TaTAMi

Taint Tracking for Application Migration

by

Lee Alexander Beckman

B.Sc., The University of British Columbia, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

October 2012

Abstract

In this thesis we addressed the question of whether taint tracking could be used to help developers make better application migration decisions. We wanted to determine if detailed dynamic dataflow traces from web applications could support automated analyses, the results of which would help one to better understand applications and guide one in optimizing them.

To investigate this problem, we had to build our own taint tracker for Java as none existed which would perform an extensive enough tracing. We then identified a set of useful analyses from a search of the literature and from our own experience with web applications. These analyses were developed to run automatically over taint tracking data, producing output which should be immediately useful to non-expert users.

Two real applications were chosen and the analyses applied to them in other to determine two important things. First, that we could write our analyses to automatically identify their targets and produce comprehensible results. Second, that the analysis targets actually existed in real applications.

In the end our analyses were mostly successful, in many cases producing clean results which consisely described non-trivial properties of the applications and possible optmizations to them. By focusing on how data moves through a system, we found a natural fit for understanding its workings. The biggest difficulties manifested as a need for further automation, to take complicated analysis results and simplify them. Even with many challenges, we believe that our techniques are valuable for helping developers, and should be more thoroughly studied.

Table of Contents

List of Tables

List of Figures

List of Programs

Acknowledgements

Thank Rodger, Eric, and Nima

Dedication

Dedicate this thesis to my wife.

Chapter 1

Introduction

Maintaining any sufficiently large application is a difficult task for programmers, especially those who did not write it. In this thesis we focus on web applications, and are concerned with the times in a system's life which require it to change. This is a common event, as has been shown in the history of any popular application which has been around for long enough.

The problem is straightforward; a team of developers will implement a system to meet the needs of their users at the current point in time. As we experiment with it later, let us say this system is the RUBiS online auctioning site (think a very simple version of eBay). Eventually, as RUBiS becomes more popular and users want more features (shopping carts, and they're complaining that pages take too long to load), the demand on the application increases. Eventually the original design is insufficient to serve the needs of its users, and some form of migration must happen. New resources are made available for RUBiS to use, such as more servers, but how best to use them? By this time, new developers are on the project, but those who originally designed the system are mostly unavailable to provide support. For these non-experts, making wise decisions about how keep RUBiS up to date is a frustrating and time-consuming process, wrought with many problems:

- Time is often short
- Changing application code without being aware of all of its side effects can break it in unexpected ways
- Real applications are often written poorly, and may not conform to expected patterns
- The path of data from where it is stored until it gets to users can be long, complex, and very difficult to piece together manually. Knowing these paths are critical to understanding web applications.

It is here that automation is useful. We want to help these developers by building tools which assist them in understanding their systems more quickly. We even want to automatically make suggestions about how to actually perform optimizations. Tools of this nature have naturally already been researched and developed, and ours represents further exploration into this area. The model of such systems is as follows:

- Automatically collect data about how applications works. This can be done statically, looking at the code, or dynamically, monitoring the runtime operation of the program.
- Use this data to build a representation of the application that can be analyzed. This is often a graph model where the application is thought of in terms of components (nodes) which are connected if the data shows a meaningful interaction between them (one component calls another, they exchange data, etc).
- Run algorithms over the representation (rather than the application itself), to discover properties of it or find ways to optimize it.
- Map these properties and optimizations back to the original applications, helping developers to better understand and improve them.

1.1 Origins of Research

The TaTAMi project is a system which uses dynamic information flow tracking (DIFT) support a variety of analyses for Java web applications. DIFT is a technique where data is tainted (or tagged) so that its progress can be tracked throughout a system, allowing one to see what parts of the system certain data interacts with and how new data is derived from existing data. The method has been used almost exclusively in the security domain. By tagging untrusted data such as user input, DIFT systems can raise alerts if that data ever propagates to a location it should not, like a config file. We realized that for many of the analyses that we wanted to perform, knowledge of the flow of an application's data throughout its components was key. We anticipated that a heavy-weight variant of taint tracking, dataflow tomography, could extract data which would support many analysis cases.

The analyses in question were inspired by works focusing on various forms of dataflow optimization. These include application partitioning, which seeks to split programs into functional pieces and optimally distribute them, and works such as the Fluxo system [18]. Fluxo provided a series of automatic analyses which could be performed over dataflow graph representations of programs, including discovering caches, finding opportunities to delay computation to lower latency, and differentiating between stateful and stateless components. Many of the analyses developed for TaTAMi were derived from those presented in the Fluxo paper, though TaTAMi operates under considerably fewer constraints, as will be discussed later.

1.2 Motivating Example

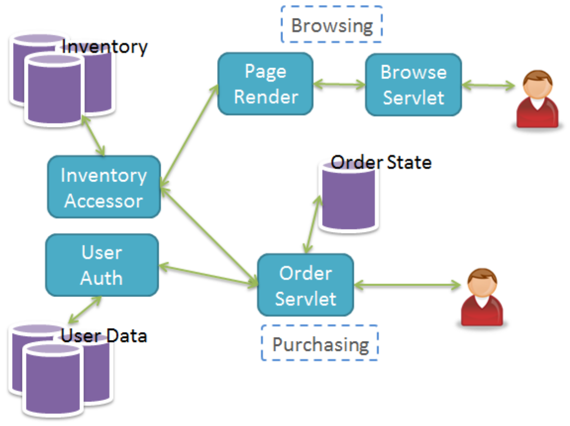


Figure 1.1: Example web store architecture.

To motivate the TaTAMi system and illustrate the natural fit of dataflow tracking to the problem at hand, consider the following:

1.2.1 The System

Presented is a simple online shopping site, the basic architecture of which is given in Figure 1.1. The system has back-end data stores for shop inventory and user account info and the logic to access them. For interacting with the site, there are two front-end components: the browsing component which accepts requests from the user to fetch shop items for display, formatted by the page render component; and the Purchasing component, which allows users to maintain a 'shopping cart' list of items they wish to buy and ultimately bill an order to their account.

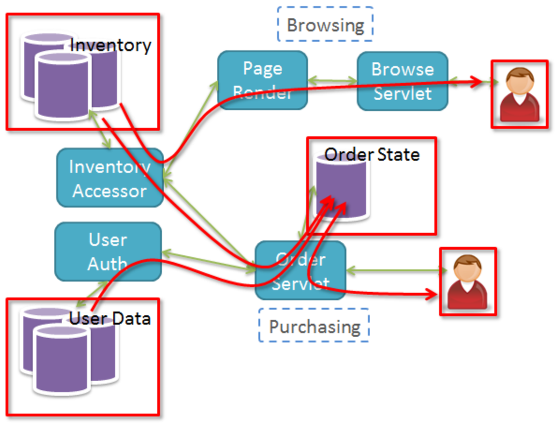


Figure 1.2: Sample dataflow for web store.

1.2.2 The Problems

Problem scenarios are as follows:

- More users begin using the site. The servers where the Browsing component resides are not fast enough and computations to render pages begin to get backed up.
- The development team has been asked to move the Purchasing component into the cloud, but the environment there only allows for stateless programs.
- The network link between the back and front-end components is carrying large amounts of data, and its maximum capacity may soon be reached.

To start, developers should have an overall picture of how the system works, or at least the part of it they are working on. As stated, we believe that capturing the dataflow of this system will naturally support developers in this effort. Figure 1.2 presents a possible dataflow graph for the system that our tool would obtain. This is very high level, the arrows indicating that data is communicated from component to component along them. The points to note here are the flow of shop item data from the Inventory database through the Browsing system to the user, and the flow of data from the back-end stores and user input into the 'Order State' shopping cart data. This

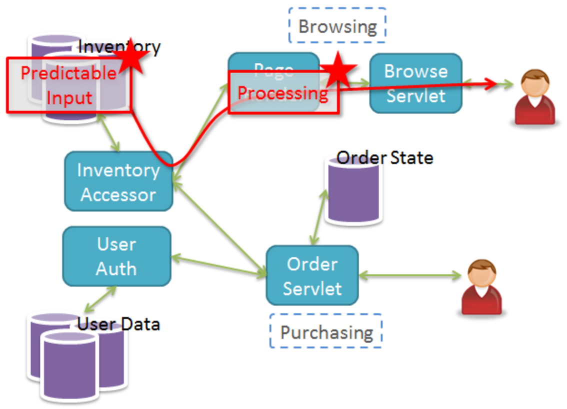


Figure 1.3: Dataflow indicative of possible caching opportunity.

view focuses immediately on how the application ferries and transforms data throughout its components, the key function of most web applications. It allows a developer to quickly answer the important question: "where did this output come from?"

The first optimization to consider, given that the Browsing component is backed up with excessive computation, is precomputation and caching. Consider Figure 1.3 and Figure 1.4, which outline two possible dataflow cases. The first shows how the dataflow could indicate that the data used to generate responses to user browse requests are entirely from predictable sources. Say that browsing involves simply displaying all of the item names, and the set of available items rarely changes, then the output browsing page would remain the same for long periods. The application could potentially be optimized by pregenerating the browsing page and serving it directly to the user, rather than going to the database and running the rendering code.

Figure 1.4 shows an alternate dataflow where taint tracking indicates that user input flows into the computation for the browsing page. Such could arise, say, if the user supplies some filters to search for items. The input may be very unpredictable, such as for a free text search, and thus the output browsing page may be unpredictable and a poor candidate for any kind of precomputation. On the other hand, if the user input occurs over some reasonably predictable range, such as if the items were being filtered over a set range of categories, the various output browsing pages may be good

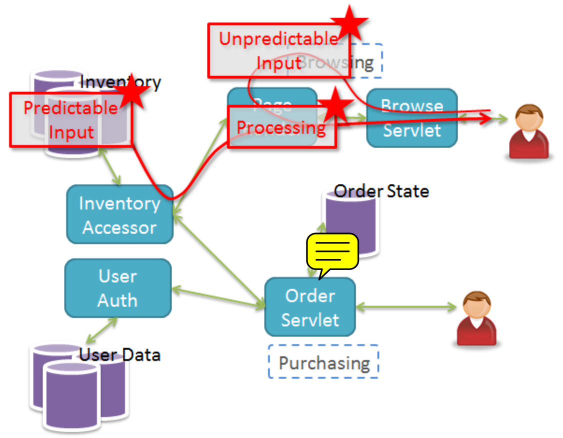


Figure 1.4: Dataflow indicative of sources to be careful of when implementing a cache.

candidates for caching. An automated analysis could point this out, and a developer is aided in applying this performance improving optimization, knowing which data sources to use for cache keys (user input parameters) and which to check for cache invalidation (the Inventory database).

The next concern is moving the Purchasing component into the cloud, to make use of the abundant computational resources there. In this example, the cloud provider doesn't support applications which keep in-memory state. Such would be the case if hosted applications were shut down periodically to conserve memory. Looking to the flow of data around the Order State persistent store, another use of dataflow tracking emerges. Dataflow can pinpoint with accuracy various points in a program where persistent data is stored. By following data into the Order State area, and observing that it persists across multiple requests to the application, an automated analysis can determine that this data is used to communicate data beyond the scope of a single request. Identifying such data is useful as it marks the distinction between stateful and stateless components in a system.

Taking this analysis further, Figure 1.5 shows a dataflow tracking which reveals how certain data items only flow to and from certain users. For data constituting persistent state, this might be personal data which is not used to communicate between multiple users, such as one's own shopping cart. Once

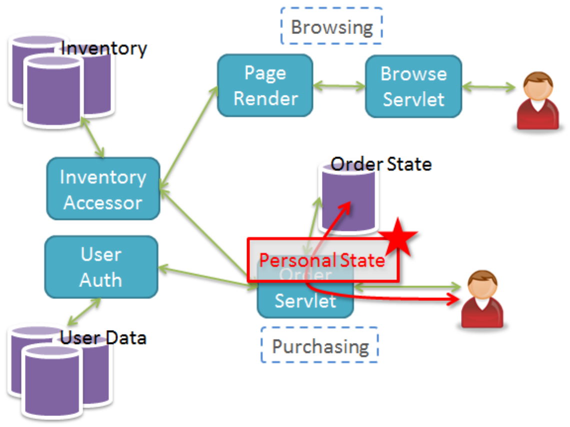


Figure 1.5: Simple dataflow indicating peristent state which is shared with only one user.

such state is identified by an analysis, one can make decisions like moving the state from the server into the client, making the application more robust. Such a suggestion would be valuable to the novice developer, indicating a way to get the state out of the Purchasing server-side components so that they could be safely moved to the cloud.

Finally, the back to front-end link capacity is a problem where analyzing dataflow is clearly useful. Partitioning is a kind of rearchitecting that might be performed on our web store. Figure 1.6 demonstrates a such a scheme, where the two front end components are moved into independent environments, communicating with the back-end over the network when database data is needed. In this scenario, knowing if the network link could be a problem is vital. Application partitioning algorithms seek to identify ways to split application components to make efficient use of resources while carefully managing communication overhead between partitions. This involves profiling the application to determine component execution and intercomponent communication costs. Obtaining a more complete view of the dataflow provides deeper knowledge into the communication cost. Not only can one know when components share data, but can additionally find out where that data is eventually used. In some cases, the data may never be used at all, and thus the dataflow tracking can obtain more accurate bounds on the communication cost. In this example, the heavy load on the link may be due to

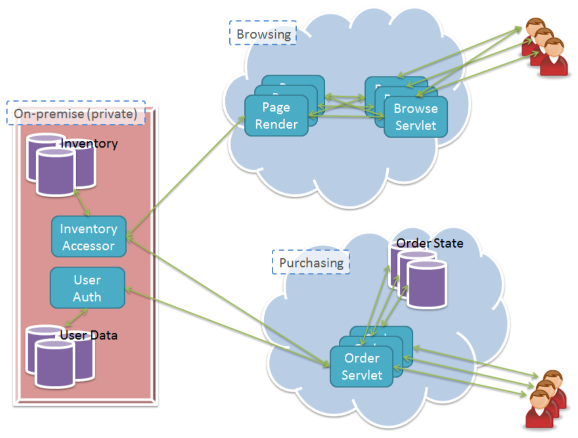


Figure 1.6: Example partitioning of application.

data which is not actually used, and which does not need to be transmitted at all.

1.3 Statement of Thesis

It is important to keep in mind that this work pursues a more general analysis approach, and that the form this thesis takes is of a more exploratory, completely qualitative nature. We do not explicitly seek to outperform other specific analysis efforts but rather to demonstrate the value of the DIFT technique for these kinds of analyses, something that we feel has not been significantly addressed in the literature. We hypothesized that due to the focus on data provided by DIFT and because web applications are often basically a means to allow users to access and share data, DIFT would naturally support analyses which could optimize web applications and help developers to understand important properties of them. We showed that several such analyses could be developed and then applied to DIFT traces of real-world web applications successfully, demonstrating both the viability of the analyses as well as the existence of the analysis targets in real applications.

1.4 Contributions

The main contributions of this work are as follows:

- A taint tracking tool for Java web applications, able to track tainted Strings and numerical values throughout a single program and report wherever they flow
- An analysis tool to both visualize taint tracking data and run analyses over it
- The identification and implementation of several analyses designed to operate over taint tracking data
- The demonstration that these analyses can be applied to real applications successfully

1.5 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 discusses the body of related work to cover attempts to address related problems and also to show how some of the techniques we use have not been sufficiently explored in this space. It also covers some relevant background material for technologies used in this project. Chapter 3 covers the implementation of the dataflow tomography tool and the analysis tool, breaking down their key components and explaining important design decisions. Chapter 3 presents an evaluation of the tools, demonstrating the output of the tomography tool and the results of the automated analyses. Chapter 4 finishes with a discussion of the results from the evaluation, possible future work, and overall conclusions.

Chapter 2

Related Work

2.1 Dynamic Information Flow Tracking

Program 2.1 Simple taint tracking example.

```
String userName = getUserNameInput();
// Any data from user input is tainted, so userName is marked as tainted.
String formattedData = "USERNAME:" + userName;
// The value of formattedData is based on userName,
// so it formattedData is marked as tainted.
```

The main stream of research contributing to this project is that of dynamic information flow tracking (DIFT). Introduced by Suh et al. in [26], DIFT refers to tagging (tainting) data used by an application at runtime so that its flow through the application can be tracked. One marks interesting sources - places in an application where input is received, such as by reading from a database or receiving input from a user - where data coming from a source in tainted. From the sources, DIFT systems employ some set of propagation rules which define how tainted data is spread through the system. As a simple case, when a variable is produced by using a tainted input value the taint tag should be copied from the input value to the new variable. This is shown in Program 2.1. Finally, DIFT systems will establish a set of monitoring sink points in the application to check and report if any tainted data flows through there. These could range from the arguments to a certain function to specific memory locations, depending on the needs of the tool. Many DIFT systems do taint tracking at a fairly low level, down to the level of individual bytes of an application's memory, in order to support such security analyses as memory corruption detection.

Taint tracking has been used almost exclusively in the domain of security, and such systems include [26] [23], where it is to detect if bytes from untrusted input sources like user input ever end up tainting such sinks as jump target addresses or executable instructions; [?], where it is used to trace sensitive data like passwords throughout a system to see where and for how long sensitive data is potentially accessible; [11], where the focus shifts to higher-level SQL injection and cross-site scripting attacks, checking if user input taints such sinks as database queries; and [2] [1], which identifies malware and network protocol analysis as uses. [2] is also interesting in that it focuses on a taint propagation policy known as control flow dependence, where values are tainted if they are assigned values in the scope of a control flow statement (such as an if branch) predicated on tainted data. Such propagation is difficult as it often leads to an explosion of taint in the system, and must be very carefully managed, and our own system does not use it as an intentional design decision. [14] additionally identifies data lineage tracing as a use, which focuses on determining the inputs responsible for outputs in various scientific computations, which is similar to some of the analyses we do. [12], [27], and [17] present more security-focused systems, as indeed the majority of taint tracking tools are. [16] and [8] describe Java taint tracking systems which modify the Java runtime to track taint in Strings. By contrast, our system uses AspectJ to obviate the need for such modifications, and we additionally have limited support for tracking taint in numerical values, as described in the Implementation chapter. These systems are, like other efforts, focused on preventing untrusted data from propagating to secure locations.

[21] presents something different, and was one of the key pieces in developing our own system. Unlike the previous systems, which are security focused and generally only place sinks at critical points where tainted data is not supposed to reach, dataflow tomogrpahy as introduced in this work does a heavier form of tracking. In tomography, tainted data is reported as it flows through every function call (or even instruction) with the ultimate goal of providing a complete end-to-end picture of how data flows through a system. The goal of their system, like ours, was to help users understand a target application. Their work shares our attention to visualization, noting the difficulty of displaying what are often very large flow graph, as well as our qualitative evaluation strategy. They make the point that while most uses of DIFT are focused on policy enforcement, as with security, their own work is one which focuses on using the analysis for discovery. They do not, however, give any concrete uses for their analysis beyond general understanding. We take a tomographic analysis, and use its output to drive a set of real, varied

analyses, demonstrating how the value of obtaining this data. We go further than general understanding, trying to develop tools which suggest specific ways to improve applications.

2.2 Data Update Propagation

Data Update Propagation (DUP) is a line of research presented in [4] [13] [6] [7] and [5] by Arun Iyengar et al. DUP is a scheme designed to help consistently cache dynamic data. It does this by maintaining a dependence graph between underlying data sources like database tables and cached objects, through any intermediate objects along the way. When underlying data is modified, the dependence graph is traversed to discover which cached objects need to be invalidated. In the first paper, the chief limitation of this strategy is the requirement that the application (and thus the developer) maintain the dependence graph itself through the use of a provided API. DUP is similar to the caching analysis performed in this thesis, where data flow graphs are searched to discover possible caches and all relevant inputs to them. In our case, however, the graphs are created automatically, without the requirement on the developer to manually specify them.

[13] does automatically generate dependence graphs, but only because the system caches at the level of immediate query results and the dependence graphs come from query analysis. These graphs desribe only the DBMS layer, and say nothing about what happens to the data once it is flowing through application code where other data items may be dependent on it. Our system builds dependence graphs automatically, in the application layer. [6] introduces such useful concepts as web page fragmenting, where dynamic fragments of a page are identified and able to be tied to underlying data, as well as page pregeneration. This last is much like another of the analyses we perform, known as precomputation, which seeks to find computation results, and even parts of output webpages which can simply be pregenerated due to being based on infrequently changing data. However, this work still uses the same manual DUP strategy developed earlier.

2.3 Automated Analysis and Decision Support

We present here an assortment of works which in some way seek to automatically aide developers in similar manners to our own work, and which use similar ideas to us. Fluxo [18] is a system from which we drew significant inspiration. In Fluxo, developers write applications in a restricted, higher level language which can then be compiled into a dataflow graph representation of the program. This representation is similar to what our own system obtains by monitoring dataflow in an application, though ours operates on unmodified, existing programs and thus the graphs tend to be less restricted and more difficult to work with. The paper supplies a series of transformations which can be applied to these graph representations to optimize the programs, with the goal of reducing end-to-end latency and addressing what they identify as a need to save developers from having to manually determine the best use of their infrastructure. These analyses include discovering caches, identifying precomputable data, finding computations which can be delayed until later, and separating stateful and stateless components. In Fluxo these optimizations can be applied automatically when found, due to the use of the restricted programming model, whereas in our work this is not yet the case. However, our analyses are run on real software systems developed in non-restricted programming environments. This presents many challenges, but is more realistic if real developers are to ever benefit from the work, and we do direct them to make similar optimizations.

The Fluxo paper is preceded by [25], which gives greater attention to the problem of discovering caches in a dataflow graph. Most importantly, this paper identifies some key points to consider when searching for caches which were encountered in our own work. These include attempting to choose caches which will actualy provide worthwhile execution savings and being sure that adding a cache does violate the semantics of the application, such as by skipping executions which have important side effects besides producing the cached data.

Tralfamadore [20] is a system which shares with our own the design where data is collected from an application dynamically at runtime to be analyzed repeatedly offline. Their system provides a much more extensive tracing at a lower level such that the execution of an entire OS can be replayed deterministically from the collected data. Using such data and their various analysis operators, one could even potentially build up a DIFT analysis similar to what is presented in the dataflow tomography paper or our own system. Like in Fluxo, the authors make the argument that developers are often given just the source code in order to understand a system, and thus are in great need of automated support tools to aide them. Like our system, they provide a varied set of analyses to be performed over the data their tool collects, though the examples presented operate at a lower level than our own reflecting the binary nature of the tool, determining such things as argument value distributions to instructions. The profiling and analyses we investigate are less general, but as such are more easily able to support the specific, higher semantic level cases we target.

Netflow [3] mines network activity logs to obtain dependency graphs for components in a network, such that one can answer such questions as which servers will be affected when some upstream component fails. This is at a much higher granularity than our own work, at the level of network devices rather than method calls in a program. However, they seek similarly to help developers understand complex systems using dataflow in order to facilitate low-risk changes. Presumably our analyses could be applied in a broader, distributed setting, given a more powerful taint tracking tool.

2.4 Application Partitioning

Application partitioning research seeks ways to take applications written to run in a single location and distribute their execution across physical devices and locations, for such benefits as offloading computationally expensive parts of an application to more capable hardware or locating specific services closer to the users which need them. Much research has gone into automatically analyzing applications to determine how best to partition them. In most cases, these analyses profile the applications to determine the execution costs of various components and the communication bandwidth between them. They then apply various algorithms to find a partitioning which places components to make efficient use of available resources while not incurring too great a communication overhead. This last point is important, as parts of an application's execution may be moved to a different physical location, and moving data to and from that location could use expensive network links.

Existing works generally use a very basic measure of communication between components, simply totaling the size of data sent in a communication event (such as a method call) [22] [24] [15] [10] [9]. While not a partitioning tool, the dataflows that our system extracts can be used to locate unnecessary flows, where data is communicated but then subsequently never used. By finding such flows, we envision that our tool could obtain more accurate bounds on optimal communication costs in a partitioning scenario, allowing better distributions of components to be found by automated tools. Our system also goes beyond this research in that it uses the dataflow information for analyses for a variety of optimizations which would be useful whether or not one wants to partition.

2.5 Aspect Oriented Programming

Aspect Oriented Programming (AOP) [19] is a technology that we rely heavily on to perform our taint tracking. AOP allows one to perform high-level instrumentation of existing programs, injecting code into them to add functionality. In order to augment a program with AOP, one writes pointcuts and advice. Pointcuts are descriptions of locations in a program where code should be added, such as "before every method call of classes within a specific package" or "whenever the value of a field on an object changes". Advice is the code which is injected into the program, and is associated with pointcuts to specify what code is added where. This naturally enables the rapid development of a high-level (and the level of methods and variables as opposed to instructions and bytes of memory) dataflow tomography tool, as one can create pointcuts targeting every location in a program where data is communicated and write advice which inspects the data communicated at those locations for taint.

2.6 Overview

In order to test the hypothesis developed in the thesis, two major tools had to be developed. The first is the Taint Tracking Tool (TTT), which performs a tomography-level tracking of targetted data for Java web applications. This tool relies on the AspectJ framework supporting Aspect Oriented Programming, and is compiled into target Java web applications to collect data from them as they run. The product of this tool is a log file, or a 'trace', describing events where tainted data moves from one location to another, which when taken together describe 'flows' of tainted data through a program as it executes. This log file is consumed by the second tool, the Analysis Tool (AT). The AT is responsible for a variety of tasks including supporting visualization of the data and allowing one to manually inspect very large traces in a controlled manner. Most importantly, the AT supplies a series of automated analyses which use the traces to discover useful properties of the target web application.

2.7 Taint Tracking Tool

Implemented in 5.2K lines of Java entirely using AspectJ and ajc as the compiler and bytecode weaver. No additional libraries were used.

2.7.1 Tool Components

The main components of the tool are as follows:

Taint I/O System

This makes use of AspectJ, monitoring certain method invocations to intercept target data as it enters an application and is written out to interesting locations. The system captures information about the source of data which will be useful for developers, marks the data as tainted so that it is tracked by the rest of the system, and uses the logging system to record these input/output events. To catch a database read, for example, the system monitors certain calls in the mysql connector library to catch the returning of ResultSets from PreparedStatement objects and reads the metadata from the PreparedStatement to get table/column descriptions as the source of the data. It then calls into the Taint Reference Management System to tell it that data read from the ResultSet should be treated as tainted. As an example of taint output, there are pointcuts on methods which set parameters for database queries and on methods which write to ServletResponse output streams, with advice to log these events. Finally, for this system it is desired that if acquiring input takes as parameters any tainted data, that input is additionally tainted by the tainted parameters, as is shown in Figure X. This is managed by the Taint I/O System as well.

Taint Reference Management System (TRMS)

The TRMS is responsible for keeping track of which objects are tainted. Other parts of the tool will call into this system to ask if a given object carries taint, and if so, where the taint came from. At the basic level, the system only keeps track of tainted Strings, character arrays, and numerical values manually targetted by the user of the tool. It doesn't track taint through primitives such as booleans and chars due to limitations of AspectJ and the fact that it cannot uniquely identify them. The TRMS marks Strings as tainted by keeping weak references to them in a map, mapping them to information about the taint source. Weak references are used so that the Objects can still be garbage collected.

Tracking of numeric values, even though they are primitive, is enabled through the following method: When a numeric value is input from a source to be tracked, the TRMS replaces it with a randomly generated value which is likely to be unique. The TRMS then maps the random value to information about the source of the value, just as it does with tainted Strings. Whenever the replaced value is output somewhere, such as to the user or as a database query parameter, the TRMS replaces it with the original value. There are cases where this form of tracking would change the operation of the program, such as if the numerical value is used in a loop counter, but in practice it was found that numeric values in need of tracking did not exhibit this behaviour.

The main function of the TRMS is the management of a backward reference tree. In order to properly track the flow of tainted data, the system needs to know when a Java Object is reachability tainted (see terms definition). A naive strategy for determining whether an object is tainted as such is to simply deep scan the object using reflection, however this is a costly operation and in practice inflicts too great a performance penalty on target applications. A better stategy is to have a system where when taint is assigned to an Object (by setting a field or adding to a collection, as caught by AspectJ), one can quickly mark the Object as tainted as well as any parent Objects (Objects from which the originally tainted Object is reachable). Thus, the TRMS maintains a map of child-parent relationships, which is updated whenever a field is set. When taint is removed from an Object, one can similarly remove it from any parent Objects by following the child-parent mappings. This kind of updating is useful for situations like the one depicted in Figure X.

There are, however, several problems with this strategy, due to limitations of AspectJ which had to be overcome. Unfortunately, AspectJ does not support pointcuts for array element access, nor is one able to instrument any classes in the Java runtime. This means that wherever an array or an object from the Java runtime is used, AspectJ is blind to reference changes made inside of it. The solution to this used here is to track the reachability of arrays and Java runtime Objects in the same way as tainted Objects themselves. Whenever one wants to check if an Object contains taint, also checked is if it contains such problematic Objects. If so, the contents of these Objects are scanned, recursively checking their contents for taint/problematic Objects in the same way as the original Object.

In order to boost performance futher, there are a series of pointcuts and advice for catching modification operations on most Java Collections Objects. This serves the same function as field set pointcuts for keeping the backward reference tree up to date, but brings it to a set of Java runtime Objects which ordinarily would have needed to be deep scanned.

This kind of bookkeeping is less important in traditional taint tracking, where checking if data is tainted occurs at fewer points, but becomes more useful when needing to check for taint very frequently as in tomography.

Tomography System

This relies on AspectJ to intercept various events where tainted data flows from one location to another. This fulfills the main function of the TTT, building a complete picture of where data one is interested in goes. The basic events that tracked are method calls, method returns, Object instantiation, and field set/get; and at each of these points the Tomography System calls into the TRMS to check if the arguments/return values are tainted. If they are, the logging system is called to record these events. In the case of a method return, the system doesn't just check the return value for taint; in many cases values are 'returned' by modifying the arguments to the call, so the system keeps a record of what taint each argument carried before the method was invoked and checks to see if they carry any new taint when the method is returning.

In addition to tracking these basic events, the Tomography System advises every method on String, StringBuffer, and StringBuilder Objects which cause several important events. These are termed 'propagation' events, where tainted Strings influence the value of other Strings (example cases would be copying a String or appending it to a StringBuilder); 'modification' events, where tainted Strings are changed in some way (examples being appended to or having characters deleted from); and 'composition/association' events, where multiple tainted Strings are combined or used together (examples being appending or equality checks). Propagation events are particularly important, as such Strings derived from tainted Strings must be added to the TRMS so that they are tracked by the rest of the Tomography System. The modification/composition/association events are logged as they are of use in performing certain analyses in the Analysis Tool.

Finally, the Tomography System employs some heuristics to attempt to track tainted Strings through methods which may 'lose' the taint. By 'lose' is meant that at a semantic level the method does pass tainted data from one location to another, but due to how it processes that data the kinds of checks allowed with AspectJ will not be able to properly track it. Real examples of this are cases where a String is processed and used to build a new String character-by-character, as was the case in some encoding methods encountered in tested applications. Properly tracking this kind of taint flow would require instrumentation at a lower level not possible in AspectJ, likely requiring one to taint individual bytes (as has been done in some taint tracking research). As such was not possible in the timeframe of this thesis, the issue was addressed using fuzzy String matching. A Levenshtein distance measure is used to check if non-tainted String outputs from a method execution match within some threshold any tainted inputs, and if so the taint is propagated from the matching inputs to the outputs. These events are separately logged, so that one can verify the correctness of the heuristic in each case, and in practice this worked well.

Basic Profiling

Code is included in the tool to gather supplementary data outside of taint tracking for the analyses. Such data could indeed be captured by other existing tools, but the implementation is provided here in order to keep the data collection process simple as well as because there would likely be problems when instrumenting a web application with multiple profilers at once. Currently there is logging to build control flow graphs and some simple profiling code to track method execution times.

Logging System

This provides a set of methods to be used by the other parts of the tool to log the events which together form the taint traces, as well as provide additional data useful for analysis. These logs are currently formatted in XML and output to a log file.

User Control System

This was added in order to provide some control over the dynamic operation of the tool. Rather than simply logging everything for every operation performed in a target web application, the control system provides socket communication with the tool in order to enable/disable various tracking components and the application runs. This allows one to disable, for example, tracking during web server initialization which may contain data one is not interested in.

2.7.2 Build Architecture

In order to completely track tainted data through an application's execution, one must instrument not only the application, but also every library that it uses. For instrumenting new web applications, and for whenever the tracker code needs to be modified, an Ant build script is provided to automate this process. Given a target web application, the build script performs the following:

- Compile the tracker code
- Precompile any JSP code that the web application uses, so that they can be instrumented as well
- Weave tracker code into all web application classes
- Weave tracker code into all web application provided libraries
- Weave tracker code into all of the servlet container's libraries

2.7.3 Example Operation of Tool

Notes: In order to better explain tool, give a high level but reasonably in depth breakdown of how the tool might operate.

2.7.4 Justification of Implementation

As has been discussed in this thesis, many taint tracking implementations exist, and even some for Java web applications. The problem is that an implementation that would provide Tomography-level tracking could not be found, which was necessary to address the research problems. Aspect Oriented Programming (AOP) was chosen as it conceptually provided the high level functionality needed to perform the tracking, saving on effort needed to write the system and allowing the tracker to be implemented in the available timeframe. The alternative would likely have been to learn the use of a Java bytecode manipulation framework, such as ASM or BCEL, and build up the tracker from a much lower level. AspectJ was specifically chosen as it was the most mature, well-documented, and stable implementation of AOP available for Java programs.

2.8 Analysis Tool

Implemented in 5.7K lines of Java using the JUNG framework for visualizating and working with graph data.

2.8.1 Tool Components

The main components of the tool are as follows:

Visualization System

This relies primarily on the JUNG framework to provide a more user-friendly means of working with the taint tracking data. To use the tool, a log file is specified which is then used to generate an internal JUNG graph representation. The presentation of taint traces in graph form is as depicted in Figure [NEED TO ADD FIGURE]. Nodes in the graph represent locations where tainted data has been detected; these are mostly method calls, where a node is identified by its class name, method name, argument types, and Object ID if the method is non-static; fields, with a class name, fieldname, and Object ID if needed; and input nodes, which provide information about where tainted data originated. The nodes are connected by directed edges describing each event where tainted data flows from one location to another, listing the type of event (Call, Return, Field Get/Set, etc) and an identifier. Using the identifier, the user can view the details of the tainted data flowing on a particular edge.

The entire graph can be viewed all at once, but the traces are often so large that it is impossible to comprehend the output. To support a more controlled viewing, the Visualization System supplies several filters to look at subsets of the full graph. These are:

- TaintID Filter: Presented in the tool is a tree-style list of the taintIDs in the trace, as is shown in Figure X. At the top level of the list are original taintIDs assigned to data as it was read from tainted input sources. Expanding an item reveals taintIDs which are result of Propagation events from the parent taintID. Selecting an item will cause the graph to only show edges which carry taint which matches the selected ID. Selecting the 'Deep' checkbox extends this to show the selected taintID and any IDs which were propagated from it, giving a complete picture of where data with the selected taintID flowed.

- RequestID Filter: For traces taken over multiple requests to the application, this allows a user to view the data collected for a single request.
- Forward Graph Filter: This allows one to specify an edge, which is then expanded into a graph by following the flow of tainted data carried by that edge. It makes use of the caller/called ContextCounter information from the logs to only expand to edges which were actually 'caused' by the specified edge.
- Manual Filter: For fine viewing control, this allows a user to view only selected nodes, and then expand to neigboring nodes to track the flow of tainted data step by step.

Having these tools for working with the graphs, while no substitute for the automated analyses, is nevertheless very important when a user needs to make use of the data. The results of analyses are themselves often presented in the same graph form, which can still be difficult to understand without some filtering.

Graph Preprocessing

Before the graphs are displayed or analyzed, the Analysis Tool performs proprocessing steps on the data in order to better work with it. The first is the addition of implied edges. There are cases when tracking the flow of tainted data where representation of it requires some care. Consider for example the case in Figure X where a tainted String is used to construct a new StringBuilder; taint flows from the calling method to the StringBuilder constructor. If the StringBuilder's toString() method is then called, taint flows from the toString() method back to the calling method. As is shown in the figure, the taint is shown to flow out into one StringBuilder node, and then back from another one, with no connection between them even though intuitively tainted data flows from one to the other. Solutions to this could be to group the StringBuilder nodes somehow, or to have a node for each Object rather than each Object method, but these either don't support analysis well or lower its granularily. Instead, the graph is searched for such cases where taint flows into an Object through some method call and that same taint (or taint propagated from it) flows out of the Object through a different method returning. Implied edges are added between the nodes in question to reflect the flow of tainted data.

The second preprocessing step finds unused flows of tainted data. These are cases where tainted data is communicated from one location to another, but is subsequently never used. This step examines reachability tainted Objects in graph edges to see if the directly tainted Objects they carry are ever accessed from them. This is done by scanning ahead in the graph, looking for flows carrying the directly tainted Objects themselves. If not, the directly tainted Objects are marked as unused for the edge and reachability tainted Object. This preprocessing step is useful for some of automated analyses which follow.

Notes: Additional preprocessing step where field reads are checked and matched against other nodes in the graph to show how taint flows into fields. Simply annotate other method nodes if the objects of those nodes are tainted fields elsewhere in the graph. Could also add implication edges, but this may make less sense.

Input Description File

This is an XML file created by the user which supplies information needed by some of the analyses. Its format is given in Appendix X. It is read by the analysis tool and gives a measure of how stable various data sources (database columns and user parameters) are, from producing values which change rarely to completely unpredictable input.

Automated Analyses

The main purpose of the Analysis Tool is to provide easy to use automatic analyses over taint trace graphs. After loading a trace, a user need only select an analysis which runs without guidance until complete. For most analyses, the results will be presented as a series of graphs added to the viewing area with some supplementary text explaining the results.

2.8.2 Available Analyses

Caching

The caching analysis seeks to find opportunities to save on computation/communication by suggesting locations where caches could be considered. By a cache we mean code and data storage mechanisms which take the results of computations over some inputs, and store the results - mapping the inputs to them. The next time the computations are invoked with the same inputs, we can return the stored results rather that redoing the computation. For a cache to work, we need to be aware of all inputs to the computation. If any inputs which affect the results are missed, when those inputs vary the cache may return invalid results from the store. What this analysis essentially does is look for regions of the graph which are only carrying taint from non-random inputs. By non-random we mean inputs with reasonably predictable values over a small enough range that the cache will be useful and not frequently missing. This analysis relies on the input description file described in Section X to determine whether or not given tainted data is random. The assumption is that if all of the inputs to a computation, represented in the graph by a network of nodes, are predictable enough, the computation is a good candidate for caching. The identified subgraph can then be presented to the user as an indication of the possibility of a cache as well as a guide for what parts of the application must be considered when doing so.

The algorithm for performing this analysis is as follows:

Precomputation

This analysis is essentially the same as the one for caching, and in fact both are computed together. The difference is that in the case of precomputation, the inputs in question must change very rarely. If this is the case, instead of having a cache implemented the user is advised to simply compute the result of the computation in advance and return it wherever the computation normally would have taken place. It is then their responsibility to update this precomputed value if the inputs should ever change.

The algorithm for this analysis is the same as for caching, except that the requirement on the variability of tainted data in subgraphs is set more strictly to only allow very stable sources.

Postcomputation

The Postcomputation analysis looks for flows of tainted data which represent computations which could be deferred until later. This is in the context of a user submitting a request to a web application, where we are interested in opportunities to send the user a response more quickly. For some applications, the user may only be interested in data which composes the response web page for a submitted request, which we call user output. Computation which outputs to the database or other parts of the application may not need to be complete before sending the user the response, and this analysis attempts to locate such computations. At a high level, this is done by tracing the flows of tainted in the graph, looking for subgraphs carrying only taint which never flows to a user output.

The algorithm used for this analysis is as follows: ADD ALGORITHM DESCRIPTION

Application State

The goal of this analysis is to locate persistent state in an application. This does not refer the data an application keeps in a database, but rather to such more temporary state as might be kept in sessions stores and static variables. This kind of state can be used to keep data associated with users to facilitate their interaction with a site over multiple requests. Examples include shopping carts, which allow users of shopping sites to collect items as they browse them, to be bought together on checkout; or something as simple as a username, displayed on each page. Knowing where such state is is useful for developers as it must be managed carefully in various scenarios. When replicating computation which relies on such state, it needs to be kept up to date and distributed to all locations where required. When migrating an application to a new environment, certain kinds of state may not be well-supported. For example, if migrating to the Google App Engine platform, state in static variables would be interfered with by the system's tendancy to shut down idle applications. This analysis identifies such state by looking for instances where the same pieces of data are accessed in over multiple requests, then presents to the user subgraphs showing where the state is stored and what parts of the application it is communicated to.

The simple algorithm used to do this is as follows: ADD ALGORITHM DESCRIPTION

User State

This analysis is similar to the one which locates application state. It goes beyond it by trying to determine whether a given piece of such state is used to communicate data to only a single user. The shopping cart example given for the application state analysis describes such state, as no other user need view another's cart. This is opposed to persistent data which supports interaction between users, such as a chat window for sharing messages or a list of online users. The motivation for finding such state is to identify opportunities to relocate state to the user. If the data is only shared with a single user, then it could potentially be moved from the server to the client. A user's browser could store the items in a shopping cart, and submit them to the server only when needed. This has such advantages as allowing a user to keep application state despite problems on the server, or easily carry their state with them if their requests need to be directed to another server hosting the application. Personal state is identified by generating a trace while interacting with the application with multiple users, and then looking for data which is only ever accessed by a single remote IP address.

The algorithm used for this analysis is as follows: ADD ALGORITHM DESCRIPTION

Wasteful Communication

Section X describes a graph preprocessing step which the Analysis Tool performs to identify instances where tainted data is communicated between locations but subsequently never used. Given this step, this analysis is easy to perform, and merely needs to compile a report of where data is communicated wastefully by checking the edges in the graph for taint marked as such.

An obvious use for such an analysis is to help a developer potentially eliminate these communication. Such is especially useful if the application is to be partitioned and such communication would be crossing costly boundaries. Another use for this is also motivated by application partitioning, where the data can be used to improve the analyses used in that space. Application partitioning algorithms generally consider module execution costs and communication costs between them when determining an optimal way to group and separate modules. A simple strategy which monitors inter-module communication events (such as method calls), can report the cost of the communication as the total size of the data communicated. However, some of this data may never be used, and knowing this can allow for better communication cost estimates. Such could lead to more optimal partitionings, as long as there is a mechanism in place to avoid the wasted communication, such as by eliminating it altogether or employing a lazy communication method where the data is only communicated when it is actually needed.

Access Path Refactoring (APR)

This last analysis is arguably the most complicated and ambitious. APR is the name we give to an analysis which attempts to discover beneficial modifications to how an applications accesses its database-resident data. This is to be used mainly in a partitioning scenario, where the data used to fulfill various requests may end up crossing expensive partition boundaries. APR takes as input a suggested partitioning of modules in a trace, and attempts to position database sources, at the granularity of columns, to lower communication costs. To do this the analysis uses the taint tracking data to locate dependencies between data sources to attempt to determine the consequences of moving various data sources - moving one data source to another partition which uses the data can save communicating data from that source, but may then require additional data from other sources to be sent to the new partition. This kind of savings/cost balance is evaluated to determine a possible 'best' arrangement of data sources.

The algorithm that we use to implement this analysis is as follows: ADD ALGORITHM DESCRIPTION

2.9 Evaluation Strategy/Goals

In using the Tracking and Analysis tools to test the claims made in the thesis, the focus was on performing a more qualitative evaluation. This was in part due to time constraints, as a proper quantitative evaluation would have required testing the tools with a wider range of applications, modifying each application according to the results of the various analyses, and testing those applications in realistic environments to assess the modifications. While such would certainly be interesting, it is beyond the scope of this thesis and must be left to future work. Additionally, a qualitative evaluation is better suited to a sufficiently succinct demonstration of the tools' usefulness. The goal was not to provide an in-depth look at any one analysis supported by the taint tracking data, but rather to show how the technique of taint tracking could be used to support a variety of useful analyses, and to show that these analyses could be applied successfully to realistic web applications. To this end, the evaluation strategy presented here is to take an application, apply an analysis to it, and then justify the correctness and usefulness of the results through manual code inspection, knowledge of the application, and light testing of the application. By showing how each of a wide range of analyses are actually successful on such applications, taint tracking is demonstrated as a robust application analysis technique.

2.10 Evaluated Applications

The applications selected for the evaluation were RUBiS, an auction site prototype, and jGossip, a web forum. What follows is a justification for the choice of each application, as well as a brief description of their functionality and design.

2.10.1 RUBiS

RUBiS is a realistic application, providing all the necessary functionality for users to put items up for auction, browse running auctions, and make bids on items. It is already a popular choice in various research efforts (REFERENCE?), making it a desirable representative example. The small code size made it easy to understand and manually inspect, which was of great help when developing and testing the analyses, but at the same time it was not written to be easily amenable to such analysis. As such it served as a reasonable introductory proving ground for the taint tracking and analysis tools.

RUBiS Details

The functionality of RUBiS is quite simple. Visiting the homepage for the application, users are presented with links describing various actions they can take in the auction, including registering an account, browsing items for auction, and selling an item of their own. Items are grouped by geographical region and item category, and a user navigates through several pages to select a region and category before bidding on or selling any items. A user does not actually log into the application to get a persistent session, but rather supplies their username and password whenever taking a sensitive action such as selling and bidding.

Architecturally, RUBiS is a very simple servlets application, relying on no external libraries beyond a MySQL driver for database connectivity. Every action on the site has a servlet dedicated to it, such as the BrowseCategories servlet, which writes out a list of item categories for the user to choose from, or the RegisterItem servlet, which takes parameters from a web form describing an item and creates a new auction from them. Most of the servlets make use of a ServletPrinter object which provides a series of methods for writing html output to the servlet response stream.

2.10.2 jGossip

jGossip was found by searching SourceForge for popular open-source Java web applications of greater complexity. It was briefly checked in the early stages of the research to see if it likely contained properties which would be amenable to interesting analyses, and along with several other applications was deemed promising. It was chosen from among these as web forums are a commonly used kind of application, and as such jGossip was a more representative example.

jGossip Details

jGossip is interesting in that it makes extensive use of libraries, such as the Apache Struts framework, JSP, and the JSP Standard Tag Library (JSTL). This is partially desirable as many applications make use these libraries, so jGossip serves as a more realistic example. However, the use of some libraries presents some difficulties for the kind of analysis performed here. In particular, JSP is somewhat problematic in that it allows application code to be created from HTML-like markup. JSP pages are compiled into Java code, and generally a developer never works with this generated code. JSTL compounds the 'problem' by providing a large set of JSP tags to perform various kinds of logic which would normally be written directly in Java. For applications which make extensive use of JSP, it may be that automated analysis results suggest making changes within generated code. Such may be less meaningful as the mapping from JSP to the generated code, and thus how to modify the generated code, may not be obvious. However, as JSP is a popular technology in Java web applications, attempting to analyze an application making use of it is a valuable exercise.

Users log into jGossip to get a persistent session to interact with the forum. Whenever data is needed from the database, such as to get information about a user or display a forum, a data access object generally instantiates and fills in various 'bean' objects (serializable, with getters/setters to store properties) from the database data. These objects are then stored as session attributes to be loaded and reused later.

2.11 Completeness of Tracker

One of the minor goals of the research was to develop the Java taint tracker itself. It is important that the tracker is complete, meaning that it is capable of tracing tainted data wherever it goes. If this were not the case, the analyses would be less reliable, having incomplete data to work with. The tracker presented here chiefly traces only data in Strings read from designated input sources, as manually specified numeric values. As this tracker does not consider taint propagation by control dependence, such as a tainted variable influencing the value of another through a control-flow statement, the completeness of the tracker can be evaluated somewhat simply.

First, the tracker needs to intercept every operation which communicates data in a program. Using AspectJ, the tracker is able to inspect the data flowing on every method call, method return, and field set/get, which captures the flow of all program data (we do not consider data exchanged outside of the context of the application). The only issue here concerns code from the Java Runtime, which cannot be instrumented. It would be hypothetically possible, for example, for such non-instrumented code to take a tainted String, copy it, and store it somewhere. The copy would not be tainted as the call to the copy method would not have been intercepted due to lack of instrumentation. In practice, however, we found that such problems did not occur. To check that all data which should be tainted actually was, the test applications were supplied with input data which could be easily identified. By performing String matching on any communicated data, the presence of data derived from the input sources could be identified, and then the taint tracking system could be checked to ensure that it had properly marked the data as tainted and from the correct source. The test applications were manually driven through various operations, looking for any data which seemed as though it should be tainted but was not. Eventually, after sufficiently developing the tracker, no such data was present, and the tracker was judged as being reasonably complete.

2.12 Application Results

2.12.1 RUBiS Input Source Description File

Figure 2.1: RUBiS Input Description File.

```
Database: rubis/categories/id - variability: STABLE
Database: rubis/categories/name - variability: STABLE
Database: rubis/regions/id - variability: STABLE
Database: rubis/regions/name - variability: STABLE
Database: rubis/users/id - variability: STABLE
Request: /rubis_servlets/SubmitChat:chatMessage - variability: RANDOM
Request: /rubis_servlets/BrowseCategories:region - variability: PREDICTABLE
Request: /rubis_servlets/BrowseCategories:nickname - variability: PREDICTABLE
Request: /rubis_servlets/BrowseCategories:password - variability: PREDICTABLE
```

2.12.2 jGossip Input Source Description File

Figure 2.2: jGossip Input Description File.

```
Database: jgossip/forum/forumtitle - variability: STABLE
Database: jgossip/forum/forumdesc - variability: STABLE
Database: jgossip/forum/forumid - variability: STABLE
Database: jgossip/forum/locked - variability: STABLE
Database: jgossip/group/groupname - variability: STABLE
Database: jgossip/group/groupid - variability: STABLE
Database: jgossip/group/groupsort - variability: STABLE
Database: jgossip/message/content - variability: RANDOM
Database: jgossip/message/from - variability: RANDOM
Database: jgossip/message/id - variability: RANDOM
Database: jgossip/thread/timestamp - variability: PREDICTABLE
Database: jgossip/thread/tid - variability: PREDICTABLE
Database: jgossip/thread/sortby - variability: PREDICTABLE
Database: jgossip/whois/id - variability: PREDICTABLE
Database: jgossip/whois/ip - variability: RANDOM
Database: jgossip/whois/sessionid - variability: RANDOM
Database: jgossip/whois/username - variability: PREDICTABLE
Database: jgossip/skinparams/paramname - variability: STABLE
Database: jgossip/skinparams/paramvalue - variability: STABLE
Database: jgossip/constants/name - variability: STABLE
Database: jgossip/constants/value - variability: STABLE
Request: ShowForum:fid - variability: PREDICTABLE
```

2.12.3 Example Trace

In order to understand the evaluation which follows, consider Figure 2.3, which shows a visualized taint trace taken from loading a page in the RUBiS application. The page loaded contains a list of item categories, showing the category name to the user in a hyperlink which includes the category id as a parameter. At the labeled point 1 in Figure 2.3 are two input nodes for database data used in displaying the page. The format of the node label for these is: column accessed / followed by [catalog / table / column] for each column selected by the query. A ResultSet which includes the id and name columns from the categories table in the rubis catalog is used, and a seperate node is present for each column accessed (the target column). The edges out of these nodes are labeled 'RIN' for 'Returning Input', marking them as points where tainted data originates. Edges are labeled with numbers to indicate the ordering of events. The full graph actually contains more edges than is shown, but for presentation multiple edges of the same type between the same nodes have been reduced to a single edge, explaining the 'missing' edge numbers. From labeled point 1 to 2 is shown the return path of the category names and ids through the getString and getInt methods to the categoryList method on the BrowseCategories servlet.

This data is then passed to a printCategory method on a ServletPrinter object at point 3. This is where the data is formatted for display to the user. Notice the three similar groups projecting from this point, two of which are labeled at point 4. These show how the data is formatted by concatenating it with other Strings, which is performed by appending to StringBuilders. After appending, the toString method is called on each StringBuilder to get the concatenated String, and the preprocessing described earlier adds 'IMP' (implied) edges to show how the data sent to the append methods is coming back on the toString call. At point 5 is shown how the data is passed to an encode method which processes the data in a way which would have lost the data were it not for the fuzzy propagation methods described earlier. The 'FZZ' edge is present to indicate the use of this heuristic. Finally, the categories are written out to the response output stream, at point 6 depicted as an 'OUT' edge to a println method.

2.12.4 Precomputation

RUBiS

Figure 2.5 shows the results of running the precomputation analysis over the same trace depicted in Figure 2.3. The first thing to notice here is that it is the same graph. Such is not always the case for this analysis, but in this instance the input trace was simple enough that the analysis identified the entire trace as being precomputable. Consider the request in question, which only needs to access the categories table to list the names and ids of the categories to the user. Both the name and id columns of this table have been manually marked as being stable. This means that one does not expect category names or ids to be unpredictable - the same values will be present in the table for extended periods of application use. Given this information about these values and the input trace, the precomputation

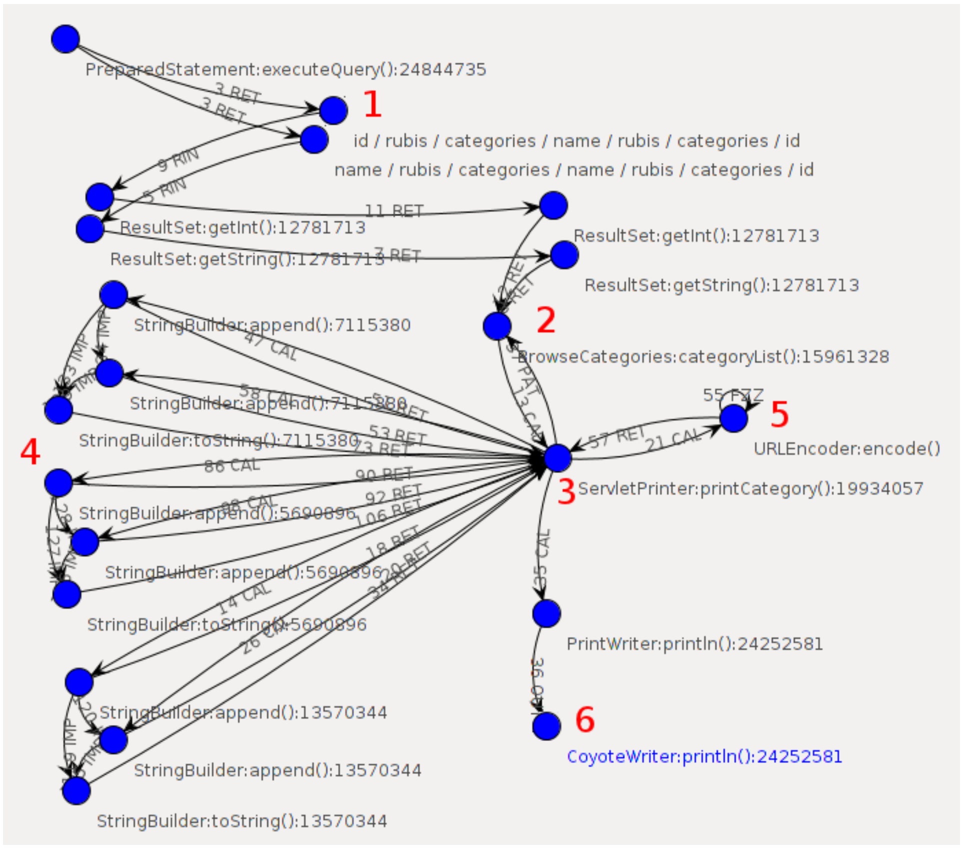

PreparedStatement:executeQuery():24844735
1
id / rubis / categories / name / rubis / categories / id
name / rubis / categories / name / rubis / categories / id
ResultSet:getInt():12781713
ResultSet:getString():12781713
ResultSet:getInt():12781713
ResultSet:getString():12781713
2
BrowseCategories:categoryList():15961328
StringBuilder:append():7115380
StringBuilder:append():7115380
StringBuilder:toString():7115380
4
3
ServletPrinter:printCategory():19934057
5
URLEncoder:encode()
StringBuilder:append():5690896
StringBuilder:append():5690896
StringBuilder:toString():5690896
PrintWriter:println():24252581
6
CoyoteWriter:println():24252581
StringBuilder:append():13570344
StringBuilder:append():13570344
StringBuilder:toString():13570344

Figure 2.3: RUBiS Browse Categories Trace.

analysis determined that the tainted data flowing in this graph was entirely stable data, and thus the output of the analysis is the entire graph with the inputs and outputs coloured orange and red respectively. Additional data is obtained by viewing the details of the analysis, which mainly shows tainted data sent to each output node in the output graph. For this graph, the output node's data shows the HTML for three hyperlinks as in Figure C1b, these being the categories presented to the user.

Interpreting this, a developer can reasonably assume that for the request responsible for this trace, the computation which reads category names and ids from the database and displays them to the user in this hyperlink form can be precomputed. At no point does this computation seemingly make use

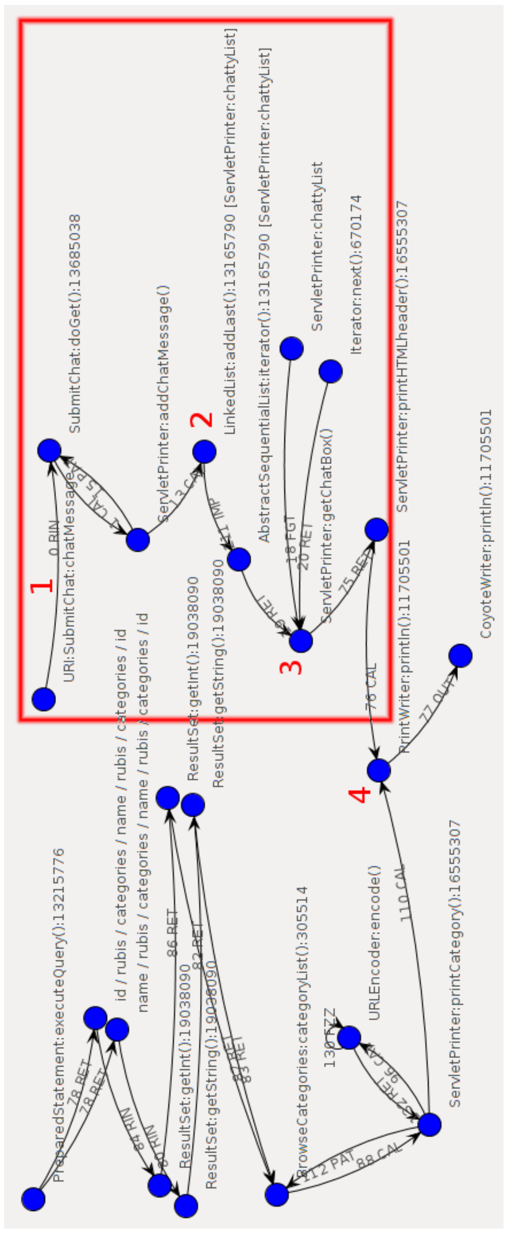

1
2
3
4
URI:SubmitChat:chatMessage
0 RIN
SubmitChat:doGet():13685038
ServletPrinter:addChatMessage()
13 CAL
LinkedList:addLast():13165790 [ServletPrinter:chattyList]
AbstractSequentialList:iterator():13165790 [ServletPrinter:chattyList]
ServletPrinter:chattyList
Iterator:next():670174
18 FGT
20 RET
ServletPrinter:getChatBox()
75 RET
ServletPrinter:printHTMLheader():16555307
ServletPrinter:println():11705501
76 CAL
PrintWriter:println():11705501
77 OUT
CoyoteWriter:println():11705501
PreparedStatement:executeQuery():13215776
78 RET
id / rubis / categories / name / rubis / categories / id
name / rubis / categories / name / rubis / categories / id
84 RIN
81 RIN
ResultSet:getInt():19038090
ResultSet:getString():19038090
86 RET
82 RET
87 RET
83 RET
BrowseCategories:categoryList():305514
112 PAT
88 CAL
URLEncoder:encode()
130 FZZ
ServletPrinter:printCategory():16555307
110 CAL

Figure 2.4: RUBiS Browse Categories With Chat Trace

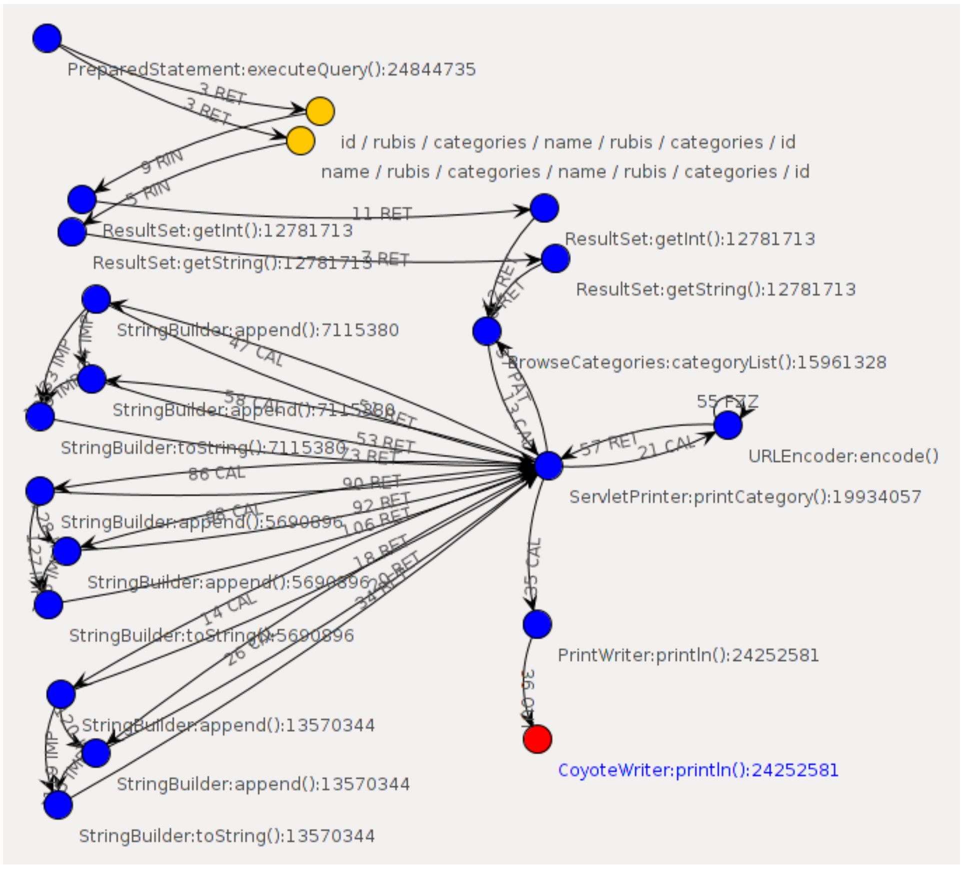

PreparedStatement:executeQuery():24844735
3 RET
3 RET
id / rubis / categories / name / rubis / categories / id
name / rubis / categories / name / rubis / categories / id
9 RIN
5 RIN
ResultSet:getInt():12781713
11 RET
ResultSet:getString():12781713
3 RET
ResultSet:getInt():12781713
ResultSet:getString():12781713
StringBuilder:append():7115380
47 CAL
BrowseCategories:categoryList():15961328
StringBuilder:append():7115380
58 CAL
StringBuilder:toString():7115380
53 RET
73 RET
55 FIZ
57 RET
21 CAL
URLEncoder:encode()
ServletPrinter:printCategory():19934057
86 CAL
90 RET
StringBuilder:append():5690896
98 CAL
92 RET
106 RET
StringBuilder:append():5690896
18 RET
StringBuilder:toString():5690896
14 CAL
26 CAL
35 CAL
PrintWriter:println():24252581
36 CAL
StringBuilder:append():13570344
StringBuilder:append():13570344
StringBuilder:toString():13570344
CoyoteWriter:println():24252581

Figure 2.5: RUBiS Browse Categories Precomputation Results.

of any unpredictable data in generating the category hyperlinks, and so one could alter the application to essentially immediately output the hardcoded hyperlinks instead of fetching the data from the database and formatting it. If and when the category data does happen to change, one would then have to update the precomputed data, but it is assumed that this would happen very rarely.

The analysis can be taken further in this case. The tracking code not only logs tainted output, but also creates logs for non-tainted output. Using these one can observe the mixing of tainted and non-tainted data as it is composed together in the output. Looking to Figure C1, generated in the

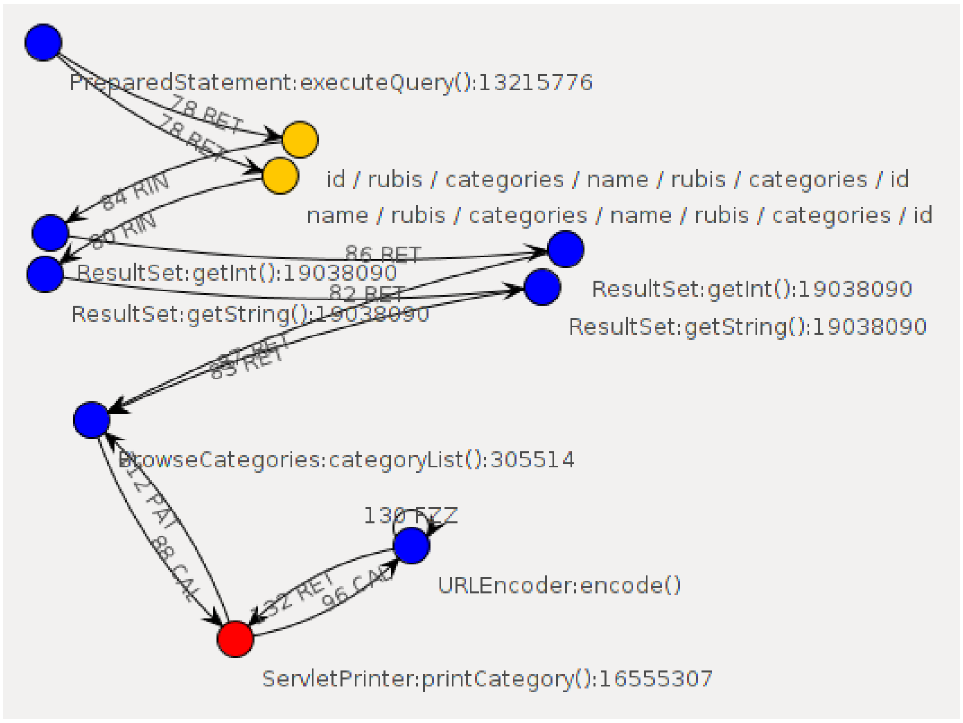

PreparedStatement:executeQuery():13215776
78 RET
78 RET
id / rubis / categories / name / rubis / categories / id
name / rubis / categories / name / rubis / categories / id
84 RIN
80 RIN
86 RET
ResultSet:getInt():19038090
82 RET
ResultSet:getString():19038090
ResultSet:getInt():19038090
ResultSet:getString():19038090
87 RET
83 RET
BrowseCategories:categoryList():305514
112 PAT
88 CAL
130 FZZ
URLEncoder:encode()
132 RET
96 CAL
ServletPrinter:printCategory():16555307

Figure 2.6: RUBiS Browse Categories With Chat Precomputation Results.

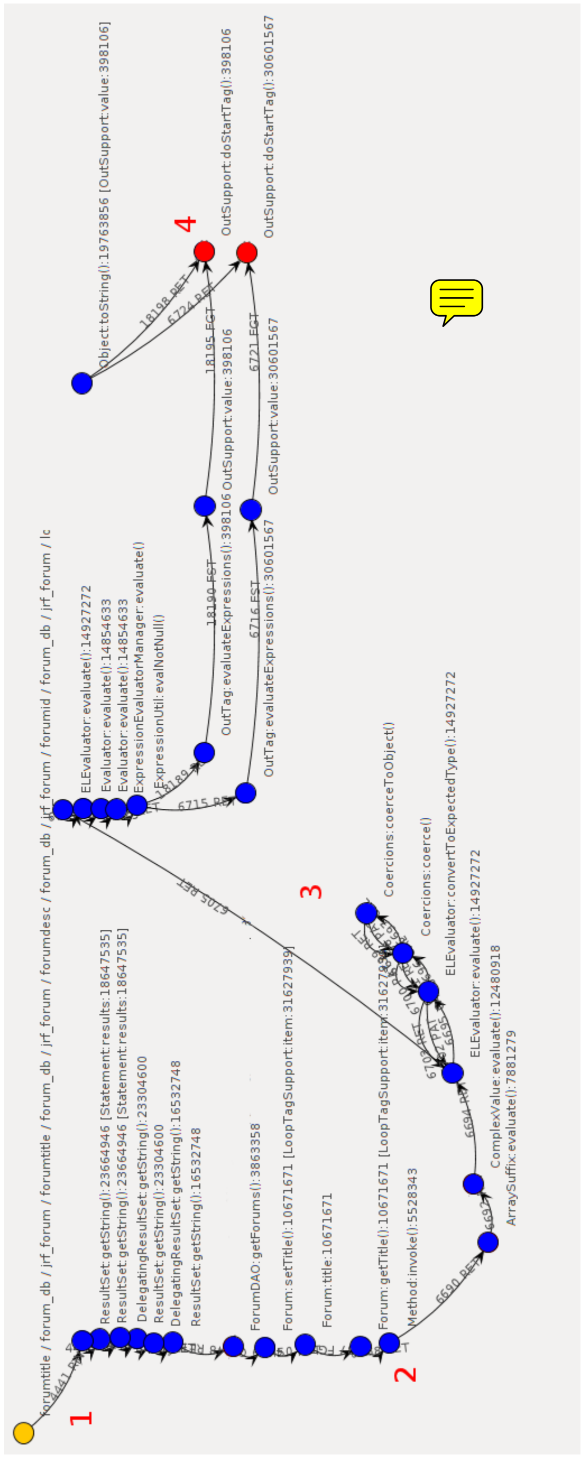
analysis tool from the output logs, we can see that the only tainted output involved in the response web page is from the three hyperlinks identified in the precomputation analysis. The rest of the data is non-tainted, and for this application this equates to it being hardcoded. Given these analysis results a developer can decide to simply precompute the entire response for this category listing page, saving on database access and formatting computation.

Going further, RUBiS was modified to display a chat box on every page, allowing anyone to post messages to the server for everyone to see. Whenever a message is submitted by a user, it is posted to a SubmitChat servlet which is responsible for adding the message to a list of messages. The important thing to note about this data is that it is very unpredictable. The list of messages can change at any time, with random data input to them, and as such any computations which work with this data are poor candidates for any kind of caching. Figure 2.4 shows a trace for the same page load as before, but including a post to the chat box. The left side of the graph in the figure is as before, the portion outlined in the red box shows the additional flow of the chat message data. This new portion of the graph actually spans two requests to the application - one to submit the chat message and another to display the list of categories along with the new message. The portion of the graph from labeled points 1 to 2 covers the submission of the message, starting with the source node showing the relative URI of the request and the name of the tainted parameter, 'chatMessage'. This parameter is accessed by the doGet method of the SubmitChat servlet, passed to the addChatMessage method of a ServletPrinter Object where it is concatenated with some formatting text, and finally appended to a LinkedList at point 2 by the addLast method. Note that the label for this addLast node ends in the name of a field, the 'chattyList' field of a ServletPrinter Object. This means that this field points to the LinkedList in question and indicates that by storing tainted data in the LinkedList it is effectively stored in that field. Such is useful for a developer to know when interpreting these graphs and mapping them back to the code responsible for them. Point 3 shows where the chat message data flows for the category list request, as a LinkedList is read from the chattyList along edge 18 in the getChatBox method. Following that an iterator over the LinkedList returned along edge 19, and the iterator is accessed along edge 20. The messages inside are concatenate with some formatting text before being passed along to point 4, being returned to the printHTMLHeader method of a ServletPrinter Object and written to the user response stream along with the list of categories.

Figure 2.6 shows the result of the precomputation analysis on the trace from Figure 2.4. It is exactly like the analysis results from Figure 2.5 except that the precomputation ends at the red PrintCategory node, no longer extending to the println nodes. This is because the println method on the same PrintWriter Object now also receives the random data of the chat messages, and thus one cannot replace calls to this method with precomputed data. Given the precomputation output in Figure C2b and the response output breakdown in Figure C2, a developer can determine that the page can be mostly precomputed, only needing to dynamically generate the portion tainted by the chatMessage data.

jGossip

Loading the main page of the jGossip application, which presents users with a list of available forums, and running the precomputation analysis over



1
2
3
4
forumtitle / forum_db / jrf_forum / jrf_forum / forumtitle / forum_db / jrf_forum / forumdesc / forum_db / jrf_forum / forumid / forum_db / jrf_forum / lc
ResultSet:getString():23664946 [Statement:results:18647535]
ResultSet:getString():23664946 [Statement:results:18647535]
DelegatingResultSet:getString():23304600
ResultSet:getString():23304600
DelegatingResultSet:getString():16532748
ResultSet:getString():16532748
ForumDAO:getForums():3863358
Forum:setTitle():10671671 [LoopTagSupport:item:31627939]
Forum:title:10671671
Forum:getTitle():10671671 [LoopTagSupport:item:31627939]
Method:invoke():5528343
ComplexValue:evaluate():12480918
ArraySuffix:evaluate():7881279
ELEvaluator:evaluate():14927272
ELEvaluator:convertToExpectedType():14927272
Coercions:coerce()
Coercions:coerceToObject()
ELEvaluator:evaluate():14927272
Evaluator:evaluate():14854633
Evaluator:evaluate():14854633
ExpressionEvaluatorManager:evaluate()
ExpressionUtil:evalNotNull()
OutTag:evaluateExpressions():398106
OutTag:evaluateExpressions():30601567
OutSupport:value:398106
OutSupport:value:30601567
OutSupport:doStartTag():398106
OutSupport:doStartTag():30601567
Object:toString():19763856 [OutSupport:value:398106]
6705 RET
18190 FGT
18195 FGT
6716 FST
6721 FGT
6715 RET
18198 RET
6724 RET
6690 RET
6694 RET

Figure 2.7: jGossip Main Page Precomputation Analysis Results

the dataflow produces the graph shown in Figure 2.7. This graph is filtered to show the flow of a particular data item, the title of a forum displayed on the page.

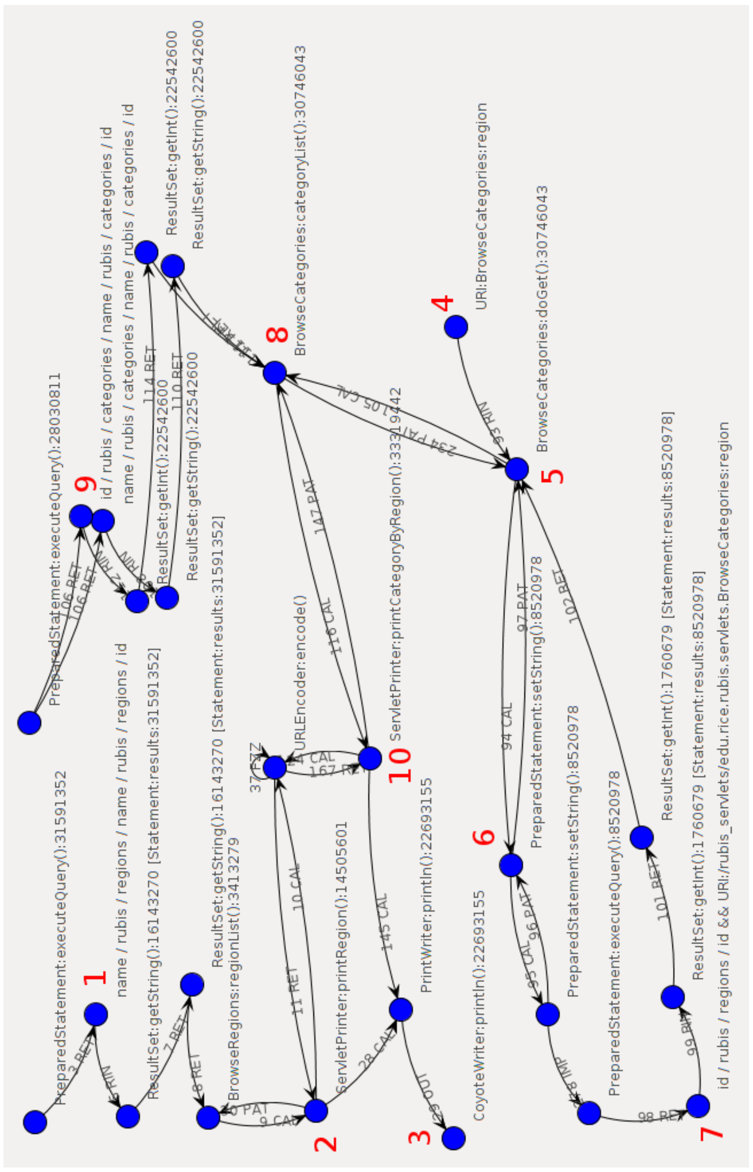
The analysis essentially indicates that the data is able to flow from the database all the way to output without any dependence on other inputs, in particular inputs with unpredictable values. Breaking this flow down, the data read from the database at point 1, and eventually stored in a field near point 2. This field is subsequently accessed by JSP generated code at point 2 (Method:invoke() indicates the use of reflection to get to the field), and the forum title passes through a series of internal methods from the JSTL tab library (the 'evaluate' nodes). It is ultimately used in the doStartTag() method at point 4, a method used by JSP tags to write data to the user response page.

Using this result, along with others which show similar results for other flows of data on the main page, one can find large portions of the main page which can simply be precomputed and then updated should the input database tables ever change. The precomputed results could be added to the JSP pages, avoiding the entire computation which reads them from the database and invokes the JSTL library.

2.12.5 Caching

RUBiS

Figure 2.8 shows a trace taken from viewing the same RUBiS categories list as previously shown with an additional constraint. For this trace, a user first submits a request to get a list of geographical regions. Once these are displayed, a user picks a region, which submits another request to obtain the category list. The region name selected in the first request is used to obtain a region id, which is included in the category hyperlinks output to the user in the second request. The important point to note is that this category list should not be outright precomputed, as the region id parameter varies based on user input. However, this region id is over a small enough range that it is reasonably predictable, and thus it seems that a cache could be implemented here. Breaking down Figure 2.8, point 1 shows the region name data being read from the database for the first request. Note that this trace has been filtered to hide string concatenation dataflow (removing StringBuilder nodes), make it more presentable. At point the region names



PreparedStatement:executeQuery():31591352
1
name / rubis / regions / name / rubis / regions / id
ResultSet:getString():16143270 [Statement:results:31591352]
ResultSet:getString():16143270 [Statement:results:31591352]
BrowseRegions:regionList():3413279
2
ServletPrinter:printRegion():14505601
URLEncoder:encode()
10
ServletPrinter:printCategoryByRegion():33319442
3
PrintWriter:println():22693155
CoyoteWriter:println():22693155
PreparedStatement:executeQuery():28030811
9
id / rubis / categories / name / rubis / categories / id
name / rubis / categories / name / rubis / categories / id
ResultSet:getInt():22542600
ResultSet:getString():22542600
ResultSet:getInt():22542600
ResultSet:getString():22542600
8
BrowseCategories:categoryList():30746043
4
URI:BrowseCategories:region
5
BrowseCategories:doGet():30746043
6
PreparedStatement:setString():8520978
PreparedStatement:setString():8520978
PreparedStatement:executeQuery():8520978
ResultSet:getInt():1760679 [Statement:results:8520978]
ResultSet:getInt():1760679 [Statement:results:8520978]
7
id / rubis / regions / id && URI:/rubis_servlets/edu.rice.rubis.servlets.BrowseCategories:region

Figure 2.8: RUBiS Browse Categories By Region Trace

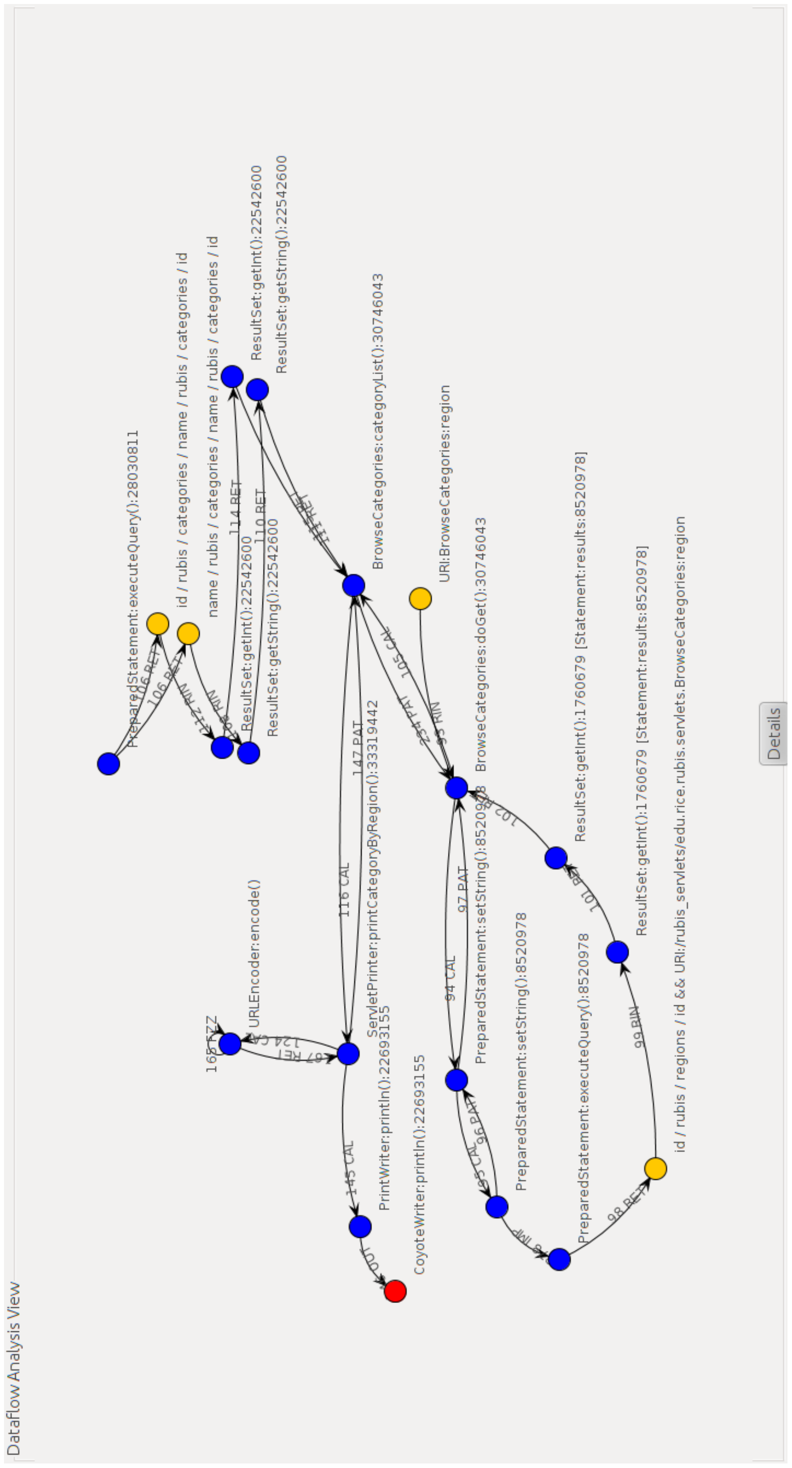

Dataflow Analysis View
Details
PreparedStatement:executeQuery():28030811
id / rubis / categories / name / rubis / categories / id
name / rubis / categories / name / rubis / categories / id
ResultSet:getInt():22542600
ResultSet:getString():22542600
BrowseCategories:categoryList():30746043
URI:BrowseCategories:region
BrowseCategories:doGet():30746043
ResultSet:getInt():1760679 [Statement:results:8520978]
ResultSet:getInt():1760679 [Statement:results:8520978]
id / rubis / regions / id && URI:/rubis_servlets/edu.rice.rubis.servlets.BrowseCategories:region
URLEncoder:encode()
ServletPrinter:printCategoryByRegion():33319442
PreparedStatement:setString():8520978
PreparedStatement:setString():8520978
PreparedStatement:executeQuery():8520978
PrintWriter:println():22693155
CoyoteWriter:println():22693155
106 RET
114 RET
110 RET
105 CAL
93 RIN
116 CAL
147 PAT
97 PAT
94 CAL
102
101
99 RIN
98 RET
96 PAT
95 CAL
124 CAL
167 RET
145 CAL
OUT

Figure 2.9: RUBiS Browse Categories By Region Caching Results

are passed to the encode method during concatenation with formatting text, and then output to the user at point 3. A user clicks on a region, submitting its name as a parameter in the next request. This is shown at point 4, as the 'region' parameter. From point 5 along to point 6 and continuing, the region name is set as a parameter on a PreparedStatement Object to be used as a predicate in a database query. When the query is executed at point 7, note how the input node is labeled as being tainted by not only the id column from the regions table, but also by the region parameter that the query was predicated on. Point 5 receives the region id, and passes it to point 8, which receives category names and ids from point 9. The category data and region id are sent point 10, the 'printCategoryByRegion' method, which concatenates them with formatting text, encodes them, and writes them out to the user at point 3.

Running the caching analysis over this graph and filtering the results to only show the dataflow for the second request, the result graph show in Figure 2.9 is obtained. The first important point to note is that this graph was not eligible for precomputation, due to the influence of the user-supplied region parameter in every part of this graph except for the upper-left portion showing the flow of the category name/id data. However, since this parameter has been marked as being predictable (see Figure Ce [datasource-infochart]), the analysis judges that a cache may be possible for the identified inputs. Interpreting this, a developer can start at the output node. Figure Cf shows the tainted data flowing here, and this can be compared to the response output breakdown shown in Figure Cg. Evidently the entire page it composed non-tainted (likely static) data and the cachable output. Looking at the inputs in the graph, the data responsible for the cachable output comes from database tables which have all been marked as stable, unlikely to change frequently, and the predictable region parameter.

Using the graph and some rudimentary knowledge of the application, a cache could be implemented as follows. First, consider the inputs. All of them, save for the region parameter, are stable, and thus the only parameter the cache need use to store and return results is that region parameter. For the other inputs, like those from the categories and regions tables, these data sources need only to be monitored to detect if they are modified. If so, the cached results are invalidated to be regenerated. As the doGet node is the first to receive the region parameter, the cache check could be performed there. If the cache hits for a given region, the output data (perhaps the entire response page or just the category hyperlinks) can be returned and written to the response stream. In the event of a cache miss, the computation can proceed as normal, and the results picked up in the printCategoryByRegion node to be placed in the cache for later.

jGossip

The caching analysis for jGossip is significantly more difficult due to the much larger traces produced by the application. This is caused by libraries, like JSTL, used by jGossip, and typically the traces were around 40K communication events for handling a single request (whereas RUBiS had around 400). This presented some very large, complex graphs, which would not be amenable to a manual analysis, and which were still challenging to assess automatically. Nevertheless, Figure 2.10 shows a graph obtained from the analysis tool which indicates the existence of a possible cache.

This was obtained by tracing the application while selecting a forum from the application's main menu, which then displayed a page giving some information about the forum including a message from it. The trace shows how the forum id (fid) supplied by the user when selecting a forum (point 1) acts as the sole input to a database query which fetches a message heading from the database (point 2). This heading is then passed through a series of methods, ultimately being output to the response page at point 3. The fact that this path was identified by the analysis means that the data which flows along the path is not influenced by random input. We can see how given a fid value, of which there are a limited set, and knowing whether or not the messages in that forum have been updated, we can either return a cached message heading or generate one to fill the cache.

Given this graph, it is up to the developer to investigate the feasibility of this cache. The analysis data can additionally provide stack traces to identify the control flow responsible for each communication event, to help in choosing where to implement any changes.

2.12.6 Postcomputation

RUBiS

For the postcomputation analysis, we sell an item on the RUBiS Store. This simply involves filling out a web form describing various details of the item

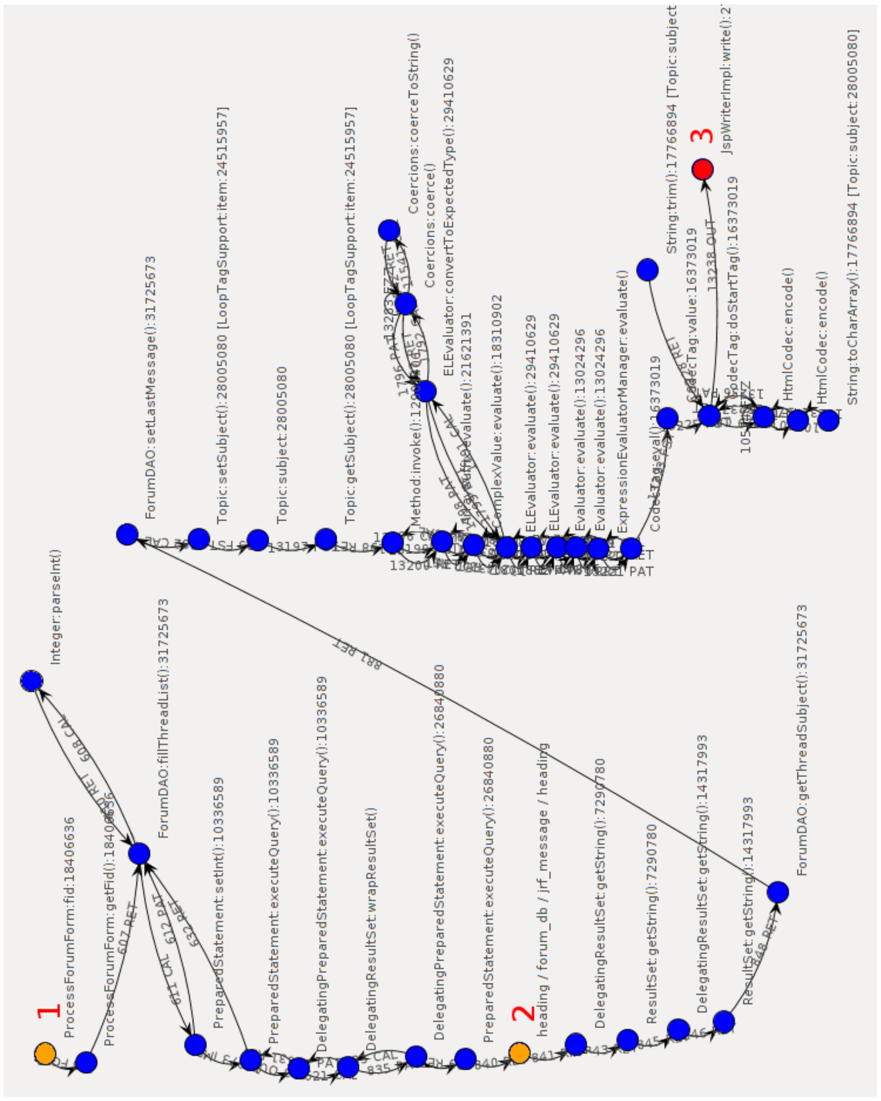
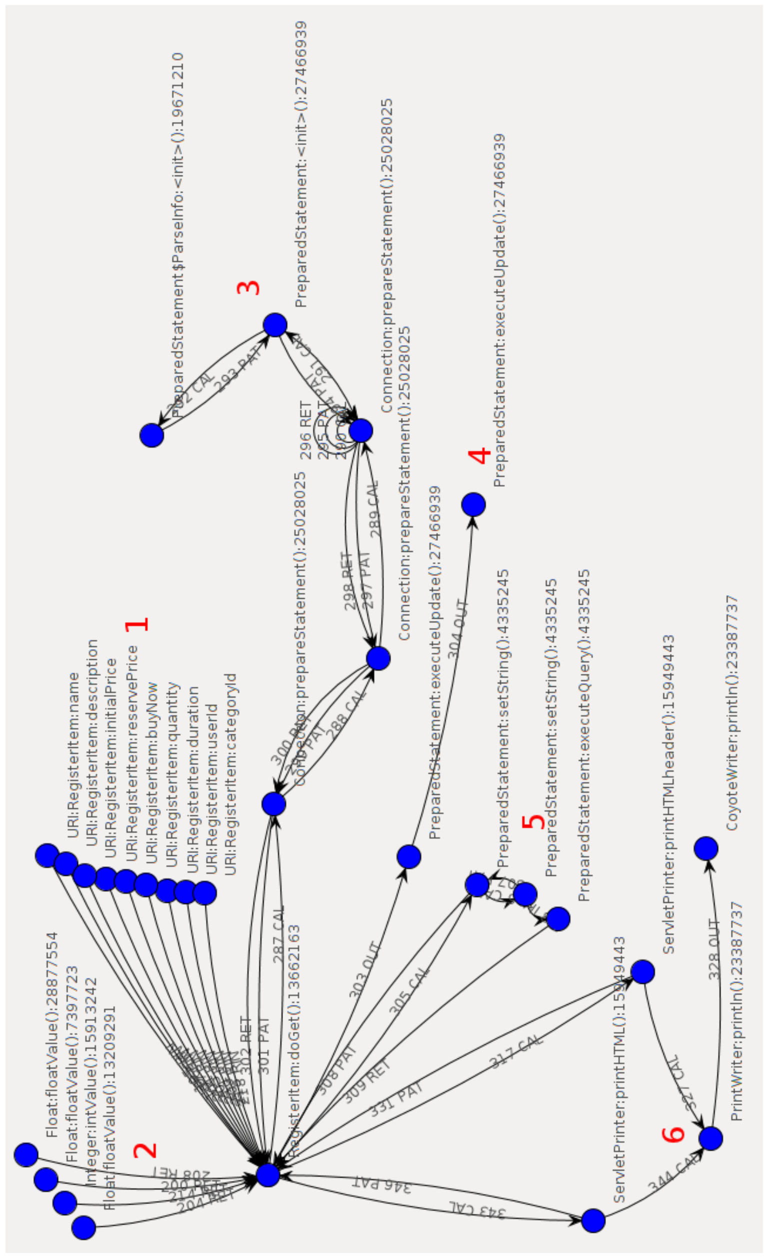

1
ProcessForumForm:fid:18406636
ProcessForumForm:getFid():18406636
Integer:parseInt()
607 RET
608 CAL
611 CAL
612 PAT
632 RET
ForumDAO:fillThreadList():31725673
PreparedStatement:setInt():10336589
PreparedStatement:executeQuery():10336589
DelegatingPreparedStatement:executeQuery():10336589
DelegatingResultSet:wrapResultSet()
DelegatingPreparedStatement:executeQuery():26840880
PreparedStatement:executeQuery():26840880
2
heading / forum_db / jrf_message / heading
DelegatingResultSet:getString():7290780
ResultSet:getString():7290780
DelegatingResultSet:getString():14317993
ResultSet:getString():14317993
848 RET
ForumDAO:getThreadSubject():31725673
881 RET
ForumDAO:setLastMessage():31725673
Topic:setSubject():28005080 [LoopTagSupport:item:24515957]
Topic:subject:28005080
Topic:getSubject():28005080 [LoopTagSupport:item:24515957]
Method:invoke()
ELEvaluator:evaluate():21621391
ComplexValue:evaluate():18310902
ELEvaluator:evaluate():29410629
ELEvaluator:evaluate():29410629
Evaluator:evaluate():13024296
Evaluator:evaluate():13024296
ExpressionEvaluatorManager:evaluate()
Coercions:coerceToString()
Coercions:coerce()
ELEvaluator:convertToExpectedType():29410629
CodecTag:eval():16373019
String:trim():17766894 [Topic:subject
CodecTag:value:16373019
13238 OUT
3
JspWriterImpl:write()
CodecTag:doStartTag():16373019
HtmlCodec:encode()
HtmlCodec:encode()
String:toCharArray():17766894 [Topic:subject:28005080]

Figure 2.10: jGossip View Forums Caching Results


Float:floatValue():28877554
Float:floatValue():7397723
Integer:intValue():15913242
Float:floatValue():13209291
2
208 RET
200 RET
214 RET
204 RET
URI:RegisterItem:name
URI:RegisterItem:description
URI:RegisterItem:initialPrice
URI:RegisterItem:reservePrice
URI:RegisterItem:buyNow
URI:RegisterItem:quantity
URI:RegisterItem:duration
URI:RegisterItem:userId
URI:RegisterItem:categoryId
1
RegisterItem:doGet():13662163
302 RET
301 PAT
287 CAL
Connection:prepareStatement():25028025
300 RET
299 PAT
288 CAL
Connection:prepareStatement():25028025
298 RET
297 PAT
289 CAL
Connection:prepareStatement():25028025
296 RET
295 PAT
PreparedStatement$ParseInfo:<init>():19671210
292 CAL
293 PAT
3
PreparedStatement:<init>():27466939
PreparedStatement:executeUpdate():27466939
4
PreparedStatement:executeUpdate():27466939
303 OUT
304 OUT
305 CAL
308 PAT
309 RET
PreparedStatement:setString():4335245
5
PreparedStatement:setString():4335245
PreparedStatement:executeQuery():4335245
317 CAL
331 PAT
ServletPrinter:printHTMLheader():15949443
ServletPrinter:printHTML():15949443
6
327 CAL
328 OUT
CoyoteWriter:println():23387737
PrintWriter:println():23387737
344 CAL
343 CAL
346 PAT
Figure 2.11: RUBiS Sell Item Trace

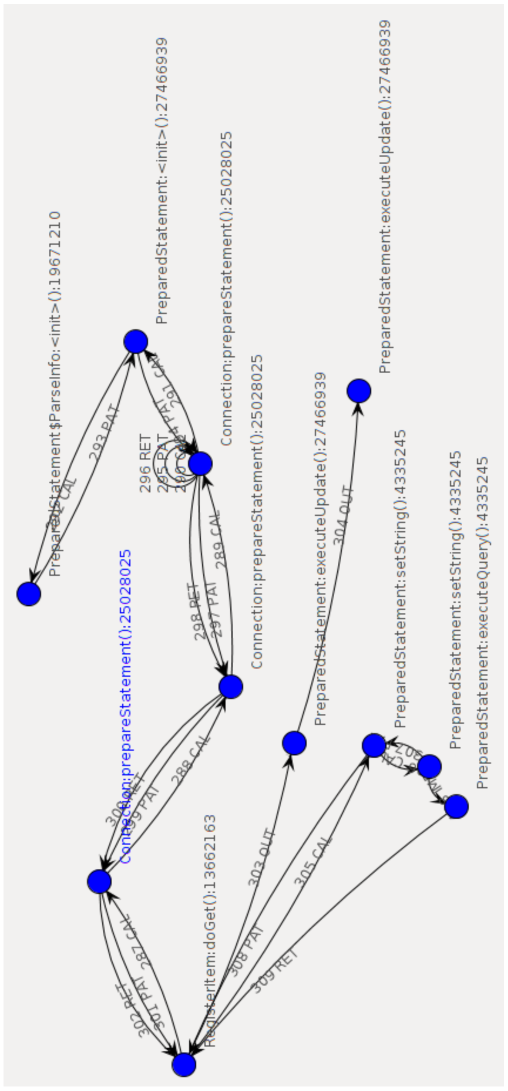

PreparedStatement$ParseInfo:<init>():19671210
PreparedStatement:<init>():27466939
Connection:prepareStatement():25028025
Connection:prepareStatement():25028025
Connection:prepareStatement():25028025
RegisterItem:doGet():13662163
PreparedStatement:executeUpdate():27466939
PreparedStatement:executeUpdate():27466939
PreparedStatement:setString():4335245
PreparedStatement:setString():4335245
PreparedStatement:executeQuery():4335245
287 CAL
288 CAL
289 CAL
291 CAL
292 CAL
293 PAT
295 PAT
296 RET
297 PAT
298 RET
299 PAT
300 RET
301 PAT
302 RET
303 OUT
304 OUT
305 CAL
308 PAT
309 RET

Figure 2.12: RUBiS Sell Item Postcomputation Analysis Results

(name, price, etc), and submitting this to a RegisterItem servlet. The item is added to the database and the item details are displayed back to the user.

The dataflow for this is given in Figure 2.11. At point 1 the various parameters describing the item are read, and some of them are converted to numeric values at point 2 (and hence tracked by the numeric tracking system). The RegisterItem:doGet() method then uses these parameters to construct a PreparedStatement at point 3, which is to execute a database update at point 4. This is the update which inserts the item into the database. Following this, some of the parameters are used at point 5 to query the database checking if the item was inserted. Finally, at point 6, the parameters are written back to the response page.

Figure 2.12 shows the results of the postcomputation analysis over this graph. What this indicates is parts of the dataflow which could be delayed until later, namely the database update/query flows at points 3, 4, and 5 from Figure 2.11. Missing from this graph is the reading of the item properties and writing them to the response page. This is because if these flows were deferred until later the output that the user sees would be incomplete. The analysis reveals that the database queries return no data which goes to the user response, and thus the analysis suggests that these queries could be done asynchronously, returning back to the user without waiting for their completion.

jGossip

A caching analysis for jGossip was significantly more difficult due to the greater complexity of the traces produced. This was largely due to the use of such external libraries as JSTL which jGossip relies on to format its output, producing traces of over 40K communication events for simple application actions vs around 300 for RUBiS. Nevertheless, the analysis tool was successfully used to capture the existence of a caching opportunity in jGossip.

To test the postcomputation analysis for jGossip, we use a trace taken while deleting a forum. We anticipated that this action would involve computation and updates to the database which could potentially be deferred, being separate from any computation responsible for displaying output to the user.

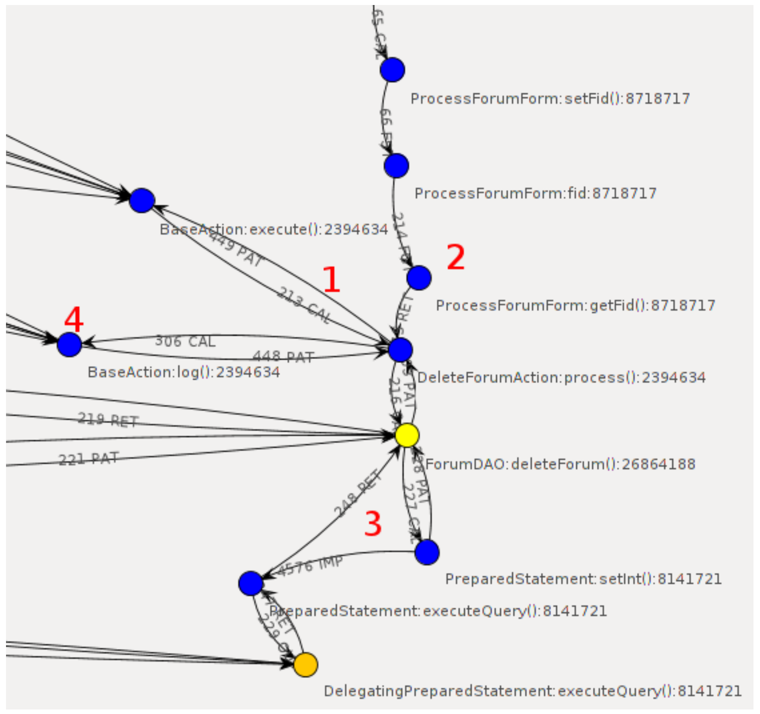

65 CAL
ProcessForumForm:setFid():8718717
66 F
ProcessForumForm:fid:8718717
214 F
2
ProcessForumForm:getFid():8718717
RET
BaseAction:execute():2394634
449 PAT
1
213 CAL
4
306 CAL
448 PAT
BaseAction:log():2394634
DeleteForumAction:process():2394634
216
PAT
219 RET
221 PAT
ForumDAO:deleteForum():26864188
248 RET
PAT
227 CAL
3
PreparedStatement:setInt():8141721
4576 IMP
PreparedStatement:executeQuery():8141721
RET
229
DelegatingPreparedStatement:executeQuery():8141721

Figure 2.13: jGossip Delete Forum Postcomputation Analysis Results

Figure 2.13 shows a partial view of the results of the analysis. At point 1 an object containing data to service the request, such as the ID of the forum to delete, is passed into an Action object (Actions basically provide the entry point functionality in an Apache struts application, like the servlets used in RUBiS). Point 2 shows the accessing of the forum ID from the data object. This ID is passed to the highlighted deleteForum() method, which in turn uses it in a query which deletes the forum from the database. Finally, at point 4 we see that not only can the process which deletes the forum be deferred, but also a logging action. The analysis is showing that the path of data presented in Figure 2.13 likely does not reach any output necessary to generate the user response page. At this point it is up to the developer to inspect this path and determine if they wish to make elements of it occur asynchronously to speed the response to the user.

2.12.7 Persistent State

RUBiS

Figure 2.14 shows the location of persistent state when interacting with the RUBiS chat box, as previously described in section 2.12.4. Only the portion of the graph which deals with the persistent state is shown here. The red node shows where the persistent state is actually kept, in a variable called 'chattyList' on the ServletPrinter class. The orange nodes indicate any nodes which communicated the persistent data in question. As expected, these orange nodes include a chatMessage parameter and its flow into the chattyList, as well as the output nodes which indicate the writing of the chat messages out to the user.

Other instances of persistent state existed in the application, in static variables like this one. All of these were known prior to the automated analysis, which was able to identify them easily.

jGossip

This analysis revealed several instances of persistent state in the jGossip application, mainly session attributes storing data read from the database for quick lookup, such as forum threads and messages. We do not present the results here, as the analysis is a basic one and explored sufficiently in the RUBiS example. Furthermore, the next analysis, user state, subsumes this one and deals with the jGossip application.

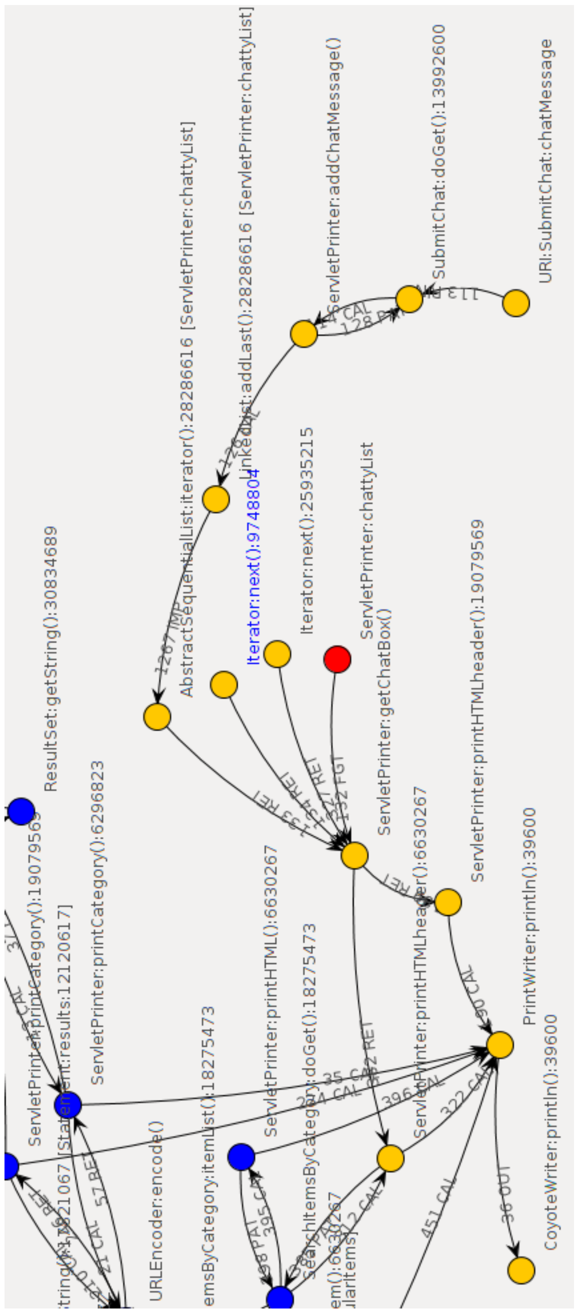

URI:SubmitChat:chatMessage
SubmitChat:doGet():13992600
ServletPrinter:addChatMessage()
LinkedList:addLast():28286616 [ServletPrinter:chattyList]
AbstractSequentialList:iterator():28286616 [ServletPrinter:chattyList]
Iterator:next():9748804
Iterator:next():25935215
ServletPrinter:chattyList
ServletPrinter:getChatBox()
ServletPrinter:printHTMLheader():19079569
ServletPrinter:printHTMLheader():6630267
ServletPrinter:printHTML():6630267
PrintWriter:println():39600
CoyoteWriter:println():39600
ResultSet:getString():30834689
ServletPrinter:printCategory():19079569
ServletPrinter:printCategory():6296823
Statement:results:12120617
URLEncoder:encode()
SearchItemsByCategory:doGet():18275473

Figure 2.14: RUBiS Browse Items State Analysis Results

2.12.8 User State

RUBiS

The RUBiS application did not contain any persistent state which was not shared between multiple users, mainly due to the fact that it had no notion of sessions.

jGossip

For this analysis we traced jGossip while logging in with two different users, each coming from a different IP address. This was done in order to generate persistent state data which potentially would be only shared with a single user. We ran the automated user state analysis, and identified the flow show in Figure 2.15. Though it is not readily apparent in the figure, the analysis tool additionally mapped the tainted data in the flow to user response output, revealing that this data is a user name used to populate a username field which appears on every page.

The fact that the analysis identified this flow indicates that this username data has been used over multiple requests, but is only ever shared with a single user. At point 1 in the figure we can see where the data is stored, in a Java Map object, which is the underlying implementation for the Session objects on which getAttribute is called as one follows the flow. The data proceeds through a series of evaluate methods, which are a result of using JSP and JSTL, before being output at point 2.

Knowing that this data is only shared with a single user, one could potentially remove this logic from the server-side code and allow the client to store their own username. The flow shown gives a developer some clues about what the implications of this would be, in that they may need to emulate some of the operations that the JSTL code performed in order to properly display the data to the user. This amounts to relocating the path shown in the figure from the server to the client, starting with moving the data (point 1), then the computation, and finally outputting it to the user (point 2).

2.12.9 Wasteful Communication

RUBiS

The RUBiS application did not exhibit significant wasteful communication, due to its simple design.

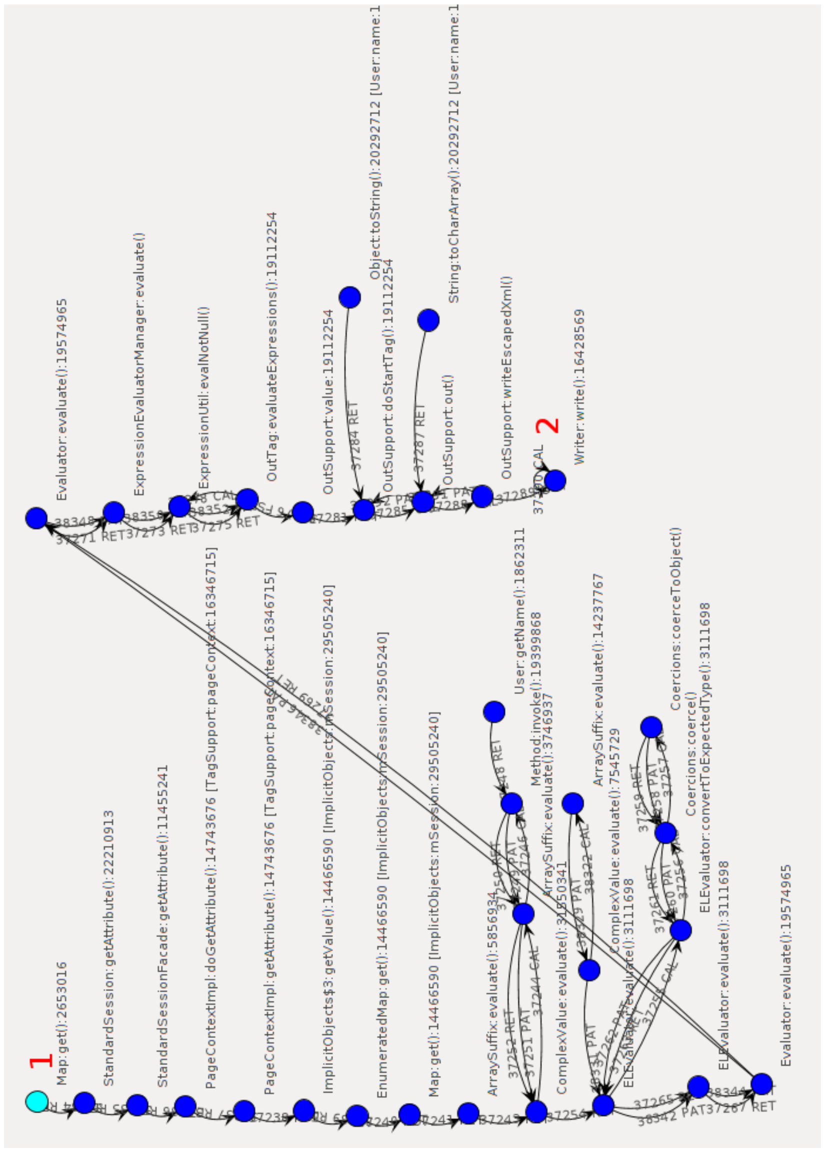

1
2
Map:get():2653016
StandardSession:getAttribute():22210913
StandardSessionFacade:getAttribute():11455241
PageContextImpl:doGetAttribute():14743676 [TagSupport:pageContext:16346715]
PageContextImpl:getAttribute():14743676 [TagSupport:pageContext:16346715]
ImplicitObjects$3:getValue():14466590 [ImplicitObjects:mSession:29505240]
EnumeratedMap:get():14466590 [ImplicitObjects:mSession:29505240]
Map:get():14466590 [ImplicitObjects:mSession:29505240]
ArraySuffix:evaluate():5856934
User:getName():1862311
Method:invoke():19399868
ArraySuffix:evaluate():3746937
ComplexValue:evaluate():32550341
ArraySuffix:evaluate():14237767
ComplexValue:evaluate():7545729
ELEvaluator:evaluate():3111698
Coercions:coerceToObject()
Coercions:coerce()
ELEvaluator:convertToExpectedType():3111698
ELEvaluator:evaluate():3111698
Evaluator:evaluate():19574965
Evaluator:evaluate():19574965
ExpressionEvaluatorManager:evaluate()
ExpressionUtil:evalNotNull()
OutTag:evaluateExpressions():19112254
OutSupport:value:19112254
Object:toString():20292712 [User:name:1
OutSupport:doStartTag():19112254
String:toCharArray():20292712 [User:name:1
OutSupport:out()
OutSupport:writeEscapedXml()
Writer:write():16428569

Figure 2.15: jGossip Login User State Analysis Results

jGossip

This analysis was a very simple one, and serves to hint at the potential for this technique. Figure 2.16 shows views of the same trace. On the left is shown every instance where tainted data is communicated when the users requests to view a forum (the same trace used in the caching analysis). On the right we have removed every edge which carries tainted data which is never actually used, resulting in a significantly smaller graph (5921 of the original 14693 edges remained).

In jGossip this is largely due to the use of JSP, which passes around objects containing references to all the data needed to render a page, whether or not they are needed in a particular function. The figure serves to show that such an occurence can be quite prevalent in certain applications, and that our analysis tool is capable of detecting it.

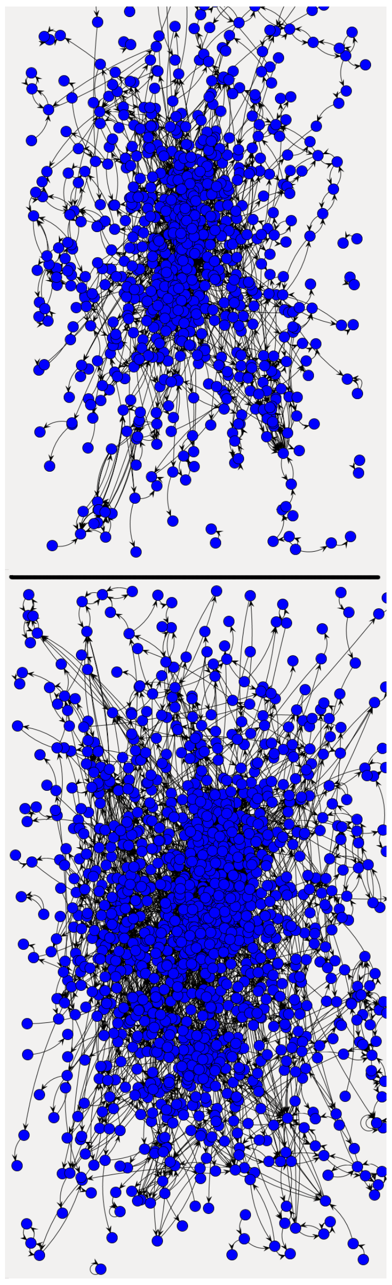


Figure 2.16: jGossip Wasteful Communication Analysis Results

Chapter 3

Conclusions

3.1 Discussion of Results

Having run most of the analyses over both applications, identifying the analysis targets from the taint trace data using the automated tool, the experiments were largely successful. In particular, the results for the precomputation analysis on both applications were impressive. In both cases the analysis was able to proceed with minimal user intervention to produce the graphs presented here. All that was needed was the initial input source descriptions files, as well as some manual filtering out of nodes for some of the jGossip graphs for presentation purposes.

The caching analysis, especially on the RUBiS application, was able to not only identify the existence of cachable output, but also show in a clean manner all the inputs to consider for the cache. For RUBiS, one can easily see how the various inputs from both the user and database come together to determine the result, and thus a developer should be able easily use this in implementing a cache for the data in question. It is useful to see the path of all the relevant data, quickly indicating the computations responsible for transforming it to what the user sees. Additionally, the small graph sizes for RUBiS means that the analysis results are easy for one to grasp immediately once presented by the tool.

Also impressive was the user state analysis for jGossip. Looking for dataflow paths present in multiple requests but not shared with more than one user produced some very clean graphs, like the one shown. It was easy to identify the location of the persistent data, and then subsequently the sequence of communications and computations needed to take the data to its destination. However, not all of the graphs produced by the analyses were so clean.

In the case of jGossip, the much larger graph sizes meant that more intervention was needed to obtain the presented results. The caching analysis for jGossip identified tainted dataflow graphs which were cachable, only involving the flow of predictable data. However, they were too large to immediately grasp, containing large groups of nodes from calls into frameworks which ultimately distracted from the dataflows which would most likely be interesting to a developer. The graph shown was obtained by inspecting some of the jGossip source to understand how the cachable data was likely flowing, and then searching in the analysis results for the occurence of this. Following that, nodes which did not contribute to the interesting dataflow were manually filtered out.

Heavier manual intervention was also needed for the jGossip post computation analysis, where again some knowledge of the code and what we expected the analysis to actually find helped to locate the results in the large output graphs. For post computation, there were many nodes in jGossip which carried tainted data which never made it to the user, and as such were identified by the analysis. Since we wanted to focus on just the nodes which contributed to a deferable database write, some filtering was needed to present the graph.

This brings us to the point of the difficulty of automating these kinds of analyses. Our results with the RUBiS application were very optimistic, as the graphs produced by the analysis tool were very easy to understand immediately in large part thanks to their small size. Additionally, since RUBiS uses no added libraries, every node in the graph is potentially relevant to the developer and amenable to change and optimization under the guidance of the analysis.

For a framework-heavy application like jGossip, and realistically many Java web applications make extensive use of large frameworks, the graphs are much larger. At their current stage, most of our analyses output results which are negatively affected by this complexity. They suffer from a "rats nest" effect, and require some manual filtering and searching through to find the parts which could be used to actually support a developer in optimizing the applicaiton. The results are still there, they are just surrounded in many nodes which are less relevant, and thus it helps at this point to actually know what you are looking for. Even knowing simply which piece of data may be involved in an optimizating can go a long way. For the jGossip caching analysis the output was greatly simplified by using the tool to focus on nodes which carried data which we suspected would be cachable.

A related problem, again most prevalent in the large jGossip graphs, is that of false positives. The analyses are written to be fairly general, so as to find their results in applications which were not written to be easily analyzed. Every pattern of dataflow cannot be reasonably anticipated, so the analyses were written to be permissive and present many possible results. From these one can find those which are truly useful. For example, in the caching analysis for jGossip it was possible for some some output to be affected by random data through control dependence, which our taint tracking technique does not consider.

A final problem, unrelated to the analysis results, is that of the speed of applications during taint tracking. Slowdowns of roughly two orders of magnitude were observed, especially in the more complex jGossip application. This is mainly because of the use of AspectJ, which introduced inefficiencies inherent in its own design as well as forced us to perform some wasteful workarounds to properly track tainted data. This reality, along with the fact that the log files could be very massive for even small amounts of application activity (10s of thousands of lines per request in jGossip), meant that we generally did not perform analyses over exhaustive taint traces. By this we mean that we did not attempt to work the application through all or even a majority of its possible execution paths. There are issues of coverage here, with the possibility of some analysis being invalidated by additional data. For example, exploring more requests for the user state analysis may have revealed that the persistent data identified was actually shared among multiple users in some cases.

Even with these shortcomings, the project was still successful given the aims of the thesis. The analyses were able to identify their targets using only the taint tracking data and in some cases light user input. Such input could additionally be automated, such as by monitoring an application's use of data to categorize the variability of its inputs as in the input description files. As the analyses found valid results, it shows both that given optimization opportunities it is possible for a taint tracking based automated analysis to target them; and also shows that the types of optimizations we sought to locate with our analyses actually exist in real web applications. Furthermore, though manual intervention is currently required to produce some of the results, the automation that was observed was promising. Our analyses are relatively simple, being composed from scratch based on rough descriptions from literature, and still they were in many cases able to produce very clean results, especially for the simpler RUBiS application. In such cases, an inexperienced developer should be able to make use of even this early stage version of our tools.

Finally, one of the side goals of this thesis was to find uses for the taint tracing data that we did not initially anticipate when planning our set of analyses to perform. The wasteful communication analysis was almost discovered by accident when seeking a way to reduce the complexity of graphs for other analyses. Having studied application partitioning systems in the early stages of the project, we later recognized the potential of knowing whether communicated data was actually used for analyses in that space and others. This was particularly exciting, as the data very easily supported an analysis that we had not originally intended, and thus had not designed for.

3.2 Future Work

Given the analysis results and our experiences working with the tools, there are a variety of improvements to be made and interesting directions to explore. These range from simple refinements to the taint tracking and analysis tools to new research directions.

Beginning with the most basic needs, the tools need to be made more efficient. The speed of the tracker has to be improved significantly, and here we suggest looking to recent taint tracking research which has focused on ways to speed up the technique. Even just moving away from AspectJ to a handwritten, byte-code level taint tracking tool would provide large speedups. For the analysis tool, the problem is more on the memory side. When dealing with very large taint traces like those obtained from jGossip, the graphs are so large that they may not fit in memory. Addressing this could mean finding a way to compress the traces, possibly by ignoring/summarizing communication events which are not particularly valuable to analysis (framework activity is a good candidate for such measures). One could also modify the analyses to work efficiently with disk resident data, or to be more clever about how they use available memory.

Beyond efficiency, moving away from AspectJ for the taint tracker could allow a more complete tracking. Under AspectJ there was no perfect solution for tracking the flow of primitive data types, and what we have in place for tracking numeric values is really a temporary hack. Implementing the tracker at a lower level, such as Java byte-code, would provide the control necessary to tag and track such values. One could also explore the effects of control-flow propagation, a more permissive form on taint flow, on the analyses presented here. As has been discussed in the literature, this may lead to many false positives, but there have been efforts to control the explosion of tainted data in such scenarios which may offer useful lessons. Tracking in this way may capture dependencies between data items that we miss, and allow for more accurate analyses.

Concerning the actual interaction with the analysis tool, the needs are mostly down to automation. If the tool was to be truly used to support developer understanding of applications, it needs to be able to offer cleaner results without expert intervention. Additional study is needed to refine the analyses to more accurately identify their targets and find ways of more concisely presenting their results. Taking things much further, it would be interesting to attempt to perform some optimizations automatically, as is done in the Fluxo system. There are many challenges here, as the applications targetted by our system were not developed in restricted programming models which could make automatic changes easier. Following improvements to the analysis tool, it would be valuable to conduct a user study to determine if non-expert developers can actually use the tool to make optimization decisions about web applications they are unfamiliar with.

Related to improving the analysis tool, the analyses themselves can be developed further. In the future we would like to do a more thorough evaluation of specific analyses, taking the results from the tool and using them to modify applications. Modified applications could be evaluated under typical use scenarious to assess the quality of the optimization suggestions. Thoroughly testing the possible execution paths over a wide variety of applications could enable more accurate analyses as we discover what kinds of dataflow patterns are common and exploitable for optimization. Further study of applications and the literature would also likely reveal additional analyses supported by the taint tracking data, as our set was not intended to be an exhaustive one.

Finally, once the tools are mature enough, we would like to attempt to integrate their operation into other analysis tools to support them. In particular, we would start by augmenting an application partitioner by supplying it with more data upon which to make better partitioning decisions. The wasteful communication analysis would be of great use here in obtaining better intermodule communication estimates. The work of combining our tools with a real application partitioner has actually already begun.

3.3 Final Words

We have demonstrated a proof of concept taint tracker for Java web applications, and a series of analyses to consume the data and identify useful properties. The goal was to show that given taint tracking data, it could be used to naturally support a wide variety of analyses. Futhermore, the results of these analyses would be geared to supporting developers in modifying applications to better deal with migration scenarios. By choosing analyses identified in the literature as useful to this end, we feel that we have made a sound first effort. As all of the analyses were generally successful, finding correct results in real applications, we see promise in this technique. It is no trivial task to take an application written in a non-restricted environment and attempt to automatically optimize it, as there are so many unexpected patterns which can arise.

The success we had was a result of focusing on getting a complete picture of application dataflow, for which taint tracking was necessary. The primary function of most web applications is the processing and serving of data for their users. Data is the most natural thing to follow when seeking to understand and improve upon the functioning of these applications. Because of the nature of the data we were collecting, and analyses were conceptually quite simple, while still providing useful results.

Though there are many points of refinement to be made in our tools and additional testing to be done with real applications, we feel that the work here is a significant first step, and one which should be taken further. By obtaining a more comprehensive taint tracking, analyses with will user intervention required, and even automatic application of optimizations, a powerful help for developers facing migration scenarios can be realized.

Bibliography

[1] MI Al-Saleh and JR Crandall. On information flow for intrusion detection: What if accurate full-system dynamic information flow tracking was possible? *Proceedings of the 2010 workshop on New ...*, 2010.

[2] MI Al-Saleh and JR Crandall. Tracking Address and Control Dependencies for Full-System Information Flow Measurement. *cs.unm.edu*, 2010.

[3] Alexandru Caracas, Andreas Kind, Dieter Gantenbein, Stefan Fussenegger, and Dimitrios Dechouniotis. Mining semantic relations using NetFlow. *2008 3rd IEEE/IFIP International Workshop on Business-driven IT Management*, pages 110–111, April 2008.

[4] Jim Challenger. A scalable system for consistently caching dynamic web data. *INFOCOM'99. Eighteenth ...*, 1999.

[5] Jim Challenger, Paul Dantzig, Arun Iyengar, and Karen Witting. A fragment-based approach for efficiently creating dynamic web content. *ACM Transactions on Internet Technology*, 5(2):359–389, May 2005.

[6] Jim Challenger, Arun Iyengar, Paul Dantzig, Daniel Dias, and Nathaniel Mills. Engineering highly accessed Web sites for performance. *Web Engineering*, pages 247–265, 2001.

[7] JR Challenger and Paul Dantzig. Efficiently serving dynamic data at highly accessed web sites. *Networking, IEEE/ ...*, 12(2):233–246, 2004.

[8] Erika Chin and David Wagner. Efficient character-level taint tracking for Java. *Proceedings of the 2009 ACM workshop on Secure web services - SWS '09*, page 3, 2009.

[9] BG Chun, S Ihm, and P Maniatis. Clonecloud: Elastic execution between mobile device and cloud. *Proceedings of the sixth ...*, 2011.

[10] Byung-Gon Chun and Petros Maniatis. Dynamically partitioning applications between weak devices and clouds. *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services Social Networks and Beyond - MCS '10*, pages 1–5, 2010.

[11] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. *...SIGARCH Computer Architecture ...*, 2007.

[12] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Tainting is not pointless. *ACM SIGOPS Operating Systems Review*, 44(2):88, April 2010.

[13] Louis Degenaro, Arun Iyengar, Ilya Lipkind, and Isabelle Rouvellou. A middleware system which intelligently caches query results. *Middleware 2000*, pages 24–44, 2000.

[14] Rajiv Gupta, Neelam Gupta, Xiangyu Zhang, Dennis Jeffrey, Vijay Nagarajan, Sriraman Tallam, and Chen Tian. Scalable dynamic information flow tracking and its applications. *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–5, April 2008.

[15] Mohammad Hajjat, Xin Sun, YWE Sung, and David Maltz. Cloudward bound: planning for beneficial migration of enterprise applications to the cloud. *ACM SIGCOMM ...*, pages 243–254, 2010.

[16] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 303–311, 2005.

[17] VPKGP Kangkook and JAD Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. pages 1–12, 2012.

[18] E Kiciman and Benjamin Livshits. Fluxo: a system for internet service programming by non-expert developers. *Proceedings of the 1st ...*, 2010.

[19] G Kiczales, J Lamping, and A Mendhekar. Aspect-oriented programming. *...-Oriented Programming*, (July), 1997.

[20] Geoffrey Lefebvre, Brendan Cully, Christopher Head, Mark Spear, Norm Hutchinson, Mike Feeley, and Andrew Warfield. Execution mining. *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments - VEE '12*, page 145, 2012.

[21] Shashidhar Mysore, Bita Mazloom, Banit Agrawal, and Timothy Sherwood. Understanding and visualizing full systems with data flow tomography. *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems - ASPLOS XIII*, page 211, 2008.

[22] K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic. Adaptive offloading inference for delivering applications in pervasive computing environments. *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, 2003. (PerCom 2003).*, pages 107–114, 2003.

[23] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[24] Shumao Ou, Kun Yang, and Jie Zhang. An effective offloading middleware for pervasive services on mobile devices. *Pervasive and Mobile Computing*, 3(4):362–385, August 2007.

[25] Alexander Rasmussen, E Kiciman, Benjamin Livshits, and M Musuvathi. Short Paper: Improving the Responsiveness of Internet Services with Automatic Cache Placement. 2009.

[26] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *ACM SIGARCH Computer Architecture News*, 32(5):85, December 2004.

[27] Angeliki Zavou, Georgios Portokalidis, and A Keromytis. Taint-exchange: a generic system for cross-process and cross-host taint tracking. *Advances in Information and ...*, 2011.

Appendix A

Terms Used

- Taint: Where this is used, such as when saying that a piece of data in an application is 'tainted', it refers to data which has been tagged to be tracked by the tracking system. Whenever tainted data is detected at some monitoring point in an application, the system reports it. Taint can include information about where the data came from, as is the case for the system described here, refereed to as the 'taint source'. In this thesis is made a minor distinction between Objects which are directly tainted and those which are reachability tainted. Directly tainted Objects include a set of basic types which, once tainted, are always tainted. These include Strings, StringBuffers, StringBuilders, character arrays, and some numeric values. Reachability tainted means that a directly tainted Object is reachable through an Object's reference graph, such as by following a heirarchy of field references.

- Tomography: This refers to the nature of the taint tracking in question. Tomography-level taint tracking is a heavy form of the analysis where tainted data is not only tagged at a source and identified at predetermined monitoring points, but is rather tracked along its complete path, indiscriminantly through many monitoring points. The basic goal of tomography is to get a complete trace of where tainted data goes in a program.

- Trace:

- Flow:

Appendix B

Taint Tracking Tool Logging Format

THIS NEEDS TO BE FORMATTED, DIFFICULT TO READ

Provided here is a breakdown of the log structure that the tool uses, for reference when describing other parts of the implementation and for gauging the completeness of the tracking.

- location Tag
  Format: <location callerClass, callerMethod, calledClass, calledMethod, requestCounter, requestURI, requestRemoteAddr, callerContextCounter, calledContextCounter />
  This tag appears in every log entry, and contains all information needed to determine where the event occured. The attributes of this tag are as follows:
  caller/called Class/Method - these specifies the fully qualified class name and method name with argument types of the caller and callee for a particular taint flow event. For example, for a record indicating that taint was passed in the arguments of a method call, these will indicate the method from which the call was made and the called method itself. requestCounter - this is a counter which uniquely identifies which user request a log was generated from. Everytime the application server must service a new request, a new counter value is assigned to it.
  requestURI - the URI of the request responsible for generating a log.
  requestRemoteAddr - the remote address (IP address of a web application user submitting a request) of the request responsible for generating a log.
  caller/called ContextCounter - these attributes supplement the caller/called Class/Method by identifying unique instances of method calls. Every time a method call starts and the call stack changes, a new counter value is assigned to the item on top of the stack. The counters for the

current caller and called methods are used when creating log records, and are useful when analysing taint flow graphs to properly determine which flows actually connect to each other.

- Xobject Tag
  Format: <Xobject type, objectID, value, taintID, taintRecord />
  Various 'object' tags (sourceObject, taintedObject, etc) appear in log records to describe important Objects in taint flow events. The attributes of this tag are as follows:
  type - the Java class name for this object
  objectID - a unique ID for the Object. We use the hashcode given by Java.
  value - if available, a String representation of the Object.
  taintID - an unique identifier for the taint an Object carries. If taint is propagated from one Object to another, they will have similar taintIDs (differing only by an identifier for the Object). If an Object is tainted by multiple sources, the taintID will reflect this.
  taintRecord - See below.

- taintRecord Attribute
  Format: TARGETCOLUMN: column CATALOG: catalog TABLE: table COLUMN: column...
  URI: uri PARAM: param
  This attribute, found in Xobject tags for Objects carrying taint, describes where the taint originally came from. In our implementation we taint reads from databases and from user web request parameters. For database reads we log the catalog, table, and column of every column involved in the selection query, and additionally specify the target column from which the data was actually taken. For request parameters we log the URI which received the request as well as the parameter name.

- field Tag
  Format: <field targetClass, targetField, static>
  These tags are used in field set/get records to identify, by class and field name, the field in question. Also indicated is whether or not this is a field of a static class.

- Propagation Log
  Format: <taintlog type="PROPAGATION"><location /><sourceObject /><destObject /></taintlog>

Indicates that taint has spread from one Object to another. These logs mostly occur as a result of String operations, such as appending and copying, which produce new Strings from tainted ones. The sourceObject tag describes the Object which was originally tainted, and the destObject tag the one receiving taint from the sourceObject.

- Fuzzy Propagation Log
  Format: <taintlog type="FUZZY"><location /><sourceObject /><destObject /></taintlog>
  Indicates that an Object was marked as tainted heuristically by fuzzy propagation, as described earlier. The sourceObject tag describes the Object which was originally tainted and which satisfied the levenshtein distance match with the non-tainted destObject. The destObject now carries the taint of the sourceObject.

- Modification Log
  Format: <taintlog type="MODIFICATION"><location /><targetObject /></taintlog>
  Indicates that some tainted Object has been modified. This usually means that a StringBuilder/Buffer Object was changed somehow, such as by an append. The targetObject tag describes this Object.

- Composition Log
  Format: <taintlog type="COMPOSITION"><location tag><composedObject><composingOb
  />...</composedObject></taintlog>
  Indicates that multiple tainted Objects have been combined into one Object which carries the combined taint. A composedObject tag describes the new Object, and nested inside it a series of composingObject tags describe the tainted Objects which were combined.

- Association Log
  Format: <taintlog type="ASSOCIATION"><location tag><associatingObjects><associatingO
  />...</associatedObjects></taintlog>
  Indicates that multiple tainted Objects have associated through certain method calls but were not combined into a single Object (such as comparing one String to another). An associatingObjects tag wraps a series of associatingObject tags describing the tainted Objects which associated.

- Calling Log
  Format: <taintlog type="CALLING"><location tag><taintedObject><subTaintedObject
  />...</taintedObject><callerObject /><calledObject /></taintlog>

Indicates that tainted Objects were passed in the arguments of a method call. For each argument which carries taint, a taintedObject tag describes the argument. If an argument is not directly tainted but from which taint is reachable, the taintedObject tag wraps a series of subTaintedObjects describing reachable directly tainted Objects. The caller/calledObject tags describe the Objects taking place in the method call.

- Returning Log
  Format: <taintlog type="RETURNING"><location tag><taintedObject><subTaintedObject />...</taintedObject><callerObject /><calledObject /></taintlog>
  This is similar to the Calling Log, but indicates that a tainted Object has been returned from a method call.

- Returning Args Log
  Format: <taintlog type="RETURNINGARGS"><location tag><taintedObject><subTaintedO />...</taintedObject><callerObject /><calledObject /></taintlog>
  This is similar to the Calling Log, but comes when the method is returning and indicates that arguments have taken on additional taint as a result of the method execution.

- Returning Input Log
  Format: <taintlog type="RETURNINGINPUT"><location tag><taintedObject><subTainted />...</taintedObject><callerObject /><calledObject /></taintlog>
  This is similar to the Returning Log, but is used for special cases to indicate that a return value is an original source of tainted data, such as a read operation from a database Object.

- Output Log
  Format: <taintlog type="OUTPUT"><location tag><taintedObject><subTaintedObject />...</taintedObject><callerObject /><calledObject /></taintlog>
  This is similar to the Calling Log, but is used for special cases to indicate that the arguments will be output to some important location, such as back to the user as web page text, or into a database.

- Non-Taint Output Log
  Format: <taintlog type="NONTAINTOUTPUT"><location tag><outputObject /><callerObject /><calledObject /></taintlog>
  This is similar to the Output Log, and is used in the same cases as it where the output data is not tainted. This allows one to see how tainted and non-tainted data mix in the output of the application.

- Field Set Log
  Format: <taintlog type="FIELDSET"><location tag><field /><tainte-dObject><subTaintedObject />...</taintedObject><callerObject /><calle-dObject /></taintlog>
  This is similar to the Calling Log, but indicates that a tainted Object has been assigned to an Object or static class field.

- Field Get Log
  Format: <taintlog type="FIELDGET"><location tag><field /><tainte-dObject><subTaintedObject />...</taintedObject><callerObject /><calle-dObject /></taintlog>
  This is similar to the Returning Log, but indicates that a tainted Object has been read from an Object or static class field.

Appendix C

Second Appendix

Here is the second appendix.