfish/`, 2006.

[30] Microsoft Corporation. Microsoft Live Labs Volta. `http://labs.live.com/volta/`, 2007.

[31] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening Web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, 2005.

[32] Parakey, Inc. Firebug, Web development evolved. `https://addons.mozilla.org/en-US/firefox/addon/966`.

[33] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the Recent Advances in Intrusion Detection*, September 2005.

[34] Jeffrey Richter. *CLR via C#*. Microsoft Press, 2006.

[35] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Compututing Surveys*, 22(4):299–319, 1990.

[36] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2.0. In *Companion to the Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 975–985, 2006.

[37] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised system. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 12–12, Berkeley, CA, USA, 2000. USENIX Association.

[38] The Samy worm. `http://namb.la/popular`.

[39] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[40] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of Usenix Security Symposium*, 2006.

[41] J. Yan. Security design in online games. In *Proceedings of teh Annual Computer Security Applications Conference*, 1993.

[42] Fan Yang, Jayavel Shanmugasundaram, Mirek Riedewald, and Johannes Gehrke. Hilda: A high-level language for data-driven Web applications. In *Proceedings of the International Conference on Data Engineering*, page 32, 2006.

[43] Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. Better abstractions for secure server-side scripting. In *Proceeding of the International Conference on World Wide Web*, pages 507–516, 2008.

[44] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *Proceedings of the Principles of Programming Languages Conference*, 2007.

[45] Steve Zdancewic and Andrew C. Myers. Secure information flow and CPS. *Lecture Notes in Computer Science*, 2028:46–??, 2001.

[46] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems, 2003.