TaTAMi

Taint Tracking for Application Migration

by

Lee Alexander Beckman

B.Sc., The University of British Columbia, 2010

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)

April 2013

Abstract

In this thesis we addressed the question of whether taint tracking could be used to help developers make better, faster decisions when improving existing web applications. This is in the context of migration scenarios, where one must respond to increasing demands on an application by optimizing and generally rearchitecting. 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 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 for analysis in order to determine two important things. First, that we could write our analyses to automatically identify their targets and produce comprehensible results. Second, that the targets actually existed in real applications.

In the end our analyses were successful, in many cases producing clean results which concisely described non-trivial properties of the applications and possible optimizations 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

I'd like to thank Eric Wohlstadter of the Software Practices Lab and Rodger Lea of the MAGIC Lab, my supervisory team. Together they kept my work on track and always moving towards completion. They were exceedingly patient and understanding in this effort, as this ended up taking quite a bit longer than I'd planned, and there were many rough edges that they helped me round out. I thank them in particular for constantly questioning my assumptions, ideas, and things I tried to be vague about.

Nima Kaviani, also of the SPL, inspired me with his work on application partitioning and dedication to his own thesis. I thank him for his words of encouragement and efforts to understand what I was doing.

Finally, thanks to Ed Knorr, my second reader in a field somewhat removed from this work. His acceptance of the task was greatly appreciated.

Dedication

I dedicate this thesis to my wife, Esther, if she'll have it. She's waited too long for me to finish this, and gracefully provided her proofreading expertise on top of all the support, food, and distractions.

Chapter 1

Introduction

Maintaining any sufficiently large application is a difficult task for programmers, especially for those who did not originally 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 many popular web applications such as LinkedIn and MySpace [19].

The problem is straightforward; a team of developers will implement a system to meet the needs of their current users. Let us take the RUBiS online auctioning site (much like a simple version of eBay) as an example since we will experiment with this system later in the paper. As RUBiS becomes more popular and users want more features (shopping carts, faster loading times), the demand on the application increases. Eventually the original design may be insufficient to serve the needs of its users, possibly necessitating a form of migration. 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 good decisions about how to keep RUBiS up to date is a frustrating and time-consuming process, wrought with many problems:

- Time is often short, so systems are needed to quickly provide correct insight into how the application works, highlighting the key processes and flows of data behind the functionality the application provides.
- Changing application code without being aware of all of its side effects can break it in unexpected ways. Knowledge of how the operation of one component affects others down the line is needed.
- Real applications are often written poorly, and may not conform to expected patterns, so analysis tactics need to be general and robust.

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 [2] [19] [22] 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 the application works. This can be done statically by looking at the code, or dynamically by monitoring the runtime operation of the program.
- Use this data to build an intermediate 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 to find ways to optimize it.
- Map these properties and optimizations back to the original application, helping developers to better understand and improve it.

We believe that our particular approach to the above model, which uses dynamic information flow tracking (DIFT) [26] to build the intermediate representation, addresses the aforementioned problems well; saving users time, being well-adjusted to tracking side-effects, and potentially being able to support a wide variety of analyses on unconstrained applications.

1.1 Origins of Research

The TaTAMi project is a system which uses DIFT to 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. This allows one to see what parts of the system interacts with certain data 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 called dataflow tomography [22] could extract data which would support many analysis cases.

The analyses in question were inspired by works focusing on 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 [19]. Fluxo provides a series of automatic analyses which could be performed over dataflow graph representations of programs. These include discovering caches, finding opportunities to delay computation to service user requests earlier, 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

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

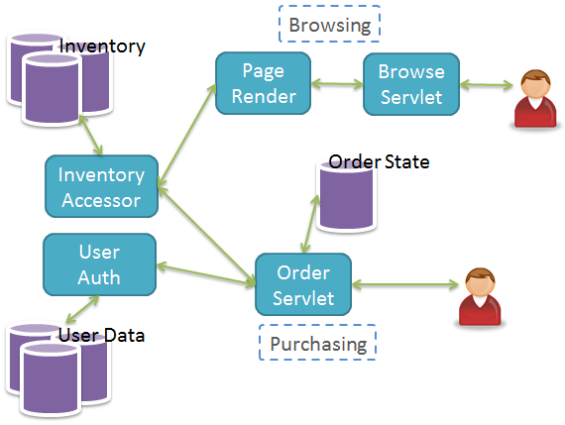

Figure 1.1: Example Web Store Architecture.

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, as well as 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 system; 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.

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. [1]
- The network link between the back and front-end components is carrying large amounts of data, and its maximum bandwidth may soon be reached.

To start, developers should have an overall picture of how the system works. 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 at a very high level, the arrows indicating how data flows from component to component. The points to note are:

- the flow of shop item data from the Inventory database (point 1) through the Browsing system (point 2) to the user (point 3)
- the flow of data from the back-end stores (point 4) and user input (point 5) into the 'Order State' shopping cart data (point 6)

[1] Such is the case with the Google App Engine Platform. Stateful applications are restricted to support transparent load balancing and virtualization, and state which remains after requests have been served is easily lost when an application is 'cycled out'.

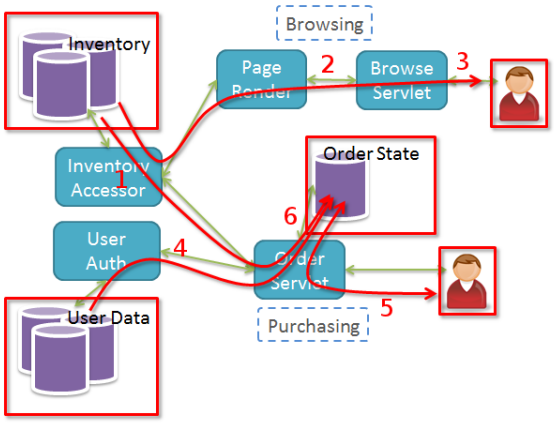

Figure 1.2: Sample Dataflow for Web Store.

This view focuses immediately on how the application 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?'. Knowing such is valuable as it quickly targets and organizes the key mechanisms by which many web applications operate. The majority of web applications are engines which take data as input, run it through various computations, and ultimately write the results to a web page for a user to consume. Being able to take a particular piece of output and trace it back through the steps which produced it is helpful. It gives a focused summary of how the application works for that piece of data. A developer can manually consume this information to gain understanding, but we can go beyond this to support automated analyses.

The analyses which follow were mainly arrived at by looking at the literature and identifying the various kinds of optimizations sought after in these works. We were also influenced by application partitioning research, which is valuable in the context of moving applications to different deployments. The state identification and cost analysis problems which follow are of particular interest in this space.

1.2.2.1 Precomputation and Caching

The first optimizations to consider, given that the Browsing component is backed up with computation, are precomputation and caching. Consider

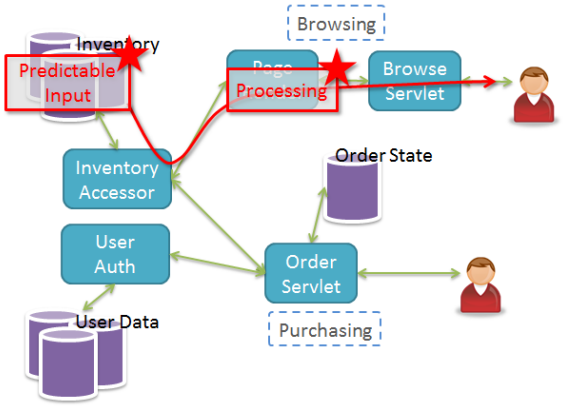

Figure 1.3: Dataflow Indicative of Possible Caching Opportunity.

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 the user's browse requests are entirely from predictable sources. By this we mean that the input data is not too random to predict—its possible values are from a small enough set to be amenable to such things as caching. Assuming 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. This may happen if the user supplies some filters to search for items. The input may be non-deterministic, such as for a text search, and thus the output browsing page may be non-deterministic 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 candidates for caching. An automated analysis could point this out, and a developer is aided in applying this performance optimization, knowing which data sources to use for cache keys (user input parameters) and which to check for cache invalidation (the Inventory database).

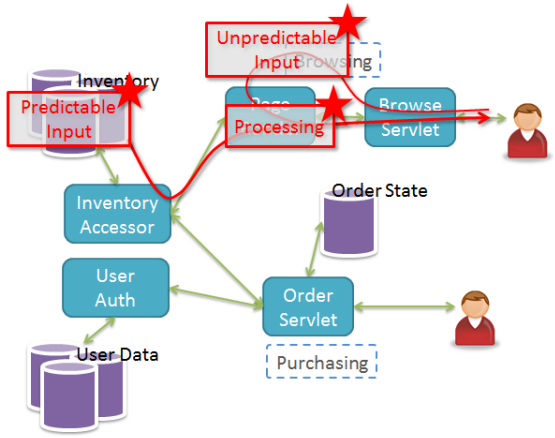

Figure 1.4: Dataflow Indicative of Sources to Be Careful of when Implementing a Cache.

1.2.2.2 State Identification

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 accurately pinpoint various places 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.

1.2.2.3 User-State Identification

Taking the state analysis further, Figure 1.5 shows a case of dataflow tracking which reveals how certain data items only flow to and from certain users. This might be personal data which is not used to communicate between multiple users, such as one's own shopping cart. Once such state is identified by an analysis, one can make decisions like moving the state from the server into the client to make the application more robust. Such a sug-

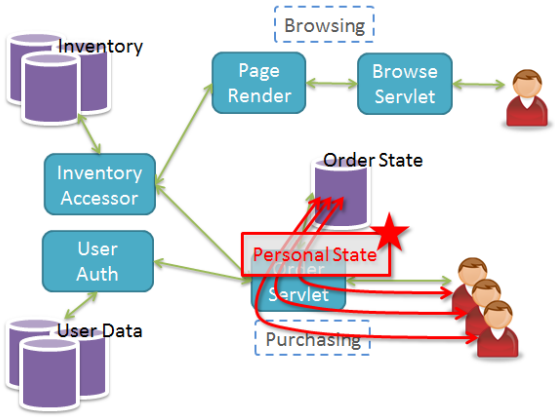

Figure 1.5: Simple Dataflow Indicating Persistent State which is Shared with Only One User.

gestion 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.

1.2.2.4 Cost Analysis

Finally, the bandwidth between the back-end and front-end components is a problem where analyzing dataflow is clearly useful. Partitioning the application 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 one 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 com-

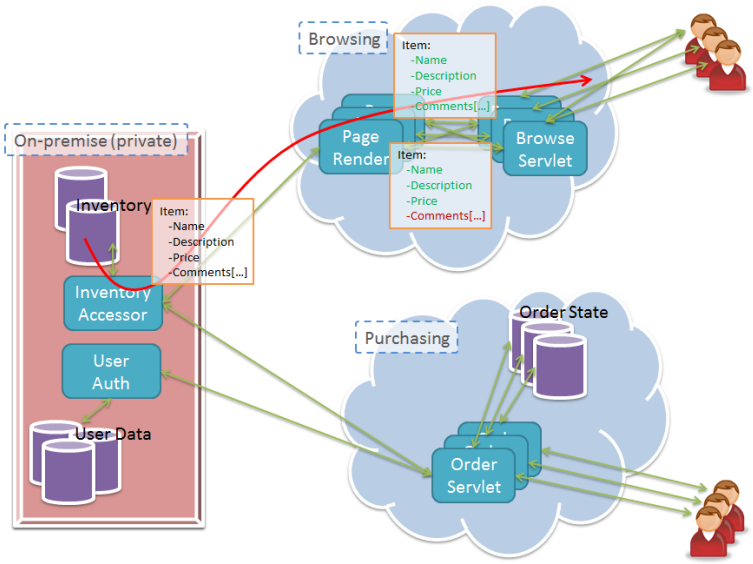

Figure 1.6: Example Partitioning of Application.

munication cost. Thus, the heavy load on the link may be due to data which is not actually used, and which does not need to be transmitted at all.

1.3 Statement of Thesis

We hypothesized that due to the focus on data provided by DIFT, it would naturally help developers in understanding how web applications function, and more importantly it would support a series of automatic analyses to optimize web applications and support other application optimization tools. Specifically, DIFT will support analyses to optimize the flow of data and computation over it, speeding up applications by finding ways to avoid or delay computation. DIFT will easily identify an application's persistent state and how that state is used, which is valuable to understand in cloud deployments or places where replication of computation occurs. Finally, DIFT will provide data useful to more specific analysis tools in the cloud migration space, such as application partitioners.

1.4 Contributions

The main contributions of this work are as follows:

- The identification and implementation of several analyses designed to operate over DIFT data
- The evaluation of these analyses over real web application DIFT traces, pointing out strengths and weaknesses of the techniques used
- Support for the hypothesis that DIFT is a useful technique for understanding and automating optimization of web applications

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 DIFT tool and the analysis tool, breaking down their key components and explaining important design decisions. It also presents an evaluation of the tools, demonstrating the output of the DIFT 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.

```
1: String userName = getUserNameInput();
2: // Any data from user input is tainted, so userName is marked
   // as tainted.
3: String formattedData = "USERNAME:" + userName;
4: // The value of formattedData is based on userName,
   // so 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. [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 reads from a database or input from a user—so that input data can be 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 of 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 [23] [26], where it is used to detect if bytes from untrusted input sources like user input ever end up tainting such sinks as

jump target addresses or executable instructions; [8], 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 [1], which identifies malware and network protocol analysis as uses of DIFT.

The work in [1] 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. Our own system does not use it as an intentional design decision. [15] additionally identifies data lineage tracing as a use of DIFT, which focuses on determining the inputs responsible for outputs in various scientific computations, which is similar to some of the analyses we do. More security-focused systems, which the majority of taint tracking tools are, are presented in [12], [27], and [18]. [7] and [17] 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 Section 3.4.1. These systems are, like other efforts, focused on preventing untrusted data from propagating to secure locations.

Something different is presented in [22], and this 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 tomography 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 graphs, 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 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 by Arun Iyengar et al. [3] [4] [5] [6] and [13] 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.

Degenaro et al. automatically generate dependence graphs [13], but only because the system caches at the level of immediate query results and the dependence graphs come from query analysis. These graphs describe 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. Challenger et al. introduce such useful concepts as page pregeneration and web page fragmenting [6], where dynamic fragments of a page are identified and able to be tied to underlying data. Pregeneration is much like another of the analyses we perform, known as precomputation, which seeks to find computation results which can simply be pregenerated due to being based on infrequently changing data. However, the work in [6] 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. Fluxo [19] 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 actually provide worthwhile execution savings and being sure that adding a cache does not violate the semantics of the application, such as by skipping executions which have important side effects besides producing the cached data.

Tralfamadore [21] 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 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 [2] 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. The benefits of this include 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) [9] [10] [14] [16] [24]. While not a partitioning tool, the dataflows that our system extracts can be used to locate unnecessary flows, where data is communicated but 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.

2.5 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [20] 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 (at 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 tainted data.

Chapter 3

Implementation

3.1 Overview

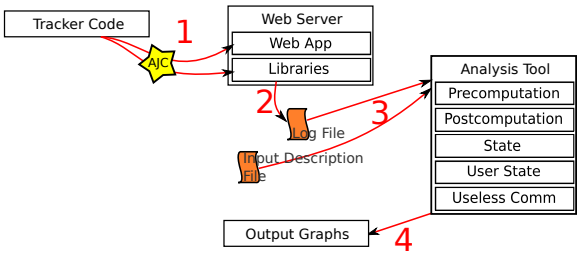

Figure 3.1: Architecture of Entire TaTAMi System.

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 targeted data for Java web applications. This tool collects data which is consumed by an Analysis Tool, which produces output useful for developers.

Consider Figure 3.1, which provides an overview of the operation of our entire system.

- At point 1, the TTT code, written in AspectJ, is mixed into the Web Application code and its libraries using the AspectJ compiler. This must be done for every web application analyzed.
- At point 2 the application is used, and the tracker code running inside it causes taint tracking event logs to be dumped to a log file.
- At point 3 the complete log file and an Input Description File (see Section 3.3.1.3) are fed to the Analysis Tool, which can immediately display the taint trace data in graph form.

- At point 4 automated analyses (Precomputation, Postcomputation, etc.) can be invoked on the taint trace graph, producing Output result graphs for the user to view.

3.2 Taint Tracking Tool

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

3.2.1 Tool Components

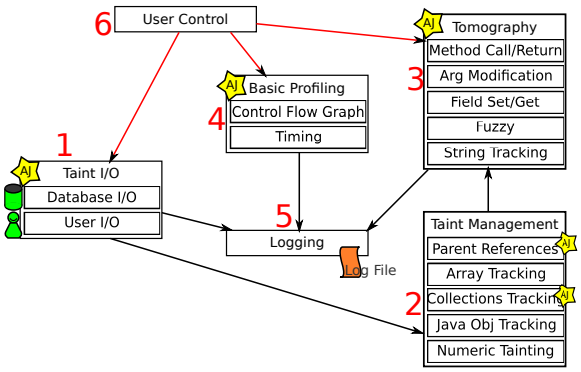

Figure 3.2: Architecture of Taint Tracking Tool.

The main components of the tool are, as shown in Figure 3.2, the following:

- Point 1, Taint I/O System. Detects when taint enters and exits the system. Responsible for telling the TRMS what data is initially tainted.
- Point 2, Taint Reference Management System (TRMS). Collection of systems used to performantly determine whether various objects are tainted.
- Point 3, Tomography System. AspectJ code which intercepts various operations an application can perform which result in the movement of tainted data. This system relies on the TRMS to check if data is tainted.

- Point 4, Basic Profiling. AspectJ code to provide supplementary profiling for certain analyses.
- Point 5, Logging. This system receives updates from the other components and logs the information to a file.
- Point 6, User Control. Provides some control to enable and disable various systems to tweak runtime performance.

3.2.1.1 Taint I/O System

Program 3.1 Simple Taint Tracking AOP Example.

```
1: aspect DatabaseTaint {
2:    pointcut QueryParameterSet():
3:       call(QueryClass.setParam(..));
4:    pointcut ResponseWrite():
5:       call(ResponseClass.write(..));
6:
7:    before(): QueryParameterSet() || ResponseWrite() {
8:       args = Pointcut.getArgs();
9:       if (TRMS.checkTainted(args))
10:         TaintLogger.log("Taint Output: " + args);
11:      }
12:   }
13:}
```

This makes use of AspectJ, monitoring certain method invocations to intercept target data as it enters an application and is written out to various locations like a database or to an output the user sees. The system captures information about the source of data which will be useful for analysis, marks the data as tainted so that it is tracked by the rest of the system, and uses the logging system to record the 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. It then reads the metadata from the PreparedStatement to get table/column descriptions as the source of the data. It then calls the Taint Reference Management System (TRMS) to tell it that the data read from the ResultSet should be treated as tainted. As an example of capturing taint output in AspectJ, consider the pseudocode in Program 3.1. Lines 2-5

declare two pointcuts, which together target any calls to setParam methods on QueryClass objects and to write methods on WebResponseClass objects. This is to capture output to the database and to the user respectively. Line 7 starts a declaration of advice, which will cause the contained code to run 'before' the calls targeted by the pointcuts. Line 8 uses AspectJ to extract the arguments to the targeted method call, and line 9 asks the TRMS if any of the arguments are tainted. If so, they are logged for analysis.

One important point to note when marking input as tainted is how to handle cases where arguments to methods which acquire input are tainted themselves. For example, this is the case when a database query is constructed with some tainted parameters and is then subsequently used to access data. This case is handled by the Taint I/O system, which marks such accessed data with both its own taint identifiers and the identifiers of the tainted arguments.

3.2.1.2 Taint Reference Management System (TRMS)

The TRMS is responsible for keeping track of which objects are tainted. Other parts of the tool will call 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 future techniques could avoid this limitation. 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.

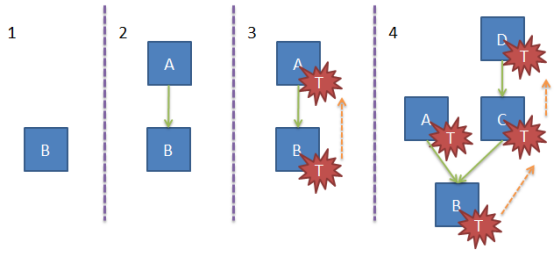

Figure 3.3: Examples of Backwards Taint Propagation.

The main function of the TRMS is the management of a backward reference map. In order to properly track the flow of tainted data, the system needs to know when a Java object is reachability tainted (see Appendix A, 'Taint').

Consider the situation depicted in Figure 3.3.

- Pane 1 shows an untainted object B.
- In pane 2, B is assigned to a field of the object A.
- In pane 3, B becomes tainted, and so A is reachability tainted as it refers to the tainted object B.
- In pane 4, B is set as a field of object C, which is already itself a field of object D. C is reachability tainted as it refers to B, like A. D is also reachability tainted, as by following the reference graph a tainted object can be reached.

A naive strategy for determining whether an object is reachability tainted in Java 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 strategy 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 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.

Consider again Figure 3.3.

- In pane 2, B is assigned to a field of the object A, and A is marked as a parent of B in the backward reference map.
- In pane 3, B becomes tainted, and using the map the TRMS locates its parent A and marks it as tainted as well.
- In pane 4, C marked as a parent of B and is immediately tainted due to a tainted object being assigned to its field. D would have also been marked as a parent of C, and TRMS uses the mapped relationship to taint D as well. For the reference graph depicted in pane 4, if B becomes untainted, all the other objects will as well, using again the backwards reference map.

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 is to track the reachability of arrays and Java runtime objects in the same way as tainted objects themselves. Whenever we check if an object contains taint, we also check 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 further, 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 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.

3.2.1.3 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 interesting data goes. The basic events that are tracked are method calls, method returns, object instantiation, and field set/get; and at each of these points the Tomography System calls 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 to properly track taint through various string composition/modification events.

Such events include creating new strings out of multiple tainted strings, or appending tainted data to untainted strings.

Finally, the Tomography System employs some heuristics to attempt to track tainted Strings through methods which may 'lose' the taint. By 'lose', we mean 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 [8]). As this 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 works well.

3.2.1.4 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 existing tools, but the implementation is provided here in order to keep the data collection process simple, and because there would likely be problems when instrumenting a web application with multiple profilers simultaneously. Currently there is logging to build control flow graphs, and some simple profiling code to track method execution times.

3.2.1.5 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 written to a log file.

3.2.1.6 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 as the application runs. This allows one to disable, for example, tracking during web server initialization which may contain data one is not interested in.

3.2.2 Justification of Implementation

As has been discussed in this thesis, many taint tracking implementations exist. However, 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 the 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.

3.3 Analysis Tool

This was tiplemented in 5.7K lines of Java using the JUNG framework for visualizing and working with graph data.

3.3.1 Tool Components

The main components of the tool are as follows:

3.3.1.1 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 depicted in Figure 3.4 and Figure 3.5. The points to note are as follows:

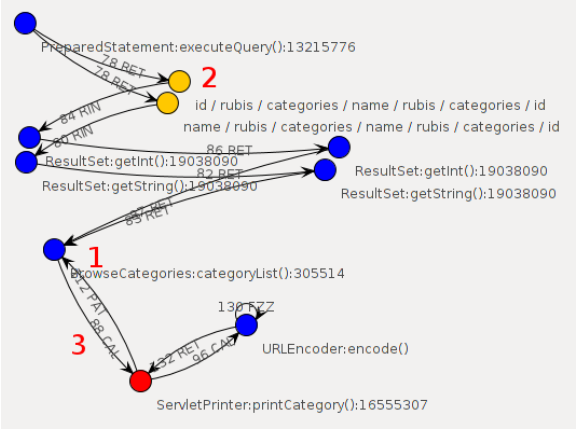

Figure 3.4: Example of Graph Visualization.

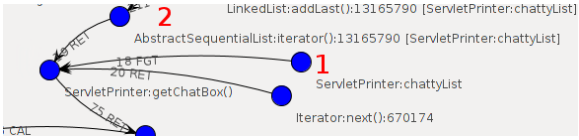

Figure 3.5: Continued Example of Graph Visualization.

- Point 1, Figure 3.4: A node, which is a location where tainted data was detected. Most are method calls like this one, listing the class name, method name (and argument types if necessary), and an object identifier if the call is on an object (as opposed to a class).
- Point 2, Figure 3.4: A special type of node, the input node. These indicate and provide information about where tainted data enters an application. Figure 3.6 and Figure 3.7 outline how to interpret these nodes. For Figure 3.6, note that the first field, identified as the 'target column', is the most important. The rest of the fields identify the various columns present in the data obtained from a database query, and this first field specifies which of these columns was actually accessed.
- Point 3, Figure 3.4: 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. The identifier can be used to lookup more detailed information about the event. See Ta-

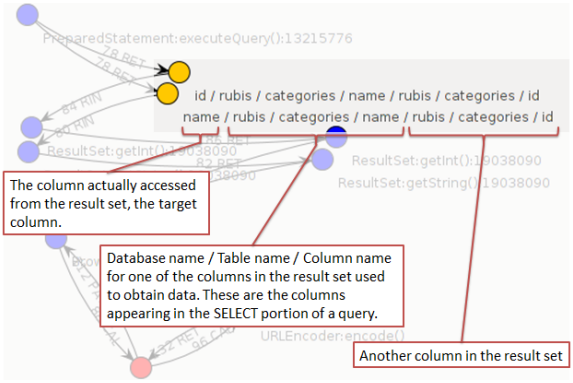

Figure 3.6: Format of Input Nodes for Database Data.

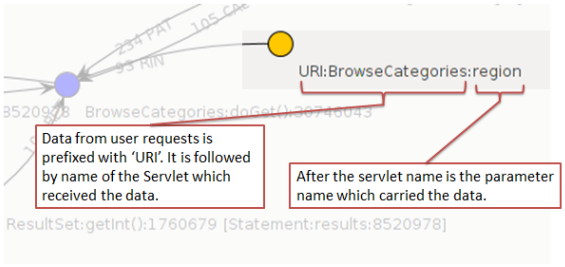

Figure 3.7: Format of Input Nodes for User Request Data.

ble 3.1 for a breakdown of the possible edge types.

- Point 1, Figure 3.5: A node describing a field of a class/object where tainted data is stored. These list the class name and the field name, along with an object identifier if needed, much like the method call nodes.

- Point 2, Figure 3.5: Note the text within the square brackets for the label of this node. It is the same as the label for the node at point 1 in the same figure. This text means that the object represented by the node (in this case an AbstractSequentialList) was accessed from the field indicated in the brackets, ServletPrinter:chattyList. This information helps to track the flow of data in the graphs.

CAL	Function call, directed towards the called function
RET	Function return, directed towards the calling function
PAT	Taint returned in modified arguments, directed towards the calling function. PAT = Post-return Argument Taint
FST	Field set, directed towards the field
FGT	Field get, directed away from the field
FZZ	Fuzzy taint flow, these will start and end on the same node where fuzzy propagation was detected. See the last paragraph of Section 3.2.1.3
IMP	Implied taint flow, directed from the input method to the output method. See Section 3.3.1.2
RIN	Input edges, directed away from input nodes
OUT	Output edges, directed towards called functions responsible for output to database or user

Table 3.1: Taint Graph Visualization Edge Types.

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, it is possible to filter the graph in various ways:

- Show only those edges for the flow of a particular piece of tainted data.
- Show the flow of tainted data for a single web request.
- Given an edge, show only the flow of tainted data which follows that edge.
- Manually filter by allowing a user to traverse the graph node-by-node.

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.

3.3.1.2 Graph Preprocessing

Before the graphs are displayed or analyzed, the Analysis Tool performs preprocessing steps on the data in order to better work with it. The first

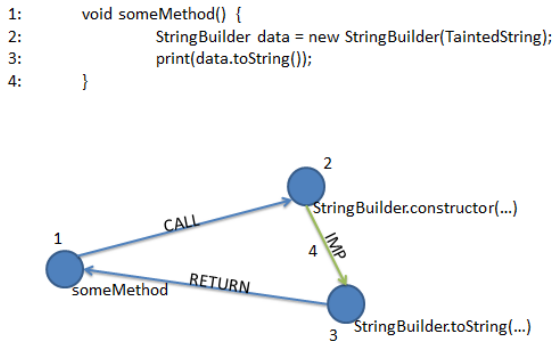

Figure 3.8: Example of an Implied Taint Flow Edge.

is the addition of implied edges. There are cases for tracking the flow of tainted data where representation of it requires some care. Consider for example the case in Figure 3.8 where a tainted String is used to construct a new StringBuilder (line 2) which is then printed (line 3);

- Taint flows from the calling method at point 1 to the StringBuilder constructor at point 2.
- The StringBuilder's toString() method is then called, and taint flows from the toString() method at point 3 back to the calling method.

Depending on the ability of the taint tracker to see the internal dataflow in certain objects (in this case a native Java object, StringBuilder, which AspectJ cannot instrument), there may be no logged taint flow event to show how the data gets from point 2 to point 3. 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 have lower granularity. Instead, the graph is searched for such cases where taint flows into an object through some method call and that same taint (or taint derived from it) flows out of the object through a different method. Implied edges are added between the nodes in question to reflect the flow of tainted data, as is shown at point 4 in the figure.

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 described in more detail in Section 3.3.2.6, and is useful for some of the automated analyses described later.

3.3.1.3 Input Description File (IDF)

This is an XML file created by the user which supplies information needed by some of the analyses. Its format is given in Appendix B, along with the data in the files used for the evaluated applications. It is read by the analysis tool and gives a measure of how deterministic various data sources (database columns and user web request parameters) are, ranging from sources with values which change rarely to non-deterministic input.

3.3.1.4 Automated Analyses

The main purpose of the Analysis Tool is to provide easy to use automated 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, in some cases with supplementary text explaining the results.

3.3.2 Available Analyses

These were chosen based on our review of the literature. The caching, precomputation, and postcomputation analyses were identified by the Fluxo [19] system. These three serve to directly improve the operation of a web application by speeding up interaction with the users, saving on computation to do so. The state-based analyses and wasteful communication analyses described below were chosen due to an influence from application partitioning research and considerations of moving an application into the cloud [9] [10] [16]. We wanted to provide data which could be of use in these scenarios, potentially supporting future analysis tools.

3.3.2.1 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 than 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, then 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 mostly deterministic inputs. By mostly deterministic we mean inputs with values over a small enough range that the cache will be useful and not frequently missing. This analysis relies on the input description file (IDF) described in Section 3.3.1.3 to determine whether or not given tainted data is non-deterministic. 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 a possible cache as well as a guide for what parts of the application must be considered when implementing it.

The high-level algorithm for performing this analysis is given in Program 3.2.

3.3.2.2 Precomputation

This analysis is essentially the same as the one for caching. 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 developer is advised to simply compute the result of the computation in advance and have the application 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 deterministic sources.

The high-level algorithm for performing this analysis is given in Program 3.4.

Program 3.2 High Level Algorithm for Caching Analysis, Part 1.

```
1: // EXECUTION START
2: TG = taint trace graph
3: CG = call graph taken during taint trace,
4:       with method call execution times
5:
6: TGPredictable = pruneToVariability(TG, PREDICTABLE)
7: TGRandom = pruneToVariability(TG, RANDOM)
8:
9: PredictableConnectedSubGraphs =
10:   getConnectedSubGraphs(TGPredictable)
11:
12: foreach PredictableCS in PredictableConnectedSubGraphs:
13:   if (checkOverCost(PredictableCS, CG, ACCEPT_THRESHOLD)):
14:     /* Check that the execution of this graph does not
15:      * result in the flow of non-deterministic data */
16:     RandomSideEffects = getSideEffects(PredictableCS,
17:                                        TGRandom, CG)
18:     /* If there are side effects, it may be that they are
19:      * cheap enough to execute even when fetching from the
20:      * cache. */
21:     if (checkOverCost(RandomSideEffects, CG,
22:                       REJECT_THRESHOLD)):
23:       continue
24:     /* Present the graph to the user */
25:     colorOutputs(PredictableCS)
26:     colorInputs(PredictableCS)
27:     showGraphToUser(PredictableCS)
28:     /* Indicate the side effects, if any, for the user to
29:      * handle. */
30:     showSideEffectsToUser(RandomSideEffects)
31:
```

Continued in Program 3.3

Program 3.3 High Level Algorithm for Caching Analysis, Part 2.

```
32: // HELPER METHODS
33: /* @InputGraph: A taint trace graph
34:  * @Variability: Specify the variability
35:  * @Return: Copy of @InputGraph with edges carrying data
36:  *  with variability != @Variability removed */
37: function pruneToVariability(InputGraph, Variability)
38:
39: /* @InputGraph: A taint trace graph, possibly with nodes
40:  *  such that no path exists between them
41:  * @Return: A partitioning of @InputGraph which removes no
42:  *  edges and within each partition there are no
43:  *  disconnected nodes */
44: function getConnectedSubGraphs(InputGraph)
45:
46: /* @TaintGraph: A taint trace graph
47:  * @CallGraph: A call graph taken during taint trace with
48:  *  method execution times
49:  * @THRESHOLD: A maximum value
50:  * @Return: True if and only if the total execution time
51:  *  for the computation represented by edges in
52:  *  @TaintGraph exceeds @THRESHOLD. Uses @CallGraph to
53:  *  compute this time */
54: function checkOverCost(TaintGraph, CallGraph, THRESHOLD)
55:
56: /* @Graph: A taint trace graph
57:  * @SideEffectGraph: A taint trace graph
58:  * @CallGraph: A call graph taken during taint trace
59:  * @Return: Any edges in @SideEffectGraph which were
60:  *  caused by edges in @Graph, by checking descendent
61:  *  method calls of edges in @Graph using @CallGraph */
62: function getSideEffects(Graph, SideEffectGraph, CallGraph)
```

Program 3.4 High Level Algorithm for Precomputation Analysis.

```
1: // EXECUTION START
2: TG = taint trace graph
3: CG = call graph taken during taint trace,
4:       with method call execution times
5:
6: TGStable = pruneToVariability(TG, STABLE)
7: TGPredictable = pruneToVariability(TG, PREDICTABLE)
8: TGRandom = pruneToVariability(TG, RANDOM)
9:
10: StableConnectedSubGraphs = getConnectedSubGraphs(TGStable)
11:
12: foreach StableCS in StableConnectedSubGraphs:
13:   /* Check that the execution of this graph does not
14:    * result in the flow of non-deterministic data or
15:    * require any non-deterministic input */
16:   if (getSideEffects(StableCS, TGRandom, CG) == {}):
17:     if (getSideEffects(StableCS, TGPredictable, CG) == {}):
18:       /* Present the graph to the user */
19:       colorOutputs(StableCS)
20:       colorInputs(StableCS)
21:       showGraphToUser(StableCS)
```

3.3.2.3 Postcomputation

The Postcomputation analysis looks for flows of tainted data which represent computations which could be deferred. 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 data in the graph, looking for subgraphs carrying only taint which never flows to a user output.

The high-level algorithm for performing this analysis is given in Program 3.5.

3.3.2.4 Application State

The goal of this analysis is to locate persistent state in an application. This does not refer to the data an application keeps in a database, but rather to more temporary state kept in session 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 stored 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 tendency to shut down idle applications. This analysis identifies such state by looking for instances where the same pieces of data are accessed over multiple requests. It then presents subgraphs showing where the state is stored and what parts of the application use the state.

The high-level algorithm for performing this analysis is given in Program 3.7.

Program 3.5 High Level Algorithm for Postcomputation Analysis, Part 1.

```
1: // EXECUTION START
2: TG = taint trace graph
3: RequestSubGraphs = getRequestSubGraphs(TG)
4:
5: foreach RequestSG in RequestSubGraphs:
6:   UserOutputEdges = getUserOutputEdges(RequestSG)
7:   DBOutputs = getDBOutputEdges(RequestSG)
8:
9:   foreach DBOutputEdge in DBOutputs:
10:     PostCompEdges = {}
11:     /* Start from an edge which outputs to the database.
12:      * These are good candidates for blocking computation
13:      * deferrable until later. Work backwards from the
14:      * output to see how much computation influencing it
15:      * can be deferred. Stop before the computation could
16:      * influence output to the user. */
17:     backwardsExpand(PostCompEdges, DBOutputEdge,
18:                     UserOutputEdges, RequestSG)
19:     /* If we found anything, show it to the user */
20:     if (PostCompEdges != {}):
21:       showGraphToUser(
22:     createGraphFromEdges(PostCompEdges))
23:
```

Continued in Program 3.6

Program 3.6 High Level Algorithm for Postcomputation Analysis, Part 2.

```
24: // HELPER METHODS
25: /* @InputGraph: A taint trace graph
26:  * @Return: A partitioning of the edges of @InputGraph
27:  *  such that the edges in each partition are all from the
28:  *  same request. */
29: function getRequestSubGraphs(InputGraph)
30:
31: /* @InputGraph: A taint trace graph
32:  * @Return: Any output type edges in @InputGraph for
33:  *  methods known to write output to the user */
34: function getUserOutputEdges(InputGraph)
35:
36: /* @InputGraph: A taint trace graph
37:  * @Return: Any output type edges in @InputGraph for
38:  *  methods known to write output to the database */
39: function getDBOutputEdges(InputGraph)
40:
41: /* @FoundEdges: An empty set used to store results
42:  * @CurrentEdge: backwardsExpand is recursive. Given an
43:  *  edge @CurrentEdge, work backwards to any edges which
44:  *  could have influenced @CurrentEdge, adding them to
45:  *  @FoundEdges
46:  * @UserOutputEdges: Provides the condition to stop
47:  *  working backwards. If from a @CurrentEdge it is
48:  *  possible, working forwards to edges influenced by
49:  *  @CurrentEdge, to reach edges in @UserOutputEdges,
50:  *  return
51:  * @Graph: A taint trace graph */
52: function backwardsExpand(FoundEdges, CurrentEdge,
53:                           UserOutputEdges, Graph)
```

Program 3.7 High Level Algorithm for Application State Analysis, Part 1.

```
1: // EXECUTION START
2: TG = taint trace graph
3: ByRequestTaintIDs = {}
4: /* Taint IDs which occur in multiple requests */
5: PersistentTaintIDs = {}
6:
7: foreach RequestSG in RequestSubGraphs:
8:   ByRequestTaintIDs.add(getTaintIDSet(RequestSG))
9:
10: /* Find any taint IDs which were present in multiple
11:  * Request Graphs. Such data had persisted beyond a single
12:  * request, and we call such taint IDs persistent. */
13: for (i = 0; i < ByRequestTaintIDs.size(); i+=1):
14:   for (j = i+1; j < ByRequestTaintIDs.size(); j+=1):
15:     SetA = ByRequestTaintIDs[i]
16:     SetB = ByRequestTaintIDs[j]
17:     PersistentTaintIDs.add(SetA intersect SetB)
18:
19: foreach PersistentTaintID in PersistentTaintIDs:
20:   foreach Edge in TG.getSortedEdges():
21:     if (edge.getAllTaintIDs().contains(PersistentTaintID)):
22:    /* Color any edges which carry persistent data */
23:       colorEdgeRed(edge)
24:       if (LastRequestID != null AND Edge.getRequestID()
25:         != LastRequestID)
26:         /* Find points where request ID changes to find
27:          * places where persistent data is stored. Color
28:          * these differently */
29:         colorEdgeGreen(edge)
30:       LastRequestID = Edge.getRequestID()
31:
32: showGraphToUser(TG)
33:
```

Continued in Program 3.8

Program 3.8 High Level Algorithm for Application State Analysis, Part 2.

```
34: // HELPER METHODS
35: /* @Graph: A taint trace graph
36:  * @Return: The set of taint IDs present among the edges
37:  *  in @Graph */
38: function getTaintIDSet(Graph)
39:
40: /* See earlier psuedocode example */
41: RequestSubGraphs = getRequestSubGraphs(TG)
```

3.3.2.5 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 high-level algorithm for performing this analysis is given in Program 3.9.

3.3.2.6 Wasteful Communication

Section 3.3.1.2 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

Program 3.9 High Level Algorithm for User State Analysis, Part 1.

```
1: // EXECUTION START
2: TG = taint trace graph
3:
4: RequestSubGraphs = getRequestSubGraphs(TG)
5: PersistentTaintIDs = As in earlier pseudocode
6:
7: /* Remove from PersistentTaintIDs any IDs which were
8:  * derived from others. This saves work later. */
9:
10: MasterMap = getMasterMap(PersistentTaintIDs,
11:                           RequestSubGraphs)
12:
13: foreach PersistentTaintID in MasterMap.keys():
14:   RequestIDToEdgeMap = MasterMap.get(PersistentTaintID)
15:   foreach RequestID in RequestIDToEdgeMap.keys():
16:     StartEdgeToFlowGraphMap =
17:   RequestIDToEdgeMap.get(RequestID)
18:     foreach StartEdge in StartEdgeToFlowGraphMap.keys():
19:       FlowGraph = StartEdgeToFlowGraphMap.get(StartEdge)
20:       /* This map is needed to compare the flow of taint
21:        * in one request with the flow of the same taint in
22:        * other requests. */
23:       OtherFlows = RequestIDToEdgeMap.copy()
24:       OtherFlows.remove(RequestID)
25:
26:       UserStateEdges = {}
27:       findUserState(UserStateEdges, {StartEdge},
28:                     FlowGraph, OtherFlows)
29:
30:       if (UserStateEdges != {}):
31:         colorEdge(StartEdge)
32:         showGraphToUser(
33:    createGraphFromEdges(UserStateEdges))
34:
```

Continued in Program 3.10

Program 3.10 High Level Algorithm for User State Analysis, Part 2.

```
35: // HELPER METHODS
36: /* @UserStateEdges: An empty set to store results
37:  * @Path: @Path starts with a single edge representing a
38:  *  read from persistent data. findUserState recursively
39:  *  attempts to grow the path along matching edges in
40:  *  @FlowGraph. The path grows if it can be found in
41:  *  @FlowGraph and in at least one of @OtherFlows, but not
42:  *  if it can be found in a graph from @OtherFlows which
43:  *  serves data to a different user than @FlowGraph. When
44:  *  @Path is successfully grown, the new edges are added
45:  *  to @UserStateEdges
46:  * @FlowGraph: A taint trace graph indicating the flow and
47:  *  propagation of a single piece of tainted data
48:  * @OtherFlows: Flow graphsb for the same data as @FlowGraph
49:  *  but from different requests */
50: function findUserState(UserStateEdges, Path,
51:                        FlowGraph, OtherFlows)
52:
53: /* @PersistentTaintIDs: Set of taint IDs which occur in
54:  *  multiple requests
55:  * @RequestSubGraphs: A partitioning of the edges of a
56:  *  taint trace graph such that the edges in each
57:  *  partition are all from the same request.
58:  * @Return: A map:
59:  *  {PersistentTaintID -> {RequestID ->
60:  *   {StartEdge -> FlowSubGraph}}}
61:  *  This map shows, for each persistent taint ID, which
62:  *  request graphs carry that taint, what are the earliest
63:  *  edges carrying that taint, and where each bit of
64:  *  persistent taint flows from these points of
65:  *  origination. */
66: function getMasterMap(PersistentTaintIDs, RequestSubGraphs)
```

such.

An obvious use for such an analysis is to help a developer eliminate potentially wasteful 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. One could attempt to eliminate it altogether or employ a lazy communication method where the data is only communicated when it is actually needed.

The high-level algorithm for performing this analysis is given in Program 3.11.

3.4 Additional Details

3.4.1 Numeric Value Tracking in TRMS

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. Note that this is a temporary workaround to deal with limitations of AspectJ, and could be replaced by something more robust in future implementations.

Program 3.11 High Level Algorithm for Wasteful Communication Analysis.

```
1: // EXECUTION START
2: TG = taint trace graph
3:
4: /* Look at every SubTaintedObject. These are objects not
5:  * directly passed as arguments or return values, but
6:  * rather those which are reachable from such. If taint is
7:  * passed in this form and subsequently never found to be
8:  * passed directly at the level of an argument or return
9:  * value, it is never accessed and the user should be
10: * informed of this. */
11: foreach Edge in TG.getSortedEdges():
12:   foreach TaintedObject in Edge.getTaintedObjects():
13:     foreach SubTaintedObject in TaintedObject.
14:                                 getSubTaintedObjects():
15:       if (forwardSearch(Edge, TG, SubTaintedObject)):
16:         SubTaintedObject.setUnused()
17:         colorEdge(Edge)
18:
19: showGraphToUser(TG)
20:
21: // HELPER METHODS
22: /* @Edge: An edge in @Graph to start searching from
23:  * @Graph: A taint trace graph
24:  * @TargetTaintedObject: A tainted object with a taint ID
25:  *  to search for
26:  * @Return: True if by working forwards from @Edge in
27:  *  @Graph (to edges which could be influenced by @Edge
28:  *  and so on, recursively, using edge context information
29:  *  to ensure that considered edges were in the same
30:  *  thread of execution), an edge can be found with the
31:  *  taintID for @TargetTaintedObject in the top level. Top
32:  *  level means actual arguments and return values as
33:  *  opposed to tainted objects which are merely reachable
34:  *  from them. */
35: function forwardSearch(Edge, Graph, TargetTaintedObject)
```

Chapter 4

Evaluation

4.1 Evaluation Strategy/Goals

To test the claims made in the thesis using the Tracking and Analysis tools, the focus was on performing a 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 for a 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.

For reference, the actual results can be found in the following sections:

- Precomputation Analysis: Section 4.4.2
- Caching Analysis: Section 4.4.3
- Postcomputation Analysis: Section 4.4.4
- Persistent State Analysis: Section 4.4.5
- User State Analysis: Section 4.4.6
- Wasteful Communication Analysis: Section 4.4.7

4.2 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.

4.2.1 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, 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. However, it was not written to be easily amenable to such analyses, and as such it served as a reasonable introductory proving ground for the taint tracking and analysis tools.

4.2.1.1 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: 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 performing 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.

4.2.2 jGossip

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

4.2.2.1 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 use these libraries, so jGossip serves as a more realistic example. However, the use of some libraries presents difficulties for the kinds of analyses performed here. In particular, JSP is 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 implement 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.

4.3 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 traces data in Strings read from designated input sources, as well as primitive numeric inputs that the tracker is manually told to trace. As most data in a Java program is ultimately held in primitives and Strings, we only really miss data in booleans, chars, and bytes. Thus as a first effort, we feel that our tracked data types capture enough to support useful analyses.

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 is 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 random input data which could be easily identified. The applications were then driven through the operations that we wanted to use in our evaluations, and the taint tracking system was told to scan any data (passed through method calls, returns, etc.) for the occurrence of the known random input data. If the data was encountered, it was checked as to whether or not the system had properly marked it as tainted. In all cases the data had been marked as such, and the tracker was judged as being reasonably complete.

4.4 Application Results

4.4.1 Example Trace

In order to understand the evaluation which follows, consider Figure 4.1, which shows a visualized taint trace taken from loading a page in the RUBiS

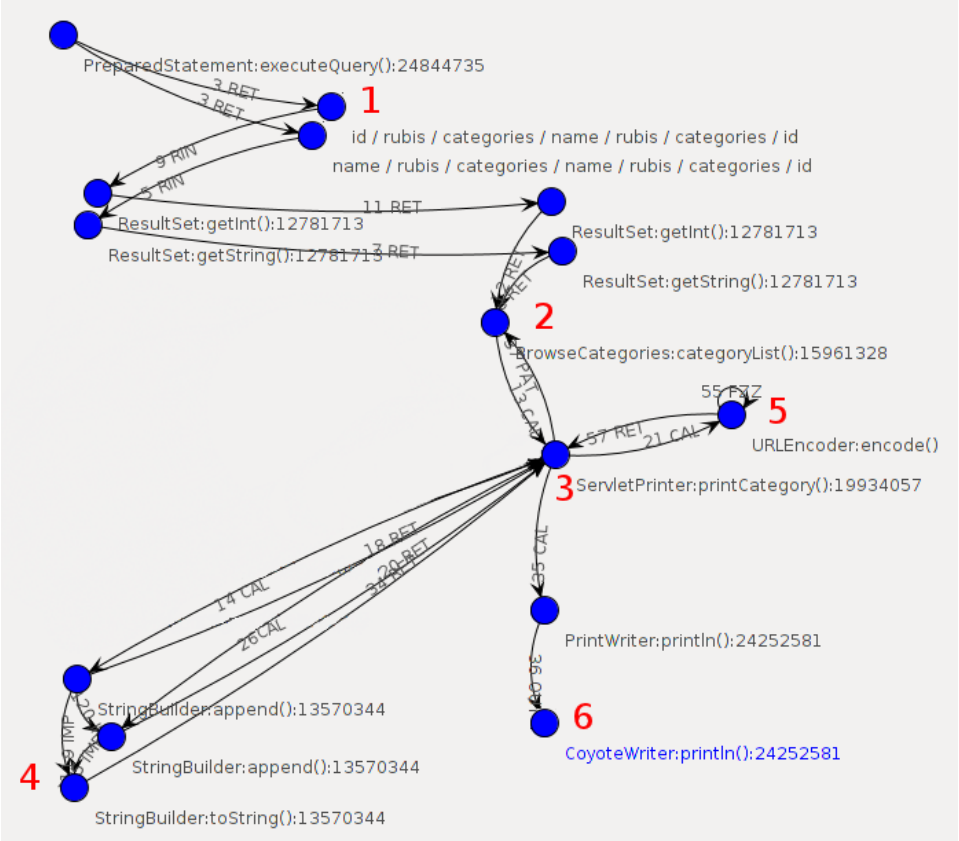

Figure 4.1: RUBiS Browse Categories Trace.

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 4.1 are two input nodes for database data used in displaying the page (see Figure 3.6 for the format of this kind of node, as well as Table 3.1 for a summary of the edge types which follow). These nodes represent the access of data fields from a ResultSet (which comes from the preceding call to executeQuery()) which includes the ID and name columns from the categories table in the RUBiS catalog. A separate node is present for each field accessed. The edges from these nodes are labeled 'RIN' for 'Returning Input', marking them as points where tainted data originates. Edges are la-

beled 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 group projecting from this point at point 4. This shows 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 the StringBuilder to get the concatenated String, and the preprocessing described earlier adds 'IMP' (implied) edges to show how the data sent to the append method 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.

4.4.2 Precomputation

4.4.2.1 RUBiS

4.4.2.1.1 Experimental Setup First, a page which displays a list of item categories is requested in RUBiS while performing taint tracking, producing a log file. RUBiS fetches the item categories from a database table and uses them to generate the output HTML which can be seen in Figure 4.6. Following this the same page is requested, and a message is entered into a form field and submitted to display the input on the page. This causes 2 requests: one to submit the message and another to display the modified page. The output HTML is shown in Figure 4.2. These tests were chosen as this page would likely be accessed frequently (a user must pick a category for most actions on the site, such as viewing and creating auctions), and it was

Figure 4.2: RUBiS Browse Categories with Chat Tainted Output.

```
1: <title>RUBiS available categories</title> [NONTAINTED]
2: <p>Chats</p><p>hello 1</p> [TAINTED]
3: <form action="SubmitChat" method=POST> [NONTAINTED]
4: <input type=text size=20 name=chatMessage /> [NONTAINTED]
5: <input type=submit value="Post" /></form> [NONTAINTED]
6: <h2>Currently available categories</h2><br> [NONTAINTED]
7: <a href="SearchItemsByCategory?category=1&
    categoryName=Antiques">Antiques</a><br> [TAINTED]
8: <a href="SearchItemsByCategory?category=2&
    categoryName=Books">Books</a><br> [TAINTED]
9: <a href="SearchItemsByCategory?category=3&
    categoryName=Business">Business</a><br> [TAINTED]
10:</body> [NONTAINTED]
11:</html> [NONTAINTED]
```

known to make use of database data which changed very rarely. We added the form field to the application ourselves, under the pretense of providing a simple means for users to post messages to each other and more importantly to introduce a flow of non-deterministic data to the page.

4.4.2.1.2 Experimental Goals For this experiment we wanted the analysis to identify computations which could be skipped completely, replaced by precomputed data also extracted by the analysis. Since RUBiS is a simple application, we hoped to find large portions of its output able to be precomputed. Finally, where non-deterministic data was introduced, we wanted the analysis to adapt and find constrained opportunities for precomputation.

4.4.2.1.3 Analysis

Taint Flow Breakdown The log for the first request generates the taint graph shown in Figure 4.1. A walkthrough of the flow of taint was already given in Section 4.4.1.

For the second request involving the user input field, the taint graph shown in Figure 4.3 is obtained. The important thing to note is that this graph contains the flow of non-deterministic data—the list of messages can

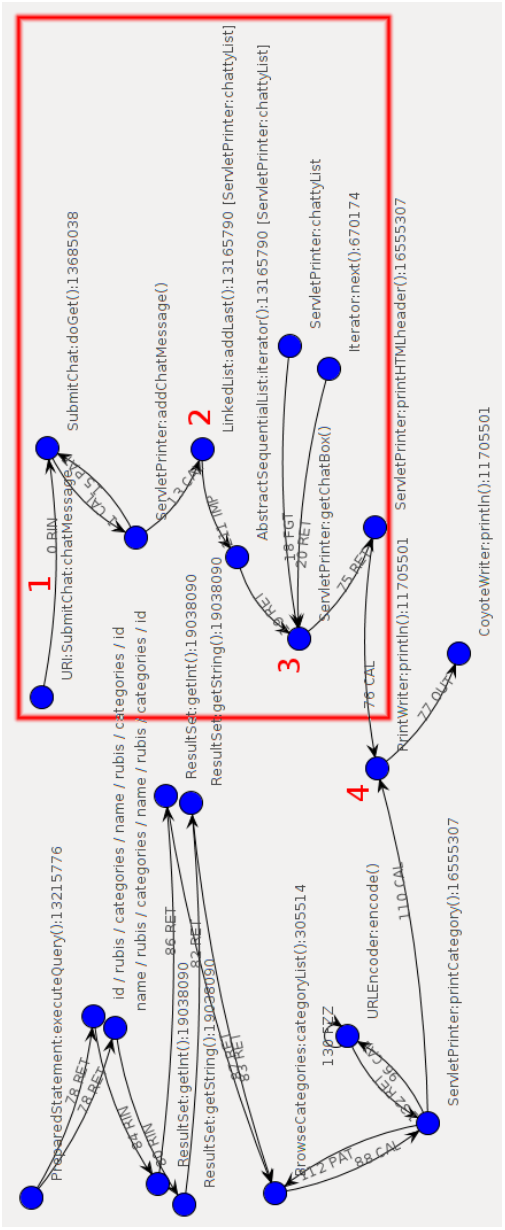

Figure 4.3: RUBiS Browse Categories With Chat Trace.

change at any time, with random data input to them, and as such any computations which rely on message data are poor candidates for caching.

Breaking this flow down, the left side of the graph in the figure is the same as in Figure 4.1, while the portion outlined in the red box shows the additional flow of the message data. This new portion of the graph actually spans two requests to the application—one to submit the 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 request which submits the message.
- Point 1 shows the source node with 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, and passed to the addChatMessage method of a ServletPrinter object where it is concatenated with some formatting text.
- At Point 2 the message is appended to a LinkedList by the addLast method. Note that the label for this addLast method call node ends in the name of a field, the 'chattyList' field of a 'ServletPrinter' object. This means that the chattyList field points to the LinkedList being appended to and indicates that by storing tainted data in the LinkedList it is stored in the chattyList field.
- Point 3 shows where the message data flows for the second request, which displays the category list request along with the messages. Starting with edge '18 FGT', the value of the chattyList field is read, which we know is a LinkedList. Edge '19 RET' shows an iterator over this LinkedList being returned from the LinkedList, and edge '20 RET' shows just one of the calls to the next() method of the iterator, which return the messages.
- The getChatBox() method combines these messages with some formatting text and returns the result to the printHTMLheader() method, which writes them out for the user to see at point 4.

Analysis Results Breakdown The taint graphs, along with the IDF given in Appendix B.2, are then input to the precomputation analysis, which runs automatically without intervention over the data. The graph presented in Figure 4.4 is obtained from the flow for the first request (no message data).

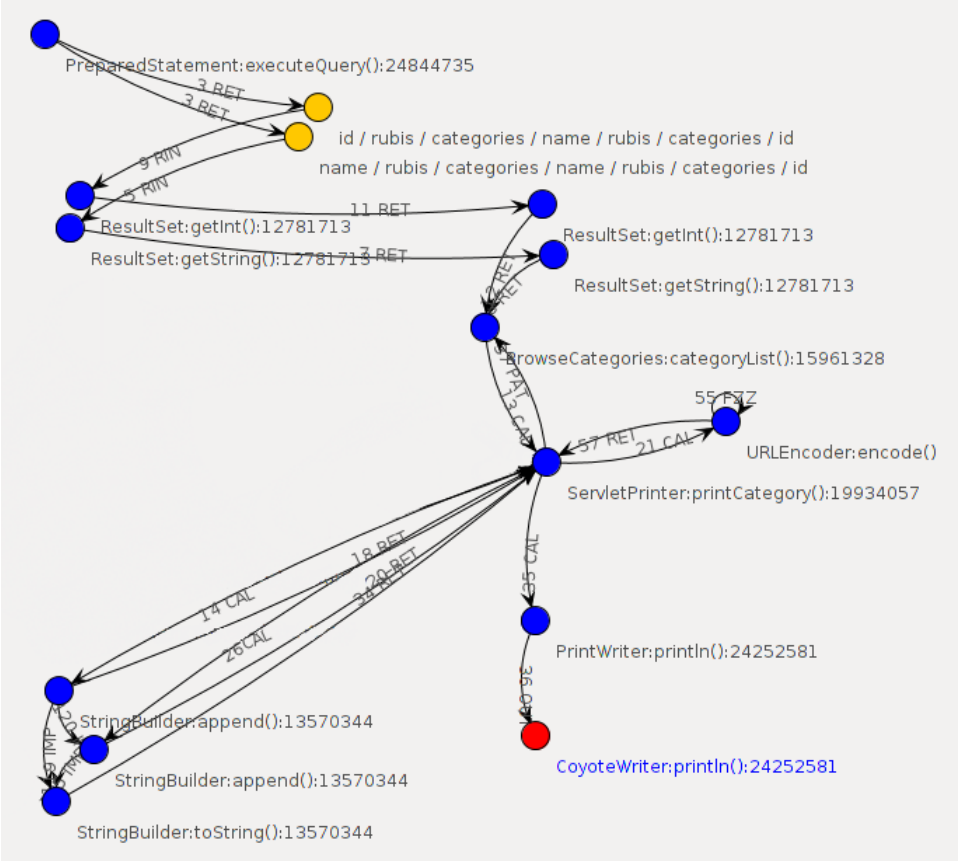

Figure 4.4: RUBiS Browse Categories Precomputation Results.

The first thing to notice here is that it is the same graph as the input taint graph. Such is not always the case for this analysis, but in this instance the analysis identified the entire flow 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 marked as being deterministic (stable) in the IDF. This means that one does not expect category names or ids to be non-deterministic—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 analysis determined that the tainted data flowing in this graph was entirely deterministic data, and thus the output of

the analysis is the entire graph with the inputs and outputs coloured orange and red respectively.

For more detailed results, one can view what precomputable tainted data was sent to the output nodes in the graph. Figure 4.7 shows the HTML data for three category selection hyperlinks. Compare this to Figure 4.6, also obtained from our tool, which shows all data written to the user along with whether or not pieces of it are tainted. Only three lines are marked as tainted, the same three which are shown to be precomputable in Figure 4.7. The rest of the lines are marked non-tainted, and thus are likely from non-computed data.

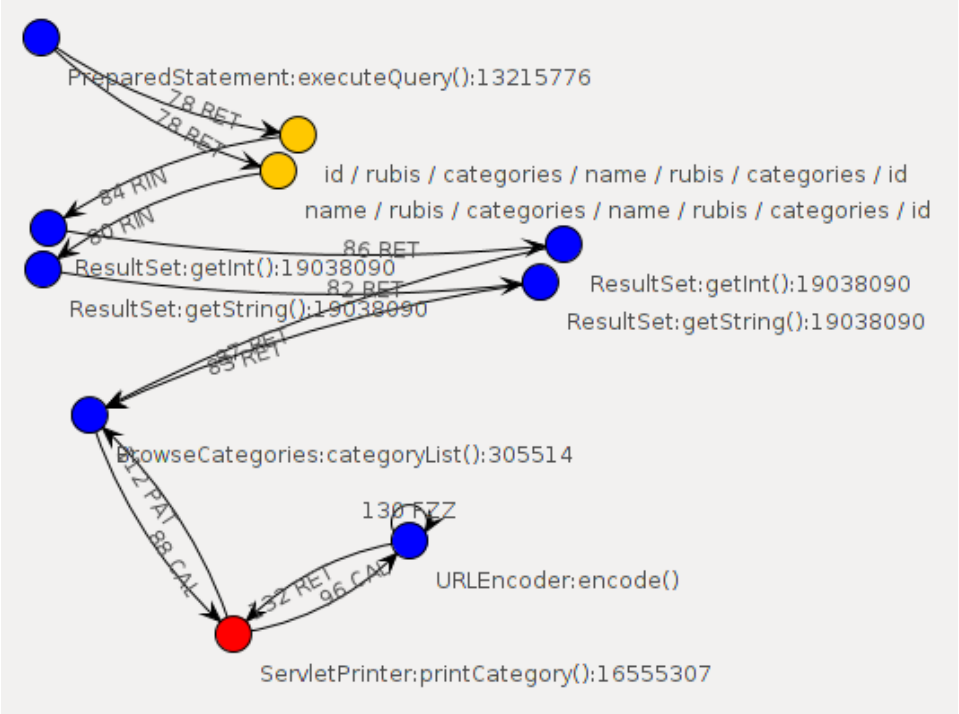

Figure 4.5: RUBiS Browse Categories with Chat Precomputation Results.

Figure 4.5 shows the result of the precomputation analysis on the trace from Figure 4.3. It is exactly like the analysis results from Figure 4.4 except that the graph ends at the red PrintCategory node, no longer extending

to the println method call nodes. Furthermore, it obviously does not have any of the nodes carrying message data. This is because those nodes and the println nodes have message data flowing through them, which has been marked in the IDF as being non-deterministic. The precomputation analysis will not return any nodes relaying random data, because if the computation that those nodes represented were replaced by precomputed data, the results would be inconsistent when the inputs changed. The part of the graph shown in Figure 4.5 only relies on deterministic data as inputs.

Figure 4.6: RUBiS Browse Categories Tainted Output.

```
1: <title>RUBiS available categories</title> [NONTAINTED]
2: <form action="SubmitChat" method=POST> [NONTAINTED]
3: <input type=text size=20 name=chatMessage /> [NONTAINTED]
4: <input type=submit value="Post" /></form> [NONTAINTED]
5: <h2>Currently available categories</h2><br> [NONTAINTED]
6: <a href="SearchItemsByCategory?category=1&
    categoryName=Antiques">Antiques</a><br> [TAINTED]
7: <a href="SearchItemsByCategory?category=2&
    categoryName=Books">Books</a><br> [TAINTED]
8: <a href="SearchItemsByCategory?category=3&
    categoryName=Business">Business</a><br> [TAINTED]
9: </body> [NONTAINTED]
10:</html> [NONTAINTED]
```

Looking to more detailed results, Figure 4.8 shows the precomputable data which flows into the 'ServletPrinter:printCategory' node. As before, compare this to the tainted/non-tainted data written to the user in Figure 4.2. Again there are three lines which match in these figures, these being precomputable, but in Figure 4.2 line 2 is also tainted and not precomputable.

4.4.2.1.4 Developer Interpretation and Response Interpreting the results for the first request, a developer can reasonably assume that the computations which produce the output page can all be precomputed. At no point do these computations make use of any non-deterministic data—the only tainted data is from sources which rarely change, and the rest of the output isn't tainted which means it is also likely non-computed. One could

Figure 4.7: RUBiS Browse Categories Precomputation Details.

```
Output Data:
Node: CoyoteWriter:println String:26517368
   Data:
    <a href="SearchItemsByCategory?category=1&
    categoryName=Antiques">Antiques</a><br>
   Data:
    <a href="SearchItemsByCategory?category=2&
    categoryName=Books">Books</a><br>
   Data:
    <a href="SearchItemsByCategory?category=3&
    categoryName=Business">Business</a><br>
```

alter the application to immediately output an entire pregenerated page 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.

For the request dealing with the message data, a developer can still mostly precompute the output. Figure 4.2 shows that only line 2 depends on random data. The application could be modified to only dynamically generate this small portion, while using pregenerated data for the rest of it.

In both cases computation can be saved, and the application will be better able to serve higher loads. One could even have more freedom in the type of server this page is served from. With significantly less computation needed, less processor power is required.

4.4.2.1.5 Summary Having identified a critical page which could be entirely pregenerated, as well as having had the analysis successfully locate precomputable data mixed with non-deterministic data, the experimental goals were met. The results were easy to understand, and a developer could easily modify the application to precompute the page and serve it instead of generating it from the database data.

Figure 4.8: RUBiS Browse Categories with Chat Precomputation Details.

```
Output Data:
Node: ServletPrinter:printCategory String int:19112841
   Data:
    <a href="SearchItemsByCategory?category=1&
    categoryName=Antiques">Antiques</a><br>
   Data:
    <a href="SearchItemsByCategory?category=2&
    categoryName=Books">Books</a><br>
   Data:
    <a href="SearchItemsByCategory?category=3&
    categoryName=Business">Business</a><br>
```

4.4.2.2 jGossip

4.4.2.2.1 Experimental Setup For this test the main page of the jGossip application is loaded while performing taint tracking. This page presents the user with a list of forums to view, as well as some other information about forum activity. The main page will likely be viewed quite frequently, being the primary point from which the rest of the forum is accessed, and is thus a good candidate for optimization.

4.4.2.2.2 Experimental Goals As with RUBiS, we hoped to find output that could be precomputed rather than calculated. For the particular page of jGossip analyzed, we hoped to find large portions of it which could be precomputed. Though we knew there would be difficulties with the complex execution paths created by autogenerated JSP code, we hoped that these details could be ignored if the precomputable execution paths reached end-to-end from database to page output.

4.4.2.2.3 Analysis

Taint Flow Breakdown We skip a breakdown of the taint flow here due to its size and complexity. It is not possible to even view the graph all at once, much less make sense of it. It is much more helpful to simply run the analysis and use the results to understand the flow of data in the application, as well as how to improve it.

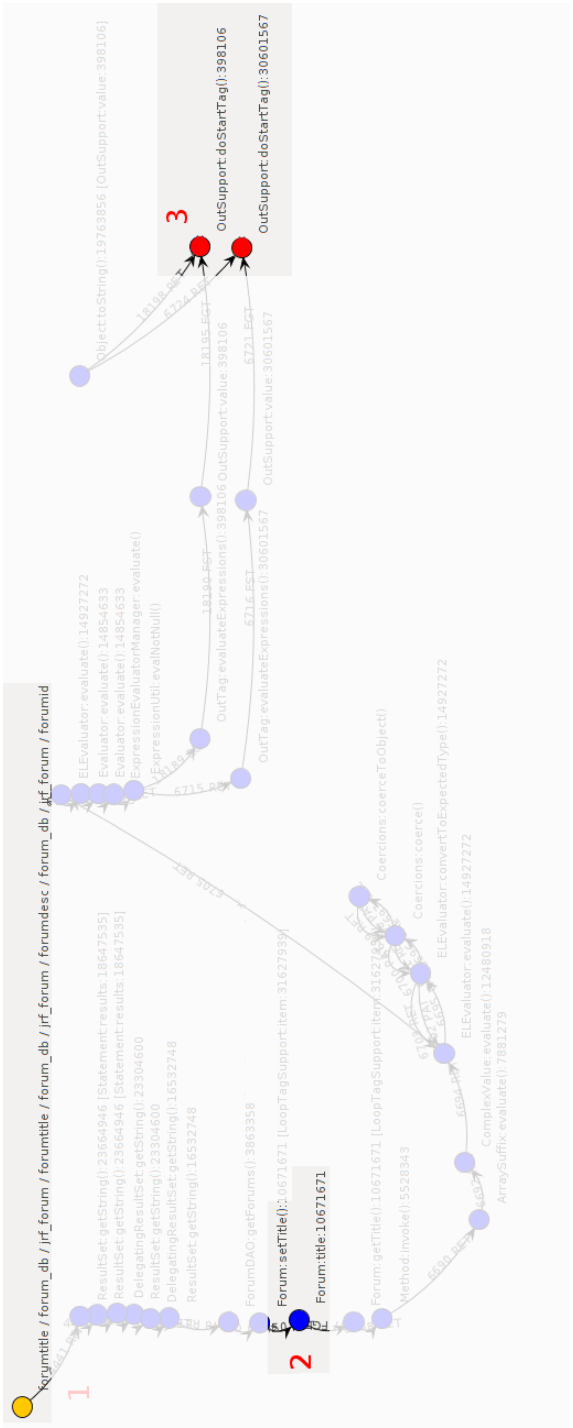

Figure 4.9: jGossip Main Page Precomputation Analysis Results.

Analysis Results Breakdown Running the precomputation analysis over the taint graph, providing the IDF given in Appendix B.3, produces the graph shown in Figure 4.9. 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 inputs with non-deterministic values. Breaking this flow down:

- The data is read from the database at point 1. Simply note that the input node starts with 'forumtitle', identifying the database column from which the data is read.
- The forum title data is eventually stored in the Forum:title field at point 2. Continuing along the flow, this field is subsequently accessed (by reflection, evidenced by the call to Method:invoke()) by a series of autogenerated methods (Java JSP applications convert HTML markup to Java code).
- From there to point 3, the data passes through a series of internal methods from JSP-related libraries.
- The forum title is ultimately used in the doStartTag() method at point 3, a method used by JSP tag objects to write data to the user.

There are similar graphs for other pieces of precomputable data on the main page. By comparing these with output summaries for this page similar to what is shown in Figure 4.2 for RUBiS, one can find large portions of the page which are precomputable.

4.4.2.2.4 Developer Interpretation and Response Having the dataflows for various elements of the main page being identified as precomputable all the way from database to user output means the following: these elements can be easily precomputed—one could even simply hardcode them in the JSP files to save on executing all the JSP/JSTL machinery, along with any user output which was identified as simply non-tainted. Furthermore, for the user output which was identified as being tainted by random sources, those flows are independent of those identified by the precomputation analysis, and so can be handled separately from them.

4.4.2.2.5 Summary The experimental goals were met in that we were able to identify large sections of a critical page which could be pregenerated. As jGossip is much more complex than RUBiS, more developer effort is needed to make use of the analysis results than with RUBiS. One would need to go through each identified precomputation result, decide if the saved computations are worth the modifications, and figure out how to modify the application given that much of its code is generated from JSP markup. One would need to take care not to interfere with execution responsible for generating the dynamic content on this page. In this case gains could be realized by simply removing the JSP tags which ultimately produce the precomputable output, and replacing them with precomputed data.

4.4.3 Caching

4.4.3.1 RUBiS

4.4.3.1.1 Experimental Setup For a data flow to be cacheable, it must depend on some predictable data but not on any non-deterministic data. By predictable data we mean data which we do not know the exact value for, but which can only take on a small enough range of values so as to be amenable to caching. To create such a scenario in RUBiS, we start with a page which lists various geographical regions. Such pages are common in RUBiS, and serve to help users limit the scope of their trading to a region they care about. From this page a region is selected from among those available, which requests the next page containing the list of categories from the previous precomputation examples. This list varies, however, in that it has been influenced by the user input provided by selecting a region.

4.4.3.1.2 Experimental Goals In this experiment the goal was to identify an obvious cache in the RUBiS application, with the analysis making the cache inputs and outputs clear. As RUBiS is a simple application, we hoped that the cache would be able to eliminate most of the computation responsible for generating the analyzed page.

4.4.3.1.3 Analysis

Taint Flow Breakdown The log file from the experiment request produces the taint flow graph shown in Figure 4.10. Remember that this graph spans 2 requests—one to display a list of geographical regions and another to display a list of categories. Also, note that this graph had StringBuilder

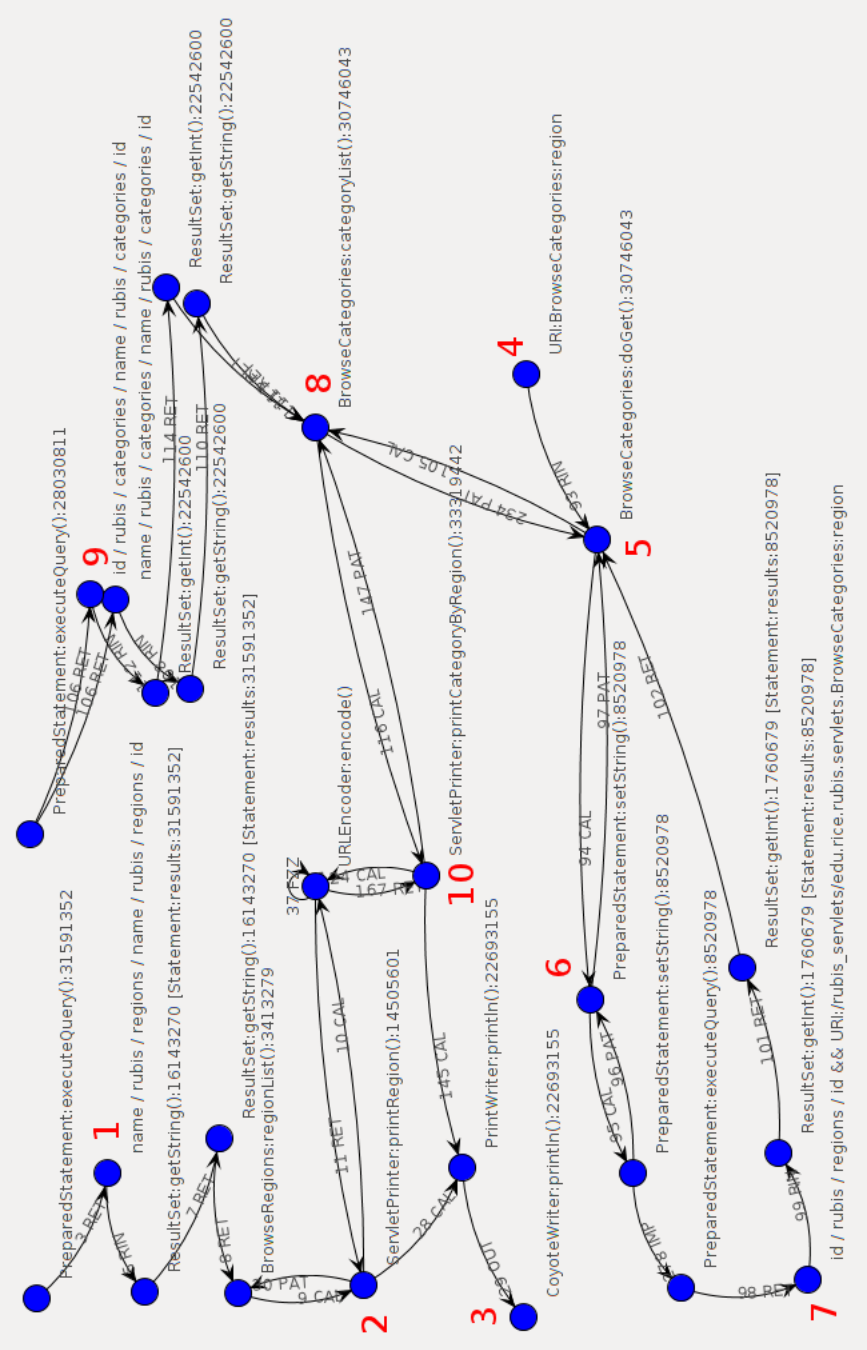

Figure 4.10: RUBiS Browse Categories by Region Trace.

method nodes removed to make it more presentable, as these are not necessary for grasping the important points of the analysis. Breaking this flow down:

- Point 1 shows the region name data being read from the database for the first request.
- At point 2 the region names are passed to an encode() which copies the data in a way which necessitates fuzzy propagation (see Section 3.2.1.3).
- The regions are output (along with any formatting text added to them) to the user at point 3, concluding the tracing for the first request.
- For the next request, a user clicks on a region, submitting its name as a request parameter. This is shown at point 4, as the 'region' parameter handled by the 'BrowseCategories' servlet.
- 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 source 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 (tainting from multiple sources is indicated with the '&&' separator).
- The query returns region IDs, which are returned to point 5 and passed into the categoryList() method at point 8.
- The categoryList() method also gets a list of category names and IDs from the database, which flow from point 9 (accessed in the same way as they are in the precomputation example).
- The category data and region IDs are sent to point 10, the 'printCategoryByRegion' method, which combines them with formatting text and passes them to the encode() method.
- The data is written out to the user, finishing the request.

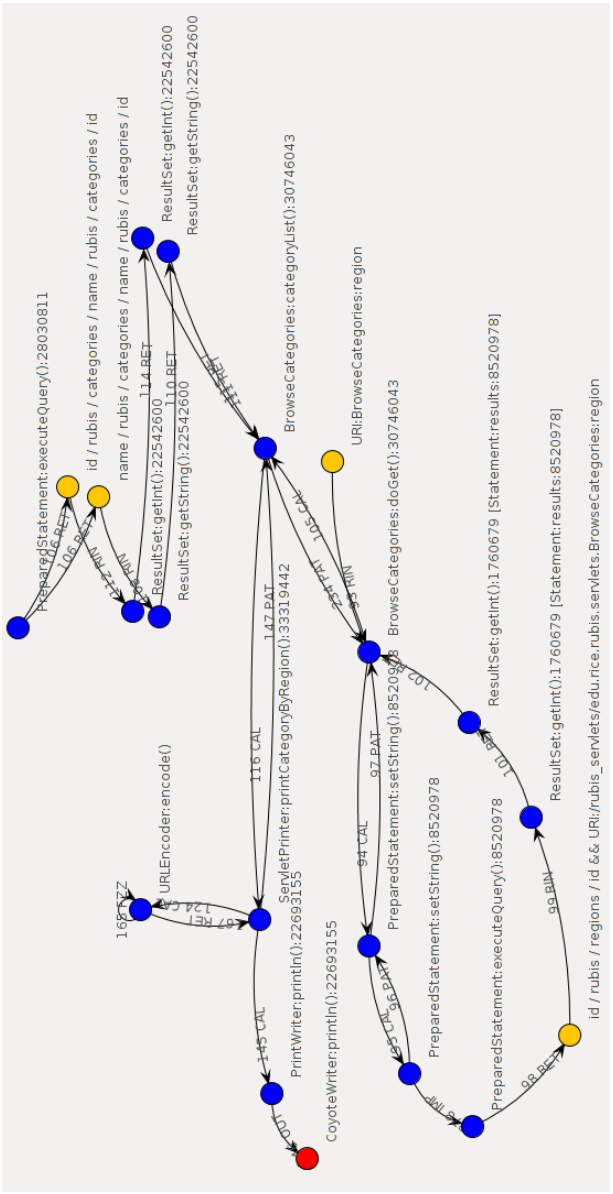

Figure 4.11: RUBiS Browse Categories by Region Caching Results.

Analysis Results Breakdown The caching analysis is run over this graph, provided the IDF given in Appendix B.2 and producing the result graph shown in Figure 4.11. First, note that this graph only has nodes from the second request—the first request could simply be precomputed and was ignored for the caching analysis. The graph shown was not eligible for precomputation due to the influence of the user-supplied region parameter, the value of which will vary. However, since this parameter has been marked as being predictable and the other inputs are stable (see Appendix A for predictable/stable definition), the analysis judges that a cache may be possible for this graph.

Figure 4.12: RUBiS Browse Region Categories Caching Details.

```
1: Output Data:
2: Node: CoyoteWriter:println String:22693155
3:    Data: <a href="BrowseCategories?region=AZ--Phoenix">
               AZ--Phoenix</a><br>
4:    Data: <a href="BrowseCategories?region=CA--Los+Angeles">
               CA--Los Angeles</a><br>
5:    Data: <a href="BrowseCategories?region=CA--Oakland">
               CA--Oakland</a><br>
6:    Data:
  <a href="SearchItemsByRegion?category=1&categoryName=Antiques&
   region=1">Antiques</a><br>
7:    Data:
  <a href="SearchItemsByRegion?category=2&categoryName=Books&
   region=1">Books</a><br>
8:    Data:
  <a href="SearchItemsByRegion?category=3&categoryName=Business&
   region=1">Business</a><br>
```

4.4.3.1.4 Developer Interpretation and Response Interpreting this, a developer can start at the red output node. Figure 4.12 shows the tainted data flowing here. Lines 3-5 are for the first request, and can be ignored. Lines 6-8 show the list of categories. Note how the link 'href' contains a region parameter with a region ID. Compare these lines to the user output breakdown shown in Figure 4.13. The entire response is composed of non-tainted (likely non-computed) data and the cacheable output. Looking at

Figure 4.13: RUBiS Browse Region Categories Tainted Output.

```
<title>RUBiS available categories</title> [NONTAINTED]
<p>Popular Items</p> [NONTAINTED]
<p>Chats</p> [NONTAINTED]
<form action="SubmitChat" method=POST> [NONTAINTED]
<input type=text size=20 name=chatMessage /> [NONTAINTED]
<input type=submit value="Post" /></form> [NONTAINTED]
<h2>Currently available categories</h2><br> [NONTAINTED]
<a href="SearchItemsByRegion?category=1&categoryName=Antiques&
 region=1">Antiques</a><br> [TAINTED]
<a href="SearchItemsByRegion?category=2&categoryName=Books&
 region=1">Books</a><br> [TAINTED]
<a href="SearchItemsByRegion?category=3&categoryName=Business&
 region=1">Business</a><br> [TAINTED]
</body> [NONTAINTED]
</html> [NONTAINTED]
```

the input (yellow) nodes in Figure 4.11, the source data responsible for the cacheable output comes from database tables which have all been marked as stable (see the IDF) (unlikely to change frequently), and the predictable region parameter.

Using the graph 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 needs to store and return results (the cache key) is the region parameter. The other inputs from the categories and regions tables only need 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) 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 can be picked up in the printCategoryByRegion node to be placed in the cache for later.

4.4.3.1.5 Summary This experiment was successful, as the analysis identified a critical page which could be entirely cached instead of generated for

each request. Furthermore, the analysis results made it obvious what the various inputs to the cache would be, and where data from the cache could be used. It is up to the developer how exactly to rearchitect the application, but the general guidelines are there.

4.4.3.2 jGossip

4.4.3.2.1 Experimental Setup The application was traced 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. We assumed that a selection from among the list of available forums could be considered predictable data, since the set of forums it not likely to change rapidly. Furthermore, this action is likely to occur frequently as users must choose their desired forums to interact with them.

4.4.3.2.2 Experimental Goals As jGossip is significantly more complex than RUBiS, we sought only to identify the existence of a possible cache. The details of the best place to implement the cache, and concerns over the amount of computation saved by the cache are left to future work. Here the concern is on identifying the inputs and outputs and ruling out the influence of non-deterministic data.

4.4.3.2.3 Analysis

Taint Flow Breakdown We skip a breakdown of the taint flow here due to its size and complexity. It is not possible to even view the graph all at once, much less make sense of it. It is much more helpful to simply run the analysis and use the results to understand the flow of data in the application, as well as how to improve it.

Analysis Results Breakdown The caching analysis for jGossip is significantly more difficult than with RUBiS 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 any manual analysis and which were still challenging to analyze automatically. Nevertheless, Figure 4.14 shows a graph obtained from the analysis tool which indicates the existence of a possible cache.

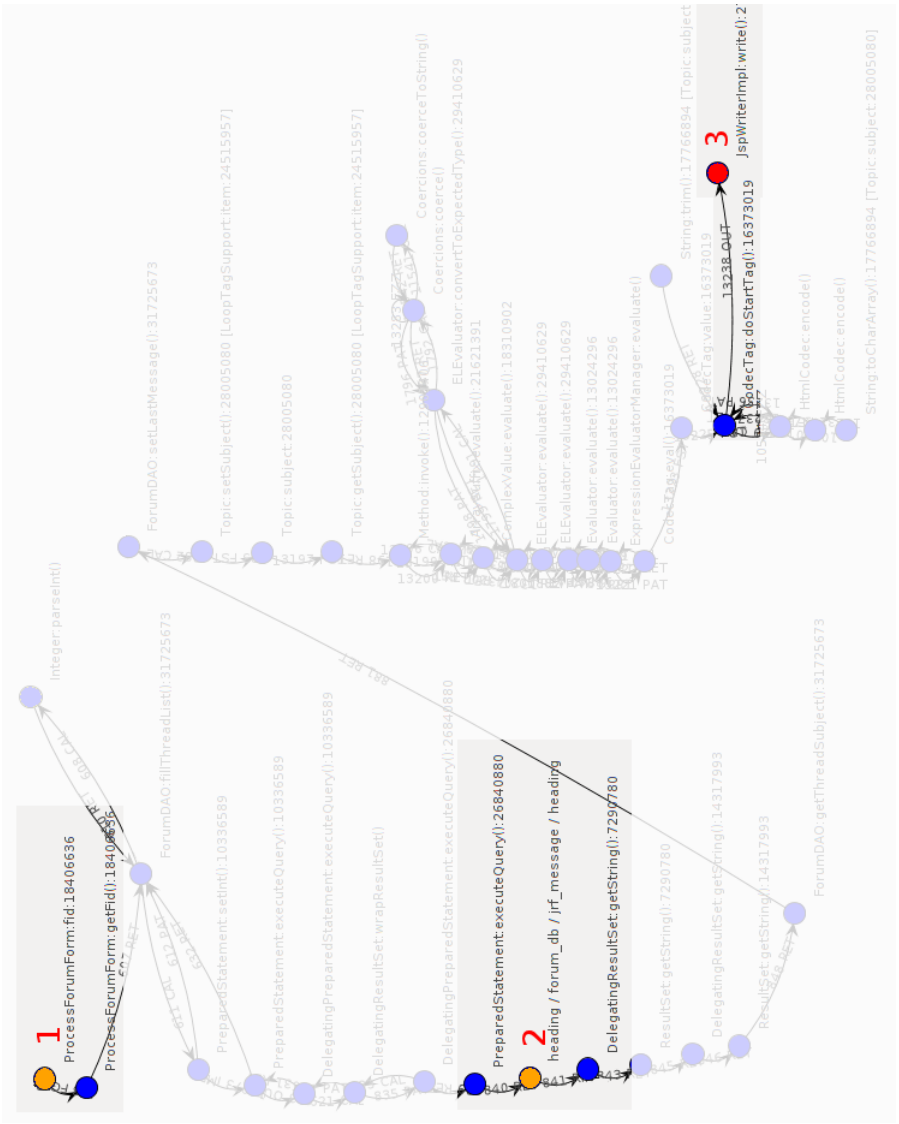

Figure 4.14: jGossip View Forums Caching Results.

The trace shows the following:

- A forum ID (fid) is supplied by the user when selecting a forum at point 1.
- The forum ID is used as the only parameter in a database query which fetches a message heading at point 2. These headings are among various descriptive data for a forum.
- The 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. Given a fid value, of which there are a limited set, and knowing whether or not the heading for a forum has been updated, we can either return a cached heading or generate one to fill the cache.

4.4.3.2.4 Developer Interpretation and Response Given this graph, it is up to the developer to investigate the feasibility of this cache. The only inputs to it, as shown by the analysis, are the fid and the forum heading. Assuming the headings are stable (and the IDF given in Appendix B.3 does specify this), the fid could be used as a cache key while the headings could be monitored for changes for cache invalidation. The developer would need to investigate the traces further, along with the application code, to determine a good place to implement such a cache. The fact that much of the logic is buried in JSP auto-generated code will make this more difficult than in the case of RUBiS.

4.4.3.2.5 Summary Our goals for this experiment were met, as a possible cache was identified for the analyzed page. The amount of data cached is small, and more work is needed to plan how to best implement the cache (non-trivial given the complexity of jGossip's architecture and use of generated code). Additionally, to obtain greater benefits from this analysis, one would need to continue searching for more caching opportunities beyond this proof of concept.

4.4.4 Postcomputation

4.4.4.1 RUBiS

4.4.4.1.1 Experimental Setup 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 (name, price, etc.), and submitting this to a RegisterItem servlet. The item is added to the database, and after that is done the item details are displayed back to the user when the request returns. This procedure importantly involves a write to a database, which is a likely target for postcomputation.

4.4.4.1.2 Experimental Goals The goal here was for the analysis to identify common computations that can be deferred. The output should make it clear at which points in the application logic the computations occur for the purpose of converting them to asynchronous versions.

4.4.4.1.3 Analysis

Taint Flow Breakdown The log file produces the graph given in Figure 4.15. Breaking this down:

- At point 1 the various request parameters describing the item are read.
- Some of these parameters are converted to numeric values at point 2. The IDF in Appendix B.2 specifies that these values should be tracked by the numeric tracking system (see Section 3.4.1).
- These values are passed along to point 3, where they are used to construct a PreparedStatement object which will be used to perform a database update.
- At point 4 the executeUpdate() method on the PreparedStatement is called, which causes the item data to be written to the database.
- Following this, some of the parameters are used at point 5 to query the database, checking if the item was successfully inserted.
- At point 6 the parameters are written back along with formatting text to the response page, showing the user the details of the item they submitted to sell.

Analysis Results Breakdown Figure 4.16 shows the results of running the postcomputation analysis over the taint flow graph. What this result graph shows are parts of the taint flow graph which could potentially be delayed until later without breaking the functioning of the application.

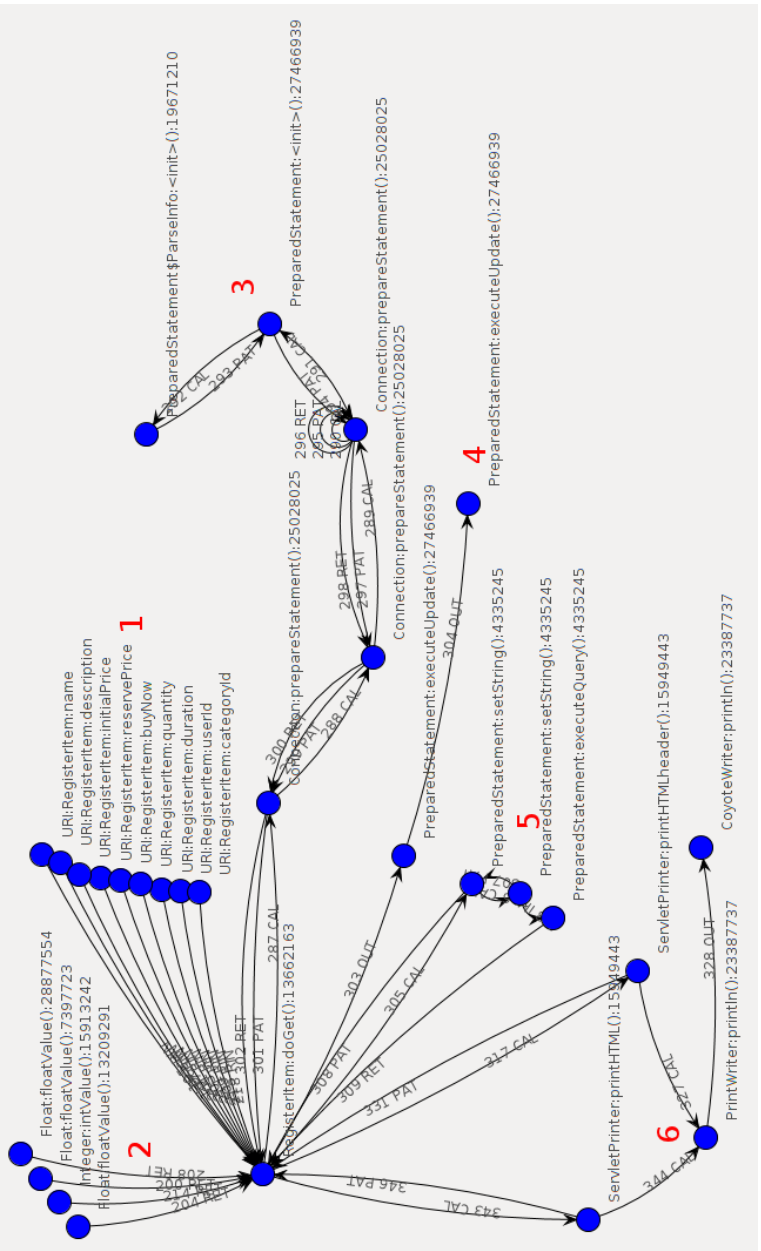

Figure 4.15: RUBiS Sell Item Trace.

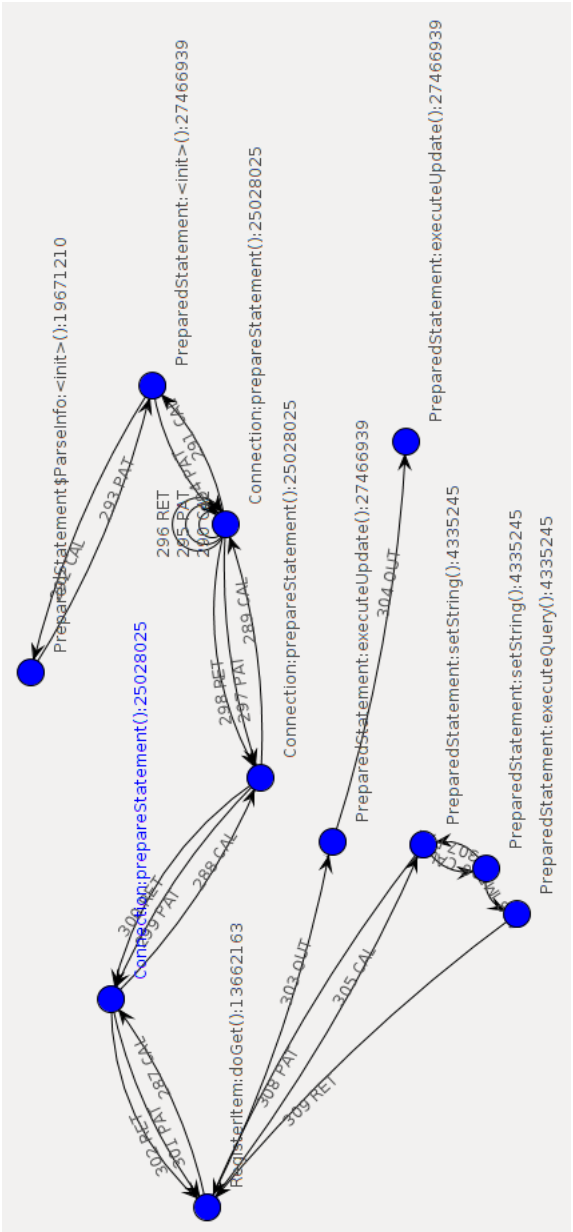

Figure 4.16: RUBiS Sell Item Postcomputation Analysis Results.

The figure shows that the database update and query flows at points 3, 4, and 5 from Figure 4.15 could be deferred. None of the queries return data which is used to generate the response page. Missing from the result graph is reading 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.

4.4.4.1.4 Developer Interpretation and Response It is now up to the developer to determine if the semantics of the application can tolerate deferring these computations. It is possible that it does not make sense to display the response page without knowing for certain whether or not the database update was successful. However, if this is acceptable, the analysis suggests that these queries could be done asynchronously, returning back to the user without waiting for their completion. The application could be altered to support this, and users would see response pages faster.

Additionally, such modifications tend to free up web servers which allocate threads to handle individual requests. The database update request can be queued in a DBMS, while the webserver is free to complete the request and free up a thread. There would likely need to be additional systems in place to deal with failed database updates in this case, such as by restarting them or asynchronously informing the user.

4.4.4.1.5 Summary Having identified a database update and subsequent update verification which could be deferred, a commonly performed request can potentially return back to the user more quickly. It is easy to see where these computations occur and thus where to begin deciding how to convert them to asynchronous versions—in the RegisterItem:doGet() method. A developer can make the appropriate modifications, and from there assess the correct and potentially improved operation of the application.

4.4.4.2 jGossip

4.4.4.2.1 Experimental Setup To test the postcomputation analysis for jGossip, we use a trace taken while submitting a delete forum request. This request sends a forum ID parameter to the server, which is used to locate the forum to delete. We anticipated that this action would involve computation and updates to the database which could potentially be deferred. It is true that this operation would likely not occur very often, but

the results are still useful as a minor optimization and as a proof of concept.

4.4.4.2.2 Experimental Goals The goal here was to find potential postcomputation opportunities. Due to the complexity of jGossip, the approach was to focus on analysis results for specific data suspected to be involved in deferrable computations. We hoped to find in the results evidence that such data underwent various computations before being written to a database, computations which did not influence output immediately visible to the user and which could be deferred.

4.4.4.2.3 Analysis

Taint Flow Breakdown We skip a breakdown of the taint flow here due to its size and complexity. It is not possible to even view the graph all at once, much less make sense of it. It is much more helpful to simply run the analysis and use the results to understand the flow of data in the application, as well as how to improve it.

Analysis Results Breakdown To obtain the result, the postcomputation analysis was run over the taint flow graph. The output was still quite complex, so we filtered the result graph down to locate nodes which communicated the forum ID used to specify the forum for deletion. Following this we looked for nodes which were from the actual jGossip code (not from framework classes), and found the section of the result presented in Figure 4.17. This graph shows a flow which could potentially be deferred to allow the user to receive a response faster. Breaking it down:

- At point 1 an object containing data from the request (user provided parameters like the forum ID) is passed into an Action object (Actions basically provide the entry point functionality in an Apache struts application, like the servlets do in RUBiS).
- Point 2 shows the accessing of the forum ID from the data object.
- At point 3 the ID is first passed from the deleteForum() method into a setInt() method which sets the ID as a parameter in a query.
- This query is then executed by the executeQuery() method call, which uses the forum ID to delete a forum from the database.

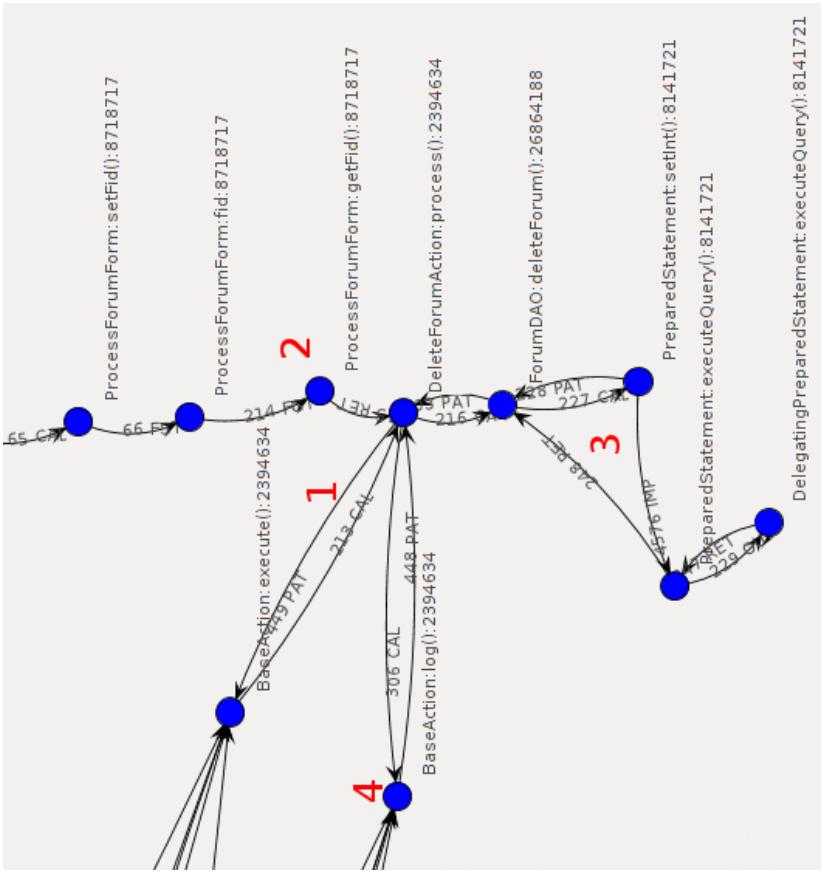

Figure 4.17: jGossip Delete Forum Postcomputation Analysis Results.

- Finally, at point 4 we see that not only can the flow which deletes the forum be deferred, but also a logging action.

The analysis is showing that the computation and flow of data presented in Figure 4.17 does not influence the production of output necessary to generate the user response page.

4.4.4.2.4 Developer Interpretation and Response At this point it is up to the developer to inspect this graph and determine the feasibility of deferring some of the computations it represents. At a high level, having received the forum ID from the user, one could modify the application to enqueue asynchronous computations to perform the forum deletion and logging. The request handler could then immediately return to the user, letting them know that their request had been received and was being processed.

As to the details of how exactly to implement this change, such would require a deeper investigation into the application's architecture. This will be made more challenging by the fact that jGossip uses a lot of generated code. Still, the analysis is useful in that it points out the possibility of this optimization, allowing one to recognize it and focus on how to realize it.

4.4.4.2.5 Summary A database update and logging action which could potentially be deferred were identified, as per the goals of this experiment. The computation occurs for an infrequently performed request, which reduces the usefulness of the optimization. However, having shown that the analysis can identify such cases acts as a proof of concept from which to search for further optimizations.

4.4.5 Persistent State

4.4.5.1 RUBiS

4.4.5.1.1 Experimental Setup The setup is the same as for the second request in Section 4.4.2.1.1.

4.4.5.1.2 Experimental Goals The goal of this experiment was to locate known persistent state added to the RUBiS application, and determine which parts of the application interacted with the state.

4.4.5.1.3 Analysis

Taint Flow Breakdown The taint flow for this setup has already been discussed in Section 4.4.2.1.3.

Analysis Results Breakdown The analysis searches the graph for any data which can be found in multiple requests. This is assumed to be persistent data, used for keeping state which exists beyond the scope of a single request. Figure 4.18 shows the location of persistent state when interacting with the RUBiS messaging system. 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 are those 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 encompass the writing of the chat messages out to the user.

4.4.5.1.4 Developer Interpretation and Response Knowing where persistent data is kept is not in itself an optimization. The knowledge of it is useful in general. Earlier we mentioned cases where an application may be in an environment where persistent state is not allowed or is unsafe. If such were the case, these analysis results could be used to target such state for removal, either by changing the logic of the application completely or moving the state into an acceptable store. The 'chattyList' data could simply be put into a database table.

Another use for this analysis would be to gauge how much of the logic relies on this kind of stateful data when performing replication. When deploying the RUBiS site across multiple servers to handle greater load, this analysis shows us what execution relies on data which must be shared and kept consistent across the servers. The 'chattyList' data could be a bottleneck for the computation encompassed by the orange nodes.

4.4.5.1.5 Summary The analysis successfully identified the in-memory persistent state in the application and clearly indicated which parts of the application made use of the data. With this information, one could perform a variety of actions depending on one's needs. One could modify the application to store the state in a database instead of in memory. One could also consider how the application interacts with the persistent state to assess the viability of replicating computations which depend on the state.

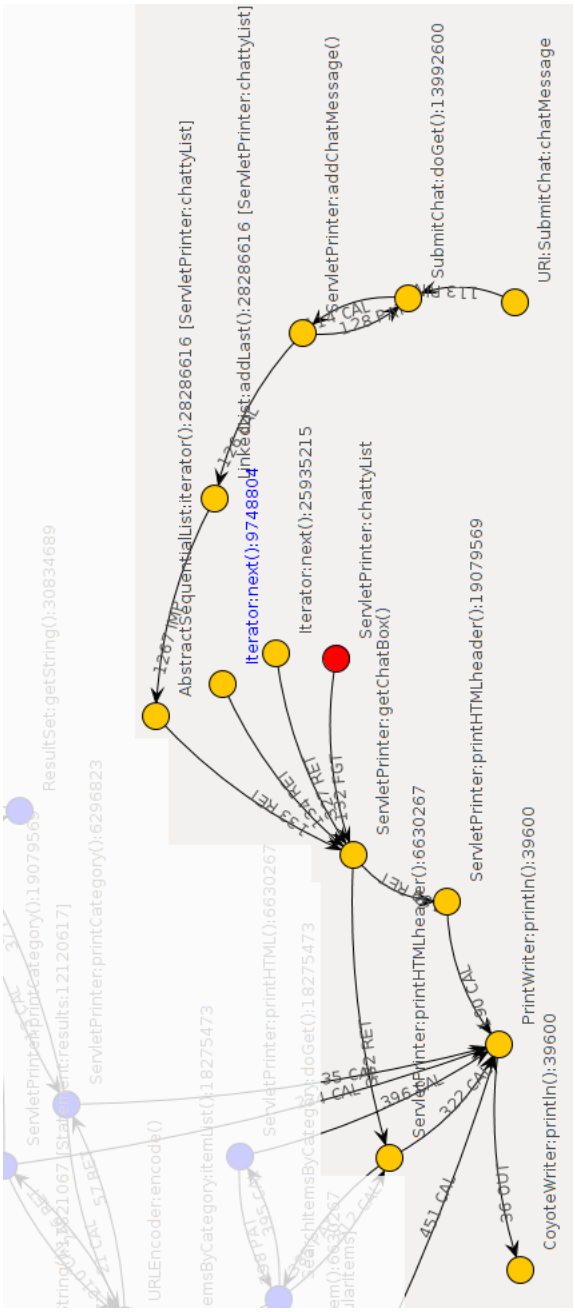

Figure 4.18: RUBiS Browse Items State Analysis Results.

4.4.5.2 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 is explored sufficiently in the RUBiS example. Furthermore, the next analysis, user state, subsumes this one and deals with the jGossip application.

4.4.6 User State

4.4.6.1 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.

4.4.6.2 jGossip

4.4.6.2.1 Experimental Setup 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 for both users, as we anticipated that some of this data would only be communicated with a single user.

4.4.6.2.2 Experimental Goals The goal of this experiment was to locate unknown persistent state in the jGossip application which was only shared with a single user, and determine which parts of the application interacted with the state.

4.4.6.2.3 Analysis

Taint Flow Breakdown We skip a breakdown of the taint flow here due to its size and complexity. Again, it is not possible to even view the graph all at once, much less make sense of it. It is much more helpful to simply run the analysis and use the results to understand the flow of data in the application, as well as how to improve it.

Analysis Results Breakdown We ran the automated user state analysis over the taint flow graph, which identified the flow shown in Figure 4.19.

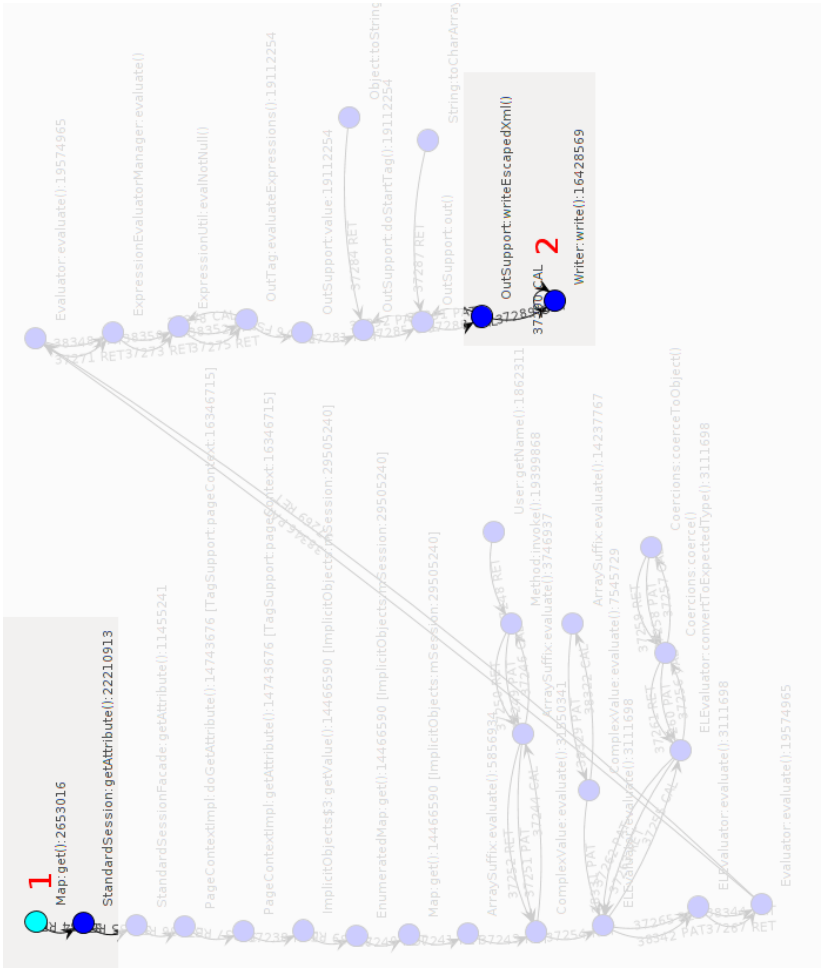

Figure 4.19: jGossip Login User State Analysis Results.

Though it is not shown in the figure, the analysis tool tells us that the tainted data in this flow is a username used to populate a username field which appears on every page of the forum.

The fact that the analysis identified this flow indicates that this username data has been used over multiple requests, but that each instance of this data is only ever shared with a single user (different users will understandably have their own unique usernames). Breaking down the figure:

- At point 1 we can see where the data is stored, in a Java Map object. The Java Map is the underlying store for the system which stores user session data in this application, and following the data from this node we see calls to the getAttribute() method which is an API call used to get session data.
- The data is passed through a series of internal, auto-generated JSP/JSTL code and is eventually written out to the user at point 2.

4.4.6.2.4 Developer Interpretation and Response 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 JSP/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).

4.4.6.2.5 Summary Satisfying our experimental goals, the analysis identified data present in multiple requests which was only shared with a single user. Additionally, it showed the sequence of components responsible for communicating the data from where it was stored to the user. This information could potentially be used to move state stored on the server into the client, making the application more resistant to failures on the server.

4.4.7 Wasteful Communication

4.4.7.1 RUBiS

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

4.4.7.2 jGossip

4.4.7.2.1 Experimental Setup The setup is the same as in Section 4.4.3.2.1.

4.4.7.2.2 Experimental Goals For this experiment, we hoped to find widespread occurrences of data being exchanged between functions and subsequently never accessed.

4.4.7.2.3 Analysis

Taint Flow Breakdown Understanding the taint flow is not necessary for the goals of this experiment.

Analysis Results Breakdown This analysis was a very simple one, and serves to hint at the potential for this technique. The wasteful communication analysis was run over the taint flow graph obtained for the jGossip caching analysis, removing any edges which carried tainted data which was subsequently never accessed. Figure 4.20 shows a before and after view. On the top is shown the unprocessed trace. On the bottom is the graph with the removed edges. The figure is merely a visual representation of the fact that only 5921 of the original 14693 edges remained. This does not mean that these removed edges carried no useful data, only that some of the data they carried was never used. 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 occurrence can be quite prevalent in certain applications, and that our analysis tool is capable of detecting it.

4.4.7.2.4 Developer Interpretation and Response At this point we envision this analysis being mainly useful as an input to more specific analysis tools. In particular, knowing whether or not data is used after it is communicated would be useful in getting better communication cost estimates for application partitioning algorithms. If an application is partitioned, sharing data between partitions incurs an additional cost, and thus only communicating data when it is actually needed (used) will provide savings. Wasteful communication should be avoided in a partitioning scenario, and so the algorithms which attempt to discover optimal partitionings of an application based on its dataflow should be aware of which dataflows are used and which aren't.

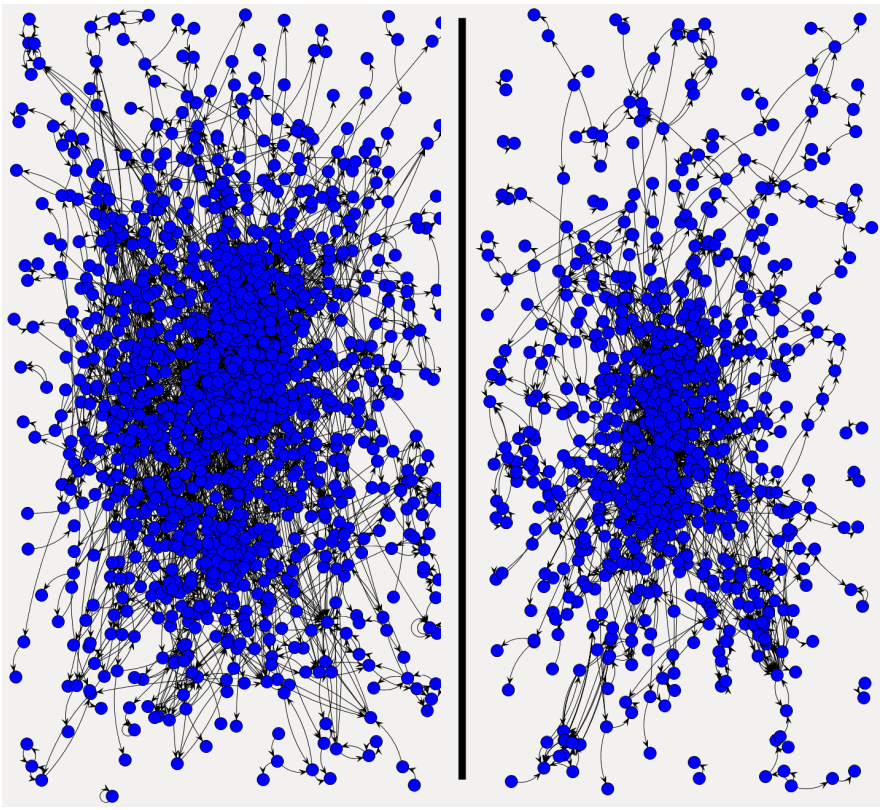

Figure 4.20: jGossip Wasteful Communication Analysis Results.

It is also possible that this analysis could be used to help a developer locate wasteful communication for the purpose of modifying the application to remove it, saving resources. This would be particularly useful if the communication was occurring across some expensive link, such as a network.

4.4.7.2.5 Summary The analysis identified almost 60% of edges as carrying unnecessary data in a trace of a commonly accessed page. Mainly, these results can be used to obtain better communication cost bounds for such tools as application partitioning algorithms. One might also use the results to identify application modifications for reducing wasteful inter-component communication.

Chapter 5

Conclusions

5.1 Discussion of Results

Overall we have been able to prove our hypothesis that DIFT can be used to help developers understand and automatically optimize web applications.

The results for the precomputation analysis on both applications were particularly successful. In both cases the analysis was able to proceed with minimal user intervention to produce the graphs in the evaluation. All that was needed was the initial input source description 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 cacheable output, but also clearly shows 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 to easily use this in implementing a cache for the data in question. It is useful to see the paths 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 mean that the analysis results are easy for one to grasp immediately.

Another analysis which worked particularly well was the user state analysis for jGossip. Looking for dataflow paths present in multiple requests but not shared with more than one user produced clean graphs like the one shown Section 4.4.6. It was easy to identify the location of the persistent data, and subsequently the sequence of communications and computations needed to take the data to its destination.

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 cacheable. How-

ever, they were too large to immediately grasp due to large groups of nodes from calls into frameworks. The graph shown in Section 4.4.3.2 was obtained by inspecting some of the jGossip source to understand how the cacheable data was likely flowing, and then searching in the analysis results for the occurrence 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 postcomputation 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 this analysis there were many nodes in jGossip which carried tainted data that never made it to the user, and as such were identified by the analysis heuristics. Since we wanted to focus on just the nodes which contributed to a deferrable database write, some filtering was needed to present the graph.

Such difficulties are examples of the challenges present in automating these kinds of analyses. Our results with the RUBiS application were very optimistic—the graphs produced by the analysis tool were very easy to understand mainly because of their small size. Additionally, since RUBiS uses no added libraries, every node in the result graphs is potentially relevant to the developer.

For a framework-heavy application like jGossip the graphs are much larger. At their current stage, most of our analysis results are negatively affected by this complexity. They suffer from a 'rats nest' effect, and require some manual filtering and searching to find the parts which could be used to actually support a developer in optimizing the application. In the future the analysis tool could be improved with IDE integration to link results more closely with source code. Such could allow a developer to focus on analysis flows which relate to specific parts of their code, making it easier for developers to navigate through the analysis results.

A related problem, again most prevalent in the large jGossip graphs, is that of false positives. The analyses were written to be fairly general in order to find their results in applications which were not written to be easily analyzed. The permissive nature of these analyses may produce a large number of results that need to be pared down to discover which of them are truly useful. In the caching analysis for jGossip there were results where it was possible for some flows to be affected by random data through control

dependence. This could make the results poor candidates for caching, but since our taint tracking technique does not consider control-flow propagation of taint, these results were allowed.

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 and 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 (tens 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 analyses 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 IDFs. As the analyses found valid results, it shows that given optimization opportunities it is possible for a taint tracking based automated analysis to target them. It also demonstrates 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 ad hoc from rough descriptions in the 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 version of our tools.

Finally, one of the side effects of this work was the identification of uses for taint tracing data that we did not initially anticipate. The wasteful communication analysis was discovered by accident when seeking a way to reduce the complexity of graphs for other analyses. Having studied application par-

titioning systems in the early stages of the project, we later recognized the potential of knowing whether communicated data was actually used. This was particularly exciting, as the data very easily supported an analysis that we had not originally intended.

5.2 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 made to the taint tracking and analysis tools to new research directions.

First of all, 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.

Moving away from AspectJ for the taint tracker could also allow a more complete tracking. With 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 measure. 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 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 proliferation 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 related to automation. If the tool was to be truly used to support developers' 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 targeted by our system were not developed in restricted programming models which make automatic changes easier. Following the 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 scenarios 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.

5.3 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 taint tracking data could be used to support a wide variety of analyses. Furthermore, 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 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 which should be taken further. By obtaining comprehensive taint tracking, analyses without user intervention, and automatic optimization of applications, a powerful resource for developers facing migration scenarios can be realized.

Bibliography

[1] Mohammed I. Al-Saleh and Jedidiah R. Crandall. On information flow for intrusion detection: what if accurate full-system dynamic information flow tracking was possible? In *Proceedings of the 2010 workshop on New security paradigms*, NSPW '10, pages 17–32, New York, NY, USA, 2010. ACM.

[2] Alexandru Caracas, Andreas Kind, Dieter Gantenbein, Stefan Fussenegger, and Dimitrios Dechouniotis. Mining semantic relations using netflow. In *BDIM*, pages 110–111, 2008.

[3] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 294 –303 vol.1, mar 1999.

[4] James R. Challenger, Paul Dantzig, Arun Iyengar, Mark S. Squillante, and Li Zhang. Efficiently serving dynamic data at highly accessed web sites. *IEEE/ACM Trans. Netw.*, 12(2):233–246, April 2004.

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

[6] Jim Challenger, Arun Iyengar, Paul Dantzig, Daniel M. Dias, and Nathaniel Mills. Engineering highly accessed web sites for performance. In *Web Engineering, Software Engineering and Web Application Development*, pages 247–265, London, UK, UK, 2001. Springer-Verlag.

[7] Erika Chin and David Wagner. Efficient character-level taint tracking for java. In *Proceedings of the 2009 ACM workshop on Secure web services*, SWS '09, pages 3–12, New York, NY, USA, 2009. ACM.

[8] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th conference on USENIX Security Symposium* -

Volume 13, SSYM'04, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[9] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[10] Byung-Gon Chun and Petros Maniatis. Dynamically partitioning applications between weak devices and clouds. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, MCS '10, pages 7:1–7:5, New York, NY, USA, 2010. ACM.

[11] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. *SIGARCH Comput. Archit. News*, 35(2):482–493, June 2007.

[12] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Tainting is not pointless. *SIGOPS Oper. Syst. Rev.*, 44(2):88–92, April 2010.

[13] Louis Degenaro, Arun Iyengar, Ilya Lipkind, and Isabelle Rouvellou. A middleware system which intelligently caches query results. In *IFIP/ACM International Conference on Distributed systems platforms*, Middleware '00, pages 24–44, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[14] Xiaohui Gu, Klara Nahrstedt, Alan Messer, Ira Greenberg, and Dejan Milojicic. Adaptive offloading inference for delivering applications in pervasive computing environments. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, PERCOM '03, pages 107–, Washington, DC, USA, 2003. IEEE Computer Society.

[15] Rajiv Gupta, Neelam Gupta, Xiangyu Zhang, Dennis Jeffrey, Vijay Nagarajan, Sriraman Tallam, and Chen Tian. Scalable dynamic information flow tracking and its applications. In *IPDPS*, pages 1–5, 2008.

[16] Mohammad Hajjat, Xin Sun, Yu-Wei Eric Sung, David Maltz, Sanjay Rao, Kunwadee Sripanidkulchai, and Mohit Tawarmalani. Cloudward bound: planning for beneficial migration of enterprise applications to the cloud. *SIGCOMM Comput. Commun. Rev.*, 40(4):243–254, August 2010.

[17] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, ACSAC '05, pages 303–311, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. libdft: practical dynamic data flow tracking for commodity systems. *SIGPLAN Not.*, 47(7):121–132, March 2012.

[19] Emre Kiciman, Benjamin Livshits, Madanlal Musuvathi, and Kevin C. Webb. Fluxo: a system for internet service programming by non-expert developers. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 107–118, New York, NY, USA, 2010. ACM.

[20] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[21] Geoffrey Lefebvre, Brendan Cully, Christopher Head, Mark Spear, Norm Hutchinson, Mike Feeley, and Andrew Warfield. Execution mining. *SIGPLAN Not.*, 47(7):145–158, March 2012.

[22] Shashidhar Mysore, Bita Mazloom, Banit Agrawal, and Timothy Sherwood. Understanding and visualizing full systems with data flow tomography. *SIGARCH Comput. Archit. News*, 36(1):211–221, March 2008.

[23] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

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

[25] Alexander Rasmussen, Emre Kiciman, Benjamin Livshits, and Madanlal Musuvathi. Improving the responsiveness of internet services with automatic cache placement. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 27–32, New York, NY, USA, 2009. ACM.

[26] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *SIGPLAN Not.*, 39(11):85–96, October 2004.

[27] Angeliki Zavou, Georgios Portokalidis, and Angelos D. Keromytis. Taint-exchange: a generic system for cross-process and cross-host taint tracking. In *Proceedings of the 6th International conference on Advances in information and computer security*, IWSEC'11, pages 113–128, Berlin, Heidelberg, 2011. Springer-Verlag.

Appendix A

Terms Used

- Predictable: This describes data with deterministic values over some range. A predictable data source will return an unknown value, but it will be from a small enough set of values that the result could be usefully cached.
- Stable: This describes data with strongly deterministic values. Generally, a stable data source should return the same known value for extended periods.
- 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, referred 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.

Appendix B

Input Description Files

B.1 Input Source Description File Format

```
<root>
  <databasesourceinfo catalog="..."
            table="..."
            column="..."
            variability="STABLE|PREDICTABLE|RANDOM"
            numtracking="true|false" />
  <requestsourceinfo   uri="..."
            parameter="..."
            variability="STABLE|PREDICTABLE|RANDOM"
            numtracking="true|false" />
</root>
```

As is shown here, the input source description file has two types of tags. databasesourceinfo tags describe inputs from database reads, and requestsourceinfo tags describe inputs from user web request parameters. For database inputs, one can specify the catalog (database name), table, and column names of data. For request parameters, one specifies the uri of the servlet which received the user data, as well as the name of the parameter associated with it.

For both tags, one also specifies the variability as being either STABLE, PREDICTABLE, or RANDOM. These are used by the caching and precomputation analyses. Basically:

- STABLE sources have values which are not expected to change. When the application reads data from these, generally the same values should always be expected.
- PREDICTABLE sources have values over a reasonably predictable range. While one does not always know what a read from one of

these sources will produce, one does have some confidence over the set of possible values.

- RANDOM sources have non-deterministic values. They could produce a different value every time.

Finally, one can optionally, for columns and parameters producing numeric values, indicate that the numeric values should be tracked by the taint tracker, using the numtracking flag. These values should not influence control statements in the program, due to how our system tracks numeric values, so numtracking is best suited for things like identifiers and values which are not predicated on in the application. If not specified, this defaults to false.

B.2 RUBiS Input Source Description File

```
The contents are specified in a non-XML, condensed format:
<databasesourceinfo ... />:
Database: catalog/table/column - variability: ...
  - numtracking: ...
<requestsourceinfo ... />:
Request: uri:parameter - variability: ...
  - numtracking: ...

Database: rubis/categories/id - variability: STABLE -
  numtracking: true
Database: rubis/categories/name - variability: STABLE
Database: rubis/regions/id - variability: STABLE
  - numtracking: true
Database: rubis/regions/name - variability: STABLE
Database: rubis/users/id - variability: STABLE
  - numtracking: true
Request: SubmitChat:chatMessage - variability: RANDOM
Request: BrowseCategories:region - variability: PREDICTABLE
Request: BrowseCategories:nickname - variability: PREDICTABLE
Request: BrowseCategories:password - variability: PREDICTABLE
Request: RegisterItem:initialPrice - variability: RANDOM
  - numtracking: true
Request: RegisterItem:reservePrice - variability: RANDOM
  - numtracking: true
Request: RegisterItem:buyNow - variability: RANDOM
```

```
  - numtracking: true
Request: RegisterItem:quantity - variability: RANDOM
  - numtracking: true
Request: RegisterItem:userId - variability: PREDICTABLE
  - numtracking: true
Request: RegisterItem:categoryId - variability: PREDICTABLE
  - numtracking: true
```

B.3 jGossip Input Source Description File

```
The contents are specified in a non-XML, condensed format:
<databasesourceinfo ... />:
Database: catalog/table/column - variability: ...
  - numtracking: ...
<requestsourceinfo ... />:
Request: uri:parameter - variability: ...
  - numtracking: ...

Some column names have been modified here for clarity

Database: forum_db/jrf_forum/forumtitle - variability: STABLE
Database: forum_db/jrf_forum/forumdesc - variability: STABLE
Database: forum_db/jrf_forum/forumid - variability: STABLE
  - numtracking: true
Database: forum_db/jrf_forum/locked - variability: STABLE
Database: forum_db/jrf_group/groupname - variability: STABLE
Database: forum_db/jrf_group/groupid - variability: STABLE
  - numtracking: true
Database: forum_db/jrf_group/groupsort - variability: STABLE
Database: forum_db/jrf_message/content - variability: RANDOM
Database: forum_db/jrf_message/from - variability: RANDOM
Database: forum_db/jrf_message/heading - variability: STABLE
Database: forum_db/jrf_message/id - variability: RANDOM
  - numtracking: true
Database: forum_db/jrf_thread/timestamp
  - variability: PREDICTABLE
Database: forum_db/jrf_thread/tid - variability: PREDICTABLE
  - numtracking: true
Database: forum_db/jrf_thread/sortby - variability: PREDICTABLE
```

```
Database: forum_db/jrf_whois/id - variability: PREDICTABLE
  - numtracking: true
Database: forum_db/jrf_whois/ip - variability: RANDOM
Database: forum_db/jrf_whois/sessionid - variability: RANDOM
Database: forum_db/jrf_whois/username - variability: PREDICTABLE
Database: forum_db/jrf_skinparams/paramname - variability: STABLE
Database: forum_db/jrf_skinparams/paramvalue
  - variability: STABLE
Database: forum_db/jrf_constants/name - variability: STABLE
Database: forum_db/jrf_constants/value - variability: STABLE
Request: DeleteForum:fid - variability: PREDICTABLE
  - numtracking: true
Request: ShowForum:fid - variability: PREDICTABLE
  - numtracking: true
```