Identifying information disclosure in web applications with retroactive auditing

Haogang Chen, Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek
MIT CSAIL

Abstract

RAIL is a framework for building web applications that can precisely identify inappropriately disclosed data after a vulnerability is discovered. To do so, RAIL introduces *retroactive disclosure auditing*: re-running the application with previous inputs once the vulnerability is fixed, to determine what data *should* have been disclosed. A key challenge for RAIL is to reconcile state divergence between the original and replay executions, so that the differences between executions precisely correspond to inappropriately disclosed data. RAIL provides application developers with APIs to address this challenge, by identifying sensitive data, assigning semantic names to non-deterministic inputs, and tracking dependencies.

Results from a prototype of RAIL built on top of the Meteor framework show that RAIL can quickly and precisely identify data disclosure from complex attacks, including programming bugs, administrative mistakes, and stolen passwords. RAIL incurs up to 22% throughput overhead and 0.5 KB storage overhead per request. Porting three existing web applications required fewer than 25 lines of code changes per application.

1 Introduction

Unintentional disclosure of sensitive information is a common problem, despite improvements in security techniques and widespread use of best practices. Newspapers frequently report such leaks at companies, hospitals, universities, government institutions, etc. This paper is based on the premise that disclosures will remain common, since even if the best security mechanism and practices are used, humans will make mistakes: a programmer may introduce a bug, a user may choose a weak password, or a system administrator may misconfigure the access control policy. Even if a state-of-the-art security system is in place, human operators can still overlook alerts [13], inadvertently disclosing confidential data.

Dealing with data leaks can be expensive because institutions are often required by law to inform their users of the security breach. For example, the University of Maryland suffered a compromise and paid for one year of credit monitoring for 309,079 potentially affected users, since it was unable to immediately pinpoint which of the users were actually affected [15]. However, a subsequent manual audit, which took about a month, revealed that only a handful of users' information was disclosed, and that the bulk of the cost was unnecessary. This example is typical of the challenges administrators face after a leak.

The usual approach for identifying data disclosures is to maintain access logs and to analyze those logs after a security breach, in an attempt to identify who accessed what data, and to separate out the legitimate accesses from the illegal ones. Although there are challenges in maintaining access logs (see, for example, Keypad [6]), the hard problem is deciding whether data accesses were legitimate or not. Manually auditing all accesses is labor-intensive and imprecise, as illustrated by the University of Maryland example.

To reduce the cost of handling leaks, this paper explores a different, automated approach for deciding which accesses were legitimate or not, based on record and replay. In particular, the paper describes the design of a new system, named RAIL (Retroactive Auditing for Information Leakage), that can precisely identify whose information was leaked in the context of web applications, such as a health care application that collects patients' personal health information or a class submission web site for assignments and grades.

RAIL's main contribution is to apply record and replay to identifying improper disclosures. Record and replay has been used for many *integrity* applications, from analyzing attacks [9] to detecting past intrusions [8, 14] and recovering integrity [2, 3, 7], but prior work did not address the problem of dealing with past data disclosures. During regular operation, RAIL records sufficient information so that it can faithfully replay an application's requests later. Once a vulnerability has been identified, an administrator repairs the underlying cause in the application (e.g., fixing a bug in the application's source code, or changing an access control list), and then asks RAIL to replay requests. If RAIL notices a difference between data sent to users in the original run and the replay run, it will report that data as having been inappropriately disclosed. For example, if one user's account was compromised, RAIL will report only the portion of that user's data that was inappropriately accessed by an adversary.

Precisely detecting data disclosures using record and replay is challenging for several reasons. The core challenge is that the application may behave differently during replay due to non-determinism. For example, a homework submission system might randomly assign students to one another for code review. If during replay some of the stu-

dents are missing (e.g., because they were the attackers), the system might produce an entirely different assignment for code review. As a result, the replay will send different homework submissions to each student, and RAIL might report all previous homeworks as having been inappropriately disclosed. Previous record and replay systems do not have adequate solutions to this problem; they take a best-effort approach, and any final state is acceptable in the end, as long as all effects of the attack are gone [2, 7]. In contrast, RAIL's goal is to minimize divergence between normal execution and replay, in order to precisely identify illegal data disclosures.

A second challenge lies in identifying what represents a data item in the first place. For example, in the homework submission system, what is the unit of data disclosure that should be reported to the administrator?

A third challenge lies in tracking dependencies in application code at a fine granularity (e.g., individual functions). Previous systems either tracked code dependencies at a coarse granularity (e.g., source files loaded by the application [2]), or made extensive changes to the interpreter to record fine-grained dependencies [8]. However, neither approach is ideal in practice.

Finally, a fourth challenge is making replay fast so that an administrator can quickly audit for data disclosures over long periods of time. One month of requests must not take a month replay.

RAIL addresses these challenges by providing an explicit API for developers to help administrators record and replay applications. For instance, in the homework submission system, the programmer uses RAIL's API to assign semantic names to random pairings between students (see §7.2), enabling the system to preserve assignments during replay, even if some students are gone. The API includes annotations to identify data, assign semantic names to non-deterministic inputs, and record dependencies on state for selective replay.

We implemented the RAIL API in the context of Meteor [11], a framework for building web applications. The API's design is not limited to Meteor. We chose Meteor because it cleanly separates data items and web interfaces via asynchronous messages. Because of this property (which is common in modern web frameworks), we were able to implement much of RAIL inside of Meteor, greatly reducing the need for application changes. In fact, we were able to port existing, deployed Meteor applications (e.g., a health survey application, a homework submission application, and a social news application) to RAIL with few changes to the application code.

We evaluated RAIL using these applications and several synthetic attacks, based on common vulnerabilities (e.g., code bugs and user mistakes) that result in direct data disclosures or back doors that leak data indirectly. Our results show that RAIL is precise, efficient, and practical: RAIL accurately flags all inappropriate disclosures with few false reports and minimal re-execution; the throughput and storage overhead of RAIL during recording is 22% and 0.5 KB per request, respectively; and porting several web applications to use RAIL's API required fewer than 25 lines of code changes per application.

RAIL cannot identify all data leaks. For example, attacks that copy the database from the server through some external mechanism (e.g., an NSA employee with access to the server) are outside of the scope of RAIL. In general, RAIL does *not* handle attacks by system administrators, or covert channels; RAIL focuses on data disclosed through the web application's normal interface.

The rest of the paper is organized as follows. §2 discusses previous related work. §3 shows how to use RAIL from the perspective of site administrators and application developers. §4 summarizes RAIL's assumptions and requirements. §5 describes the high-level design of RAIL. §6 presents RAIL's uniform interface for managing shared objects. §7 details the replay and handling of non-determinism. §8 describes our prototype implementation of RAIL. §9 evaluates RAIL's effectiveness. §10 discusses our experience of with RAIL. §11 concludes.

2 Related work

RAIL is the first practical system for precisely auditing unauthorized data disclosures. Much of the previous work on auditing has focused on logging *all* accesses to confidential data. For example, Keypad [6] and Pasture [10] use either cryptography or trusted hardware to maintain a centralized audit log of data accesses, while allowing low-overhead access to this data across many distributed devices. While this model is a good one for auditing unauthorized access when a user's device is stolen, it cannot distinguish legitimate from unauthorized accesses if there is a mistake in the access control policy.

Information flow control and taint tracking systems, such as TaintDroid [5] and TightLip [16], try to prevent disclosure of confidential data in the first place. However, we believe such systems cannot be 100% effective, and disclosures will still happen. For example, a system administrator may misconfigure labels, or a user's password may be guessed by an attacker. Unlike these systems, RAIL does not try to prevent any data leaks; rather, it can detect the leak after the fact without a priori knowledge about which data is sensitive. Moreover, although RAIL and TightLip [16] share the common idea of comparing execution outputs, RAIL addresses the unique challenge of reconciling state divergence, which improves auditing accuracy and performance.

Similarly, encryption is often used to prevent data disclosure in the face of a compromised server, such as in the Mylar web framework [12]. However, encryption does not protect against all disclosures, such as when an ad-

```
1      var Users = App.getDBCollection('users');
2      var Homeworks = App.getDBCollection('hws');
3      var Answers = App.getDBCollection('answers');
4      App.publish('pub_ans', function (userid) {
5   -    var uid = userid;
6   +    var uid = App.getSessionUserId();
7        var u = Users.findOne( {_id: uid} );
8        if (u && u.profile.type === 'staff')
9          return Answers.findAll();
10       return Answers.find( {user: uid} );
11     });
12     App.method('submit', function(hw_id, answer) {
13       var uid = App.getSessionUserId();
14       var hw = Homework.findOne( {_id: hw_id} );
15       var ctx = Rail.inputContext(hw_id, uid);
16       if (!uid || !hw || hw.dueDate < ctx.date())
17         throw new Error('Submission failed');
18       Answers.insert( {_id: ctx.random(),
19         hw: hw_id, user: uid, answer: answer} );
20     });
```

Figure 1: Part of the server-side code from a homework submission application as our running example. Invocations of built-in framework APIs are prefixed with `App`; Rail-specific API are prefixed with `Rail`.

ministrator misconfigures a system or when programmers make mistakes in the application logic.

Rail uses many ideas from prior work on record and replay, such as the action history graph and selective replay from Retro [7], and the comparison of normal execution and replay from Rad [14] and Poirot [8]. Rail's key contribution lies in providing an API that programmers can use to minimize divergence during replay. Prior replay systems were focused on restoring integrity, and reducing divergence was not a priority, resulting in heuristic solutions that attempted to match up replay with normal execution but did not offer strong guarantees. For example, Retro matches non-deterministic system calls in sequential order [7], which might not make sense in our homework submission system's code review assignments. Poirot [8] stops replay when it detects the entry point of an attack, and reports only the initial problematic request; Rail identifies leaked data in all future requests that are indirectly affected by the attack, and reports precisely which data items were disclosed.

Brown's undoable mail server [1] proposes structuring server software around a *verb* API to handle replay after state changes. However, Brown's API is not designed to identify data items that may be disclosed.

3 Using Rail

Using Rail with an application involves three main phases. First, the application developer modifies their application's source code to invoke the Rail API. Second, during normal operation, Rail records inputs to the web application, along with other information specified through the Rail API, to a log. Third, when an administrator detects that there was a problem, she can describe the problem to Rail (e.g., supply a patch or fix an access control list,

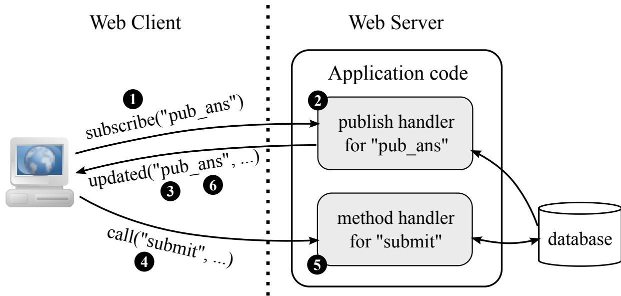

Figure 2: Typical workflow for the running example. 1) The client sends an RPC request to subscribe to the "`pub_ans`" dataset. 2) The corresponding publish handler is executed, which returns a query. 3) The server runs the query and sends the initial dataset to the client via "`updated`" messages. 4) The client calls the RPC method "`submit`" to hand in an answer to a homework. 5) The server runs the method handler, which updates the database. 6) The server reruns published queries affected by the update, and pushes updates to *all* clients that subscribed to it, via several "`updated`" messages.

and pinpoint the time when the problem first arose). Rail will replay requests from the start of the problem, detect which data items may have been inappropriately disclosed as a result, and report them to the administrator.

To understand how this works, consider an example application: a website for submitting homework assignments. Figure 1 shows the server-side code of this application, written in the Meteor framework [11], along with changes that the developer would make to use the Rail API. Figure 2 illustrates a typical workflow for the code. The application defines an RPC method "`submit`" (line 12), which allows students to submit their answers to a homework. The framework does not explicitly send data to the clients, but adopts a publish–subscribe pattern: the server publishes a database query with a name (line 4); when the results of the query might change, the server reruns the query and pushes any updates to all clients that subscribed to it. As we can see from lines 7–10, the publish code returns different queries based on the user's account profile: course staff members are permitted to see all submissions, but students can see only their own.

Suppose the application developer made a mistake checking permissions; as can be seen on lines 5–6, the mistake allows the client to supply the current user ID as an argument to the `pub_ans` subscription, instead of using the `App.getSessionUserId` method, which returns the currently authenticated user ID; such a mistake was discovered in the Telescope social news application [4]. This mistake could have been exploited by an adversary to view all students' submissions, by supplying the user ID of a staff member when subscribing to `pub_ans`.

After running the application with Rail for a while, the site administrator discovers the vulnerability. She wants to know if an adversary exploited the bug, and whose homework submissions were disclosed as a result. To do so, she first applies a patch that fixes the bug (lines 5–6),

```
Leaked data for session RuZw9cCaDMJLdsj8G:
Login: evil_student @ 4/24/2014 3:14:15 PM
   IP: 192.168.0.10
  - answers/fNKXudhNDF7  fields: answer, grade, ...
  - answers/jxT5w7jRJpm  fields: answer, grade, ...
...
```

Figure 3: An example report from Rail indicating that several homework submissions were inappropriately disclosed.

and specifies a time before any possible disclosures (e.g., the time of the first submission). Then she launches the web application again in replay mode. Rail re-executes all subsequent events which might be affected by the patch. Finally, Rail compares the new data items sent to each client with those from the original execution, and generates a disclosure report that details any differences. For example, Figure 3 shows a possible report for this example, indicating that several homework submissions were inappropriately disclosed to a client at a particular IP address.

In order to precisely identify disclosed data, Rail requires application developers to use Rail APIs to name and access shared objects and to annotate non-deterministic inputs in their code. These names, known as *context identifiers*, help Rail match up semantically equivalent operations between the original execution and re-execution. For example, on line 15 of Figure 1, the code creates an input context with an identifier composed of the homework ID and user ID, and uses the context to generate dates and random numbers (lines 16 and 18). As long as the identifier remains unchanged during re-execution, Rail will reproduce the same date and random number from the context. Rail also relies on context identifiers to track dependencies, as we describe in §6.

All non-deterministic inputs and shared objects, including current date and time, random numbers, session variables, database records, and top-level functions, must be accessed via a Rail wrapper to preserve access semantics during replay. In principle, this could be a burden for the programmer, but in our experience, most of the wrapping can be confined to the web framework itself, requiring little additional per-application effort from the developer. In the example application from Figure 1, the developer uses standard APIs from the underlying web framework to retrieve the currently logged-in user (lines 6 and 13), and access the database (lines 7, 9, 10, 14, and 18). Behind the scenes, the web framework itself contains calls to the Rail APIs that wrap these objects, taking care of object naming and dependency tracking.

4 Assumptions

Rail relies on the following assumptions to work properly. First, the developer should correctly use Rail APIs to access shared objects, read non-deterministic inputs, and generate outputs. Developers should also name context identifiers appropriately so that states can be matched up during re-execution.

Second, Rail assumes that the inputs from clients' web browsers remain the same during replay. In general, this might not be true if the user reacts differently to changes in the UI (e.g., some buttons might have changed during replay), but in all of the examples that we have considered, the user's interaction with the application is unchanged. In cases when the administrator knows about client-side changes that must be accounted for, Rail allows the administrator to supply a script to update the client inputs.

Third, Rail assumes that the mistakes leading to disclosures, either administrative or programming, are discovered before Rail's log rolls over.

Fourth, Rail deals only with data leaked through the web application. It cannot detect data revealed through other channels, such as an attacker directly querying the database or accessing the file system. Rail also cannot detect timing attacks, such as an attacker inferring a secret based on how long a response took.

Finally, Rail assumes that the software stack on the server is not compromised, which includes the operating system, system libraries, the web server, framework, and Rail itself. The adversary can, however, take advantage of vulnerabilities in the web application code.

5 System overview

At its core, Rail is a record and replay system. Rail views an application's execution as a stream of *actions*. Each action can read and write *objects*, such as database contents, session state, output, and non-deterministic inputs. This fine-grained view of an application's execution enables Rail to precisely track dependencies between actions and objects. This, in turn, allows Rail to replay a subset of actions when auditing, if it can determine that certain actions were not affected by a mistake. To maintain this dependency information, Rail records dependencies in a *log* during normal operation, and Rail's *replay controller* uses this log to decide what actions to replay for auditing.

Figures 4 and 5 summarize Rail's architecture and API, which we will outline in the rest of this section.

Action APIs. All application code in Rail is executed in the context of some *action*. Actions are the unit of dependency tracking and the unit of replay in Rail. Rail assumes that all application code runs in response to some event, such as an RPC request or a periodic timer event; there are no long-running threads. The web framework maintains a mapping between events and handlers for those events. For example, in Figure 1, the application registers two handlers: one for `pub_ans` subscription events, and one for `submit` RPC events. The handler for each event, stored by the web framework, is actually a named Rail object representing the code for that handler. The ex-

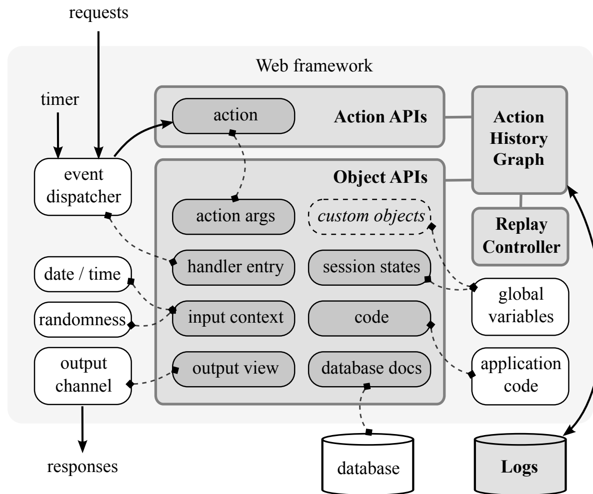

Figure 4: The architecture of RAIL. Strong shading indicates components introduced by RAIL. RAIL's object API constructs shadow objects for most of the shared state, inputs, and outputs in the web framework; these relationships are shown as dashed lines.

isting APIs provided by the web framework create these named object wrappers on the application's behalf; for example, both `App.publish` and `App.method` create such wrappers in Figure 1.

When the web framework receives an event, it retrieves the appropriate handler from its own tables, creates a new action to represent the execution of this event's handler, and invokes the handler in the context of this action, with the event as an argument. The last two steps are performed using the `doAction` API, as shown in Figure 5. In the example in Figure 1, the publish handler (lines 4–11) will run in a new action in response to each subscription request, and the `submit` method handler (lines 12–20) will run in a new action for every `submit` RPC request.

Each action has a *timestamp*, the time at which the action is triggered. Since the handler is a wrapper object, as described above, when the web framework invokes the handler, the handler first records a dependency from the handler object's name to the current action, and then runs the code wrapped by the object. This helps RAIL determine which actions need to re-execute when some code object changes.

Object APIs. Every shared object in RAIL, such as a database record, a function (code) object, or a session variable, is identified by a globally unique name. For each shared object, RAIL maintains two things: first, a set of dependencies between actions and objects, used to track down the set of actions that accessed an object during recording, and second, multiple versions of the object's state at different points in time, used during replay to implement rollback and to check for equivalence.

RAIL assumes that all application code uses object accessors to read and write shared objects, so that RAIL can track the input and output dependencies of actions, and can checkpoint the state of an object at different times. RAIL wraps existing framework objects using accessors, so that in most cases there is no need for the application developer to change the application code. For instance, on lines 6 and 13 of Figure 1, the application code uses the web framework's interface to access the user ID for the current session, which is session-level shared state.

RAIL provides an API for naming and accessing shared objects, which is used both by application code and by web framework code. The `findObject()` function returns a shared object given its unique name. Applications can perform two kinds of operations on a shared object: they can either read it, using `getValue()`, which registers a dependency *from* the object to the current action, or they can modify it, using an object-specific mutator, which registers a dependency *to* the object from the current action, and also records a checkpoint of the object's value.

This object API is used to handle dependencies for different kinds of objects, as we describe in more detail in §6.1. Some RAIL shared objects actually hold the state represented by the object. For example, this is the case for session state objects (accessed by the `getSessionUserId` method in Figure 1). In such situations, RAIL takes care of checkpointing, rollback, etc. In other cases, the RAIL shared object is just a placeholder, and the actual value is stored elsewhere. For example, this is the case for objects representing database state (where checkpointing and rollback takes place in the database, as opposed to in RAIL's log). This is also the case for code objects, since it is difficult to store a JavaScript closure in a log and restore it later on. Each object type defines its own mutators, using the `defineMutator` function; we will discuss these in more detail in §6.2.

Logs and dependency graph. RAIL's dependency graph is an *action history graph* [7] that connects action and object nodes. An edge from object o to action a means o is a's input ($o \rightarrow a$, or a reads o). Conversely, an edge from a to o means o is a's output ($a \rightarrow o$, or a writes o).

Since an object's state can change over time, dependencies in the action history graph refer to an object at a particular time. More precisely, $o \rightarrow a$ indicates that a depends on o's state right before time t_a, where t_a is a's timestamp. Here, RAIL assumes that actions are atomic, since all dependencies to and from an action effectively take place at a single instant in time. This is a reasonable assumption if the web framework provides serializability, which is true in the Meteor framework that our RAIL prototype is built on.

During an action's execution, RAIL connects edges from and to the current action's node as the action accesses objects. When the action completes, RAIL appends an entry to its persistent log, which contains the action's timestamp, its arguments (from the event), and the names

Return type	API	Description
	Public APIs for web framework and application developers	
–	doAction(*args*, *func*)	Allocate a new action and run *func* within its context.
action	getCurrentAction()	Return the current running action.
object	findObject(*id*)	Create or return the RAIL object identified by *id*.
<any>	*object*.getValue()	Accessor. Return the object's current state and update dependency.
function	defineMutator(*func*)	Return a mutator function based on *func*, which alters the binding object's state and updates dependency.
–	registerObjectType(*type*, *proto*)	Register a custom object *type* using template object *proto*.
function	registerCode(*id*, *func*)	Shortcut for creating a code object with the given *id*; returns a wrapper function that takes care of dependency tracking.
object	inputContext(*args*, ...)	Shortcut for creating an input context object identified by *args*.
	Private APIs for the replay controller	
number	*action*.timestamp	t_a, the timestamp when *action* starts. Also used to identify the action.
list<object>	*action*.reads	List of objects that the *action* depends on (inputs).
list<object>	*action*.writes	List of objects that depend on the *action* (outputs).
object	*action*.args	Return the argument object associated with the *action*.
–	replayAction(*action*)	Re-execute the given *action* based on its *args*.
string	*object*.type	The type name of the *object*.
number	*object*.time	The timestamp of the *object*'s current state during replay.
list<action>	*object*.actions	List of actions that read or write the *object*.
boolean	equiv(*object*, *ts*)	Check if the *object*'s current state is semantically equivalent to its state at time *ts* during original execution.
–	rollback(*object*, *ts*)	Revert *object*'s current state to its state at time *ts* during original run.

Figure 5: List of RAIL APIs.

of its input and output objects. The action history graph can be reconstructed from the log during replay. RAIL also logs every object mutation, so that during replay it can reconstruct the object's state at any instant. Objects that do not store actual state in the RAIL's shared object must maintain their own versioning outside of RAIL's log.

Replay controller. During auditing, the site's administrator initiates replay through the *replay controller*, by supplying either a code patch (e.g., fixing a software vulnerability), or a short JavaScript program that fixes the state of the system (e.g., correcting a mistake in an access control list). The administrator can also manipulate the action history graph and the versioned database state through JavaScript APIs, if necessary. The replay controller, in turn, reconstructs the action history graph from the log, and replays the relevant actions that were affected by the administrator's change, as we describe in §7.1. The replay controller computes the *view* of each session during replay, which represents the set of data objects sent to that client, and compares the views during replay with those during the original execution. Data objects that no longer show up in the view during replay are reported as inappropriate data disclosures.

6 Shared objects

To simplify dependency tracking and replay, RAIL defines a uniform API for managing different shared objects.

Object type	Naming convention
Action argument	args/<action id>
Code	code/<identifier>
Handler table entry	handler/<table>/<key>
Database document	db/<collection>/<doc id>
Session user ID	userid/<session id>
Session subscriptions	subs/<session id>
Session output view	view/<session id>
Input context	input/<action id>/<context id>

Figure 6: List of built-in object types.

6.1 Object types

RAIL identifies objects by globally unique names in the form of "object_type/path_name". There are several predefined object types in RAIL, as shown in Figure 6. These types represent most of the abstractions exposed by the web framework. When needed, the developer can also define their own object types using the registerObjectType API. In the rest of this subsection, we will describe what each object type represents.

Action argument. Every action depends on an argument object associated with it. If the action is triggered by a client request, for example, the argument contains the request message. Argument objects are immutable during replay, but the administrator can alter them before replay so as to force certain actions to be re-executed. For example, to cancel a request that creates a malicious account,

the administrator can change the corresponding action argument object to a null request.

Code object. RAIL must be able to determine which actions executed a given piece of code, so that if the code turns out to be buggy, RAIL can replay just the actions that may have been affected by that bug. To do this, RAIL uses a *code object* for every piece of application code, and records a dependency between an action and the code object when the action invokes the code.

RAIL creates code objects at function granularity, because it is easy to interpose on function invocation through a wrapper. The wrapper, created by the `registerCode` API function, records a dependency on the unique identifier of the function's code object, and then executes the function. This ensures that even if an action invokes many functions, the action history graph will contain dependencies to all functions invoked by that action.

RAIL automatically wraps global functions, and names the corresponding objects `code/filename/funcname`. For anonymous functions supplied as callbacks, the developer must assign a name to the anonymous callback in the function that accepts the callback argument. For example, in Figure 1, the `App.publish` function assigns the name `code/publish/pub_ans` to its anonymous callback, and the `App.method` function assigns the name `code/method/submit` to its anonymous callback.

During replay, RAIL's replay controller checks if any of the code objects have changed by comparing the textual representation of the new code object to the original textual representation of the code object as recorded in the log. If any of the code objects have been modified, the replay controller marks all of the actions that executed that code for replay. The textual representation of a function is insufficient to compare closures—e.g., references to variables in outer scopes are not well-defined in the textual representation. However, this is not a problem in JavaScript, because the only way to create an outer scope is to define another function, and if the outer scope of a function changes, the textual representation of that outer scope's function will be different, and will be flagged for replay by RAIL.

Handler table. In addition to tracking dependencies on functions, RAIL also needs to keep track of dependencies on the handler for a given type of event. For example, in Figure 1, the application developer may register a different, non-anonymous function as the handler for the `submit` RPC method. In this case, if the handler for the `submit` RPC method changes during replay, RAIL must detect this and replay all subsequent `submit` RPC invocations. To do this, the RAIL web framework creates a *handler table* object for every kind of handler registered in the web framework. For example, in Figure 1, `App.publish` records a dependency to the `handler/publish/pub_ans` object, and `App.method` records a dependency to the `handler/method/submit` object. The handler table object's value contains the function that will be invoked for that event (which, in practice, is likely to be a code object wrapper).

Database documents. RAIL assumes that the web application uses a key-value store as its persistent storage. Each data item, namely a *document*, has a unique identifier and other mutable fields. However, RAIL's approach is general enough so that it is also applicable to other storage models, including SQL databases and file systems. In particular, every database document is represented by an object named `db/collection/docid`. For efficiency, the RAIL web framework does not store the actual data in the RAIL database object; instead, the RAIL object is a placeholder for dependency tracking, and the actual data is stored, versioned, and rolled back in the database.

Output channel view. RAIL models a view of each session (i.e., the set of data items sent to that client) as a separate view object. View objects accumulate all data items disclosed through the corresponding output channel. By adding a data object, such as a database document, to the view, the application or framework code records that it sent the current state of the object through the output channel associated with the view. The RAIL web framework implements two types of view objects: a *session view* object, which represents all documents sent to a web browser, and an *email view* object, which represents all documents sent to a particular email address. Application developers can define new view objects for other types of output channels.

Other shared state. Accesses to in-memory global state, either application-level or session-level, should also go through the object API. Currently, RAIL defines two types of session state objects: *current user ID objects* and *subscription objects*. The current user ID object stores the logged-in user's identifier for each session. Subscription objects are necessary for interactive web applications that adopt a publish–subscribe pattern. They store a list of database queries that a session is interested in, so that whenever the results of any of these queries change, the web framework can notify the client about the updates.

Input context. Input context objects handle non-deterministic inputs requested by an action, such as current date and random numbers. They are important for stable re-execution, as we will discuss in §7.2.

6.2 Accessors and mutators

Every object has an accessor and a few mutators. Mutators vary with object types. For example, session user ID (`userid`) objects have two mutating methods:

`login(userid)` assigns the given user ID to the object's current state, and `logout()` resets its current state to `null`.

During normal execution, the accessor connects the object to the current action in the action history graph, and returns the current state of the object. Similarly, mutators connect the current action to the object, and change the current state of the object accordingly. In addition, mutators also log the mutating operation, so that by replaying the log during re-execution, RAIL can reconstruct checkpoints for all history states of the object.

During re-execution, accessors and mutators behave differently than during normal execution. If an object has been rolled back, the accessor returns the object's latest state; otherwise it returns the checkpoint state right before the current action. Mutators do not log changes during re-execution, but roll back the object before updating the object (see TRYROLLBACK in Figure 7). Since two executions are not identical, replay can introduce new dependencies that did not show up in the original execution. RAIL must keep updating the action history graph during replay to capture the new dependencies.

For performance reasons, accessors and mutators for database document objects are handled differently. RAIL employs a *time-travel database* [2] to keep every version that ever existed for each document in the database. Different versions of the same document are distinguished by two additional fields, `start_ts` and `end_ts`, which indicate the time interval within which the version is valid. Application code uses the web framework's database API to access the database as before. RAIL interposes on query processing and cursor accesses such that only the desired version is returned or updated. RAIL also performs dependency bookkeeping for the corresponding placeholder object of each affected document.

7 Replay

In order to determine what data was inappropriately disclosed, RAIL must re-compute the view objects for every session, and if it detects any session whose new view object is not a superset of the old view object, it reports the difference as a leak. Note that new data disclosed during replay does not result in a report.

RAIL recomputes the view objects by replaying previously recorded events and re-executing the corresponding actions. There are two challenges in doing so. First, for efficiency, RAIL should not re-execute every action; to this end, RAIL implements selective re-execution (§7.1). Second, for precision, RAIL should minimize divergence between replay and the original execution; to do this, RAIL uses context-based matching (§7.2).

7.1 Selective re-execution

Figure 7 shows the pseudo-code for RAIL's selective replay algorithm, inspired by Retro [7]. The algorithm relies

```
 1: procedure INITIALIZEREPLAY
 2:     objects ← LOADLOGS()
 3:     for all o ∈ objects do
 4:           ▷ Admin might change code or argument objects
 5:         if o.type ∈ { "code", "args" } then o.time ← 0
 6:         else  o.time ← ∞
 7:     return objects
 8: procedure NEXTACTION(objs)
 9:     acts ← {a | ∀o ∈ objs (a ∈ o.actions ∧ t_a > o.time)}
10:     if acts = ∅ then return nil
11:     return argmin_a{t_a | a ∈ acts}
12: procedure TRYROLLBACK(o, t)
13:     if o.time > t then
14:         ROLLBACK(o, t)
15:         o.time ← t
16: procedure MOVEFORWARD(o, t)
17:     if EQUIV(o, t + 1) then
18:         ROLLBACK(o, ∞)
19:         o.time ← ∞
20:     else  o.time ← t
21: procedure SELECTIVEREPLAY
22:     objects ← INITIALIZEREPLAY()
23:     a ← NEXTACTION(objects)
24:     while a ≠ nil  do
25:         C_in ← ∃o ∈ a.reads (o.time < ∞ ∧ ¬EQUIV(o, t_a))
26:         C_out ← ∃o ∈ a.writes (o.time < t_a)
27:           ▷ Replay if either inputs or outputs are changed
28:         if C_in ∨ C_out then
29:             for all o ∈ a.writes do TRYROLLBACK(o, t_a)
30:             REPLAYACTION(a)
31:             for all o ∈ a.writes do MOVEFORWARD(o, t_a)
32:         for all o ∈ a.reads do o.time ← max(o.time, t_a)
33:         a ← NEXTACTION(objects)
```

Figure 7: The selective replay algorithm. The algorithm uses private RAIL APIs listed in Figure 5.

on the *time* variable of each object, which indicates the timestamp of an object's current state and controls the progress of the re-execution.

Initially, every object is in its latest state ($time = \infty$), except for code and action argument objects, which the administrator could change to kick off replay.

In each round, RAIL picks the first action (action with the minimal timestamp) from a set of candidate actions to replay. Candidate actions are actions that read or wrote an object, and whose timestamps are bigger than the object's current *time*, meaning that they happen after the object's current state. Then, RAIL checks if the picked action needs re-execution. If any of its inputs have changed (C_{in} is true), RAIL must rerun it to generate new outputs; similarly, if any output has been rolled back to a state before the action took place (C_{out} is true), RAIL must rerun the action to reconstruct the output. Otherwise, RAIL can skip the action, and advance the timestamps for all of its inputs, so that the same action will not be selected again.

To replay an action, RAIL first rolls back all of the action's output objects recorded during the original execu-

tion to the state right before the action. This is important because the replayed action might not update the same objects as original. Then Rail reruns the action and updates the timestamps for the action's output objects. Note that during replay, mutators might roll back other output objects which were not captured in the original execution. As an optimization, if after replaying an action, an object's state is equivalent to its next state in the original execution, Rail will directly roll forward the object to its latest state to avoid considering actions that access the object in the future.

The selective re-execution algorithm is guaranteed to terminate because Rail always chooses the earliest available action. After each iteration, every relevant object will have a timestamp which is no smaller than the chosen action's. Therefore the timestamp of the picked action in each iteration monotonically increases.

Because of Rail's precise dependency tracking, the selective re-execution algorithm can minimize the number of actions replayed to just those that may have been affected by the mistake that triggered the audit. In our experience, selective re-execution replays only a small fraction of the total number of recorded actions.

One concern of selective re-execution is dealing with patches that significantly alter the control flow of an application. Rail works in this scenario because its unit of replay is an individual action (e.g., a client RPC request), and Rail's report is based on the set of objects that end up in a session's view. As long as the original and patched code add the same objects to the view, no disclosures will be reported regardless of code changes. In the case that the new code accesses many different shared objects, such as by issuing new database queries, Rail will replay more actions due to additional dependencies.

7.2 Context matching

Rail's goal is to precisely identify inappropriately disclosed data. Since Rail computes the set of disclosed data items as the difference between the original and the replayed view objects, Rail will report a data object as inappropriately disclosed if it fails to show up in the replayed view. This is desirable if the data item fails to show up in the replay view due to a fixed vulnerability. However, this is undesirable if it is a result of non-determinism, and some other choice of non-deterministic inputs could have led to the data *not* being flagged as disclosed.

This problem is made more complicated by the fact that *some* inputs to an action may have changed during replay. To minimize false reports, programmers must ensure that during replay, the behavior of non-deterministic code in the application remains as close as possible to that of the original execution, even in the face of input changes. We refer to this property as *application stability*. To help application writers to achieve this property, Rail

```
1     App.method('populate_admins', function() {
2  -    var admins = ['Alice', 'Mallory', 'Bob'];
3  +    var admins = ['Alice', 'Bob'];
4       for (var i = 0; i < admins.length; ++i) {
5         var pwd = Math.random();
6         /* BETTER: var pwd = Rail.inputContext(
7               'populate', admins[i]).random(); */
8         Users.insert({name: admins[i], passwd: pwd});
9       }
10    });
```

Figure 8: Example code demonstrating the necessity of using context identifiers to retrieve non-deterministic inputs.

provides helpful APIs. In some cases, using these APIs, it is easy for the application writers to achieve application stability, while in other cases it requires some thought. We provide a few examples to illustrate the issues in achieving application stability. Note that if an application does not achieve application stability, Rail will work correctly, but may generate false reports.

For some application functions, it is relatively easy to make them stable. For example, in Figure 1, the `submit` RPC handler checks the current time (to see if the submission is late), and assigns a random ID to the submission. To ensure stability for this code, the programmer creates an *input context* object on line 15, which instructs Rail to reuse that same randomness during replay.

As a more complex case, consider the code fragment and patch shown in Figure 8. The code is intended to populate the database with a few predefined administrator accounts. Suppose that the site administrator later found out that Mallory was not supposed to have administrative privileges, and she wanted to see what information may have been disclosed as a result of this mistake. She does this by running Rail in auditing mode with Mallory removed from the list (see the change on lines 2–3 in Figure 8). Since Mallory was in the middle of the list, a simple heuristic that returns non-deterministic outputs in the same order as they were requested in the original run would reuse Mallory's password (the second invocation of `Math.random()`) for Bob during replay (since it is now the second invocation of `Math.random()`). Thus Bob's recorded login requests will fail during replay, causing many data items to be flagged as leaks. Prior systems such as Retro [7] and Warp [2] use this heuristic.

Rail tackles this problem by asking the programmer to assign stable *context identifiers* to non-deterministic inputs. During replay, Rail supplies non-deterministic values from the same context ID. Moreover, the current action's timestamp is also considered part of any context ID, so that all non-deterministic inputs are local to each action. Non-deterministic values with the same context ID are supposed to be semantically equivalent, therefore the programmer should make sure that the identifiers they choose are semantically stable. As an example, in the comments in lines 6–7 of Figure 8, we use the account

```
function pair_reviews(ids) {
  var seeds = [];
  for (var i = 0; i < ids.length; i++) {
    var ctx = Rail.inputContext('seed', ids[i]);
    seeds[i] = { sid: ids[i], rnd: ctx.random() };
  }
  var shuffle = _.sortBy(seeds, function (e) {
    return e.rnd;
  });
  for (var i = 0; i < ids.length; i++) {
    var reviewer = shuffle[i].sid;
    var reviewee = shuffle[(i+1) % ids.length].sid;
    var ctx = Rail.inputContext('pair', reviewer);
    Pairings.insert({ _id: ctx.random(),
        reviewer: reviewer, reviewee: reviewee });
  }
}
```

Figure 9: Illustration of a stable pairing algorithm.

name as part of the context identifier, which effectively ensures that the same account name will get the same password during replay, even if the list order has changed. In contrast, including the loop iterator `i` in the context ID is a bad idea, because it does not preserve semantics.

In some cases, making a function stable is an even more difficult problem. Consider the problem of pairing up students in a homework submission system for peer review. If one of the students is removed during replay, the set of pairwise assignments produced by most pairing algorithms would be quite different. To solve this problem, the programmer must devise a stable algorithm using context identifiers. Figure 9 illustrates such an algorithm that we designed for the homework submission application. The algorithm works by assigning every student a random pairing order, and then sorting the students by this pairing order. The pairing order is chosen through a context identifier tied to the user's username. This ensures that if students are added or removed, the overall sorted order is largely the same. Students are then paired up with other students next to them in this sort order. As a result, if a student is added or removed, this results in only a small number of changes to the overall pairings.

8 Implementation

We implemented a prototype of Rail on top of the Meteor web framework. Meteor has a clean interface for exchanging data between browser and server, which allows Rail to clearly identify data items. The core of the prototype is a standalone package that implements Rail's action APIs and object APIs.

The core package also maintains the action history graph and on-disk logs in two B-tree-like data structures—one stores actions (indexed by timestamps) and the other stores objects (indexed by object identifiers). Edges between actions and objects are stored twice (in both B-trees). During replay, Rail reconstructs the graph progressively without scanning the entire log. The prototype caches recently used B-tree blocks in memory and writes back dirty blocks in the background.

The prototype changes a few built-in packages in Meteor, so that accesses to standard Meteor abstractions are wrapped using Rail APIs. These abstractions include session user IDs, RPC dispatchers, session subscriptions, and MongoDB documents. Application developers can use standard interfaces to access these objects as before.

The prototype also includes a code rewriter, which automatically names and wraps top-level JavaScript functions using Rail's code objects when the application is loaded.

Our prototype consists of about 3,800 lines of JavaScript code, of which 2,987 lines are in the core package, 422 lines are for Meteor integration, and another 358 lines are for the code rewriter and command line tools.

9 Evaluation

We evaluate the Rail prototype with three real-world applications under synthetic attack workloads. Our evaluation aims to answer the following questions:

- What is the effort to port applications to Rail? (§9.1)
- What attack scenarios can Rail handle? (§9.2)
- How precise are Rail's data disclosure reports? (§9.3)
- What are Rail's performance and storage overheads during recording? (§9.4)
- How do the techniques described in §7 improve Rail's accuracy and performance for auditing? (§9.5)

9.1 Applications and developer effort

We ported three real-world web applications to Rail. Two of them are privacy-sensitive: one is *Submit*, a website that manages homework and grades, written by course staff from our department; the other is *EndoApp*, a medical survey application. Both applications run in production and have dozens to hundreds of users. We also ported *Telescope*, a widely used open-source social news application, to see how well Rail can support a full-fledged application with a relatively large code base.

Figure 10 summarizes the effort for porting these applications to Rail APIs. We had to modify fewer than 25 lines of code for each application. Most of the changes are related to non-deterministic inputs: programmers must provide context identifiers when generating date and random numbers in the application.

For Submit, modifications of two staff-only method handlers were necessary to ensure auditing correctness. First, the `getGrades` method summarizes grades of each assignment for all students, and returns a grid of grades directly to the client (not using the standard publish–subscribe mechanism). We rewrote the code using Rail's object API to explicitly add revealed data items to the current session's view object. Second, we modified the `pairing` function, as described in §7.2.

Appliation	Description	LoC (in JavaScript)		
		changed	server	client
Submit	homework grading	24	769	891
EndoApp	medical survey	2	599	900
Telescope	social news	20	1,169	1,781

Figure 10: Real-world web applications used in our evaluation, and the developer effort to port them to RAIL APIs. We do not count HTML/CSS and third-party library code. Only server-side code is modified.

9.2 Attack case study

To evaluate whether RAIL can identify disclosures after an attack, we chose the following common mistakes that can lead to data breaches in real-world settings.

Access control list error. In Submit, a course staff member erroneously grants "staff" privileges when creating a student account. The student logs in with this account and sees other students' homework solutions and grades. The staff later realizes the mistake, rectifies the initial request that created the account, and wants to know what unintended information has been revealed to the student. RAIL identifies the leaks because during re-execution, the student's subscription request will be rejected by the server given the correct user privilege.

Stolen password. In EndoApp, a careless surgeon chooses a weak password, which is obtained by an outside attacker. Managing to stay concealed, the attacker creates another surgeon account, and logs in to the new account several times to retrieve sensitive patient profiles. After the administrator discovers the breach, presumably by looking for logins from unintended IP addresses, she cancels the suspicious login request to the careless surgeon's account, and wants to know what has been disclosed as a result of the suspicious login. RAIL reports all breaches from both accounts, because without the initial login, all subscriptions and the account creation request would fail. All subsequent logins to the new account would also be denied since that bad account no longer exists.

Code bugs. This attack is based on a real bug in Telescope's commit history [4], in which the application performs permission checks according to a client-supplied current user ID, and publishes sensitive user emails for all accounts based on the flawed security check. An attacker can exploit the bug by executing JavaScript code with a chosen user ID from the browser console. After patching the code, the administrator wants to know if anyone exploited the bug, and whose emails were leaked. RAIL detects the code change and reruns all subscription requests that depend on the code. The malicious request will be rejected during re-execution, while the legitimate ones (where the supplied user ID is the same as the session user ID) will return the same result as before. Therefore RAIL can precisely identify leaked data from sessions that truly exploited the bug.

9.3 Auditing precision

To see if RAIL can precisely report leaked data, we run the three attack workloads in parallel with other benign workloads in the background as noise. For each attack workload we consider two traces: a short trace in which background workloads stop soon after the attack, and a longer trace where benign accesses continue for several minutes. We compare the total number of data items accessed during the trace with the number reported by RAIL. We manually inspect the report to count false reports (legitimate disclosures flagged by RAIL as inappropriate) and missed ones (inappropriate disclosures according to our knowledge of the workload *not* reported by RAIL). The result is shown in the last group of columns in Figure 11.

The number in the "accessed" column simulates what an access log based system, like Keypad, would report. As we can see, RAIL precisely differentiates disclosures to the attacker from disclosures to legitimate users, resulting in fewer reports. RAIL's report is stable: the duration of the trace and when the attack begins in the trace do not lead to more false reports.

After changing applications to use the RAIL APIs, as described in §9.1, we do not observe any missed disclosures in our experiment. There is a single false report, however, for EndoApp workload. The reported database item is the malicious surgeon account added by the attacker, which *is* an expected disclosure for other legitimate surgeons logged in after the attack, because surgeons automatically subscribe to all user accounts upon login. But RAIL flags it because the malicious surgeon does not appear in the re-execution anymore. We believe this false report is acceptable since it is related to the attack, and also helps the system administrator to identify the vulnerability.

9.4 Performance and overhead

We measure the performance of RAIL using two machines running recent versions of Debian Linux. The server has an Intel Core i7 3.3 GHz processor and 24 GB of RAM; the client has eight 10-core Intel Xeon E7-8870 2.4 GHz processors with 256 GB of RAM. The client and the server are connected via a 1 Gbps network. To get a stable result, we pin the web server process to a single core of the server machine. The client machine is significantly more powerful to allow us to run enough browser instances to saturate the server. We use Splinter to drive PhantomJS browsers for all experiments.

Performance during normal execution. We compare the performance of RAIL during normal execution to the performance of an unchanged version of Meteor, using Submit as the benchmark. In the "browse" workload, each

Attack workload	Trace	Number of requests			Running time (seconds)				Number of data items			
		total	attacker	replayed	original	replay	exec.	other	accessed	reported	false	missed
ACL error	Submit.short	2,972	45	16	161.0	1.6	0.9	0.7	1,142	193	0	0
	Submit.long	12,366	45	16	664.0	3.2	2.0	1.2	1,121	193	0	0
Stolen password	EndoApp.short	2,967	25	42	149.0	0.9	0.6	0.3	1,871	137	1	0
	EndoApp.long	8,597	25	270	640.0	10.0	3.1	6.9	3,521	197	1	0
Code bugs	Telescope.short	1,426	14	20	113.0	1.2	0.9	0.3	23	10	0	0
	Telescope.long	7,763	14	833	603.0	61.2	25.9	35.3	23	10	0	0

Figure 11: Replay performance and auditing precision under various workloads. "attacker" counts requests from the attacker's session. "original" is the duration for recording the trace. Replay time is broken down into two parts: "exec." shows the time for re-executing actions; "other" shows the time spent in other parts of the replay loop (all except line 30 of Figure 7). "reported" counts distinct data items flagged by RAIL, out of all data items accessed by the trace.

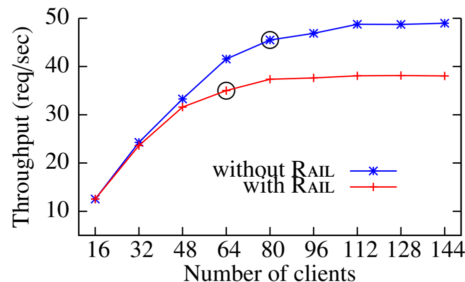

Figure 12: Server throughput when running Submit with the "browse" workload with an increasing number of concurrent clients. Circles mark the points where the server's average CPU usages exceeds 90%.

client repeatedly logs in using a random student account, browses the account's grades, and then logs out.

With a single client, the average latency for handling an individual request increases by 34% (from 15.0 to 20.1 msec). Profiling shows that executing wrappers, updating logs, and handling time-travel database queries contribute to the majority of the overhead.

To see how RAIL performs under heavy workloads, we stress the server with an increasing number of clients, which send RPC requests as fast as possible. We measure the server's throughput and average CPU usage. As shown in Figure 12, the stock Meteor saturates at about 80 concurrent clients, while RAIL saturates at 64 clients. For an under-loaded server (under 48 clients), RAIL incurs less than 5% throughput overhead; for an over-loaded server (112 clients), the overhead is about 22%.

We also measure the throughput overhead for workloads with different write ratios. In the "upload" workload, there is a 20% probability that the user will submit a new answer to a homework after logging in (based on our historical logs), which leads to more write requests. Figure 13 shows the result: increasing the write ratio has a small impact for the overall performance, with the overhead going up from 16.9% to 17.9%.

Storage overhead. Figure 13 also shows the storage overhead. RAIL's storage overhead consists of two parts: the compressed log, which contains the action history graph and the objects' mutation history, and the time-traveling database, which preserves all history versions of database records. The average overhead is 0.46 KB per request for the "browse" workload, and 0.49 KB per request for the "upload" workload. Note that login and logout also write to the database, updating login timestamps and tokens; this is why the numbers for the two workloads are close. With this overhead, a 500 GB disk can store 1 year worth of logs even for a fully utilized server. The time span is sufficient for most disclosure auditing tasks.

Workload	Throughput (reqs/sec)			Storage (KB/req)		
	w/o RAIL	RAIL	overhead	logs	DB	total
Browse	45.14	37.52	16.9%	0.34	0.12	0.46
Upload	45.50	37.36	17.9%	0.35	0.14	0.49

Figure 13: Performance and storage overhead during normal execution. Numbers are from a fully utilized server running Submit and serving 80 concurrent clients, which send requests as fast as possible. Two workloads with different write ratios are shown.

Replay performance. We measure RAIL's replay performance using the traces shown in Figure 11. We consider two metrics: the number of replayed requests versus total number of requests, and the time to finish the replay, as shown in the first two groups of columns in Figure 11.

In Submit and EndoApp, RAIL replays only a small fraction of all requests—just those related to the attack. For Submit, the number is even smaller than the total number of requests from the attacker's session, indicating that RAIL's selective replay algorithm can effectively pick out just the relevant actions. In the Telescope workload, because the patched code is in the publish handler, RAIL has to rerun subscription requests from all sessions, which causes each session's view object to be rolled back, and in turn triggers more replays. All of the re-executions are necessary to ensure that RAIL captures all undesired leaks.

The "original" column shows the time to record each trace, which represents a typically loaded web server incurring about 20%–30% of CPU overhead. As we can see, RAIL can replay lengthy traces in a small amount of time, and can report data leaks with high precision.

To understand what the performance bottleneck is during replay, we break down the replay time into two parts:

Technique	# of requests		# of leaked data	
	total	replayed	reported	false
Rail	103	5	2	0
w/o selective replay	103	93	2	0
w/o context matching	103	13	16	14

Figure 14: Impact of disabling each of Rail's techniques.

the time to re-execute actions (column "exec." in Figure 11), and the time spent in other parts of the replay loop ("other" in Figure 11), which comprises the overheads of the replay algorithm (including selecting actions, checking object equivalence, and updating object states). Both times increase as the number of replayed requests increases. For the "EndoApp.long" and "Telescope.long" workloads, the "other" time is higher than the "exec" time of re-executing actions. The "other" time, however, is time well spent: it is the overhead paid for avoiding re-execution of irrelevant actions.

9.5 Technique effectiveness

To demonstrate the value of selective re-execution and context identifiers, we use Submit with a setup of 30 students and one staff account. The staff member first creates a new account for a malicious student; then she initiates pairing, assigning each student (including the malicious one) two random reviewees using an algorithm similar to Figure 9. After five students log in and browse the reviews for their code, the staff runs Rail in replay mode after canceling the creation of the malicious account.

Figure 14 shows the result. Rail flags exactly two data items: one is the malicious user and the other is the pairing record for the user. Out of 103 requests, Rail replayed only five, which includes the pairing request and four requests from students paired with the malicious account.

Without selective re-execution, Rail reruns 93 requests in total, including all requests that follow the account creation. Disabling context matching introduces 14 false reports: as pairings change, most students see homework answers from different peers during replay; the IDs of pairing records will also be different, constituting the rest of the false reports. This demonstrates the importance of Rail's selective re-execution and context matching.

10 Discussion

This section discusses our experience with Rail.

10.1 Supporting Rail in other frameworks

Our Rail prototype demonstrates that by utilizing the rich semantics available in the web framework, one can achieve fine-grained information tracking at low cost. Although our prototype is based on a specific framework (Meteor), the design of Rail's core API is framework-independent. Rail's techniques can be applied to other web frameworks as long as they meet a few assumptions.

First, the framework should force developers to use the framework's abstractions and APIs to access web objects, such as requests, responses, sessions, databases, and files. Bypassing these interfaces should be considered rare or prohibited entirely. This helps Rail interpose on accesses to these objects at a level where useful semantics are preserved. Adopting Rail to another framework involves wrapping the framework's object APIs with Rail APIs.

Second, the framework should provide a mechanism that separates data items from their web representation. Meteor attains the separation by sending data items directly over the wire, and constructing web pages purely on the client side. Other commonly used frameworks, such as Ruby and Django, do not share this paradigm. However, they do adopt the model-view-controller (MVC) pattern using server-side template rendering systems that clearly separate data and views. The major difference in porting Rail to these frameworks lies in how to track responses: one could wrap the template rendering system (as opposed to the publish system in Meteor) with Rail's output view object API to capture revealed data.

Third, the framework should maintain as little global state as possible. To correctly support selective re-execution, Rail must interpose on accesses to all global objects in order to track dependencies and make continuous checkpoints. Excessive use of global state can introduce false dependencies among requests and increase space overhead. Fortunately, most web frameworks do not maintain global state other than the persistent storage (e.g., the database) and a simple session store in their core packages. External packages, however, might keep their own shared state. As an example, Meteor's account package does not reuse Meteor's session store, but keeps per-session authentication tokens on its own. When porting Rail to a new framework, one must also examine external packages to ensure that all package-defined global objects are properly wrapped.

Finally, Rail's current design has a simplified API that assumes action serializability. We believe this captures an important class of real-world web applications: for instance, Node.js applications fall into this sequential execution model. Nevertheless, Rail's API could be extended to support concurrent action execution. This would require finer-grained dependency tracking and replay at a lower level. For instance, one could treat each access to a shared object (e.g., database query) as atomic, and record dependencies between such operations. During replay, each action might be interleaved with the replay of other actions. This is similar to how multi-threaded record-replay systems work, and to how Retro [7] dealt with record-replay of concurrent processes.

10.2 Porting applications

Porting an application to RAIL is easy, because the framework wrappers do most of the work, such as wrapping and trapping accesses to global objects. In rare cases, if an application defines its own class of global objects, the programmer must wrap accesses to these objects using the RAIL API.

For most applications, no matter how large the code size is, the majority of changes will be for handling non-deterministic input. Since application stability must exploit high-level knowledge unavailable in the code, it cannot be implemented without help from developers. For example, no one knows better than the developer what the context identifier should be for a non-deterministic value. Identifying non-determinism in the code could be a potential challenge when porting applications.

Fortunately, there are only a handful of sources of non-deterministic input that have to be handled—for most cases they are date, time, and random numbers. These values usually come from the language's library calls, such as `now()` and `random()`. Simply hiding these library interfaces from developers could help them identify sources of non-determinism and force them to use RAIL's wrappers.

Simple program analysis can also help identify these sources of non-determinism, and can be used to suggest context identifiers, as we will discuss next.

10.3 Choosing context identifiers

The goal of context IDs is to preserve application stability. As a general guideline, the context ID usually contains the primary key of the data item tied to the non-deterministic value, plus an optional string describing the purpose of the value. In this subsection, we illustrate this rule with examples we encountered in benchmark applications.

The most common use of context IDs is to generate random identifiers for new data items. For example, when generating a document ID for a new homework submission in Figure 1, the context ID should be the pair (*homework_ID*, *student_ID*), which uniquely identifies a homework submission. Similarly, when adding a comment in Telescope, the context ID should contain the topic ID and the user ID. If multiple random values are requested using the same primary key within a single action, one can add descriptive strings to distinguish different invocations, like on lines 4 and 13 of Figure 8. In practice, this process could be automated by a simple analysis of the database schema.

Another common use is to generate dates and timestamps. For instance, when a student adds a homework submission, the application needs to check the current date against the homework's deadline. Since the current date is not tied to any data item, we simply supply a constant string "`checkdeadline`" as the context identifier. The descriptive string helps distinguish this date query from others in the same action, if any. Often, the calling function's name and signature can be used as the descriptive string when requesting the current date.

Context IDs also play an important role in preserving cryptographic randomness. For example, Meteor's account package uses the SRP protocol to authenticate user logins. Internally, SRP generates random values, and if they are not preserved, login will not replay correctly. In this case, we use the encrypted password as the context ID when generating the random SRP verifier, so that the same login password yields the same verifier during replay.

10.4 Misuse of RAIL APIs

Inadvertent misuses of RAIL APIs might affect RAIL's accuracy. RAIL requires the developer to use the framework's standard interface to access global objects, such as the database. If the developer forgets to wrap application-defined global state with RAIL APIs, RAIL will miss the dependency, omitting relevant actions from selective replay. This will leave the replayed application in an inconsistent state, and likely lead to false negatives.

If the developer does not use RAIL's wrappers to retrieve non-deterministic values, or inappropriately chooses context IDs, RAIL will produce a different value during replay. Consequently, requests that depend on these non-deterministic inputs (such as logins) will behave differently. The divergence will cause more actions to be replayed, and introduce false positives and false negatives.

Note that the worst outcome of API misuse is unnecessary re-execution and inaccurate reports. The entire replay process is guaranteed to terminate.

11 Conclusion

RAIL is the first system for precisely auditing past data disclosures in web applications. Based on rollback and replay, RAIL introduces an explicit API that application developers must use to identify data items, track dependencies, and match up states. The API helps RAIL minimize state divergence and unnecessary re-execution, providing fast and precise auditing. Measurements with a RAIL prototype show that RAIL can precisely distinguish legitimate data disclosures from illegal ones caused by human mistakes. RAIL requires only minor changes to web applications and incurs a moderate performance overhead. RAIL's source code is publicly available at https://github.com/haogang/rail.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Landon Cox, for their feedback. This research was partially supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143.

References

[1] A. B. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 1–14, San Antonio, TX, June 2003.

[2] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 101–114, Cascais, Portugal, Oct. 2011.

[3] R. Chandra, T. Kim, and N. Zeldovich. Asynchronous intrusion recovery for interconnected web services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 213–227, Farmington, PA, Nov. 2013.

[4] M. DeBergalis. Use this.userId() in publish rather than an arg, Sept. 2012. https://github.com/TelescopeJS/Telescope/pull/6.

[5] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[6] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An auditing file system for theft-prone devices. In *Proceedings of the ACM EuroSys Conference*, pages 1–16, Salzburg, Austria, Apr. 2011.

[7] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 89–104, Vancouver, Canada, Oct. 2010.

[8] T. Kim, R. Chandra, and N. Zeldovich. Efficient patch-based auditing for web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 193–206, Hollywood, CA, Oct. 2012.

[9] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 23(1):51–76, Feb. 2005.

[10] R. Kotla, T. Rodeheffer, I. Roy, P. Stuedi, and B. Wester. Pasture: Secure offline data access using commodity trusted hardware. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[11] Meteor Development Group. Meteor: A better way to build apps. https://www.meteor.com/.

[12] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan. Building web applications on top of encrypted data using Mylar. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 157–172, Seattle, WA, Apr. 2014.

[13] M. Rile, B. Elgin, D. Lawrence, and C. Matlack. Missed alarms and 40 million stolen credit card numbers: How Target blew it. Bloomberg Businessweek, Mar. 2013. http://www.businessweek.com/articles/2014-03-13/target-missed-alarms-in-epic-hack-of-credit-card-data.

[14] X. Wang, N. Zeldovich, and M. F. Kaashoek. Retroactive auditing. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, Shanghai, China, July 2011.

[15] A. G. Wylie. University of Maryland: Data breach, Mar. 2014. http://www.umd.edu/datasecurity/.

[16] A. R. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping applications from spilling the beans. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, Apr. 2007.